Ю.И.Рыжиков

ПРОГРАММИРОВАНИЕ НА ФОРТРАНЕ POWERSTATION ДЛЯ ИНЖЕНЕРОВ. ПРАКТИЧЕСКОЕ РУКОВОДСТВО

Предлагаемая вашему вниманию книга является практическим руководством но новейшей версии Фортрана-90 для ПК, включая технологию разработки программ, работу с библиотеками и построителем графиков. Ряд примеров сложных программ имеет непосредственную практическую ценность. Все программы, приведенные в книге, автор широко использует на практике.

Автор — доктор технических наук. заслуженный деятель науки РФ, профессор Юрий Иванович Рыжиков, основавший кафедру математического обеспечения ЭВМ Военного инженерно-космического университета им. А. Ф. Можайского, которая дала компьютерному миру таких известных авторов, как Б. С. Богумирский и А. Д. Хомоненко. Преподаватель программирования, информатики, вычислительной математики, моделирования вычислительных систем, автор 170 научных работ, Ю. И. Рыжиков адресует свой труд студентам, инженерам, научным работникам, интересы которых лежат в области программирования численных задач.

Оглавление

Введение 3

1. Элементы и объекты программы 7

1.1. Набор символов 7

1.2. Формат программы 7

1.3. Понятие о проекте 8

1.4. Текстовый редактор 11

1.4.1. Работа с файлами 11

1.4.2. "Горячие" клавиши 11

1.4.3. "Заказная" распечатка 13

1.4.4. -Управление окнами просмотра 13

1.5. "Хороший стиль" программирования 13

1.6. Лексемы 14

1.7. Описания 15

1.8. Выражения 17

1.8.1. Арифметические выражения 17

1.8.2. Встроенные функции 18

1.8.3. Логические выражения 19

1.9. Инициализация и изменение значений переменных 21

1.10. Простейший ввод-вывод 22

2. Операторы управления 23

2.1. Разветвления 24

2.2. Циклы 26

2.2.1. Цикл с шагом 26

2.2.2. Цикл с условием 27

2.2.3. Вложенные циклы 29
2.2.4. Дополнительные средства 30
3. Массивы 32
3.1. Роль массивов 32
3.2. Описание массивов 32
3.3. Вырезки и сечения массивов 33
3.4. Задание массивов 34
3.5. Поэлементные операции 35
3.6. Выборочные действия 35
3.7. Встроенные функции для векторов и матриц 36
3.7.1. Справочные функции 36
3.7.2. Преобразование массивов 37
3.7.3. Неэлементные операции 38
3.8. Динамическая память 39
4. Обработка строк 40
4.1. Строки и массивы строк 40
4.2. Функции от строк 40
5. Программные компоненты 42
5.1. Программа и ее компоненты 42
5.2. Подпрограммы 44
5.3. Функции 45
5.4. Расположение операторов 46
5.5. Области видимости меток и имей 46
5.6. Внутренние процедуры 47
5.7. Интерфейс процедур 47
5.8. Специальные виды параметров 49
5.8.1. Массивы как параметры 49
5.8.2. Процедуры как параметры 52
5.8.3. Необязательные и ключевые параметры 53
5.9. Рекурсивные процедуры 54
5.10. Общие области и подпрограммы данных 56
5.11. Модули и работа с ними 58
5.11.1. Пример "модульной" программы 59
5.11.2. Специальные виды модулей 61
5.12. Доступ к объектам модуля 63
6. Ввод-вывод данных 64
6.1. Список ввода-вывода 64
6.1.1. Ввод, управляемый списком 65
6.1.2. Группа NAMELIST 65
6.2. Спецификации формата 65
6.2.1. Управление размещением информации 67
6.2.2. Переменные спецификации формата 67

6.3. Взаимодействие списков объектов и форматов 68
6.4. Внешние файлы 69
6.5. Внутренние файлы 70
7. Производные типы данных 72
7.1. Производные типы данных 72
7.2. Операции над определяемыми типами 73
8. Ссылки и списки 76
8.1. Базовые понятия 76
8.2. Ссылки как псевдонимы 77
8.3. Связные списки 77
8.4. Целые указатели 80
9. Вычислительные методы 82
9.1. Построение частотных характеристик 82
9.2. Рекуррентные вычисления 84
9.3. Генерация случайных чисел 84
9.4. Сортировка и поиск 86
9.4.1. Поиск в массиве 86
9.4.2. Понятие о сортировках 86
9.4.3. Линейный выбор с обменом 87
9.4.4. Пузырьковая сортировка 87
9.4.5. Челночная сортировка 88
9.4.6. Быстрая сортировка 89
9.4.7. О внешних сортировках 91
9.5. Работа с разреженными матрицами 91
9.6. Обращение матрицы методом Гаусса 97
9.7. Решение уравнения методом секущих 98
9.8. Вычисление определенных интегралов 99
9.9. Решение дифференциальных уравнений 102
9.10. Математические библиотеки IMSL 107
9.11. Numerical Recipes 110
10. Пакет научной графики 112
10.1. Знакомство с пакетом 112
10.2. Константы SciCraph 113
10.3. Структуры установок 114
10.4. Типы данных и осей 116
10.5. Вызывающая последовательность 117
10.6. Установка основных параметров 117
10.6.1. График в целом 117
10.6.2. Ряды данных 118
10.6.3. Координатные оси 119
10.7. Рисование графика 120
10.7.1. Основа графика 120

10.7.2. Собственно графики 120
10.8. Коды возврата. 121
10.9. Пример применения Scigraph 122
10.10. Опыт применения 127
11. Организация разработки программ 130
11.1. Состав и основные команды MDS 130
11.2. Определение директорий 130
11.3. Командная консоль 131
11.4. Организация проектов 132
11.4.1. Цель проекта 132
11.4.2. Рабочее поле 132
11.4.3. Структура проекта 133
11.4.4. Типы проектов 133
11.4.5. Задание целевых опций 133
11.5. Проекты и модули 133
11.6. Создание проектов 134
11.7. Компиляция, связывание, запуск 136
11.7.1. Метакоманды и опции компилятора 137
11.7.2. Связывание 140
11.7.3. Работа с модулями I'll
11.7.4. Создание выполняемой программы 142
11.7.5. Выполнение программы 142
11.8. Просмотр проекта 142
11.9. Отладчик 143
11.9.1. Отладка синтаксиса 143
11.9.2. Организация семантической отладки 144
11.10. Профилирование проекта 147
11.11. Работа с личными библиотеками 148
11.11.1. Вазовые понятия 148
11.11.2. Статическая библиотека 149
11.11.3. Динамические библиотеки 150
11.11.4. Библиотекарь 152
Задачи для упражнений 153
Литература 155

Введение

Бурное развитие вычислительной техники, в особенности персональных ЭВМ, обусловило их проникновение во все стороны общественной жизни (и почти все — личной). Волна программных продуктов "для непрограммирующих пользователей" — текстовые редакторы, электронные таблицы, тривиальные СУБД, системы подготовки презентаций — определила и господствующие тенденции в выпуске литературы. Однако рынок как этих продуктов, так и литературы по ним обнаруживает отчетливую тенденцию к насыщению, определяемому общим кризисом потребительски-распределительной надстройки над экономикой, крахом множества банков и предприятий спекулятивного бизнеса. Нельзя долго существовать, ничего не производя. Возрождение и развитие научно-технического и военного потенциала России неизбежно приведет к возрождению престижа инженерных профессий, росту потребности в технических и иных математических расчетах и соответствующих программных продуктах.

Появление пакетов математических программ породило у многих потенциальных пользователей их надежды избавиться, наконец, от изучения математики и программирования. Однако пакеты прикладных программ — даже такие мощные, как Mathematica 3 и Maple V — работают в режиме интерпретации и являются скорее инструментами поисковых исследований, чем трудоемкого счета с перебором вариантов. Поэтому знание программирования (хотя бы его элементарных основ) необходимо каждому, кто желает использовать ЭВМ при решении имеющей количественное содержание задачи. Для инженеров не по диплому, а по характеру работы освоение программирования (в дополнение к навыкам использования ПЭВМ как заменителя пишущей машинки и конторской книги) остается важнейшим элементом технической культуры и залогом успешной профессиональной деятельности. К тому же, программирование для ЭВМ является уникальным средством развития логического мышления.

Философская, воспитательная, общеобразовательная и, наконец, непосредственная практическая ценность программирования определяют необходимость прочного усвоения этой дисциплины студентами всех инженерных (и тем более математических) специальностей.

Алгоритмический язык Фортран — первый (1954–1957 гг.) и наиболее распространенный язык, ориентированный на программирование расчетных задач. Массовость применения Фортрана объяснялась наличием в нем средств, позволяющих более полно использовать аппаратные возможности ЭВМ, встроенной арифметики комплексных чисел, независимой компиляцией процедур и поддерживалась колоссальным объемом созданного на Фортране математического обеспечения. Его возможности неуклонно возрастали, что позволило ему успешно конкурировать с несколькими поколениями языков-соперников. К примеру, Американский национальный центр атмосферных исследований "является исключительно фортранным учреждением". В Ливерморской Национальной лаборатории "почти все вычисле-

ния, производимые на больших машинах, делаются на Фортране; это по-прежнему самый лучший язык для физиков" [5, с.24]. Циники заявляют, что через 30 лет язык научных вычислений будет очень сильно отличаться от того, что мы имеем сейчас, но по-прежнему будет называться Фортраном.

Начиная с 1978 г. велась работа над новой, отвечающей современным требованиям версией этого языка программирования. Главной задачей было способствовать мобильности, надежности, эффективности и удобству сопровождения языка. Его стандартизованная версия получила название Фортран 90 (ниже — Ф90). Важнейшие из новшеств — это обработка массивов с использованием кратких, но достаточно мощных обозначений и возможность введения определяемых пользователем типов данных и манипулирования ими. Кроме того в язык введены:

- Свободный формат записи операторов.
- Ссылки.
- Параметризация встроенных типов, дающая возможность процессорам поддерживать короткие целые и упакованные логические типы, очень большие наборы символов, а также более двух точностей для действительных и комплексных типов.
- Улучшенные средства для численных расчетов, в том числе ряд числовых справочных функций.
- Новый тип программной компоненты — модуль, пригодный для глобальных описаний данных и библиотек процедур и обеспечивающий надежный метод инкапсуляции производных типов данных.
- Управляющая конструкция `SELECT CASE` и новые формы оператора цикла.
- Внутренние и рекурсивные процедуры, возможность использования необязательных и ключевых параметров при вызове процедур.
- Динамическое выделение памяти.
- Множество новых встроенных процедур.

Отдельные элементы Ф90 реализовывались на машинах типа IBM PC, начиная с Microsoft Fortran 5.0, а в полном объеме — в Фортране PowerStation. Четвертая его версия, далее сокращенно именуемая FPS, вполне оправдывает это название. Она содержит не только систему программирования (язык с "фирменными" расширениями[1], транслятор, загрузчик, отладчик, библиотеки подпрограмм, справочные средства), но и объединенный в `Microsoft Development Studio` набор инструментов поддержки больших программных проектов (текстовый редактор, идентификация и разграничение версий, контроль связей программных единиц, профилировка трудоемкости и отлаженности, перенос на другие платформы и программные среды).

Фортран 77 целиком содержится в Ф90, но некоторые его средства отмечены в стандарте как устаревшие. Применять их не рекомендуется, и они, возможно, будут удалены из языка при запланированном на 2000 г. очередном пересмотре. Приведем список конструкций языка, рассматриваемых разработчиками как устаревшие, и их альтернативы:

[1] Последние в электронной документации выделены синим цветом.

EQUIVALENCE — для совместного использования памяти двумя объектами (заменяется ссылками и функцией transfer),

COMMON-блоки и BLOCK DATA — обмен информацией между программными единицами через общую область памяти (заменяется модулями),

ENTRY — описатели дополнительных точек входа в процедуры (компенсируется появлением модульных процедур),

DO WHILE — заменяем на "глухой цикл"

```
do
   if (<ЛВ>) exit
   ...............
end do
```

ФИКСИРОВАННЫЙ ФОРМАТ строк может использоваться, если размещать метки в позициях 1-5, а операторы в 7-72; использовать для обозначения комментария только ! (но не в 6-й позиции); при переносе строки ставить амперсанд в позиции 73 переносимой строки и в позиции 6 строки-продолжения.

ВЕЩЕСТВЕННЫЙ ТИП ДВОЙНОЙ ТОЧНОСТИ заменяется вещественным типом разновидности kind(0.d0).

ОПИСАТЕЛЬ ДЛИНЫ СТРОКИ character*<длина> заменяется на character([len=]<длина>).

АРИФМЕТИЧЕСКИЙ IF заменяется блочной конструкцией

if - then - else.

ВЫЧИСЛЯЕМЫЙ GO TO заменяется конструкцией case.

НАЗНАЧАЕМЫЙ GOTO заменяется вызовом внутренней процедуры.

ОПЕРАТОРНЫЕ ФУНКЦИИ потеряли актуальность в связи с появлением внутренних процедур.

Большой объем исходной информации о языке и системе программирования при обычных ограничениях на объем и время подготовки книги вынудили произвести жесткий отбор материала. В основу этого отбора и компоновки были положены следующие принципы.

1. Учебник — не гипертекстовый справочник. Материал отбирался по принципу наибольшей полезности для прикладного программиста (разумеется, субъективно), не претендует на полноту, группируется в удобной для освоения последовательности, излагается неформально и (по крайней мере в первых главах) доступен даже для начинающего программиста. Описание конструкций языка сопровождается методическими рекомендациями по применению и примерами различной степени сложности. Очевидные ограничения на употребление тех или иных конструкций (например, недопустимость subroutine в основном тексте головной программы) не оговариваются. Необходимые для полноты вынужденные забегания вперед и ссылки на позже определяемые понятия сведены к возможному минимуму.

Как дополнительный источник можно использовать книгу [13], включенную в электронную документацию к системе. Разумеется, ее содержание было учтено при написании данной книги.

2. Фортран был и остается языком преимущественно для разработки численных приложений. Поэтому средства обработки текстовой информации, создания разветвленных динамических структур данных, работы с файлами, графического вывода освещаются в минимальном объеме.

3. Фортран — самый старый язык программирования, и на нем написано колоссальное количество программ. Эти программы могут и должны быть использованы при работе на FPS, который тем самым обречен на длительное сосуществование с официально признанными "устаревшими" конструкциями. Опыт массового

применения FPS и давление пользователей вполне могут заставить авторов языка пересмотреть эти оценки. Поэтому наиболее удобным "древностям" пришлось найти место в тексте.

Отметим некоторые отклонения от канонической терминологии перевода [6]. В упомянутой книге делается попытка разграничить понятия "субпрограмма" и "подпрограмма". В русском языке она заведомо обречена на неудачу, и первый термин мы в соответствии с сорокалетней отечественной традицией заменяем на "процедуру", объединяющую понятия подпрограммы и функции. Пришлось также заменить "сечение" на заимствованное из Алгола-68 понятие "вырезки", более точно соответствующее объекту FPS. Термин "сечение" оставлен для частного случая, когда границы вырезки совпадают с границами массива в целом (как в PL/1).

В соответствии с неявным международным стандартом все конструкции языка и примеры программ набраны машинописным шрифтом. При пояснении синтаксиса родовые понятия, подлежащие в реальных программах замене на конкретику, заключены в угловые скобки, а в квадратных скобках указаны необязательные элементы. Поскольку в Фортран-программах заглавные и строчные буквы не различаются (кроме строковых констант), выбор регистра определялся эстетическими соображениями и необходимостью выделения в поясняющем тексте *отдельных* ключевых слов. Обозначение [F9] предполагает нажатие клавиши F9; соответственно [LM] и [RM] означают щелчок левой или правой кнопкой мыши.

Пунктуация в приводимых примерах подчинена грамматике Фортрана. В тексте рекомендуются упражнения — главным образом на базе приведенных в конце книги задач. Автор просит читателей рассматривать отмечаемые в тексте случаи "отказа" некоторых конструкций Ф90 как задание на самостоятельное преодоление возникших трудностей и будет признателен за сообщения о найденном решении.

Освоение программирования требует определенной перестройки мышления и (во всяком случае на первых порах) дается с заметным трудом. Поэтому изложению существа предмета предпошлем несколько адресованных начинающему программисту методических рекомендаций:

1) После изучения теоретической части каждого раздела тщательно разберите приведенные в его тексте примеры программ. Затем вновь вернитесь к теории и проследите формы ее реализации в примерах. Выпишите исходные данные задачи и составьте программу ее решения, не глядя в книгу. Проверьте, как будет выполняться ваша программа, *формально* истолковывая ее конструкции в соответствии с правилами Фортрана. Исправьте обнаруженные ошибки. Откройте книгу и сравните приведенный в ней вариант с вашим. Выясните расхождения и оцените их значимость. Составьте программы одной-двух дополнительных задач, рекомендованных для упражнений.

2) Тщательно продумывайте схему программы и не жалейте времени на ее составление: ошибки, допущенные на этом этапе, неизбежно потребуют радикальной переделки программы.

3) К овладению программированием есть только один путь: *самостоятельное составление программ.*

Набор книги выполнен автором в издательской системе emTEX с применением кириллических шрифтов высокого разрешения семейства LH.

Глава 1.

Элементы и объекты программы

1.1. Набор символов

В алфавит Ф90 входят 26 букв английского алфавита (заглавные и строчные буквы различаются только в строках), символ подчеркивания и перечисленные ниже специальные символы:

=	знак присваивания	:	двоеточие
+	знак плюс		пробел
-	знак минус	!	восклицательный знак
*	звездочка	"	кавычки
/	слэш	%	процент
(	левая скобка	&	амперсанд
)	правая скобка	;	точка с запятой
,	запятая	<	меньше
.	десятичная точка	>	больше
$	денежный знак	?	вопросительный знак
'	апостроф		

Другие символы, допускаемые конкретной ПЭВМ и ее программным обеспечением, могут использоваться только в комментариях и текстовых константах.

1.2. Формат программы

Приведем пример простейшей программы:

```
program pr1                              ! pr1 - имя программы
   real x,y,z                            ! объявление типов переменных
   x=4.7                                 ! задание исходных значений
   y=6.5                                 !          "
   z=x*y                                 ! формирование результата
   print *, 'Произведение ',z            ! Вывод на экран пояснения
                                         ! и значения результата
end program pr1
```

Текст ее открывается ключевым словом `program` с именем программы и должен быть закрыт посредством `end` с тем же именем. Имя файла не обязано совпадать с именем процедуры. Заданию действий над объектами программы предшествует *описание* их свойств посредством специальных операторов, условно называемых невыполняемыми. Любая строка может завершиться полем комментария, которое отделяется восклицательным знаком. Комментарий может содержать произвольные символы, не входящие в алфавит языка. Он простирается до конца строки и не влияет на работу программы, но включается в листинг и тем самым служит целям документирования. Пустая строка также считается комментарием.

Текст Фортран-программы можно записать в фиксированном формате (с жестким закреплением позиций за метками, операторами и комментариями) или в свободном формате. Формат играет роль только при компиляции и не влияет на последующие этапы технологического цикла (компилятор по умолчанию считает программы, записанные в файлах с расширением `.f90`, имеющими свободный формат, а с `.f` и `.for` — фиксированный). Фиксированный формат применяется для совместимости со старыми версиями языка. В фиксированном формате позиции 1–5 отводятся для записи метки, а 7–72 — для текста единственного оператора. Отличный от нуля и пробела символ в шестой позиции указывает, что строка продолжает ранее начатый оператор (здесь продолжение может иметь до 19 строк). Звездочка `*` или символ `C` в первой позиции указывают, что вся строка является комментарием. Восклицательный знак в любой позиции, кроме шестой, служит началом комментария.

Мы будем работать в *свободном* формате, позволяющем наглядно показать вложенность конструкций программы. Здесь длина строки допускается до 132 символов и практически определяется шириной строки устройства вывода. Оператор может иметь до 54 строк продолжения. Признаком наличия продолжения является завершающий строку амперсанд (&). Если первый непробел очередной строки тоже амперсанд, то началом продолжения считается первый следующий за ним символ (это существенно при наборе длинных символьных констант, где пробелы значимы). Продолжение комментария следует начинать с восклицательного знака.

Перед любым оператором FPS, не являющимся частью составного оператора, можно поставить (через пробел) *метку*. Метка задается целым числом не более чем из 5 цифр, начальные нули незначимы. Ссылаться по меткам можно только на выполняемые операторы и операторы `format`. *Именные* метки используются только для идентификации составных конструкций языка — главным образом в связи с проблемой выхода из них.

При записи нескольких коротких операторов подряд их можно разместить в одной строке, разделив точкой с запятой:

```
b=1; k=1; s=0
```

1.3. Понятие о проекте

Любая программа рассматривается в FPS как *проект*. Для запуска новой программы необходимо прежде всего создать проект. Пока мы ограничимся простейшим (консольным) проектом, предполагающим ввод данных непосредственно из программы или с клавиатуры и вывод результатов на экран дисплея. Более сложные случаи будут рассмотрены в главе 11. Для входа в систему следует последовательно выбрать в *Win95* `FPS/Microsoft Developer Studio` ("мастерская разработчика" — далее сокращенно MDS), выбрать пункт меню **Программы** или щелкнуть по иконке на рабочем столе — в зависимости от режима установки системы.

На рис. 1.1 представлен экран MDS с панелью инструментов и окнами

- `InfoView` с оглавлением верхнего уровня (слева),
- фрагмента исходного файла (справа),
- окна отладчика с сообщением компилятора об ошибках (внизу).

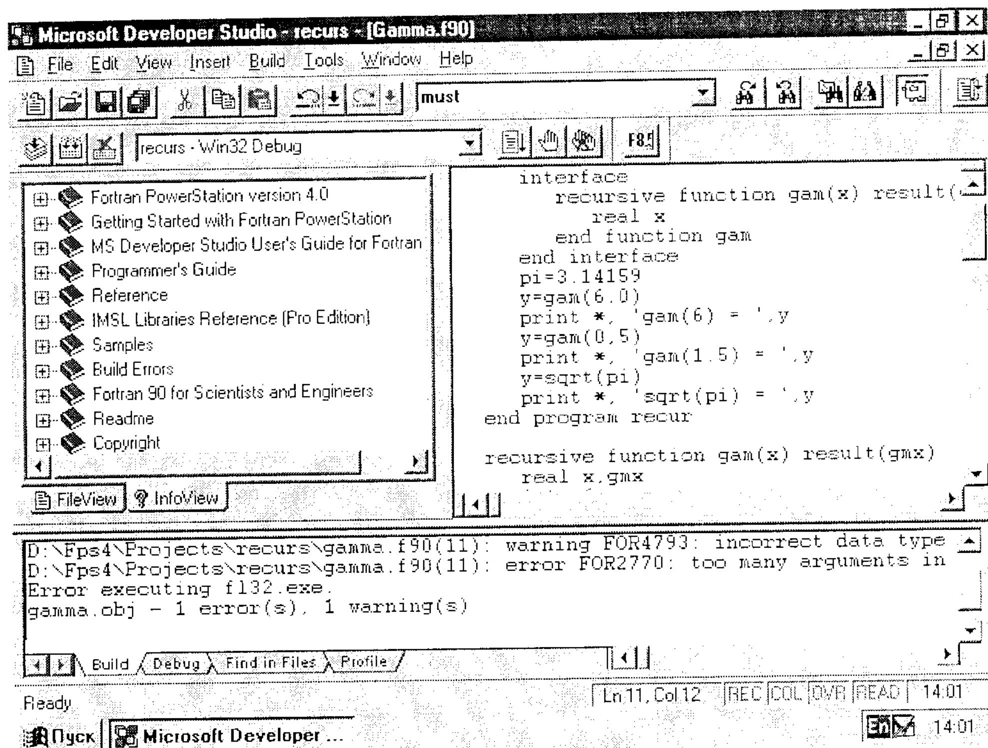

Рис. 1.1. Рабочее окно Microsoft Developer Studio

В данном случае компилятор предупреждает о неверном типе данных и указывает на ошибку в числе аргументов при обращении к функции `gamma` (эти сообщения считаются имеющими разную степень серьезности). Причина обоих сообщений — неверный разделитель при задании аргумента функции (запятая вместо точки между 1 и 5).

При выборе для левого окна режима `FileView` в нем отображаются содержащиеся в проекте файлы и связи между ними, включая отношения использования модулей и вставки файлов — см. аналогичный рисунок в главе 11. Каждый проект представлен папкой древовидной структуры, стандартное раскрытие которой позволяет детально ознакомиться с ее содержимым. Конечные элементы (листья) упомянутых деревьев можно редактировать. Двойной щелчок мышью вызывает соответствующий редактор. Щелчок правой кнопкой мыши высвечивает команды, наиболее употребительные при текущем выборе экрана.

Вызовом ярлыков внизу рабочего окна можно получить информацию о ранее созданных проектах, о включенных в проект графических ресурсах, о содержании включенных в систему справочных электронных книг (`InfoView`). Эта информация

разворачивается и сворачивается обычными для *Windows* методами. Отношения между проектами не обязательно совпадают с отношениями их директорий. Выбор любого элемента проекта и нажатие [Alt+Enter] открывает таблицу его свойств. Двойной щелчок вызывает в окно соответствующий объект.

Распечатка содержимого активного окна производится по [Ctrl+P]. Набранные кириллическим шрифтом фрагменты будут воспроизведены правильно.

Создание нового консольного проекта требует выполнения следующих операций:

- Реализовать цепочку `File/New/Project Workspace/OK/Console Application`.
- Ввести имя проекта.
- Задать расположение проекта в дисковой памяти.

После нажатия `Create` будет создана папка с именем проекта. В этой папке будут размещены файлы проекта с расширениями `.mac` и `.mdp`, фиксирующие структуру проекта. Уже существующий проект открывается по цепочке

`File/Open Workspace/Выбор файла .mdp/Open`,

а закрывается командами `File/Close Workspace`.

Для начала работы с программой выполним `File/New/Text File/OK`. Наберем далее в правом окне текст программы. В процессе набора ключевые слова Фортрана и имена встроенных функций отображаются синим цветом, а комментарии — зеленым. Начальный отступ предыдущей строки наследуется, что облегчает набор структурированных программ в свободном формате. Запишем набранный текст в дисковую память в режиме `Save As...`. Затем добавим его в проект: `Insert/File Into Project/Выбор файла с программой/Add`. Для удаления файла из открытого проекта достаточно выбрать этот файл в окне `File View` и нажать [Del].

При *компиляции проекта* по `Build/Compile` программа переводится с FPS на язык машинного (объектного) уровня. Исправим обнаруженные компилятором ошибки, сообщения о которых появятся в нижнем окне (каталог ошибок можно просматривать по цепочке `Infoview/Build Errors` и далее — в зависимости от фазы обработки программы).

По `Build/Build` создадим *выполняемый* `.exe`-файл, в который связываются объектные коды главной программы и вызываемых ею вспомогательных процедур — как встроенных (системных), так и составленных программистом. Далее по `Build/Execute` запустим его на счет. Результат выводится в специальное окно, которое для облегчения считывания результатов целесообразно развернуть (кнопка в правом верхнем углу). При этом соответственно укрупнится шрифт. Сразу же отметим, что кириллические тексты на экране воспроизводятся в искаженном виде, но в выходные файлы (см. главу 6) пишутся правильно.

Фрагменты содержимого окна результатов можно переносить в другие файлы через системный буфер (`Clipboard`). Для выхода из рабочего окна нажмем клавишу [Esc].

Компиляцию, сборку и запуск программы можно также выполнить с помощью кнопок панели инструментов или клавишных комбинаций `[Ctrl+F8]`, `[Shift+F8]`, `[Ctrl+F5]` соответственно.

1.4. Текстовый редактор

Встроенный текстовый редактор `Emacs` системы FPS, помимо обычных операций с файлами и текстом, для файлов с расширениями `.f90, .f, .for` распознает синтаксис языка и выделяет цветом ключевые слова и имена встроенных функций. Можно также задать выделение цветом отдельных зон (меток, колонки продолжения, колонок 73–80) для работы в фиксированном формате.

1.4.1. Работа с файлами

В меню `File` пользователю предоставляется стандартный набор действий по созданию нового файла (`New`), открытию ранее существовавшего (`Open`), записи файла под старым (`Save`) или новым (`Save As`) именем, закрытия (`Close`) и удаления (`Delete`) файла, его распечатки (`Print`, а также [Ctrl+P]). Некоторые из этих действий требуют дополнительного диалога (обычно — указания директорий и расширений файлов). Файл может быть открыт также щелчком мыши по его иконке в поле проекта.

Одновременно могут быть открыты (естественно, в разных окнах) несколько файлов. Переключение между окнами осуществляется щелчком мыши по нужному окну. Все операции меню `File` выполняются над содержимым активного окна. Звездочка в панели инструментов указывает, что файл изменялся с момента его последнего сохранения.

В нижней части меню `File` постоянно отображаются имена четырех последних файлов, находившихся в работе.

1.4.2. "Горячие" клавиши

Прежде всего отметим команды [Ctrl+Z] отмены последней операции и [Ctrl+A] — выполнения ошибочно отмененной.

Работа с файлами обеспечивается комбинациями [Ctrl+N] — новый файл, [Ctrl+O] — открыть имеемый, [Ctrl+S] — сохранить, [F12] — сохранить как... (запрашивается имя). Компилятор вызывается по [Ctrl+F8], редактор связей — по [Shift+F8]. Программа запускается по [Ctrl+F5]. [Ctrl+Break] прерывает любую обработку. Распечатка содержимого активного окна делается по [Ctrl+P].

Перемещение курсора на слово влево/вправо делается нажатием [Ctrl] в сочетании с клавишей горизонтального смещения. [Ctrl+Home/End] смещает курсор в начало/конец файла. Нажатие [F4] перемещает курсор в строку со следующей ошибкой, найденной компилятором, [Shift+F4] — с предыдущей. Нажатие [Ctrl+G] открывает окно управления переходами. Указанием `What` могут служить номер строки, закладка (`Bookmark`), ошибки и другие отметки в выходном окне MDS, определение объекта программы, ссылка на объект.

Правильность набора сложных выражений можно контролировать переходом к парной скобке. Для этого точку вставки помещают перед скобкой и набирают комбинацию [Ctrl+M]. Точка вставки перемещается к парной скобке, а при отсутствии таковой подается звуковой сигнал.

Неименованные закладки создаются в *активной строке* (содержащей точку вставки) нажатием [Ctrl+F2]. После этого строка подсвечивается или отмечается с края. Переход к ближайшей закладке ниже по тексту выполняется нажатием [F2], в обратном направлении — по [Shift+F2]. Закладка в активной строке уничтожается нажатием [Ctrl+F2]. Неименованные закладки *не сохраняются* между сеансами редактирования.

Именованная закладка создается в активной строке через диалог в меню `Edit/Bookmark`. Переход к конкретной закладке выполняется выбором ее имени в списке `Edit/Bookmark/Name` и нажатием [Go To]. Аналогично производят удаление закладки. Переход к ближайшей закладке выполняется так же, как к неименованной.

Выделение текста начинается щелчком левой кнопки мыши и продолжается протаскиванием ее указателя. Выделение текстового прямоугольного блока начинается по [Ctrl+Shift+F8]. Расширять выделенную зону можно нажатием [Shift] в сочетании с одной из клавиш управления курсором, а также протаскивая мышь с нажатой [Alt]. Те же действия при нажатых [Ctrl+Shift] будут расширять выделенную зону на *слово*. [Ctrl+Shift+Home/End] расширят выделение до начала/конца файла соответственно. Нажатие [Ctrl+Shift+M] при курсоре, установленном на скобку, выделит текст до парной ей скобки. Удобным средством заранее спланированного выделения фрагмента является щелчок [LM] в его начале и [Shift+LM] — в конце.

Преобразование выделенного текста в нижний регистр производится по [Ctrl+U], а в верхний — по [Ctrl+Shift+U].

Операции с буфером выполняются над выделенным текстом: [Ctrl+X] вырезает текст в буфер (`Clipboard`), [Ctrl+C] — копирует, [Ctrl+V] вставляет текст из буфера.

Удаление выделенного текста выполняется нажатием [Del]. [Ctrl+Back] удаляет слово слева от курсора, [Ctrl+Del] — справа. Нажатие [Ctrl+Y] удаляет текущую строку.

Табуляция задается нажатием [Tab], последующие строки автоматически наследуют заданный отступ. Отмена отступа делается по [Shift+Tab]. Отступы для вложенных конструкций можно регулировать заданием шага табуляции в меню `Tools/Options/Tabs`. На "посимвольно" заданные отступы эта установка не влияет. Прохождение [Backspace] через символ табуляции отменяет табуляцию. Для вставки символа табуляции перед *группой строк* нужно выделить эти строки и нажать [Tab]. Обратные действия выполняются по [Shift+Tab].

Поиск вхождений выделенного текста начинается по [Alt+F3]; по [Ctrl+F3] находится следующее вхождение, по [Shift+F3] — предыдущее. Полезны режимы нахождения парных конструкций: [Ctrl+M] — круглой скобки, [Ctrl+Shift+.] — `if`-ограничителя. При этом курсор должен стоять на одном из элементов пары. В режиме `Browse` (см. разд. 11.8) по [Alt+F1] можно найти определение (описание атрибутов) имени, а по [Alt+Shift+F1] — вхождения имени в программу.

С помощью команды `Edit/Find` можно организовать поиск в активном окне следующих типов строк (в смысле "буквенно-цифровых последовательностей"):

`Match Whole Word` — строки алфавитно-цифровых символов, ограниченных любыми символами, кроме алфавитно-цифровых и подчеркивания;

`Match Case` — то же с учетом регистров букв;

`Regular Expression` — по задаваемому шаблону (правила описания шаблона довольно сложны, и мы их не приводим),

`Set Bookmarks on All` — установка закладок при каждой встрече с искомой строкой.

Поиск организуется от точки вставки по командам `Find Next` или `Mark All`. Для поиска (вставки) следующего вхождения строки вниз по тексту используется [F3], в обратном направлении — [Shift+F3]. Ниспадающий список хранит 16 вхождений,

найденных последними. Имеется возможность выполнить поиск в группе выделенных файлов с общим расширением.

Аналогичным образом, вызвав функцию `Replace`, можно заменить найденную строку другой — с тем же набором вариантов. Команда `Replace All` выполняет замены во всем файле.

1.4.3. "Заказная" распечатка

Система позволяет распечатать весь файл, выделенный текст, содержимое активного окна. Такая "заказная" распечатка организуется через меню `File/Print`, а также нажатием [Ctrl+P]. Имеется возможность настройки печати через диалог в `Page Setup` — можно менять поля, выравнивание текста, задавать верхний и нижний колонтитулы и нумерацию страниц.

1.4.4. Управление окнами просмотра

При редактировании мы можем управлять окнами с файлами, переключаться между ними и расщеплять (`Split`) изображение. Последняя команда нужна для одновременного вывода на дисплей двух частей файла. Здесь необходимо

1) Активировать окно с нужным файлом.

2) Выбрать команду `Window/Split`.

3) Появившуюся разделительную линию перетащить в нужное место файла.

4) Щелкнуть левой кнопкой мыши.

Развертывание активного окна выполняется командами `View /Full Screen`. Для отображения панели с инструментами и в этом режиме нужно воспользоваться командами `Tools/Options`.

1.5. "Хороший стиль" программирования

Хорошая программа должна удовлетворять требованиям правильности, эффективности, читабельности, удобства модификации, живучести (устойчивости), документированности. Степень важности этих качеств зависит от конкретной задачи и условий эксплуатации программы. Наивысший приоритет имеет правильность.

Общее правило разработки — сначала *обдумать* задачу. Поскольку обдумывание идет в привычных разработчику терминах предметной области и математики, оно обычно порождает простое, сжатое и правильное описание проблемы и метода решения. Далее выполняется пошаговая детализация с постепенным приближением формы описания к языку программирования. Детализация касается не только программ, но и структур данных. Каждый шаг детализации для гарантии его правильности должен быть достаточно простым.

Хороший стиль программирования уменьшает количество ошибок разработчика, делает логику программы более прозрачной и облегчает ее понимание пользователями. Желательно как можно раньше и в комплексе освоить соответствующие рекомендации и на их основе сформировать собственный стиль. Поэтому в приводимом ниже перечне советов программисту некоторые пункты вынужденно "забегают вперед".

1. Обязательно комментируйте: назначение каждой программной единицы; идею, заложенную в ее построение (возможна ссылка на опубликованное описание алгоритма); область применения и/или ограничения на использование; особые указания по использованию; нетривиальную логику; взаимодействие с операционной системой. Длинные комментарии размещайте перед поясняемым текстом, короткие включайте непосредственно в текст. *Смысл* формальных параметров процедур, переменных и констант комментируйте при их объявлении. Вообще количество комментариев в программе должно быть больше, чем это кажется необходимым ее автору (хотя бы потому, что с этой программой придется иметь дело и другим программистам).

2. Для процедур и переменных применяйте мнемонические обозначения, перечисляйте их в программной единице в алфавитном или ином логически обоснованном порядке. Сохраняйте эту логику в пределах всей программной разработки.

3. Завершенные части текста разделяйте пробельными строками и комментариями. Вложенность структур программы показывайте соответствующими отступами. Пользуйтесь пробелами для улучшения читаемости громоздких выражений.

4. Группируйте вместе операторы формата.

5. Избегайте употребления операторов перехода `go to` и устаревших конструкций языка. Ставьте метки только у тех операторов, на которые реально потребовались ссылки. Сами метки для быстрого нахождения их в тексте должны быть вынесены в крайнее левое поле и упорядочены по возрастанию.

1.6. Лексемы

Основные значимые последовательности буквенно-цифровых или специальных символов — метки, ключевые слова, имена, константы и разделители

`/  (  )  (/  /)  ,  =  =>  :  ::  ;  %`

называются *лексемами*. Имя, константа или метка должны отделяться от соседней конструкции перечисленных классов хотя бы одним пробелом. Пробелы сами по себе значимы только в текстовых константах и для улучшения читаемости текста могут расставляться произвольно.

Понятие *тип данных* охватывает множество допустимых значений, средства их обозначения и набор разрешенных над ними действий. Шесть типов являются встроенными (доступны программисту без каких-либо дополнительных мер):

целый	—	`integer`
вещественный	—	`real`
вещественный с двойной точностью	—	`double precision`
комплексный	—	`complex`
текстовый	—	`character`
логический	—	`logical`

Комбинируя данные встроенных типов, можно получить *производные* типы — записи, списки, объекты интервальной арифметики и т.п. Примеры буквальных целых констант:

`1    0    666    -44811    +36`

Типы данных могут иметь дополнительные разновидности. Можно использовать положительные целые числа, записанные в двоичной, восьмеричной и шестнадцатиричной системе.

Вещественные буквальные константы (литералы) состоят из целой части числа со знаком или без знака, десятичной точки, дробной части и степенной части. Все эти элементы необязательны, но величина числа должна определяться

однозначно. Должен присутствовать также хотя бы один из признаков "вещественности" (точка в мантиссе либо разделитель степенной части), поскольку целые и вещественные числа принципиально по-разному представляются в ЭВМ. Примеры:

`1.    .01    1e-6    6.62e3`

В записи константы двойной точности разделитель `e` заменяется на `d`, например `pi=3.141592653589793D+00`.

Стандартные типы числовых данных в FPS обеспечивают представление целых в диапазоне $\pm 2,147 \times 10^9$, вещественных с одинарной точностью по модулю $1,18 \times 10^{-38}..3,4 \times 10^{38}$, вещественных с двойной точностью по модулю $2,22 \times 10^{-308}..1,8 \times 10^{308}$.

Обозначением комплексной константы служит пара буквальных констант целого или вещественного типа, разделенная запятой и заключенная в скобки.

Текстовые литералы состоят из строки *произвольных* символов, заключенных в кавычки либо апострофы (пример: `'Рога и копыта'`). Кавычка или апостроф, являющиеся *частью* строки, должны быть удвоены: `'O''Henry'`. Заметим, что длина последней строки считается равной семи.

Логических констант в FPS имеется две: `.true.` и `.false.` ("истина" и "ложь" соответственно).

Имена (идентификаторы) используются для обозначения объектов программы: простых переменных, массивов, процедур, встроенных функций. Они могут состоять из букв, знака доллара, цифр и знака подчеркивания, причем первым символом должна быть буква или знак $. Длина имени ограничивается 31 символом. Заметим, что запись "`ab`" означает не умножение `a` на `b`, но двухсимвольное имя. Предпочтительно выбирать мнемонические имена переменных (как это было сделано выше для π) и по возможности сохранять наименования объектов, выбранные при математической постановке задачи.

Зарезервированных слов в FPS нет; однако, используя ключевые слова (их мы будем вводить постепенно) или имена встроенных функций не по назначению, программист принимает на себя всю ответственность за возможные последствия.

Строчные и заглавные буквы различаются только в текстовых константах и комментариях. Имена могут быть глобальными и локальными. К глобальным именам относятся имена главной программы, вызываемых внешних процедур, используемых модулей, блоков данных и общих блоков. На них можно ссылаться из любой точки программы. Локальные имена имеют смысл только в своей области видимости (см. разд. 5.5) и могут иметь совершенно иное значение за пределами этой области.

1.7. Описания

Объекты разных типов (прежде всего целые и вещественные) принципиально по-разному представляются на машинном уровне, и однотипные действия над ними (например, сложение) программируются с помощью различных команд. Для выполнения компиляции необходимо задать свойства операндов, т.е. указать их типы. Эти указания даются в ***операторах описания***, которые определяют только процесс компиляции и потому не считаются выполняемыми операторами. Описания могут группироваться различными способами: по свойствам (для каждого свойства перечисляется список имеющих его объектов) и по объектам (для каждого объекта задается набор свойств). Для структурированных объектов (массивов) рекомендуется второй подход. Часто используются и комбинированные варианты.

Оператор описания в любом случае начинается с указания свойства или списка свойств. Далее (при нескольких атрибутах — обязательно после сдвоенного

двоеточия) перечисляются имена обладающих этими атрибутами объектов. Один объект может упоминаться в нескольких описаниях, но ему не могут присваиваться повторяющиеся или противоречащие друг другу свойства. Примеры описания переменных встроенных типов:

```
integer            :: k
real               :: x,y
double precision   :: z
complex            :: voltage
character(len=20)  :: heading
logical            :: test
dimension          :: x(30),y(0:4,5)
```

Переменная heading имеет дополнительным атрибутом описателя длину строки. Этот описатель может быть задан и по правилам Ф77 в виде character*20. Оператор dimension дополнительно к ранее назначенным типам значений x и y указывает структуру упомянутых объектов: это *массивы* с определяемой граничными парами нумерацией компонент (x — вектор, y — матрица).

FPS не принадлежит к языкам со строгой типизацией. Если тип некоторого объекта не задан, он определяется по умолчанию: объекты с именами, начинающимися на i,j,k,l,m,n, относятся к стандартному целому типу, остальные — к вещественному. Правила умолчания, применяемые в данной программной единице, можно изменить и/или дополнить оператором implicit вида

```
implicit real (k,n,x), complex (p-t)
```

Предполагается, что последняя спецификация избавит программиста от длинного перечня вводимых им комплексных переменных.

Случайная описка в имени порождает подменяющую нужный объект новую переменную, что приводит к трудно диагностируемым ошибкам периода выполнения программы. Полезную возможность ужесточения контроля в процессе компиляции через полную отмену умолчаний предоставляет оператор implicit none. При его использовании любая ссылка на неописанную переменную будет восприниматься как синтаксическая ошибка.

Обычно трудно составить исчерпывающий список объектов программы до того, как записаны ее операторы. Практически начинают с минимума обозначений и, программируя операторы, дополняют этот список по мере необходимости. Покончив с операторами, окончательно оформляют описания — объединяют по типам, упорядочивают имена по алфавиту, комментируют описания отдельных объектов или их групп, задают начальные значения.

Часто употребляемые константы для защиты от случайных изменений нужно задавать операторами parameter вида

```
real, parameter :: c=299792.458 !! скорость света в км/с
```

и затем ссылаться на них *по именам*. Любая попытка изменить значение именованной константы будет блокирована. Этот прием автоматически распространяет однократную коррекцию значения на всю программную единицу (не нужно отыскивать и править каждое вхождение буквальной константы). В одном операторе можно указать через запятую несколько констант, причем последующие могут выражаться через предыдущие. Через параметры можно также задавать границы массивов, изменяемые от одного выполнения программы к другому (это не относится к границам, вычисляемым в процессе счета).

Диапазон стандартных целых зависит от длины машинного слова и на уровне языка не оговаривается. Стандартное представление вещественных переменных соответствует формату машинного слова (на IBM PC это 6–7 десятичных цифр). Повышенная точность (`double precision`, `real*8`) достигается при работе с двойными словами (16 цифр). Промежуточные значения разрядности при счете эмулируются двойной, и их использование с учетом возможности вывода на печать требуемого количества цифр бесполезно.

Комплексные константы записываются парой чисел, заключенных в круглые скобки. Вещественная и мнимая части комплексной переменной, объявленной как `complex`, получают атрибут `real`. При указании `double complex` они будут представлены с двойной точностью.

1.8. Выражения

Выражение — это формула для получения значения. Выражения образуются из операндов и знаков операций, объединяемых по правилам синтаксиса FPS. В качестве операндов выражения используются буквальные и именованные константы, переменные, указатели функций. Они могут быть скалярами или массивами. *Элемент* массива вызывается по имени массива, за которым в круглых скобках перечисляются индексы: `a(3,i)`. Индексы должны быть целыми и находиться в пределах, заданных описанием массива.

1.8.1. Арифметические выражения

Для *арифметических* операций установлен следующий порядок старшинства операций:

$**$ — возведение в степень,
$*$, $/$ — умножение и деление,
$+$, $-$ — сложение и вычитание.

Операции одного ранга старшинства выполняются слева направо (возведение в степень — справа налево). Приведем примеры записи арифметических выражений:

$\dfrac{a+b}{cd}$	`(a+b)/(c*d)` или `(a+b)/(c*d)`
$p + q\sin^2\alpha$	`p+q*sin(alpha)**2`
$e - f_{i,k}^2 g_{k,j}$	`e-f(i,k)**2*g(k,j)`

Несколько указаний начинающим программистам:

1) Нельзя пропускать знак умножения или обозначать его точкой.

2) Деление нельзя изображать дробью с горизонтальной чертой или двоеточием.

3) Запрещается применять квадратные и фигурные скобки.

4) При переносе длинного выражения на следующую строку нельзя повторять знак операции.

В случае данных целого типа результат деления *усекается* в сторону нуля (дробная часть отбрасывается). Этот эффект следует иметь в виду и при возведении в целую отрицательную степень, заключительным шагом которого является вычисление обратной величины. Допускается смешение операндов разных типов. В этом случае производится автоматическое приведение типов к более мощному по шкале "целый – действительный – комплексный", а при разной разрядности — к большей разрядности. Таким образом, для профилактики упомянутого выше усечения при делении достаточно сделать один из операндов *формально* вещественным.

Возведение в целую степень выполняется как серия последовательных умножений, а в действительную — через логарифмирование основания, умножение на показатель степени и вычисление показательной функции. Атрибут показателя определяется по формальным признакам. Поэтому возведение в степени `2` и `2.0`, а также в степени `k` и `x` при общем числовом значении названных переменных, будет существенно различным по трудоемкости. Для вычисления значения квадратного корня рекомендуется использовать не операцию возведения в степень `0.5`, а стандартную функцию `sqrt`. Кстати, при возведении в степень `1/2` результатом всегда будет единица (нулевая степень основания).

Ограниченность разрядности и диапазона чисел, представимых в любой ЭВМ, делают порядок выполнения действий одного ранга не вполне безразличным. При сложении чисел с существенно различными порядками нужно учитывать возможную потерю значимости меньшего по модулю в процессе выравнивания порядков. Поэтому сложение чисел одного знака во избежание потери точности следует начинать с меньших по модулю слагаемых. В процессе накопления произведения при неудачном выборе пар сомножителей можно получить переполнение разрядной сетки или "машинный нуль". Нужно стремиться к тому, чтобы каждое очередное произведение было по модулю возможно ближе к единице.

FPS по понятным причинам считает некорректными деление на нуль, возведение нуля в нулевую или отрицательную степень, возведение отрицательного числа в вещественную степень.

1.8.2. Встроенные функции

Естественным требованием к языку, ориентированному на научные приложения, является достаточно богатый набор *встроенных*, т.е. являющихся частью языка, функций и процедур. Перечислим функции, необходимые в простых численных расчетах. Прежде всего это функции с возможным преобразованием типов:

`nint(a)` — ближайшее целое,

`anint(a)` — вещественная форма ближайшего целого,

`aint(a)` — вещественная форма целой части,

`ceiling(a)` — ближайшее большее целое,

`floor(a)` — наибольшее целое, не превосходящее аргумента,

`abs(a)` — модуль аргумента (для целого — целый),

`real(a)` — вещественная часть комплексного аргумента,

`aimag(z)` — мнимая часть комплексного аргумента,

`cmplx(x[,y])` — комплексное число с указанными частями.

Перечисляемые ниже функции сохраняют тип своего аргумента:

`conjg(z)` — сопряженное значение комплексной переменной `z`,

`max(a1,a2[,a3,...])` — максимальное из значений аргумента (аналогично `min` — минимальное),

`modulo(a,m)` = `a-floor(a/m)*m` (`a` и `m` целые либо вещественные),

`sign(a,b)` — модуль `a` со знаком `b`.

Математические функции `sqrt`, `sin`, `asin`, `cos`, `acos`, `tan`, `atan`, `exp`, `log` (натуральный), `log10`, `sinh`, `cosh`, `tanh` с учетом того, что префикс `a` означает обратную круговую функцию, дополнительных пояснений не требуют. Отметим лишь, что перечисленные имена являются родовыми и фактически определяют *семейства* функций, каждое из которых выполняет нужные вычисления для аргументов различных встроенных типов и возвращает результат того же типа. Как правило можно ограничиться этим родовым именем, однако при вызове процедур с аргументом-функцией необходимо подставлять специфическое имя последней (см. более подробные руководства). Обратные гиперболические функции в FPS доступны через вспомогательную математическую библиотеку IMSL.

К встроенным процедурам можно обращаться с *ключевыми* фактическими параметрами. Например,

```
call date_and_time (time=t)
```

присваивает текущее время текстовой переменной `t` в формате `ччммсс.ссс`, где `ч`, `м` и `с` означают цифры часов, минут и секунд соответственно. Другие необязательные параметры в данном обращении отсутствуют.

1.8.3. Логические выражения

В программировании для ЭЦВМ большую роль играет исчисление высказываний. *Высказыванием* называется суждение, относительно которого имеет смысл утверждать, что оно истинно или ложно. Алгебра логики занимается *формальным* выведением истинности сложных высказываний из истинности элементарных и эквивалентными преобразованиями их. При этом происходит абстрагирование от содержательной стороны, так что высказывания "в огороде бузина" и "в Киеве – дядька", если оба они истинны или ложны, формально считаются эквивалентными.

Логическими константами являются `.true.` и `.false.`, логическими переменными — объявленные с атрибутом `logical`. Простейшими примерами логических *выражений* являются *отношения*:

```
.LT.        <
.LE.        <=
.EQ.        ==
.NE.        /=
.GT.        >
.GE.        >=
```

(для каждой операции допускаются оба варианта, пробелы между отдельными символами недопустимы). Если хотя бы один из операндов комплексный, то возможна только проверка на равенство и неравенство. Вычислению отношений предшествует выполнение арифметических действий и, если необходимо, приведение к более

мощному типу. Проверку *равенства* вещественных и комплексных чисел следует производить с допуском, например в форме `abs(X-3.0)<1e-7` .

Логические *операции* — это в порядке убывания приоритетов

`.not.` – отрицание,
`.and.` – логическое умножение (И),
`.or.` – логическое сложение (ИЛИ),
`.xor.` – исключающее ИЛИ (в стандарте Ф90 отсутствует),
`.eqv. , .neqv.` – эквивалентность или ее отрицание.

Логические операции, как и отношения, дают значения `.true.` или `.false.` Вычисление отношений предшествует выполнению логических операций. Приведем значения истинности логических операций `A<оп.>B` — табл. 1.1:

Таблица 1.1. **Результаты логических операций**

A	.true.		.false.	
B	.true.	.false.	.true.	.false.
.not. a	.false.	.false.	.true.	.true.
a .and. b	.true.	.false.	.false.	.false.
a .or. b	.true.	.true.	.true.	.false.
a .xor. b	.false.	.true.	.true.	.false.
a .eqv. b	.true.	.false.	.false.	.true.
a .neqv. b	.false.	.true.	.true.	.false.

Старшинство логических операций соответствует порядку их перечисления в этой таблице (две последних имеют одинаковый ранг старшинства). Операции логического сложения и умножения естественно обобщаются на большее число операндов. Доказано, что любая логическая функция может быть выражена через значения исходных переменных с помощью операций отрицания, логического сложения и логического умножения.

Пример.

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} < 1 \wedge x > 0 \vee /y < x$$

(отрицание последнего отношения обозначено слэшем) должно быть запрограммировано как

```
(x/a)**2+(y/b)**2<1 .and. x>0 .or. .not. y<x
```

Полезно иметь в виду упрощающие анализ сложных логических выражений тождества

```
b .and. .not. b   .eqv.  .false.
b .or.  .not. b   .eqv.  .true.
b .and. .false.   .eqv.  .false
b .and. .true.    .eqv.   b
b .or.  .false.   .eqv.   b
b .or.  .true.    .eqv.  .true
```

а также правила де Моргана

```
.not. a .or.  .not. b .eqv. .not.(a .and. b)
.not. a .and. .not. b .eqv. .not.(a .or. b)
```

Вычисление логического выражения прекращается при выяснении его истинности (выявлении истинного слагаемого суммы или ложного сомножителя произведения). Поэтому отдельные подвыражения могут оказаться не вычисленными и ссылаться на них в последующем тексте нельзя.

Для производных типов программист может определить собственные операции и ввести произвольные обозначения для них — см. главу 7.

1.9. Инициализация и изменение значений переменных

Оператор *присваивания* задает переменной, указанной в его левой части, значение правой. Правая часть в общем случае определяется выражением, которое предварительно вычисляется. Расчеты индексов левой части (если таковые имеются) и значения правой части выполняются независимо и не оказывают взаимного влияния. Выражение в *левой* части стоять не может.

Приведем примеры операторов присваивания:

```
y(n)=a*x**2+b*x+c
n=n+1
```

Второй пример (читается "n положить равным n+1") наглядно демонстрирует различие между равенством и присваиванием и *динамический* смысл последнего: в правой части используется старое, а в левой — новое значение переменной.

Левая и правая части оператора присваивания должны быть согласованы по типу: обе быть числового, логического, символьного или общего производного типа. Предполагается, что все переменные, входящие в правую часть, были предварительно означены (нарушение этого требования компилятором не обнаруживается).

При выполнении "численного" присваивания тип результата преобразуется к типу левой части, причем допускается как сужение типа (например, присваивание целой переменной вещественного значения), так и его обобщение. Разумеется, при сужении происходит частичная потеря информации — в частности, округление значения правой части.

Выражение-массив может быть присвоено переменной-массиву такой же формы. Правая часть оператора присваивания может быть скалярной. Массивы должны быть конформны. Если правая часть — скаляр, всем элементам левой части присваивается его значение. Работа с массивами подробно рассмотрена в главе 3.

Никакой *автоматической* инициализации переменных (например, обнуления) FPS не производит. Начальные значения переменных могут быть присвоены обычными операторами присваивания — см. пример программы на стр. 7, а также заданы непосредственно при их описании, например

```
real:: x=1.7, y=6.94
```

Для означивания большого числа переменных предпочтительно применение оператора `data x/1.7/, y/6.94/` или `data x,y /1.7, 6.94/`. Этот способ следует использовать и для инициализации массивов. Так, матрицу

$$a = \begin{bmatrix} 1,6 & 2,5 & -11,4 & 7,6 \\ 6,3 & 2,5 & 4,8 & -5,2 \\ 2,5 & -0,8 & 4,8 & 4,4 \end{bmatrix}$$

с описанием `real a(3,4)` можно означить оператором

```
data a/1.6, 6.3, 3*2.5, -0.8, -11.4, 2*4.8, 7.6, -5.2, 4.4/
```

Отметим здесь порядок заполнения матрицы (*по столбцам*)! и использование повторителей для сокращенного представления идущих подряд одинаковых значений.

1.10. Простейший ввод-вывод

Простейший ввод данных в программу организуется с клавиатуры оператором вида

```
read *, x,y
```

Предполагается, что значения перечисленных объектов набраны в одной строке через пробел или запятую. Оператор `read *` с пустым списком ввода можно применять для организации паузы в вычислениях (например, внутри циклов).

Оператор вида

```
print *, '<текст>', <имя_результата>
```

позволит вывести результат с поясняющим его текстом (например, "сумма ряда равна "). Значение результата будет выведено в формате, определяемом описанием соответствующей переменной. Более сложные конструкции, позволяющие управлять формой вывода и обменом информацией с файлами, обсуждаются в главе 6.

Глава 2.

Операторы управления

Выполнение программы начинается с первого выполняемого оператора главной программы и заканчивается выходом за закрывающий ее `end`. Оператор `stop` применяется в диагностических целях и при обработке "нештатных ситуаций". Конструкция `stop [<n>]` приводит к останову программы в заданном месте с выводом на печать `n` — целой константы без знака, позволяющей установить, какой из возможных остановов сработал. Вместо `n` может быть задана (в апострофах или кавычках) строка символов, дающая необходимые пояснения "открытым текстом".

В нормальном режиме операторы главной программы или процедуры выполняются в порядке их записи. Для изменения этого порядка служат *операторы управления.* Управляющее условие всегда является скаляром или приводится к скаляру.

Некоторые операторы управления имеют *блочную* конструкцию, которая начинается с ключевого оператора-заголовка, может содержать промежуточные ключевые операторы и заканчивается парным заголовку оператором завершения. Войти в такую конструкцию можно *только через ее заголовок.* Из блока возможны переходы

- на другой оператор того же блока,
- на заключительный оператор той же конструкции,
- на оператор, находящийся вне данной конструкции.

Исполняемые конструкции могут быть вложенными.

Оператор *перехода* имеет вид

```
go to <метка>
```

Метка (числовая) должна быть при *выполняемом* операторе.

Отдельного разговора заслуживает *назначаемый* переход. В этом случае в программу включается оператор вида

```
go to <целая_переменная> (<список_меток>)
```

Пусть эта переменная — `j`. Тогда после выполнения "назначения"

```
assign 139 to j
```

оператор `go to j (30,70,139,260)` вызовет переход к оператору с меткой 139. Переменная `j` может быть использована только в обсуждаемом контексте, список меток необходим для контроля допустимости запланированных возвратов.

Современный стиль рекомендует "программирование без `go to`", и FPS имеет достаточный набор обсуждаемых ниже альтернатив операторам перехода. Однако

возможны ситуации, когда go to окажется наиболее простым и логичным средством.

2.1. Разветвления

Простейший условный оператор имеет вид

```
if (<скалярное ЛВ>) <действие>
```

Здесь и далее ЛВ — логическое выражение. Оно должно записываться обязательно в скобках. Пример:

```
if (x<y) z=x
```

Если необходимо связать с условием выполнение нескольких действий, применяется более сложная блочная конструкция:

```
if (<ЛВ>) then
    <блок 1>
else
    <блок 2>
end if
```

Каждый из блоков может содержать произвольную последовательность выполняемых операторов. При отсутствии альтернативы второй блок вместе с **else** опускается. Наконец, при необходимости в дополнительных проверках используется **else if**.

Пусть необходимо найти $z = \min\{x_1, x_2, x_3\}$. Тогда следует записать условный оператор

```
if (x(1)<x(2)) then
   if (x(3)<x(1))  then
      z=x(3)
   else
      z=x(1)
   end if
else
   if (x(3)<x(2))  then
      z=x(3)
   else
      z=x(2)
   end if
end if
```

Разумееется, возможны (но нежелательны) и более сложные конструкции. В подобных случаях рекомендуется обязательно выделять вложенность отступами (как в вышеприведенном нарочито переусложненном примере) или метить *if-именами* начало и конец условной конструкции каждого уровня. Цепочка **else if** может быть сколь угодно длинной, но просто **else** для каждого **if** должен быть один. Сразу же отметим критерий правильной вложенности: каждая конструкция должна *целиком* содержаться в другой или в ее части (неполное зацепление ошибочно). Это правило относится также к обсуждаемым ниже циклам и процедурам.

Примерами задач на условные конструкции могут служить

- вычисление функции, задаваемой различными выражениями в различных диапазонах аргумента,
- описание геометрических фигур и проверка принадлежности точек к ним (например, точки — к внутренней части круга с заданными радиусом и положением центра),
- описание работы релейно-контактных схем и т.п.

При использовании в программе отношений арифметических выражений, принимающих действительные значения, следует помнить о возможных погрешностях их представления и проверять равенства с определенным допуском.

Приведем пример условного оператора, задающего характеристику линейного звена системы автоматического управления с ограничением `c` при модуле входного сигнала больше `b` и зоной нечувствительности шириной `a`:

```
if (abs(x)<a) then
   z=0
else
   if (abs(x)>b) then
      z=c*sign(x)
   else
      z=c/(b-a)*(x-a*sign(x))
   end if
end if
```

Длинной последовательности проверок иногда удается избежать, применив оператор `select`:

```
[<имя>:] select case (<выражение>)
            case (<список_1>)
                <блок_1>
            case (<список_2>)
                <блок_2>
            ..................
           [case default
                <блок_n>]
         end select[<имя>]>
```

Вычисление выбирающего выражения должно давать скаляр текстового, логического или целого (но не вещественного!) типа. С каждым из блоков действий связывается список значений и/или записанный через двоеточие диапазон значений селектирующей переменной, при котором этот блок выполняется (например, `1:3,5,8:10`). В "крайних" случаях одна из границ диапазона может отсутствовать, в частных вариантах диапазон может сводиться к конкретному значению. Перекрытие диапазонов не допускается. Заключительный (необязательный) случай `default` задает действия для "непредвиденных" ситуаций.

Выполнение любой условной конструкции в итоге сводится к некоторой результирующей совокупности операторов или пустому оператору. Если эта совокупность сама не определяет своего преемника, т.е. не является оператором перехода, то вслед за ней выполняется текстуально следующий за условной конструкцией оператор.

2.2. Циклы

Идущие подряд однотипные вычисления можно запрограммировать однократно с помощью условного оператора перехода на начало соответствующей группы команд ("тела цикла") — разумеется, после его настройки на очередное повторение. Однако эти функции во всех языках программирования реализуются более наглядными *операторами цикла*. Все циклы в FPS строятся с *верхним окончанием*, т.е. необходимость действий из тела цикла проверяется перед первым выполнением входящих в него операторов.

2.2.1. Цикл с шагом

Цикл (do-группа) с шагом задается конструкцией вида

```
do i=n1,n2[,n3]
   <блок>
end do
```

Здесь `n1` и `n2` — начальное и конечное значения управляющей переменной (в данном случае `i`). По умолчанию необязательный параметр `n3` (шаг) равен 1. Шаг может быть и отрицательным. Все параметры могут быть константами или выражениями одного из стандартных числовых типов (`integer`, `real`, `double precision`), переменная цикла также принадлежит к тому же типу. Параметры и требуемая кратность выполнения вычисляются *статически* — до входа в цикл. Переменная цикла не должна переопределяться в теле цикла (эта ошибка обнаруживается компилятором).

Циклы указанного вида обычно используются при обработке массивов. Покажем это на поучительной задаче расчета многочлена по схеме Горнера:

$$a_5x^5 + a_5x^5 + a_5x^5 + a_5x^5 + a_5x^5 + a_0 = ((((a_5x + a_4)x + a_3)x + a_2)x + a_1)x + a_0$$

"Скобочный" вариант сводится к повторяющейся последовательности умножений и сложений и идеально реализуется циклом обсуждаемого типа. Пример программы:

```
program gorner
   integer i
   real a,x,p
   dimension a(0:5)
   data a/2,3,4,5,6,7/
   x=0.7
   p=a(5)
   do i=4,0,-1
      p=p*x+a(i)
   end do
   print *, p
end program gorner
```

Аналогично можно запрограммировать расчет непрерывной дроби и обобщенного многочлена (обобщенная степень $x^{[k]}$ по определению равна $x(x-1)\dots(x-k+1)$).

Если переменная цикла — вещественная (например, при расчете таблицы значений функции для заданного диапазона значений аргумента), погрешности

представления текущего и граничных ее значений могут привести к ошибке в числе повторений. В таких случаях рекомендуется добавлять к граничному значению n2 некоторую долю (например, половину) шага — с учетом его знака. Проще и надежнее, однако, предварительно вычислить количество значений аргумента и построить цикл по их *номерам*, пересчитывая аргумент в теле цикла.

Переменная цикла и ее границы в некоторых случаях могут отсутствовать. Тогда функции заголовка (продвижение переменной и проверка на окончание цикла) должны быть реализованы в его теле.

2.2.2. Цикл с условием

Циклы с условием используются при заранее неизвестном количестве повторений. Они задаются конструкцией

```
do while (<ЛВ>)
   <блок>
end do
```

Тело цикла выполняется, пока <ЛВ> истинно.

Классическим примером такой задачи является расчет суммы членов степенного ряда. Пусть необходимо вычислить

$$S(x) = \sum_{k=0}^{\infty}(-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

с точностью $\varepsilon = 10^{-6}$. Прежде всего обсудим математическую сторону проблемы. Предположим, что $x > 0$. Погрешность суммы членов $\{b_k\}$ знакочередующегося ряда не превосходит модуля первого отброшенного члена; следовательно, условие выполнения тела цикла имеет вид $|b_k| > \varepsilon$. Для любого k было бы вопиющей нерациональностью считать входящие в выражение для общего члена степень и факториал "от печки", имея их значения при предыдущем k. Далее, при $|x| > 1$ как степень, так и факториал будут возрастать по k, что может привести к потере точности и/или переполнению разрядной сетки при *раздельном* вычислении их. Удобнее всего найти отношение последовательных членов ряда

$$\frac{b_k}{b_{k-1}} = \frac{(-1)^k x^{2k+1}(2k-1)!}{(2k+1)!(-1)^{k-1}x^{2k-1}} = -\frac{x^2}{2k(2k+1)}$$

и в дальнейшем рекуррентно вычислять члены $b_k = b_{k-1} \cdot (-x^2)/(2k(2k+1))$ для $k = 1, 2, \ldots$ при очевидном начальном условии $b_0 = x$. Программа решения этой задачи может иметь следующий вид:

```
program sinser
   real k, b, s, x/0.5/, y
   y=-x**2/2
   b=x; s=b; k=1.0
   do while (abs(b)>1e-6)
      b=b*y/(k*(2*k+1.0))
      s=s+b; k=k+1.0
   end do
   print *, s
end program sinser
```

Переменная **k** во избежание избыточных преобразований типа выбрана действительной — она используется не для индексирования, а как операнд в арифметических вычислениях. Обратите внимание на вспомогательную переменную **y**, вынесение расчета которой из цикла уменьшает трудоемкость последнего ("чистка цикла").

Пересчет членов ряда можно делать не только от предыдущего к текущему, но и от текущего к последующему. Правило пересчета и порядок следования операторов в теле цикла влияют на задаваемые перед входом в цикл начальные значения переменных. Выбор способа пересчета и порядка операторов для формирования устойчивой техники программирования рекомендуется сделать раз и навсегда. Задание начальных значений переменных удобнее программировать *хронологически после* тела цикла. Например, при пересчете от b_k к b_{k+1} следует задать начальное $k = 0$.

Аналогично программируются задачи накопления бесконечных произведений; здесь тело цикла выполняется, пока модуль *разности между очередным сомножителем и единицей* превышает заданный допуск.

Часто встречаются задачи на метод последовательных приближений (итераций). Итерации обычно продолжаются, пока модуль разности между двумя последовательными результатами превышает допуск (это не всегда правильно, но данный вопрос является скорее математической, чем программистской проблемой, и здесь не место входить в более детальное его обсуждение). Применим этот метод к решению уравнения $x = \cos(x)$ при начальном приближении $x_0 = 0,75$:

```
program iter
   real x0,x1,x2
   x0=0.75; x2=x0
   do while (abs(x2-x1)>1e-6)
      x1=x2; x2=cos(x1)
   end do
   print *, x2
end program iter
```

Полезно задуматься над порядком и смыслом присваиваний в этой программе. Что нужно изменить дополнительно, если переставить операторы в теле цикла?

Принципиальной особенностью данной задачи является зависимость нового приближения только от предыдущего. Это позволяет, организовав их циркуляцию, хранить только два результата (предыдущий и очередной). Разумеется, при необходимости пошагового представления хода вычислений можно организовать достаточно большой массив и последовательно записывать в него все результаты или выводить результаты внутри цикла, что резко замедлит работу программы.

В заключение приведем комплексный пример использования цикла данного вида в сочетании с условными конструкциями и назначаемым **go to**. Пусть необходимо найти единственный корень многочлена $x^3 - 20x^2 + 69x + 90 = 0$ на интервале $[0, 12]$ методом половинного деления. Эту задачу решает программа

```
program bisec
   real a/0/, b/12/, x,xa,xb,y,ya,yb,yc
   integer i
   x=a
   assign 10 to i
   go to  50
10 ya=y
```

```
      assign 20 to i
      go to  50
 20   ya=y
      xa=a; xb=b;
      do while (xb-xa>1e-3)
         c=(xa+xb)/2
         assign 30 to i
         go to  50
 30      yc=y
         if (ya*yc<0) then
            xb=xc; yb=yc
         else
            xa=xc; ya=yc
         end if
      end do
      print *, 'корень равен ', xc
 50   y=((x-20)*x+69)*x+90
      go to i, (10,20,30)
   end program bisect
```

2.2.3. Вложенные циклы

Программы решения многих задач требуют нескольких циклов. Приведем примеры таких ситуаций:

- упорядочение массивов,
- табулирование многочлена для ряда значений аргумента,
- расчет таблицы значений функции, заданной степенным рядом,
- решение методом итераций уравнения вида $x = P_n(x)$, где многочлен $P_n(x)$ вычисляется по Горнеру,
- вычисление произведения матриц,
- расчет среднего $\bar{x}$ и дисперсии D результатов измерений:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \qquad D = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2.$$

В подобных случаях очень важно правильно определить структуру будущей программы — прежде всего количество и относительное расположение циклов (последнее задание в простейшем варианте порождает два *последовательных* цикла). Только после этого имеет смысл начинать собственно программирование.

Программирование вложенных циклов любой кратности на языках высокого уровня не требует специальных средств программирования. В качестве примера приведем фрагмент программы с перемножением матриц $c = a * b$:

```
   ................................
   integer   i, j, k
   real      a, b, c, s
```

```
dimension a(4,6), b(6,5), c(4,5)
.................................
do i=1,4
   do j=1,5
      s=0.0
      do k=1,6
         s=s+a(i,k)*b(k,j)
      end do
      c(i,j)=s
   end do
end do
.................................
```

Анализ данного примера позволяет сформулировать две рекомендации по программированию вложенных циклов:

- внутренние циклы необходимо приводить в исходное состояние — это касается операций накопления разного вида — *непосредственно* перед его началом (обратите внимание на положение оператора обнуления `s`);
- внутренние циклы, определяющие основную трудоемкость выполнения программы, нужно строить предельно экономно, вынося из них вверх или вниз по программе повторяющиеся вычисления и операции индексирования[1].

Вторая рекомендация в примере реализована введением промежуточной переменной `s` вместо работы непосредственно с элементом `c(i,j)`.

Вложенность циклов для наглядности нужно указывать отступами. При глубокой вложенности можно делать заголовок и конец цикла именованными. Полезно также построение циклов с числовыми метками типа

```
   do 10 i=1,30
      s=s+x(i)
10 end do
```

Тело цикла можно ограничить и другим помеченным выполняемым оператором. На вид заключительного оператора налагаются некоторые ограничения; поэтому цикл с числовой меткой рекомендуется заканчивать помеченным оператором продолжения `continue`, не выполняющим никаких действий.

2.2.4. Дополнительные средства

Досрочный (по дополнительному условию) выход из цикла на следующий за `end do` оператор возможен с помощью оператора

```
exit [<имя>]
```

При заданном имени выход производится из помеченного этим именем цикла. По умолчанию выход будет только из самого внутреннего, непосредственно охватывающего `exit`. Аналогично оператор

```
cycle [<имя>]
```

выводит на управляющую конструкцию поименованного цикла.

---

[1] Эти функции можно переложить на оптимизирующий компилятор.

Цикл `do while` считается устаревшей конструкцией; планируется его замена "глухим" циклом

```
do
   if <ЛВ> exit
   <блок>
end do
```

с возможной перестановкой условного оператора и блока.

Глава 3.

Массивы

3.1. Роль массивов

Массивом называется упорядоченное множество однотипных значений (список, проекции вектора, таблица результатов эксперимента, матрица преобразования координат). Элементы массивов обычно обрабатываются некоторым стандартным образом. Ссылка на элемент массива состоит из общего имени массива и (в круглых скобках) набора индексов, характеризующих положение в нем избранного элемента: `a(i,j)` означает элемент матрицы `a` на пересечении `i`-й строки и `j`-го столбца. Тем самым отпадает надобность в сотнях различных имен для родственных объектов программы. Два массива считаются равными, если равны их элементы, *стоящие в одинаковых позициях.*

Индексы сами могут быть элементами массивов (такая возможность в предшествующих версиях Фортрана отсутствовала).

Матрично-векторные обозначения являются основой языка современной математики. Циклическое описание покомпонентных операций всегда было важным элементом практического программирования, и в ряд языков программирования (Альфа, ПЛ/1) еще в 60-е гг. были введены средства их компактного задания. Сейчас в связи с массовым распространением векторных процессоров и других видов параллельно работающих устройств суперЭВМ настало время введения в языки программирования более сложных операций над массивами.

3.2. Описание массивов

Массивы характеризуются типом значений их элементов, *рангом* — числом измерений[1] (не более семи) и граничными парами — диапазоном индексов по каждому измерению. Границы каждого диапазона разделяются двоеточием; нижняя граница +1 при описании может быть опущена. Приведем несколько примеров описания массивов:

```
real        a,b,c
integer     k
dimension   a(0:4), b(3,7), c(3,7), k(6)
```

[1] Не путать с рангом матрицы!

Второй вариант того же описания указывает для каждого объекта полный набор атрибутов:

```
integer, parameter       :: n=3
real,    dimension          a(0:n+1)
real,    dimension (3,7) :: b,c
integer, dimension          k(6)
```

Здесь для задания границы массива a применено *константное* выражение, которое может быть вычислено на этапе компиляции. Напомним, что граничные пары могут быть указаны и непосредственно в описании типа.

Из рассмотренных примеров следует, что атрибуты могут задаваться как порознь, так и вместе, причем объекты с общими атрибутами можно группировать в единый список. Второй вариант как более устойчивый к ошибкам считается предпочтительным.

Протяженность массива по некоторому измерению называется его *экстентом*, а упорядоченный полный набор экстентов — *формой* массива. Массивы одинаковой формы считаются *конформными*. Подчеркнем, что для конформности массивов совпадение граничных пар не требуется. Скаляр считается конформным с любым массивом. Произведение экстентов задает *размер* массива.

В FPS массив может иметь нулевую длину по одному или нескольким измерениям и, следовательно, нулевой размер. Никакие вычисления с ним реально не выполняются. Тем не менее, он должен быть по своей форме совместим с остальными операндами выражения или левой частью оператора присваивания. Скаляр *формально* не является массивом с нулевым рангом и единичным экстентом и не может подставляться вместо такового.

Элементы массивов располагаются в памяти таким образом, что первым меняется старший (самый левый) индекс. В частности, матрицы размещаются *по столбцам*. Это обстоятельство следует учитывать при организации ввода и вывода многомерных массивов.

3.3. Вырезки и сечения массивов

Для ссылки на элемент массива задается индексированная переменная, на массив в целом ссылаются по его имени. В FPS есть возможность непосредственно работать с частями массива — вырезками и сечениями. *Вырезка* из массива представляет собой подмассив вида

`<имя_массива>(<диапазон>,...,<диапазон>)`

Элементы вырезки из массива могут быть взяты с шагом, отличным от единицы. В этом случае вырезка по соответствующему измерению задается *триплетом*. Так, `a(i,2:8:2)` — это четные элементы `i`-й строки матрицы `a`. Шаг может быть и отрицательным; тогда второй элемент триплета должен быть меньше первого, и элементы исходного массива будут выбираться в обратной последовательности. Опущенные нижние или верхние границы вырезки заимствуются у массива в целом. Если по какому-то измерению опущены обе границы вырезки (двоеточие должно быть сохранено), то говорят о *сечении* массива: `a(i,:)` означает `i`-ю строку упомянутой матрицы `a`. Понятие конформности массивов естественно переносится и на вырезки.

Вырезку можно задать также с помощью *векторного индекса*. Конструкция вида `x((/2,4,8,1/))` означает вектор, составленный из четырех компонент вектора

x в указанной последовательности. Номера извлекаемых компонент должны находиться в отведенных описанием границах и могут повторяться. При использовании векторного индекса в *левой* части оператора присваивания и в списке оператора ввода повторения исключены — иначе пришлось бы присвоить одному и тому же элементу несколько разных значений.

3.4. Задание массивов

Для означивания массивов существуют следующие средства:

- описание начальных значений,
- оператор ввода,
- оператор присваивания.

Приведем примеры использования этих средств (кроме ввода, обсуждаемого отдельно). Переменные и массивы можно инициализировать прямо в описании типа:

```
real                      :: z=0.0
integer, dimension(3) :: m /2,5,11/
integer, dimension(4) :: k=(/1,3,5,7/)
```

(запись в правой части последнего описания называется *конструктором массива*). В тех же целях можно использовать оператор

`data <список_объектов>/<список_значений>/`

где `<список_объектов>` включает переменные и неявные циклы. При означивании происходит поэлементное сопоставление обоих списков с учетом структуры объектов (массивов) и повторителей вида `4*` в списке значений:

```
integer i,j
integer, parameter :: l=4, m=l*(l+1)/2
real q(l,l)
data ((q(i,j), j=1,i), i=1,l) /m*1.0/
```

Здесь заполняется вещественными единицами нижняя треугольная часть матрицы `q` размером `l*l`. Обратите внимание на использование в этом примере именованных констант и константных выражений! Отметим дополнительно, что

- циклический список заключается в скобки,
- циклический список может быть вложенным,
- во всех циклах можно дополнительно указать после правой границы шаг приращения параметра (по умолчанию берется $+1$),
- каждая переменная или часть ее в программе могут инициализироваться только один раз,
- при различии в типах элементов массива и задаваемых значений происходит автоматическое преобразование последних (как в операторах присваивания).

Пар списков (как и самих операторов `data`) может быть несколько. В нижеследующем примере

```
real, dimension s(24)
data  s/24*2.0/
..................................
s(1:12)=(/(2*i, i=1,10), 12.0,15.0/)
```

оператор `data` выполняет первоначальное заполнение массива `s` двойками. Далее вырезке из этого массива уже в выполняемой части программы присваивается значение *конструктора массива*. Последний состоит из определяющего ряд 2, 4,. . . ,20 циклического списка и двух отдельных значений.

Конструктор позволяет задать только одномерный массив. Однако с помощью встроенной функции `reshape` его легко преобразовать в многомерный:

```
d=reshape((/(2*i, i=1,10), 12.0,15.0/), (/3,4/))
```

Здесь такими же элементами заполняется (по столбцам!) массив `d(3,4)`. Сразу же отметим возможность применения `reshape` для других преобразований — например, матрицы в вектор (при этом первым аргументом указывается матрица, а второй может быть опущен).

Использование вырезок позволяет упростить формирование матриц со стандартными фрагментами. Пусть

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 3 & 4 & 2 \\ 6 & 1 & 5 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 2 & 1 & 1 & 0 & 0 \\ 3 & 4 & 2 & 0 & 1 & 0 \\ 6 & 1 & 5 & 0 & 0 & 1 \end{bmatrix}.$$

Если `ID` — единичная матрица размером 3×3, то `B` можно сформировать двумя операторами присваивания

```
b(1:3,1:3)=a
b(1:3,4:6)=id
```

3.5. Поэлементные операции

В FPS возможно компактное описание поэлементных операций над конформными массивами и вырезками. Во фрагменте

```
real a(6), c(6), d(8)
a=7.0
d=6.67
c=a+0.5*d(3:8)
a=exp(c)
```

первые два присваивания разрешены из-за конформности скаляра и произвольного массива; третье допустимо благодаря выбранной протяженности вырезки и поэлементному характеру сложения векторов; четвертое определяет поэлементное вычисление показательной функции.

3.6. Выборочные действия

Конструкция `where` позволяет запрограммировать действия с некоторыми элементами массива, отбираемыми в соответствии с логическим массивом той же формы (*маской*). Так, во фрагменте

```
where (a<0) a=0
```

все отрицательные элементы матрицы a будут заменены нулями. Заметим, что все компоненты этой и родственных ей конструкций являются матричными. Наиболее общим является оператор

```
where (<ЛВ-массив>)
     <присваивания_массивов>
elsewhere
     <присваивания_массивов>
end where
```

(блок `elsewhere` необязателен). Операторов присваивания в каждом блоке может быть несколько, но все они должны относиться к массивам одинаковой формы, совпадающей с формой маски. Логическое выражение-массив может порождаться покомпонентным выполнением операции отношения или произвольно формируемой логической маской:

```
real x(100)
integer sparse (100)
logical, dimension(100) :: mask
................................
mask(1:100:2)=.true.
where(mask)
   x=x+1.0
   sparse=1
end where
```

В этом примере каждый нечетный элемент из `x` будет увеличен на единицу, а из `sparse` получит единичное значение.

3.7. Встроенные функции для векторов и матриц

3.7.1. Справочные функции

Эти функции позволяют получить информацию о структуре массива и его измерениях. Такая информация используется в процедурах, параметрами которых служат массивы переменного размера. Функция `lbound(a[,dim])` возвращает массив нижних границ `A` (при вызове `lbound(a,2)` — границы по второму измерению). Аналогичные результаты для верхних границ дает `hbound`. Функция `shape(a)` выдает вектор экстентов массива-аргумента, `size(a[,dim])` — экстент по заданному измерению (при опущенном номере измерения — полное число элементов).

К группе работающих с массивами справочных функций мы отнесем и необходимую для расчета времени выполнения участка программы процедуру `date_and_time`. Ее ответный ключевой параметр `values` предполагает массив из восьми целых чисел, четырьмя последними компонентами которого являются часы (h), минуты (m), секунды (s), тысячные доли секунды (t). Поэтому фрагмент

```
.................
integer c(8)
integer h,m,s,t, dh,dm,ds,dt
real time
```

```
call date_and_time(values=c)
h=c(5); m=c(6); s=c(7); t=c(8)
< измеряемый участок>
call date_and_time(values=c)
dh=c(5)-h; dm=c(6)-m; ds=c(7)-s; dt=c(8)-t
time=(dh*60+dm)*60+ds+0.001*dt
```

обеспечит определение времени прогона участка в секундах[2].

3.7.2. Преобразование массивов

Функция `transpose` выполняет транспонирование двумерного массива. Уже использованная нами функция `reshape(<вход>,<форма>)` преобразует входной массив к форме с заданными постоянными экстентами. Преобразование производится с сохранением стандартного порядка следования элементов. Если

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 3 & 4 & 2 \end{bmatrix},$$

то оператор

```
b=reshape(a, shape=(/3,2/)
```

преобразует эту матрицу в

$$B = \begin{bmatrix} 1 & 4 \\ 3 & 1 \\ 2 & 2 \end{bmatrix}.$$

Эта функция часто используется для преобразования вектора, заданного конструктором массива, в массив требуемой формы.

Преобразующая функция `pack(a,mask,v)` упаковывает отобранные по маске элементы `a` (в порядке их размещения в памяти) в вектор `v`. Предполагается, что длина `v` будет достаточной.

Функция `spread(a,dim,n)` возвращает массив с рангом, на единицу большим `a`, пристраивая `n-1` копию исходного массива к `a` по указанному вторым параметром измерению `dim`. Пусть $a = [1\ 2\ 3]$. Тогда результатом выполнения

```
b=spread(a,1,4); c=spread(a,2,3)
```

будут матрицы

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}, \qquad C = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}.$$

Функция `merge` производит управляемое слияние конформных массивов по заданному критерию. Например, оператор

```
c = merge(a, b, a<b)
```

формирует матрицу с элементами $c_{i,j} = \min\{a_{i,j}, b_{i,j}\}$.

---

[2] При условии, что весь прогон реализуется в течение одних календарных суток.

3.7.3. Неэлементные операции

Функция dot_product(a,b) обеспечивает скалярное перемножение векторов a и b. При комплексном a для b берутся комплексно сопряженные значения. Функция matmul(a,b) выполняет матричное умножение операндов. В случае логических операндов при вычислении обеих этих функций арифметические умножение и сложение заменяются одноименными логическими операциями.

Обозначим через mask логический массив той же формы, что исходная матрица (например, результат покомпонентной проверки положительности элементов). Функции maxval, minval, product и sum имеют очевидный числовой смысл. Например, конструкция product(a, mask=a/=0) подсчитает произведение ненулевых элементов матрицы a. Функции maxloc и minloc возвращают массивы координат экстремальных значений (при наличии нескольких таких значений — первого из них). Все они могут иметь необязательным вторым параметром маску. Функции all, any возвращают соответственно значение истинности логического произведения и логической суммы mask, а count — количество истинных элементов.

Приведем несколько примеров на использование неэлементных функций от массивов. Для подсчета произведения элементов матрицы z, превышающих по модулю единицу, следует воспользоваться функцией

```
product (z, mask=abs(z)>1.0)
```

Здесь использование маски обязательно. Если необходимо вычислить

$$\prod_{i=1}^{m}\sum_{j=1}^{n} a_{i,j},$$

это может быть сделано вызовом

```
product (sum (a, dim=2))
```

(суммирование выполняется по второму измерению — номерам столбцов).

Наконец для накопления суммы парных произведений строки первой матрицы на столбец второй (внутренний цикл задачи о перемножении матриц) можно воспользоваться конструкцией

```
c(i,j)=sum(a(i,:)*b(:,j))
```

Покажем комплексное использование встроенных средств обработки массивов. Пусть дана матрица игры с нулевой суммой

$$A = \begin{bmatrix} 4 & 5 & 9 \\ 7 & 6 & 8 \\ 8 & 4 & 3 \end{bmatrix}$$

и необходимо найти ее канонические нормы, верхнюю и нижнюю цены игры w и v соответственно и координаты седлового элемента. Последний по определению одновременно является минимумом в своей строке и максимумом — в столбце. Задача решается программой

```
program matrix
   real a,lnorm,mnorm,se,sc,sr,v,w
   integer i
   dimension a(3,3),i0(1),j0(1),sc(3),sr(3)
   data a/4,7,8,5,6,4,9,8,3/
   ! Считаем нормы матрицы
   do i=1,3
```

```
      sr(i)=sum(abs(a(i,:)))  ! Суммы модулей строк
    end do
    mnorm=maxval(sr)            ! Максимум из них
    print *, 'mnorm = ',mnorm   ! 21
    sc=sum(abs(a),1)            ! Аналогично по столбцам,&
                                ! но обошлись без цикла
    lnorm=maxval(sc)            ! Максимум из них
    print *, 'lnorm = ',lnorm   ! 20
    se=sqrt(sum(a*a))           ! эвклидова норма
    print *, 'enorm = ',se      ! 18.973670
    ! Ищем седловую точку
    v=maxval(minval(a,2))
    w=minval(maxval(a,1))
    i0=maxloc(minval(a,2))
    j0=minloc(maxval(a,1))
    print *, 'Нижняя цена игры = ',v        ! 6.0
    print *, 'Верхняя цена игры = ',w       ! 6.0
    print *, 'Координаты седлового элемента'
    print *, 'i0 = ',i0,' j0 = ',j0         ! i0 = 2, j0 = 2
 end program matrix
```

В дополнение к комментариям, включенным в текст программы, отметим принципиальную необходимость объявления `i0` и `j0` как массивов для совместимости левой и правой частей операторов присваивания им ответных значений.

3.8. Динамическая память

Размер массива часто определяется после некоторых вычислений. Именно тогда (т.е. строго по фактической потребности) и следует выделить память для него. Предварительно массив должен быть объявлен как *размещаемый* — с атрибутом `allocatable`, указанием его ранга и атрибутов элементов, например

```
real, allocatable  :: a(:,:), b(:)
```

Заметим, что в описании `dimension` указывается (двоеточиями) только *количество граничных пар.* После определения значений всех входящих в границы переменных оператором вида

```
allocate (a(n,0:n+2))
```

производится собственно выделение памяти (список создаваемых объектов заключается в скобки, даже если объект один). Выражения для граничных пар должны быть целого типа. В операторе `allocate` может быть указан дополнительный параметр `stat=<имя_скалярной_переменной>`. При успешном выделении памяти эта переменная получает нулевое значение, иначе — положительное. Если `stat=` отсутствует, а память выделить не удалось, выполнение программы прекращается. Повторное выполнение `allocate` для уже размещенного массива фиксируется как ошибка фазы счета.

По миновании надобности в массивах занятая ими память освобождается аналогичным оператором

```
deallocate (<список_массивов>)
```

где граничные пары уже не указываются. Этот оператор нужно использовать также в случае переопределения границ массива. Выполнение `deallocate` для не размещенного массива приводит к непредсказуемым последствиям.

Размещаемые массивы нельзя инициализировать оператором `data`.

Глава 4.

Обработка строк

4.1. Строки и массивы строк

Первоначальное понятие о строках как текстовых константах было введено в главе 1. Строка, хотя бы и состоящая из нескольких символов, рассматривается как скаляр (скаляр–число тоже состоит из нескольких цифр!). В то же время имеется доступ к ее частям. Подстрока задается аналогично сечению массива (диапазоном номеров позиций — через двоеточие, начальная позиция строки всегда первая). Границы подстроки, совпадающие с границами строки, можно не указывать, однако двоеточия обязательны. Для выбора отдельного символа его позицию повторяют как нижнюю и верхнюю границы подстроки.

Массивы строк могут быть составлены только из строк одинаковой длины и используются аналогично числовым массивам (разумеется, к ним применяются другие операции). Примеры описаний:

```
character (len=5)                :: rank
character (len=6), dimension(30) :: list
```

Длина строки в главной программе должна задаваться константным выражением.

При ссылке на подстроку *массива строк* нужно сначала указать имя массива и индекс, а затем границы подстроки, например

`list(18)(3:)`

(элементы с 3-го до последнего из 18-й строки массива `list`).

Единственная встроенная операция для *строк* — это склеивание (конкатенация), обозначаемая //. Строки, формируемые как продукт конкатенации (`'KINDER'//'GARTEN'='KINDERGARTEN'`), должны быть объявлены в расчете на максимальную длину. При различии в длинах левой и правой частей оператора присваивания правая часть усекается или дополняется пробелами *справа*.

4.2. Функции от строк

Из функций от строк мы рассмотрим только три:

`iachar(C)` — номер символа `C` в сортирующей последовательности `ASCII` (заметим необходимое для перевода цифр из символьной в числовую форму равенство `iachar(0)=48`);

`char(n)` — символ, отвечающий номеру `n` в кодировке `ASCII`;

`index(str,sub)` — позиция, начиная с которой подстрока `sub` входит в строку `str`.

Как пример работы со строками опишем перевод числа в символьное представление:

```
program numch
   integer              :: number=3946790
   character*7 string
   call numchar(number,7,string)
   print *, string
end program numch

subroutine numchar(n,m,c)
   integer              :: n ! входное число
   integer              :: m ! количество цифр
   character(len=*)     :: c ! выходная строка
   integer              :: i,j,k,l
   character*1          :: c1
   k=n                  ! исходное числовое значение
   do i=m,1,-1          ! позиция
      l=k/10            ! целое частное
      j=k-10*l          ! очередная цифра =
                        ! остатку от деления
      k=l               ! новое делимое
      c1=char(j+48)     ! перевод цифры в символ
      c(i:i)=c1         ! включение в ответную строку
   end do
end subroutine numchar
```

Глава 5.

Программные компоненты

5.1. Программа и ее компоненты

Программа на Ф90 представляет собой определенным образом оформленную совокупность операторов. Она может быть монолитной или составной.

Минимальный состав законченной программы — только *головная программа*. Однако любая достаточно сложная задача требует ее рассмотрения на нескольких уровнях детальности, что определяет иерархическое построение программы из нескольких субпрограмм (процедур). Обычно выясняется, что некоторые из частных задач относятся к стандартному арсеналу математика или типичны для рассматриваемого круга приложений — следовательно, уже были или должны быть запрограммированы для многократного применения. Использование процедур делает программу в целом компактной и обозримой, уменьшает трудоемкость разработки и в особенности отладки, позволяет распараллелить разработку между несколькими исполнителями, облегчает внесение исправлений в сложный алгоритм с однотипными фрагментами.

Процедуры могут быть внутренними, внешними и модульными. Описание *внутренней* процедуры непосредственно включается в текст охватывающей программной единицы (носителя) и не может содержать вложенных процедур. *Внешняя* процедура существует автономно и может быть разработана на других языках (C и его версии, ассемблер).

Программные единицы бывают следующих видов: главная программа, внешняя процедура, модуль и блок данных. Каждая из них

- физически отделена от других;
- начинается оператором-заголовком, содержащим специфическое для данной программной единицы ключевое слово: `program`, `subroutine`, `function`, `module`, `block data`;
- заканчивается предложением `end` с повторением (иногда не обязательным, но рекомендуемым) того же ключевого слова и собственного имени программной единицы;
- обрабатывается компилятором отдельно от остальных.

Любая программная единица может содержать неограниченное число предложений вида

```
include '<имя_файла>'
```

(при необходимости — с указанием пути к файлу). Стандартные директории для поиска вставок задаются настройкой MDS. Вставленный текст далее обрабатывается компилятором как составная часть программной единицы.

Выполняемая программа состоит из головной программы и произвольного числа остальных программных единиц. Частью головной программы могут быть внутренние процедуры. Обобщенный вариант структуры программы показан на рис. 5.1.

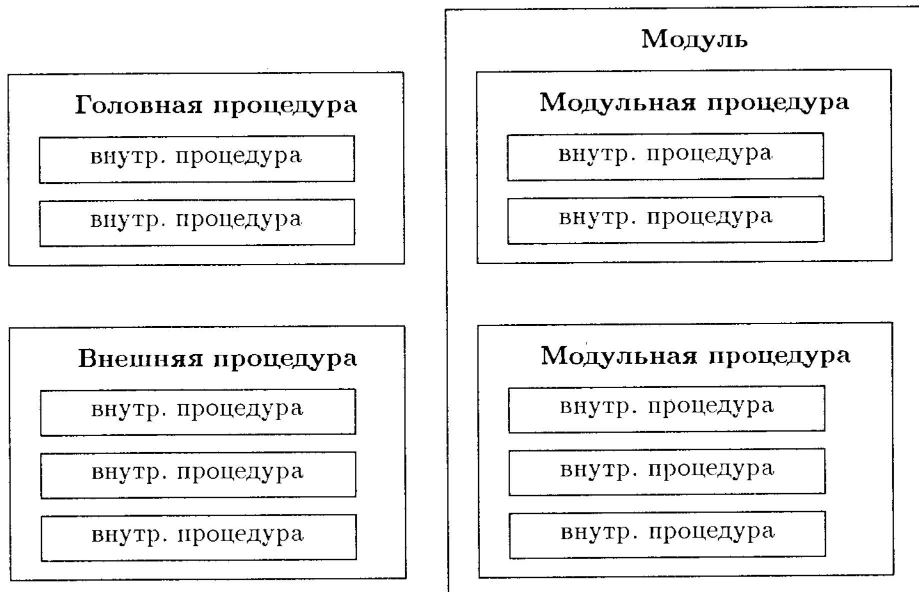

Рис. 5.1. Структура программы

Вложенные процедуры любого вида имеют доступ ко всем объектам своего носителя.

Модуль может содержать несколько модульных процедур, при необходимости включающих в себя внутренние. Кроме того, в нем могут содержаться описания данных, определения производных типов, интерфейсные блоки, группы **namelist**. Обычно модули создают специализированную среду и инструментарий для решения некоторого класса задач (работа с упакованными матрицами и матрицами специального вида, интервальная арифметика, алгебра нечетких множеств).

Блоки данных содержат только описательные операторы. Они используются для инициализации переменных в именованных общих блоках.

Головная программа имеет следующий вид:

```
[program <имя_программы>]
   [<операторы_описания>]
   [<исполняемые_операторы>]
[contains
      <внутренние_процедуры>]
 end [program [<имя_программы>]]
```

Обязательный оператор **end** ограничивает текст программной компоненты и завершает ее выполнение. Прекращение выполнения программы задается также оператором **stop**. Последний может сопровождаться *кодом останова* — текстовой константой или последовательностью цифр (до 5 знаков), которые выводятся при срабатывании данного останова. Этот прием полезен при отладке и обработке аварийных ситуаций (например, некорректных исходных данных).

Внешняя процедура отличается от головной программы только заголовком и ключевым словом в завершающей строке. Она приводится в действие оператором **call** из вызывающей программной единицы или указателем функции (для функций). Имена внешних процедур и формальные параметры-процедуры должны быть специфицированы описателем в вызывающей процедуре как **external** либо **intrinsic**.

Внутренние процедуры выглядят так же, как модульные, но не могут иметь вложенных в них процедур.

После загрузки всего программного комплекса в оперативную память управление передается головной программе.

5.2. Подпрограммы

Заголовок процедуры-подпрограммы имеет вид

SUBROUTINE <имя_процедуры>(<список_формальных_параметров>)

Формальные параметры вводят обозначения для описания стандартных действий, выполняемых процедурой. *Вызов* подпрограммы производится оператором

call <имя_подпрограммы>(<список_аргументов>)

и отрабатывается в первом приближении как ее тело (описание) с заменой формальных параметров фактическими — аргументами. Отсюда вытекает требование соответствия аргументов параметрам по количеству, типу и смыслу. Из этого правила имеются исключения, связанные с обсуждаемыми ниже необязательными параметрами и с параметрами-массивами. Совпадение имен не обязательно, но допустимо.

Характер формальных параметров определяет их представление и обработку на машинном уровне; следовательно, они должны быть специфицированы (объявлены) внутри процедуры (иначе их атрибуты будут назначены по умолчанию). Кроме того, для них может (иногда — должен) быть установлен вид связи: атрибут **intent** с указанием в скобках одного из вариантов **in, out, inout** (входной, выходной, переопределяемый). Аргумент, замещающий входной параметр, может быть выражением; тогда его значение вычисляется и фиксируется при входе в процедуру. Выходному и переопределяемому параметрам может соответствовать только переменная (в частности, массив).

Пример. Пусть необходимо вычислить

$$y = \frac{f(x) + 2f(2x)}{x - f^2(x + 0,5)},$$

где $f(u) = \sin u - e^{-u}$ и $x = 0,2$. Эту задачу решает программа

```
program  pp
   real y,y1,y2,y3, x/0.2/
   call p(x,y1); call p(2.0*x,y2); call p(x+0.5,y3)
   y=(y1+2.0*y2)/(x-y3**2)
   print *, 'y = ',y
end program pp
```

```
subroutine p(a,z)
   real a,z
   z=sin(a)-exp(-a)
end subroutine p
```

Иногда в процессе обращений к процедуре накапливается некоторый "опыт", который может быть использован при последующих обращениях (например, результат итераций может быть принят за начальное приближение при очередном вызове). Атрибут **save** сохраняет значения имеющих его переменных (исключая массивы с переменными границами) для следующего входа в процедуру. Таким образом можно обеспечить подсчет количества обращений к процедуре, суммарное время ее наработки, накапливать результаты обработки. Пример его использования приводится в разделе о рекурсивных процедурах. *Оператор* **save** позволяет придать это свойство нескольким объектам, перечисленным в его списке.

5.3. Функции

Обращение к *функции* может осуществляться непосредственно из выражения, и возвращаемое значение используется в том же выражении. Это значение есть единственный результат работы функции (возможно, структурированный, т.е. массив). Он сопоставляется имени функции; поэтому в заголовке функции обычно отсутствует ответный параметр, а в спецификации для имени функции указываются атрибуты результата:

```
integer function sumints(x)
```

(при отсутствии такого описания в заголовке или внутренних операторах объявления атрибутов тип результата будет назначен по умолчанию в соответствии с первой буквой имени функции). Вычисление функции должно заканчиваться присваиванием ее имени ответного значения, после которого следует завершающий

```
end function[<имя>]
```

Имя функции должно быть объявлено в вызывающей процедуре с атрибутами возвращаемого значения.

Указатель функции, вызываемой с пустым списком аргументов, тем не менее должен содержать скобки.

При использовании внешней процедуры-функции программа поставленной на стр. 44 задачи может иметь вид

```
program pf
   real f,y,x/.2/
   y=(f(x)+2*f(2*x))/(x-f(x+.5)**2)
   print * ,' y = ',y
end program pf

function f(a)
   real a
   f=sin(a)-exp(-a)
end function f
```

Обращение к функции не должно переопределять переменных, используемых в одном операторе с ее вызовом — в частности, ее аргументов (требование отсутствия побочного эффекта). Если, например, функция **f(x)**, кроме ответного

значения, будет менять еще и свой аргумент, то нарушится естественное равенство `f(x)+x` и `x+f(x)`, а произведение `f(x)*f(x)` не будет равно `f(x)**2`.

5.4. Расположение операторов

Приведем сводную таблицу относительного расположения операторов программной единицы (табл. 5.1). Относительный порядок конструкций из вертикальных блоков таблицы, имеющих общее горизонтальное сечение, произволен. Все операторы описания данных должны предшествовать выполняемым операторам. Операторы, разделенные горизонтальными линиями, перемежаться не могут.

Таблица 5.1. **Относительное расположение операторов**

<table>
<tr><td colspan="3">Операторы PROGRAM, FUNCTION, SUBROUTINE или MODULE</td></tr>
<tr><td colspan="3">Операторы USE</td></tr>
<tr><td rowspan="4">Операторы FORMAT</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>Операторы PARAMETER</td><td>Операторы IMPLICIT</td></tr>
<tr><td>Операторы PARAMETER и DATA</td><td>Определения производных типов, интерфейсные блоки, операторы описания типа и другие операторы описания</td></tr>
<tr><td colspan="2">Исполняемые операторы</td></tr>
<tr><td colspan="3">Оператор CONTAINS</td></tr>
<tr><td colspan="3">Внутренние или модульные процедуры</td></tr>
<tr><td colspan="3">Оператор END</td></tr>
</table>

Комментарии допустимы в любом месте программы. Любые ссылки на именованные константы могут размещаться только после соответствующего оператора `parameter`. Если программная единица содержит метакоманды глобального управления трансляцией `$integer`, `$real`, `$optimize`, они должны предшествовать ее заголовку; остальные метакоманды размещаются произвольно.

5.5. Области видимости меток и имен

Видимыми называются объекты, на которые можно ссылаться из данного места программы. Имена программных компонент и внешних процедур глобальны, т.е. доступны из любого места программы. Каждое из них должно отличаться от всех остальных, а также от имен локальных объектов.

Каждое имя видимо из своего непосредственного носителя, исключая те вложенные конструкции (процедуры, интерфейсные блоки и декларации производных типов), где оно переопределяется. Аналогично обстоит дело с использованием числовых меток. Любой оператор перехода возможен только на видимую из данной точки метку. Напомним, что переходы внутрь блоков, минующие их начало, потенциально опасны, и компилятор при обнаружении таких ситуаций выдает предупреждения.

5.6. Внутренние процедуры

Внутренние процедуры (как подпрограммы, так и функции) могут быть описаны в конце главной программы, внешней или модульной процедуры после ключевого слова contains. Обычно так оформляют частные алгоритмы, особенно тесно связанные с конкретной задачей. Внутренняя процедура работает в среде своего носителя и может быть вызвана только из него или другой внутренней процедуры того же носителя. Она может ссылаться на локальные имена объектов своего носителя, что позволяет передавать ей информацию помимо параметров. Это уменьшает требуемое число параметров и упрощает ее вызов. В предельном случае возможно использование внутренних процедур без параметров.

При использовании внутренней процедуры-функции обсуждавшаяся выше задача оформляется следующим образом:

```
program pf
   real y,x/0.2/
   y=(f(x)+2.0*f(2.0*x))/(x-f(x+0.5)**2)
   print *, 'y = ',y
contains
   function f(a)
      real a
      f=sin(a)-exp(-a)
   end function f
end program pf
```

Разумеется, она может быть решена и с помощью внутренней подпрограммы.

Внутренние процедуры не могут содержать вложенных в них других внутренних процедур. Они не могут быть фактическими параметрами обращений к другим процедурам. Это обстоятельство не позволяет, например, применить стандартную процедуру численного интегрирования для подинтегральной функции, часть объектов которой определена в ее носителе.

5.7. Интерфейс процедур

Интерфейсом процедуры называется минимально необходимая для ее вызова совокупность сведений о ней: подпрограмма это или функция, каковы структура списка параметров и атрибуты последних. Интерфейс внутренней процедуры, а также модульной процедуры из модуля, присоединенного к вызывающей программной единице оператором use, непосредственно доступен компилятору и потому считается явным. Для успешной работы с *внешними* процедурами их имена должны быть перечислены в операторе external вызывающей программной единицы перед первым исполняемым оператором. Интерфейс для каждой из них должен быть задан явно блоком

```
interface
   <тело_интерфейса>
end interface
```

Тело интерфейса — это функционально эквивалентная копия заголовка процедуры, спецификации параметров и завершающего оператора end с точностью до имен

параметров и размещения информации (допустима иная группировка атрибутов параметров).

Интерфейс может быть групповым. В этом случае семейству процедур дается общее имя, по которому в каждом конкретном случае вызывается одна из них. Процедуры семейства должны различаться хотя бы одним из следующих признаков:

- количество или имена необязательных параметров,
- атрибуты параметра,
- ранг (размерность) параметров-массивов.

Приведем пример группового интерфейса к процедурам определения эвклидова расстояния между двумя точками на плоскости в непрерывном случае и по модулям разностей одноименных координат на квадратной сетке — в дискретном:

```
interface dist
   real function dr(x,y)
      real,    dimension(2), intent(in) :: x,y
   end real function dr

   integer function di(x,y)
      integer, dimension(2), intent(in) :: x,y
   end integer function di
end interface dist
```

В любом случае функция вызывается как `dist(x,y)`.

Использование блока интерфейса обязательно, если внешняя процедура

- вызывается с ключевым или с отсутствующим аргументом;
- переопределяет оператор или присваивание при работе с производными типами данных;
- формирует ответный массив либо строку переменной длины;
- имеет параметром массив предполагаемой формы, указатель или цель;
- является формальным или фактическим параметром;
- вызывается по родовому имени.

Интерфейс или ссылка на содержащий его модуль должны быть помещены до первого вызова соответствующей процедуры.

5.8. Специальные виды параметров

5.8.1. Массивы как параметры

Этот раздел мы начнем с примера заголовка процедуры, специфицирующего разные виды формальных параметров-массивов:

```
subroutine up(n,p,r,s,t)
   real, dimension(1:15,5,10)       :: p ! с явн. границами
   real, dimension(n,n+2)           :: t ! автоматический
   real, dimension(1:15,n,*)        :: r ! подразумеваемого
!                                          размера
   real, dimension (:,5:)           :: s ! подразумеваемой
!                                          формы
```

Мы рекомендуем подобную форму записи заголовка процедуры (с дополнительным указанием в комментариях назначения процедуры, идеи или ссылки на описание алгоритма, *смысла* формальных параметров) для серьезного практического программирования.

Все переменные границы *автоматического* массива должны явно выражаться через другие формальные параметры процедуры.

Текстовый формальный параметр может быть описан со звездочкой вместо указания длины. Тогда он автоматически перенимает длину фактического параметра. Если внутри процедуры понадобится длина фактического параметра-строки, ее можно получить обращением к встроенной функции `len`.

Массив *подразумеваемой формы* заимствует ее у связанного с ним фактического параметра с возможным сдвигом границ индексов (в данном примере у массива `s` нижние границы индексов будут 1 и 5 соответственно независимо от нижних границ аргумента). Для уменьшения числа передаваемых параметров фактические границы аргументов-массивов можно определить обращением к встроенной функции `size(<имя_параметра_массива>,<dim>)`, где `<dim>` — номер интересующего нас измерения. По умолчанию нижняя граница равна 1. Подчеркнем, что передаются не границы массива, а его *форма* (экстенты). Если, например, фактический массив `w` описан как `real, dimension(0:10,0:12) :: w`, а формальный `fw` —посредством `real, dimension(:,:) :: fw`, то элементу `w(i,j)` будет сопоставлен `fw(i+1,j+1)`.

Можно, однако, потребовать, чтобы структура параметра полностью перенималась у аргумента. В этом случае его спецификация `dimension` по каждому измерению должна включать в себя нижнюю границу.

Массив *подразумеваемого размера* заимствует у фактического параметра только *число* элементов. Его верхняя граница по последнему измерению определяется с учетом заданных в описании экстентов по старшим измерениям. При этом может происходить преобразование формы — см. пример:

```
      ....................
      dimension frr(3,4,5)
      ....................
      call trp(arr)
      ....................
   contains
      subroutine trp(arr)
         real, dimension (15,0:*) :: arr
      ....................
```

Здесь исходный трехмерный массив будет обрабатываться в подпрограмме как матрица с $3 \times 4 \times 5/15 = 4$ столбцами и номерами столбцов `0:3`.

Работа с параметром данного вида исключает контроль диапазона индексов и тем понижает надежность программы.

Приведем несколько примеров работы с параметрами-массивами:

```
program parmatr
   integer, dimension(4,4) :: a
   integer b,i,j
   do i=1,4
      do j=1,4; a(i,j)=i+4*(j-1); end do
   end do
   write (*,'(1x,4i6)') ((a(i,j),j=1,4),i=1,4)
   print *, ' '
   b=sumv(4,a(3,:))
   print *, 'Для третьей строки b = ',b                    !  36
   b=sumarr(a(3,2:3),2)
   print *, 'Для вырезки из третьей строки b = ', b       !  18
   b=sumarr(a(2:3,2),2)
   print *, 'Для вырезки из второго столбца b = ', b      !  13
   b=sumarr(a,16)
   print *, 'Для всей матрицы b = ', b                    ! 136
   b=sumarr(a(:2,2:),6)
   print *, 'Для квадратной вырезки из a  b = ', b        !  57
contains
   integer function sumv(a)
      integer a(:)
      integer i,n,s
      n=size(a)
  s=0
  do i=1,n; s=s+a(i); end do
  sumv=s
   end function sumv

   integer function sumarr(a,k)
  integer a(*)
  integer i,k,s
  s=0
  do i=1,k; s=s+a(i); end do
  sumarr=s
   end function sumarr
end program parmatr
```

Для сопоставления представленных в комментариях результатов выполнения отдельных операторов этой программы с требуемыми приведем распечатку исходной матрицы:

```
     1    5    9    13
     2    6   10    14
     3    7   11    15
     4    8   12    16
```

Отметим, что в процедуре `sumarr` принципиально необходимо указание числа элементов суммируемого массива, поскольку соответствующий параметр не имеет формы и к нему нельзя обращаться с функцией `size`.

Результатом функции в FPS может быть массив. В этом случае в ее заголовке должна присутствовать спецификация (`result <имя>`), а для массива в спецификации параметров должны быть указаны явные границы. Естественно, что ответное значение должен получать массив, специфицированный в `result`.

Приведем программу решения системы линейных алгебраических уравнений $Ax = b$ по формуле $x = A^{-1}b$ для двух векторов b при известной обратной матрице коэффициентов с помощью матричной функции.

```
program vecfun
   interface
      function mvm(a,b) result (c)
         real a,b,c
         dimension a(:,:),b(:),c(size(a,1))
      end function mvm
   end interface
   real ainv,b1,b2,x1,x2
   dimension ainv(3,3),b1(3),b2(3),x1(3),x2(3)
   data ainv /1.0,2.6,3.4, -1.7,4.4,0.8, 3.6,-4.3,2.7/, &
          b1 /0.8,-4,2.5/, b2 /2.6,0.8,3.7/
   x1=mvm(ainv,b1)
   print *, x1       ! 16.60  -26.27   6.27
   x2=mvm(ainv,b2)
   print *, x2       ! 14.56   -5.63  19.47
end program vecfun

function mvm(a,b) result (c)
   real a,b,c
   dimension a(:,:),b(:),c(size(a,1))
   do i=1,size(a,1)
      c(i)=dot_product(a(i,:),b)
   end do
end function mvm
```

Здесь применена покомпонентная операция скалярного перемножения векторов `dot_product`. Отметим также обязательность в данном случае явного интерфейса.

Процедуре, имеющей в качестве параметров массивы изменяемого размера, могут понадобиться и локальные массивы с изменяемым размером. Границы таких массивов при их отсутствии в списке параметров могут определяться с помощью функции `size`.

Размещаемые массивы не могут быть формальными параметрами процедур или результатами функций, однако их использование в качестве *фактических* параметров допустимо. Размещаемые массивы могут использоваться и как локальные объекты процедур. Они описываются по схеме

```
real, allocatable,dimension(:,:) :: u
```

Память выделяется им оператором `allocate` после вычисления требуемых границ. Перед выходом из процедуры занятая размещаемыми массивами память должна

быть освобождена оператором **deallocate**, иначе возникнут неприятности при повторном обращении.

Любые переопределения по ходу выполнения процедуры переменных, входящих в выражения для граничных пар массивов-параметров, не изменят фактических границ этих массивов.

Если вместо имени массива в качестве фактического параметра подставляется его элемент, то синонимом массива на машинном уровне становится адрес этого элемента. Следовательно, описанный выше механизм сопоставления фактического и формального параметров будет работать с дополнительным сдвигом, и передаваться будут лишь элементы, начиная с указанного. В связи с обсуждаемой возможностью напомним, что матрицы располагаются в памяти по столбцам.

Применение данного приема не рекомендуется.

5.8.2. Процедуры как параметры

Иногда параметром подпрограммы (функции) должна быть другая подпрограмма или функция. Подобные параметры называются формальными процедурами. В частности, они необходимы в стандартных *функциональных* процедурах: минимизации, аппроксимации, интегрирования и дифференцирования произвольных функций, решения уравнений и т.п. Атрибут **intent** для такого параметра не нужен, а блок **interface** в вызывающей процедуре — обязателен.

Пример. Необходимо подсчитать

$$\int_0^b e^{-y}\sin y\,dy - \int\limits_0^c \cos z\,dz$$

с помощью функции, вычисляющей $\int_0^a f(x)\,dx$ по формуле трапеций с разбивкой интервала на 10 частей.

```
program numint
   real b,c,z,z1,z2
   interface
      real function esin(y)
         real y
      end function esin

      real function mycos(y)
         real  y
      end function mycos
   end interface
   b=2.0; c=1.6
   z1=sintg(b,esin); z2=sintg(c,mycos)
   z=z1-z2
   print *, ' z = ',z ! -5.347379e-01
contains
   real function sintg(a,f)
      real  a,f
      real  h,x,s
      integer i
      s=(f(0.0)+f(a))/2.0
      h=a/10.0; x=h
```

```
         do i=1,9; s=s+f(x); x=x+h; end do
         sintg=s*h
      end function sintg
   end program numint

   real function esin(y)
      real y
      esin=exp(-y)*sin(y)
   end function esin

   real function mycos(y)
      real y
      mycos=cos(y)
   end function mycos
```

Эту задачу удалось решить только после переопределения встроенной функции (см. функцию `mycos`). Теоретически должны были сработать комбинация описателя `intrinsic acos` (специфическое имя вместо родового) и соответственно скорректированный оператор вычисления `z2`.

Отмеченные ограничения относятся к случаю, когда процедура передается как *правило* для получения значения. Готовые значения соответствуют статическим формальным параметрам и вычисляются *перед входом* в процедуру.

5.8.3. Необязательные и ключевые параметры

Список параметров бывает довольно длинным, что затрудняет работу с ним и часто приводит к ошибкам. Если некоторые параметры не обязательны, то их можно объявить с дополнительным атрибутом `optional` и задавать при вызове только по необходимости. В теле такой процедуры необходимо обращением к встроенной логической функции `present(<имя_параметра>)` проверить наличие необязательного параметра и в зависимости от ее результата запрограммировать дальнейшие действия. Описание процедуры не фиксирует способ задания аргумента, что позволяет при вызове использовать любой из них или их комбинацию. Обсуждаемый прием является удобным средством реализации различных *умолчаний*.

Выполним расчет часто встречающегося в задачах оптимизации отношения скалярных произведений векторов $s = (a \cdot b)/(a \cdot a)$:

```
   program optparms
      interface
         real function dpropt(a,c)
            real a(:), c(:)
            optional c
         end function dpropt
      end interface
      real a,b,s
      dimension a(3),b(3)
      data a/2,6,5/, b/1,8,3/

      s=dpropt(a,b)/dpropt(a)
      print *, 's = ',s
   end program optparms
```

```
real function dpropt(a,c)
   real a(:), c(:)
   optional c
   if (present(c)) then
          dpropt=dot_product(a,c)
      else
          dpropt=dot_product(a,a)
   end if
end function dpropt
```

Этот прием хорошо сочетается с применением *ключевых* аргументов, подставляемых при вызове по схеме

<имя_формального_параметра>=<значение>.

Для работы с ключевым параметром в программе `optparms` достаточно заменить только одну строку на `dropt=dot_product(a,c=b)`.

При вызове процедуры сначала перечисляются позиционные аргументы, а затем ключевые. Использование ключевых параметров усугубляет необходимость интерфейса, поскольку только из него вызывающая программа может узнать имя параметра. Применение ключевых аргументов с мнемоническими именами улучшает читабельность программы и исключает ошибки из-за перестановки параметров.

5.9. Рекурсивные процедуры

По умолчанию процедура не может вызвать сама себя ни прямо, ни косвенным образом (посредством цепочки вызовов). Однако рекурсия иногда желательна. Для примера сошлемся на проблемы, которые могут быть решены как серия задач меньшего размера (вычисление гамма-функции, сортировки — см. разд. 9.4.6, динамическое программирование, метод ветвей и границ), а также вычисление кратного интеграла. В последнем случае внешний интеграл использует в качестве подынтегральной функции внутренний, и оба могут быть вычислены повторным обращением к одной и той же процедуре численного интегрирования. Для возможности таких действий заголовок вызываемой процедуры должен иметь префикс `recursive`.

Если функция вызывает себя непосредственно, то для ее результата необходимо имя, отличное от имени функции. Тогда значение результата присваивается выбранному имени, и оно же указывается в заголовке процедуры после списка параметров в форме `result(<имя>)`. Проиллюстрируем эту технику на классической задаче вычисления факториала:

```
program recur
   integer x,y
   interface
      recursive function fact(x) result(fx)
         integer x,fx
      end function fact
   end interface
   y=fact(6)
   print *, 'fact(6) = ',y
end program recur
```

```
recursive function fact(x) result(fx)
   integer fx
   integer x
   if (x>0) then
      fx=x*fact(x-1)
   else
      fx=1
   end if
end function fact
```

Более содержательна задача рекурсивного расчета многочленов Чебышева. Эти многочлены вычисляются по формулам

$$
\begin{aligned}
T_0(x) &= 1, \quad T_1(x) = x, \\
T_k(x) &= 2xT_{k-1}(x) - T_{k-2}(x), \quad k = 2, 3, \ldots
\end{aligned}
$$

Ниже приведена программа ее решения:

```
program chebrec
   real x /2/,y
   integer k
   interface
      recursive function cheb(k,x) result(tch)
         integer k
         real    tch
      end function cheb
   end interface
   do k=0,3
      y=cheb(k,x)
      print *, 'k = ',k, ' y = ',y ! 1, 2, 7, 26
   end do
end program chebrec

recursive function cheb(k,x) result(tch)
   integer      k
      real,save ::  t0
      real          t1,tch
      t0=1.0; t1=x
      if (k==0) then
         tch=t0
      else if (k==1) then
         tch=t1
      else
         t1=cheb(k-1,x)
         tch=2*x*t1-t0
         t0=t1
      end if
 end function cheb
```

В рекурсивной функции `cheb` принципиальную роль играет атрибут `save` переменной `t0`, обеспечивающий сохранение ее значений между вызовами. Для таких переменных инициализация выполняется только при первом вызове процедуры.

Реализация рекурсии связана с повторным входом в блок (процедуру) до выхода из предыдущего вхождения. Соответственно она порождает одновременно существующие копии блока в количестве, определяемом кратностью выполняемой рекурсии, и может быть весьма накладной по расходу памяти и времени счета. Если рекурсию удается заменить рекуррентным счетом, такую возможность нужно обязательно использовать. Приведем *рекуррентный* вариант программы решения той же задачи о многочленах Чебышева:

```
program recurr
   real x /2/,y
   integer k
   interface
      real function cheb(k,x)
         integer k
         real    x
      end function cheb
   end interface
   do k=0,3
      y=cheb(k,x)
      print *, 'k = ',k, ' y = ',y ! 1, 2, 7, 26
   end do
end program recurr

real function cheb(k,x)
   integer   k
   real      t0,t1
   t0=1.0;   t1=x
   if (k==0) then
      cheb=t0
   else if (k==1) then
      cheb=t1
   else
      do n=2,k
         t2=2*x*t1-t0
         t0=t1
         t1=t2
      end do
      cheb=t2
   end if
end function cheb
```

5.10. Общие области и подпрограммы данных

Информация из одной программной единицы в другую обычно передается посредством аппарата формальных и фактических параметров. Однако для этого существует и другая возможность. Под объекты, объявленные в операторах описания *общих областей* разных программных единиц, выделяется один и тот же участок памяти. Так, если в одной программной единице использовалась простая переменная `x`, а в другой — переменная `p` и потребовалось, чтобы эти переменные означали один и тот же объект, то, написав в одной программной единице оператор вида `common x`, а в другой `common p`, мы обеспечим выделение под величины `x` и `p`

одного и того же поля памяти. И если в процессе реализации одной программной единицы изменится **x**, то в другой точно так же изменится **p**.

Использование аппарата общих блоков позволяет

- экономить память ЭВМ, исключая дублирование информации;
- гарантировать *целостность* общей информации, т.е. ее идентичность для разных потребителей;
- максимально распараллелить процесс программирования, сведя к минимуму число предопределенных обозначений;
- облегчить программирование процедур и обращений к ним благодаря сокращению списка параметров.

С другой стороны, этот аппарат в связи с уменьшением наглядности обмена информацией снижает *надежность* программирования.

Оператор описания общих областей относится к невыполняемым операторам и имеет вид

```
common /<имя_области>/ <список_данных>
       ................................
       /<имя_области>/ <список_данных>
```

Имена областей могут и отсутствовать (точнее, записываться в виде //). Тогда содержимое этих областей относится к *единственному* непоименованному общему блоку. Элементы одноименных (в частности, непоименованных) общих областей, перечисленные в разных программных единицах, отождествляются в порядке их перечисления. Разумеется, это предполагает однотипность сопоставляемых объектов. Имена общих областей не имеют отношения к другим именам: одним и тем же именем можно назвать как общую область, так и другой объект программы. Список состоит из элементов, отделяемых друг от друга запятыми. Элементами списка могут быть переменные и определения массивов — например, q(2,12).

Длины одноименных общих областей должны совпадать. Непомеченные общие блоки, определенные в разных программных единицах, могут иметь разную длину.

Программная единица **block data** используется для инициализации объектов *именованных* общих блоков. Структура ее видна из примера:

```
block data startup
   character band(len=12), range(len=15), pre(len=9), &
             alloc(len=17)
   integer line_width, font, sl
   common /head/ band, range, pre, alloc
   common /page/ line_width, font, sl
   data range, pre /'range of values', 'precision'/
   data alloc /'mem_alloc'/, line_width, sl /79,60/
end block data
```

```
block data
   integer u(3)
   real m
   logical l1,l2
   dimension s(4),t(2)
   complex z
   commom /oba/u,m,l1,l2 /obb/s,z,t(2)
   data u(1),u(2),u(3)/3*12/, m/13.1/, &
      & l1,l2/.true.,.false./,         &
      & s(1),s(2),s(3),s(4)/2*3.1,2*7.8/, z(1.0,1.0)/, &
      & t(1),t(2)/1.15,.26/
end block data
```

Здесь общие объекты сначала специфицируются, затем распределяются по общим областям и, наконец, получают значения в операторе `data`. Поскольку в этом операторе имена общих областей не указываются, среди имен получающих значения объектов не должно быть совпадающих.

Элементам *непоименованных* общих областей присвоить начальные значения подобным образом нельзя.

5.11. Модули и работа с ними

Модули оформляются отдельными файлами в составе проекта и компилируются отдельно от главной программы. Они могут

- содержать несколько обычно используемых процедур,
- объявлять глобальные переменные и производные типы,
- объявлять интерфейс для содержащихся в библиотеке внешних процедур,
- инициализировать глобальные данные и динамические массивы.

Модули могут использоваться для организации библиотеки подпрограмм. Модульные процедуры могут содержать собственные внутренние процедуры. Переменные, в прежних версиях Фортрана хранимые в общих блоках, в FPS могут быть перенесены в именованные модули. При переработке таких программ описания общих блоках должны быть заменены операторами `use <имя_модуля>`.

Применение модулей напоминает явную вставку в программу операторами `include <имя_файла>` текстов используемых процедур, но имеет следующие преимущества:

- Может определять производные типы данных, специальные операторы для производных типов, обобщать встроенные операторы на производные типы данных.
- Избавляет программиста от детального знакомства с упомянутыми объектами ("инкапсулирует" их).
- Контролирует имена констант, переменных и процедур и при необходимости обеспечивает временное переименование названных объектов.

- Организует для вызывающей программы интерфейс с модульными процедурами.
- Концентрирует всю нужную информацию и обеспечивает ее единообразное использование разными вызывающими программными единицами.

Таким образом, модуль является идеальным инструментом программной реализации абстрактных типов данных.

Модуль имеет следующую структуру:

```
module <имя_модуля>
   [<операторы_описания>]
   [contains
      <модульные_процедуры>]
end [module [<имя_модуля>]]
```

Объекты модуля доступны программной единице, содержащей (сразу после заголовка) оператор `use <имя_модуля>`. Используемый модуль может, в свою очередь, ссылаться на другой модуль и т.д., однако транзитивный вызов объектов не допускается: каждому используемому модулю должен соответствовать свой `use`.

Оператор `use <имя_модуля>` открывает вызывающей компоненте доступ ко всем объектам указанного модуля, исключая имеющие атрибут `private`. Такими объектами могут быть производные типы, переменные, именованные константы, группы `namelist`, блоки интерфейса и общие идентификаторы, именующие семейство процедур.

5.11.1. Пример "модульной" программы

Приводимый ниже модуль содержит процедуры работы с верхними треугольными матрицами: упаковки квадратной матрицы в треугольную, распаковки треугольной в квадратную с заполнением нижней левой части нулями и перемножение упакованных матриц.

```
module trimatr
contains
   subroutine packm(n,a,p)
      integer n
      real a(:,:), p(:)
      integer i,j,k
      k=1
      do i=1,n
         do j=i,n
            p(k)=a(i,j); k=k+1
         end do
      end do
   end subroutine packm

   subroutine unpackm(n,a,p)
      integer n
      real a(:,:), p(:)
      integer i,j,k
```

```
      k=1
      do i=1,n
         do j=1,i-1; a(i,j)=0; end do
         do j=i,n
            a(i,j)=p(k); k=k+1
         end do
      end do
   end subroutine unpackm

   subroutine triprint(n,c)
   ! "Triangle" printing
      integer n
      real c(:)
      integer f1,f2,m1,m2,i,j
      character(len=12) sf
      sf=repeat(' ',12)
      sf(1:5)='(  x,' ;  sf(8:)='f8.3)'
      f1=1; f2=n
      m1=1; m2=n
      do i=1,n
         write(sf(2:3),'(i2)') f1
         write(sf(6:7),'(i2)') f2
         write(*,sf) (c(j),j=m1,m2)
         f1=f1+8; f2=f2-1
         m1=m2+1; m2=m2+n-i
      end do
   end subroutine triprint

   subroutine trimult(n,a,b,c)
      integer, intent(in) :: n
      real, intent(in),  dimension(:) :: a,b
      real, intent(out), dimension(:) :: c
      real s

      integer i,j,k,l,m,r
      l=0; r=1
      do i=1,n
         do j=i,n
            s=0; m=l+j
            do k=i,j
               s=s+a(l+k)*b(m); m=m+n-k
            end do
            c(r)=s; r=r+1
         end do
            l=l+n-i
      end do
   end subroutine trimult
end module trimatr
```

Кроме того, в модуль включена обсуждаемая в главе 6 процедура "треугольной распечатки" `triprint`.

Из приведенных выше процедур нуждается в серьезном анализе только выполняющая перемножение матриц. Заложенные в нее построение циклов и переадресацию нетрудно получить, записав алгоритм перемножения *распакованных* треугольных матриц (см. вызывающую программу):

```
program triangle
   use trimatr
   integer, parameter   :: n=5
   integer, parameter   :: kn=n*(n+1)/2
   real, dimension(n,n) :: a,b,c
   real, dimension(kn)  :: au,bu,cu
   real                    s
   character(len=10)    :: fvar
   do i=1,15
      au(i)=0.1*i; bu(i)=0.2*i ! Задание упакованных
   end do
   call trimult(n,au,bu,cu)    ! Упакованное умножение
   call triprint(n,cu)         ! Треугольная распечатка
   call unpackm(n,a,au)        ! Распаковка a
   call unpackm(n,b,bu)        ! Распаковка b
   ! Распакованное умножение
   do i=1,n
      do j=1,n
         s=0.0
         do k=1,n; s=s+a(i,k)*b(k,j); end do
         c(i,j)=s
      end do
   end do
   print *,''                   !  Вывод пробельной строки
   write (*,'(1x,5f8.3)') c     !  Контрольная печать
end program triangle
```

Приведем контрольный вывод результатов работы этой программы:

```
 .020   .280    .940   2.100   3.800
 .000   .720   2.240   4.580   7.700
 .000   .000   2.000   5.060   9.080
 .000   .000    .000   3.380   7.840
 .000   .000    .000   0.000   4.500
```

`Triprint` элементы ниже диагонали не выводит.

5.11.2. Специальные виды модулей

Модуль с глобальными данными

Можно создать модуль, который содержит только данные и определения производных типов:

```
module data_module
   save
   real a(10), b, c(20,20)
   integer, parameter :: j=10
```

```
      integer            :: i=0
      complex d(j,j)
      type nonzero
         integer one, two
         real three
      end type nonzero
   end module data_module
```

Оператор **save** обеспечивает сохранение текущих значений объектов модуля при выходе из него.

Модуль интерфейсов

Для удобства работы с большими библиотеками интерфейсы к их процедурам (в особенности при наличии взаимных ссылок) целесообразно собрать в интерфейсный модуль. Таких модулей может быть несколько: для различных форматов данных, аппаратной и операционной среды, разной проблемной ориентации.

Глобальные динамические массивы

Если несколько программ имеют большие динамические массивы, их объявление можно сделать в общем модуле:

```
program globe
   call configarr
   call compute
end program globe

module workarr
   integer n
   real, allocatable, save :: a(:), b(:,:), c(:,:,:)
end module workarr


subroutine configarr
   use workarr
   read (*,*) n
   allocate (a(n), b(n,n), c(n,n,2*n))
end subroutine configarr

subroutine compute
   use workarr
...........
end subroutine compute
```

Семейства процедур

Интерфейсный блок позволяет реализовать *семейство* процедур общего назначения, различающихся атрибутами параметров и вызываемых по общему (родовому) имени — см. разд. 5.7. В этом случае интерфейсный блок задает родовое имя и содержит несколько интерфейсных тел (по количеству членов рода).

Если членами семейства являются модульные процедуры, то задавать тело интерфейса излишне. Вместо этого в интерфейсный блок вставляется оператор

```
module procedure <список_имен_процедур>
```

Его естественно разместить в самом модуле.

Вызов процедур семейства осуществляется по родовому имени, которое на этапе компиляции для каждого обращения заменяется специальным. Выбор последнего производится сопоставлением фактических аргументов очередного обращения со спецификациями внутри родового интерфейса.

5.12. Доступ к объектам модуля

При открытии доступа к нескольким модулям не исключено совпадение имен их внутренних объектов. Возможна также коллизия имен объектов модуля и вызывающей программной единицы. В таких случаях применяют оператор

```
use <имя_модуля>, <список_переименований>
```

Элементы списка имеют вид `<локальное_имя>=><use_имя>`. Объекты будут вызываться по локальным именам. Ограничительная спецификация `only: <список>` в операторе `use` разрешает ссылаться только на объекты, перечисленные в ее списке. Так, в результате оператора

```
use stat_lib, only: gauss, erl=>erlangian
```

из библиотеки `stat_lib` будут доступны только два модуля — `gauss` и `erlangian`, причем последний должен вызываться по имени `erl`.

Объекты модуля (переменные, процедуры, производные типы) можно сделать доступными только внутри модуля посредством описателя `private`. Аналогичные *операторы* `private` и `public` переопределяют действующее в блоке видимости относительно этого свойства правило умолчания.

Глава 6.

Ввод-вывод данных

Операции ввода-вывода используются для пересылки данных между переменными исполняемой программы в памяти компьютера и внешним устройством (терминал, принтер, диск, кассета с магнитной лентой). В начале данной главы (до разд. 6.4) мы будем рассматривать только работу с консолью.

Для ввода и вывода информации применяются операторы `read`, `print` и `write`. В каждом операторе указываются номер "устройства", которое ведет обмен данными с оперативной памятью, список данных и список форматов, управляющий обменом. В частных случаях перечисленные элементы могут назначаться по умолчанию (это относится только к устройствам и форматам) или заменяться соответствующими ссылками.

Операторы `print` и `read` при отсутствии списка управляющей информации задают последовательный форматный обмен. Если формат заменен звездочкой, представление информации определяется списком данных, номера устройств предполагаются стандартными, возникновение особых ситуаций прекращает выполнение программы.

Управляющая информация для `read` и `write` должна как минимум включать в себя параметр `unit`. Если этот параметр идет первым, то ключевое слово можно опускать. Состав управляющей информации обсуждается в разд. 6.4.

6.1. Список ввода-вывода

Элементами списка ввода-вывода могут быть простая переменная, переменная с индексами, массив в целом, вырезка или сечение массива, циклический список. Если элементом списка обмена является массив или вырезка, то компоненты последних участвуют в обмене в том порядке, в каком они хранятся в оперативной памяти (матрицы — по столбцам).

При необходимости передать только часть массива или изменить естественный порядок следования его элементов применяется циклический список вида `(l1,l2,...,ln, i=m1,m2[,m3])`, в котором `l1,l2,...,ln` — элементы, зависящие от переменной i (в общем случае — вложенные циклические списки), а `m1,m2,m3` имеют тот же смысл, что в обычном операторе цикла с шагом. Приведем несколько примеров циклических списков вывода:

`(q(i), i=1,10,3)` — выводятся `q(1), q(4), q(7), q(10)`,

(b(j),c(j), j=1,3) — выводятся b(1), c(1), b(2), c(2), b(3), c(3), т.е. с чередованием массивов,

((q(i,j), j=1,4), i=1,6) — матрица q выводится по строкам (в отличие от стандартной очередности по столбцам).

Элементами списка ввода-вывода могут быть *выражения*, которые перед выводом автоматически вычисляются, например

```
print *, exp(1.5)/2.0
```

Каждое выполнение оператора обмена связано со значительными системными затратами на его организацию (обращение к диску). Ввод-вывод массива в целом или циклического списка значительно экономичнее, чем выполнение поэлементного обмена внутри оператора цикла. Поэтому предпочтительно накапливать результаты счета в рабочем массиве и выводить этот массив с использованием минимального числа операторов вывода.

6.1.1. Ввод, управляемый списком

Входная запись в данном случае содержит последовательность значений, разделенных пробелами или запятыми и пробелами. Пробелы не интерпретируются как нули. Сами значения представляются аналогично соответствующим их типу буквальным константам — см. разд. 1.6. Им могут предшествовать повторители. Слэш означает конец информации, воспринимаемой данным оператором ввода.

6.1.2. Группа NAMELIST

Иногда — в особенности при отладке — бывает удобно собрать ряд переменных в группу, чтобы затем ссылаться на группу в целом и тем упростить программирование вывода. Это делается оператором

```
namelist /<имя_группы>/ <список_переменных>
```

Таких групп может быть несколько. Одна и та же переменная может входить в несколько namelist-групп. Оператор

```
write (*,[nml=]<имя_группы>)
```

эквивалентен включению в список вывода всех связанных с этим именем переменных. В выходном тексте выводу элементов группы предшествует сообщение &<имя_группы> (заглавными буквами); скалярные элементы списка выводятся по одному в строке в виде <имя>=<значение>, причем формат значений определяется атрибутами переменных. Имена групповых объектов (массивов) выводятся однократно, в строке размещается несколько элементов. Группа заканчивается слэшем.

Аналогично готовят данные для ввода по namelist.

6.2. Спецификации формата

При необходимости вывода в нестандартном (отличном от умолчания) формате или в целях управления размещением информации на выходном документе применяются *спецификации формата*. Они задаются в виде текстовой строки непосредственно в операторе ввода-вывода или отдельным оператором format. Данные, назначенные для ввода-вывода, представляются в наборах данных на внешних носителях последовательностью букв, цифр и специальных знаков. Позиции, отводимые под значение некоторой величины, образуют *поле* этой величины. При

необходимости пробелов между данными эти пробелы удобнее добавить к величине поля. Приведем примеры основных видов спецификаций формата:

`i10` — целое, поле из 10 позиций,

`f10.5` — вещественное, поле 10 позиций, 5 знаков после точки,

`e12.4` — вещественное с плавающей точкой в поле из 12 позиций; мантисса из 4 цифр со старшей цифрой сразу после точки; показательная часть состоит из буквы `e`, знака и двух цифр (при модуле порядка более 99 — из знака и трех цифр);

`a10` — текст из 10 символов.

Имеется возможность управлять положением точки в мантиссе. Спецификация типа `en` (инженерная) формирует десятичный порядок кратным трем и мантиссу между 1 и 1000; спецификация `es` (научная) — мантиссу между 1 и 10 с точкой после старшей цифры. По умолчанию вывод организуется в универсальном `g`-формате, который при малой абсолютной величине порядка выводит числа с фиксированной точкой, а в противном случае — с плавающей. Если число затребованных значащих цифр меньше, чем во внутреннем представлении, то производится округление.

Предшествующий (через пробел или запятую) числовым форматам дополнительный описатель `ss` подавляет вывод знаков '+' у положительных чисел, `sp` задает вывод плюсов, `s` восстанавливает стандартный режим.

Логические переменные выводятся по формату `lw` — в правой части поля из `w` позиций помещается буква `f` или `t`.

Текстовый формат `a` может быть указан без спецификации поля: тогда поле будет назначено по фактической потребности. При несоответствии заданного поля длине текста последний будет обрезан справа или дополнен пробелами — слева.

Спецификация формата состоит из списка дескрипторов, заключенного в скобки. Она может быть запрограммирована как встроенное в оператор вывода текстовое выражение или как отдельный оператор, на который ссылаются по его метке. Таким образом, для распечатки переменных `k,h,t` целого, вещественного и текстового типов соответственно следует записать

```
      print '(i10,f10.5,a10)', k,h,t
```

либо

```
         print 10, k,h,t
   10    format '(i10,f10.5,a10)'
```

Фрагменты текста, содержащие имена или поясняющие смысл выводимых объектов, могут быть непосредственно включены в оператор формата:

```
      110 format(' k = ',i10,' h = ',f10.5,' t = ',a10)
```

На оператор формата возможны многократные ссылки. Обычно операторы формата помещают непосредственно за использующими их операторами обмена или собирают в одном месте программной единицы.

Оператор формата должен находиться в той же программной единице, что и ссылающиеся на него операторы. По этой причине, в отличие от общего правила использования локальных имен, недопустимы ссылки на оператор формата носителя из внутренней процедуры.

При замене спецификации формата звездочкой данные выводятся в стандартной форме, зависящей от описания переменной и используемого компьютера. Это средство обычно используется для отладочных и сигнальных выдач.

При нехватке поля для размещения выводимого объекта все оно заполняется звездочками.

В инструментальной панели MDS имеется специальная кнопка для вызова Редактора форматов. Она срабатывает, когда курсор стоит в содержащей описатель формата строке программного окна. В открывающейся панели можно выбрать режимы `New Field`, `Remove Field` или `Change Value`. Последний режим служит для корректировки описания поля.

6.2.1. Управление размещением информации

Управление *размещением* выводимой информации внутри строки производится дескрипторами `tn`, `trn`, `tln`, из которых первый продолжает печатать с `n`-й позиции текущей строки, а второй и третий — с `n`-й позиции справа или слева от текущей соответственно. Дескриптор `nx` эквивалентен `trn` и обычно используется для вставки `n` пробелов. Напомним, что `n` должно быть целой константой.

Переход на новую запись (строку) в спецификации формата задается слэшем (/). Двойной слэш обеспечит пропуск чистой строки. Обратный слэш указывает необходимость продолжения прежней записи (это средство часто используется при выводе на терминал).

Первый символ каждой записи, выдаваемой на печатающее устройство оператором вывода, рассматривается как символ управления кареткой. Стандартом языка предусмотрены следующие действия:

пробел	начать новую строку
+	остаться на той же строке
0	пропустить одну строку
1	начать новую страницу

Любой отличный от указанных символ вызовет переход на новую строку, но сам будет "проглочен". Во избежание неприятных неожиданностей рекомендуется вставлять пробел в начало каждой записи или предусматривать для нее расширенное поле. При повторной отработке формата для продолжения печати с новой строки пробел должен входить в повторяемую часть. Несколько простейших примеров:

```
integer i
real, dimension(10) :: a
character(len=20)   word
print '(3f10.3),    a(1),a(2),a(3)
print '(a(10)',     word(5,14)
print '(5f10.3)',   (a(i), i=1,9,2)
print '(f10.3)',    sqrt(a(3))
```

6.2.2. Переменные спецификации формата

Спецификация формата должна быть полностью установлена перед началом операции ввода-вывода и не может переопределяться в ходе ее выполнения. При

необходимости в изменении формата (чаще всего повторителей, зависящих от результатов счета или переменных цикла), следует

- задать его шаблон как строковую переменную (например, `stf`),
- предусмотреть настройку шаблона с помощью встроенных функций для работы с символами и строками (см. разд. 4.2),
- сослаться на преобразованный шаблон в конструкции типа `write (12, fmt=stf) <список-вывода>`.

Пример практического решения такой задачи приводится в разд. 6.5.

Эти же проблемы в FPS теоретически преодолеваются с помощью "фирменного" расширения языка, допускающего в операторах формата целочисленные *выражения* в угловых скобках <...>, например

```
40 format (<7*(n-j)>x, <j>f7.3)
```

Значение такого выражения может перевычисляться в процессе выполнения оператора ввода-вывода. К сожалению, заставить эту конструкцию работать не удалось, хотя компиляция проходила без диагностики ошибок.

6.3. Взаимодействие списков объектов и форматов

При выводе происходит попарное сопоставление элементов списка вывода и списка форматов с учетом числа элементов в массиве (сечении, вырезке, строке или подстроке) и повторителей форматов. Повторители могут быть только целыми числовыми константами. На каждое комплексное число требуются два элемента формата. Повторитель может быть отнесен и к заключенной в скобки группе форматов. Если порядок извлечения элементов массива не указывается явно в списке вывода, они выводятся в порядке их расположения в памяти.

При исчерпании списка форматов он обрабатывается заново. Если формат содержит вложенные скобки, то управление будет передано к левой скобке, соответствующей предпоследней правой скобке, или к стоящему перед ней повторителю (если таковой имеется).

Рассмотрим несколько примеров реальных проблем управления выводом. Пусть необходимо распечатать матрицу `C(8,5)` по строкам с указанием против каждой из них суммы строк и под каждым из столбцов — суммы столбцов. Эта задача решается программой

```
program printmatr
   real c(8,5),g(5),h(8)
   do i=1,8;
      do j=1,5; c(i,j)=i+j; end do     ! Заполнение
   end do
   do i=1,8; h(i)=sum(c(i,:)); end do ! Суммы строк
   do j=1,5; g(j)=sum(c(:,j)); end do ! Суммы столбцов
   write (6,6)
6  format(t25,'Матрица C',t57,'Суммы строк') ! Заголовок
   print *, ' '                          ! Пробельная строка
   write (6,7) ((c(i,j),j=1,5),h(i), i=1,8)
7  format(5(2x,f8.3),7x,f6.3)
   print *, ' '                          ! Пробельная строка
   write (6,9)  g
```

```
9  format (5(2x,f8.3),2x,'<= Суммы столбцов')
end program printmatr
```

Результатом ее работы будет таблица

```
                       Матрица C                    Суммы по строкам

    2.000    3.000    4.000    5.000    6.000           20.000
    3.000    4.000    5.000    6.000    7.000           25.000
    4.000    5.000    6.000    7.000    8.000           30.000
    5.000    6.000    7.000    8.000    9.000           35.000
    6.000    7.000    8.000    9.000   10.000           40.000
    7.000    8.000    9.000   10.000   11.000           45.000
    8.000    9.000   10.000   11.000   12.000           50.000
    9.000   10.000   11.000   12.000   13.000           55.000

   44.000   52.000   60.000   68.000   76.000  <= Суммы по столбцам
```

Таблицы можно выводить с дополнительным графлением их по горизонтали (звездочками, минусами или подчеркиванием) и по вертикали (восклицательными знаками, буквами I, прямыми скобками, двоеточиями и т.д.).

Принципиально теми же средствами можно задать и построение графиков, однако в связи с дискретностью текстовых режимов эти графики смотрятся плохо. Для их построения лучше использовать специальные графические системы (например, Gnuplot [8]), входом для которых будет служить заполненный Ф90 выходной файл, или обсуждаемый ниже пакет `SciGraph`.

6.4. Внешние файлы

Файл — это именованная последовательность записей. К записям файла возможен последовательный или произвольный доступ. Мы ограничимся рассмотрением первого случая.

У каждого устройства (за исключением терминала) есть связанный с ним номер. При выполнении оператора `open` с этим номером сопоставляется некоторый существующий или вновь создаваемый файл, и тогда все дальнейшие ссылки на данный номер в операторах обмена будут адресованы к ассоциированному файлу. Эта связь разрывается оператором `close`, и только после его выполнения к тому же устройству может быть подключен новый файл. Стандартные файлы ввода-вывода считаются подсоединенными постоянно ("0" — терминал, "5" — ввод, "6" — вывод на печать). Звездочка определяет клавиатуру для ввода и экран — для вывода.

При операциях с файлами используются необязательные ключевые операнды

`unit=<номер>` — номер устройства,

`iostat=<имя-пер>` — целая переменная, которая получает нулевое значение при нормально выполненной операции и положительное в противном случае,

`err=<метка>` — метка оператора, на который передается управление при обнаружении ошибки,

`file='<имя-файла>'` — текстовая переменная, задающая имя файла (при необходимости с путем),

`status=<состояние>` — текстовое значение из множества `'old'`, `'new'`, `'replace'`, `'keep'`, `'delete'`,

а также некоторые другие. Для вновь создаваемого файла обязательно указание статуса `'new'`. При закрытии файла достаточно указать номер устройства: файл будет сохранен даже при отсутствии `keep`.

Для управления файлами с последовательным доступом служат операторы

`backspace` — переход к началу предшествующей записи,

`rewind` — возврат к началу файла,

`endfile` — запись признака конца файла

В качестве примера опишем открытие файла по имени `'points.for'`:

```
open (3, iostat=nio, err=666, file='points.for', &
         status='new')
```

Запись результатов в него может производиться оператором `write (3, <mf>) y`, где `<mf>` — числовая метка оператора `format`. Этот же файл по миновании надобности закрывается оператором

```
close (3, iostat=nio, err=666, status='keep')
```

Благодаря указанию `keep` (действует по умолчанию) файл будет сохранен во внешней памяти. Для возможности его просмотра файл нужно включить в проект по `Insert/Files into Project`. Напомним, что кириллические тексты правильно пишутся только в файлы (но не в окно вывода).

Все операторы `open` предпочтительно группировать. На подсоединенные файлы можно ссылаться в любом месте программы.

6.5. Внутренние файлы

В качестве внутреннего файла может рассматриваться строка символов. Символы представляются их `ASCII`-кодами, а при считывании их в числовую переменную автоматически преобразуются в числовое значение. Разумеется, возможно и обратное преобразование. В качестве примера приведем процедуру "треугольной" распечатки верхней треугольной матрицы по формату `f8.3` в предположении, что строка матрицы полностью помещается в строке выходного устройства.

```
subroutine triprint(n,c)
   integer n
   real c(:)
   integer f1,f2,m1,m2,i,j
   character(len=12) sf
   sf(1:5)='(  x,' ;  sf(8:)='f8.3)'
   f1=1; f2=n
   m1=1; m2=n
   do i=1,n
          write(sf(2:3),'(i2)') f1
          write(sf(6:7),'(i2)') f2
          write(*,sf) (c(j),j=m1,m2)
          f1=f1+8; f2=f2-1
```

```
        m1=m2+1; m2=m2+n-i
   end do
 end subroutine triprint
```

Основная проблема здесь заключалась в формировании строки

`sf='(ddx,ddf8.3)'`,

управляющей выводом вырезок исходного линейного массива `c` (апострофы в строку не входят, буквами `d` обозначены десятичные цифры). В начале процедуры были означены позиции `sf`, содержащие неизменную информацию. Затем были заданы начальные значения для количества пробелов `f1=1` (один обязательный пробел нужен для перехода на новую строку) и `f2=n` — полное количество элементов в строке матрицы. Внутри цикла по номерам строк текущие значения названных переменных записывались в требуемые позиции `sf` как во внутренний файл и строка отрабатывалась, после чего переменные пересчитывались.

Работа с границами выводимой в каждой строке вырезки `m1` и `m2` становится понятной после распаковки треугольной матрицы.

Разумеется, формат вывода каждого числа можно сделать дополнительным параметром процедуры — здесь новых приемов программирования не потребуется. Эта задача послужит полезным упражнением и даст ценный для практики инструмент.

Глава 7.

Производные типы данных

7.1. Производные типы данных

Новые типы данных определяются программистом на основе встроенных типов и образуют *структуры*, в общем случае состоящие из неоднородных данных. Элементы структуры в свою очередь могут быть структурами, но обязательно ранее определенных типов. В приведенной ниже программе последовательно определяются два типа: дата и включающая ее персоналия. Переменная `David` квалифицируется как персоналия и ей присваивается составное значение. Затем идет контрольный вывод всей информации и отдельно — даты (последняя в составе объемлющей структуры выводится неправильно). При выводе поля структуры не именуются, а их значения разделяются пробелами.

Далее иллюстрируется порядок изменения отдельных полей: имена структуры и ее полей разделяются знаками процента или точками (второй вариант является менее общим, но удобнее и согласуется с принятым в других языках программирования). Контрольный вывод подтверждает правильное срабатывание присваивания.

```
program newtypes
   type dat
      integer            :: day
      character(len=4)   :: month
      integer            :: year
   end type dat
   type person
      character(len=20) :: name
      type(dat)          :: birthday
      integer            :: idnum
      integer            :: salary
   end type person
   type(person)          :: David

   David=person('Copperfield D.',dat(10,' oct',1933), &
                769969,570)
   print *, 'about David ',David
   print *, 'David.birthday = ',David.birthday
   David.salary =  David.salary +75
```

```
     print *, 'new David = ',David
  end program newtypes
```

Описанные здесь конструкции заменили аналогичные им понятия **structure** для определения составного типа значений и **record** — для объявления переменных составных типов, которые существовали в первой версии FPS и используются пакетом научной графики SciGraph. Отличия сводятся к тому, что имя типа в обоих этих случаях обрамляется слэшами.

7.2. Операции над определяемыми типами

Объекты считаются однотипными, если они *формально* отнесены к одному типу (именная эквивалентность типов). Любому объекту может быть присвоено только однотипное с ним значение. Объект может быть означен оператором присваивания лишь после определения типа и отнесения объекта к данному типу — см. вышеприведенную программу.

Если программист хочет, чтобы ко вновь заданному производному типу были применимы какие-то дополнительные операции, то их тоже надо задать. В случае бинарной операции для этого пишется функция с двумя входными (intent(in)) параметрами, определяющая зависимость результата от операндов, а также интерфейсный блок, связывающий функцию с лексемой операции. Например, для типа randvar, задающего среднее и среднеквадратическое отклонение случайной величины, мы можем дать определение типа и операции сложения двух случайных величин посредством модуля

```
module  randvars
   type randvar
      real aver, sigma
   end type randvar
   interface operator(+)
      module procedure add_vars
   end interface
contains
   function add_vars(x,y)
      type (randvar):: x,y,add_vars
      intent(in) :: x,y
      add_vars.aver=x.aver+y.aver
      add_vars.sigma=sqrt(x.sigma**2+y.sigma**2)
   end function add_vars
end module randvars
```

(при сложении независимых случайных величин суммируются их средние и дисперсии; последние равны квадратам среднеквадратических отклонений).

В головной программе с обязательным включением оператора use randvars и копии интерфейсного блока из этого модуля запись обычного сложения переменных типа randvar будет приводить к вызову процедуры add_vars:

```
program tstsumrand
   use randvars
   interface operator(+)
```

```
      module procedure add_vars
   end interface
   type(randvar):: p,q,r
   p.aver=6.0;  p.sigma=3.0
   q.aver=7.0;  q.sigma=4.0
   r=add_vars(p,q)
   print *, r
end program tstsumrand
```

Как и следовало ожидать, результатом явилась пара (13,5).

Операции можно задавать для любых типов операндов при условии, что для этих типов нет встроенной операции с тем же обозначением. Можно ввести новые операции и для встроенных типов. Встроенную операцию переопределять нельзя. В качестве лексемы операции можно использовать любую из лексем встроенных операций или ограниченную с двух сторон точками произвольную буквенную последовательность (исключая занятые под обозначения логических констант и логических операций). Определяемые операции имеют самый низкий приоритет.

Покажем, как определить операцию матричного умножения. Создадим модуль

```
module matmux
   interface operator(.x.)
      module procedure mxm,mxv  ! функции !
   end interface
contains
   function mxm(a,b)
      integer, dimension(:,:)                   :: a,b
      integer,dimension(size(a,1),size(b,2)) :: mxm
      intent(in)                                :: a,b
      mxm=matmul(a,b)            ! intrinsic
   end function mxm

   function mxv(a,x)
      integer, dimension(:,:)                   :: a
      integer, dimension(:)                     :: x
      integer,dimension(size(a,1))              :: mxv
      intent(in)                                ::a,x
      mxv=matmul(a,x)            ! intrinsic
   end function mxv
end module matmux
```

Собственно перемножение матриц организуется встроенной процедурой **matmul**.

Теперь покажем использование этого модуля для перемножения

$$\begin{bmatrix} 1 & 2 & 1 \\ 3 & 4 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 2 & 2 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 6 & 7 \\ 13 & 14 \end{bmatrix}.$$

Нам необходимы:

```
program promatr
   use matmux
   interface operator(.x.)
      module procedure mxm,mxv
   end interface

   integer:: a(2,3)=reshape((/1,3,2,4,1,2/),(/2,3/))
   integer:: b(3,2)=reshape((/1,2,1,0,2,3/),(/3,2/))
   integer:: d(2,2)=reshape((/1,3,2,4/),(/2,2/))
   integer,dimension(2)    :: z, y/2,3/
   integer                 :: c(2,2)

   c= a.x.b
   print '(2i4)', ((c(i,j),j=1,2),i=1,2)
   print *,' '
   z= d.x.y
   print '(2i4)', z
end program promatr
```

Заметим, что использование $*$ в качестве символа операции умножения потребовало бы введения для матриц специального типа и переопределения всех остальных операций — следовательно, не окупилось бы.

Если производный тип определен в разделе спецификаций модуля, то включение в модуль предложения `private` после предложения `type`, но перед первым его использованием ограничивает возможность ссылок только данным модулем. Действующий по умолчанию атрибут `public` означает доступность из любого места, откуда "виден" оператор `use` этого модуля.

Глава 8.

Ссылки и списки

8.1. Базовые понятия

В FPS ссылка `pointer` — это не тип данных, а атрибут, который может быть задан для любого объекта программы в его описании или специальном операторе `pointer`. Ссылка может быть *связана* с адресатом (`target`). Тогда `pointer` будет на него указывать и может заменить объект в любом контексте. Связывание производится оператором

```
<pointer> => <target>
```

после выполнения которого ссылка занимает то же поле памяти, что адресат.

В любой момент с указателем может быть связан только один адресат (в общем случае различный на разных этапах выполнения программы), но на одного адресата могут указывать несколько ссылок. В частности, после операторов

```
p1 => t
p2 => p1
```

и `p1`, и `p2` будут указывать на один и тот же объект `t` (происходит копирование); присваивание `p2=1.0` эквивалентно `t=1.0`. Логическая функция

```
associated(<объект>[,<объект>])
```

позволяет проверять наличие связи между объектами (в частности, между двумя ссылками на одну цель).

Указатель-массив должен иметь ту же форму и тип значений, что адресат. Он объявляется аналогично размещаемым массивам (т.е. с пустыми граничными парами), но с ключевым словом `pointer` вместо `allocatable`. Перед связыванием с матричным объектом такой указатель должен получить границы в операторе `allocate`. Элемент или сечение массива не могут быть указателями.

С другой стороны, каждый потенциальный адресат должен иметь атрибут `TARGET`. Примеры назначения ссылок:

```
real    a, x(:,:), b, z(6,6)
pointer a,x
target  b,z
a => b      ! скалярное назначение ссылки
x => z      ! ссылка-массив
```

8.2. Ссылки как псевдонимы

Если одно и то же сечение массива, скажем `table(m:n,p:q)`, часто используется в программе при неизменных границах, то на него было бы удобнее ссылаться как на самостоятельно именованный массив. В FPS эта возможность реализуется с помощью ссылок-массивов и операторов назначения ссылок. В данном случае вспомогательный массив можно было бы описать как

```
real, dimension(:,:), pointer :: temp
```

и прикрепить к `table` оператором

```
temp=>table(m:n,p:q)
```

Массив `table` должен иметь атрибут `target` или `pointer`. Если впоследствии вырезку нужно будет изменить, то потребуется только новое прикрепление. Нужно иметь в виду, что нижние границы индексов вспомогательного массива будут всегда единичными, а верхние определятся экстентами адресата. Соответственно он должен индексироваться в новых границах.

Скаляр или массив с атрибутом `pointer` не имеет поля памяти, пока он не связан с адресатом, имеющим память, или пока не назначен оператором `allocate` (в этом случае его адресат — выделенное поле памяти). Динамическое управление созданием, ассоциированием и отключением ссылок обеспечивается ссылочными операторами присваивания и операторами `allocate`, `deallocate` и `nullify`. Ссылочное присваивание связывает ссылки с существующими целями. `allocate` создает цель (выделяет память) для указателей; `nullify` отсоединяет ссылку от цели (уводит ее "в никуда"), `deallocate` освобождает память из-под целей.

Сложные структуры с динамически выделяемой памятью в общем присущи скорее информационно-логическим, чем вычислительным применениям (исключая проблемы типа метода ветвей и границ). Обсуждаемые в [6, 13] методы записи треугольных матриц выглядят противоестественными.

8.3. Связные списки

Простейший список создается построением цепочки элементов (узлов), каждый из которых хранит некоторое значение и указывает на предыдущий узел. Определим сначала *тип* "узел" (`node`) и две переменных этого типа:

```
type node
   real                    value
   type(node), pointer :: next
end type node
```

Заметим, что производный тип `node` определен рекурсивно — через себя же. Это можно делать только для указателей.

Вначале список пуст, и мы должны отразить это обстоятельство указанием на отсутствие смежного элемента:

```
nullify (next)
```

Ниже "для разминки" приведен фрагмент программы формирования списка чисел, введенных с клавиатуры, которая завершает работу при вводе первого отрицательного числа.

```
 type(node), pointer    :: current, last
do
  read *, number            ! считывание
  if (number>=0) then       ! принимаем число
     allocate (current)     ! новый узел
     current.value = number ! запись введенного числа
     current.next => last   ! ссылка на предыдущий узел
     last => current        ! обновление конца списка
  else
     exit
  end if
end do
```

Теперь приведем модуль с определением типа элементов списка целых чисел

```
module eni
   type entry
      integer val            ! Значение элемента
      integer index          ! Номер элемента
      type(entry),pointer :: next
   end type entry
end module eni
```

и программу стандартных действий со списками:

```
program listwork
   use eni
   interface                 ! Для процедуры вывода списка
      subroutine view(top)
         use eni
         type(entry),pointer:: top
      end subroutine view
   end interface
   type (entry), pointer :: tree,top,current,new,last
   integer i

   nullify(tree) ! Нет предшественников
   ! Заполнение списка
   do i=1,9
      allocate(top)
      top=entry(11.0*i,i,tree)
      tree=>top
   end do
   call view(top) ! Контрольный вывод

   ! Коррекция списка
   tree=>top
   do while(associated(tree))
       if (tree.index==5) then
          tree.val=500
          tree.next.val=400
          exit
```

```
         end if
         current=>tree.next
         tree=>current
      end do
      call view(top)   ! Контрольный вывод

      ! Исключение элемента
      current=>top; nullify(last)
      do while(associated(current))
         if (current.index==8) then
            if (.not. associated(last)) then
                top=>top.next
            else
                last.next=>current.next
            end if
            deallocate(current)
            exit
         end if
         last=>current
         current=>last.next
      end do
      call view(top) ! Контрольный вывод

      ! Вставка элемента
      current=>top; nullify(last)
      do while(associated(current))
         if (current.index==7) then
            allocate(new)
            new=entry(800,8,last.next)
            last.next=>new
            exit
         end if
         last=>current
         current=>last.next
      end do
      call view(top) ! Контрольный вывод
   end program listwork
```

Просмотр результатов этой программы обеспечивает внешняя процедура `view`:

```
   subroutine view(top)
      use eni
      type(entry),pointer:: top,current
      current=>top
      do while(associated(current))
         print *,current.val,current.index
         current=>current.next
      end do
      read * ! Ожидание [Enter] для продолжения работы
    end subroutine view
```

Результаты (информационные поля) были следующими:

```
    Сразу после заполнения
99  88  77  66  55  44  33  22  11
    После коррекции
99  88  77  66 500 400  33  22  11
    После исключения
99  77  66 500 400  33  22  11
    После вставки
99 800  77  66 500 400  33  22  11
```

Рассмотренные примеры относятся лишь к одной из списковых структур — стеку (магазину), доступ к элементам которого осуществляется только через его вершину. Наряду с этим с помощью указателей можно построить очередь (выбор начиная с первого элемента), двунаправленный список (доступ с обоих концов), дерево (иерархическую структуру) и т.п.

8.4. Целые указатели

Целые указатели FPS являются расширением стандарта языка и отличаются от обычных указателей Ф90. Они используются для прямого управления памятью (функции, полезной в опытных руках, но справедливо расцениваемой как опасная для надежности программы).

Целый указатель имеет три компоненты: собственно указатель; базированную на нем переменную, только через которую идет обмен значениями; указуемый объект. Процесс связывания переменной с объектом — двухфазный. Сначала целый указатель связывается с базированной переменной предложением

```
pointer (p,var)
```

которое должно находиться в описательной части программной единицы. Базированная переменная `var` может быть любого типа, включая массив и символьные строки. Далее целый указатель связывается с памятью, выделенной под объект (функция `loc`), или затребованным новым участком (`malloc`):

```
real var, a
pointer (p,var)
p=loc(a)
```

Теперь `p` содержит адрес указуемого объекта (в примере — `a`). Значения, присваиваемые базированной на указателе переменной (в примере `var`), помещаются в адрес, удерживаемый указателем. Пример:

```
real      var(5), a(5)
pointer  (p,var)
p=loc(a) var(2)=0.0  ! приравнивает a(2) нулю
```

При работе с целыми указателями нельзя применять команды `allocate` или `deallocate`. Память назначается по `malloc` и освобождается командой `free`.

К целым указателям применима обычная целочисленная арифметика. Это позволяет легко пересчитывать адреса последовательных элементов массива, зная формат его компонент (в приводимом ниже примере — 8 байтов):

```
double precision a(10), var
pointer (p,var)
p=loc(a)
do i=1,10
   var=0.0
   p=p+8
end do
```

Глава 9.

Вычислительные методы на Фортране 90

9.1. Построение частотных характеристик

При расчете систем автоматического управления используется частотная передаточная функция линейного звена $W(i\omega)$, позволяющая определить зависимость коэффициента усиления и фазового сдвига выхода в зависимости от частоты ω гармонического входного воздействия. Пусть

$$W(j\omega) = \frac{K(1 + j\omega\tau)}{j\omega(1 - T^2\omega^2) + 2\xi Tj\omega}$$

Для функции $W(j\omega)$ при различных значениях ω определяются ее модуль

$$|W(j\omega)| = \frac{K\sqrt{1 + \tau^2\omega^2}}{\omega\sqrt{(1 - T^2\omega^2)^2 + 4\xi^2T^2\omega^2}}$$

и фаза

$$\psi(\omega) = -\frac{\pi}{2} + \arctan\omega\tau - \begin{cases} \arctan\frac{2\xi T\omega}{1-T^2\omega^2} & \text{если } \omega < 1/T \\ \pi - \arctan\frac{2\xi T\omega}{1-T^2\omega^2} & \text{иначе} \end{cases},$$

по которым строится амплитудно-фазовая характеристика в осях $\{\Re(W(j\omega)), \Im(W(j\omega))\}$ и логарифмическая амплитудная характеристика $L(\omega) = 20\lg|W(j\omega)|$.

Приводимая ниже программа выполняет расчет перечисленных показателей только для одного значения ω. Она иллюстрирует наиболее существенные с точки зрения техники программирования приемы работы с комплексными переменными.

```
program freqcmplx
   real a,b,l,psi,s,s1,y1,y2,omega,phase
   real,parameter    :: pi=3.14159
   complex,parameter :: j=(0,1)
   real,parameter    :: K=100.0, T=0.02, &
                        xi=0.05, tau=0.015
   complex w,jom
   omega=50.0
   jom=j*omega
   w=K*(1.0+jom*tau)/                      &
     (jom*(1.0-(T*omega)**2+2.0*xi*T*jom))
   a=real(w)
   b=aimag(w)
      print *, 'w = ',w  ! (-19.999990,-15.000010)
      print *, 'a = ',a  !  -19.999990
      print *, 'b = ',b  !  -15.000010)

   s=abs(w)
   s1=K*sqrt(1.0+(tau*omega)**2)/          &
     (omega*sqrt((1.0-(T*omega)**2)**2     &
     +4*(xi*T*omega)**2))
   print *, 's = ',s     ! 25.00
   print *, 's1 = ',s1   ! 25.00
   phase=atan2(b,a)
   y1=-pi/2+atan(omega*tau)
   y2= atan(2.0*xi*T*omega/(1.0-(T*omega)**2))
   if (omega>=1.0/T) y2=pi-y2
   psi=y1-y2
   print *, 'phase = ',phase         ! -2.498091
   print *, 'psi = ',psi             ! -2.498088
   l=20*log10(s)
   print *, 'l = ',l                 ! 27.958800
end program freqcmplx
```

Здесь поучительны

- способ задания $\sqrt{-1}$ как комплексной константы с нулевой вещественной частью,
- выделение и предварительное вычисление повторяющегося подвыражения `jom`,
- вычисление `b` как вещественного коэффициента при мнимой единице в комплексном значении `w`,
- расчет фазового сдвига обращением к встроенной функции арктангенса `atan2` от *двух* аргументов из диапазона $[-\pi, \pi]$, автоматически назначающей правильный квадрант,
- совпадение результатов вычисления модуля и фазы $W(j\omega)$ по различным формулам,
- применение встроенной функции десятичного логарифма.

9.2. Рекуррентные вычисления

Здесь мы предлагаем самостоятельно составить программы рекуррентного расчета

- N чисел Фибоначчи: $F_1 = F_2 = 1, \quad F_i = F_{i-1} + F_{i-2},\ i = 3, 4, \ldots$
- N -го многочлена Лежандра, определяемого равенствами $P_0(x) = 1$, $P_1(x) = x$, $(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$, $n = 1, 2, \ldots$
- N -го многочлена Лагерра, определяемого равенствами $P_0(b, x) = 1$, $P_1(b, x) = b + 1 - x$, $nP_n(b, x) = (-x + 2n + b - 1)P_{n-1}(b, x) - (n + b - 1)P_{n-2}(d, x)$, $n = 2, 3, \ldots$
- таблицы значений гамма-функции для аргументов от 0,5 до 6,5 с шагом 0,5 на основе равенств $\Gamma(0,5) = \sqrt{\pi}$, $\Gamma(k+1) = k\Gamma(k)$, $\Gamma(k) = (k-1)!$ для целых аргументов k .

9.3. Генерация случайных чисел

Генерация случайных чисел с требуемым законом распределения является ключевым элементом статистического моделирования, широко применяемого во многих задачах математики (вычисление многомерных интегралов), физики (рассеяние частиц), исследования операций. Такие числа обычно получаются преобразованием псевдослучайных чисел $\{U_i\}$, равномерно распределенных на $[0, 1)$.

Встроенная подпрограмма `random_number(u)` возвращает вещественное число `U` (если `u` — массив, то массив таких чисел). Подпрограмма
`random_seed([size] [,put] [,get])`
служит для установки генератора; `size` возвращает длину затравочного массива, `get` — текущий затравочный массив, `put` — задаваемый затравочный массив. При обращении к этой подпрограмме может быть задан только один аргумент — обязательно в ключевой форме. Примеры:

```
program rands
   real u(10)
   integer m,p(1),r(1)

   call random_number(u)
   write (*,55) u
55 format(1x,f8.6)
   call random_seed(m)
   print *,'m = ',m  !=1
   call random_seed(get=p)
   print *, 'p = ',p ! 1596680831
   r=1718281828
   call random_seed(put=r)
   call random_seed(get=p)
   print *, 'p = ',p ! 1718281828
end program rands
```

Фиксация начальных установок датчиков случайных чисел и произвольной смены их является необходимым условием реализации современных методов уменьшения дисперсии результатов имитационного моделирования [7].

Обычно используются *линейные конгруэнтные генераторы*, в которых u_{k+1} зависит только от u_k. При этом длина периода не превосходит количества M различных чисел. Если же $u_{k+1} = f(u_{k-1}, u_k)$, то максимальный период равен M^2. Поэтому лучшие результаты дают генераторы высших порядков вида

$$u_{i+n} = (a_1 u_{i+n-1} + a_2 u_{i+n-2} + \ldots a_n u_i) \pmod m.$$

Их частным видом являются *генераторы Фибоначчи*. Например, в программе UNI из ([5])

$$x_k = x_{k-17} - x_{k-5}.$$

Значения членов последовательности зависят от начальных 17 $\{x_i\}$ и стратегии, которую мы примем для ситуации $x_i < 0$ (стандартное решение — добавление единицы). Теоретически период здесь ограничивается $p = (2^{17} - 1)2^m$, где m — число битов мантиссы. Для стандартной арифметики $m = 24$, $p = 2^{17} \cdot 2^{24} \approx 10^{12}$. Приведем фрагмент *Фортран*-программы, иллюстрирующий циркуляцию чисел в стандартных 17 ячейках:

```
uni=u(i)-u(j)
if (uni<0.0) uni=uni+1.0
u(i)=uni
i=i-1
j=j-1
if (i<0) i=17
if (j<0) j=17
```

Далее в подпрограмме UNI значение u на выходе генератора комбинируется с выходным значением конгруэнтного генератора с плавающей точкой. Такая комбинация имеет период, равный произведению периодов составляющих генераторов, т.е. $2^{17} \cdot 2^{48} \approx 10^{19}$. Алгоритм инициализации задает случайным образом значения каждого бита мантиссы элементов затравочного массива с помощью встроенного генератора. Для последнего нужно задать одно исходное значение.

Наконец возможно введение дополнительного перемешивания, при котором сначала заполняется таблица из M чисел. Далее генерируется целое число $i \in [0, M-1]$, выдается i-й элемент таблицы, а на его место записывается новое случайное число. Если выбрать, например, M =128, то можно на каждом шаге формировать только одно случайное число, а в качестве входа в таблицу использовать семь младших разрядов этого числа.

Для получения случайной затравки можно воспользоваться показаниями системных часов:

```
..................................................
real, dimension(10)        :: u
integer, dimension(1)      :: seed
integer                       count

call system_clock(count)   ! Считывание системных часов
seed=count                 ! Запись в массив установок
call random_seed(put=seed) ! Задание установки датчика
call random_number(u)      ! Формирование случайных чисел
```

При установке от таймера результаты счета невоспроизводимы.

Много примеров и задач на использование ДСЧ для вычисления площадей фигур методом Монте-Карло приведено в [3].

9.4. Сортировка и поиск

9.4.1. Поиск в массиве

Существенной частью многих алгоритмов является поиск элемента в списке (массиве) по заданному критерию. Предположим, что все элементы массива x(1:n) различны и требуется найти номер элемента, равного z, если таковой имеется. Для поиска в среднем потребуется n/2 проверок, а в наихудшем случае — n.

Если массив x упорядочен (для определенности по возрастанию), для поиска можно применить метод половинного деления: проверить элемент в середине диапазона и в зависимости от результатов проверки в дальнейшем повторить процедуру для левой или правой половины его. Здесь среднее число проверок составит $[\log_2 n] + 1$. Приведем фрагмент программы дихотомического поиска:

```
l=1; r=n
do while(l<=n)
   i=(l+r)/2
   if (x(i)=z) then
      print *, i
      stop
   else
      if (l=r) then
         exit
      else
         if (x(i)<z) then
            l=i+1
         else
            r=i
         end if
      end if
   end if
end do
print *, 'элемент не найден'
```

9.4.2. Понятие о сортировках

Сортировка — это процесс расстановки элементов вектора в некотором порядке. Она имеет целью эффективное выполнение вычислений, предполагающих упорядоченность данных (поиск, построение функций распределения случайных величин и т.п.), и часто облегчает интерпретацию результатов. Простейшие примеры упорядоченности — по алфавиту, по возрастанию числовых значений (например, моментов наступления предстоящих событий при имитационном моделировании). Целесообразность сортировки определяется сопоставлением преимуществ работы с упорядоченным массивом и затрат на упорядочение. Сортировка не обязательно предполагает физическое упорядочение записей: можно ограничиться упорядочением ссылок на них (как в библиотеке — каталожных карточек).

Признак, по которому производится упорядочение, называется *ключом.* Различные сортировки файла могут использовать разные ключи. При необходимости упорядочения внутри упорядочения используется составной ключ.

Известно множество алгоритмов сортировки. Методы сортировки делятся на внутренние и внешние (последние — для больших файлов, размещаемых в дис-

ковой памяти). Внешние сортировки в качестве своих подалгоритмов используют внутренние — для упорядочения частей файлов, вызываемых в оперативную память.

Линейные алгоритмы предполагают отсутствие структуры в упорядочиваемом объекте: его элементы просматриваются подряд. Их средняя трудоемкость имеет порядок n^2, где n — длина массива. Нелинейные методы ориентированы на структурированные объекты (обычно — двоичные деревья). Все *эффективные* сортировки, т.е. требующие теоретически минимального среднего числа сравнений порядка $n \log_2 n$, являются нелинейными. Ниже описаны несколько наиболее популярных линейных и нелинейных сортировок применительно к упорядочению массива ключей по возрастанию. При сортировке записей в целом возможны два варианта действий: записи могут сопровождать ключи при всех перестановках или быть выстроены в результирующем массиве согласно финальной расстановке ключей.

Модификация алгоритмов применительно к другим критериям упорядоченности сводится к замене проверяемых условий.

9.4.3. Линейный выбор с обменом

Здесь в начале первого просмотра предполагается, что первый элемент списка имеет наименьший ключ. Этот ключ вместе с его адресом пересылается в рабочую память и затем сравнивается со всеми последующими элементами, пока не встретится меньший. Тогда последний заменяет собой базу сравнения, и просмотр списка продолжается.

По окончании первого просмотра базовая запись переставляется с записью из вершины списка. Второй просмотр начинается со второй записи списка, и в его итоге следующий по величине элемент занимает вторую позицию, и т.д. Работа заканчивается, когда свое место занимает $(n-1)$-я запись:

```
subroutine chanlin(n,x)
   integer, intent(in)     :: n
   real,intent(inout)      :: x(:)
   integer i,j,k
   real p
   do i=1,n-1
      p=x(i); k=i
      do j=i+1,n
         if (x(j)<p) then
            p=x(j); k=j
         end if
      end do
      x(k)=x(i); x(i)=p
   end do
end subroutine chanlin
```

9.4.4. Пузырьковая сортировка

В этом методе производится попарное сравнение *соседних* элементов и при необходимости — их перестановка. Число сравнений каждый раз уменьшается на единицу. При завершении очередного просмотра с нулевым числом инверсий сортировка заканчивается досрочно.

```
subroutine bubble(n,x)
   integer, intent(in) :: n
   real,intent(inout)::x(:)
   integer i,inv,j
   real p
   do i=1,n-1
      inv=0
      p=x(i); k=i
      do j=n-1,i,-1
         if (x(j)>x(j+1)) then
             p=x(j);   x(j)=x(j+1);
             x(j+1)=p; inv=inv+1
         end if
      end do
      if (inv==0) exit
   end do
end subroutine bubble
```

Работа с отрицательным шагом в цикле по j необходима для размещения минимальных элементов в начале массива. В ходе этой сортировки малые элементы постепенно "всплывают" к началу массива, что и определяет название метода. Большие элементы "оседают на дно": в первом прогоне самый большой, затем следующий по величине и т.д. Это обстоятельство можно использовать, когда достаточна частичная упорядоченность (например, нужно найти 5 наибольших чисел).

9.4.5. Челночная сортировка

Данный метод работает аналогично стандартному обмену до тех пор, пока не надо выполнять перестановку. Сравниваемая величина с меньшим ключом поднимается вверх по списку, насколько это возможно, сопоставляясь со всеми ее предшественниками по направлению к вершине (началу) списка. Если ее ключ меньше, чем у предшественника, то выполняется обмен и начинается очередное сравнение. Здесь в любой момент список выше последнего первичного сравнения упорядочен. Поэтому, когда передвигаемый элемент встречает элемент с меньшим ключом, этот процесс прекращается и нисходящее сравнение возобновляется с той позиции, с которой выполнялся первый обмен. Сортировка заканчивается, когда нисходящее сравнение захватит n-й элемент. Отмеченное выше свойство частичной упорядоченности позволяет применить для поиска места вставки метод половинного деления:

```
subroutine shutbin(n,x)
   integer, intent(in) :: n
   real,intent(inout)  :: x(:)
   integer i,j
   real    z
   do i=1,n-1
      if (x(i)>x(i+1)) then
         z=x(i+1);
         l=1; m=i;          ! Границы зоны вставки
         do while(m-l>1)
            k=(l+m)/2;      ! середина зоны
            if (x(k)<z.and.x(k+1)>z) then
```

```
                    exit
                 else
                    if (x(k)<z) then
                       l=k
                    else
                       m=k
                    end if
                 end if
              end do
              do j=i,k+1,-1
                 x(j+1)=x(j)
              end do
              x(k+1)=z
           end if
        end do
     end subroutine shutbin
```

9.4.6. Быстрая сортировка

Предложенная Ч. Хоаром в 1960 г. нелинейная сортировка каждый шаг своей реализации сводит к сортировке двух массивов меньшей длины. Она основана на определении места начального элемента массива в упорядоченном множестве. Покажем идею алгоритма на упорядочении по возрастанию массива

```
77   14   95   26   8   13   66   48   83   57
L1                                          R1
```

Обозначим L, R соответственно левую и правую позиции списка. Введем дополнительные рабочие счетчики L1, R1 и будем искать правильную позицию для крайнего левого значения (точки разделения). Попытаемся двигать позицию R1 влево, пока каждое значение справа от нее превосходит точку разделения. Такие сдвиги невозможны. Переставим крайние элементы и начнем сдвигать L1 вправо, пока каждое значение слева от него строго меньше точки разделения. В результате получаем

```
57   14   95   26   8   13   66   48   83   77
          L1                                R1
```

Теперь переставим помеченные L1, R1 элементы:

```
57   14   77   26   8   13   66   48   83   95
          L1                                R1
```

Снова двигая R1 влево, получаем

```
57   14   77   26   8   13   66   48   83   95
          L1                      R1
```

и после перестановки элементов в отмеченных позициях —

```
57   14   48   26   8   13   66   77   83   95
          L1                      R1
```

В этой ситуации попытка двигать L1 вправо приводит к перекрещиванию границ, что свидетельствует о завершении фазы алгоритма. В самом деле, все элементы слева от точки разделения (77) меньше ее, а все элементы справа — больше; следовательно, правильная позиция для нее найдена. Нетрудно сообразить, что тот же метод теперь можно применить к отрезкам текущей последовательности слева и справа от найденной позиции. Итак, задача может быть решена посредством *рекурсивной* процедуры:

```
   recursive subroutine quicksort(x,l,r)
      integer, intent(in) :: l,r    ! Начальные границы
      real,intent(inout)  :: x(:)   ! Имя массива
      integer l1,r1                 ! Рабочие границы
      l1=l; r1=r
 z00: do
         do while(x(r1)>x(l1))
                r1=r1-1             ! Правая - влево
                if (l1==r1) exit z00
         end do
         call swap(x(l1),x(r1))     ! Перестановка граничных
         do while(x(l1)<x(r1))
                l1=l1+1             ! Левая - вправо
                if (l1==r1) exit z00
         end do
         call swap(x(l1),x(r1))     ! Перестановка граничных
      end do z00                    ! Имя цикла обязательно !
      if (r1-l>1) call quicksort(x,l,r1-1) ! Слева
      if (r-r1>1) call quicksort(x,r1+1,r) ! Справа
   end subroutine quicksort

   subroutine swap(a,b)
      real,intent(inout)::a,b
      real p
      p=a; a=b; b=p
   end subroutine swap
```

Здесь заслуживают быть отмеченными

- рекурсивность подпрограммы `quicksort`, в двух местах обращающейся сама к себе;
- условный характер этих обращений, исключающий сортировку массивов менее чем из двух элементов;
- ”глухой” цикл `z00` без условий в его заголовке;
- вынужденное оформление этого цикла как именованного (`exit` без имени выводил бы только из внутренних циклов).

9.4.7. О внешних сортировках

При необходимости упорядочения очень больших файлов они делятся на части, каждая из которых целиком помещается в свободной оперативной памяти и может быть упорядочена одним из описанных методов. Затем упорядоченные промежуточные файлы считываются по частям, части сливаются и продукты слияния последовательно пересылаются в выводной файл.

Пусть имеются исходные упорядоченные по возрастанию массивы `real a(na)`, `b(nb)`. Их слияние в общий упорядоченный массив `c(nc)` суммарной длины можно организовать следующим образом:

```
subroutine merging(x,y,z)
   real,dimension(:) ::x,y,z
   integer nx,ny
   integer ix,iy,iz
   nx=size(x); ny=size(y);

   ix=1; iy=1; iz=1
   do while(ix<=nx.and.iy<=ny) ! Не исчерпаны оба
      if (x(ix)<y(iy)) then    ! Берем элемент из X
          z(iz)=x(ix); ix=ix+1
      else                     ! из Y
          z(iz)=y(iy); iy=iy+1
      end if
      iz=iz+1
   end do
   if (ix>nx) then             ! Присоединяем остаток Y
      do i=iy,ny
             z(iz)=y(i); iz=iz+1
      end do
   else                        ! Присоединяем остаток X
      do i=ix,nx
         z(iz)=x(i); iz=iz+1
      end do
   end if
end subroutine merging
```

Многократные слияния выполняются по схеме двоичного дерева снизу вверх (последовательно получаются сдвоенные, счетверенные и т.д. фрагменты).

9.5. Работа с разреженными матрицами

В приложениях часто встречаются сильно разреженные (с небольшим числом ненулевых элементов) матрицы большого размера, которые целесообразно хранить в виде списка ненулевых элементов с указанием их координат — номеров строки и столбца. Составим модуль, содержащий описание элемента соответствующего типа и процедуры: упаковки матрицы в список, просмотра списка, распаковки списка, сложения и умножения упакованных матриц. Поскольку списки являются стеками, при прямом просмотре исходной матрицы в верхней части стека окажутся ее последние элементы. При сложении упакованных матриц порядок элементов сум-

мы окажется прямым, что порождает необходимость в процедуре инверсии списка. Она также входит в состав модуля.

Все процедуры формирования списка помещают в вершину стека размеры распакованной матрицы.

```
module spmatr
! Работа с разреженными матрицами
   type espm
      integer im
      integer jm
      real    em
      type (espm),pointer ::prec
   end type espm

contains
   subroutine packmr(a,spa)
   ! Упаковка матрицы A в список spa
   !       по строкам
   ! в обратной последовательности
   ! ненулевых элементов
   ! В вершине списка будут размеры A
      real,dimension(:,:) :: a
      type (espm),pointer :: spa,nextm
      integer i,j,m,n
      m=size(a,1); n=size(a,2)
      nullify(nextm)
      do i=1,m
         do j=1,n
            if (abs(a(i,j))>1e-7) then
               allocate(spa)
               spa=espm(i,j,a(i,j),nextm)
               nextm=>spa
            end if
         end do
      end do
      allocate(spa)
      spa=espm(m,n,0,nextm)
   end subroutine mpack

   subroutine munpack(a,spa)
   ! Распаковка разреженной матрицы A из списка spa
      real,dimension(:,:) :: a
      type (espm)         :: spa
      type (espm),pointer :: current
      integer i,j,m,n

      m=spa.im; n=spa.jm;
      current=>spa.prec
      do i=1,m
         do j=1,n; a(i,j)=0; end do
      end do
      do while(associated(current))
```

```
          i=current.im; j=current.jm
          a(i,j)=current.em
          current=>current.prec
       end do
  end subroutine munpack

  subroutine view(spm)
  ! Просмотр разреженной матрицы, упакованной в список
     type(espm),pointer :: spm,current
     current=>spm
     do while(associated(current))
        print *, current.im, current.jm, current.em
        current=>current.prec
        read *     ! Ожидание нажатия клавиши
     end do
  end subroutine view

  subroutine addsm(sa,sb,sc)
  ! Сложение разреженных матриц sa и sb,
  ! представленных в списковой форме.
  ! sa и sb упакованы по строкам в обратном
  ! порядке,  результат sc - в прямом.
  ! В вершине списка будут размеры результата
  ! в распакованной форме
     type(espm),pointer:: sa,sb,sc,cura,curb,next
     integer i,ia,ib,j,ja,jb,m,m1,n,n1
     real a,b,x
     m=sa.im; n=sa.jm; cura=>sa.prec
     m1=sb.im; n1=sb.jm; curb=>sb.prec

     if (m/=m1.or.n/=n1) then
             print *, 'Matrices are non-additives'
             return
     end if
     nullify(next)
     do while (associated(cura).or.associated(curb))
     ia=cura.im; ja=cura.jm
     ib=curb.im; jb=curb.jm
     a=cura.em;   b=curb.em
     allocate (sc)
     if (ia==ib.and.ja==jb) then ! позиции совпали
        i=ia; j=ja; x=a+b
  ! продвижение списков обоих слагаемых
        cura=>cura.prec; curb=>curb.prec
     else
        if ((ia-1)*n+ja<(ib-1)*n+jb) then
  ! ненулевое слагаемое из b
           i=ib; j=jb; x=b
           curb=>curb.prec ! продвижение списка b
        else
  ! ненулевое слагаемое из a
```

```
            i=ia; j=ja; x=a
            cura=>cura.prec ! продвижение списка a
         end if
      end if
   ! продвижение списка результата
      sc=espm(i,j,x,next); next=>sc
   end do
   allocate (sc)
   sc=espm(m,n,0,next)       ! запоминание размера
end subroutine addsm

subroutine revert(sa,sb)
! Инвертирование списка sa разреженной матрицы
   type(espm),pointer:: sa,sb,curb
   integer i,j
   real x
   nullify(curb)
   m=sa.im; n=sa.jm;
   do while (associated(sa))
      i=sa.im; j=sa.jm; x=sa.em
      allocate(sb)
      sb=espm(i,j,x,curb)
      curb=>sb; sa=>sa.prec
   end do
   allocate(sb)
   sb=espm(m,n,0,curb)
end subroutine revert

 subroutine matspmul(sa,sb,sc)
 ! Перемножает разреженные матрицы
 ! a(m,n)*b(n,p)=c(m,p) ,
 ! упакованные в инверсном порядке
 ! левый сомножитель sa - по строкам,
 ! правый sb - по столбцам
 ! результат упакован по строкам

    type(espm), pointer :: sa,sb,sc
    integer                i,j,j0,m,m1,n,p
    real                   e
    type(espm),pointer  :: cura,curb,curc,row,rowi,curr

 !  Определение формы сомножителей
    m=sa.im;  n=sa.jm; sa=>sa.prec  ! Сдвиг
    m1=sb.im; p=sb.jm; sb=>sb.prec  ! Сдвиг
    if (n/=m1) then
       print *, 'Размеры матриц не согласованы'; stop
    end if
    cura=>sa; nullify(curc)
    do while(associated(cura))      ! По строкам результата
 ! Копирование строки левого сомножителя sa
       nullify(curr) ; i0=cura.im;
```

```
        do while (cura.im==i0)
           i=cura.im; j=cura.jm; e=cura.em
           allocate(rowi);
           rowi=espm(i,j,e,curr); curr=>rowi
           cura=>cura.prec
           if (.not. associated(cura)) exit
        end do
   ! Инвертирование ее
        nullify(curr)
           do while(associated(rowi))
              i=rowi.im; j=rowi.jm; e=rowi.em
              allocate(row)
              row=espm(i,j,e,curr)
              curr=>row
              rowi=>rowi.prec
           end do
        curb=>sb
 colsb: do while(associated(curb)) ! по столбцам результата
           s=0.0; j0=curb.jm; curr=>row
           do while (associated(curr).and.(curb.jm==j0))
           ! Накопление элемента произведения
           jr=curr.jm; ib=curb.im;
           if (jr==ib) then
              s=s+curr.em*curb.em
              curr=>curr.prec; curb=>curb.prec
           else if (jr>ib) then
              curr=>curr.prec
           else
              curb=>curb.prec
           end if
           if (.not. associated(curr)) exit
        end do
        if (abs(s)>1e-7) then
        ! Запись результата
           allocate(sc)
           sc=espm(i,j0,s,curc); curc=>sc
        end if
        ! переход к новой колонке правого сомножителя
        if (.not.associated(curb)) exit colsb
        do while (curb.jm==j0)
           curb=>curb.prec
           if (.not.associated(curb)) exit colsb
        end do
     end do colsb ! по столбцам
  end do           ! по строкам
  allocate(sc)
  sc=espm(m,p,0,curc); curc=>sc
end subroutine matspmul
end module spmatr
```

Для подготовки второго сомножителя matspmul в состав модуля включим также аналогичную packmr процедуру packmc упаковки матрицы по столбцам[1]. С помощью этого модуля проведем упаковку и суммирование разреженных матриц:

$$\begin{bmatrix} 11 & 0 & 13 & 0 & 15 \\ 0 & 22 & 23 & 24 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 41 & 42 & 0 & 0 & 45 \\ 0 & 0 & 53 & 54 & 0 \end{bmatrix} + \begin{bmatrix} 7 & -2 & 0 & 0 & 1 \\ 2 & 1 & 0 & 3 & 0 \\ 6 & 0 & 0 & 4 & 0 \\ 0 & 3 & 0 & 0 & -10 \\ 1 & 0 & 0 & -30 & 0 \end{bmatrix} = \begin{bmatrix} 18 & -2 & 13 & 0 & 16 \\ 2 & 23 & 23 & 27 & 0 \\ 6 & 0 & 0 & 4 & 0 \\ 41 & 45 & 0 & 0 & 35 \\ 1 & 0 & 53 & 24 & 0 \end{bmatrix},$$

а затем умножение

$$\begin{bmatrix} 11 & 0 & 13 & 0 & 15 \\ 0 & 22 & 23 & 24 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 41 & 42 & 0 & 0 & 45 \\ 0 & 0 & 53 & 54 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & -2 \\ -1 & 0 & 0 \\ -6 & 0 & 5 \\ 2 & 0 & 0 \\ 0 & 0 & -2 \end{bmatrix} = \begin{bmatrix} -67 & 0 & 13 \\ -112 & 0 & 115 \\ 0 & 0 & 0 \\ -1 & 0 & -172 \\ -210 & 0 & 265 \end{bmatrix}.$$

Эти действия задаются программой

```
program packmatr
   use spmatr
   type(espm),pointer       :: spa,spb,spc,spd,spf,crev
   real,dimension(5,5)      :: a,da,b,c
   real,dimension (5,3)     :: d,f

   a=0.0; b=0.0; d=0.0

   a(1,1)=11.0; a(1,3)=13.0; a(1,5)=15.0
   a(2,2)=22.0; a(2,3)=23.0; a(2,4)=24.0
   a(4,1)=41.0; a(4,2)=42.0; a(4,5)=45.0
   a(5,3)=53.0; a(5,4)=54.0

   b(1,1)=7.0; b(1,2)=-2; b(1,5)=1.0
   b(2,1)=2.0; b(2,2)=1.0; b(2,4)=3.0
   b(3,1)=6.0; b(3,4)=4;
   b(4,2)=3.0; b(4,5)=-10.0
   b(5,1)=1.0; b(5,4)=-30.0

   d(1,1)=1.0; d(1,3)=-2.0; d(2,1)=-1.0
   d(3,1)=-6.0; d(3,3)=5.0; d(4,1)=2.0; d(5,3)=-2.0

   call packmr(a,spa)           ! Упаковка a по строкам
   call view(spa)               ! Просмотр
   call packmr(b,spb)           ! Упаковка b по строкам
   call munpack(da,spa)         ! Распаковка a и вывод
   write(*,'(1x,5f8.1)') ((da(i,j),j=1,5),i=1,5)
   call addsm(spa,spb,spc)      ! Сложение упакованных
```

[1] Процедуру перемножения упакованных матриц можно было бы построить значительно проще на основе списков ненулевых элементов строк первого сомножителя и столбцов второго. К сожалению, в FPS нельзя использовать *массивы* указателей.

```
    call view(spc)                 ! Просмотр стека-суммы
    call revert(spc,crev)          ! Инверсия стека-суммы
    call view(crev)                ! Просмотр инвертированного
    call munpack(c,crev)           ! Распаковка инвертированной
                                   ! суммы и ее распечатка
    write(*,'(1x,5f8.1)') ((c(i,j),j=1,5),i=1,5)
    call packmc(d,spd)             ! Упаковка d по столбцам
    print *, ' '
    call matspmul(spa,spd,spf) !" Упакованное" умн. f=a*d
    call munpack(f,spf)            ! Распаковка и печать
    write(*,'(1x,3f8.1)') ((f(i,j),j=1,3),i=1,5)
  end program packmatr
```

Результаты ее работы совпали с полученными при распакованных операндах.

9.6. Обращение матрицы методом Гаусса

Ниже приводится модуль с функцией обращения матрицы и головная программа, применяющая ее для решения системы линейных алгебраических уравнений $AX = B$ по формуле $X = A^{-1}B$. В функции обращения поиск главного элемента производится по столбцу, при отсутствии ненулевых элементов в столбце процедура завершается выдачей сигнального сообщения.

```
module matmult

contains
   function inv(matr) result(v)
   ! См. Агеев, алгоритм 42b, с.108
      real,dimension(:,:) :: matr
      real,dimension(size(matr,1),size(matr,1))   :: v
      real,dimension(size(matr,1),2*size(matr,1)) :: a
      real,dimension(2*size(matr,1)):: row
      real t,x,z
      integer i,j,k,m,n

      n=size(matr,1); m=2*n
      a(:,:n)=matr                       ! Копируем исходную
      a(:,n+1:)=0
      do i=1,n; a(i,n+i)=1.0; end do ! Единичная
      do i=1,n
      ! Выбор ведущего элемента по столбцу
         k=i; t=a(i,i); z=abs(t)
         do j=i+1,n
            x=abs(a(j,i));
            if (x>z) then
               k=j; z=x; t=a(j,i);
            end if
         end do
         if (z==0) then
            print *, 'Нет ненулевого ведущего'; return
         end if
```

```
         if (k/=i) then  ! Перестановка строк
            row=a(i,:); a(i,:)=a(k,:); a(k,:)=row
         end if
         do j=m,i,-1; a(i,j)=a(i,j)/a(i,i); end do
         do k=1,n
            if (k/=i) then
               do j=m,i,-1;
                  a(k,j)=a(k,j)-a(i,j)*a(k,i);
               end do
            end if
         end do
      end do
      v=a(:,n+1:)
   end function inv
end module matmult
```

Обратите внимание на компактность записи алгоритма в терминах вырезок и сечений массивов.

Головная программа может иметь вид:

```
 program invtest
    use matmult
    real,dimension(4,4) :: a,ainv,p
    real,dimension(4)   :: x,b
    data a/1,2,2,0, 3,4,0,1, -2,1,-5,2, 0,-7,3,-1/, &
         b/-9,-25,1,-1/
    ainv=inv(a)
    x=matmul(ainv,b) ! Решение системы уравнений Ax=b
    print *, x
    print *, ' '
    b=matmul(a,x)    ! Контроль правых частей
    print *, b
    print *, ' '
    p=matmul(a,ainv) ! Должны получить единичную
    write (*,44) p
 44 format(1x,4f7.3)
 end program invtest
```

9.7. Решение уравнения методом секущих

Пусть уравнение $f(x) = 0$ на интервале $[a, b]$ имеет единственный корень. Обозначим $y_a = f(a)$, $y_b = f(b)$. Аппроксимируя $f(x)$ прямой, находим точку ее пересечения с осью абсцисс

$$x = a - y_a(b - a)/(y_b - y_a).$$

Вычислив $y(x)$, можно найти новый интервал с переменой знака x и т.д. до тех пор, пока этот интервал не станет достаточно мал или не станет пренебрежимо малым модуль левой части. Ниже используется второй подход, поскольку первый при неподвижности одного из концов интервала гарантирует зацикливание. Возможным критерием выхода из цикла является также малость модуля уточнения точки пересечения.

```
program secant
   real a,b,eps,x,y
   real,external ::f
   interface
      subroutine secroot(a,b,eps,f,x)
         real,intent(in)  :: a,b,eps
         real,intent(out) :: x
         real,external    :: f
      end subroutine secroot
   end interface

   a=1.0; b=2.0; eps=1e-6
   call secroot(a,b,eps,f,x)
   y=f(x)
   print *, 'x=',x, ' y=',y ! x=1.213412, y=-5.047e-7
end program secant

subroutine secroot(a,b,eps,f,x)
   real,intent(in)  ::a,b,eps ! Интервал, точность
   real,intent(out) ::x       ! Корень
   real f
   real l,r,y,yl,yr
   l=a; r=b                   ! Инициализация
   yl=f(a); yr=f(b)
   y=yl
   do while(abs(y)>eps)       ! Проверка модуля функции
      x=l-yl*(r-l)/(yr-yl)    ! Пересечение оси хордой
      y=f(x)
      if (yl*y<0) then        ! Корень слева
         r=x; yr=y
      else                    ! Корень справа
         l=x; yl=y
      end if
   end do
end subroutine secroot

real function f(x)
   real x
   f=x**3+x-3
end function f
```

Здесь полезно обратить внимание на интерфейс и спецификации параметров процедур.

9.8. Вычисление определенных интегралов

Для вычисления определенных интегралов весьма полезны адаптивные процедуры, реализующие составные квадратурные формулы. Из последних предпочтительна формула Симпсона, позволяющая минимизировать количество обращений к подинтегральной функции. В приводимой ниже программе модуль **integrals**

объединяет базовую версию и два варианта введения поправок: по Ромбергу и по Эйткену (см. разд. 3.2 [10]). Все они тестируются на задаче вычисления

$$\int_0^1 \exp(x)\,dx = e - 1 = 1.718281828459045.$$

В процедуры для возможности их сопоставления включены расчет эталона, вычисление и вывод пошаговых погрешностей. Разумеется, при последующем "боевом" использовании их для вычисления произвольных интегралов эти фрагменты должны быть изъяты. В целях экономии места для процедур-вариантов мы воспроизводим только специфические фрагменты, отличающие их от базовой.

```
module integrals
contains
  real*8 function simps(a,b,f,eps)
     interface
real*8 function f(x)
           real*8    x
end function f
     end interface
     real*8 a,b,eps
     real*8 h,e1,s1,s2,s4,v,v1,x

     e1=exp(1d0)-1d0
     h=(a+b)/2.0;
     s1=f(a)+f(b); s2=0; s4=f(a+h);
     v=(s1+4*s4)*h/3.0; n=1;
     print *, 'n = ',n, ' v = ',v-e1
     v1=0
     do while (abs(v-v1)>eps)
        v1=v; h=h/2.0;  s2=s4+s2; s4=0;
        do x=a+h, b-h/2, 2*h
           s4=s4+f(x)
        end do
        v=(s1+2*s2+4*s4)*h/3.0; n=n+1
        print *, 'n = ',n, ' v = ',v-e1
        end do
     simps=v
  end function simps

  real*8 function simpsrom(a,b,f,eps)
   .....................................
        dv=(v-v1)/15.0; v2=v+dv
     simpsrom=v2
   .....................................
  real*8 function simpseit(a,b,f,eps)
   .....................................
     if (n>2) then
        d2=v-v1; d1=v1-v2;
        de=d2**2/(d2-d1); dv=d2
     end if
  end do
```

```
  simpseit=v2
   ....................................
end module integrals

program testint
   use integrals
   real*8,external :: f
   real*8             u,y,z
   y=simps(0d0,1d0,f,1d-9)
   z=simpsrom(0d0,1d0,f,1d-9)
   u=simpseit(0d0,1d0,f,1d-9)
end program testint

real*8 function f(x)
   real*8 x
   f=exp(x)
end function f
```

В этих процедурах переменные `s1,s2,s4` обозначают суммы ординат, которые учитываются с весовыми коэффициентами 1, 2 и 4 соответственно. При каждом делении шага пополам заново вычисляется только `s4`; `s2` пересчитывается рекуррентно, а `s1` остается неизменной. Отметим необходимость задания числовых констант в виде, соответствующем спецификации формальных параметров (двойная точность).

Прогон программы дал следующие результаты:

Таблица 9.1. **Сопоставление программ интегрирования**

Прогон	Погрешность интегрирования		
	по Симпсону	по Ромбергу	по Эйткену
1	5.79e-4	5.79e-4	5.79e-4
2	3.70e-5	8.59e-7	3.70e-5
3	2.33e-6	1.38e-8	-4.40e-8
4	1.46e-7	2.16e-10	-6.92e-10
5	9.10e-9	3.39e-12	-1.08e-11
6	5.69e-10	5.26e-14	-1.69e-13

При отладке этой программы выяснилось, что "функциональные" процедуры должны быть либо внутренними для вызывающей программы (см. разд. 5.8.2), либо входящими в состав используемых модулей.

Описанные варианты не оптимальны в том смысле, что предполагают деление шага каждый раз на всем интервале интегрирования. Значительно выгоднее (по числу обращений к подинтегральной функции) дробить шаг только для участка, имеющего наибольшую оценку погрешности [5]. Такой подход используется в стандартных подпрограммах. Он резко усложняет программу и увеличивает потребность в памяти.

9.9. Решение дифференциальных уравнений

Рассмотрим задачу на динамику тела переменной массы — вертикальный подъем воздушного шара, увлекающего за собой сложенный на Земле трос. Масса воздушного шара меняется по закону

$$m = m_0 + \frac{\gamma}{g}h = (G_0 + \gamma h)/g,$$

где m_0 — масса воздушного шара без троса, γ — удельный (погонный) вес троса. Объем шара W, площадь поперечного (горизонтального) сечения S, а также ускорение силы тяжести g будем считать постоянными, а длину и прочность троса — достаточной. Плотность воздуха меняется по закону $\rho = \rho_0 e^{-\alpha h}$. Требуется рассчитать характеристики движения воздушного шара: скорость подъема v и высоту h как функции времени.

На шар действуют силы: подъемная (архимедова) $P = \rho g W$; сила тяжести $G = G_0 + \gamma h$ и сила сопротивления воздуха $R = C_x S \rho v^2/2$, направленная *против* движения шара. Здесь C_x — коэффициент лобового сопротивления. Составим дифференциальное уравнение в форме Мещерского для вертикального движения шара:

$$m\ddot{h} = -\dot{h}\,dm/dt - (G_0 + \gamma h) + gW\rho_0 e^{-\alpha h} - C_x S \rho_0 e^{-\alpha h}\,\dot{h}^2/2. \tag{9.1}$$

Здесь $dm/dt = d/dt\{(G_0 + \gamma h)/g\} = \gamma\dot{h}/g$, так как G_0, γ, g — постоянные. Поделив уравнение (9.1) на массу $m = (G_0 + \gamma h)/g$ и учтя, что $\ddot{h} = d\dot{h}/dt = dv/dt$, после элементарных преобразований получим

$$\begin{aligned} dv/dt &= -g + \{Wg^2\rho_0 e^{-\alpha h} - (\gamma + bG_0\rho_0 e^{-\alpha h})v|v|\}/(G_0 + \gamma h), \\ dh/dt &= v \end{aligned} \tag{9.2}$$

В соотношениях (9.2) учтена смена знака силы сопротивления воздуха при спуске шара, а множитель

$$b = \frac{C_x S}{2m_0} = \frac{C_x S g}{2G_0} \tag{9.3}$$

называется баллистическим коэффициентом. Первое уравнение (9.2) существенно нелинейно, так что задачу приходится решать численно. Условием окончания интегрирования выберем достижение максимальной высоты — получение первого отрицательного значения скорости V.

Значения числовых констант и их размерности (последние — в комментариях) будут указаны непосредственно в программе. Начальный вес шара считается как вес оболочки плюс вес полезного груза (10 н).

Для численного решения дифференциальных уравнений в стандартной форме

$$y' = f(x, y), \qquad y(0) = y_0$$

обычно используется метод Рунге-Кутты четвертого порядка, согласно которому

$$y(x + \Delta) = y(x) + (k_1 + 2k_2 + 2k_3 + k_4)/6.$$

Здесь

$$\begin{aligned} k_1 &= \Delta \cdot f(x, y), \\ k_2 &= \Delta \cdot f(x + \Delta/2, y + k_1/2), \\ k_3 &= \Delta \cdot f(x + \Delta/2, y + k_2/2), \\ k_4 &= \Delta \cdot f(x + \Delta, y + k_3). \end{aligned} \tag{9.4}$$

Построим программу решения задачи по схеме, приведенной в [13]. Программа состоит из четырех программных единиц:

- модуля drglob с глобальными описаниями производных типов и переменных;
- задающей уравнения внешней процедуры deqs;
- модуля drut со вспомогательными процедурами, включая описание метода Рунге-Кутты;
- головной программы maindif.

```
module drglob
   type vartype               ! Тип переменной с компонентами
      character(4) name       ! имя
      real inval              ! начальное значение
      real val                ! текущее значение
   end type vartype           ! Тип параметра с компонентами
   type partype               ! имя
      character(6) name       ! значение
      real val
   end type partype

   type(partype),allocatable,target ::parms(:)
   type (vartype),allocatable       ::vars(:)
   real,allocatable,target          :: x(:) ! Текущие
   real t,dt                                ! Время и шаг
   integer itime,runtime                    ! Счетчик
   integer numvars,numparms                 ! Количество
   character(1) opt                         ! Для диалога
end module drglob
```

Каждый элемент массива vars (или производного типа vartype) представляет свойства переменной в модели: имя, начальное значение и текущее значение. Это позволяет в любой момент продолжить счет с начального или с текущего значения. Массив parms представляет параметры модели.

Удобно иметь массив x с атрибутом target для представления текущих значений переменных модели, чтобы допускать алиасы в deqs. Остальные переменные описаны комментариями.

Пользователь задает дифференциальные уравнения модели во внешней процедуре deqs:

```
subroutine deqs(f)
 ! Вычисляет правые части дифференциальных уравнений,
 ! записанных в стандартной форме.
 ! Назначает алиасы, стыкуя содержательные обозначения
 ! со стандартными элементами расчетной схемы
 !
  use drglob
  implicit none
  real,intent(out) ::f(:)
  real,pointer     ::aa,alpha,bb,g,g0,gamma,h,v
  real erho
  h=>x(1); v=>x(2)
  g=>parms(1).val;      alpha=>parms(4).val
```

```
    gamma=>parms(8).val;  g0=>parms(11).val
    aa=>parms(13).val;    bb=>parms(14).val
    erh0=exp(-alpha*h)

  ! f(1)=dh/dt=v; f(2)=dv/dt

    f(1)=v
    f(2)=(aa*erho-(gamma+bb*erho*v*abs(v)) &
         /(g0+gamma*h)-g
 end subroutine deqs
```

Она возвращает правые части дифференциальных уравнений в массиве f. Приведем теперь инструментальный модуль:

```
module drut
   use drglob
   implicit none
   interface
      subroutine deqs(f)
         real f(:)
      end subroutine deqs
   end interface
contains
   subroutine headings
    ! Открывает файл для записи результатов счета
    ! Выводит заголовки для распечатки
    ! и начальные значения переменных
      integer i
      open(33,file='difeqs.res',status='new')
      write(33, '(3a14)',  &
           'time', (vars(i).name,i=1,numvars)
      write (33,'(3f14.2)') t,x
   end subroutine headings


   subroutine initial
   ! Задает количество переменных и параметров,
   ! запрашивает динамически выделяемую память,
   ! назначает имена и значения параметров и переменных
   ! К параметрам отнесены неперевычисляемые
   ! промежуточные значения
      numvars=2
      numparms=14
      allocate (vars(numvars), &
                parms(numparms),x(numvars))
      vars(1).name='h'
      vars(1).inval=0
      vars(2).name='v'
      vars(2).inval=0

      parms(1).name='g'       ! Ускорение свободного падения
      parms(2).name='pi'      !
```

```
   parms(3).name='rho(0)' ! Плотн. атмосферы на ур. моря
   parms(4).name='alpha'  ! Коэф. убывания плотности
   parms(5).name='C_x'    ! Коэф. лобового сопротивления
   parms(6).name='r'      ! Радиус шара
   parms(7).name='z'      ! Плотность ткани
   parms(8).name='gamma'  ! Погонный вес троса
   parms(9).name='S'      ! Площадь сечения шара
   parms(10).name='SS'    ! Площадь поверхности шара
   parms(11).name='G_0'   ! Вес шара
   parms(12).name='W'     ! Объем шара
   parms(13).name='aa'    !
   parms(14).name='bb'    ! Баллистический коэффициент

   parms(1).val=9.81      ! м/(с*с)
   parms(2).val=3.14159   !
   parms(3).val=1.225     ! кг/(м*м*м)
   parms(4).val=0.15e-3   ! 1/м
   parms(5).val= 5.5e-3   !
   parms(6).val= 1.5      ! м
   parms(7).val= 0.6      ! н/(м*м)
   parms(8).val= 0.12     ! н/м
   parms(9).val= parms(2).val*parms(6).val**2  ! м*м
   parms(10).val=4*parms(9).val                ! м*м
   parms(11).val=parms(7).val*parms(10).val+10 ! н
   parms(12).val=parms(10).val*parms(6).val/3  ! м*м*м
   parms(13).val=parms(3).val*parms(12).val    &
                *parms(1).val**2
   parms(14).val=parms(5).val*parms(9).val     &
                *parms(1).val*parms(3).val/2

   dt=1.0; t=0.0
   vars.val=vars.inval
end subroutine initial

subroutine run
! Непосредственно организует прогон:
   call headings                    ! распечатка заголовка
                                    ! и начальных значений
   do while(x(2)>=0)                ! Цикл по шагам
      t=t+dt
      call runge                    ! Шаг Рунге-Кутты
      write (33,'(3f14.2)') t,x ! Вывод результатов шага
   end do
   vars.val=x                       ! Обновление текущих
end subroutine run

subroutine runge
! 4-го порядка метод Рунге-Кутты
! Процедура DEQS вычисляет правые части уравнений
   real :: f(numvars)
   real,dimension(numvars) ::  a, b, c, d, e
```

```
      ! = k1,k2,k3,k4
      e=x                      ! обновление переменных
      call deqs(f); a=dt*f; e=v+a/2
      call deqs(f); b=dt*f; e=v+b/2
      call deqs(f); c=dt*f; e=v+c
      call deqs(f); d=dt*f
      x=e+(a+2.0*b+2.0*c+d)/6.0
   end subroutine runge


   subroutine tup
    ! Освобождает динамическую память,
    ! закрывает ранее открытый файл
      deallocate (vars,parms,x)
      close (33)
   end subroutine tup
end module drut
```

Наконец запишем головную программу:

```
  program maindif
   ! Управляет прогоном модели через диалог
     use drglob                    ! Глобальные объявления
     use drut                      ! Глобальные подпрограммы

     call initial                  ! Начальные значения
     print *, 'Run starting...'
     print *, ' '
     x=vars.inval
     call run                      ! Запуск интегрирования
     do
        print *, 'C: продолжение '
        print *, 'I: с начала'
        print *, 'Q: конец моделирования'
        print *,         ' '
        read *,opt
        select case(opt)
           case('Q','q')
              exit
           case('C','c')
              x=vars.val;
           case('I','i')
              x=vars.inval; t=0;
        end select
        call run                   ! счет
     end do
     call tup                      ! Осв. динам. памяти
  end program maindif
```

Анализ выведенных на печать результатов показывает, что шар достигает максимальной высоты 1220 м на 38-й секунде полета, после чего совершает медленно затухающие колебания относительно высоты 995 м с периодом около 65 секунд.

Обсуждаемая программа интересна как образец современного стиля программирования и источник многих полезных аналогий.

9.10. Математические библиотеки IMSL

В состав профессиональной версии FPS входят библиотеки IMSL, содержащие около тысячи программ в объектной форме. Большая часть процедур имеется в вариантах с одинарной и с двойной точностью. Каталоги библиотек можно просмотреть в соответствующем разделе систематической части `help`'а IMSL. Библиотек имеется две: прикладная математика и специальные функции; статистика. Для подготовки их использования в вызывающую программу включается оператор `use`, обеспечивающий интерфейс с членами библиотеки и контроль синтаксической корректности вызовов. Параметр `use` назначается "по потребности":

`msimsl` — полный состав библиотеки,

`msimslmd` — математические программы с двойной точностью,

`msimslms` — математические программы с одинарной точностью,

`msimslsd` — статистические программы с двойной точностью,

`msimslss` — статистические программы с одинарной точностью.

Упоминаемые в операторах вызова члены библиотеки подключаются к программе на этапе линкования.

Разделы библиотек начинаются со структурированного перечня входящих в него процедур. Для имен процедур используются стандартные соглашения, указанные в электронной документации. По каждой процедуре указываются: ее имя; назначение; обращение (форма вызова и список аргументов); спецификация аргументов (смысл и статус — входной, выходной, изменяемый); рекомендации по использованию и выделению дополнительной памяти; описание алгоритма со ссылками на источники более детальной информации; пример применения; результаты расчета примера.

Процедуры запрограммированы так, что возникновение ситуации "потеря значимости" (`underflow`) не влияет на их работу (при условии, что в регистр заносится нуль), а переполнение возникает лишь при некорректных исходных данных. Приведем каталоги основных библиотек IMSL:

Математическая

Линейные системы.
Задачи на собственные значения.
Интерполяция и аппроксимация.
Интегрирование и дифференцирование.
Дифференциальные уравнения.
Интегральные преобразования.
Нелинейные уравнения.
Оптимизация.
Основные матрично-векторные операции.

Специальные функции

Элементарные функции.
Тригонометрические и гиперболические функции.
Гамма-функция и связанные с ней.
Функция ошибок (нормальная плотность) и связанные с ней.
Функции Бесселя.
Функции Кельвина.
Функции Эйри.
Эллиптические интегралы.
Эллиптические и связанные с ними функции.
Функции распределения вероятностей и им обратные.
Функции Матье.
Разные функции.

Статистика

Основы статистики.
Регрессия.
Корреляция.
Анализ дисперсии.
Категорийный и дискретный анализ данных.
Непараметрическая статистика.
Критерии согласия и случайности.
Анализ временных рядов и прогнозирование.
Ковариации и факторный анализ.
Дискриминантный анализ.
Кластерный анализ.
Выборки.
Выживание и надежность.
Многомерное шкалирование.
Оценка плотностей и вероятностей.
Построчная принтерная графика.
Функции распределения вероятностей и им обратные.
Генерация случайных чисел.
Утилиты.
Математическая поддержка.

Для лучшего представления о содержательности разделов приведем характеристику процедур раздела "Интегрирование и дифференцирование":

Одномерное интегрирование

Общая адаптивная, особенность на границе.
Общая адаптивная.
Общая адаптивная, разрывы.
Общая, бесконечный интервал.
Адаптивная осциллирующая.
Адаптивная (Фурье).
Адаптивная с алгебраическим весом, разрывы.
Адаптивная взвешенная, главное значение по Коши.
Общая неадаптивная.

Многомерные квадратуры

Двумерная квадратура (повторный интеграл).
Адаптивная N-мерная квадратура по гиперпрямоугольнику.

Правила Гаусса и трехчленные рекурсии

Правило Гаусса для классических весов.
Правило Гаусса для рекуррентных коэффициентов.
Рекуррентные коэффициенты для классических весов.
Рекуррентные коэффициенты из квадратурного правила.
Квадратурное правило Фейера.

Дифференцирование

Аппроксимация первой, второй или третьей производной.

Несомненно, что названные библиотеки являются ценнейшим инструментом, использовать который может и должен каждый работающий на Фортране программист.

В заключение приведем пример программы вычисления уже обсуждавшегося в разделе 9.8 определенного интеграла с применением библиотеки IMSL.

```
program intsiml
   use msimslmd ! Только математика, двойная точность
   real*8        a,b,errabs,errel,errest,f,result,z
   external      f
   z=exp(1d0)-1d0 ! Точное значение интеграла
   a=0d0; b=1d0
   errabs=0d0; errel=1d-9
   call dqdags &
        (f,a,b,errabs,errel,result,errest)
   print *, 'result = ',result ! 1.718281828459045
   print *, 'errest = ',errest ! 1.907e-14
   print *, 'dres = ',result-z ! 0.0
end program intsiml

real*8 function f(x)
   real*8 x
   real*8, intrinsic :: dexp
   f=dexp(x)
end function f
```

Вызываемая процедура реализует адаптивную схему вычисления интеграла по 21-точечному правилу Гаусса–Кронрода на каждом подынтервале разбиения исходного участка и допускает наличие сингулярностей на границах участка интегрирования. Имена фактических параметров совпадают с именами формальных. Из них `errest` есть оценка абсолютной погрешности результата, а смысл остальных очевиден. В данной программе поучительна работа с данными двойной точности (в частности, использование специфического имени функции вычисления экспоненты).

9.11. Numerical Recipes

В профессиональной работе с математическими библиотеками часто возникает желание заглянуть "внутрь" используемых процедур в целях

- их адаптации к отдельным классам задач,
- улучшения их характеристик (расширения области применения, уменьшения трудоемкости, повышения точности),
- выяснения причин отклонений результатов от ожидаемых,

наконец, из педагогических соображений и просто по здоровой любознательности. Такую возможность дают входящие в состав FPS "Численные рецепты" (Numerical Recipes). В директории `FPS4\numrec\nrf207` имеется гипертекстовый справочник, содержащий тексты 350 процедур и 250 примеров применения на Фортране-77 (напомним, что составленные на нем программы совместимы снизу вверх с Ф90). Все процедуры написаны в расчете на стандартную точность, но в пакет входит процедура `NR2DP`, формально преобразующая их к двойной точности[2]. В сопровождающей "Рецепты" подборке выдержек из рецензий книга с рецептами (издания Кембриджского университета) аттестуется как "воистину золотая жила" и "классический источник", доставляющий "пользу, удовольствие и радость" всем программирующим на Фортране и содержащий массу ценных советов и деталей. К сожалению, упомянутая книга поставляется на отдельной дискете, а в электронной документации собственно FPS ни к процедурам, ни к вызывающим их программам нет *никаких* комментариев. Это заметно ограничивает "удовольствие и радость" от "Рецептов", но польза остается несомненной.

Работать с "Рецептами" удобно через файл `Numerical Recipes`, раздел `Subroutines and Functions by Chapter`. Приведем соответствующее оглавление:

Разное (вплоть до астрологии).
Решение линейных алгебраических уравнений.
Интерполяция и экстраполяция.
Интегрирование функций.
Вычисление функций (аппроксимации).
Специальные функции.
Случайные числа (различные законы распределения).
Сортировки.
Решение уравнений и систем уравнений.
Минимизация и максимизация функций (одномерный поиск,
 линейное и нелинейное программирование).
Задачи на собственные значения.
Быстрое преобразование Фурье.
Спектральный анализ.
Обработка статистики.
Выравнивание экспериментальных данных.
Обыкновенные дифференциальные уравнения.
Двухточечные краевые задачи.
Интегральные уравнения и обратные преобразования.
Дифференциальные уравнения в частных производных.
Нечисловые задачи (главным образом помехоустойчивые коды).

[2] Следует считаться с возможной недостаточностью этого средства, когда для достижения высокой точности потребуется принципиально иной алгоритм.

Сопоставление содержимого разделов с IMSL показывает, что библиотеки взаимно дополняют друг друга. Это определяет целесообразность использования их обеих.

Выбрав в каталоге раздела нужный файл, его можно скопировать в нужную директорию (в панели инструментов `Recipes` имеется кнопка `Set Path`), включить в проект и откомпилировать. Запасать продукты такой компиляции в библиотеке нет смысла, поскольку многие процедуры в связи с отсутствием в Фортране-77 динамических массивов перед использованием нуждаются в настройке параметров.

Глава 10.

Пакет научной графики

10.1. Знакомство с пакетом

Пакет научной графики SciGraph позволяет строить по данным, предварительно занесенным в файл, графики различного вида: в декартовых и полярных координатах, столбчатые, с допусками и т.п. Имеются богатые возможности выбора вида осей, характера линий, оцифровки шкал, цветовых решений (для дисплея). Обзор технологии его использования имеется в файле `scigraph.wri`. Пакет работает в графическом подмножестве FPS — QuickWin.

Знакомство хорошо начать с демонстрационной программы. Для этого достаточно войти в каталог `d:\fps4\help\samples\fps\general\scigraph` и запустить `sgdemo.exe`. Каждый тип графика будет представлен в отдельном окне, активное окно может быть развернуто на весь экран при нажатии [Alt+Enter] (выход — по [Esc]).

Для использования SciGraph в прикладной программе нужно

1) Обеспечить, чтобы линкеру был указан путь к библиотеке `scigraph.lib` через переменную окружения `LIB` или дополнительными средствами MDS.

2) Обеспечить, чтобы компилятору был указан путь к модулю `scigraph.mod`. Компилятор использует для этого переменную окружения `INCLUDE`.

3) Вставить предложение `use scigraph` во все программные единицы, которые используют SciGraph.

4) Вызвать, как описано ниже, требуемые функции SciGraph.

Все процедуры пакета оформлены в виде функций с побочным эффектом. В левой части оператора вызова функции обычно указывается целая переменная `retv`, запоминающая код возврата (степень серьезности ошибок отработки).

Выполнение графика проходит четыре фазы:

1) Вставить в программную единицу предложение `use scigraph`.

2) Собрать или рассчитать исходные данные и записать их в виде массива.

3) Задать стандартным образом установки о графике в целом, осях, линиях, надписях и т.п., а затем скорректировать их применительно к решаемой задаче.

4) Запустить графическую библиотеку.

5) Дать команду построения графика с необходимыми установками.

Данные должны храниться в массивах. Каждый ряд данных должен быть полностью определен отдельным массивом или сечением общего массива. Единственное исключение — когда x-значения текстовые (`labels`). Тогда текстовые данные должны храниться в одном массиве, а все числовые — в другом (обязательно формата `real*4`). Примерами текстовых данных являются, например, названия месяцев или географических регионов.

Установки задаются

1) для графика в целом (положение, обрамление, название, тип, цвет фона);

2) для отдельных рядов данных (название, тип линий и маркеры точек, количество точек, диапазон данных);

3) для осей (тип шкалы, диапазон данных, положение осей, засечки, сетка).

Построение графика проходит две фазы. Вызов `PlotGraph` открывает окно, рисует рамку, заполняет фон, размечает оси, дает наименования осям и графику в целом. На второй фазе вызывается одна из подпрограмм с именем вида `Plot*Data`, которая отображает данные.

10.2. Константы SciGraph

Константы пакета `SciGraph` собраны в файле `sgdata.f90`. На внутреннем уровне они получают конкретные (целые, в некоторых случаях текстовые) значения. Для наших целей будет достаточно знать смысл и имена наиболее употребительных из них, а также структуру необходимых установок. Обозначения объектов `SciGraph` начинаются знаком доллара и последующими минимум двумя заглавными буквами.

Прежде всего определим варианты задания базовых элементов графиков:

Таблица 10.1. **Цвета 16-элементной палитры**

$CIBLACK	черный	$CIGRAY	серый
$CIBLUE	синий	$CILIGHTBLUE	голубой
$CIGREEN	зеленый	$CILIGHTGREEN	салатный
$CICYAN	бирюзовый	$CILIGHTCYAN	светлобирюзовый
$CIRED	красный	$CILIGHTRED	светлокрасный
$CIMAGENTA	фиолетовый	$CILIGHTMAGENTA	сиреневый
$CIBROWN	коричневый	$CIYELLOW	желтый
$CIWHITE	белый	$CIBRIGHTWHITE	ярко-белый

Таблица 10.2. **Маркеры данных**

$MKNONE	нет маркеров	$MKX	косые крестики
$MKSQUARE	квадраты	$MKFISQUARE	заполненные квадраты
$MKTRIANGLE	треугольники	$MKFITRIANGLE	заполненные треугольники
$MKDIAMOND	ромбы	$MKFIDIAMOND	заполненные ромбы
$MKCIRCLE	кружки	$MKFICIRCLE	заполненные кружки
$MKPLUS	плюсы		

Таблица 10.3. **Типы линий**

$LTNONE	нет линий
$LTTHICKSOLID	толстая сплошная
$LTSOLID	сплошная
$LTTHICKDASH	толстая штриховая
$LTDASH	штриховая
$LTTHICKDASHDOT	толстый штрих-пунктир
$LTDASHDOT	штрих-пунктир
$LTTHICKDASHDOTDOT	толстые тире-точка-точка
$LTDASHDOTDOT	тире-точка-точка
$LTTHICKDOT	жирный пунктир
$LTDOT	пунктир

Поля ошибок имеют обозначения соответственно `$EBNONE` — отсутствуют, `$EBTHIN` — тонкие, `$EBTHICK` — толстые.

Положение осей задается одним из следующих вариантов: `$APNONE` — без осей, `$APBOTTOM` — снизу, `$APTOP` — сверху, `$APLEFT` — слева, `$APRIGHT` — справа.

Засечки на осях определяются командами `$TTNONE` — засечки отсутствуют, `$TTOUTSIDE` — снаружи, `$TTINSIDE` — изнутри, `$TTBOTH` — по обе стороны.

Таблица 10.4. **Построение сетки**

$GSNONE	сетки нет	$GSZERO	линия сетки в нуле
$GSMAJOR	по большим засечкам	$GSZEROMAJOR	в нуле и больших
$GSBOTH	по большим и малым	$GSZEROBOTH	в нуле и всех засечках

Таблица 10.5. **Типы шрифтов**

$FTCOUR	courier	$FTTROMANSM	малый roman
$FTCOURSM	малый courier	$FTSANSSERIF	SANSSERIF (без засечек)
$FTTROMAN	roman	$FTSANSSERIFSM	малый SANSSERIF

10.3. Структуры установок

Теперь мы можем в терминах введенных в предыдущем разделе констант обсудить структуры установок.

Установки графика

```
STRUCTURE /PrivateGraphSettings/
 CHARACTER*40    title        ! название графика
 INTEGER         titleFont    ! шрифт названия: $FTCOUR,$FTCOURSM,...
 INTEGER         titleColor   ! цвет названия
 CHARACTER*40    title2       ! второе название
 INTEGER         title2Font   ! шрифт второго названия: $FTCOUR,...
 INTEGER         title2Color  ! цвет его  (0..numcolors)
 INTEGER         graphType    ! вид графика: $GTBAR,$GTLINE,...
 INTEGER         graphColor   ! цвет границы графика
 INTEGER         graphBgColor! цвет фона
 INTEGER         x1,y1,x2,y2 ! физические границы экрана
```

```
 LOGICAL          setGraphMode! если устанавливается графическая мода
Три следующих работают, если вызваны Get*Default процедуры
 INTEGER          numSetsReq  ! число требуемых установок данных
                              ! (автоматический выбор цвета)
 INTEGER          numXAxesReq ! требуемое число осей X
 INTEGER          numYAxesReq ! требуемое число осей Y
Три следующих  PlotGraph и  Plot*Data устанавливают  всегда
 INTEGER          didPlotGraph! 1 при успешном построении графика
 INTEGER          numSets     ! число используемых наборов данных
 INTEGER          numSetsDone ! число обработанных наборов
END STRUCTURE
```

Установки данных

```
STRUCTURE /DataSettings/
 CHARACTER*20 title          ! название (легенда) для ряда данных
 INTEGER      titleFont      ! шрифт названия: $FTCOUR,$FTCOURSM,...
 INTEGER      titleColor     ! цвет шрифта
 INTEGER      markerType     ! вид маркера: $MKNONE,$MKSQUARE,...
 INTEGER      markerColor    ! цвет маркера
 INTEGER      lineType       ! тип линии: $LTNONE, $LTSOLID,...
 INTEGER      lineColor      ! цвет линии
 INTEGER      barType        ! тип линий "ошибок": $BTNONE,$BTSOLID,..
 INTEGER      barColor       ! цвет "ошибок"
 INTEGER      errorbarType   ! стиль "ошибок": $EBNONE, $EBTHIN,...
 INTEGER      errorbarColor! цвет линий "ошибок"
 INTEGER      dataType       ! #,# или текст,# ?  $DTTEXT, $DTNUM
 LOGICAL      xFirst         ! x записан перед y в той же строке
 INTEGER      numPoints      ! число точек или меток (bar/line)
 INTEGER      numElements    ! число элементов на точку (2 или 3)
 REAL*8       xLowVal        ! нижняя граница x
 REAL*8       xHighVal       ! верхняя граница x
 REAL*8       yLowVal        ! нижняя граница y
 REAL*8       yHighVal       ! верхняя граница y
END STRUCTURE

STRUCTURE /AxisSettings/
 CHARACTER*25 title             ! надпись для оси
 INTEGER      titleFont         ! шрифт: $FTCOUR, $FTCOURSM,...
 INTEGER      titleColor        ! цвет надписи
 INTEGER      axisFont          ! шрифт для оцифровки оси
 LOGICAL      axisFontRotated !.TRUE., если числа вращаются(для оси y)
 INTEGER      axisPos           ! положение оси: $APBOTTOM, $APTOP,...
 INTEGER      axisType          ! вид оси: $ATX, $ATY,...
 INTEGER      axisColor         ! цвет линии оси
 INTEGER      axisFunc          ! функция по оси: $AFNONE, $AFLOG10,...
 LOGICAL      axisCW            ! если полярная ось по часовой стрелке
 REAL*8       lowVal            ! начальное значение по оси
 REAL*8       highVal           ! конечное значение по оси
 REAL*8       increment         ! шаг по оси между большими засечками
 INTEGER      numDigits         ! число изображаемых десятичных цифр
 INTEGER      tickType          ! тип засечки: $TTNONE, $TTINSIDE,...
```

```
 INTEGER      tickColor        ! цвет больших засечек
 INTEGER      minorTickColor   ! цвет малых засечек
 INTEGER      tickRatio        ! число малых засечек на одну большую
                               !  1= все большие (||||),
                               ! 2=(|.|), 3=(|..|),4= |...| и т.д.
 INTEGER      gridStyle        ! частота сетки: $GSNONE, $GSMAJOR,...
 INTEGER      gridLineType     ! линии сетки: $LTNONE, $LTSOLID,...
 INTEGER      gridColor        ! цвет линий сетки
END STRUCTURE
```

10.4. Типы данных и осей

Типы данных и графиков обозначаются следующим образом:

```
$DTNUM             ! данные типа число-число
$DTTEXT            ! данные типа текст-число
$GTBAR             ! столбчатый график (по оси x - текст,
                   ! по оси y - число),
$GTLINE            ! линейчатый график (по оси x - текст,
                   ! по оси y - число),
$GTLINEWERRBAR     ! то же - с "ошибками",
$GTXY              ! числовой декартов график,
$GTXYWERRBAR       ! то же с "ошибками",
$GTPOLAR           ! числовой график в полярных координатах.
```

Если среди данных имеются текстовые, то допустимо использование только `$GTBAR`, `$GTLINE`, `$GTLINEWERRBAR`. Для чисто числовых графиков следует применять `$GTXY`, `$GTXYWERRBAR`, `$GTPOLAR`. Под "ошибками" (`errorbars`) понимаются вертикальные отрезки, указывающие возможные погрешности используемых значений в выбранном масштабе. Распределение ошибки предполагается симметричным относительно опорного значения, каждая точка графика задается двумя координатами и размахом возможной ошибки.

Ряд данных есть совокупность точек, соотнесенных друг с другом, использующих единую установку и печатаемых совместно. График может иметь несколько рядов данных. Данные для `xy`-графика с тремя рядами данных по 7 точек в каждом должны быть представлены в форме

```
real*4 abc (2,7,3)
data abc /x1,y1, x2,y2, x3,y3,.....x7,y7, ! Ряд 1
          x1,y1, x2,y2, x3,y3,.....x7,y7, ! Ряд 2
          x1,y1, x2,y2, x3,y3,.....x7,y7/ ! Ряд 3
```

Напомним, что массивы в Фортране заполняются в порядке следования измерений. Таким образом, экстент по первому измерению должен быть равным двум (с учетом ошибок — трем), по второму — количеству точек кривой, по третьему — числу кривых.

Покажем подготовку данных для столбчатого графика с четырьмя рядами данных по три точки в каждом:

```
real*4        arrnum (3,4)
character*6   arrlabel(3)
data arrnum /y1,y2,y3, ! Ряд 1
```

```
                y1,y2,y3, ! Ряд 2
                y1,y2,y3, ! Ряд 3
                y1,y2,y3/ ! Ряд 4
     data arrlabel /'bars 1', 'bars 2', 'bars 3'/
```

Массив для двух рядов по 4 точки в каждом "с ошибками":

```
real*4 errs (3,4,2)
data errs /x1,y1,yerr1, x2,y2,yerr2, x3,y3,yerr3, ! Ряд 1
           x1,y1,yerr1, x2,y2,yerr2, x3,y3,yerr3/ ! Ряд 2
```

Для работы с массивами, содержащими более одного ряда, обязательно использование при настройке `GetMultiDataDefaults` или `GetLabelMultiDataDefaults`. Если нужные значения были получены с помощью другой программы или соответствуют экспериментальным данным, то массив должен быть заполнен в процессе считывания из файла.

Типы осей определяются параметрами

`$AFNONE, $AFLINEAR`	! без преобразования значений,
`$AFLOG10`	! десятичный логарифм,
`$AFLOG`	! натуральный логарифм,
`$AFLOG2`	! двоичный логарифм.

10.5. Вызывающая последовательность

Приводимые ниже операторы вызова должны выполняться строго в указанной последовательности — иначе `SciGraph` будет работать неправильно:

```
GetGraphDefaults
GetLabel[Multi]DataDefaults
GetAxis[Multi]Defaults
PlotGraph
PlotLabel[Multi]Data
```

Фрагмент `Multi` включается только при использовании нескольких рядов данных. Предложение `GetAxis` записывается отдельно для каждой оси. При работе с чисто числовыми данными фрагменты `Label` исключаются.

10.6. Установка основных параметров

10.6.1. График в целом

Функция `GetGraphDefaults` задает для графика в целом наиболее вероятные уставки, которые затем могут быть изменены пользователем. Типы графиков перечислялись в разд. 10.4. Приведем интерфейс с этой функцией:

```
interface
   integer function GetGraphDefaults(graphType,graph)
      integer graphType          ! вход
      type(GraphSettings) graph ! выход
   end integer function GetGraphDefaults
end interface
```

Пример типичного вызова:

```
retv=GetGraphDefaults($GTXY,myXYGraph)
```

Он заполнит структуру myXYGraph разумными значениями для графика XY. Эти значения потом могут быть скорректированы применительно к конкретному случаю. Обычная вещь — коррекция размера и положения графика на экране установкой координат левого верхнего и правого нижнего углов:

```
retv=GetGraphDefaults($GTXY,myXYGraph)
myXYGraph.x1=10
myXYGraph.y1=20
myXYGraph.x2=200
myXYGraph.y2=300
```

Напомним, что начало координат расположено в левом верхнем углу экрана и координаты измеряются в пикселах. Нужно знать размер вашего экрана (типичный вариант — 640×480).

10.6.2. Ряды данных

Эти установки включают цвета маркеров, линий и "ошибочных" отрезков, типы маркеров и линий, минимальные и максимальные значения по оси, название ряда. Используется также количество точек в ряду (numPoints, numLabels). Пример:

```
retv=GetDataDefaults(graph,7,data,dSettings)
dSettings.markerType=$MKFISQUARE
dSettings.lineType=$LTDOT
dSettings.lineColor=$CIGREEN
```

Данные будут нанесены заполненными квадратами и соединены пунктирными линиями зеленого цвета.

Приведем интерфейс для числового графика:

```
interface
   integer function &
      GetDataDefaults(graph,numPoints,data,dSettings)
      type(GraphSettings) graph     ! вход/выход
      integer             numPoints ! вход (к-во точек)
      real*4              data(1)   ! вход (факт. данные)
      type(DataSettings)  dSettings ! выход
   end integer function GetDataDefaults
   integer function &
      GetMultiDataDefaults(graph,numPoints,data, &
                           numSettings,dSettingsArray)
      type(GraphSettings)   graph       ! вход
      integer               numPoints   ! вход (к-во точек)
      real*4                data(1)     ! вход (факт. данные)
      integer*2             numSettings ! вход
      record(DataSettings)  dSettingsArray(numSettings)!выход
   end integer function GetMultidataDefaults
end interface
```

При работе с текстовыми значениями:

```
interface
   integer function &
   GetLabelDataDefaults(graph,numLabels,labels,data,dSettings)
      type(GraphSettings) graph          ! вход
      integer             numLabels      ! вход (к-во точек)
      character*(*) labels(numLabels)    ! вход
      real*4              data(1)        ! вход (факт. данные)
      type(DataSettings)  dSettings      ! выход
   end integer function GetLabelDataDefaults
   integer function &
   GetLabelMultiDataDefaults(graph,numLabels,labels,data, &
                             numSettings,dSettingsArray)
      type(GraphSettings)   graph        ! вход
      integer               numLabels    ! вход (к-во точек)
      character*(*) labels(numLabels)    ! вход
      real*4                data(1)      ! вход (факт. данные)
      integer               numSettings ! вход
      type(DataSettings)    dSettingsArray(numSettings) ! выход
   end integer function GetLabelMultidataDefaults
end interface
```

Спецификация `character*(*)` соответствует массиву строк, длина элементов и численность компонент которого заимствуются у фактического параметра функции.

10.6.3. Координатные оси

Функция `GetAxisMultiDefaults` работает с *массивом* установок и определяет максимальное и минимальное значения и шаг по оси. Функция `GetAxisDefaults` делает то же для отдельной оси. Параметр `axisType` ∈ {`$ATX, $ATY`} для декартовых графиков, `axisType` ∈ {`$ATR, $ATTHETA`} — для полярных. Функции преобразования координат задаются из множества {`$AFLINEAR, $AFLOG, $AFLOG2, $AFLOG2, $AFLOG10`}. Пример вызова с дополнительной настройкой:

```
retv=GetAxisDefaults(graph,dSettings,$ATX,$AFLINEAR,axisFunc,axis)
axis.gridStyle=$GTMAJOR
axis.gridColor=$CIBLUE
axis.tickColor=$CIRED
axis.minorTickColor=$CIGREEN
```

По его результатам `x`-ось будет иметь сетку синей, большие засечки красными, малые — зелеными. Интерфейс к названным функциям:

```
interface
   integer function &
      GetAxisDefaults(graph,dSettings,axisType,axisFunc,axis)
      type(GraphSettings) graph          ! вход
      type(DataSettings)  dSettings      ! вход
      integer             axisType       ! вход ($ATX, $ATY)
      integer             axisFunc       ! вход ($AFNONE, $AFLOG,...)
      type(AxisSettings)  axis           ! выход
   end integer function GetAxisDefaults
   integer function &
   GetAxisMultiDefaults(graph,numSettings,dSettingsArray, &
```

```
                                  axisType,axisFunc,axis)
      type(GraphSettings) graph          ! вход
      integer             numSettings    ! вход
      type(DataSettings)  dSettingsArray(numSettings)  ! вход
      integer             axisType       ! вход ($ATX, $ATY)
      integer             axisFunc       ! вход ($AFNONE, $AFLOG,...)
      type(AxisSettings)  axis           ! выход
   end integer function GetAxisMultiDefaults
end interface
```

10.7. Рисование графика

10.7.1. Основа графика

Функция PlotGraph рисует окаймление графика, подпись и оси в соответствии с заданными установками. На декартовом графике может быть до четырех окаймляющих его осей (как правило, параллельные оси используются с различными единицами измерения). Полярный график имеет две оси. Интерфейс к функции:

```
    interface
       integer function &
       PlotGraph(graph,NumAxes,axisArray,numSettings)
          type(GraphSettings) graph                ! вход
          integer             numAxes              ! вход
          type(AxisSettings)  axisArray(numAxes)   ! вход
          integer             numSettings          ! вход
       end integer function PlotGraph
    end interface
```

10.7.2. Собственно графики

Если все данные собраны в единственном массиве, то вместо многократного вызова PlotData можно однократно использовать PlotMultiData. Хотя декартов график может иметь до четырех осей, для него нужно определить две оси (сначала это делается обязательно для оси x). Типичный пример вызова:

```
retv=PlotData(graph,data,dSettings,xaxis,yaxis)
```

Интерфейс для этих функций:

```
interface
   integer function &
         PlotData(graph,data,dSettings,axis1,axis2)
      type(GraphSettings) graph       ! вход
      real*4              data(1)     ! вход (факт. данные)
      type(DataSettings)  dSettings   ! вход
      type(AxisSettings)  axis1       ! вход
      type(AxisSettings)  axis2       ! вход
   end integer function PlotData
   integer function &
         PlotMultiData(graph,data,numSettings,    &
                       dSettingsArray,axis1,axis2)
      type(GraphSettings) graph         ! вход
      real*4              data(1)       ! вход (факт. данные)
```

```
      integer             numSettings  ! вход
      type(DataSettings)  dSettingsArray(numSettings) ! вход
      type(AxisSettings)  axis1        ! вход
      type(AxisSettings)  axis2        ! вход
   end integer function PlotMultiData
end interface
```

Типичный вызов PlotLabelData:

```
      retv=PlotLabelData(graph,labels,data,dSettings,xaxis,yaxis)
```

Соответственно для множественных данных обсуждаемого вида

```
retv=PlotLabelMultiData(graph,labels,data,2,dSettings,xaxis,yaxis)
```

Интерфейс для функций, печатающих графики с символьными значениями по оси абсцисс:

```
interface
   integer function &
   PlotLabelData(graph,labels,data,dSettings,axis1,axis2)
      type(GraphSettings) graph         ! вход
      character*(*)       labels(1)     ! вход
      real*4              data(1)       ! вход (факт. данные)
      type(DataSettings)  dSettings     ! вход
      type(AxisSettings)  axis1         ! вход
      type(AxisSettings)  axis2         ! вход
   end integer function PlotLabelData
   integer function                                    &
   PlotLabelMultiData(graph,labels,data,numSettings, &
                       dSettingsArray,axis1,axis2)
      type(GraphSettings) graph          ! вход
      character*(*)       labels(1)      ! вход
      real*4              data(1)        ! вход (факт. данные)
      integer             numSettings    ! вход
      type(DataSettings)  dSettingsArray(numSettings) ! вход
      type(AxisSettings)  axis1          ! вход
      type(AxisSettings)  axis2          ! вход
   end integer function PlotLabelMultiData
end interface
```

10.8. Коды возврата

Каждая функция SciGraph сообщает об успехе своего выполнения:

```
      $GEOK           ! ошибок нет
      $GEGRAPHMODE    ! не войти в графическую моду
      $GEFONT         ! не установить шрифты
      $GEGRAPH        ! плохая информация о графике
      $GEAXIS         ! плохая информация об осях или функциях
      $GEDATA         ! ошибка в данных или установках
      $GEDIMS         ! ошибка в размерности массивов
      $GEPARAMS       ! ошибка в параметрах
      $GENOPLOTGRAPH  ! невозможно построить график
```

Частой причиной ошибок является неверная последовательность вызова функций SciGraph.

10.9. Пример применения Scigraph

Покажем наиболее типичные научно-технические приложения Scigraph. Построим в *линейном* масштабе графики плотностей гамма-распределения длительности обслуживания заявок

$$f(t) = \frac{\mu(\mu t)^{\alpha-1}}{\Gamma(\alpha)} e^{-\mu t}$$

для четырех значений α при среднем значении $b = \alpha/\mu = 1$ и в *полулогарифмическом* — стационарные распределения числа заявок в системе $M/G/1$ с простейшим входящим потоком интенсивности $\lambda = 0.8$ и упомянутыми распределениями длительности обслуживания. Последняя задача решается в два этапа. Сначала вычисляются вероятности прибытия ровно j заявок за время обслуживания

$$q_j(t) = \int_0^\infty \frac{(\lambda t)^j}{j\,!} e^{-\lambda t} f(t)\,dt, \quad j = 0, 1, \ldots$$

по рекуррентным формулам

$$\begin{aligned} q_0 &= \left(\frac{\mu}{\lambda+\mu}\right)^\alpha, \\ q_j &= q_{j-1}\frac{\lambda}{\lambda+\mu}\frac{\alpha+j-1}{j}, \quad j = 1, 2, \ldots \end{aligned}$$

Затем опять же рекуррентно рассчитываются стационарные вероятности:

$$\begin{aligned} p_0 &= 1 - \lambda b, \\ p_j &= \left(p_{j-1} - p_0 q_{j-1} - \sum_{i=1}^{j-1} p_i q_{j-i}\right) \Big/ q_0, \quad j = 1, 2, \ldots \end{aligned}$$

Эти два этапа оформлены в виде отдельных подпрограмм. Ниже представлена программа решения поставленной задачи.

```
MODULE DemSciGraph
   USE MSFLIB
   USE SCIGRAPH
CONTAINS

   SUBROUTINE DemoStart
      integer i
      open(10,file='user',title='XY Graph')
      call gamdens
      open(40,file='user',title='Logarithmic XY Graph')
      call probabilities
```

```
   i = setexitqq(QWIN$EXITPERSIST)
END SUBROUTINE

SUBROUTINE gamdens
   RECORD /GraphSettings/ GamD
   RECORD /DataSettings/ DSets(4) ! 4 набора данных
   RECORD /AxisSettings/ Axes(2)  ! 2 оси : y,x

   REAL*4       Dens(2,11,4)      ! 4 набора данных по 11 точек
   CHARACTER*20 Legends(4)        ! 4 легенды

   INTEGER      retcode
   RECORD /windowconfig/ wc
   INTEGER      i,j
   REAL         a(4),g(4),pi,s,t

   pi=3.14159
   a(1)=0.5; a(2)=1.0;
   a(3)=1.5; a(4)=3.0
   g(1)=sqrt(pi); g(3)=g(1)/2.0
   g(2)=1e0;      g(4)=2e0
   t=0.2
   do i=2,11
      do j=1,4
         Dens(1,i,j)=t
         s=a(j)*t
         Dens(2,i,j)=&
         a(j)*exp((a(j)-1)*log(s)-s)/g(j)
      end do
      t=t+0.2
   end do
   do j=3,4; Dens(:,1,j)=0; end do
   Dens(1,1,2)=0; Dens(2,1,2)=1.0 ! Экспонента - в нуле
   t=0.05;         Dens(1,1,1)=t
   Dens(2,1,1)=&
      a(1)*(a(1)*t)**(a(1)-1)*exp(-a(1)*t)/g(1)

   DATA Legends / 'a=0.5','a=1.0','a=1.5','a=3.0'/
   if(.not.GetWindowConfig(wc)) stop 'Window Not Open'

   retcode=GetGraphDefaults($GTXY,GamD)
   GamD.setGraphMode=.FALSE.
   GamD.x2=510; GamD.y2=380  ! Размеры графика в пикселах
   GamD.graphBgColor=$CIBRIGHTWHITE ! Белый фон
   GamD.graphColor=$CIBLACK        ! Черное обрамление
   GamD.title='Gamma-distribution densities'
   GamD.titleColor=$CIBLUE

   retcode=GetMultiDataDefaults(GamD,11,Dens,4,DSets)
   DSets(1).lineType=$LTDOT         ! Пунктир
   DSets(2).lineType=$LTSOLID       ! Сплошная
```

```
   DSets(3).lineType=$LTDASH          ! Штриховая
   DSets(4).lineType=$LTDASHDOT       ! Штрих-пунктир

   DSets(1).lineColor=$CIBLUE         !
   DSets(2).lineColor=$CIRED          !
   DSets(3).lineColor=$CIGREEN        !> цвета линий
   DSets(4).lineColor=$CIBROWN        !

   DSets(1).markerType=$MKSQUARE      ! квадрат
   DSets(2).markerType=$MKTRIANGLE    ! треугольник
   DSets(3).markerType=$MKDIAMOND     ! ромб
   DSets(4).markerType=$MKPLUS        ! плюс

   DSets(1).markerColor=$CIBLUE       !
   DSets(2).markerColor=$CIRED        !
   DSets(3).markerColor=$CIGREEN      !> цвета маркеров
   DSets(4).markerColor=$CIBROWN      !

   DSets(1).titleColor=$CIBLUE        !
   DSets(2).titleColor=$CIRED         !
   DSets(3).titleColor=$CIGREEN       !> цвета легенды
   DSets(4).titleColor=$CIBROWN       !
   DSets.xHighVal=2.0
   do i=1,4
      DSets(i).title=Legends(i)
   end do

   retcode=GetAxisMultiDefaults(GamD,4,DSets,  &
      $ATX,$AFLINEAR,Axes(1))         ! Ось абсцисс
   Axes(1).title='Time t'
   Axes(1).tickType=$TTINSIDE         ! Засечки изнутри
   Axes(1).gridStyle=$GSZERO          ! Без сетки
   Axes(1).axisColor=$CIBLACK         ! Черная
   Axes(1).axisPos=$APBOTTOM          ! Снизу
   Axes(1).numDigits=2                ! Два знака после точки

   retcode=GetAxisMultiDefaults(GamD,4,DSets,  &
      $ATY,$AFLINEAR,Axes(2))         ! Ось ординат
   Axes(2).tickType=$TTINSIDE
   Axes(2).title='Density gamma(t)'
   Axes(2).gridStyle=$GSZERO
   Axes(2).axisColor=$CIBLACK
   Axes(2).axisPos=$APLEFT            ! Слева
   Axes(2).numDigits=2

   retcode=PlotGraph(GamD,2,Axes,4)
   retcode=PlotMultiData(GamD,Dens,4,DSets,Axes(1),Axes(2))
END SUBROUTINE

SUBROUTINE probabilities
   INTERFACE
```

```
   subroutine qgam(lam,a,mu,jmax,q)
      real lam,a,mu,q(0:jmax)
      integer jmax
   end subroutine qgam
   subroutine pmg1(lam,b,jmax,q,p)
      real lam,b,q(0:),p(0:)
      integer jmax
   end subroutine pmg1
END INTERFACE

RECORD /GraphSettings/ LGR
RECORD /DataSettings/  ldsets(4) ! 4 набора данных
RECORD /AxisSettings/  laxes(2)  ! 2 оси
REAL*4         probs(2,0:20,4)   ! 4 набора по 21 паре
CHARACTER*20 loglegs(4)          ! Легенды
INTEGER        i,j,jmax,retcode
REAL           a(4),mu(4),b(4),q(0:20),lam
RECORD /windowconfig/ wc
DATA a /0.5, 1.0, 1.5, 3.0/
DATA loglegs / 'a=0.5','a=1.0','a=1.5','a=3.0'/
lam=0.8; jmax=20

do i=1,4
   mu(i)=a(i); b(i)=a(i)/mu(i)
   call qgam (lam,a(i),mu(i),jmax,q)
   call pmg1(lam,b(i),jmax,q,probs(2,:,i))
   do j=0,jmax; probs(1,j,i)=j; end do
end do

if( .not. GetWindowConfig(wc) ) stop 'Window Not Open'

retcode=GetGraphDefaults($GTXY,LGR)
LGR.setGraphMode = .FALSE.
LGR.x2=510; LGR.y2=380
LGR.graphBgColor=$CIBRIGHTWHITE
LGR.graphColor=$CIBLACK
LGR.titleColor=$CIBLUE
LGR.title='Distribution of Demands in M/G/1 system'

retcode=GetMultiDataDefaults(LGR,21,probs,4,ldsets)
ldsets(1).lineType=$LTDOT
ldsets(2).lineType=$LTSOLID
ldsets(3).lineType=$LTDASH
ldsets(4).lineType=$LTDASHDOT
ldsets.lineColor=$CIBLACK
ldsets.titleColor=$CIBLACK
ldsets.markerType=$MKNONE
do i=1,4
   ldsets(i).title=loglegs(i)
end do
```

```
      retcode=GetAxisMultiDefaults(LGR,4,ldsets,$ATX, &
                 $AFLINEAR,laxes(1))
      laxes(1).title='Number of demands'
      laxes(1).tickType=$TTOUTSIDE
      laxes(1).tickRatio=2  ! В большом интервале два малых
      laxes(1).increment=2  ! Большие засечки с шагом 2
      laxes(1).gridStyle=$GSNONE  ! Без сетки
      laxes(1).numDigits=0  ! Оцифровка без дробных разрядов

      retcode=GetAxisMultiDefaults(LGR,4,ldsets,$ATY, &
                 $AFLOG10,laxes(2))
      laxes(2).title='Probabilities'
      laxes(2).tickType=$TTOUTSIDE
      laxes(2).gridStyle=$GSNONE
      laxes(2).axisColor=$CIBLACK
      laxes(2).axisPos=$APLEFT

      retcode=PlotGraph(LGR,2,laxes,4)
      retcode=PlotMultiData(LGR,probs,4,ldsets, &
                  laxes(1),laxes(2))
   end subroutine
END MODULE

   subroutine qgam(lam,a,mu,jmax,q)
      real lam          ! Интенсивность входящего потока
      real a            ! Параметр формы гамма-плотности
      real mu           ! Масштабный параметр
      integer jmax      ! Число необходимых q
      real q(0:jmax) ! Массив q

      real x,y
      integer j
      x=mu/(lam+mu); y=1.0-x
      q(0)=x**a
      do j=1,jmax
         q(j)=q(j-1)*y*(a+j-1.0)/j
      end do
   end subroutine qgam

   subroutine pmg1(lam,b,jmax,q,p)
      real lam              ! Интенсивность входящего потока
      real b                ! Среднее время обслуживания
      integer jmax          ! Max индекс
      real q(0:)
      real p(0:)            ! Стационарные вероятности

      p(0)=1.0-lam*b
      do j=1,jmax
         s=p(j-1)-p(0)*q(j-1)
         do i=1,j-1
            s=s-p(i)*q(j-i)
```

```
        end do
        p(j)=s/q(0)
      end do
   end subroutine pmg1

PROGRAM SGDemo
        use  DemSciGraph
        call DemoStart()
END
```

10.10. Опыт применения

На рис. 10.1, 10.2 приведены построенные по этой программе графики.

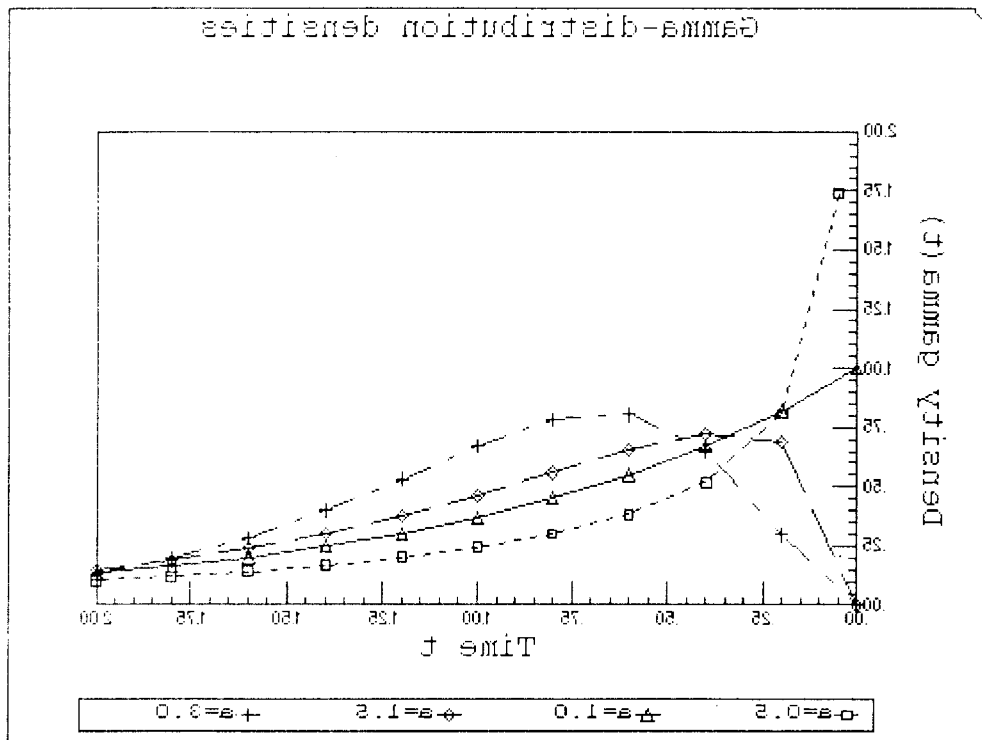

Рис. 10.1. Плотность гамма-распределения

В ходе отладки программы был приобретен весьма поучительный опыт:

1. Среда SciGraph, в которой можно просмотреть графики, практически не имеет полезных инструментов. Редактировать и масштабировать графики в ней

нельзя, запоминать графики можно только в весьма расточительном формате `.bmp`, справочная система не содержит никаких сведений о *программировании* в `SciGraph`.

2. Графики хорошо смотрятся на экране, но при попытке распечатки на черно-белом принтере получается "Черный квадрат" Малевича. По умолчанию фон графика задан темносиний, а оси, засечки, оцифровка и линия первой кривой — белыми. Корректировка умолчаний и тем более возможность работы с двумя комплектами установок (для экрана и принтера) отсутствуют. Поэтому приходится делать кустарные "заготовки" графических фрагментов программ. В частности, фон должен назначаться ярко-белым!

3. Кириллические надписи на графиках не проходят.

4. При попытке смещения графика фактически происходит смещение фона.

5. Цвета линий, маркеров и "легенд" могут задаваться независимо.

6. Возможности управления форматом оцифровки явно недостаточны. Отсутствует явно необходимый для логарифмических осей экспоненциальный формат, в оцифровке с целыми аргументами невозможно избавиться от точки (см. рис. 10.2). Длина засечек не регулируется, оцифровка намертво привязана к "большим" засечкам.

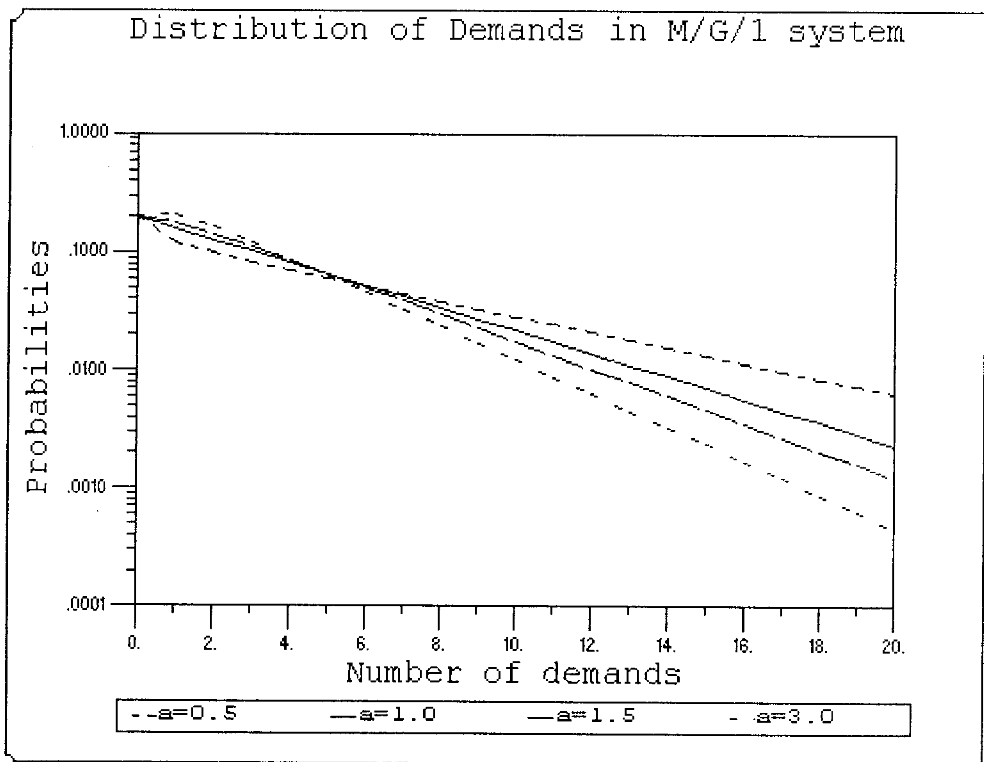

Рис. 10.2. Распределение числа заявок в системе $M/G/1$

В общем, в классе *систем программирования* `SciGraph` выглядит весьма солидно. Однако с графическими разделами математических пакетов класса *Maple*, *Mathematica*, *Scientific WorkPlace* или примыкающей к LaTeX'у системой *Gnuplot* он ни по возможностям, ни по удобству работы сравнения не выдерживает.

Глава 11.

Организация разработки программ

11.1. Состав и основные команды MDS

Работа на FPS4 происходит в "мастерской разработчика" — *Microsoft Developer Studio*, далее сокращенно именуемой MDS и применяемой также в других продуктах фирмы Microsoft, работающих под управлением *Windows 95* и *Windows NT*. MDS включает в себя текстовый редактор, средства создания проектов, оптимизирующий компилятор, средства просмотра и связывания программ, отладчик, профилировщик и инструменты разработки интерфейса приложений. В каждом из перечисленных средств имеются специализированные панели инструментов, которые можно адаптировать к потребностям пользователя, а также контекстно ориентированные `Help` и справки о необходимых командах (выдаются по нажатию [RM] при указателе мыши, зафиксированном на выделенном объекте). Через меню `View/Toolbars` можно вывести на экран несколько панелей. Меню `Tools/Customize/Toolbars tab` позволяет добавлять к панелям дополнительные кнопки. Команда `View/Full Screen` распахивает рабочее окно на весь экран, нажатие [Esc] возвращает к исходной структуре окон. По [Alt+2] высвечивается содержимое окна результатов счета.

Последние открывавшиеся файлы можно смотреть внизу меню `File`. Работа в MDS существенно ускоряется при использовании "горячих клавиш", сводную таблицу назначения которых можно просмотреть через `Help/Keyboards`. Нажатие [Ctrl+Break] прерывает текущий процесс.

11.2. Определение директорий

При инсталляции MDS автоматически определяются директории для файлов различного назначения:

- выполняемых (`fl32, link`),
- включаемых в текст программы по оператору `include`,
- библиотечных (содержащих наборы часто используемых стандартных процедур, вызываемых по ссылке),

- исходных (для возможности отладки динамически линкуемых библиотек с привязкой диагностики к фортрановским текстам).

Добавлять к этим директориям новые можно через **Tools/Options/Directories**, набирая их имена в конце списка соответствующей категории. Порядок просмотра каждого списка является существенным, поскольку поиск прекращается при нахождении первого объекта с заданным именем; очередность просмотра директорий можно менять перетаскиванием элемента списка. Выделенный элемент списка можно удалить.

11.3. Командная консоль

Работа в MDS удобна, но вынужденно ориентирована на некоторый набор стандартных режимов. Полный спектр возможностей предоставляет *командная консоль*. Для работы с командной консолью в стартовом меню *Windows 95* надлежит выбрать "Сеанс MS DOS". В этом режиме остаются доступными многие команды *Windows* и можно выполнять команды DOS при защищенном (расширенном) режиме использования оперативной памяти, легко переключаться между консолью и другими *Windows*-приложениями. Выход из консольного режима обеспечивается набором `exit`. Имеется возможность настроить диалоговое окно под индивидуальные потребности через меню **Пуск/Установки/Панель задач**. В частности, можно изменить размеры и положение окна, цветовую палитру, вид и размер шрифта, емкость буфера команд и т.п.

При начале консольного сеанса пути поиска библиотек, модульных файлов и т.п. соответствуют установленным в `autoexec.bat`. Для их изменения на текущий сеанс можно воспользоваться командой `set`. Если нужны изменения "многоразового" характера, они могут быть сведены в пакетный файл, выполняемый в начале каждого сеанса. В частности, их можно добавить к системному файлу `fpsvars.bat` вида

```
@echo off
set PATH=D:\MSDEV\BIN;%PATH%
set INCLUDE=D:\MSDEV\INCLUDE;%INCLUDE%
set LIB=D:\MSDEV\LIB;%LIB%
```

Пакетный файл вставляет директории, используемые FPS, в начало существующих списков путей; поэтому они будут просматриваться ранее путей *Windows*, что обеспечит более быстрый поиск и правильное разрешение коллизий обозначений.

Любая часть командной строки может быть помещена в *командный файл*, что снимает ограничение на длину командной строки (не более 255 символов). В этом случае компилятор или линкер вызываются командами вида

```
fl32 @<имя_командного_файла>
```

Между именем файла и "собакой" `@` не должно быть пробелов! В имя файла может включаться абсолютный или относительный путь, указывать расширение имени обязательно. Перед каждой опцией ставится пробел, а за ним знак "минус" или слэш. После точки с запятой до конца строки можно записывать комментарии. Некоторые опции требуют аргумента, который указывается после двоеточия. Пробелы при задании опции не допускаются. Имена опций и файлов нечувствительны к регистру, но идентификаторы, используемые как аргументы — чувствительны.

Опции и операнды компилятора должны предшествовать относящимся к линкеру. Перед последними должно стоять `/link`.

В одной команде можно задать несколько командных файлов, но вложение их не допускается.

Пересечение задач, решаемых в консольном режиме и в MDS, определяет целесообразность дальнейшего изложения материала *по задачам*. Принадлежность описываемых средств их решения к тому или иному режиму легко устанавливается из контекста и ниже специально не оговаривается.

11.4. Организация проектов

MDS организует разработку в форме *проектов*. Проект задается необходимыми исходными файлами и спецификациями на проект. Спецификации определяют цель проекта (их может быть несколько), среду реализации (практически — только *Windows 95*) и инструментальные установки. Внутри проекта создаются папки, объединяющие родственные файлы. Установки можно задавать как для проекта в целом, так и для его отдельных частей (по целям и файлам — например, специальный режим компиляции). Проект может содержать любое число подпроектов. В процессе построения проекта выполняются компиляция и связывание (линкование).

11.4.1. Цель проекта

Под *целью* проекта понимается создание отладочной или рабочей версии его. В первом случае формируется проект с возможностями получения отладочной информации, имеющий соответственно больший объем и меньшее быстродействие. По умолчанию генерируются оба варианта. Цель задается в диалоговых боксах `New Project` или `Insert Project`, а инструментальные установки — в `Project Settings`. Совокупность заданных целей и реализующих их файлов называется *конфигурацией проекта*. Приведем данные по объему компонент одного из простейших проектов: `proc1.f90` — 350 байтов, `proc1.mak` — 4697 байтов, `proc1.mdp` — 34304 байтов, `proc1.exe` — 180224 байтов, `proc1.ilk` — 184380 байтов, `proc1.pdb` — 222208 байтов. Отсюда следует очевидный практический вывод: из завершенных проектов сохранять только исходные файлы.

Назначение компонент будет поясняться по мере обсуждения этапов развития проекта.

11.4.2. Рабочее поле

Рабочее поле проекта (`Project WorkSpace`) состоит из корневой директории его и различных файлов, описывающих компоненты проекта, а также конфигурацию проекта в целом. Имена исходных, объектных и библиотечных файлов, опции компиляции и связывания содержатся в файле `.mak`, который может быть использован утилитой `NMAKE` для создания DOS-приложений. Текущее состояние среды (размеры и расположение окон, выбор шрифтов и цветов, расстановка отладочных точек и т.п.) хранятся в файле `.mdp`. В рабочем поле содержатся также файлы, формируемые при просмотре и обеспечивающие трассировку имен внутри процедур (`.sbr`). При формировании рабочей версии проекта этот этап может быть исключен. Имена всех служебных файлов по умолчанию совпадают с именем проекта. Многоцелевые проекты имеют несколько конфигураций.

11.4.3. Структура проекта

Имеются три основных варианта структуры проекта:

- проект верхнего уровня,
- проект верхнего уровня с одним подпроектом,
- множество подпроектов (без верхнего уровня), возможно, имеющих свои подпроекты.

11.4.4. Типы проектов

По *типу* проекты классифицируются следующим образом:

- консольный (`.exe`) — однооконный главный проект без графики,
- статически линкуемая библиотека (LIB) — библиотека подпрограмм в объектной форме, связываемых в `.exe` - модуль вместе с вызывающей программой,
- динамически линкуемая библиотека (DLL), элементы которой подключаются в процессе выполнения вызывающей программы по необходимости.

Проекты двух последних типов создаются без головной программы. При создании "расчетного" проекта с использованием библиотек последние добавляются к ней в процессе диалога `Edit Project`. При этом вводятся путь и имя библиотеки с соответствующим расширением (`.lib, .dll`).

Имеются еще три типа проектов с графическими средствами, в обсуждение которых мы пока входить не будем. Выбор типа проекта определяет набор входных установок (например, путей доступа к библиотекам), задаваемых в процессе диалога.

11.4.5. Задание целевых опций

После определения типа проекта командами `Settings` в меню `Build` задаются его опции: директории выходных файлов, опции компиляции и связывания, режимы оптимизации и просмотра. Их иерархия соответствует иерархии проекта (опции могут переопределяться на нижних уровнях применительно к частным подпроектам и отдельным файлам). При использовании библиотек указываются имена библиотек в целом, но не их членов.

В пункте `Set Default Configuration` можно изменить стандартные опции формирования проектов.

11.5. Проекты и модули

Если некоторые исходные файлы используются во многих проектах, удобно организовать процедуры в модули. К этой организации возможны два подхода.

Внутренняя инкапсуляция. Одна сложная программа может состоять из многих модулей. Каждый модуль является самодостаточной единицей, включающей все необходимые процедуры и данные некоторой совокупности задач. Все модули включаются в главную директорию проекта. Если много проектов совместно используют один и тот же модуль, модуль достаточно иметь только в одной директории. Все проекты, его использующие, должны иметь опцию компилятора /I, указывающую место модуля.

Внешние модули. Если используются модули из внешнего источника, необходимы только .mod–файл на этапе компиляции и .obj–файл при связывании. Для указания местонахождения этих файлов, если оно не совпадает с директорией проекта, нужно применить опцию /I или переменную окружения INCLUDE.

Предварительно откомпилированные модули (файлы с расширением .mod) нужно хранить в директории, включенной в переменную окружения PATH. Компилятор, встретив предложение use, находит этот модуль по его имени и компилирует вместе с вызывающей программой. Если проект использует модули, физически добавлять их к проекту не нужно. Все добавляемые, вызываемые и используемые файлы будут *автоматически* выявлены как входящие в проект и занесены в папку Dependencies. MDS, просматривая программу, формирует список вызываемых модулей и компилирует их раньше вызывающих программных единиц. Аналогично разыскиваются файлы, вставляемые посредством оператора include.

11.6. Создание, развитие и удаление проектов

Монолитные проекты верхнего уровня создаются для независимого от других приложения (разовое консольное, статическая библиотека). Для создания такого проекта нужно

1) Из меню File выбрать New.

2) Из списка выбрать New Project Workspace.

3) Выбрать тип проекта.

4) В боксе Name задать имя проекта.

5) В бокс Location ввести имя директории или воспользоваться назначаемой по умолчанию. В первом случае автоматически произойдет смена умолчания и для последующих проектов.

6) Нажать [Create].

Затем при необходимости можно добавлять файлы, модифицировать исходные тексты, изменять установки. Конфигурация приложения задается после входа в меню Build выбором из ниспадающего списка Default Configuration.

Для создания проекта с подпроектом (например, консольного приложения с использованием статической библиотеки) нужно

1) Описанным выше способом создать проект верхнего уровня.

2) В меню Build выбрать Subprojects/New.

3) В диалоговом боксе Insert Project подтвердить опцию Subproject и существующий проект верхнего уровня.

4) В боксе Name ввести имя подпроекта.

5) Из списка Type выбрать тип подпроекта.

6) Нажать [Create].

Для создания только подпроекта следует выбрать с помощью ниспадающих списков панели инструментов `Project` вариант `Subproject` и его конфигурацию и выполнить команду `Build` из меню `Build`.

Если подпроектом создается динамически линкуемая библиотека, нужно добавить ее к списку `DLL`, используемых отладчиком.

"Децентрализованная" архитектура используется при разработке связанных приложений — например, для двух консольных приложений, одно из которых использует статическую библиотеку. Последовательность разработки с *пустым* проектом верхнего уровня:

1) Создать проект верхнего уровня. В этом случае при выборе типа `Application` такой проект не будет содержать исходных файлов.
2) Из меню `Build` выбрать `Subprojects`.
3) Выбрать `New`. В появившемся диалоговом боксе подтвердить предлагаемые установки.
4) В боксе `Name` ввести имя проекта.
5) Выбрать тип проекта `Console Application`.
6) Нажать [Create]. Ответить на вопросы диалогового бокса.
7) Повторить шаги 3 – 6 для второго подпроекта.
8) Выбрать проект верхнего уровня из списка бокса `Insert Project`.
9) Повторить шаги 3 – 6 для подпроекта, содержащегося в первом выполняемом проекте. В диалоговом боксе `Insert Project` выбрать `Dynamic-Link Library`.
10) Нажать [Close].

После этого можно при необходимости добавлять исходные файлы и менять установки.

Для создания только подпроекта `DLL` нужно выбрать подпроект `DLL` и конфигурацию его построения из списка `Default Configuration` в меню инструментов `Project` и выполнить команду `Build` из меню `Build`. Для создания целой серии приложений выбирается конфигурация с пустым проектом верхнего уровня.

Построение иерархического проекта начинается с подпроектов. Добавляемые позже проекты образуют дополнительные поддиректории. Вставка подпроектов в поле проекта осуществляется через меню `Insert/Project` и последующий стандартный диалог.

Удалить проект можно через меню `Build/Configurations`. Будет высвечено дерево конфигураций, в котором нужно выделить удаляемую и выполнить команду `Remove`. Удаление последней конфигурации приводит к удалению проекта в целом (в частности, из всех других проектов, подпроектом которых он являлся).

Исходные файлы можно добавлять к проекту из любой директории *без их физического копирования или перемещения.*

Имеется возможность указывать директории для формируемых проектом файлов. Это позволяет, например, держать отдельно отладочную и рабочую версии проекта.

Начальные сведения о создании простейшего проекта сообщались в главе 1. В более общем случае необходимо:

1) Из меню `File` выбрать `New`. Откроется бокс диалога.

2) Выбрать из списка `Project Workspace`. Появится новый бокс.

3) Задать имя проекта.

4) Из ниспадающего меню выбрать тип приложения.

5) При необходимости указать диск и директорию, в которой должна создаваться поддиректория. Бокс `Platform` игнорировать.

6) Выбрать `Create`.

Далее к проекту выбором меню `Insert/Source Files` добавляются уже существующие файлы с расширениями `.for` или `.f90`. Для добавления более чем одного файла при выборе имен из списка нужно удерживать [Ctrl].

Проект, состоящий из одного исходного файла, может быть создан или открыт непосредственно в окне текстового редактора. Дальнейшие действия реализуются в процессе диалога через меню `Build`.

Открытие рабочего пространства существующего проекта выполняется через пункт `File` главного меню выбором соответствующего элемента списка (файл с расширением `.mdp`) с последующим нажатием [OK]. Для его закрытия в том же меню выбирается команда `Close Workspace`.

Создание нового подпроекта или добавление в этом качестве ранее созданного проекта выполняется через меню `Build/Subprojects/Select Project/Modify`. Затем выбираются соответственно `New` или `Include`. Дальнейшие действия подсказываются диалоговыми окнами. Так же производится удаление подпроекта.

При добавлении к проекту некоторого файла он добавляется ко всем конфигурациям этого проекта. Это делается через меню `Insert/Files into Project` (указываются директория и расширение, а затем производится выбор из списка). Если при изменении входящих в проект файлов были добавлены новые операторы `include` или `use`, в меню `Build` должна быть выполнена команда `Update All Dependencies`.

Для удаления файлов из проекта через `FileView` высвечивается список файлов и выделяется желаемый. Затем в меню `Edit` выбирается функция `Delete` или нажимается [Del]. Аналогичным образом можно переносить или копировать файлы из одного проекта в другой. При коррекции проекта выполнение команды `Build` вызывает обработку заново только тех включенных в проект файлов, которые не менялись с момента выполнения предыдущего построения этого проекта. При необходимости полного контроля выбирается команда `Rebuild All`. Информация о выявленных ошибках отображается в выходном окне.

11.7. Компиляция, связывание, запуск

Прогон выполняемого проекта может быть выполнен через меню `Build /Execute Project`. Отладочный вариант запускается через `Build/ Debug` после задания режима отладки. MDS дает возможность указания опций компилятора и линкера в диалоговых боксах, где эти опции логически сгруппированы и прокомментированы.

Удобное средство добавления часто используемых опций и файлов к командам управления компиляцией и связыванием — применение переменных окружения. Например, команда

```
set fl32=/G4
```

обеспечивает построение машинной программы, оптимальной для 486-го процессора (G5 — для Pentium'a). Установки подобного типа действительны только в текущем сеансе работы с FPS.

Хотя MDS обеспечивает операционную среду редактирования, построения и отладки проектов, этой работой можно управлять и из командной строки:

- с помощью команды `fl32` компилировать и связывать программные файлы;
- посредством `link` связывать существующие объектные файлы и библиотеки;
- помещать команды компилятора и линкера в файл и с помощью команды `nmake` строить сложные проекты.

Компилятор FPS преобразует исходные фортрановские файлы в один или несколько объектных файлов, содержащих эквивалентные машинные команды, и собщает о найденных ошибках. Компилятор может составить список всех переменных программы, специально отформатированный листинг исходного, объектного и машинного кодов. Встроенные возможности просмотра обеспечивают их сопоставление. Объектные файлы связываются (вместе с указанными дополнительными) и формируют выполняемый (`.exe`) программный файл.

Управлять работой компилятора можно заданием опций в диалоге `Settings`, а также из командной строки консоли. Аналогично обстоит дело при управлении линкером. На фазе связывания автоматически определяются встроенные библиотеки шага выполнения.

Опции линкера и файлы можно использовать для указания дополнительных библиотек. Если в команде указан файл с расширением `.lib`, то fl32 передает его имя линкеру как имя библиотеки.

Компилятор fl32 может обрабатывать исходные, объектные и библиотечные файлы, а также их комбинации в зависимости от их расширений:

Расширение файла	Действия компилятора или линкера
`.f90`	Компилирует исх. текст свободн. формата
`.for, .f`	Компилирует исх. текст фиксир. формата
`.obj`	Выполняется связывание
`.lib`	Нужные члены библиотеки переносятся в выполняемый файл
`.dll`	Устанавливается динамическая связь

Символы * и ? могут быть использованы для записи команд групповой обработки файлов в соответствии со стандартной нотацией MS DOS. Файлы, формируемые при компиляции и связывании, получают имя первого выбранного исходного файла.

11.7.1. Метакоманды и опции компилятора

Метакоманда — это специальный оператор управления трансляцией. Трансляцией можно управлять также через опции компилятора, выбираемые через MDS в меню `Options/Project/Compiler` или задаваемые в командной строке.

Метакоманды могут включаться и выключаться из программы. Они действуют, начиная с места своего появления в тексте и до конца файла, в том числе на

включаемые по **include** дополнительные файлы. Последние могут иметь свои метакоманды, действие которых будет распространяться и на включающий файл (исключая метакоманды, меняющие форму записи и длину строки). Метакоманды в программной единице, использующей модуль, не влияют на компиляцию модуля.

Многие метакоманды имеют противоположные им (например, пары $DEBUG и $NODEBUG); другие могут устанавливать параметры ($REAL, $INTEGER меняют заданное по умолчанию число байтов для представления вещественных и целых чисел соответственно). Это позволяет определять разные режимы трансляции для отдельных частей программы. Опции компилятора задаются предварительно и для программы в целом, но в случае их конфликта с метакомандами последние имеют приоритет.

Метакоманды делятся на следующие категории:

- Обеспечивающие строгое следование стандарту языка и тем самым — совместимость с другими компиляторами. Это $STRICT, $NOFREEFORM и им обратные $NOSTRICT, $FREEFORM, а также $FIXEDFORMLINESIZE.

- Используемые при условной компиляции ($DEFINE, $UNDEFINE, $IF DEFINED, $IF, $ELSE, $ELSEIF, $ENDIF).

- Управляющие режимом отладки ($DEBUG, $NODEBUG, $DECLARE, $NODECLARE, $LINE, $MESSAGE).

- Изменяющие принятые по умолчанию типы данных ($INTEGER, $REAL).

- Управляющие распечаткой исходного кода ($LIST, $NOLIST, $LINESIZE, $PAGE, $PAGESIZE, $TITLE, $SUBTITLE).

- Управляющая оптимизацией ($OPTIMIZE).

- Указывающая библиотеку для линкера в объектном коде ($OBJCOMMENT).

Признаками метакоманды являются знак доллара в *первой* колонке или комбинация символов !MS$ с предшествующим пробелом. Второй вариант используется при работе с расширениями Microsoft по отношению к стандарту Ф90. Он заставляет другие компиляторы воспринимать метакоманду как комментарий и тем облегчает перенос программы в другие среды. При включенной $STRICT все метакоманды этого типа воспринимаются как комментарии. Метакоманда и ее аргументы должны помещаться в одной (отдельной) строке исходного текста. Она может иметь комментарий, оформляемый по обычным для фортрановских программ правилам. На размещение метакоманд накладываются естественные ограничения: $OPTIMIZE, $INTEGER, $REAL, $STRICT, $NOSTRICT могут помещаться только в начале программной единицы. Прокомментируем наиболее полезные группы метакоманд.

Средства условной компиляции. Эти средства позволяют вставлять или обходить отладочные фрагменты, а также выполнять генерацию объектных кодов для различных условий применения. Смысл и синтаксис (за исключением обязательного знака доллара) метакоманд этой группы аналогичны соответствующим операторам Ф90. В управляющих условной компиляцией выражениях могут использоваться только целые переменные, задаваемые метакомандами вида $DEFINE J=6. Совпадение их имен с используемыми в программе проблем не создает.

Некоторые метакоманды и опции компилятора оказывают одинаковое воздействие на процесс компиляции:

Метакоманда	Эквивалентная опция
$DEBUG	/4Yb
$NODEBUG	/4Nb
$DECLARE	/4Yd
$NODECLARE	/4Nd
$DEFINE <символ>	/D<символ>
$INTEGER:<опция>	/4I<опция>
$FIXEDFORMLINESIZE:<опция>	/4L<опция>
$FREEFORM	/4Yf
$NOFREEFORM	/4Nf
$OPTIMIZE:'ON:<опция>'	/O<опция>
$PACK:<опция>	/Zp<опцня>
$REAL:<опция>	/4R<опция>
$STRICT	/4Ys
$NOSTRICT	/4Ns

Любое из имен метакоманд может иметь префикс !MS. Опции fl32 позволяют выбрать

- тип приложения,
- установки оптимизации,
- целевой процессор,
- выбираемые по умолчанию типы данных,
- режим обработки ошибок,
- языковый стандарт,
- содержание и формат листинга,
- дополнительные пути поиска включаемых файлов.

Дадим их сокращенный обзор по группам.

Управление генерацией кода

/G3, /G4, /G5	Код оптимизируется под процессор 386, 486 и Pentium соответственно с возможным снижением эффективности на процессорах других типов
/Ob2	Включает режим открытой вставки коротких подпрограмм вместо операторов вызова
/Od	Отключает оптимизацию
/Ox	Отключает контроль аргументов и результатов обращения к встроенным функциям, диагностика математических ошибок фазы выполнения отключена
/Oxp	То же, диагностика математики включена

Управление отладкой

/4{Y|N}b	Включает/отключает расширенный контроль ошибок фазы выполнения (переполнение при работе с целыми, контроль диапазонов индексов и подстрок)
/4{Y|N}d	Включает/отключает контроль объявления переменных
/4L{72|80|132}	Длина строки для исходного кода в фиксированном формате
/W0	Подавление предупреждений
/WX	Интерпретация предупреждений как ошибок

	Стандарт языка
`/4{Y|N}f`	Свободный/фиксированный формат
`/4{Y|N}s`	Включает/отключает расширения фирмы Microsoft относительно стандарта Ф90
`/4I2`	Переопределяет длину целых переменных
`/4R8`	Переопределяет длину вещественных переменных
`/4fps1`	Использование компилятора Fortran PowerStation 1.0

	Управление листингом
`/FL<имя-файла>`	Формирует листинг исходного файла
`/Fs<имя-файла>`	Готовит листинг с нумерованными строками, списками переменных и диагностикой компилятора. Посредством метакоманд `$LIST`, `$NOLIST`, `$LINESIZE`, `$PAGE`, `$PAGESIZE`, `$TITLE`, `$SUBTITLE`) можно задать дополнительные детали оформления

	Управление построением проекта
`/Fd<имя-файла>`	Отладочная информация пишется в `<имя>.pdb`
`/Fe<имя-файла>`	Назначение имени программе или DLL
`/FR<имя-файла>`	Запись расширенной информации просмотра
`/I<путь>`	Дополнительный путь поиска вставляемых и `.mod` – файлов (при необходимости опция повторяется).
`/LD`	Компилирует и связывает указанные файлы в DLL
`/ML`	Создает обычную библиотеку фазы выполнения
`/MWs`	Формирует стандартное графическое приложение
`/V<строка>`	Помещает строку (обычно — номер версии или копирайт) в формируемый объектный файл
`/Zi`	Формирует `.obj`-файл с возможностями символьной отладки и привязкой к строкам исходного текста
`/Zs`	Только синтаксический контроль

Команда /? вызывает список опций компилятора.

11.7.2. Связывание

Собственно связывание выполняется по команде `link <аргументы>` или как вторая фаза вызова компилятора. В последнем случае опции и файлы для связывания указываются после `/link`. Вслед за `link` может быть задано имя библиотечного файла. Когда связывание для данного проекта выполняется впервые, обрабатывается вся совокупность файлов проекта и формируется загрузочный модуль с расширением `.ilk`. При последующих его генерациях обрабатываются только "приращения" (и, разумеется, измененные объектные файлы).

Аргументы включают в себя опции и имена файлов и разделяются пробелами. Имена файлов могут содержать соответствующие пути. Линкер можно использовать для получения выполняемого файла, статической или динамической библиотеки. Для создания библиотек используется опция `/LIB`. Указывать имена системных фортрановских библиотек, с которыми устанавливается связь, необходимости нет. Выполняемая программа получает имя первого файла, указанного в списке аргументов.

Опции компилятора и линкера фиксируются в переменной окружения `fl32` и при работе в MDS могут быть изменены через диалог `Project Settings`. Они отрабатываются перед опциями командной строки, так что последние могут их изменять. Ниже в левом столбце перечисляются опции линкера в форме, используемой таблицей `link` из MDS, а в правом — для командной строки.

Общая категория

Output File Name	/OUT:<имя-файла>
Object/Library Modules	<имя-файла>
Generate Debug Info	/DEBUG
Ignore All Default Libraries	/NODEFAULTLIB
Link Incrementally	/INCREMRNTAL:{YES|NO}
Enable profiling	/PROFILE

Ввод

Object/Library Modules	<имя-файла>
Ignore Libraries	/NODEFAULTLIB:<библиотека>
Ignore All Default Libraries	/NODEFAULTLIB

Настройка

Link Incrementally	/INCREMENTAL:{YES|NO}

Отладка

Generate Debug Info	/DEBUG

Для компиляции без связывания (при отладке модулей и частей программы) нужно использовать опцию `/c`.

Сообщения об ошибках и предупреждения FL32 содержат имя исходного файла, номер строки (в скобках), диагностику и код возврата. Документация содержится в `InfoViewer`, раздел `Build Errors`. Порядок работы с сообщениями описан в разделе об отладчике. При работе в MDS объяснение ошибки можно получить, поместив курсор на ошибку и нажав [F1]; переход к соответствующему месту исходного файла осуществляется нажатием [F4] или двойным щелчком мыши.

11.7.3. Работа с модулями

Если головная программа содержит модули, они должны быть откомпилированы перед головной программой. Это можно сделать отдельно с опцией компилятора `/c` — будут получены файлы `.obj` и `.mod`. Далее дается команда компиляции головной программы и связывания:

```
fl32 modules.obj usemod.f90
```

Тот же результат можно получить и одной командой:

```
fl32 modules.f90 usemod.f90
```

Заметим, что в данном случае результирующая программа вопреки общему правилу получит имя `usemod.exe`.

11.7.4. Создание выполняемой программы

Меню Build предоставляет возможность

- скомпилировать файл без связывания,
- построить проект,
- перестроить все части проекта,
- построить пакетный (многоцелевой) проект,
- выполнить программу в отладочном или "боевом" режиме.

После определения проекта можно построить выполняемую программу выбором кнопки Partial Build или Full Build на панели инструментов (в первом случае будут заново компилироваться только измененные или добавленные исходные файлы). Для добавления файла нужно вызвать его в рабочее окно проекта. Далее достаточно нажать [Shift+F8].

11.7.5. Выполнение программы

Программа, построенная описываемой версией FPS, может быть выполнена только на компьютере под управлением *Windows 95* или *Windows NT* версии 3.51 и старше. Она запускается с консоли, а также с помощью Program Manager, File Manager и MDS. Простейший способ выполнения программы из среды MDS заключается в нажатии [Ctrl+F5].

Ошибки фазы выполнения в зависимости от приложения отображаются на консоли или в диалоговом боксе. Для их переадресации в файл следует в команде запуска указать > <имя-файла>. При указании >> выход можно добавить к существующему файлу.

11.8. Просмотр проекта

После закрытия диалога Project Files система MDS отобразит все файлы и их взаимные зависимости.

Средство просмотра (Browser) позволяет выполнять такие ценные функции процесса отладки, как поиск определения объекта и всех вхождений его имени в тексте программы. С его помощью строится также граф вызовов процедур. Режимы поиска базируются на продукте последнего связывания файла и не будут давать правильных результатов, если исходные файлы редактировались дополнительно. Для ускорения обработки проекта генерацию файлов просмотра можно отключить.

Окна просмотра имеют различный вид в зависимости от типа запрашиваемой информации:

- обо всех именах исходного файла,
- строка кода, где имя определено,
- каждая строка, содержащая ссылку на имя,
- вызываемые и вызывающие процедуры.

При построении проекта компилятор создает промежуточные `.sbr`–файлы. Утилита `bscmake` собирает их в единый `bsc`–файл. Просмотр любого из этих файлов осуществляется через панель инструментов или команды меню. Эти файлы обновляются только при выполнении команды `Build (Rebuild)`.

Информация об используемых в программе именах предоставляется через меню `Tools/Browse`. В открывшемся диалоговом боксе следует выбрать опцию `Definitions and References`. В левой панели окна появятся имена переменных и функций программы. При выделении одного из этих имен в правой панели будут указаны места определения имени и ссылок на него.

Простейшим средством поиска описания имени в позиции курсора является нажатие [Alt+F1]. По [Alt+Shift+F1] отыскивается очередное вхождение.

Граф вызовов отображает все функции, вызываемые выбранной программной единицей. Для его получения надлежит:

- выбрать имя функции в исходном файле или в боксе `Find` стандартного окна,
- войти в `Tools/Browse`,
- из списка `Select Query` выбрать `Call Graph` и нажать [OK].

В левой панели высвечивается граф вызовов выбранной функции. Свертка или развертка его производится щелчком [LM] по знаку "+" или "–". Для отображения информации о функции нужно щелкнуть по имени или изображению папки. Двойной щелчок открывает исходный файл с функцией.

11.9. Отладчик

При создании проекта MDS по умолчанию формирует отладочную и рабочую конфигурации проекта. Отладочная версия содержит полную информацию для символической отладки в терминах входного языка; при формировании ее объектного кода для облегчения сопоставления последнего с исходным текстом оптимизация исключается.

11.9.1. Отладка синтаксиса

Первая фаза отладки — устранение ошибок, мешающих завершить компиляцию и связывание. Ошибки, обнаруживаемые компилятором и линкером, высвечиваются в выходном окне. В случае "ошибки компилятора" указываются имя исходного файла, ошибочная строка, номер ошибки и поясняющее сообщение; при "ошибке линкера" — номер ошибки и сообщение. Информацию из выходного окна можно копировать и распечатывать. Предельное число выявляемых за один прогон компилятора ошибок ограничено 50. Превышение лимита при работе подготовленного пользователя обычно происходит, если не объявлен (посредством `use`) используемый модуль.

Выбор ошибки в выходном окне производится двойным щелчком мыши, перемещение по списку ошибок вниз делается нажатием [F4], вверх — по [Shift+F4]. При этом подсвечивается соответствующая строка исходного текста. При потребности в дополнительных пояснениях к сообщению об ошибке нажатием [F1] можно вызвать контекстно привязанный `Help`.

Для выбора строки исходного кода по ее номеру надлежит из меню `Edit` выбрать `Go To`. Откроется бокс диалога. Затем в поле `Enter Line Number` нужно ввести номер строки и нажать [OK].

Найденные компилятором ошибки предпочтительно просматривать и устранять последовательно, поскольку некоторые из них могут оказаться "наведенными", т.е. формальным следствием ранее допущенных ошибок, и не требуют отдельных исправлений. Типичные примеры таких ошибок — использование оператора `implicit none` (любая ссылка на неописанную переменную вызовет диагностику об ошибке), пропуск завершающего блочную конструкцию `end` и т.п.

Аналогично (но без ссылок на исходные строки) обрабатываются сообщения линкера. Типичная ошибка "`cannot find _main`" фиксируется при отсутствии в "связке" *головной* программы (не являющейся подпрограммой или функцией). Эта ситуация может возникнуть, когда первая строка головной программы начинается с первой позиции и рассматривается как комментарий.

Если линкер сообщает о неразрешенной внешней ссылке (вызове процедуры), нужно проверить список ее аргументов на соответствие списку параметров по числу и типу значений. Вторая возможная причина такой ошибки — работа с необъявленным массивом, ссылки на компоненты которого воспринимаются как вызов функции.

11.9.2. Организация семантической отладки

После устранения синтаксических ошибок проекта, выявленных компилятором и линкером, проводится семантическая (смысловая) отладка. Поводом к ее началу служат:

- аварийное завершение работы программы — с уведомлением типа "Программа выполнила недопустимую операцию и будет закрыта. Если эта ошибка будет появляться в дальнейшем, обратитесь к разработчику",
- отличие результатов счета от ожидаемых[1].

Наиболее гибким, но трудоемким методом семантической отладки является включение в текст программы выдач промежуточных результатов и сообщений о прохождении участков программы. Значительно удобнее работать со встроенными в системы программирования автоматическими отладчиками. Отладчик вызывается нажатием [F8]. С помощью отладчика MDS можно

- задать точки останова в программе,
- задать события, при наступлении которых должен быть останов,
- затребовать вывод значения интересующих программиста переменных — после контрольного останова или при любой смене значений.

Указанные действия нужно выполнить перед отладочным запуском программы.

Для фиксации контрольной точки *перед* оператором в соответствующей строке устанавливается курсор и нажимается клавиша [F9]. О выполнении установки свидетельствует появление слева красного кружка (повторным выполнением этих действий контрольная точка отменяется). *Все* контрольные точки можно отменить нажатием кнопки панели главного меню MDS.

Запуск программы на отладку производится нажатием [F5]. Таким же образом выполняется переход к следующей (по логике работы программы) контрольной

[1] Для любой программы нужно *предварительно* подобрать пример с известным решением.

точке. В момент останова желтая стрелка в окне с исходным текстом отмечает текущую контрольную точку. Окна отладчика имеют следующее назначение:

Call Stack. Отображаются имена всех вызванных, но еще не выполненных процедур.

Output. Реализуется запрограммированный вывод (отработка операторов `print` и `write`) и информация о построении проекта — в частности, диагностика компилятора.

Variables Window. Auto Tab. Отображаются выражения из предыдущего, текущего и последующего операторов программы.

Variables Window. Locals Tab. Отображаются значения локальных переменных текущей процедуры, отслеживаемой по шагам.

Watch. Отображаются тип, имя и значение переменных и выражений.

Окно `Watch` содержит четыре таблицы: `Watch1`, `Watch2`, `Watch3`, `Watch4`. В каждую таблицу удобно занести формируемый пользователем список объединяемых логикой программы переменных. Для добавления в окно переменной или выражения нужно: выбрать таблицу; набрать, вставить или втащить имя переменной или выражение; нажать [Enter]. Для отображения внутренних объектов вызываемых процедур следует пользоваться конструкцией `{<процедура>}<переменная>`. Значения отображаются в назначаемом по умолчанию формате. При необходимости этот формат может быть изменен. Двойной щелчок [LM] по разделителю окна `Variable` или `Watch` автоматически расширяет колонку для размещения всего содержимого.

Окно памяти открывается нажатием [Alt+6], окно `Watch` — по [Alt+3], окно `Quick Watch` для быстрой оценки выражения — по [Shift+F9]. Для вызова информации о типе переменной нужно выбрать в окне `Watch` строку с ее именем, щелкнуть [RM] и выбрать из ниспадающего меню `Properties`. В состоянии контрольного останова MDS автоматически отображает значение выражения, на котором стоит курсор в окне исходного текста. Значение (`DataTip`) переменной высвечивается при установке на него курсора, значение выражения — при *выделении* этого выражения. Этот режим не действует для некорректных ситуаций типа деления на нуль.

Структурированные переменные (в частности, массивы) появляются в окнах в "свернутом" виде, т.е. аналогично каталогам верхних уровней в *Windows*-продуктах, и могут быть развернуты щелчком [LM] по соответствующему плюсику. Значения переменных, изменившиеся с предыдущей контрольной точки, выделяются красным цветом.

На рис. 11.1 показано окно MDS в процессе отладки программы решения системы линейных алгебраических уравнений методом Гаусса.

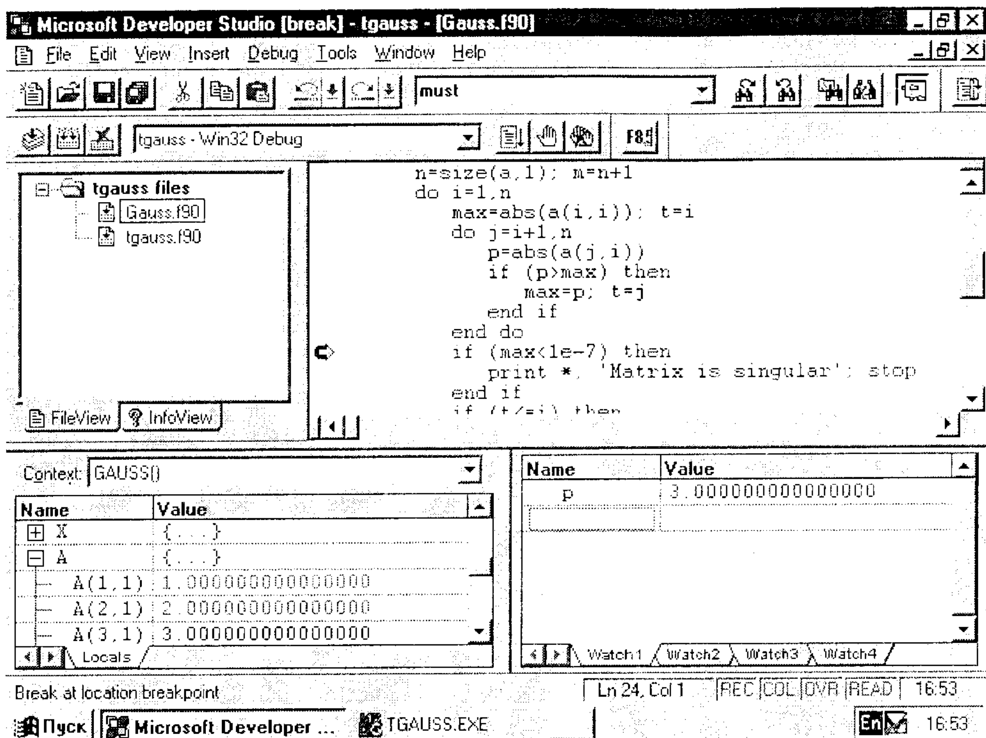

Рис. 11.1. Отладка программы в MDS

В окне с программой (вверху справа) видна контрольная точка (темный кружок), наложенная на который стрелка показывает место текущего останова. Слева внизу (в окне `Variables`) выведены структурированные данные `X` и `A`, причем `A` — в развернутом виде. В окне `Watch` (внизу справа) показано выведенное по запросу значение переменной `p`. В окне `Variables` можно редактировать значения, в окне `Watch` — имена и значения. Выражения и переменные удаляются из окна `Watch` выбором соответствующих строк и нажатием [Del].

При щелчке [RM] в отладочном окне появляется ниспадающее меню, содержащее наиболее употребительные в данном режиме команды. В зависимости от оперативной ситуации для очередного шага отладки могут быть выбраны следующие действия (в скобках указаны "горячие клавиши"):

`Go` — переход к следующей контрольной точке ([F5]),

`Step To Cursor` — переход в позицию курсора ([F7]),

`Step Into` — вход в процедуру([F8]),

`Step Over` — выполнение процедуры и останов сразу после ([F10]),

`Step Out` — выход из процедуры ([Shift+F7]),

`Restart` — повторный прогон ([Shift+F5]),

`Stop Debugging` — конец отладки ([Alt+F5]).

Эти действия выполняются также по нажатию соответствующих кнопок панели инструментов отладчика, имеющих наглядную символику. Меню отладчика появляется в панели инструментов, когда отладчик находится в активном состоянии (в частности, при контрольном останове). Меню может быть вызвано также через `View/Toolbars/Debug`.

Контрольные остановы можно задать также по изменению значения переменной или по истинности заданного выражения. В первом случае нужно набрать в боксе `Find` имя переменной, во втором — логическое выражение. Для установки других типов остановов по данным и отмены остановов указанного вида следует пользоваться диалоговым боксом `Breakpoints`.

11.10. Профилирование проекта

Профилировщик используется для изучения поведения программы в фазе прогона — в частности, чтобы определить, какие части кода занимают наибольшее машинное время и нуждаются в дополнительной настройке и оптимизации.
С его помощью можно выявить не выполнявшиеся части проекта, что существенно для управления процессом отладки и определения безопасности заимствованных программ. Порядок использования профилировщика ниже описан в соответствии с электронной документацией к системе. Поскольку она не соответствует реальному набору возможностей в меню `Build/Settings`, заставить профилировщик работать не удалось.

Профилирование проекта возможно лишь при условии, что он был создан в специальном режиме линкования, для задания которого надлежит:

- в меню `Build` выбрать `Settings` и затем вкладку `Link`,
- из ниспадающего списка `Category` выбрать `General`. Отметить опцию подключения профилировщика `Enable Profiling` — для профилирования фукций и `Generate Debug Info` — для профилирования строк,
- выбрать вкладку `Fortran` и далее из ниспадающего меню — `Program Database` или `Line Numbers Only`. Нажать [OK],
- из меню `Build` выбрать `Build`.

После этого программа будет подготовлена к профилированию. Далее нужно дать задание профилировщику, для чего

- из меню `Tools` выбрать `Profile`. Откроется диалоговый бокс с возможными режимами (`Function Timing`, `Function Coverage`, `Line Coverage`),
- выбрать нужный режим и нажать [OK].

В режиме `Function Timing` профилировщик записывает время работы программы, общее количество вызовов, число вызываемых функций, процент покрытия (доля от общего числа функций в проекте), наибольшую вложенность вызова, кратность вызова каждой функции и затраченное на ее работу время (с точностью до микросекунды). Время указывается чистое и с учетом дополнительно вызываемых процедур — абсолютное значение и в процентах.

В режиме `Function Coverage` выводится общее количество функций, процент покрытия и список вызванных функций.

В режиме `Line Coverage` профилировщик отмечает каждую выполненную строку исходного текста и процент покрытия.

Фактически профилировщик выполняет три различных программы: PREP, PROFILE, PLIST, из которых первая готовит измерения, вторая их выполняет, а третья организует вывод. Максимальная гибкость в управлении логикой и форматом вывода и выборе прослеживаемых частей кода может быть обеспечена собственными batch-файлами, вызывающими упомянутые программы. Список опций для них и примеры имеются в документации. Пакетные файлы MDS (fcount.bat, lcount.bat, fcover.bat, lcover.bat) могут использоваться как прототипы.

11.11. Работа с личными библиотеками

11.11.1. Базовые понятия

Применение личных библиотек для реализации однотипных функций уменьшает общий объем программного кода, упрощает структуру системы, облегчает доработку и обновление программных систем и обеспечивает работу разных частей таковых с тождественными версиями подпрограмм. Для обновления набора приложений с измененной библиотекой будет достаточно заново выполнить линкование.

Исходные файлы компилируются при задании режимов Partial Build или Full Build. Компиляция для создания библиотеки с командной строки выполняется с опцией /c. Объектный код откомпилированных подпрограмм собирается в .lib-файл, не проходя этап связывания. По умолчанию библиотека получает имя проекта. Большую по объему или составу библиотеку следует хранить в специальной поддиректории. В другой директории имеет смысл собрать исходные тексты членов библиотеки.

При создании "расчетного" проекта используемые им ранее созданные библиотеки добавляются через диалог Edit Project. Путь и имя библиотеки с соответствующим расширением (.lib, .dll) вводятся в бокс Files in Project. После изменения пути нужно перезагрузить операционную систему!!!

В консольном режиме для связи с другими библиотеками нужно выполнить одно из следующих действий:

- использовать в командной строке f132 /link с указанием нужного имени,
- запустить link с указанием имен библиотек,
- указать в команде f132 файл <имя_библиотеки>.lib (без /link).

Опция defaultlib:<библиотеки> добавляет библиотеки к списку библиотек, которые линкер просматривает при разрешении ссылок. Элементы списка библиотек разделяются запятыми. Очередность поиска библиотек:

1) указанные в командной строке,
2) определенные опцией defaultlib,
3) указанные в переменной окружения PATH,
4) указанные в переменной окружения LIB.

Для отладки библиотек нужна головная программа (драйвер), вызывающая подпрограммы (члены) библиотеки. Драйверы и вызываемые процедуры на этапе отладки должны компилироваться с отладочными опциями.

В работе с библиотеками существенно используются *переменные окружения*. Это часть среды, которую устанавливает *Windows*. Их можно менять в ходе сеанса

с консоли, но изменения действительны только на текущий сеанс. Если определена переменная **LIB**, линкер использует указанный в ней путь для поиска файлов и библиотек, указанных в командной строке. Переменную можно устанавливать в MDS, выбрав `Options/Directories`.

Линкер принимает стандартные COFF-библиотеки (Common Object File Format), обычно имеющие расширение `.lib`. Стандартные библиотеки содержат объектные подпрограммы.

При работе со статической библиотекой в выполняемый `.exe`-модуль включаются только реально необходимые процедуры. Их автоматически отбирает линкер для удовлетворения вызовов внешних процедур. *Динамически связываемые* библиотеки (DLL) подключаются к головной программе перед выполнением последней.

11.11.2. Статическая библиотека

Опишем практические действия автора при создании статической библиотеки из пяти перечисляемых ниже подпрограмм, обеспечивающих расчет системы массового обслуживания вида $M/D/n$ (многоканальная с пуассоновским входящим потоком и регулярным обслуживанием).

1) Вошел в меню `File/New/Project WorkSpace`; нажал [OK].

2) Выбрал тип проекта `Static Library`; задал имя `mdnlib`; нажал [Create].

3) Создал все вызываемые файлы (`gauss, exproot, mfact, mdn, pst`) — фактически были переброшены ранее отлаженные. Откомпилировал их.

4) Выполнил команду `Insert/Files into Project` .

5) Через меню `Build` выполнил предложенную в ситуационном режиме команду `Build mdnlib`. Получено сообщение `Creating library...` В отладочной директории проекта `Debug` находятся пять объектных модулей и заказанная библиотека `mdnlib.lib`. Никакого оглавления (в отличие от работы с версиями Фортрана-77 фирмы Microsoft) не обнаружено.

6) Закрыл проект с библиотекой.

7) Открыл новый консольный проект `tlib` . Набрал и откомпилировал тестовый файл.

8) Вызвал диалог `Edit Project/Insert Files into Project`. Указал в окне полное имя библиотеки с путем от корня и расширением:

   `d:\fps4\projects\mdnlib\debug\mdnlib.lib`.

   Нажал [OK].

9) По [Shift+F8] выполнил связывание.

10) По [Ctrl+F5] запустил программу на счет.

Получены совпадающие с эталонными результаты:

```
M/D/1 model, s = .81667E+00
M/D/2 model, s = .69056E+00
M/D/3 model, s = .60116E+00
```

11.11.3. Динамические библиотеки

DLL (dynamic link libraries, библиотеки динамической компоновки) предоставляют программистам место для хранения процедур, которые могут быть вызваны из одной или нескольких программ. DLL подключаются всегда *целиком* — обычно непосредственно перед запуском программы, но имеются средства явного указания в программе моментов загрузки и выгрузки ее. Одна и та же DLL может загружаться и выгружаться несколькими приложениями, причем для работы всех таких приложений достаточно одного экземпляра библиотеки. С загруженной DLL связывается "счетчик потребителей". Каждый новый потребитель добавляет к нему единицу, отключившийся — вычитает. При обнулении счетчика библиотека выгружается.

DLL повышают степень повторной используемости процедур и помогают управлять очень большими проектами. Максимальная потребность в памяти уменьшается при дроблении DLL на более мелкие библиотеки, используемые последовательно. Изменение тела процедур не влияет на работу вызывающих программных единиц, что облегчает коррекцию сложных программных систем.

Программы, содержащие обращения к DLL, компонуются на этапе выполнения. Во время компиляции и связывания программа просто информируется о наличии DLL, содержащей необходимые ей процедуры. Директория с DLL должна быть указана в пути поиска или быть той же, где главный проект. Если библиотеку используют несколько программ, она должна быть в отдельной директории, упомянутой в переменной окружения `PATH`, или перенесена в таковую посредством команды `Custom Build`. Корректировка переменных окружения возможна с консоли через меню `Tools/Settings`, но действительна только на текущий сеанс работы. Не забудьте перезагрузить систему после изменения!

Если подпрограмма, включаемая в DLL, должна быть доступна извне, в ней после заголовка должна следовать конструкция

```
!MS$ATTRIBUTES DLLEXPORT :: <имя_процедуры>
```

В вызывающих процедурах блоки интерфейса к вызываемым процедурам оформляются аналогично с заменой `DLLEXPORT` на `DLLIMPORT`.

При разработке DLL-проекта кроме собственно библиотеки (`.dll`) формируется файл с расширением `lib`, который в отличие от DOS'овских `lib`-файлов, включающих реализации функций, содержит лишь ссылку на DLL и *перечень* находящихся в ней функций. Оба эти файла по умолчанию имеют имя исходного проекта. При формировании библиотеки с консоли должна задаваться опция `/LD`. "Заголовочный" файл обязательно должен быть включен в проект с путем и именем. Затем его нужно прилинковать к основной программе точно так же, как и любую другую библиотеку. Объектный код DLL в выполняемую программу не включается.

Далее нужно запустить `exe`-файл. При загрузке выполняемого файла система просматривает его для того, чтобы определить, какие DLL используются при его работе, после чего пытается определить требуемые DLL.

Подготовку к созданию динамически линкуемой библиотеки мы покажем на примере, уже рассмотренном для статической библиотеки. Будет достаточно привести заголовок процедуры `mdn`, которая экспортируется из DLL и сама импортирует процедуры `exproot` и `pst`.

```
   subroutine mdn(lam,t,n,eps,jmax,p,c)
     !MS$ATTRIBUTES DLLEXPORT :: mdn
!*************************************/
!* Расчет стационарного распределения */
!*    числа заявок в системе M/D/n    */
!*       Разработчик Ю.И.Рыжиков      */
!*                (C)                 */
!*          Версия 09.03.1995         */
!*************************************/

   double precision lam  !! интенсивность входящего потока
   double precision t    !! время обслуж. заявки (постоянное)
   integer          n    !! число каналов обслуживания
   double precision eps  !! точность решения ур-ния (6.1.6)
   integer          jmax !! max индекс ответных вероятностей
   double precision p    !! ответные вероятности
   double precision c    !! lim p(j+1)/p(j) при больших j
   dimension        p(0:jmax)

   interface
      subroutine exproot(k,l,eps,n,w)
         !MS$ATTRIBUTES DLLIMPORT :: exproot
         integer  k           !!
         double precision l   !!
         double precision eps !!
         integer          n   !!
         double complex   w   !!
         dimension        w(n)
      end subroutine exproot

      subroutine pst(k,n,lt,r,w)
         !MS$ATTRIBUTES DLLIMPORT :: pst
         integer          k
         integer          n
         double precision lt
         double complex   r
         double precision w
         dimension w(0:k*n),r(*)
      end subroutine pst
   end interface
...............................................
```

Члены DLL, как в данном примере, могут ссылаться друг на друга. Интерфейс к транзитивно вызываемым процедурам не задается. Атрибут DLLEXPORT для "технологических" процедур (к их числу принадлежит, например, не нужная конечным пользователям pst) не указывается. В интерфейсных к mdn и mfact блоках тестирующей программы tmdn указан атрибут DLLIMPORT.

Результаты расчета, как и следовало ожидать, совпали с полученными посредством статической библиотеки.

Достоин дополнительного обсуждения вопрос об использовании памяти. При работе с DLL суммарный объем приложений, опирающихся на библиотеку, уменьшается, однако объем самой DLL на порядок больше, чем аналогичной статической

библиотеки. В нашем примере из пяти сравнительно небольших процедур с объемом исходных файлов 8034 байта статическая и динамическая библиотеки занимали 26584 и 232448 байтов соответственно. Правда, один и тот же экземпляр процедуры из DLL одновременно могут использовать несколько вызывающих программ. Это дает значительный выигрыш при работе примитивов операционной системы (что и имеет место в *Windows*). Для вычислительных приложений, на которые ориентирован Ф90, некоторой пользы от DLL можно ожидать лишь в больших сетях с архитектурой "клиент — сервер". Напомним, что для этого члены библиотеки должны быть реентерабельными (см. в электронной документации раздел `Creating multithread applications`).

11.11.4. Библиотекарь

Обработчик библиотек (`Microsoft Library Manager` — `lib.exe`), ниже именуемый библиотекарем, используется в консольном режиме для формирования и коррекции содержимого библиотек, создания экспортных файлов и импортируемых библиотек для работы с DLL. Он может быть вызван из командной строки в формате

`lib [<опции>] <файлы>`

с опциями и именами файлов, соответствующими затребованной функции. Исходными являются `.lib` и `.obj`-файлы в формате `COFF`. При отсутствии опций подразумевается режим создания единой библиотеки, получающей имя первого объекта списка и расширение `.lib`. Этот режим обеспечивает создание библиотек, их слияние, пополнение и корректировку (последняя выполняется при наличии в библиотеке объекта, одноименного с одним из членов списка, причем заменяющий элемент должен следовать сразу за библиотекой с заменяемым). Опция `/REMOVE:` позволяет удалить из библиотеки перечисляемые за ней файлы. Опция `/LIST:` формирует список указанной библиотеки, который может быть направлен в указанный файл стандартными средствами DOS. Для сохранения существующей библиотеки следует использовать опцию `/OUT:<имя_новой>`. Может использоваться комбинация опций в одной команде. При большом объеме задания библиотекарю вышеуказанную команду заменяют командным файлом `lib @<файл>`. По-видимому, функции модификации библиотек и контрольного формирования оглавления следует выполнять именно в консольном режиме.

Библиотекарь не использует переменную окружения LIB.

Задачи для упражнений

1. Запрограммировать расчет сумм нижеприведенных рядов с заданной точностью:

$$\begin{aligned}
\arcsin x &= x + \frac{1}{2\cdot 3}x^3 + \frac{1\cdot 3}{2\cdot 4\cdot 5}x^5 + \frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6\cdot 7}x^7 + \ldots; \\
\Phi(x) &= \frac{2}{\sqrt{\pi}}\left[x - \frac{x^3}{1\cdot 3} + \frac{x^5}{1\cdot 2\cdot 5} - \frac{x^7}{1\cdot 2\cdot 3\cdot 7} + \frac{x^9}{1\cdot 2\cdot 3\cdot 4\cdot 9} - \ldots\right]; \\
I_n(x) &= \frac{x^n}{2^n\cdot n\,!} - \frac{x^{n+2}}{2^{n+2}\cdot(n+1)\,!} + \frac{x^{n+4}}{2^{n+4}\cdot 2\cdot(n+2)\,!} - \frac{x^{n+6}}{2^{n+6}\cdot 6\cdot(n+3)\,!} + \ldots \\
S(x) &= \sum_{n=0}^{\infty}\frac{(-1)^n x^n}{n\,!!}.
\end{aligned}$$

2. Вычислить бесконечное произведение

$$\Gamma(z) = \frac{1}{z}\prod_{k=1}^{\infty}\frac{(1+1/k)^2}{1+z/k}.$$

3. Вычислить $\sqrt[m]{y}$ с точностью 0,001 методом последовательных приближений, считая известным начальное приближение x_0.

Указание: воспользоваться разложением левой части равенства $(x+\Delta x)^m = y$ по степеням поправки с сохранением линейного члена. При расчете принять $m = 5$.

4. Подсчитать количество N и сумму S положительных чисел в убывающей последовательности $\{x_1,\ldots x_{70}\}$. Число повторений цикла не должно превышать $N+1$.

5. Найти три начальных момента последовательности $\{x_1,\ldots x_{98}\}$ по формуле вида

$$m_k = \sum_{i=1}^{98} x_i^k \Big/ 98, \qquad k = 1, 2, 3.$$

В программе этой задачи должен быть только один цикл.

6. Составить таблицу значений многочлена $\tilde{T}_9(x)$ для $x = -0,7(0,01)0,2$, пользуясь рекуррентной формулой

$$\tilde{T}_{n+1}(x) = 2x\tilde{T}_n(x) - \tilde{T}_{n-1}(x), \qquad n = 1, 2, \ldots$$

при начальных $\tilde{T}_0(x) = 1$ и $\tilde{T}_1(x) = x$.

7. Сторона a_{2n} правильного вписанного многоугольника с удвоенным числом сторон выражается через a_n и радиус круга R формулой

$$a_{2n} = \sqrt{2R^2 - 2R\sqrt{R^2 - a_n^2/4}}.$$

Отправляясь от шестиугольника, воспользоваться этой формулой для определения числа π. Процесс прекратить, когда уточнение за один шаг станет меньше 10^{-6}. Вывести конечное значение n.

8. Рассчитать рекуррентно таблицу биномиальных коэффициентов

$$C_n^m = \frac{n!}{m!\,(n-m)!}, \qquad m = 0, 1, \dots n,$$

полагая $n = 11$.

9. Рассчитать массив коэффициентов

$$a_j = \left(\frac{\mu}{\lambda+\mu}\right)^k \left(\frac{\lambda}{\lambda+\mu}\right)^j \frac{\Gamma(k+j)}{j\,!\Gamma(k)}, \qquad j = 0, 1, \dots, 50$$

где $\Gamma(z)$ — гамма-функция; $\Gamma(z+1) = z\Gamma(z)$, $\Gamma(1) = 1$.

10. Имеется таблица функции $y(x)$ для возрастающей последовательности $\{x_1, \dots x_{70}\}$, причем узлы $\{x_i\}$ расположены неравномерно. Найти значение $y(z)$, $x_1 < z < x_{70}$, линейной интерполяцией.

11. Рассчитать значение обобщенного многочлена

$$R_k(x) = a_k x^{[k]} + a_{k-1} x^{[k-1]} + \dots + a_1 x^{[1]} + a_0$$

по схеме, аналогичной методу Горнера. Обобщенная степень $x^{[m]} = x(x-1)\cdots(x-m+1)$, $x^{[1]} = x$. Для определенности считать $k = 14$.

12. Выполнить расчет цепной дроби

$$a_0 + \cfrac{x}{a_1 + \cfrac{x}{a_2 + \cfrac{x}{a_3 + \cfrac{x}{a_4}}}}$$

по схеме, аналогичной методу Горнера.

13. Составить процедуру вычисления

$$\tanh(x) = \sum_{i=0}^{20} \frac{x^{2i+1}}{(2i+1)!} \Big/ \sum_{i=0}^{20} \frac{x^{2i}}{(2i)!}$$

с параллельным расчетом членов обеих сумм и экономией вычислений. Применить эту процедуру для расчета

$$y = \frac{2\tanh(1/2) - 3\tanh(x - 0,1)}{4 + \tanh(4x - 1)}.$$

14. Составить программу для умножения матрицы A(6,10) на B(10,4). Найти наименьший из максимумов строк матрицы-произведения C с указанием строки и столбца, в которых он находится.

15. Матрица A имеет 5 строк и 5 столбцов. Наибольший по модулю элемент каждой строки поменять местами со стоящим на главной диагонали.

16. Найти наименьший по модулю элемент матрицы A(6,10) с помощью подпрограммы выбора наименьшего по модулю элемента вектора B из n компонент. Встроенными функциями не пользоваться!

Литература

[1] *Баррон Д.* Введение в языки программирования. /Пер. с англ. — М.: Мир, 1980. — 190 с.

[2] *Бартеньев О. В.* Современный Фортран. — М.: "Диалог-МИФИ", 1998. — 397 с.

[3] *Бартеньев О. В.* Фортран для студентов. — М.: "Диалог-МИФИ", 1999. — 400 с.

[4] *Боглаев Ю. П.* Вычислительная математика и программирование. — М.: Высшая школа, 1990. — 544 с.

[5] *Каханер Д., Моулер К., Нэш С.* Численные методы и математическое обеспечение: Пер. с англ. М.: Мир, 1998. — 575 с.

[6] *Меткалф М., Рид Дж.* Описание языка программирования Фортран-90. / Пер. с англ. М.: Мир, 1995. — 302 с.

[7] *Рыжиков Ю. И.* Имитационное моделирование систем массового обслуживания. — Л.: ВИККИ им.А.Ф.Можайского, 1991. — 111 с.

[8] *Рыжиков Ю. И.* Подготовка рукописей в издательской системе LaTeX с графическими средствами. — СПб.: ВИККА им.А.Ф.Можайского, 1996. — 111 с.

[9] Сборник научных программ на Фортране. Руководство для программиста. Вып. 1 -2. — М.: Статистика, 1974.

[10] *Турчак Л. И.* Основы численных методов. — М.: Наука, Физмат, 1987. — 320 с.

[11] Фортран 90. Международный стандарт. /Пер. с англ. — М.: Финансы и статистика, 1998. — 378 с.

[12] *Gehrke W.* Fortran-90 Language Guide. — London: Springer Verlag, 1995. — 384 pp.

[13] *Hahn B.* Fortran 90 for scientists and engineers. — London: Edward Arnold, 1994 (входит в электронную документацию).

[14] Numerical recipes: The art of scientific computing (Fortran version). — Cambridge: University Press, 1989. — 702 pp.

[15] *Willé D. R.* Advanced scientific Fortran. — Chichester etc.: Wiley & Sons, 1995. — 234 pp.

Оглавление

Введение **3**

1. Элементы и объекты программы **7**
1.1. Набор символов . . . 7
1.2. Формат программы . . . 7
1.3. Понятие о проекте . . . 8
1.4. Текстовый редактор . . . 11
1.4.1. Работа с файлами . . . 11
1.4.2. "Горячие" клавиши . . . 11
1.4.3. "Заказная" распечатка . . . 13
1.4.4. Управление окнами просмотра . . . 13
1.5. "Хороший стиль" программирования . . . 13
1.6. Лексемы . . . 14
1.7. Описания . . . 15
1.8. Выражения . . . 17
1.8.1. Арифметические выражения . . . 17
1.8.2. Встроенные функции . . . 18
1.8.3. Логические выражения . . . 19
1.9. Инициализация и изменение значений переменных . . . 21
1.10. Простейший ввод-вывод . . . 22

2. Операторы управления **23**
2.1. Разветвления . . . 24
2.2. Циклы . . . 26
2.2.1. Цикл с шагом . . . 26
2.2.2. Цикл с условием . . . 27
2.2.3. Вложенные циклы . . . 29
2.2.4. Дополнительные средства . . . 30

3. Массивы **32**
3.1. Роль массивов . . . 32
3.2. Описание массивов . . . 32
3.3. Вырезки и сечения массивов . . . 33
3.4. Задание массивов . . . 34
3.5. Поэлементные операции . . . 35
3.6. Выборочные действия . . . 35
3.7. Встроенные функции для векторов и матриц . . . 36
3.7.1. Справочные функции . . . 36
3.7.2. Преобразование массивов . . . 37

3.7.3. Неэлементные операции 38
3.8. Динамическая память 39

4. Обработка строк **40**
4.1. Строки и массивы строк 40
4.2. Функции от строк 40

5. Программные компоненты **42**
5.1. Программа и ее компоненты 42
5.2. Подпрограммы 44
5.3. Функции 45
5.4. Расположение операторов 46
5.5. Области видимости меток и имен 46
5.6. Внутренние процедуры 47
5.7. Интерфейс процедур 47
5.8. Специальные виды параметров 49
5.8.1. Массивы как параметры 49
5.8.2. Процедуры как параметры 52
5.8.3. Необязательные и ключевые параметры 53
5.9. Рекурсивные процедуры 54
5.10. Общие области и подпрограммы данных 56
5.11. Модули и работа с ними 58
5.11.1. Пример "модульной" программы 59
5.11.2. Специальные виды модулей 61
5.12. Доступ к объектам модуля 63

6. Ввод-вывод данных **64**
6.1. Список ввода-вывода 64
6.1.1. Ввод, управляемый списком 65
6.1.2. Группа NAMELIST 65
6.2. Спецификации формата 65
6.2.1. Управление размещением информации 67
6.2.2. Переменные спецификации формата 67
6.3. Взаимодействие списков объектов и форматов . . . 68
6.4. Внешние файлы 69
6.5. Внутренние файлы 70

7. Производные типы данных **72**
7.1. Производные типы данных 72
7.2. Операции над определяемыми типами 73

8. Ссылки и списки **76**
8.1. Базовые понятия 76
8.2. Ссылки как псевдонимы 77
8.3. Связные списки 77
8.4. Целые указатели 80

9. Вычислительные методы **82**
9.1. Построение частотных характеристик 82
9.2. Рекуррентные вычисления 84
9.3. Генерация случайных чисел 84
9.4. Сортировка и поиск 86
9.4.1. Поиск в массиве 86
9.4.2. Понятие о сортировках 86
9.4.3. Линейный выбор с обменом 87
9.4.4. Пузырьковая сортировка 87
9.4.5. Челночная сортировка 88
9.4.6. Быстрая сортировка 89
9.4.7. О внешних сортировках 91
9.5. Работа с разреженными матрицами 91
9.6. Обращение матрицы методом Гаусса 97
9.7. Решение уравнения методом секущих 98
9.8. Вычисление определенных интегралов 99
9.9. Решение дифференциальных уравнений 102
9.10. Математические библиотеки IMSL 107
9.11. Numerical Recipes 110

10. Пакет научной графики **112**
10.1. Знакомство с пакетом 112
10.2. Константы SciGraph 113
10.3. Структуры установок 114
10.4. Типы данных и осей 116
10.5. Вызывающая последовательность 117
10.6. Установка основных параметров 117
10.6.1. График в целом 117
10.6.2. Ряды данных 118
10.6.3. Координатные оси 119
10.7. Рисование графика 120
10.7.1. Основа графика 120
10.7.2. Собственно графики 120
10.8. Коды возврата 121
10.9. Пример применения Scigraph 122
10.10. Опыт применения 127

11. Организация разработки программ **130**
11.1. Состав и основные команды MDS 130
11.2. Определение директорий 130
11.3. Командная консоль 131
11.4. Организация проектов 132
11.4.1. Цель проекта 132
11.4.2. Рабочее поле 132
11.4.3. Структура проекта 133
11.4.4. Типы проектов 133
11.4.5. Задание целевых опций 133
11.5. Проекты и модули 133
11.6. Создание проектов 134
11.7. Компиляция, связывание, запуск 136
11.7.1. Метакоманды и опции компилятора 137
11.7.2. Связывание 140

11.7.3. Работа с модулями 141
11.7.4. Создание выполняемой программы 142
11.7.5. Выполнение программы 142
11.8. Просмотр проекта 142
11.9. Отладчик 143
11.9.1. Отладка синтаксиса 143
11.9.2. Организация семантической отладки 144
11.10. Профилирование проекта 147
11.11. Работа с личными библиотеками 148
11.11.1. Базовые понятия 148
11.11.2. Статическая библиотека 149
11.11.3. Динамические библиотеки 150
11.11.4. Библиотекарь 152

Задачи для упражнений **153**

Литература **155**