Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ
ГИДРОМЕТЕОРОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

В.Ю. Чанцев

# ЯЗЫК ПРОГРАММИРОВАНИЯ ФОРТРАН ДЛЯ РЕШЕНИЯ ОКЕАНОЛОГИЧЕСКИХ ЗАДАЧ

## Часть 1

Учебное пособие

Санкт-Петербург
РГГМУ
2022

УДК 519.682.3+519.677
ББК 32.973.22:22.193
Ч–18

**Чанцев, Валерий Юрьевич.**

Ч-18 Язык программирования ФОРТРАН для решения океанологических задач. Часть 1. Учебное пособие / В.Ю, Чанцев.– Санкт-Петербург : РГГМУ, 2022. – 156 с.

Представлены основы объектно-ориентированного языка программирования высокого уровня FORTRAN 90. Вмести с описанием элементов Фортрана приводятся примеры построения вычислительных алгоритмов и программных кодов для их реализации. Изложение материала поострено логично с последовательным усложнением структуры вычислительных примеров и введением все новых и новых элементов языка Фортран. Каждый раздел пособия завершается набором упражнений для самостоятельного закрепления пройденного материала.

Предназначается для студентов океанологического профиля гидрометеорологического направления, обучающихся по дисциплинам «Численные методы решения задач в гидрометеорологии», «Программирование инженерных и научных задач океанологии» и «Моделирование природных систем».

УДК 519.682.3+519.677
ББК 32.973.22:22.193



ISBN 978-5-86813-556-9

СОДЕРЖАНИЕ

# Введение

Сегодня язык программирования Фортран основывается на стандарте Fortran 90, который приводит его в соответствие с большинством современных языков структурного программирования [1].

Двумя наиболее важными достижениями, вероятно, являются новые возможности работы с массивами (главы 9 и 15) и впечатляюще расширенный набор встроенных процедур, включая так называемые элементарные функции, которые работают со всеми или выбранными элементами аргументов массива [2]. Теперь вы можете определять свои собственные типы или структуры (глава 12) и даже создавать связанные с ними списки с помощью указателей (глава 13). Модули (глава 8) могут компилироваться независимо и могут содержать определения типов и объявления переменных, а также процедуры. Использование интерфейсных блоков позволяет перегружать определенные имена процедур общими именами, а также перегружать операторы. Условные циклы теперь возможны с DO WHILE (глава 6), а также с оператором CASE (глава 6) [3].

Есть три конкретных области, в которых это учебное пособие может внести особый вклад в формирование устойчивых навыков программирования научных и инженерных задач в области гидрометеорологии.

Во-первых, выбранный в пособие подход является подходом к решению проблем, разработанным в течение многих лет обучения программированию студентов университета, не имеющих большого опыта работы с компьютером. Компьютер представлен как инструмент (вероятно, самый захватывающий в 21 веке) для решения интересных, реальных проблем, и обсуждаются примеры из многих областей, особенно науки и техники. Поэтому технические особенности каждой новой конструкции на языке Фортран обычно представляются только после мотивации путем постановки подходящей задачи. Поскольку цель этого пособия — по возможности полно научить студентов решать задачи с помощью компьютера, первые 12 глав в некотором смысле являются подготовкой к последним трем. В представленных разделах проводится знакомство с некоторыми современными компьютерными приложениями, такими как моделирование, моделирование и численные методы. Также имеется большое количество упражнений, включающих самые разные приложения. Для большинства из них предусмотрены решения. Те, у которых нет

решений, могут подойти для использования в качестве лабораторных проектов в учебной ситуации.

Во-вторых, структурированные проблемы развиваются повсюду. Кто еще недостаточно знаком с Фортраном защищен от разрушительного эффекта оператора GOTO до тех пор, пока не углубится в текст данного пособия.

В-третьих, в пособии делается упор на то, что стало называться стилем программирования, и представлены рекомендации по написанию понятных, читаемых программ.

Таким образом, данное учебное пособие может использоваться в качестве руководства для дополнительного самообучения любым обучающимся, кто хочет изучить FORTRAN 90, текущий международный стандарт, версия которого использовалась при составлении здесь учебного материала [4].

Хотя это пособие в первую очередь предназначено для начинающих изучать проблемы программирования, более опытные программисты смогут найти здесь много интересного. Он сможет даже чему-то научиться! Приводимые приложения содержат сводку практически всех встроенных функций и операторов FORTRAN 90 (включая те, которые не рекомендуются по стилистическим соображениям), с примерами их общего использования.

Чтобы следовать большинству примеров, не требуется специальной математической подготовки. Время от времени встречаются набеги на высшую математику, но они самодостаточны и могут быть проигнорированы без потери преемственности (вы даже можете найти их поучительными!).

Есть надежда, что это учебное пособие даст некоторое представление о том, как компьютеры могут использоваться для решения реальных проблем, и что после его изучения вы сможете больше узнать об этом увлекательном предмете – программирование для себя.

# Глава 1 Начало работы

Может так сложиться, что вы раньше вообще не пользовались компьютером (что крайне невероятно), но, наверняка, вы знакомы с использованием калькулятора. Калькулятор может выполнять только арифметические действия и отображать ответ. Конечно, у более умных из них есть элементы памяти, где могут храниться промежуточные результаты, и функциональные клавиши, такие как *sin*, *log* и т.д. Самые сложные калькуляторы позволяют хранить последовательность операций (инструкций), необходимых для решения задачи. Эта последовательность инструкций называется программой. Для проведения всего комплекса вычислений вам нужно всего лишь загрузить программу в калькулятор, нажать клавишу запуска, ввести необходимые данные и сидеть сложа руки пока калькулятор выдает ответ. Компьютер, будь то небольшой персональный или большой безличный кластер, в принципе является лишь продвинутым программируемым калькулятором, способным хранить и выполнять наборы инструкций, называемых программами, для решения конкретных задач.

Не исключено, что вы уже использовали компьютер раньше, но только для запуска программных пакетов, написанных кем-то другим. В эту категорию можно отнести электронные таблицы, базы данных и текстовые редакторы (исключая компьютерные игры). Если вы все-таки решили приступить к чтению этого учебного пособия, вы, вероятно, интересуетесь наукой или техникой и достаточно интересуетесь программированием, чтобы писать свои собственные программы для решения своих конкретных задач, вместо того, чтобы полагаться на чей-то более общий пакет. Тем более, что в выбранной профессии умение программировать решаемые задачи является одним из приоритетов для осуществления профессионального лифта.

## 1.1 Почему Фортран

Конкретный набор правил для кодирования инструкций для компьютера называется языком программирования. Таких языков много, например Fortran, BASIC, Pascal и C++. Fortran, что означает FORmula TRANslation, был первым языком программирования «высокого уровня». Это позволило использовать символические имена для представления математических величин и записывать математические формулы в достаточно понятной форме, например, $X$

= *B*/(2***A*). Идея Фортрана была предложена в конце 1953 года Джоном Бэкусом в Нью-Йорке.

Использование Фортрана распространилось так быстро, что вскоре возникла необходимость стандартизировать его, чтобы программа, написанная в соответствии со стандартом, гарантированно работала на любом компьютере, заявляющем о поддержке стандарта. В 1966 году был опубликован первый в истории стандарт языка программирования. Эта версия стала известна как Fortran 66. Новый стандарт, Fortran 77, был опубликован в 1978 году. Несмотря на конкуренцию со стороны более новых языков, таких как Pascal и C, Фортран продолжал процветать настолько, что в августе 1991 года вышел очередной стандарт, Fortran 90, который является основой современных версий языка [5].

Для более полного понимания использования того или иного языка программирования необходимо принять во внимание то, что все они разделяются по способу реализации на интерпретируемые и компилируемые. Последние выполняют все вычисления гораздо быстрее.

При этом начиная с 2000 годов появился, и приобрел огромную популярность язык интерпретатора Python [6]. Популярность Питона связана с простотой его использования и наличием большой библиотеки полезных подключаемых программ для работы с базами данных и графической визуализации используемой числовой информации [7, 8]. Но, как было отмечено выше, скорость выполнения вычислительных операций оставляет желать лучшего.

В качестве основных недостатков Питона можно отметить:

- Медленная работа. Питон плохо подходит для создания высокопроизводительных приложений, лучше сразу выбрать другой язык.
- Крайне мало возможностей для процессорных оптимизаций. Их сводит на нет особая модель памяти языка.

Тем не менее, язык Питон является, как и большинство языков программирования, объектно-ориентированным. Это означает, что получив хорошую практику программирования на языке Фортран, очень легко освоить и Питон самостоятельно.

## 1.2 Запуск программ на Фортране

Компиляция

Текст программы Fortran 90 можно записать в любом текстовом редакторе, поддерживающем кодировку ASCII. Имя файла программы должно иметь расширение «.f90» (например, `spin.f90`).

После написания программы и оформления ее в виде файла, необходимо выполнить так называемую компиляцию программы. Для этого необходимо установить на компьютере один из компиляторов языка Фортран. Наиболее простой вариант – компилятор GFortran или Intel Fortran.

Самый простой способ компиляции с помощью компилятора GFortran осуществляется в командной строке OS Windows или в терминале OS Linux записью следующей строки (как пример):

```
gfortan o spin spin.f90
```

где `spin` – имя исполняемого файла, который сформируется после компиляции; а `spin.f90` – исходный файл программы. Имя spin в данном случае вымышленное. Вы сами должны назначать эти имена.

В результате этой операции будет создан исполняемый файл `spin.exe`, запуск которого и будет выполнять вашу программу.

Наиболее полную информацию об использовании того или иного компилятора нужно искать в специальной литературе по программированию.

Написание программ

Если вы новичок в Фортране, а тем более в программировании, вам следует запустить примеры программ из этого раздела как можно скорее, не пытаясь даже понять, как они работают. Объяснения будут представлены позже. Вам нужно будет узнать, из руководства или от кого-то еще, как вводить и запускать программы на Фортране в вашей компьютерной системе.

Необходимо помнить, что Фортран, как и все современные языки программирования, воспринимает только буквы латинского алфавита и арабские цифры. В некоторых случаях можно комментарии писать и на кириллице, но нужно знать, что на другом компьютере ваш комментарий будет выглядеть как последовательность непонятных символов (абракадабра). Поэтому рекомендуется использовать исключительно латинские символы.

*Пример программы приветствия*

Эта программа поприветствует вас, если вы наберете на клавиатуре свое имя:

```
! My first Fortran 90 program!
! Greetings!

CHARACTER NAME*20

PRINT*, 'What is your name?'
READ*, NAME
PRINT*, 'Hi there, ', NAME
END
```

Вы должны получить на экране ПК следующий вывод (ваш ответ выделен курсивом):

```
What is your name?
Vladimir
Hi there, Vladimir
```

*Программа случаев СПИДа*

Чтобы ознакомиться с простейшими операциями вывода сообщений на экран монитора и ввода данных с клавиатуры компьютера, а также с выполнением элементарных арифметических действий представлена небольшая программа AIDS. Эта программа вычисляет количество накопленных случаев СПИДа $A(t)$ в США в год $t$ по формуле

$$A(t) = 174{,}6(t - 1981{,}2)^3$$

```
PROGRAM AIDS
! Calculates number of accumulated AIDS cases in USA
 INTEGER T          ! year
 REAL  A          ! number of cases

 PRINT*, ‘Enter Year:’
 READ*, T
 A = 174.6 * (T - 1981.2) ** 3
 PRINT*,'Accumulated AIDS cases in US by year',T,':',A
END PROGRAM AIDS
```

Если вы укажете значение 2000 для года, вы должны получить вывод

```
Accumulated AIDS cases in US by year 2000: 1.1601688E+06
```

Ответ дан в научной системе счисления: E+06, что означает умножить предшествующее число на $10^6$, поэтому количество случаев в примере составляет около 1,16 миллиона. Методом проб и ошибок повторно запускайте программу, чтобы узнать, когда накопится около 10 миллионов случаев.

Попробуйте ввести ошибку в значении *t* (например, 2,000), чтобы посмотреть, как отреагирует Фортран.

**Резюме**

- Компьютерная программа представляет собой набор закодированных инструкций для решения конкретной задачи.
- Оператор READ* предназначен для передачи данных в компьютер.
- Оператор PRINT* предназначен для печати (отображения) результатов.

**Упражнения**

1.1 Напишите программу для вычисления и вывода суммы, разности, произведения и частного двух чисел *A* и *B* (вводится с клавиатуры).

Знаки арифметических действий:

+ — сложение
– — вычитание
* — умнжение
/ — деление
** — возведение в степень.

Используйте программу, чтобы узнать, как Фортран реагирует на попытку деления на ноль.

1.2 Энергия, запасенная в конденсаторе, равна $CV^2/2$, где *C* — емкость, а *V* — разность потенциалов. Напишите программу для вычисления энергии для некоторых выборочных значений *C* и *V*.

# Глава 2 Правила программирования на Фортране

В этой и следующей главах мы подробно рассмотрим, как писать программы на Фортране для решения простых задач. Есть два основных требования для успешного овладения этим искусством:

- необходимо изучить точные правила кодирования операторов и инструкций;
- должен быть разработан логический план решения задач.

Первые две главы посвящены в основном первому требованию: изучению некоторых основных правил кодирования. Как только они будут освоены, можно будет переходить к более существенным задачам.

## 2.1 Анатомия работы программы

Для получения представления о работе Фортран-программы можно записать и запустить программу MONEY для вычисления сложных процентов:

```
PROGRAM MONEY
! Calculates balance after interest compounded
 REAL BALANCE, INTEREST, RATE

 BALANCE = 1000
 RATE = 0.09
 INTEREST = RATE * BALANCE
 BALANCE = BALANCE + INTEREST
 PRINT*, 'New balance:', BALANCE

END PROGRAM MONEY
```

Когда вы запускаете программу на Fortran 90, происходят два отдельных процесса. Сначала компилируется программа. Это означает, что каждое выражение транслируется в машинный код, понятный компьютеру. Далее, выполняется скомпилированная программа. На этом шаге выполняется каждая инструкция программы.

Во время компиляции место в оперативной памяти (ОЗУ) компьютера выделяется для любых чисел (данных), которые будут генерироваться программой. Эту часть памяти можно представить как набор (комплект) ящиков или ячеек памяти, каждая из которых может

содержать только одно число. Этим ячейкам памяти в программе соответствуют символические имена. Итак, оператор

```
BALANCE = 1000
```

выделяет число 1000 ячейке памяти, которая связана только с именем BALANCE. Поскольку содержимое BALANCE может изменяться во время работы программы, BALANCE называется переменной.

В переведенном (скомпилированном) виде алгоритм нашей программы выглядит примерно так:

1) Помещаем число 1000 в ячейку памяти BALANCE
2) Помещаем число 0,09 в ячейку памяти RATE.
3) Умножаем содержимое RATE (СТАВКИ) на содержимое BALANCE и помещаем ответ в INTEREST (ПРОЦЕНТЫ)
4) Добавляем содержимое BALANCE к содержимому INTEREST и помещаем ответ в BALANCE
5) Печатаем (отображаем) сообщение New balance:, за которым следует содержимое BALANCE
6) Стоп.

Во время выполнения эти переведенные операторы выполняются в порядке сверху вниз. После выполнения используемые ячейки памяти будут иметь следующие значения:

```
BALANCE : 1090
INTEREST : 90
RATE   : 0.09
```

Обратите внимание, что исходное содержимое BALANCE будет утеряно.

Оператор PROGRAM в первой строке представляет имя программы. Этот оператор не является обязательным, и за ним может следовать необязательное имя. Вторая строка, начинающаяся с восклицательного знака, является комментарием для читателя и не влияет ни на компиляцию, ни на выполнение программы. Переменные в программе могут быть разных типов; оператор REAL объявляет их тип в этом примере. Первые три непустые строки этой программы являются неисполняемыми, т.е. никакие действия операторами этих строк не выполняются.

Попробуйте выполнить следующие упражнения:

1) Измените первый исполняемый оператор на чтение `BALANCE = 2000` и убедитесь, что вы понимаете, что происходит при повторном запуске программы.
2) Запустите программу.
3) Пропустите строку `BALANCE = BALANCE + INTEREST` и повторно запустите. Можете ли вы объяснить, что происходит?
4) Попробуйте переписать программу так, чтобы исходное содержимое `BALANCE` не потерялось.

Вероятно, у вас уже возник ряд вопросов, таких как

- Какие имена можно использовать для ячеек памяти?
- Как можно представить числа?
- Что произойдет, если оператор не помещается на одной строке?
- Как мы можем организовать вывод более аккуратно?

Ответы на эти и, надеюсь, многие другие вопросы будут даны в следующих разделах.

## 2.2 Структура программы

Общая структура простой программы на Фортране выглядит следующим образом (элементы в квадратных скобках необязательны):

```
[PROGRAM имя программы]
      [операторы описания]
      [исполняемые операторы]
END [PROGRAM [имя программы]]
```

Как видите, единственным обязательным оператором в программе на Фортране является `END`. Этот оператор сообщает компилятору, что больше нет операторов Фортрана для компиляции.

```
END [PROGRAM [имя программы]]
```

означает, что имя программы может быть опущено в операторе `END`, но если есть имя программы, ключевое слово `PROGRAM` не может быть опущено.

*Операторы* (*инструкции*)

Операторы составляют основу любой программы на Фортране и могут содержать от 0 до 132 символов. Все операторы, кроме оператора присваивания (например, BALANCE = 1000), начинаются с ключевого слова. До сих пор встречались следующие ключевые слова:

END, PRINT, PROGRAM и REAL. Оператор может быть пустым, т.е. не только не выполнять какое-либо действие, но и не декларировать какие-либо свойства. Пустые операторы (например, оператор комментария) поощряются, чтобы сделать программу более читаемой за счет разделения логических разделов.

Как правило, в каждой строке рекомендуется размещать по одному оператору. Однако в строке может появиться несколько операторов, если они разделены точкой с запятой. Для ясности это рекомендуется только для очень коротких заданий, таких как

```
A = 1; B = 1; C = 1
```

*Значение пробелов*

Пробелы, как правило, несущественны, т.е. вы можете использовать их для улучшения удобочитаемости, добавляя операторы с отступом (добавляя пробелы слева) или заполняя операторы. Однако есть места, где пробелы недопустимы. Чтобы быть конкретным, необходимо определить технический термин: token.

Термин *token* в Fortran 90 представляет собой основную значащую последовательность символов, например, метки, ключевые слова, имена, константы, операторы и разделители. Пробелы не могут появляться в *token*. Таким образом, `INTE GER`, `BAL ANCE` и `< =` не разрешены (`<=` — это оператор), а `A * B` разрешено и совпадает с `A*B`.

Однако имя, константа или метка должны быть отделены от соседнего ключевого слова, имени, константы или метки хотя бы одним пробелом. Таким образом, `REALX` и `30CONTINUE` не допускаются (30 — это метка во втором случае, а `CONTINUE` — пустой оператор).

*Комментарии*

Любые символы, следующие за восклицательным знаком (!) (кроме строки символов), являются комментариями и игнорируются компилятором. Вся строка может быть комментарием. Пустая строка также интерпретируется как комментарий. Комментарии следует использовать свободно для улучшения читабельности.

*Продолжение строк*

Если оператор слишком длинный и не помещается на одной строке, он будет продолжен на следующей строке, если последним непустым символом в нем является амперсанд (&):

```
A = 174.6 *             &
    (T - 1981.2) ** 3
```

Продолжение обычно идет к первому символу следующей строки без комментариев. Однако, если первым непустым символом в строке продолжения является &, продолжение выполняется до первого символа после &. Таким образом, *token* может быть разделен на две строки, хотя это не рекомендуется, так как это делает код менее читаемым.

Амперсанд & в конце строки комментария не будет продолжать комментарий, так как & интерпретируется как часть комментария. Это означает, что продолжение комментария на следующей строке должно начинаться опять с оператора (!).

Типы данных

Концепция типа данных является фундаментальной в Fortran 90. Тип данных состоит из набора значений данных (например, целых чисел), средства обозначения этих значений (например, –2, 0, 999) и набора операций (например, арифметика), которые разрешены на них. Стандарт Fortran 90 требует пяти внутренних (то есть встроенных) типов данных, которые делятся на два класса, числовые и нечисловые. Числовые типы бывают целыми, действительными и комплексными. Нечисловые типы являются символьными и логическими.

С каждым типом данных связаны и другие различные типы. По сути, это количество битов, доступных для хранения, так что, например, могут быть два типа целых чисел: короткие и длинные.

В дополнение к внутренним типам данных вы можете определить свои собственные производные типы данных, каждый со своим собственным набором значений и операций.

## 2.3 Буквенные константы

*Буквенные константы* (часто называемые просто константами) — это токены, используемые для обозначения значений определенного типа, то есть реальных символов, которые могут использоваться.

Прежде чем мы подробно рассмотрим константы, нам нужно кратко рассмотреть, как информация представлена в компьютере.

*Биты и байты*

Основной единицей информации в компьютере является бит. Это что-то, что имеет только два возможных состояния, обычно описываемых как включенное и выключенное. В вычислительной технике это двоичные цифры 0 и 1, которые используются для математического представления этих двух состояний. Слово «бит» в сокращении означает «двоичная цифра». Поэтому числа в памяти компьютера должны быть представлены в двоичном коде, где каждый бит в последовательности соответствует большей степени 2. Например, десятичные числа от 0 до 15 кодируются в двоичном виде следующим образом:

Таблица 2.1 Десятичная и двоичная кодировка числа

| Десятичный | Двоичный | Десятичный | Двоичный |
|---|---|---|---|
| 0 | 0000 | 8 | 1000 |
| 1 | 0001 | 9 | 1001 |
| 2 | 0010 | 10 | 1010 |
| 3 | 0011 | 11 | 1011 |
| 4 | 0100 | 12 | 1100 |
| 5 | 0101 | 13 | 1101 |
| 6 | 0110 | 14 | 1110 |
| 7 | 0111 | 15 | 1111 |

Байт – это объем компьютерной памяти, необходимый для одного символа, и имеет длину восемь бит. Поскольку каждый бит в байте может находиться в двух возможных состояниях, это дает $2^8$, т.е. 256 различных комбинаций.

Размер памяти компьютера (и любых носителей данных) измеряется в байтах, поэтому 64 КБ, например, означает чуть больше 64 000 байт (поскольку 1 КБ на самом деле означает 1024 байта). Компьютеры иногда называют 8-, 16-, 32- или 64-битными машинами. Это описывает длину единиц информации, обрабатываемых их микропроцессорами (чипами). Чем длиннее эти единицы, тем быстрее компьютер обрабатывает информацию.

Ц е л о ч и с л е н н ы е  к о н с т а н т ы

Целочисленные буквенные константы используются для обозначения значений целочисленного встроенного типа. Самым простым и очевидным представлением является целое число без знака или со знаком (целое число), например,

1000  0  +753  -999999  2501

Диапазон целых чисел не указан в стандарте, но существует большой ряд вычислительных устройств с 8-, 16-, 32- и 64-битной системой, которые имеют разный диапазон доступных целых чисел:

Таблица 2.2 Диапазоны целых чисел для 8-, 16-, 32- и 64-битных устройств

| Тип устройства | Диапазон | |
|---|---|---|
| 8 бит | –128 ÷ 127 | $-2^7 \div 2^7-1$ |
| 16 бит | –32768 ÷ 32767 | $-2^{15} \div 2^{15}-1$ |
| 32 бит | −2147483648 ÷ 2147483647 | $-2^{31} \div 2^{31}-1$ |
| 64 бит | –9223372036854775808 ÷ 9223372036854775807 | $-2^{63} \div 2^{63}-1$ |

Положительные целые числа также могут быть представлены в двоичной, восьмеричной (с основанием 8) или шестнадцатеричной форме, например,

```
binary:     B'1011'
octal:      O'0767'
hexadecimal:   Z'12EF'
```

В Фортране можно использовать как буквы верхнего, так и нижнего регистра. Кавычки (") могут использоваться вместо апострофов (') в качестве *разделителей*. Эти формы ограничены использованием с оператором DATA, а также во внутренних и внешних файлах в виде строк цифр без начальных букв и разделителей.

Д е й с т в и т е л ь н ы е  к о н с т а н т ы

Они используются для обозначения значений вещественного (real) внутреннего типа и принимают две формы. Первая форма очевидна и называется позиционной формой или фиксированной

точкой и состоит из строки цифр с десятичной точкой. Он может быть подписан или не подписан. Например:

0.09 37. 37.0 .0 –.6829135

Фортран позволяет не записывать 0 перед десятичной точкой или после нее, но десятичная точка сама по себе должна присутствовать.

Вторая форма называется экспоненциальной формой или с плавающей запятой. Она в основном состоит либо из целого числа (со знаком или без знака), либо с фиксированной точкой (со знаком или без знака), за которой в обоих случаях следует буква E, за которой следует целое число (со знаком или без знака). Число, следующее за E, является показателем степени и указывает степень числа 10, на которую нужно умножить число, стоящее перед E. Например:

```
2.0E2       (= 200.0)
2E2        (= 200.0)
4.12E+2      (= 412.0)
-7.321E-4     (= -0.0007321)
```

Действительные константы хранятся в памяти в *экспоненциальной форме*, независимо от того, как они на самом деле записаны. Если вещественное число имеет дробную часть, оно может быть представлено приближенно (иногда ее называют *конечной машинной точностью*). Даже если нет дробной части, вещественное число хранится иначе, чем целое число того же значения. Например. 43 — целое число, а 43,0 — действительное. В памяти они будут представлены по-разному.

## 2.4 Имена и переменные

Мы уже видели, что ячейкам памяти можно давать символические *имена*, такие как BALANCE и RATE. В Fortran 90 имена могут даваться не только ячейкам памяти, но и другим вещам, например, самой программе. Имя должно содержать от 1 до 31 буквенно-цифрового символа и должно начинаться с буквы. Буквенно-цифровые символы представляют собой 26 латинских букв, 10 цифр и *знак подчеркивания* (_).

За исключением строк символов, Fortran 90 нечувствителен к регистру, т.е. имена MYNAME и MyName представляют одно и то же. Возможно, следует отметить, что программисты на Фортране имеют давнюю традицию писать программы только в верхнем регистре. Это восходит к временам, когда приходилось использовать машины для

перфорации карт — они могли представлять только прописные буквы. Однако смесь прописных и строчных букв читается намного легче (она содержит больше информации, чем только прописные буквы). Поэтому может быть лучше использовать `NoOfStudents`, чем `NOOFSTUDENTS`. Также, как правило, лучше использовать осмысленные имена (однако не слишком длинные), такие как `NoOfStudents`, а не просто *N*.

В Фортране нет зарезервированных слов; поэтому вы можете использовать имя `END` для ячейки памяти, хотя это, конечно, не рекомендуется! В следующей таблице показаны некоторые допустимые и недопустимые имена.

Таблица 2.3 Пример допустимых и недопустимых имен Фортрана

| Допустимые имена | Недопустимые имена (почему?) |
|---|---|
| `X` | `X+Y` |
| `R2D2` | `SHADOW FAX` |
| `Pay_Day` | `2A` |
| `ENDOFTHEMONTH` | `OBI-WAN` |

*Переменная* — это ячейка памяти, значение которой может быть изменено во время выполнения программы. Имя переменной строится по указанным выше правилам. Переменная имеет тип, который определяет тип константы, которую она может содержать. Ему присваивается тип в объявлении типа, например:

```
INTEGER X
REAL INTEREST
CHARACTER LETTER
REAL :: A = 1
```

Обратите внимание, что переменная может быть инициализирована в ее объявлении. В этом случае необходимо использовать двойное двоеточие (::). Значение переменной, инициализированной таким образом, может быть изменено позже в программе. Хотя при инициализации разрешены более сложные выражения, по стилистическим соображениям рекомендуется ограничить инициализацию простыми присваиваниями, как показано выше.

Хотя переменные `X`, `INTEREST` и `LETTER` были объявлены в приведенном выше фрагменте программы, они пока не определены,

так как не имеют значения. Следует избегать ссылок на неопределенную переменную. Переменная может быть определена несколькими способами, например, инициализируя ее (A выше) или присваивая ей значение, как в других примерах, которые мы видели. Переменной также может быть присвоено начальное значение в операторе DATA после объявления, например:

```
REAL A, B
INTEGER I, J
DATA A, B / 1, 2 / I, J / 0, -1/
```

Имя в программе должно быть уникальным. Например, если программа называется MONEY, попытка объявить переменную с таким же именем вызовет ошибку. Описанные здесь переменные являются скалярными, поскольку могут содержать только одно значение.

*Правило неявного (Implicit) типа*

В более ранних версиях Фортрана было так называемое правило неявного типа. Переменные, начинающиеся с букв от *I* до *N* включительно, автоматически определялись как целочисленные, а переменные, начинающиеся с любой другой буквы, автоматически определялись как вещественные. Это (иногда бесполезное) правило все еще применяется в Fortran 90 по умолчанию, чтобы обеспечить совместимость кода, написанного в более ранних версиях.

Правило неявного типа может привести к серьезным ошибкам программирования. Реальное значение может быть непреднамеренно присвоено переменной, которая по умолчанию является целым числом; затем дробная часть усекается (отрезается). Например, утверждение

```
Interest_rate = 0.12
```

в программе, где Interest_rate явно не объявлен, переменной будет присвоено значение 0.

Для защиты от таких ошибок настоятельно рекомендуется использовать оператор IMPLICIT NONE в начале всех программ. Этот оператор отключает правило неявного типа; следовательно, все переменные, используемые в программе, должны быть объявлены. Кстати, это способствует хорошему стилю программирования; необходимость объявлять переменную означает, что вы вынуждены думать о том, что она представляет

## 2.5 Логическое построение плана решения задачи

Рассмотрим построение плана решения задачи на примере закона тяготения. Если камень брошен вертикально вверх с начальной скоростью *u*, его вертикальное перемещение *s* по истечении времени *t* определяется формулой

$$s = u \cdot t - \frac{g \cdot t^2}{2}, \qquad (2.1)$$

где *g* – ускорение свободного падения. Сопротивление воздуха не учитывалось. Мы хотели бы вычислить значение *s*, учитывая *u* и *t*. Обратите внимание, что здесь нас интересует не то, как вывести формулу, а то, как вычислить ее значение. Логическая подготовка этой программы выглядит следующим образом:

1. Введите в компьютер значения *g*, *u* и *t*.
2. Вычисляется значение *s* по формуле (2.1)
3. Выведите значение *s* и *t*
4. Стоп

Этот план может показаться тривиальным и пустой тратой времени. Тем не менее, вы были бы удивлены, узнав, как много новичков, предпочитающих бежать прямо к компьютеру, пытаются запрограммировать шаг 2 перед шагом 1. Прежде всего стоит развить умственную дисциплину планирования своей программы. Если ручка и бумага отталкивают вас, почему бы не использовать ваш текстовый процессор? Вы всегда можете ввести план в виде строк комментариев в программе.

Программа выглядит следующим образом:

```
PROGRAM Vertical
! Vertical motion under gravity

IMPLICIT NONE

REAL,PARAMETER :: G = 9.8 !acceleration due to gravity
REAL S           ! displacement (m)
REAL T           ! time
REAL U           ! initial speed (m/s)

PRINT*, ' Time    Displacement'
PRINT*
U = 60
```

```
T = 6
S = U * T - G / 2 * T ** 2
PRINT*, T, S

END PROGRAM Vertical
```

Странный способ объявления G делает ее именованной константой, так как ее значение точно не должно меняться в программе. Именованные константы рассмотрим несколько позднее, а сейчас нужно напомнить, как описываются в программе Фортрана простые арифметические действия (табл. 2.4).

Таблица 2.4 Числовые встроенные операторы

| Оператор | Приоритет | Значение | Пример |
|---|---|---|---|
| ** | 1 | возведение в степень | 2 ** 4 (=24) |
| * | 2 | умножение | 2 * A |
| / | 3 | деление | B / DELTA |
| | | | |
| + | 4 | сложение | A + 6.9 |
| – | 5 | вычитание | A – B |

Стиль программирования

Программы, которые написаны давно способом, хотя и могут делать то, что требуется, могут быть трудны для понимания, когда их читаешь несколько месяцев спустя. Поэтому чрезвычайно важно развивать искусство написания хорошо продуманных программ с четко описанной логикой. Это называется стилем программирования и будет проявляться в большинстве программ в этом учебном пособии. Программа в предыдущем разделе была написана с учетом сказанного:

- В начале есть комментарий, описывающий, что делает программа.
- Все переменные объявлены и описаны в отдельных строках в алфавитном порядке. Вы можете включить инициализацию с объявлением и описанием, например.

  ```
  REAL :: T = 6          ! time
  ```

- Пробелы использовались по обе стороны от знаков равенства и операторов (например, **), а также после запятых.

- Пустые строки использовались для разделения отдельных частей программы.

## 2.6 Числовые выражения

Программа `Vertical` в разделе 2.5 использует следующий код:

```
U * T - G / 2 * T ** 2
```

Это пример числового выражения — формулы, объединяющей константы, переменные (и функции, такие как квадратный корень) с использованием встроенных числовых операторов. Он определяет правило для вычисления значения. Поскольку он вычисляет только одно значение, это скалярное числовое выражение. Существует пять числовых встроенных операторов, показанных в таблице 2.4. Ввод пробелов по обе стороны от операторов сделает выражения более читабельными. Эти операторы называются внутренними, потому что они встроены. Позже мы увидим, как определять новые операторы и как перегружать встроенный оператор, т.е. придавать ему другое значение. Оператор с двумя операндами, как в `A + B`, называется бинарным или диадическим оператором. Когда оператор появляется только с одним операндом, как в `–Z`, он называется *унарным* или *монадическим*.

Порядок выполнения операций в выражении определяется приоритетом операторов в соответствии с приведенной выше таблицей, за исключением того, что круглые скобки () всегда имеют наивысший приоритет. Поскольку умножение имеет более высокий приоритет, чем сложение, это означает, например, что 1 + 2 * 3 оценивается как 7, а (1 + 2) * 3 оценивается как 9. Обратите внимание также, что –3 ** 2 оценивается как – 9, а не 9.

Когда операторы с одинаковым приоритетом встречаются в одном и том же выражении, они, за одним исключением, всегда оцениваются слева направо, поэтому `1/2 * A` оценивается как `(1/2) * A`, а не `1/(2 * A)`.

Исключением из правил приоритета является то, что в выражении формы `A ** B ** C` сначала оценивается правосторонняя операция `B ** C`.

*Целочисленное деление*

Это деление вызывает столько душевной боли у ничего не подозревающих новичков, что заслуживает отдельного раздела. Когда

целая величина (константа, переменная или выражение) делится на другую целочисленную величину, результат также имеет целочисленный тип, поэтому он усекается до нуля, т.е. теряется дробная часть. Например:

10 / 3 оценивается как 3
19 / 4 оценивается как 4
4 / 5 оценивается как 0 (что может вызвать нежелательное деление на ноль)
–8 / 3 оценивается как -2
3 * 10 / 3 оценивается как 10
10 / 3 * 3 оценивается как 9

*Смешанные выражения*

Fortran 90 позволяет операндам в выражении быть разных типов. Общее правило состоит в том, что более слабый или простой тип преобразуется или принуждается к более сильному типу. Поскольку целочисленный тип является самым простым, это означает, что операции с вещественными и целочисленными операндами будут выполняться в действительной арифметике. Это относится к каждой операции в отдельности, не обязательно к выражению в целом. Так, например,

10 / 3.0 оценивается как 3.33333
4. / 5 оценивается как 0.8
2 ** (–2) оценивается как 0 (почему?)

Однако обратите внимание, что

3 / 2 / 3.0 оценивается как 0.33333, т.к. сперва оценивается 3 / 2, что дает целое число 1.

## 2.7 Числовое присвоение

Цель числового присваивания — вычислить значение числового выражения и присвоить его переменной. Его общая форма

*variable = expr*

Знак равенства не имеет того же значения, что и знак равенства в математике, и его следует читать как «становится». Итак, выпажение

```
X = A + B
```

следует читать как «(содержимое) X становится (содержимым) A плюс (содержимое) B». Таким образом, присваивание

```
N = N + 1
```

имеет смысл и означает «увеличение значения N на 1».

Если *expr* не имеет того же типа, что и *variable*, перед присваиванием оно преобразуется в этот тип. Это означает, что может быть потеря точности. Например, предположим, что N — целое число, а X и Y — вещественные:

```
N = 10. / 3     (значение N равно 3)
X = 10 / 3      (значение X равно 3,0)
Y = 10 / 3.     (значение Y равно 3,33333)
```

Опасность непреднамеренного выполнения целочисленного деления трудно переоценить. Например, вы можете захотеть усреднить две оценки, которые являются целыми числами M1 и M2. Самое естественное утверждение, которое нужно написать, это

```
FINAL = (M1 + M2) / 2
```

но при этом теряется десятичная часть среднего. Всегда безопаснее писать константы как вещественные числа, если вам нужна действительная арифметика:

```
FINAL = (M1 + M2) / 2.0
```

*Примеры*

Рассмотрим математические формулы:

$$f = gm\frac{e}{r^2}$$
$$c = \frac{\left(A^2 + B^2\right)^{0.5}}{2A} \qquad (2.2)$$
$$A = P\left(1 + \frac{r}{100}\right)^n$$

Формулы (2.2) могут быть переведены в следующие присваивания Фортрана:

```
F = G * M * E / R ** 2
C = (A ** 2 + B ** 2) ** 0.5 / (2 * A)
```

```
A = P * (1 + R / 100) ** N
```

Второе также может быть записано с помощью встроенной функции SQRT как

```
C = SQRT ( A ** 2 + B ** 2 ) / (2 * A)
```

но никогда как

```
C = (A ** 2 + B ** 2) ** (1/2) / (2 * A)
```

т.к. 1/2 в показателе степени оценивается как ноль из-за целочисленного деления.

## 2.8 Простой ввод и вывод

В этом разделе мы более подробно рассмотрим операторы READ* и PRINT*. Процесс получения информации в компьютер и из него является аспектом того, что называется передачей данных. Простейшая форма передачи данных в Fortran 90 — это READ* и PRINT*, и она называется направленной по списку. Более продвинутые формы передачи данных будут обсуждаться в главе 10.

*Ввод*

До сих пор в этой главе переменным присваивались значения с помощью операторов числового присваивания, как в программе MONEY:

```
BALANCE = 1000
RATE = 0.09
```

Это негибкий способ предоставления данных, поскольку для запуска программы для различных остатков или процентных ставок приходится изменять эти параметры и перекомпилировать программу. В более сложной программе таких назначений может быть много, и перекомпилировать их каждый раз, когда вы хотите изменить данные, — пустая трата времени. Однако оператор READ* позволяет предоставлять данные во время работы программы. Замените эти два оператора присваивания одним оператором READ*:

```
READ*, BALANCE, RATE
```

Когда вы запускаете программу, компилятор будет ждать, пока вы наберете значения двух переменных с клавиатуры, если вы используете ПК (персональный компьютер). Они могут находиться на

одной строке, разделенной пробелом, запятой или косой чертой, или на разных строках. Вы можете исправить число клавишей *Backspace (пробел)* при его вводе. Если вы используете какую-либо другую систему, вам может понадобиться совет о том, как предоставлять данные для READ*.

Общая форма оператора READ* такова:

```
READ*, list
```

где *list* – список переменных, разделенных запятыми.

Обратите внимание на следующие общие правила:

- Одна строка ввода или вывода называется записью (например, в случае ПК, с клавиатуры или на экране).
- Для каждого оператора READ требуется новая входная запись. Например, оператор

  ```
  READ*, A, B, C
  ```

  удовлетворится одной записью, содержащей три значения:

  ```
  3 4 5
  ```

  тогда как заявления

  ```
  READ*, A
  READ*, B
  READ*, C
  ```

  требуют трех входных записей, каждая из которых содержит одно значение:

  ```
  3
  4
  5
  ```

- Когда компилятор встречает новый READ, непрочитанные данные в текущей записи отбрасываются, и компилятор ищет новую запись для предоставления данных.
- Данные для READ могут попасть в последующие записи. В основном компилятор ищет данные во всех входных записях до тех пор, пока список ввода/вывода (ввод/вывод) не будет удовлетворен.
- Если данных недостаточно для выполнения команды READ, программа аварийно завершает работу с сообщением об ошибке.

*Пример*

Операторы

```
READ*, A
READ*, B, C
READ*, D
```

с входными записями

```
1 2 3
4
7 8
9 10
```

имеют тот же эффект, что и присваивания

```
A = 1
B = 4
C = 7
D = 9
```

*Чтение данных из текстовых файлов*

Часто бывает, что нужно протестировать программу, прочитав много данных. Предположим, вы пишете программу для нахождения среднего значения, скажем, 10 чисел. Вводить 10 чисел каждый раз при запуске программы становится большой неприятностью (поскольку программы редко работают правильно с первого раза). Следующий трюк очень полезен.

Идея состоит в том, чтобы поместить данные в отдельный (внешний) файл, который хранится в вашей компьютерной системе, например, на жестком диске, если вы используете ПК. Затем программа считывает данные из файла при каждом запуске, а не с клавиатуры ПК. Например, используйте текстовый процессор для сохранения следующей строки в ASCII (текстовом) файле с именем DATA:

```
3 4 5
```

Теперь используйте эту программу, чтобы прочитать эти три числа из файла и отобразить их на экране:

```
OPEN( 1, FILE = 'DATA' )
READ(1, *) A, B, C
PRINT*, A, B, C
END
```

Оператор `OPEN` связывает номер устройства (1) с внешним файлом DATA. Показанная здесь форма инструкции `READ` указывает компилятору искать в файле, подключенном к устройству 1, его данные (номер устройства обычно может быть от 1 до 99).

*Вывод*

Оператор `PRINT*` очень полезен для вывода небольших объемов данных, обычно при разработке программы, поскольку вам не нужно заботиться о точных деталях формы вывода.

Общая форма

PRINT*, *list*

где список может быть списком констант, переменных, выражений и строк символов, разделенных запятыми. Строка символов представляет собой последовательность символов, разделенных кавычками (") или апострофами (').

```
PRINT*, "The square root of", 2, 'is', SQRT( 2.0 )
```

Вот несколько общих правил:

- Каждый оператор `PRINT*` создает новую выходную запись.
- Способ печати действительных чисел зависит от вашей конкретной системы. Если вы хотите самостоятельно определять способ печати, вы должны использовать спецификации формата *format*. Например, следующие операторы будут печатать число 123,4567 в форме с фиксированной точкой в 8 столбцах с точностью до двух знаков после запятой:

```
    X = 123.4567
    PRINT 10, X
10 FORMAT( F8.2 )
```

- Если строка символов в `PRINT*` слишком длинная и не помещается на одной строке, она будет отображаться без разрыва, если использовать & для продолжения строки:

```
PRINT*, 'Now is the time for all go&
&od men to come to the aid of the party'
```

*Отправка вывода на принтер*

Это можно сделать следующим образом (на ПК):

```
OPEN( 2, FILE = 'prn' )
WRITE(2, *) 'This is on the printer'
PRINT*, 'This is on the screen'
```

Обратите внимание, что `WRITE` необходимо использовать вместе с номером блока. Это более общее утверждение, чем `PRINT`.

**Резюме**

- Успешное решение задач на компьютере требует знания правил кодирования и четкого логического плана.
- Компилятор переводит операторы программы в машинный код.
- Операторы Фортрана могут иметь длину до 132 символов и могут начинаться с любого места строки.
- Все операторы, кроме присваиваний, начинаются с ключевого слова.
- Токен Фортрана представляет собой последовательность символов, образующих метку, ключевое слово, имя, константу, оператор или разделитель.
- Для улучшения удобочитаемости следует использовать пробелы, за исключением ключевых слов и имен.
- Комментарии печатаются после восклицательного знака! Их следует широко использовать для описания переменных и объяснения того, как работает программа.
- Оператор с последним непустым символом & будет продолжен на следующую строку.
- Существует пять встроенных типов данных: целые, действительные, комплексные, логические и символьные.
- Значения каждого типа данных представлены литеральными константами.
- Целочисленные константы также могут быть представлены в двоичном, восьмеричном и шестнадцатеричном виде.
- Вещественные константы представлены в форме с фиксированной или плавающей запятой (экспоненциальной).
- Буквенно-цифровые символы — это буквы, цифры и знак подчеркивания.
- Имена могут содержать до 31 буквенно-цифрового символа, начиная с буквы.

- Переменная — это символическое имя ячейки памяти.
- Следует использовать оператор `IMPLICIT NONE`, чтобы избежать неявного присвоения типа переменным.
- Числовая переменная должна быть объявлена целочисленной или вещественной в операторе объявления типа.
- Числовые выражения могут быть сформированы из констант и переменных с помощью пяти встроенных числовых операторов, которые действуют в соответствии со строгими правилами приоритета.
- Десятичные части усекаются, когда целые числа делятся или когда целые числа присваиваются действительным числам.
- Числовое присваивание вычисляет значение числового выражения и присваивает его вещественной или целочисленной переменной.
- Группам переменных могут быть присвоены начальные значения в операторе DATA.
- `PRINT*` используется для печати (отображения) вывода.
- `READ*` используется для ввода данных с клавиатуры во время работы программы.
- Данные также можно считывать из внешнего файла (например, из файла на диске).

**Упражнения**

2.1 Оцените следующие числовые выражения, учитывая, что `A = 2`, `B = 3`, `C = 5` (действительные числа); и `I = 2`, `J = 3` (целые числа). Ответы даны в скобках.

```
A * B + C                    (11.0)
A * (B + C)                  (16.0)
B / C * A                    (1.2)
B / (C * A)                  (0.3)
A / I / J                    (0.333333)
I / J / A                    (0.0)
A * B ** I / A ** J * 2      (4.5)
C + (B / A) ** 3 / B * 2.    (7.25)
A ** B ** I                  (512.0)
-B ** A ** C                 (-45.0)
J / (I / J)                  (деление на ноль)
```

2.2 Решите, какие из следующих констант не записаны в стандартном Фортране, и объясните, почему:

| | |
|---|---|
| (a) 9,87 | (e) 3.57*E2 |
| (b) .0 | (f) 3.57E2.1 |
| (c) 25.82 | (g) 3.57E+2 |
| (d) –356231 | (h) 3,57E–2 |

2.3 Покажите с указанием причин, какие из следующих имен переменных не являются именами переменных Фортрана:

| | | | |
|---|---|---|---|
| (a) A2 | (b) A.2 | (c) 2A | (d) 'A'ONE |
| (e) AONE | (f) X_1 | (g) MiXedUp | (h) Pay Day |
| (i) U.S.S.R. | (j) Pay_Day | (k) min*2 | (l) PRINT |

2.4 Найдите значения следующих выражений, написав короткие программы для их вычисления (ответы в скобках):

| | |
|---|---|
| сумма 5 и 3, деленная на их произведение | (0,53333) |
| кубический корень из произведения 2,3 и 4,5 | (2,17928) |
| квадрат 2π | (39,4784 — принять π = 3,1415927) |

2.5 Переведите следующие выражения на Фортран:

| | |
|---|---|
| (a) $ax^2 + bx + c = 0$ | (b) $F(x) = 0.5 - r(at + bt^2 + ct^3)$ |
| (c) $r = \exp\left(-0.5x^2\right)/\sqrt{2\pi}$ | (d) $t = 1/(1 + 0.3326x)$ |

2.6 В галлоне восемь пинт, а в литре — 1,76 пинты. Объем бака указан как 2 галлона и 4 пинты. Напишите программу, которая считывает этот объем в галлонах и пинтах и переводит его в литры. (Ответ: 11,36 литра)

2.10 Напишите несколько строк на Фортране, которые поменяют содержимое двух переменных A и B, используя только одну дополнительную переменную T.

2.11 Попробуйте решить предыдущую задачу без дополнительных переменных!

2.12 Постарайтесь обнаружить синтаксические ошибки (то есть ошибки в правилах написания кода) в этой программе, прежде чем

запускать ее на компьютере, чтобы сверить свои ответы с сообщениями об ошибках, сгенерированными вашим компилятором:

```
PROGRAM Dread-ful
REAL: A, B, X
X:= 5
Y = 6,67
B = X \ Y
PRINT* 'The answer is", B
END.
```

# Глава 3 Конструкции операторов

До сих пор мы видели, как считывать данные в программу на Фортране, как производить с ними некоторые арифметические действия и как выводить ответы. В этой главе мы рассмотрим две мощные конструкции, которые используются в большинстве реальных программ: DO и IF. Мы также рассмотрим еще два внутренних типа, характерный и сложный, и обсудим концепцию типов.

## 3.1 Циклы DO

Запустите следующую программу:

```
INTEGER I
REAL R

DO I = 1, 10
  PRINT*, I
END DO

END
```

Чтобы вместо этого получить несколько случайных чисел, замените оператор PRINT следующими двумя операторами:

```
CALL RANDOM_NUMBER( R )
PRINT*, R
```

При каждом запуске новой программы вы будете получать одни и те же 10 «случайных» чисел, что довольно скучно. Чтобы увидеть, как каждый раз получать новый случайный набор, вам придется подождать до главы 14.

Для разнообразия попробуйте следующее:

```
DO I = 97, 122
  WRITE( *, 10, ADVANCE = 'NO' )ACHAR( I )
  10    FORMAT( A1 )
END DO
```

Приведенная выше форма оператора WRITE вводит новую функцию, которую с радостью примут опытные мастера: ввод-вывод без продвижения вперед.

Чтобы получить алфавит в обратном порядке, замените DO на

```
DO I = 122, 97, -1
```

Цикл DO (или его эквивалент) является одним из самых мощных операторов в любом языке программирования. Одной из простейших его форм является

```
DO I = J, K
     block
END DO
```

где `I` – целочисленная переменная, `J` и `K` – целочисленные выражения, а `block` означает любое количество операторов. `block` выполняется повторно; значения `J` и `K` определяют количество повторений. В первом цикле `I` принимает значение `J`, а затем увеличивается на 1 в конце каждого цикла (включая последний). Цикл останавливается, как только `I` достигает значения `K`, и выполнение продолжается с оператора после `END DO`. `I` будет иметь значение `K+1` после завершения цикла (обычный выход).

*Вычисление квадратного корня методом Ньютона*

Квадратный корень $x$ из любого положительного числа $a$ можно найти, используя только арифметические операции сложения, вычитания и деления по методу Ньютона. Это итеративная (повторяющаяся) процедура, уточняющая исходное предположение.

Структурный план алгоритма нахождения квадратного корня и программа с примерным выводом для $a = 2$ выглядят следующим образом:

1. Введите $a$
2. Инициализируйте $x$ значением 1
3. Повторите 6 раз (можно и больше):
   Замените $x$ на $(x + a/x)/2$
   Распечатать $x$
4. Стоп.

```
PROGRAM Newton
! Square rooting with Newton

IMPLICIT NONE
REAL    A              ! number to be square rooted
INTEGER I              ! iteration counter
REAL    X              ! approximate square root of A

WRITE(*,10,ADVANCE='NO')'Enter number to be square rooted: '
   10  FORMAT( A )
```

```
READ*, A
PRINT*
X = 1                    ! initial guess (why not?)

DO I = 1, 6
  X = (X + A / X) / 2
  PRINT*, X
ENDDO

PRINT*
PRINT*, 'Fortran 90''s value:', SQRT( A )

END
```

*Вывод на экран:*

```
'Enter number to be square rooted:
```
Введите число для извлечения квадратного корня: 2

```
   1.5000000
   1.4166666
   1.4142157
   1.4142135
   1.4142135
   1.4142135

Fortran 90's value:   1.4142135
```

Значение X сходится к пределу, который равен $\sqrt{a}$. Обратите внимание, что оно идентично значению, возвращаемому встроенной функцией SQRT Fortran 90. Большинство компьютеров и калькуляторов используют аналогичный метод для вычисления квадратных корней и других стандартных математических функций.

*Замечания:*

- использование «приглашения» в операторе WRITE для получения ввода от пользователя — опытные специалисты еще раз отмечают, что ввод-вывод без продвижения позволяет ввод в той же строке, что и приглашение;
- что для печати апострофа (') в строке апостроф должен повторяться (");
- что некоторые пары ключевых слов, такие как ENDDO, не нужно разделять.

## 3.2 Решение с IF-THEN-ELSE

Подробно обсудим оператор IF-THEN-ELSE. В качестве примера предположим, что итоговая оценка студентов, посещающих университетский курс, рассчитывается следующим образом. По окончании курса пишутся две экзаменационные работы. Окончательная оценка представляет собой либо среднюю оценку двух работ, либо оценку как среднее от оценки двух работ и оценки работы в семестре (средняя или единственная оценка за семестр), в зависимости от того, что выше. Следующая программа вычисляет и печатает оценку каждого студента с комментарием «ПРОШЕЛ» (PASS) или «НЕ ПРОШЕЛ» (FAIL) (проходной балл – 50 %).

```
PROGRAM Final_Mark
!Final mark for course based on class record and exams

IMPLICIT NONE
REAL     CRM        ! Class record mark
REAL     ExmAvg     ! average of two exam papers
REAL     Final      ! final mark
REAL     P1         ! mark for first paper
REAL     P2         ! mark for second paper
INTEGER  Stu        ! student counter

OPEN( 1, FILE = 'MARKS' )
PRINT*, ' CRM         Exam Avg    Final Mark'
PRINT*

DO Stu = 1, 3
  READ( 1, * ) CRM, P1, P2
  ExmAvg = (P1 + P2) / 2.0
  IF (ExmAvg > CRM) THEN
    Final = ExmAvg
  ELSE
    Final = (P1 + P2 + CRM) / 3.0
  END IF
  IF (Final >= 50) THEN
    PRINT*, CRM, ExmAvg, Final, 'PASS'
  ELSE
    PRINT*, CRM, ExmAvg, Final, 'FAIL'
  END IF
END DO

END
```

Как объяснялось выше, данные хранятся во внешнем файле (MARKS), чтобы сделать чтение более эффективным. Например, для выборочной группы из трех студентов данные могут быть такими:

```
40 60 43
60 45 43
13 98 47
```

т.е. у первого студента оценка семестра – 40, а экзаменационные оценки – 60 и 43. Его итоговая оценка должна быть 51,5 (оценка семестра не используется), тогда как оценка второго учащегося должна быть 49,3 (оценка семестра используется). Запустите программу, как она есть.

### Конструкция IF

В приведенном выше примере мы видим ситуацию, когда компьютер должен принимать решения: включать или нет оценку семестра, сдал или не сдал студент. Программист не может предвидеть, какая из этих возможностей возникнет при написании программы, поэтому она должна быть спроектирована таким образом, чтобы учесть все из них. Нам нужен условный переход, который является еще одним из самых мощных средств в любом языке программирования. Общая форма конструкции IF, как она называется в Fortran 90, такова:

```
IF condition THEN
   block1
[ELSE
   blockE]
END IF
```

где *condition* – это логическое выражение, имеющее истинное значение, либо ложное, а *block1* и *blockE* – блоки утверждений. Если условие истинно, выполняется *block1* (а не *blockE*), в противном случае выполняется *blockE* (а не *block1*). Часть ELSE необязательна и может быть опущена. Выполнение продолжается в обычном последовательном порядке со следующего оператора после END IF.

Условие может быть сформировано из числовых выражений с операторами отношения, такими как <, <=, == (равно) и /= (не равно), и из других логических выражений с логическими операторами, такими как .NOT., .AND. и .OR...

*Оператор* IF

Более короткой формой конструкции IF является оператор `IF`:

```
IF (condition) statement
```

В этом случае выполняется только один оператор (*statement*), если условие истинно. Ничего не происходит, если оно ложно.

Слово «конструкция» подразумевает конструкцию с более чем одним оператором (и, следовательно, с более чем одним ключевым словом).

## 3.3 Символы

Вопиющим недостатком вышеуказанной программы является то, что имена студентов не читаются и не печатаются. Чтобы исправить это, мы используем *символьные переменные*. Внесите следующие изменения в `Final_Mark`:

Вставьте оператор

```
CHARACTER (Len = 15) Name    ! Name
```

в раздел деклараций программы. Измените оператор, который печатает заголовок:

```
PRINT*, 'Name       CRM       Exam Avg   Final Mark'
```

Измените оператор READ:

```
READ( 1, * ) Name, CRM, P1, P2
```

Измените два оператора, которые печатают отметки:

```
PRINT*, Name, CRM, ExmAvg, Final, 'PASS'
PRINT*, Name, CRM, ExmAvg, Final, 'FAIL'
```

Наконец, измените файл данных MARKS, вставив некоторые имена (не забудьте апострофы):

```
'Ivanov, RJ'      40 60 43
'Smirnov, NX'     60 45 43
'Khaimy, FW'      13 98 47
```

Если вы запустите измененную программу, вы должны получить такой вывод:

| Name | CRM | Exam Avg | Final Mark | |
|---|---|---|---|---|
| Ivanov, RJ | 40.0000000 | 51.5000000 | 51.5000000 | PASS |
| Smirnov, NX | 60.0000000 | 44.0000000 | 49.3333321 | FAIL |
| Khaimy, FW | 13.0000000 | 72.5000000 | 72.5000000 | PASS |

### Символьные константы

До сих пор мы имели дело в основном с двумя внутренними типами Fortran 90: целочисленным и действительным. Теперь мы подошли к характеру внутреннего типа.

Базовая символьная константа представляет собой строку символов, заключенную в пару апострофов (') или кавычек ("). Допускается большинство символов, поддерживаемых компьютером, за исключением "управляющих символов" (например, **escape**). Апострофы и кавычки служат разделителями и не являются частью константы.

Пробел в символьной константе имеет значение, поэтому

```
"B Shakespeare"
```

не то же самое, что

```
"BShakespeare"
```

Fortran 90 "чувствителен к регистру" только в случае символьных констант, поэтому

```
Charlie Brown
```

не то же самое, что

```
CHARLIE BROWN
```

Есть два способа представления самих символов-разделителей в символьной константе. Разделитель любого типа может быть встроен в строку, разделенную другим типом, как в

```
'Professor said, "What is truth?"'
```

В качестве альтернативы разделитель должен повторяться, как в

```
'Student ansved, ''It is Universe'''
```

Строка символов может быть пустой, т.е. ' ' или " ". Количество символов в строке называется ее длиной. Пустая строка имеет нулевую длину.

### Символьные переменные

Оператор

```
CHARACTER LETTER
```

объявляет LETTER символьной переменной длиной 1, т.е. может содержать один символ. Можно объявлять более длинные символы, как в программе Final_Mark:

```
CHARACTER (Len = 15) Name
```

Это означает, что символьная переменная Name может содержать строку длиной до 15 символов.

Альтернативная форма декларации

```
CHARACTER Name*15
```

Символьные константы могут быть присвоены переменным очевидным образом:

```
Name = 'V. Ivanov'
```

Во входной записи разделители кавычек или апострофов не нужны для символьной константы, если константа не содержит пробела, запятой или косой черты. Поскольку имена в приведенном выше примере содержат запятые и пробелы, во входном файле необходимы разделители.

## 3.4 Именованные константы

В программе Vertical, рассмотренной в главе 2, оператор декларации

```
REAL, PARAMETER :: G = 9.8
```

использовался для объявления G как именованной константы или параметра. Следствием этого является то, что G нельзя изменить позже в программе — любая попытка сделать это вызовет сообщение об ошибке.

Атрибут PARAMETER является одним из многих, которые могут быть указаны в операторе объявления типа. Дополнительные атрибуты будут введены позже.

Именованные константы сами могут использоваться при инициализации. Сформированное таким образом выражение является выражением инициализации (выражения инициализации на самом деле являются частными случаями константных выражений, которые могут появляться в других контекстах). Например,

```
REAL, PARAMETER :: Pi = 3.141593
INTEGER, PARAMETER :: Two = 2
REAL, PARAMETER :: OneOver2Pi = 1 / (2 * Pi)
REAL, PARAMETER :: PiSquared = Pi ** Two
```

Поскольку выражения инициализации задаются во время компиляции, на их форму накладываются определенные ограничения. На данном этапе актуальными являются:

– они могут включать только встроенные операторы;
– оператор возведения в степень должен иметь целую степень;
– встроенные функции должны иметь целочисленные или символьные аргументы и результаты.

Поэтому следующее не допускается, учитывая приведенное выше определение `Pi`:

```
REAL, PARAMETER :: OneOverRoot2Pi = 1 / SQRT(2 * Pi)
```

Как правило, двойное двоеточие должно стоять везде, где указывается атрибут или используется выражение инициализации; в противном случае это необязательно. Если указан атрибут PARAMETER, должно появиться выражение инициализации. Если именованная константа имеет символьный тип, ее длина может быть объявлена звездочкой. Фактическая длина затем определяется компилятором, избавляя вас от необходимости подсчитывать все символы. Например,

```
CHARACTER (LEN = *), PARAMETER &
 :: Message = 'Press ENTER to continue'
LEN =*,character constantcharacter constant: LEN = *
```

## 3.5 Разряды параметров и переменных

Концепция разряда – это новая функция Fortran 90, с которой опытным программистам Фортрана необходимо будет разобраться.

У каждого из пяти встроенных типов есть разряд (*kind*) по умолчанию – это требуется стандартом. Может быть несколько других разрядов – они будут системно-зависимыми и не указаны в стандарте. С каждым видом связано неотрицательное целое число, называемое разрядом параметра типа. Значение параметра *kind* позволяет вам идентифицировать различные доступные типы.

Р а з р я д ы  ц е л ы х  ч и с е л

Например, компилятор FTN90 поддерживает на ПК три целочисленных разряда. Значение параметра `kind` по умолчанию равно 3 и представляет целые числа в диапазоне от $-2^{31}$ до $2^{31} - 1$ .

Существует ряд встроенных функций, которые позволяют вам устанавливать свойства, связанные с типом, и, что более важно, указывать разряд, который будет соответствовать вашим требованиям к точности. Целочисленные константы автоматически имеют разряд по умолчанию (именно это означает слово по умолчанию).

*Функция* `KIND(I)` KIND возвращает значение параметра вида своего аргумента (вещественное или целочисленное), поэтому `KIND(0)` вернет разряд целого числа по умолчанию. Простое задание

```
INTEGER I
```

указывает `I` с разрядом целого числа по умолчанию.

*Функция* `HUGE(I)` возвращает наибольшее значение, представленное ее аргументом (действительным или целым числом). Чтобы найти наименьшее значение, просто добавьте 1 и распечатайте результат. Значения целочисленного вида циклически перемещаются между своей нижней и верхней границами. Следующий фрагмент установит разряд целого числа по умолчанию, а также верхнюю и нижнюю границы:

```
INTEGER I

BIG = HUGE(I)
SMALL = BIG + 1
PRINT*, 'Default kind: ', KIND(I)
PRINT*, 'Largest:      '
PRINT*, 'Smallest:     ', SMALL
```

Обратите внимание, что аргументы `KIND` и `HUGE` не нужно определять. Установив значение параметра `kind` по умолчанию, вы можете немного поэкспериментировать, чтобы установить другие типы, доступные в вашем компиляторе. Например, оператор

```
INTEGER ([KIND =] 2) I
```

задает `I` с параметром разряда, равным 2, точнее, с разрядом параметра типа (содержание квадратных скобок является необязательным).

*Функция* `SELECTED_INT_KIND(N)` возвращает значение параметра разряда для вида, который сможет представлять все целочисленные значения в диапазоне $-10^N$ до $10^N$. Эта функция может использоваться для определения того, какие типы доступны:

```
INTEGER K, N
N = 0

DO
  N = N + 1
  K = SELECTED_INT_KIND( N )
  IF( K == -1 ) EXIT
  PRINT*, N, K
END DO

END
```

Числовое значение разряда параметров зависит от системы. То есть, в то время как разряды целых чисел равны 1, 2 и 3 в одном компиляторе, в другом компиляторе их значения могут быть 2, 4 и 8, хотя типы могут иметь идентичные свойства. Это поднимает вопрос переносимости – можем ли мы написать программу, которая задает определенный разряд, и которая будет работать на любом компиляторе, поддерживающем стандарт? Ответ заключается в том, что мы можем, используя функцию SELECTED_INT_KIND, назвать константу, которая, в свою очередь, используется в операторе объявления типа:

```
INTEGER, PARAMETER :: K6 = SELECTED_INT_KIND( 6 )
INTEGER (K6) I
```

Правая часть первого оператора является примером константного выражения. Это гарантирует, что разряд K6 сможет представлять все целые числа в диапазоне от -999999 до 999999 (и, возможно, больше). Хотя константы имеют тип по умолчанию, другой тип может быть указан, если после константы поставить знак подчеркивания и целочисленную константу без знака или именованную целочисленную константу, например.

```
123_2   123456_K6
```

указывает 123 с разрядом 2 (который зависит от системы), а 123456 указывается с разрядом K6, выбранным в объявлении выше (который является переносимым). Очевидно, что переносная форма более безопасна в использовании и поэтому рекомендуется. При вычислении выражений, в которых операнды имеют разные значения параметра рода, результат имеет параметр вида операнда с большей точностью.

### Разряды вещественных чисел

Функции KIND и HUGE, описанные выше, также принимают реальные аргументы. При использовании реального типа попытка выйти за пределы HUGE вызывает ошибку переполнения.

Функция SELECTED_REAL_KIND(P, R) возвращает параметр kind реального типа с точностью P (количество значащих десятичных знаков) и диапазоном экспоненты не менее $10^{-R}$ до $10^{R}$ (если доступно). P и R должны быть целыми числами. Разряд вещественной константы может быть указан так же, как целочисленная константа.

Стандарт требует, чтобы в дополнение к вещественному типу по умолчанию должен быть хотя бы один вещественный тип с большей точностью, чем значение по умолчанию (это соответствует ныне устаревшему типу DOUBLE PRECISION в более ранних версиях). Если это более точное представление имеет, например, значение параметра вида, равное 2, программа AIDS в главе 1 может быть изменена путем замены оператора REAL на

```
REAL (KIND=2)  A    ! number of cases
```

и числовое присвоение с

```
A = 174.6 * (T - 1981.2_2) ** 3
```

Обратите внимание, что для получения значительно отличающегося ответа выражение должно быть приведено к более сильному типу. Запустите программу, чтобы увидеть, как отличается ответ.

### Разряды символьных величин

Символьная константа по умолчанию включает все символы, поддерживаемые компьютерной системой, за исключением управляющих символов. Стандарт требует, чтобы разряд по умолчанию удовлетворял определенной последовательности. Это необходимо для сортировки символов.

Другие наборы символов (например, греческий) могут поддерживаться вашей системой, и они будут иметь другие параметры разряда. В случае символьных констант параметр разряда, если он есть, предшествует константе. Таким образом, если бы именованные константы ASCII и GREEK имели значения по умолчанию и греческий тип, константы этих типов могли бы быть записаны как

```
ASCII_"abcde"
GREEK_"αβγδε"
```

В случае с целыми и вещественными числами выше мы видели, что параметр разряда может быть указан с параметром типа. Поскольку длина символьной переменной также может быть указана при объявлении, символ – единственный тип, имеющий два параметра разряда: один для длины и один для вида. Примеры:

```
CHARACTER (LEN = 10, KIND = GREEK) Greek_Word
CHARACTER (LEN = 10) English_Word       ! default kind
CHARACTER  (KIND = GREEK)Greek_Letter   ! default length of 1
CHARACTER (10, GREEK ) Greek_Word
```

Обратите внимание, что спецификаторы "LEN=" и "KIND=" являются необязательными. Однако, если указан только один безымянный параметр, он принимается за длину, а не за тип. Функция KIND также принимает символьный аргумент.

## 3.6 Комплексный тип данных

Комплексные числа и комплексная арифметика поддерживаются Fortran 90. Например,

```
COMPLEX, PARAMETER :: i = (0, 1)    ! sqrt(-1)
COMPLEX X, Y
X = (1, 1)
Y = (1, -1)
PRINT*, CONJG(X), i * X * Y
```

Вывод программы:

```
(  1.0000000, -1.0000000) (  0.0000000E+00,  2.0000000)
```

Когда комплексная константа вводится с помощью READ*, она должна быть заключена в круглые скобки.

Многие встроенные функции могут принимать сложные аргументы.

## 3.7 Введение во внутренние функции

Пока что вы должны быть в состоянии написать программу, которая получает данные в компьютер, выполняет простые арифметические операции над данными и выводит результаты вычислений в понятной форме. Однако более интересные задачи, вероятно, связаны со специальными математическими функциями,

такими как синусы, косинусы, логарифмы и т.д. Так же, как большинство калькуляторов имеют клавиши для этих функций, Фортран позволяет вычислять многие функции напрямую. Эти функции называются внутренними (или встроенными) функциями.

*Программа движения китобойного гарпуна*

Давайте напишим программу для вычисления положения (координаты $x$ и $y$) и скорости (величина и направление) китобойного гарпуна, учитывая $t$, время с момента запуска, $u$, скорость запуска, $a$ – начальный угол запуск (в градусах) и $g$ – ускорение свободного падения.

Горизонтальные и вертикальные перемещения определяются по формулам

$$x = ut\cos a, \quad y = ut\sin a - gt^2/2$$

Скорость имеет такую величину $V$, что $V^2 = \sqrt{V_x^2 + V_y^2}$, где ее горизонтальная и вертикальная составляющие $V_x$ и $V_y$ равны

$$V_x = u\cos a, \quad V_y = u\sin a - gt$$

и $V$ образует с землей угол θ, такой что $\tan(\theta) = V_x/V_y$. Программа:

```
PROGRAM Projectile
IMPLICIT NONE

REAL, PARAMETER :: g = 9.8 ! acceleration due to gravity
REAL, PARAMETER :: Pi =3.1415927 ! a well-known constant

REAL A      ! launch angle in degrees
REAL T      ! time of flight
REAL Theta  ! direction at time T in degrees
REAL U      ! launch velocity
REAL V      ! resultant velocity
REAL Vx     ! horizontal velocity
REAL Vy     ! vertical velocity
REAL X      ! horizontal displacement
REAL Y      ! vertical displacement

READ*, A, T, U
A = A * Pi / 180       ! convert angle to radians
X = U * COS( A ) * T
Y = U * SIN( A ) * T - g * T * T / 2.
```

```
Vx = U * COS( A )
Vy = U * SIN( A ) - g * T
V = SQRT( Vx * Vx + Vy * Vy )
Theta = ATAN( Vy / Vx ) * 180 / Pi
PRINT*, 'x: ', X, 'y: ', Y
PRINT*, 'V: ', V, 'Theta: ', Theta
END
```

Если вы запустите эту программу с данными

```
45  6  60
```

по отрицательному значению θ вы увидите, что снаряд падает. Аргументом функции может быть любое допустимое выражение Фортрана соответствующего типа, включая другую функцию. Таким образом, *V* можно было бы вычислить непосредственно следующим образом:

```
V = SQRT( (U * COS( A )) ** 2 + (U * SIN( A ) - g*T)**2)
```

Аргумент `SQRT` здесь всегда положителен (почему?), поэтому никаких проблем возникнуть не может.

Углы для тригонометрических функций должны быть выражены в радианах и возвращаются в радианах, где это уместно. Чтобы преобразовать градусы в радианы, умножьте угол в градусах на π/180, где π – известное трансцендентное число 3,1415926... Однако, если вы хотите произвести впечатление на своих друзей, вы можете хитро использовать математический факт, что дуга тангенс (арктангенс) 1 равен π/4, и используйте функцию ATAN (попробуйте).

### Некоторые полезные встроенные функции

Приведем здесь список некоторых из наиболее распространенных встроенных функций. *X* означает действительное выражение, если не указано иное. Необязательные аргументы указаны в квадратных скобках.

- `ABS(X)`: абсолютное значение целого, действительного или комплексного числа *X*.
- `ACOS(X)`: арккосинус arcCOS *X*.
- `ASIN(X)`: арксинус arcSIN *X*.
- `ATAN(X)`: арктангенс arctg *X* в диапазоне от -π/2 до π/2.
- `COS(X)`: Косинус действительного или комплексного *X*.
- `COSH(X)`: гиперболический косинус ch *X*.

- `COT(X)`: котангенс ctg *X*.
- `EXP(X)`: значение экспоненциальной функции $e^x$, где *X* может быть действительным или комплексным.
- `INT(X [,KIND])`: INT преобразует целое, действительное или сложное число *X* в целочисленный тип с усечением до нуля, например. `INT(3.9)` возвращает 3, `INT(–3.9)` возвращает –3. Если присутствует необязательный аргумент KIND, он определяет значение параметра вида результата. В противном случае результат имеет целочисленный вид по умолчанию.
- `LOG(X)`: натуральный логарифм вещественного или комплексного ln *X*. Обратите внимание, что целочисленный аргумент вызовет ошибку.
- `LOG10(X)`: логарифм по основанию 10 lg *X*.
- MAX(X1, X2[, X3, ...]): MAX максимум два или более целых или вещественных аргумента.
- `MIN(X1, X2[, X3, ...])`: минимум из двух или более целых или действительных аргументов.
- `MOD(K, L)`: остаток при делении *K* на *L*. Аргументы должны быть как целыми, так и вещественными.
- `NINT(X [,KIND])`: целое число, ближайшее к *X*, например, `NINT(3.9)` возвращает 4, а `NINT(–3.9)` возвращает –4.
- `REAL(X [,KIND])`: функция `REAL` преобразует целое, действительное или сложное число *X* в вещественный тип, например. `REAL(2)/4` возвращает 0,5, тогда как `REAL(2/4)` возвращает 0,0.
- `SIN(X)`: синус действительного или комплексного *X*.
- `SINH(X)`: гиперболический синус sh *X*.
- `SQRT(X)`: квадратный корень из действительного или комплексного *X*.
- `TAN(X)`: Тангенс от X.
- `TANH(X)`: гиперболический тангенс для *X*.

### Внутренние подпрограммы

Fortran 90 также имеет ряд встроенных подпрограмм. Подпрограммы немного отличаются от функций тем, что они вызываются оператором `CALL`, а результаты возвращаются через аргументы. В приведенном ниже примере показано, как можно

отобразить дату и время. Он также иллюстрирует использование подстрок символов и конкатенации.

```
DATE_AND_TIMECHARACTER*10 DATE, TIME, PRETTY_TIME
CALL DATE_AND_TIME( DATE, TIME )
PRINT*, DATE
PRETTY_TIME =TIME(1:2)//':'//TIME(3:4)//':'// TIME(5:10)
PRINT*, PRETTY_TIME
END
```

*Вывод программы:*

```
 19930201
 15:47:23.0
```

**Резюме**

- Цикл DO используется для повторения блока (набора) операторов.
- Конструкция IF-THEN-ELSE позволяет программе выбирать между альтернативами.
- Оператор `IF` представляет собой более короткую форму конструкции IF.
- Символьные константы – это строки символов, заключенные в апострофы (') или кавычки (").
- Именованные константы (параметры) нельзя изменять в программе.
- Переменные могут быть инициализированы в объявлении типа.
- Каждый из встроенных типов данных имеет тип по умолчанию и количество других типов, зависящее от системы.
- Параметр разряда типа, связанный с типом данных, представляет собой целое число, которое оценивается как разряд этого типа данных.
- Значение параметров типа `kind` зависит от системы.
- Символы могут иметь два параметра типа в своих объявлениях типа: один для длины и один для вида.
- Комплексные числа и арифметика поддерживаются встроенным типом `COMPLEX`.
- Комплексные константы должны быть заключены в круглые скобки для ввода с помощью `READ*`.

- Внутренние (встроенные) функции могут использоваться для непосредственного вычисления множества математических, тригонометрических и других функций.

***Упражнения***

3.1 Переведите следующие выражения в операторы Фортрана:

(а) Добавьте 1 к значению *I* и сохраните результат в *I*.
(б) Куб *I*, добавьте к нему *J* и сохраните результат в *I*.
(в) Установите *G* равным большей из двух переменных *E* и *F*.
(г) Если *D* больше нуля, приравняйте *X* к минус *B*.
(д) Разделите сумму *A* и *B* на произведение *C* и *D* и сохраните результат в *X*.

3.2 Если *C* и *F* представляют собой температуры в градусах Цельсия и Фаренгейта соответственно, формула преобразования градусов Цельсия в градусы Фаренгейта будет выглядеть так: F = 9C/5 + 32.

(а) Напишите программу, которая запросит у вас температуру в градусах Цельсия и отобразит эквивалентную температуру в градусах Фаренгейта с каким-либо комментарием, например,

```
The Fahrenheit temperature is: ...
```

Попробуйте при следующих температурах по Цельсию (ответы в скобках): 0 (32), 100 (212), –40 (–40!), 37 (нормальная человеческая температура: 98,6).

(б) Измените программу, чтобы она использовала цикл DO для вычисления и записи эквивалента Фаренгейта температур Цельсия в диапазоне от 20° до 30° с шагом в 1°.

3.3 Напишите программу, которая отображает список целых чисел от 10 до 20 включительно, рядом с каждым из которых стоит квадратный корень.

3.4 Напишите программу для поиска и отображения суммы последовательных целых чисел 1, 2, ..., 100. (Ответ: 5050)

3.5 Напишите программу для поиска и отображения суммы последовательных четных целых чисел 2, 4, ..., 200. (Ответ: 10100)

3.6 Десять студентов в группе пишут тест. Оценки из 10. Все оценки заносятся во внешний файл MARKS. Напишите программу, которая будет считывать все десять оценок из файла, находить и отображать среднюю оценку. Попробуйте на следующих метках (каждая на отдельной строке в файле):

```
5  8  0  10  3  8  5  7  9  4  (Ответ: 5.9)
```

3.7 Проходной балл за тест в предыдущей задаче — 5 из 10. Измените свою программу, чтобы она использовала конструкцию IF-THEN, чтобы узнать, сколько студентов прошли тест.

3.8 Напишите программу, которая генерирует несколько случайных чисел *R* с

```
CALL RANDOM_NUMBER( R )/
```

и подсчитывает, сколько из них больше 0,5, а сколько меньше 0,5. Попробуйте увеличить количество генерируемых случайных чисел. Что вы ожидаете?

3.9 Каковы значения *X* и *A* (оба действительные) после выполнения следующего раздела программы?

```
A = 0
I = 1
X = 0
A = A + I
X = X + I / A
A = A + I
X = X + I / A
A = A + I
X = X + I / A
A = A + I
X = X + I / A
```

3.10 Перепишите программу из предыдущего упражнения более экономично, используя цикл DO.

3.11 Обработайте вручную вывод следующей программы:

```
PROGRAM Mystery
REAL S, X
INTEGER N, K

N = 4
```

```
S = 0

DO K = 1, N
  X = K
  S = S + 1 / (X * X)      ! faster than X ** 2
END DO

PRINT 10, Sqrt( 6 * S )
10  FORMAT( F10.6 )

END
```

Если вы запускаете эту программу для все больших и больших значений *N*, вы обнаружите, что результат приближается к хорошо известному пределу.

3.18 В метре 39,37 дюйма, в футе – 12 дюймов, а в ярде – три фута. Напишите программу для чтения длины в метрах (которая может иметь десятичную часть) и преобразования ее в ярды, футы и дюймы. (Проверьте: 3,51 метра преобразуются в 3 ярда 2 фута 6,19 дюйма.)

3.19 Напишите несколько операторов Фортрана, которые будут:

а) найти длину *C* гипотенузы прямоугольного треугольника через длины *A* и *B* двух других сторон;
б) найдите длину *C* стороны треугольника, зная длины двух других сторон *A* и *B* и величину в градусах прилежащего к ней угла θ, используя правило косинусов:

$$C^2 = A^2 + B^2 - 2AB\cos\theta$$

3.20 Переведите следующие формулы в выражения Фортрана:

(а) $\log\left(x + x^2 + a^2\right)$
(б) $\left(e^{3t} + t^2 \sin 4t\right)\cos^2 3t$
(в) $4\arctan 1$
(г) $\sec^2 x + \cot y$
(д) $\cot^{-1}\left(x/a\right)$

# Глава 4 Подготовка программы

До сих пор наши примеры были очень простыми с логической точки зрения, поскольку мы концентрировались на технических аспектах правильного написания операторов Фортрана. Однако настоящие задачи намного сложнее, и для успешного программирования нам необходимо тщательно понять проблему и разбить ее на наиболее фундаментальные логические этапы. Другими словами, мы должны разработать систематическую процедуру или алгоритм для решения проблемы. Существует ряд методов, которые помогают в этом процессе разработки алгоритма. В этой главе мы наметим два из них: блок-схемы и структурные планы, о которых уже упоминалось вкратце.

## 4.1 Блок-схемы

Этот подход довольно старомоден и, как правило, вызывает неодобрение в определенных компьютерных кругах. Однако инженеры часто предпочитают этот визуальный метод, поэтому по этой причине и для исторического интереса здесь приведены некоторые примеры.

Предположим, мы хотим написать программу для преобразования температуры по шкале Фаренгейта (где вода замерзает и кипит при 32° и 212° соответственно) в более знакомую шкалу Цельсия. Блок-схема проблемы представлена на рисунке 4.1.

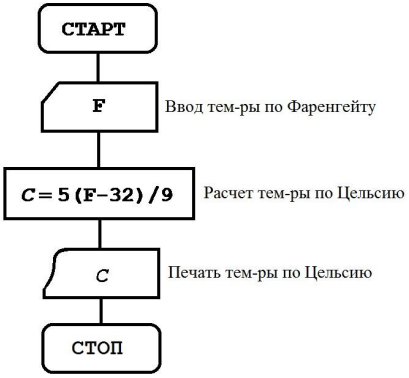


Рисунок 4.1 Преобразование Фаренгейта в Цельсий

Основные символы, используемые в блок-схемах, поясняются в таблице 4.1.

Таблица 4.1 Элементы блок-схемы

| **Фигура** | **Обозначение** |
|---|---|
| | Старт / Стоп |
| | Ввод данных |
| | Процесс вычисления |
| | Циклический процесс |
| | Комплекс условного перехода |
| | Вывод на печать или внешние носители |

*Квадратное уравнение*

Когда вы учились в школе, вы, вероятно, решали сотни квадратных уравнений вида

$$ax^2 + bx + c = 0$$

Полный алгоритм нахождения решения (решений) $x$ при любых значениях $a$, $b$ и $c$ показан на рис. 4.2.

*Метод Ньютона для извлечения квадратного корня*

В главе 3 мы написали программу `Newton` для вычисления квадратных корней, в которой использовался цикл DO. Не существует общепринятого способа составления блок-схемы для цикла DO. Используем наиболее простой способ, при котором выполняется блок операторов в цикле (тело цикла), как показано на рисунке 4.3. Обратите внимание, что содержимое полей может быть либо операторами Фортрана, либо более общими математическими выражениями.

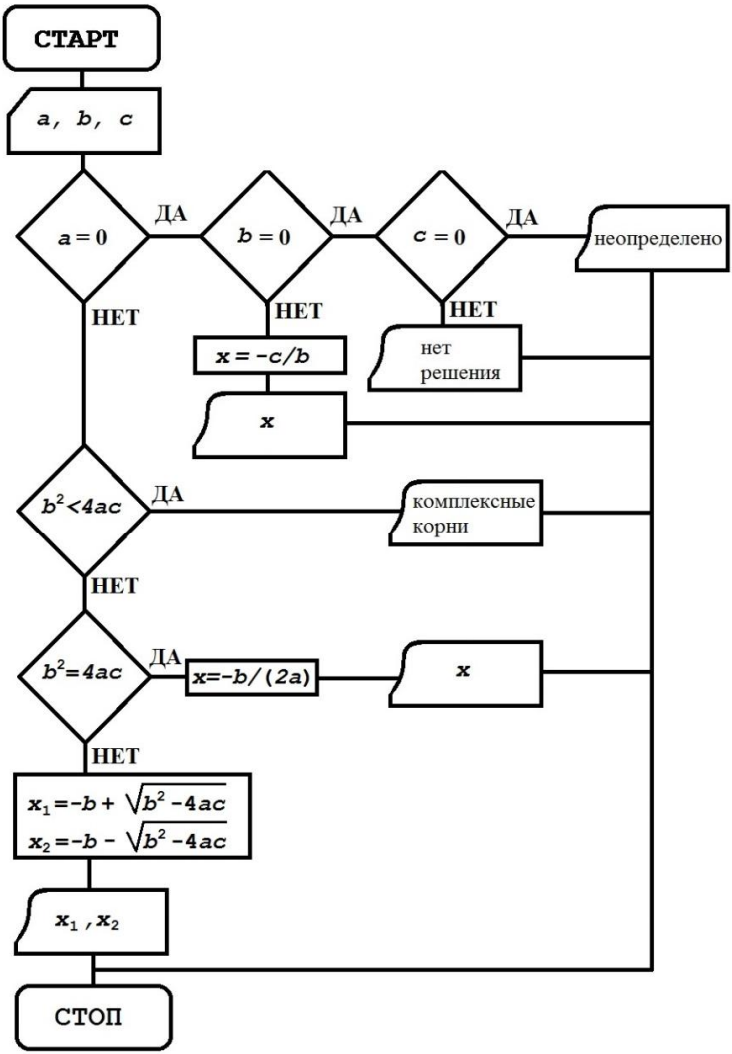


Рисунок 4.2 Блок-схема квадратного уравнения

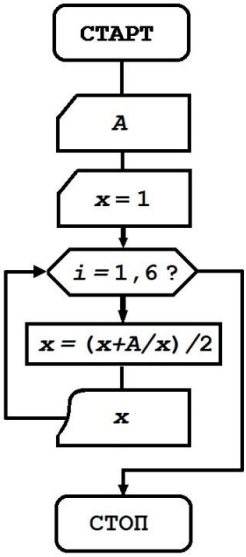


Рисунок 4.3 Метод Ньютона для извлечения квадратного корня

## 4.2 Структурные планы

Это альтернативный метод подготовки программы, который имеет преимущества, когда эквивалентная блок-схема становится достаточно большой. Это пример того, что называется псевдокодом. План может быть написан на нескольких уровнях, каждый из которых имеет возрастающую сложность по мере развития логической структуры программы. Например, план первого уровня задачи преобразования температуры на рис. 4.1 выше может быть простой постановкой задачи:

1. Прочитайте температуру по Фаренгейту
2. Рассчитайте и напишите температуру по Цельсию.
3. Стоп.

Шаг 1 довольно прост, но шаг 2 нуждается в доработке, поэтому план второго уровня может выглядеть примерно так:

1. Введите температуру по Фаренгейту (*F*)
2. Рассчитайте температуру по Цельсию (*C*):
   2.1 Вычтите 32 из *F* и умножьте на 5/9

3. Выведите значение С
4. Стоп.

Не существует жестких и быстрых правил о том, как писать блок-схемы и структурировать планы; вы должны использовать любой метод, который вы предпочитаете (или даже их смесь). Важным моментом является развитие умственной дисциплины, чтобы прояснить логику программы, прежде чем пытаться написать программу. Подход «сверху вниз» к блок-схемам или структурным планам означает, что общая структура программы четко продумана до того, как вам придется беспокоиться о деталях синтаксиса (кодирования), и это значительно снижает количество ошибок.

Эквивалентный структурный план решения квадратного уравнения, представленный на рис. 4.3, можно записать следующим образом:

1. Старт
2. Вводим данные [$a$, $b$, $c$]
3. Если $a = 0$ то
    Если $b = 0$
        Если $c = 0$ то
            Печатаем «Решение не определено»
        Иначе
            Печатаем «Нет решения»
    Иначе
        $x = -c/b$
        Печать x (только один корень)
    Иначе, если $b^2 < 4ac$ то
        Печатаем «Комплексные корни»
    Иначе, если $b^2 = 4ac$ то
        $x = -b/(2a)$
        Печать x (только один корень)
    Иначе
        $x_1 = \left(-b + \sqrt{b^2 - 4ac}\right)/(2a)$
        $x_2 = \left(-b - \sqrt{b^2 - 4ac}\right)/(2a)$
        Печать $x_1$, $x_2$
4. Стоп

## 4.3. Структурированное программирование с процедурами

Многие примеры, приведенные далее, будут довольно сложными. Более сложные программы, подобные этой, должны быть структурированы с помощью процедур (подпрограмм), которые подробно рассматриваются ниже. *Процедура* – это автономный раздел кода, который может определенным образом взаимодействовать с основной частью программы, и которая может быть вызвана основной программой. Тогда основная программа будет очень похожа на структурный план задачи первого уровня. Например, задача квадратного уравнения может быть спланирована на первом уровне следующим образом:

1. Прочитайте данные
2. Найдите и распечатайте решение(я)
3. Стоп.

С помощью процедуры (на самом деле подпрограммы в этом примере) это может быть переведено непосредственно в основную программу на Фортране:

```
READ*, A, B, C
CALL SOLVE_QUADRATIC( A, B, C )
END
```

***Резюме***

- Алгоритм – это систематический логический метод решения проблемы.
- Алгоритм должен быть разработан для проблемы, прежде чем ее можно будет закодировать.
- Блок-схема – это схематическое представление алгоритма.
- Структурный план представляет собой представление алгоритма в псевдокоде.
- Процедура (или подпрограмма) – это отдельный набор операторов Фортрана, предназначенных для выполнения конкретной задачи, которые можно активировать (вызвать) при необходимости.

***Упражнения***

Все задачи в этих упражнениях должны быть структурно спланированы или представлены блок-схемами до того, как они будут закодированы на Фортране.

4.1 Этот план конструкции определяет геометрическую конструкцию. Выполните план, зарисовав конструкцию:

1. Нарисуйте две перпендикулярные оси $x$ и $y$.
2. Нарисуйте точки $A(10, 0)$ и $B(0, 1)$
3. Пока $A$ не совпадает с началом координат повторяем:
   Нарисуйте прямую линию, соединяющую $A$ и $B$
   Переместите $A$ на одну единицу влево по оси $x$.
   Переместите $B$ на одну единицу вверх по оси $y$
4. Стоп.

4.2 Рассмотрим следующий план структуры, где $M$ и $N$ представляют целочисленные переменные Фортрана:

1. Установите $M = 44$ и $N = 28$.
2. Пока $M$ не равно $N$ повторяем:
   Пока $M > N$ повторяем:
   Замените $M$ на $M – N$
   Пока $N > M$ повторяем:
   Замените $N$ на $N – M$
3. Напечатать $M$
4. Стоп.

(a) Проработайте план конструкции, зарисовывая содержание $M$ и $N$ во время выполнения. Дайте вывод.
(b) Повторите (a) для $M = 14$ и $N = 24$.
(c) Какую общую арифметическую процедуру выполняет алгоритм (при необходимости попробуйте другие значения $M$ и $N$)?

4.3 Напишите программу для преобразования температуры по Фаренгейту в температуру по Цельсию. Проверьте это на данных в упражнении 3.2.

4.4 Строителю даны размеры пяти досок в футах (') и дюймах ('). Он хочет преобразовать длины в метры. Один фут равен 0,3048 метра, а один дюйм равен 0,0254 метра. Размеры досок: 4 фута 6 дюймов, 8 футов 9 дюймов, 9 футов 11 дюймов, 6 футов 3 дюйма и 12 футов 0 дюймов (т. е. первая доска имеет длину 4 фута 6 дюймов). Сохраните данные в файле. Напишите программу для отображения (под соответствующими заголовками) длины каждой доски в футах, дюймах и метрах, а также для нахождения и отображения общей длины доски в метрах. (Ответ: общая длина 12,624 метра)

4.5. Напишите программу, которая считывает любые два действительных числа (которые, как вы можете предположить, не равны), и записывает большее из них с подходящим сообщением.

4.6 Напишите программу для вычисления суммы ряда

$$1 + 1/2 + 1/3 + ... + 1/100$$

Программа должна записывать текущую сумму после каждых 10 слагаемых (т.е. сумма после 10 слагаемых, после 20 слагаемых, ..., после 100 слагаемых).

Подсказка: встроенная функция MOD($N$, 10) будет равна нулю только тогда, когда $N$ кратно 10. Используйте это в операторе IF, чтобы записать сумму после каждого 10-го члена. (Ответ: 5,18738 после 10 сроков)

# Глава 5 Построение возможных решений

Помимо способности очень быстро складывать числа, другим важным свойством компьютера является способность принимать решения, как мы вкратце видели в главе 3. компьютер обладает огромной способностью решать проблемы. Фундаментальной конструкцией принятия решений в Фортране является конструкция IF, другой формой которой является конструкция CASE.

## 5.1. Конструкция IF

Мы уже видели несколько примеров простого оператора IF и конструкции. Дополнительные примеры, которые становятся более сложными, приведены в этом разделе.

Решение для изгибающего момента в балке

Очень часто при проведении натурных исследований приходится принимать технические решения. Одним из вариантов такого решения является расчет балки. Легкая однородная балка $0 < x < L$ защемлена концами на одном уровне и несет сосредоточенную нагрузку $W$ в точке $x = a$. Изгибающий момент $M$ в любой точке $x$ вдоль балки определяется двумя разными формулами, зависящими от значения $x$ по отношению к $a$, а именно.

$$M = \begin{cases} W(L-a)^2\left[aL - x(L+2a)\right]/L^3 & \text{если } 0 \le x \le a \\ Wa^2\left[aL - 2L^2 + x(3L-2a)\right]/L^3 & \text{если } a < x \le L \end{cases} \qquad (5.1)$$

Следующий фрагмент программы вычисляет изгибающий момент через каждый метр вдоль 10-метровой балки с нагрузкой 100 ньютонов в точке на расстоянии 8 метров от конца $x = 0$:

```
INTEGER X
REAL A, L, M, W

L = 10
W = 100
A = 8

DO X = 0, L
 IF( X <= A )THEN
```

```
  M = W*(L - A) **2 * (A * L - X * (L + 2 * A)) / L ** 3
 ELSE
  M = W*A*A* (A*L - 2*L*L + X * (3 * L - 2 * A)) / L**3
 END IF
 PRINT*, X, M
END DO
```

Обратите внимание, что `X` является целым числом для использования в цикле `DO`.

Продемонстрировать использование конструкции IF можно и на следующем примере, в котором определяется лучший студент в группе.

*Лучший студент в группе*

Группа студентов пишет тест, и имя каждого из них (максимум 15 символов) и оценка заносятся в файл данных. Предположим, что отрицательных оценок нет. Мы хотим написать программу, которая выводит имя ученика с наивысшей оценкой вместе с его оценкой. Мы предполагаем, что существует только одна высшая оценка. Проблема того, что делать, когда двое или более студентов делят высшую оценку. Структурный план первого уровня для этой задачи может быть таким:

1. Старт
2. Найдите лучшего студента и высшую оценку
3. Распечатайте лучшего ученика и высшую оценку

Шаг 2 нуждается в доработке, поэтому более подробный план может быть таким:

1. Старт
2. Инициализировать `TopMark` (для запуска процесса)
3. Повторить для всех студентов
   Прочитать имя и отметку
   Если `Mark` > `TopMark`, то
   Замените `TopMark` на `Mark`
   Замените `TopName` на `Name`
4. Распечатайте `TopName` и `TopMark`
5. Стоп

Программа (для группы из 3 студентов):

```
 IMPLICIT NONE
 INTEGER    I       ! student counter
```

```
REAL     Mark    ! general mark
CHARACTER*15 Name    ! general name
REAL :: TopMark=0 ! top mark; can't be less than zero
CHARACTER*15 TopName   ! top student
OPEN( 1, FILE = 'MARKS' )
DO I = 1, 3
 READ( 1, * ) Name, Mark
 IF (Mark > TopMark) THEN
  TopMark = Mark
  TopName = Name
 END IF
END DO

PRINT*, 'Top student: ', TopName
PRINT*, 'Top mark:  ', TopMark
```

Поработайте с программой несколько раз вручную, чтобы убедиться, что она работает. Попробуйте это на следующих образцах данных (помните апострофы, потому что имена содержат запятые):

```
"Ivanov, VV" 40
"Khaimy, PA" 60
"Golovin, PS" 13
```

### Использование оператора **ELSE IF**

Вспомним программу `Final_Mark` в главе 3. Чтобы вывести определение (1, 2+, 2–, 3 или *F*) итоговой оценки каждого студента, у нас может возникнуть соблазн заменить сегмент

```
IF (Final >= 50) THEN
...
END IF
```

с набором простых операторов `IF` следующим образом:

```
IF (Final >= 75) PRINT*, Name, CRM, ExmAvg, Final, '1'
IF (Final >= 70 .AND. Final < 75)
  PRINT*, Name, CRM, ExmAvg, Final, '2+'
IF (Final >= 60 .AND. Final < 70)
  PRINT*, Name, CRM, ExmAvg, Final, '2-'
IF (Final >= 50 .AND. Final < 60)
  PRINT*, Name, CRM, ExmAvg, Final, '3'
IF (Final < 50) PRINT*, Name, CRM, ExmAvg, Final, 'F'
```

О логическом операторе `.AND.`, использованном в приведенном фрагменте, речь пойдет в следующих разделах этого учебного пособия. А сейчас о фрагменте. Хотя он и работает, но работает

неэффективно и может привести к потере драгоценного вычислительного времени. Есть пять отдельных операторов `IF`. Логические выражения во всех пяти (например, `Fin >= 75`) должны быть оценены для каждого учащегося, хотя мы знаем, что только одно из них может быть верным; студент не может получить аттестат первого класса и при этом потерпеть неудачу! Ниже приведен более эффективный способ кодирования проблемы. На всякий случай мы также посчитаем, сколько сдали на первый класс, сколько на второй и так далее. Целочисленные переменные `Firsts`, `UpSeconds`, `LowSeconds`, `Thirds` и `Fails` представляют количество учеников в каждом из этих соответствующих классов.

```
IF (Final >= 75) THEN
 PRINT*, Name, CRM, ExmAvg, Final, '1'
 Firsts = Firsts + 1
ELSE IF (Final >= 70) THEN
 PRINT*, Name, CRM, ExmAvg, Final, '2+'
 UpSeconds = UpSeconds + 1
ELSE IF (Final >= 60) THEN
 PRINT*, Name, CRM, ExmAvg, Final, '2-'
 LowSeconds = LowSeconds + 1
ELSE IF (Final >= 50) THEN
 PRINT*, Name, CRM, ExmAvg, Final, '3'
 Thirds = Thirds + 1
ELSE
 PRINT*, Name, CRM, ExmAvg, Final, 'F'
 Fails = Fails + 1
END IF
```

Это экономит время, поскольку Фортран прекращает проверку, как только находит истинное логическое выражение. Поэтому, если `Final` >= 75 верно, дальнейшая проверка не будет выполняться. Поэтому на вас лежит ответственность за правильное кодирование конструкции, чтобы только одно из логических выражений было истинным. Обратите также внимание на то, как отступы облегчают понимание структуры.

### Конструкция **IF** в целом

Более общая форма конструкции IF выглядит следующим образом:

```
IF (logical-expr1) THEN
            block1
      ELSE IF (logical-expr2) THEN
```

```
            block2
      ELSE IF (logical-expr3) THEN
            block3
      ...
      ELSE
            blockE
END IF
```

Если логическое выражение `logical-expr1` истинно, операторы в блоке `block1` выполняются, и управление переходит к следующему оператору после `END IF`. Если логическое-выражение `logical-expr1` ложно, оценивается логическое-выражение `logical-expr2`. Если оно - правда, выполняются операторы в блоке `block2`, за которыми следует следующий оператор после `END IF`. Если ни одно из логических выражений не является истинным, выполняются операторы блока `blockE`. Ясно, что логические выражения должны быть расположены так, чтобы только одно из них могло быть истинным в каждый момент времени. Может быть любое количество `ELSE IF` (или вообще ни одного), но может быть не более одного `ELSE`. Конструкция `IF` может быть дополнительно названа для помощи читателю (обычно для пояснения сложного вложения), например.

```
[GRADE:] IF (Final >= 50) THEN
      PRINT*, 'Pass'
     ELSE [GRADE]
      PRINT*, 'Fail'
     END IF [GRADE]
```

Блок `ELSE` или `ELSE IF` может быть назван только в том случае, если соответствующие блоки `IF` и `END IF` названы, и им должно быть присвоено одно и то же имя. Имя должно быть действительным и уникальным именем Фортрана.

Обратите внимание, что после ключевого слова `THEN` в первой строке конструкции ничего не может следовать.

### Вложенные **IF**

Когда конструкции IF являются вложенными, расположение END IF имеет решающее значение, так как это определяет, к каким IF принадлежат ELSE IF. Оператор ELSE IF или ELSE принадлежит последней открытой IF, которая еще не была закрыта. Для иллюстрации рассмотрим еще раз программирование решения

вездесущего квадратного уравнения, $ax^2 + bx + c = 0$. Необходимо проверить условие $a = 0$, чтобы предотвратить деление на ноль:

```
Disc = B * B - 4 * A * C
Outer: IF (A /= 0) THEN
     Inner: IF (Disc < 0) THEN
         PRINT*, 'Complex roots'
        ELSE Inner
         X1 = (-B + SQRT( Disc )) / (2 * A)
         X2 = (-B - SQRT( Disc )) / (2 * A)
        END IF Inner
    END IF Outer
```

Что произойдет, если `END IF Inner` переместится на 3 строки вверх, как показано ниже?

```
Outer: IF (A /= 0) THEN
     Inner: IF (Disc < 0) THEN
         PRINT*, 'Complex roots'
        END IF Inner  ! Wrong place now!
        ELSE Inner
         X1 = (-B + SQRT( Disc )) / (2 * A)
         X2 = (-B - SQRT( Disc )) / (2 * A)
    END IF Outer
```

Что ж, компилятор будет возражать из-за конфликта имен: `ELSE Inner` не может появиться после того, как `END IF Inner` закроет `Inner IF`. Однако, если все имена опущены, сегмент будет скомпилирован, но сделает деление на ноль определенным, если $a = 0$, поскольку `ELSE` теперь будет принадлежать первому `IF`. Необходимо самостоятельно попробовать этот вариант.

Вложенность может распространяться на любую глубину; в таких случаях следует осторожно использовать отступы и/или, чтобы сделать логику более ясной.

### Взаимодействие операторов **DO** и **IF**

Цикл DO может содержать конструкцию IF и наоборот. Основное правило состоит в том, что если конструкция начинается внутри другой конструкции, она также должна заканчиваться внутри этой конструкции. Таким образом, незаконно следующее:

```
DO I = 1, 10
 IF (I > 5) THEN
  ...
 END DO   ! Недопустимо: IF должен заканчиваться до DO
END IF
```

```
and so is this:
IF ( ... ) THEN
 DO I = 1, 10
  ...
 END IF   ! Недопустимо: DO должен заканчиваться до IF
END DO
```

## 5.2 Логический тип

До сих пор обсуждались четыре из пяти встроенных типов данных: целочисленный, действительный, символьный и комплексный. Пришло время обсудить четвертый тип: логический.

*Логические константы*

Семейство логического типа по умолчанию имеет две литеральные константы: .TRUE. и .FALSE. (верхний или нижний регистр). Значение параметра этого семейства по умолчанию возвращается обычным способом, функцией `KIND( .TRUE. )`

В компиляторе могут быть логические типы, отличные от стандартных; их можно использовать, например, для более компактного хранения логических массивов.

*Логические выражения*

Мы ранее кратко рассмотрели логические выражения. Они могут быть сформированы двумя способами: из числовых выражений в сочетании с шестью операторами отношения или из других логических выражений в сочетании с логическими переменными и пятью логическими операторами. Операторы отношения и их значения, с некоторыми примерами, представлены в таблице 5.1:

Таблица 5.1. Операторы отношений

| **Оператор отношения** | **Значение** | **Пример** |
|---|---|---|
| .LT. или < | Меньше чем | A < 1e–5 |
| .LE. или <= | Меньше или равно | B**2 .LE. 4*A*C |
| .EQ. или == | Равно | B**2 == 4*A*C |
| .NE. или /= | Не равно | A /= 0 |
| .GT. или > | Больше чем | B**2 – 4*A*C > 0 |
| .GE. или >= | Больше или равно | X >= 0 |

*Логические операторы*

Fortran 90 имеет пять логических операторов, которые работают с логическими выражениями. Таблица 5.2 позволяет понять смысл и значение логических операторов:

Таблица 5.2 Логические операторы

| **Логический оператор** | **Приоритет** | **Значение** |
|---|---|---|
| .NOT. | 1 | логическое отрицание |
| .AND. | 2 | логическое пересечение |
| .OR. | 3 | логическое объединение |
| .EQV. и .NEQV. | 4 | логическая эквивалентность и неэквивалентность |

Следующая «таблица истинности» (табл. 5.3) показывает влияние этих операторов на логические выражения *lex*1 и *lex*2 (T = true; F = false):

Таблица 5.3 Истинность отношений логических выражений

| **lex1** | **lex2** | **.NOT. lex1** | **lex1 .AND. lex2** | **lex1 .OR. lex2** | **lex1 .EQV. lex2** | **lex1 .NEQV. lex2** |
|---|---|---|---|---|---|---|
| T | T | F | T | T | T | F |
| T | F | F | F | T | F | T |
| F | T | T | F | T | F | T |
| F | F | T | F | F | T | F |

Порядок приоритета, показанный выше, может быть заменен круглыми скобками, которые всегда имеют наивысший приоритет.

*Примеры:*

(B * B == 4 * A * C) .AND. (A /= 0)

(Final >= 60) .AND. (Final < 70)
(A /= 0) .or. (B /= 0) .or. (C /= 0)
.not. ((A /= 0) .and. (B == 0) .and. (C == 0))

Между прочим, последние два выражения эквивалентны и ложны только тогда, когда A = B = C = 0 . Это заставляет задуматься, не так ли?

*Логические переменные*

Переменная может быть объявлена с логическим типом в операторе `LOGICAL`. Логические константы или выражения могут быть присвоены логическим переменным:

```
LOGICAL L1, L2, L3, L4, L5
REAL A, B, C
...
L1 = .TRUE.
L2 = B * B - 4 * A * C >= 0
L3 = A == 0
L4 = L1 .and. .not. L2 .or. L3
L5 = (L1 .and. (.not. L2)) .or. L3
```

Правила приоритета делают `L4` и `L5` логически эквивалентными. Значения истинности логических переменных представлены *T* и *F* в списковом вводе-выводе.

*Моделирование схемы переключения*

В следующем программном сегменте логические переменные `S1` и `S2` представляют состояние двух переключателей (ВКЛ. = *true*; ВЫКЛ. = *false*), а *L* представляет состояние лампы. Программа моделирует схемы на рис. 5.1, где переключатели расположены последовательно или параллельно.

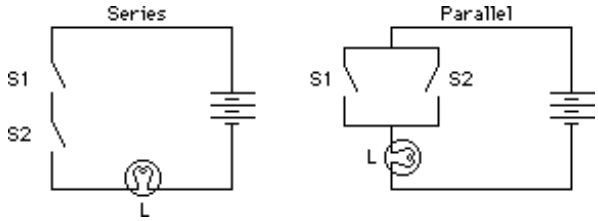


Рисунок 5.1 Цепи переключения

```
LOGICAL L, S1, S2
READ*, S1, S2
L = S1 .and. S2    ! series
!L = S1 .or. S2    ! parallel
PRINT*, L
```

Когда выключатели соединены последовательно, свет будет гореть, только если оба ключа включены. Эта ситуация представлена `S1.and.S2`. Когда переключатели включены параллельно, свет будет гореть, если один или оба переключателя включены. Это представлено `S1.or.S2`.

*Функции управления битами*

Некоторые языки программирования, такие как Pascal и C, имеют операторы, называемые побитовыми операторами, которые работают непосредственно с битами своих операндов. Они обычно обсуждаются в контексте логических (или Boolean) переменных. В Fortran 90 их аналогами являются встроенные функции манипулирования битами, которые работают с битами своих целочисленных аргументов.

## 5.3 Конструкция CASE

Конструкция CASE аналогична конструкции IF. Это позволяет выбирать между несколькими ситуациями или случаями на основе селектора. В таких случаях это удобнее, чем IF. Рассмотрим следующий фрагмент программы:

```
CHARACTER CH

DO
 READ*, CH
 PRINT*, ICHAR( CH )
 IF (CH == '@') EXIT

 IF(CH >= 'A'.and.CH <='Z'.or.CH >='a'.and.CH<='z') THEN
 SELECT CASE (CH)
   CASE ('A','E','I','O','U','a','e','i','o','u')
    PRINT*, 'Vowel'
   CASE DEFAULT
    PRINT*, 'Consonant'
  END SELECT
 ELSE
  PRINT*, 'Something else'
 END IF

END DO
```

Он решает, является ли символ гласным, согласным или чем-то еще. Он останавливается, когда прочитан символ @. Это можно было бы полностью запрограммировать с помощью IF, но это привело бы к созданию гораздо большего количества кода, который было бы труднее читать (попробуйте).

Общая форма CASE

```
SELECT CASE (expr)
             CASE (selector1)
                          block1
             CASE (selector2)
                          block2
             [CASE DEFAULT
                          blockD]
END SELECT
```

где выражение должно быть целым, символьным или логическим. Если он соответствует определенному селектору, выполняется этот блок, в противном случае выбирается `CASE DEFAULT`. Оператор `CASE DEFAULT` необязателен, но может быть только один. Это не обязательно должно быть последним предложением конструкции CASE.

Общая форма селектора представляет собой список непересекающихся значений и диапазонов того же типа, что и `expr`, заключенный в круглые скобки, например,

```
CASE( 'a':'h', 'i':'n', 'o':'z', '_')
```

Обратите внимание, что двоеточие может использоваться для указания диапазона значений. Если верхняя граница диапазона отсутствует, `CASE` выбирается, если expr дает значение, большее или равное нижней границе, и наоборот. Части конструкции `CASE` могут быть названы так же, как и конструкция `IF`.

Выбор оценок в измененной программе `Final_Mark` раздела 5.1 также можно запрограммировать с помощью `CASE`, если оценка `Final` преобразована в целочисленный тип:

```
SELECT CASE ( INT(Final) )
 CASE (75:)
  PRINT*, Name, CRM, ExmAvg, Final, '1'
  Firsts = Firsts + 1
 CASE (70:74)
  PRINT*, Name, CRM, ExmAvg, Final, '2+'
  UpSeconds = UpSeconds + 1
 CASE (60:69)
```

```
    PRINT*, Name, CRM, ExmAvg, Final, '2-'
    LowSeconds = LowSeconds + 1
   CASE (50:59)
    PRINT*, Name, CRM, ExmAvg, Final, '3'
    Thirds = Thirds + 1
   CASE DEFAULT
    PRINT*, Name, CRM, ExmAvg, Final, 'F'
    Fails = Fails + 1
  END SELECT
```

Бывают случаи, когда CASE более эффективен, чем IF, так как нужно вычислить только одно выражение *expr*.

## 5.4 Оператор GO TO

Трудно переоценить ущерб, нанесенный таким языкам, как Fortran и Basic, неразборчивым и бездумным использованием оператора GOTO. Сторонники классически более структурированных языков, таких как Pascal и C, считают его четырехбуквенным словом программиста. GO TO является безусловным переходом и имеет вид

```
GO TO label
```

где *label* – это метка оператора: число в диапазоне от 1 до 99999, предшествующее оператору в той же строке. Управление безоговорочно передается оператору с меткой. Например,

```
GO TO 99
X = 67.8
99 Y = -1
```

Оператор `X = 67,8` никогда не выполняется, что может привести даже к крушению корабля, плохому прогнозу или разрушению гидротехнических сооружений.

Новички могут спросить, зачем вообще нужен оператор GO TO. Его использование восходит к старым недобрым временам, когда в более старых версиях Фортрана отсутствовала блочная конструкция IF, и приходилось обходиться простым оператором IF. Рассмотрим следующий фрагмент кода (`L1` и `L2` – две определенные логические переменные):

```
IF (L1) THEN
 I = 1
 J = 2
ELSE IF (L2) THEN
 I = 2
```

```
 J = 3
ELSE
 I = 3
 J = 4
END IF
```

В отсутствие конструкции IF это должно быть закодировано как следующий клубок «спагетти»:

```
   IF (.NOT.L1) GOTO 10
    I = 1
    J = 2
    GOTO 30
10 IF (.NOT.L2) GOTO 20
    I = 2
    J = 3
    GOTO 30
20 I = 3
    J = 4
30 CONTINUE   ! Фиктивный оператор - ничего не делает
```

Нужно ли говорить больше? за исключением того, что никогда не следует использовать GOTO — его нет ни в одном примере этой книги. Он упоминается здесь исключительно по историческим и педагогическим причинам.

***Резюме***

- Конструкция `IF` допускает условное выполнение блоков операторов.
- Оператор `IF` допускает условное выполнение одного оператора.
- Конструкция `IF` может иметь любое количество предложений `ELSE IF`, но не более одного ключевого слова `ELSE`.
- Конструкции `IF` могут быть вложенными.
- Логические константы, переменные и выражения могут иметь только одно из двух значений: `.TRUE.` или `.FALSE.`
- Логические выражения могут быть сформированы из числовых выражений с операторами отношения <, <= и т.д.
- Логические операторы (`.NOT.`, `.AND.` и т.д.) могут использоваться для формирования более сложных логических выражений из других логических выражений и переменных.
- Фортран имеет функции обработки битов, которые работают непосредственно с битами, представляющими целые числа.

- Конструкция `CASE` может использоваться для выбора конкретного действия.
- Оператор `GOTO` выполняет безусловное ветвление, и его следует избегать любой ценой.

***Упражнения***

5.1 Напишите программу, которая считывает два числа (которые могут быть равными) и записывает большее из них с подходящим сообщением, или, если они равны, выводит соответствующее сообщение.

5.2 Напишите структурный план и программу для следующей задачи: прочтите 10 целых чисел и запишите, сколько из них положительных, отрицательных или нулевых. Напишите программу с конструкцией `IF`, а затем перепишите ее с конструкцией `CASE`.

5.3 Напишите программу для общего решения квадратного уравнения $ax^2 + bx + c = 0$. Используйте план структуры, разработанный в главе 4. Ваша программа должна иметь возможность обрабатывать все возможные значения данных $a$, $b$ и $c$. Попробуйте это на следующих значениях $a$, $b$ и $c$:

а) 1, 1, 1 (комплексные корни);
б) 2, 4, 2 (равные корни из –1,0);
в) 2, 2, –12 (корни из 2,0 и –3,0).

Перепишите свою программу со сложными типами, чтобы она могла работать со сложными корнями, а также со всеми другими особыми случаями.

# Глава 6 Циклы

В главе 3 мы представили мощную конструкцию DO для многократного выполнения блока операторов. Ситуацию, когда количество повторений можно определить заранее, иногда называют *детерминированным* повторением. Однако часто бывает так, что условие завершения повторяющейся структуры (или цикла) выполняется только во время цикла выполнения самого цикла. Этот тип повторяющейся структуры называется *недетерминированным*. Обе эти (логически совершенно разные) ситуации программируются с помощью конструкции DO в Fortran 90.

## 6.1 Детерминированное повторение

В этом разделе мы увидим, как обобщить конструкцию DO, предварительно рассмотрев еще несколько примеров.

*Расчет факториала*!

Переменная в цикле `DO` может использоваться в любом выражении внутри цикла, хотя ее значение не может быть изменено явно, например, с помощью оператора присваивания как быстрого способа досрочного завершения цикла. Следующая программа выводит список из $n$ и $n!$, где $n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$.

Выводу действительных или целочисленных результатов вы доверяете больше и почему?

```
INTEGER ::  NFACT = 1
INTEGER N
REAL    ::  XFACT = 1

DO N = 1, 20
  NFACT = NFACT * N
  XFACT = XFACT * N
  PRINT*, N, NFACT, XFACT
END DO
```

*Расчет биномиального коэффициента*

Биномиальный коэффициент широко используется в статистике. Количество способов выбрать $r$ объектов из $n$ без учета порядка равно

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{n(n-1)(n-2)\cdots(n-r+1)}{r!} \qquad (6.1)$$

Например, $\binom{10}{3} = \frac{10!}{3!\times 7!} = \frac{10\times 9\times 8}{1\times 2\times 3}$.

Если используется форма, включающая факториалы, числа могут стать очень большими, вызывая проблему зацикливания, показанную в предыдущем примере. Но использование самого правого выражения выше намного эффективнее:

```
INTEGER :: BIN = 1
INTEGER K, N, R

PRINT*, 'Give values for N and R'
READ*, N, R

DO K = 1, R
  BIN = BIN * (N - K + 1) / K
END DO

PRINT*, N, 'c', R, '=', BIN
```

*Вычисление предела числовой последовательности*

Циклы DO идеально подходят для вычисления последовательных членов числового ряда *a*. Этот пример также подчеркивает проблему, которая иногда возникает при вычислении предела. Рассмотрим последовательность

$$x_n = a^n / n!, \quad n = 1,2,3,\ldots \qquad (6.2)$$

где *a* – любая константа, а n! факториальная функция, определенная выше. Вопрос в том, каков предел этой последовательности, когда n становится неопределенно большим? Возьмем случай *a* = 10. Если мы попытаемся вычислить напрямую, у нас могут возникнуть проблемы, потому что *n*! увеличивается очень быстро по мере увеличения *n*, что может привести к зацикливанию или переполнению. Однако ситуация резко изменится, если мы заметим, что $x_n$ связан с $x_{n-1}$ следующим образом:

$$x_n = a x_{n-1} / n. \qquad (6.3)$$

Теперь проблем с цифрами нет. Следующая программа вычисляет для *a* = 10 и возрастающих значений *n* и выводит результат для каждого десятого значения *n*:

```
REAL :: X = 1
REAL :: A = 10
INTEGER N

DO N = 1, 100
  X = A * X / N
  IF (MOD( N, 10 ) == 0) PRINT*, N, X
END DO
```

*Комплексная передаточная функция*

Отклик (выход) линейной системы, которую можно рассматривать как «черный ящик», характеризуется своей передаточной функцией. Входной сигнал с заданной угловой частотой ($\omega$ рад./с) подается на одну сторону «ящика». Выход с другой стороны определяется входом, умноженным на абсолютное значение передаточной функции, с его фазой, сдвинутой на фазовый угол передаточной функции.

Предположим, что некоторое устройство характеризуется передаточной функцией

$$T(i\omega) = \frac{K(1+0.4i\omega)(1+0.2i\omega)}{i\omega(1+2.5i\omega)(1+1.43i\omega)(1+0.02i\omega)^2} \qquad (6.4)$$

где $i$ – единичное мнимое число $\sqrt{-1}$, $K$ – коэффициент усиления, $T(i\omega)$ – комплексное число. Если его действительная и мнимая части (возвращаемые `REAL` и `AIMAG`) равны $a$ и $b$ соответственно, тогда его абсолютное значение равно $\sqrt{a^2+b^2}$, а его фазовый угол $f$ определяется как arctg($b/a$). Если используется встроенная функция `ATAN2`, возвращаемый угол будет находиться в диапазоне от – π до π, так что будет задан правильный квадрант (это не относится к `ATAN`).

В приведенной ниже программе показано, как система реагирует на различные входные частоты ω. Эта информация необходима при проектировании устойчивых систем управления с обратной связью. Начальная входная частота составляет 0,02 рад/сек. Это умножается на коэффициент (`Fact`) 1,25 каждый раз для заданного количества шагов. Коэффициент усиления $K$ равен 900. Фазовый сдвиг $f$ выходного сигнала указан в градусах. Обратите внимание, что именованная комплексная константа ($i$) используется для $\sqrt{-1}$. Комплексная переменная *iom* образована из $i$ и `Omega` исключительно для удобства записи.

```
IMPLICIT NONE

INTEGER                N              ! counter
INTEGER             :: Steps          ! iteration count
REAL                   A, B           ! Re(T), Im(T)
REAL                :: Fact  = 1.25   ! scaling factor
REAL                :: K     = 900    ! amplification
REAL                :: Omega = 0.02   ! angular frequency
REAL                   Phase          ! phase angle
REAL, PARAMETER     :: Pi   = 3.1415927
COMPLEX, PARAMETER :: i = (0, 1)     ! sqrt(-1)
COMPLEX                iom            ! sqrt(-1) * omega
COMPLEX                T      ! complex transfer function

READ*, Steps
PRINT 20
20 FORMAT('Omega',T12,'Real T',T27,'Im T',T42,'Abs T', &
                   T62, 'Phase' )
PRINT*

DO N = 0, Steps
  iom = i * Omega
  T = (K*(1 + 0.4*iom)*(1 + 0.2*iom)) /              &
   (iom*(1 + 2.5*iom)*(1 + 1.43*iom)*(1 + 0.02*iom)**2)
  A = REAL( T )
  B = AIMAG( T )
  Phase = ATAN2( B, A ) * 180 / Pi       ! phase degrees
  PRINT 10, Omega, A, B, ABS( T ), Phase
10 FORMAT(F7.3,T10,E11.4,T25,E11.4,T40,E11.4,T60, F7.2)
  Omega = Omega * Fact
END DO
```

*Пример вывода:*

| Omega | Real T | Im T | Abs T | Phase |
|---|---|---|---|---|
| 0.020 | –0.3024E+04 | –0.4483E+05 | 0.4493E+05 | –93.86 |
| 0.025 | –0.3018E+04 | –0.3578E+05 | 0.3591E+05 | –94.82 |
| 120.371 | –0.1651E-01 | 0.1830E-01 | 0.2465E–01 | 132.07 |
| 150.463 | –0.7512E-02 | 0.1100E-01 | 0.1332E–01 | 124.33 |

Если вы запустите программу, вы увидите, как входной сигнал сначала усиливается, а затем ослабевает. Фазовый сдвиг начинается

примерно с –90° и постепенно перемещается примерно до –180°, после чего он осциллирует на оси по мере увеличения входной частоты.

## 6.2 Конструкция DO в целом

Попробуйте воспроизвести следующие фрагменты программы (результаты отображаются после каждого цикла):

```
DO I = 2, 7, 2
  WRITE( *, '(I3)', ADVANCE = 'NO' ) I
END DO
```

Вывод: 2 4 6

```
DO I = 5, 4
  WRITE( *, '(I3)', ADVANCE = 'NO' ) I
END DO
```

Вывод: ничего

```
DO I = 5, 1, -1
  WRITE( *, '(I3)', ADVANCE = 'NO' ) I
END DO
```

Вывод: 5 4 3 2 1

```
DO I = 6, 1, -2
  WRITE( *, '(I3)', ADVANCE = 'NO' ) I
END DO
```

Вывод: 6 4 2

Общая форма конструкции DO, которую мы будем использовать, такова:

```
[name:] DO variable = expr1, expr2 [, expr3]
           Block
        END DO [name]
```

где `variable` (переменная DO) – целочисленная переменная, `expr1`, `expr2` и `expr3` – любые допустимые целочисленные выражения, а `name` – необязательное имя конструкции. `expr3` является необязательным, его значение по умолчанию равно 1. Выражения `expr1`, `expr2` и `expr3` называются *параметрами оператора* DO. *Переменная* DO инициализируется параметром `expr1` до того, как будет принято решение о том, зацикливаться или нет, в соответствии с приведенной ниже формулой. По завершении каждого цикла парметр

*expr*3 добавляется к переменной DO, опять же перед принятием решения о цикле.

Отсюда следует, что после завершения конструкции DO *переменная* DO не будет иметь того значения, которое она имела при последнем выполнении блока. Например, в первом фрагменте выше конечное значение *I* равно 8. Количество итераций конструкции DO определяется по формуле MAX((*expr*2–*expr*1+*expr*3)/*expr*3,0), где MAX – встроенная функция, возвращающая максимум своих аргументов.

Поскольку MAX возвращает значение того же типа, что и его аргументы, возвращаемое значение в этом случае будет значением выражения (при необходимости усеченным) или нулем, в зависимости от того, что больше. Эта формула называется счетчиком итераций или счетчиком циклов DO. Вы должны убедиться, что счетчики итераций для четырех сегментов выше равны 3, 0, 5 и 3 соответственно.

Обратите внимание, что блок DO может вообще не выполняться. Это называется циклом с нулевым срабатыванием и возникает всякий раз, когда первый аргумент функции MAX в формуле оценивается как неположительная величина.

```
DO I = J, K, L
```

- Если *L* положителен, блок выполняется с *I*, начиная с *J* и увеличиваясь на *L* каждый раз, пока он не будет выполнен для наибольшего значения *I*, не превышающего *K*.
- Если $L > 0$ и $J > K$, блок вообще не выполняется (нулевой счетчик срабатываний).
- Если *L* отрицательное, блок выполняется с *I*, начинающимся с *J* и уменьшающимся на $|L|$ каждый раз, пока не будет выполнено для наименьшего значения *I* не меньше *K*.
- Если $L < 0$ и $J < K$, блок вообще не выполняется (нулевой счетчик срабатываний).

Формула для подсчета итераций вычисляется перед первым выполнением блока. Даже если значения параметров DO впоследствии будут изменены внутри блока, это не повлияет на количество итераций. *Переменная* DO и *параметры* могут быть реальными. Эта функция, однако, была объявлена устаревшей (т.е. может быть полностью удалена из следующего стандарта), поэтому следует очень постараться не использовать ее. Это приводит к всевозможным неприятным ошибкам округления.

## 6.3 Оператор DO с нецелочисленными приращениями

В научных и инженерных вычислениях бывает много ситуаций, когда нужно делать нецелочисленные приращения в цикле. Рассмотрим снова камень, брошенный вертикально вверх. Предположим, что он запущен в момент времени $t = 0$ секунд, и мы хотим вычислить его положение $s(t)$ между моментами времени $t = t_0$ и $t+t_1$ каждые $dt$ секунд. Эти времена вряд ли будут целыми числами. Один из способов решить эту проблему — рассчитать собственное количество итераций для этой проблемы. Количество задействованных интервалов равно $(t_1 - t_0)/dt$. Поскольку нам нужен результат на каждом конце интервала между $t_0$ и $t_1$, нам нужно добавить к этому 1. Таким образом, очевидное значение для нашего счетчика итераций равно $(t_1 - t_0)/dt+1$.

Теперь, поскольку это должно быть целое число, мы усекаем его с помощью функции `INT` или округляем с помощью функции `NINT`? Чтобы ответить на этот вопрос, мы должны сначала решить, что должно произойти, если $t_1–t_0$ не является точным кратным $dt$. Допустим, мы не хотим, чтобы расчеты производились за пределами указанного нами диапазона. Это исключает функция `NINT`. Предположим, что $t_0 = 0$, $t_1 = 5$ и $dt = 0{,}4$. Наше количество итераций равно 12,5. Округление с помощью функции `NINT` даст 13 с нежелательным вычислением после $t_1$. Поэтому необходимо использовать функции `INT`. Однако и в этом есть свои проблемы. Из-за ошибки округления количество интервалов $(t_1 - t_0)/dt$ может легко упасть до целого числа (например, 10/0,1 получается как 99,9999 вместо 100), поэтому усечение приведет к потере одного цикла. Fortran 90 предлагает изящное решение этой проблемы.

Новая встроенная функция `SPACING(`*X*`)` возвращает абсолютный интервал между значениями рядом с *X*. Таким образом, наиболее удовлетворительным ответом является добавление `SPACING(`*dt*`)` к счетчику итераций перед усечением.

Следующая программа считывает значения для $t_0$ (`TSStart`), $t_1$ (`TEnd`) и $dt$ и печатает количество итераций (`TRIPS`) перед вычислением и печатью положения камня каждые $dt$ секунд. Обратите внимание, что теперь $t$ должно быть явно обновлено в блоке DO, так как *I* используется исключительно как счетчик.

```
REAL, PARAMETER :: G = 9.8
REAL               dT, S, T, TStart, TEnd
REAL            :: U = 60
```

```
INTEGER            I, TRIPS

READ*, TStart, TEnd, dT
TRIPS = INT( (TEnd - TStart) / dT + SPACING(dT) ) +
1
PRINT*, TRIPS
T = TStart

DO I = 1, TRIPS
  S = U * T - G / 2 * T * T
  PRINT*, I, T, S
  T = T + dT
END DO
```

Возникает еще одна интересная проблема. Предположим, что мы по-прежнему хотим вычислять $s(t)$ каждые $dt$ секунд, но хотим выводить его только каждые $h$ секунд — это обычная проблема численного анализа, где длина шага $dt$ может быть очень маленькой. Учитывая, что нам нужен вывод на первой итерации, мы должны пропустить следующие итерации $h/dt$ перед повторной печатью. Поскольку переменная $I$ оператора DO начинается с 1, это означает, что мы хотим выводить каждый раз, когда $(I–1)$ является точным кратным $h/dt$. Этого можно добиться, заменив PRINT выше на

```
IF(MOD(I-1,INT(H / dT + SPACING(dT))) == 0) PRINT*,I,T,S
```

где такая же поправка была сделана на ошибку округления, и необходимо ввести значение $h$.

Обратите внимание, что цикл в этой задаче действительно детерминирован – мы могли бы заранее определить количество итераций. Хотя это рекомендуемый способ обработки нецелочисленных приращений, существует другое решение, которое будет упомянуто позже, когда мы будем рассматривать недетерминированные циклы.

## 6.4 Недетерминированные циклы

Все детерминированные циклы основаны на том факте, что вы можете точно определить количество итераций до начала цикла. Но в следующем примере нет принципиального способа вычислить количество итераций, поэтому необходима другая форма конструкции DO.

*Игра в угадайку*

Проблема легко формулируется. Программа «думает» о целом числе от 1 до 10 (то есть генерирует его случайным образом). Вы должны угадать это. Если ваша догадка слишком высока или слишком мала, программа должна сообщить об этом. Если ваша догадка верна, должно появиться сообщение с поздравлением. Здесь требуется немного больше размышлений, поэтому план структуры может быть полезен:

1. Генерировать случайное целое число
2. Спросите пользователя (предположительно, мужчину) для предположения
3. Повторяйте до тех пор, пока предположение не станет правильным:
   Если предположение слишком мало, то
   Скажи ему, что это слишком низко
   В противном случае
   Скажи ему, что это слишком высоко
   Спросите его о другом предположении
4. Вежливое поздравление
5. Стоп.

Прежде чем мы рассмотрим всю программу, давайте посмотрим, как генерируется случайное целое число. Оператор

```
CALL RANDOM_NUMBER( R )
```

сначала генерирует случайное действительное $R$ в диапазоне [0, 1). $10*R$ будет в диапазоне [0, 10), а $10*R + 1$ будет в диапазоне [1, 11), т.е. между 1,000000 и 10,999999 включительно. Использование `INT` для этого даст целое число в диапазоне от 1 до 10, как требуется.

Если вы хотите играть более одного раза, каждый раз с разными случайными числами, вам нужно будет «переустанавливать» генератор случайных чисел определенным образом каждый раз, когда вы запускаете программу. При первом запуске программы укажите в качестве начального числа любое целое число. Но в последующих случаях вы должны использовать новое число, напечатанное в конце предыдущей игры.

```
INTEGER                    FtnNum, MyGuess
INTEGER, DIMENSION(1) ::   Seed
REAL                       R

WRITE( *, '("Seed: ")', ADVANCE = 'NO' )
READ*, Seed(1)              ! user supplies seed
```

```
CALL RANDOM_SEED(PUT=Seed) !seeds the random number generator
CALL RANDOM_NUMBER( R )
FtnNum = INT( 10 * R + 1)
WRITE( *, '("Your guess: ")', ADVANCE = 'NO' )
READ*, MyGuess
DO
  IF (MyGuess == FtnNum) EXIT
    IF (MyGuess > FtnNum) THEN
      PRINT*, 'Too high.  Try again'
    ELSE
      PRINT*, 'Too low.  Try again'
    END IF
    WRITE( *, '("Your guess: ")', ADVANCE = 'NO' )
    READ*, MyGuess
END DO
PRINT*, 'BINGO!  Well done!'
CALL RANDOM_SEED(GET=Seed) !get the new seed for another game
PRINT*
PRINT*, 'New seed: ', Seed(1)
```

Попробуйте несколько раз. Обратите внимание, что цикл DO (который теперь не имеет переменных или параметров) повторяется до тех пор, пока `MyGuess` не равен `FtnNum`. В принципе невозможно узнать, сколько циклов потребуется, прежде чем они сравняются, и поэтому здесь важна эта новая форма конструкции DO. В этом случае цикл завершается, когда выполняется оператор `EXIT`. Проблема действительно недетерминирована.

Поразмыслив, вы можете подумать, что кодирование немного расточительно. Фрагмент

```
  WRITE( *, '("Your guess: ")', ADVANCE = 'NO' )
  READ*, MyGuess
```

должен появиться дважды. Один раз, чтобы запустить цикл (иначе `MyGuess` будет неопределенным), и второй раз в самом цикле. Измените программу, как указано ниже, и попробуйте запустить ее (воспроизведен только участок с изменениями):

```
FtnNum = INT( 10 * R + 1)
! remove two lines
DO
    WRITE(*,'("Your guess: ")',ADVANCE = 'NO') ! move up
    READ*, MyGuess                             ! move up
    IF (MyGuess > FtnNum) THEN
      PRINT*, 'Too high.  Try again'
    ELSE IF (MyGuess < FtnNum) THEN    ! ELSE IF now
      PRINT*, 'Too low.  Try again'
```

```
    ELSE
      PRINT*, 'Well done!'          ! congrats here now
    END IF
  IF (MyGuess == FtnNum) EXIT              ! move down
END DO

! remove congrats
CALL RANDOM_SEED( GET=Seed) !get the new seed for another game
```

Эквивалентный план структуры для новой версии:

1. Генерировать случайное целое число
2. Повторите:
   Попросите пользователя предположить
   Если предположение слишком низкое
   Скажи ему, что это слишком низко
   В противном случае, если предположение слишком велико
   Скажи ему, что это слишком высоко
   В противном случае
   Вежливое поздравление
   Пока предположение верно
3. Стоп.

Существенное отличие состоит в том, что EXIT находится в начале блока DO в первой версии, но в конце блока во второй версии. Однако есть более тонкая разница: в первом случае условие выхода проверяется наверху; во втором случае он проверяется только в конце.

*Оператор* DO *и* EXIT *по условию*

Мы видели еще две версии конструкции оператора DO:

```
DO
     IF (logical-expr) EXIT
       block
END DO
```

и

```
DO
       block
     IF (logical-expr) EXIT
END DO
```

Оператор EXIT предоставляет средства для выхода из бесконечного цикла. На самом деле он может пойти куда угодно в цикле. Однако лучше всего записывать либо вначале цикла, либо в конце; тогда не нужно искать в цикле условие выхода. Некоторые

могут возразить, что EXIT всегда должен находиться наверху такого недетерминированного цикла, чтобы было ясно, чем закончится цикл, когда он впервые встретится с ним. Конструкция *while-do* таких языков, как Pascal, более легко поддается этому соглашению. Fortran 90, делает более естественным размещение EXIT в конце.

Есть одна ситуация, в которой EXIT должен находиться в верхней части цикла, и это когда нулевой счетчик поездок логически возможен. Примером может служить исходная форма игры в угадайку выше: если пользователь правильно угадывает число с первого раза, выполнение блока DO выполняться не должно.

### Конструкция **DO WHILE**

Конструкция DO может начинаться с оператора DO WHILE:

```
DO  WHILE (logical-expr)
        block
      END DO
```

Это логически эквивалентно

```
DO
      IF (.NOT.logical-expr) EXIT
      block
     END DO
```

Конструкция DO WHILE – очень привлекательная конструкция, поскольку условие повторения четко указано в начале цикла. Однако при определенных обстоятельствах это может повлечь за собой расплату за оптимизацию. Есть много примеров его использования позже.

### *Оператор* DO: *варианты, которые не рекомендуются*

Оператор EXIT также может использоваться в конструкции DO с переменной DO и параметрами:

```
DO I = 1, N
  ...
  IF (I == J) EXIT
  ...
END DO
```

Эта форма крайне не рекомендуется! Если у вас возникает соблазн попробовать это, чтобы выйти из сложной ситуации, это,

вероятно, означает, что вы недостаточно четко продумали логику. Вы должны быть в состоянии указать все возможные условия для выхода недвусмысленно либо в верхней, либо в нижней части цикла. Некоторые примеры возникновения такой ситуации приведены ниже.

Оператор

```
CYCLE [name]
```

передает управление оператору `END DO` соответствующей конструкции, поэтому, если еще предстоит выполнить дальнейшие итерации, инициируется следующая. Его использование не рекомендуется – это затрудняет просмотр логики. Конструкция DO может использовать метку оператора следующим образом:

```
DO 100 I = 1, N
...
100  CONTINUE
```

Еще раз напомним, `CONTINUE` – это фиктивный оператор, который ничего не делает. Конструкция также может заканчиваться помеченным `END DO`. Эта форма не рекомендуется — метки не нужны и затеняют логику избыточной информацией.

*Простые числа*

Многие люди одержимы простыми числами, и в большинстве книг по программированию должна быть программа для проверки того, является ли заданное число простым.

Число является простым, если оно не является точным кратным никакому другому числу, кроме самого себя и 1, т.е. если оно не имеет делителей, кроме самого себя и 1. Самый простой план атаки состоит в следующем. Предположим, что $P$ – число, которое необходимо проверить. Посмотрите, можно ли найти числа $N$, которые делятся на $P$ без остатка. Если их нет, то $P$ простое число. Какие числа $N$ мы должны попробовать?

Что ж, мы можем ускорить процесс, ограничив $P$ нечетными числами, поэтому нам нужно попробовать только нечетные делители $N$. Когда мы прекратим тестирование? Когда $N = P$? Нет, мы можем остановиться гораздо раньше. На самом деле, мы можем остановиться, как только $N$ достигнет $\sqrt{P}$, так как если есть множитель больше $\sqrt{P}$, то должен быть и соответствующий множитель меньше $\sqrt{P}$, что мы бы и нашли. И с чего мы начнем? Ну, так как $N = 1$ будет фактором любого $P$, мы должны начать с $N = 3$. План структуры следующий:

1. Читать *P*
2. Инициализировать *N* до 3
3. Найдите остаток *R* при делении *P* на *N*.
4. Повторяйте до тех пор, пока *R* = 0 или *N*:
   Увеличить *N* на 2
   Найдите *R*, когда *P* делится на *N*
5. Если то *P* является простым
   Еще *P* не является простым
6. Стоп.

Необходимо отметить, что условие выхода проверяется в начале цикла, поскольку в первый раз *R* может быть нулевым. Также обратите внимание, что есть два условия, при которых цикл остановится. Следовательно, после завершения цикла требуется `IF`, чтобы определить, какое условие остановило его. Рассмотрим программу:

```
PROGRAM Prime
! Tests if an odd integer > 3 is prime

IMPLICIT NONE
INTEGER ::  N = 3
INTEGER     P, Rem

PRINT*, 'Gimme an odd integer:'
READ*, P
Rem = MOD( P, N )

DO
  IF (Rem == 0 .OR. N >= SQRT( REAL(P) )) EXIT
    N = N + 2
    Rem = MOD( P, N )
END DO

IF (Rem /= 0) THEN
  PRINT*, P, ' is prime'
ELSE
  PRINT*, P, ' is not prime'
END IF

END
```

Попробуйте это на следующих числах: 4 058 879 (не простое число), 193 707 721 (простое число) и 2 147 483 647 (простое число). Очевидно, что эта программа не может проверить слишком большое число, поскольку оно больше, чем наибольшее целое число, которое

может быть представлено внутренним типом Фортрана. Способы проверки таких огромных чисел на их простоту описаны в книге Д.Е. Кнут [9]. Здесь было бы очень удобно использовать форму DO WHILE конструкции DO. Шаг 4 плана конструкции необходимо изменить на

4. Пока $R \neq 0$ и $N < \sqrt{P}$ повторяем:

и DO должен быть перефразирован как

```
DO WHILE (Rem /= 0 .AND. N < SQRT( REAL(P) ))
  N = N + 2
  Rem = MOD( P, N )
END DO
```

Обратите внимание, что условие является логическим отрицанием условия выхода, заданного изначально.

*Чтение неизвестного объема данных*

Следующая программа использует DO WHILE со специальной функцией общего оператора READ, чтобы прочитать неизвестное количество данных из файла DATA и найти их среднее значение:

```
REAL :: A, SUM
INTEGER :: N = 0
INTEGER :: IO = 0

OPEN( 1, FILE = 'DATA' )
SUM = 0

DO WHILE (IO == 0)
  READ (1, *, IOSTAT = IO) A
  IF (IO == 0) THEN
    SUM = SUM + A
    N = N + 1
    PRINT*, A
  END IF
END DO

PRINT*, "Mean:", SUM / N
```

Это самое грубое решение проблемы: данные должны подаваться по одному значению на строку в файле. Более элегантные решения будут даны позже. IOSTAT — это спецификатор, который устанавливается равным нулю, если операция READ завершается успешно, или отрицательному значению, если во время операции

READ возникает условие конца файла. Это обсуждается более подробно позже.

Конструкцию DO WHILE можно заменить на

```
DO
  IF ((ABS( Term ) <=  Err) .or. (K > MaxTerms))
EXIT
  ...
END DO
```

Обратите внимание, что логическое условие должно быть отменено — теперь это условие для выхода. DO WHILE или EXIT в начале цикла здесь уместны, поскольку начальный член может быть достаточно мал, и в этом случае $k$ все равно будет равно 1. Также обратите внимание, что есть два условия для завершения цикла. Следовательно, после DO требуется IF, чтобы установить, какое условие было выполнено. У вас может возникнуть соблазн использовать DO с параметрами и EXIT для выхода, если Term становится достаточно маленьким:

```
DO I = 1, MaxTerms
  IF (ABS( Term ) <=  Err) EXIT
  ...
END DO
```

Хотя это прекрасно работает, это определенно не рекомендуется (некоторые программисты точно не согласятся!). Причины следующие. Возражение состоит в том, что все условия для выхода не ясны в самом начале цикла — после беглого взгляда вы можете подумать, что это детерминированный цикл. Но, вы можете возразить, я делю волосы; ведь второе условие для выхода уже в следующей строке. Но через несколько месяцев вы можете ввести дополнительные условия позже в блоке. Беда в том, что эта невинно выглядящая структура допускает специальные поправки позже — на этом этапе программист может легко упустить из виду все условия для выхода.

Принцип таков: все условия для выхода должны быть четко указаны в одном месте — в начале или в конце цикла.

### *Резюме*

- Конструкция DO с параметрами должна использоваться для программирования детерминированного цикла, когда количество итераций (количество итераций) известно программе (т.е., в принципе, программисту) до того, как цикл

встретится. Эта ситуация характеризуется общим планом строения:

Повторить *N* раз:

Блок утверждений для повторения

- где *N* известно или вычислено до того, как цикл встретится в первый раз, и не изменяется блоком. Синтаксис DO в этом случае следующий:

```
[name:] DO  variable = first, last, step
                        block
END DO [name]
```

Все формы конструкции могут быть факультативно названы. Если *step* опущен, по умолчанию он равен 1.

Если *step* отрицательный, *variable* будет уменьшаться до тех пор, пока *first* больше или равно *last*.

- DO с EXIT может использоваться для программирования недетерминированного цикла, в котором количество итераций заранее неизвестно, т. е. всякий раз, когда в блоке DO изменяется значение истинности *condition* для выхода. Эта ситуация характеризуется следующими двумя структурными планами:

Повторяйте, пока *condition* не станет истинным:

Блок для повторения (сброс значения истинности *условия*).

или же

Повторение:

Блок для повторения (сброс значения истинности *условия*)

пока *условие* не станет истинным.

Следует учитывать, что *condition* — это условие выхода из цикла.

Синтаксис этих форм

```
DO
     IF  (condition) EXIT
     block
END DO
```

и

```
DO
     block
     IF(condition) EXIT
END DO
```

- Недетерминированный цикл также может быть запрограммирован с помощью конструкции DO WHILE. Вот общий план строения
  Пока *condition* верно повторяем:
  Блок утверждений для повторения.
  Обратите внимание, что *condition* теперь является условием выполнения еще одной итерации, а не выхода.
  Синтаксис этой структуры такой

```
DO WHILE (condition)
                                    block
                    END DO
```

  Эта конструкция может привести к наказаниям за оптимизацию.

- Переменная DO и параметры должны быть целыми числами.
- Переменная DO не должна явно изменяться в блоке DO.
- Количество итераций (которое может быть равно нулю) рассчитывается на основе начальных значений параметров.
- Конструкции DO могут быть вложены на любую глубину.
- Хороший стиль программирования требует, чтобы EXIT из DO происходил как можно ближе к началу или концу цикла.
- Спецификатор IOSTAT можно использовать с READ для обнаружения состояния конца файла.

***Упражнения***

6.1 Напишите программу для нахождения суммы последовательных четных целых чисел 2, 4, ..., 200. (Ответ: 10100)

6.2 Напишите программу, которая выводит таблицу sin *x* и cos *x* для углов *x* от 0° до 90° с шагом 15°.

6.3 Существует множество формул для вычисления числа π (отношение длины окружности к ее диаметру). Самый простой

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

что вытекает из разложения

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \ldots, \quad \text{где} \quad x = 1$$

(а) Напишите программу для вычисления числа π с использованием последовательности из упражнения 6.1. Используйте столько терминов в серии, сколько разумно позволяет ваш компьютер (начните скромно, скажем, со 100 членов, и каждый раз перезапускайте программу, добавляя все больше и больше членов). Вы должны обнаружить, что ряд сходится очень медленно, т.е. требуется много членов, чтобы приблизиться к числу π.

(б) Перестановка ряда ускоряет сходимость:

$$\frac{\pi}{8} = \frac{1}{1\times 3} + \frac{1}{5\times 7} + \frac{1}{9\times 11} + \ldots$$

Вместо этого напишите программу для вычисления числа π, используя этот ряд. Вы должны обнаружить, что вам нужно меньше терминов, чтобы достичь того же уровня точности, который вы получили в (а).

(в) Один из самых быстрых рядов для числа π является

$$\frac{\pi}{4} = 6\arctan\left(\frac{1}{8}\right) + 2\arctan\left(\frac{1}{57}\right) + \arctan\left(\frac{1}{239}\right)$$

Используйте эту формулу для вычисления числа π. Не используйте стандартную функцию ATAN для вычисления арктангенсов, так как это будет обманом. Скорее используйте последовательность из упражнения 6.2.

6.4 Следующий метод вычисления числа π принадлежит Архимеду:

1. Пусть $A = 1$ и $N = 6$.
2. Повторите 10 раз, указывая:
   Заменить $N$ на $2N$
   Заменить A на $\sqrt{2-\sqrt{4-A^2}}$
   Пусть $NA/2$
   Пусть $U = L\Big/\sqrt{1-A^2/2}$
   Пусть $P = (U+L)/2$ (оценка числа пи)
   Пусть $E = (U-L)/2$ (оценка ошибки)
   Печать $N$, $P$, $E$
3. Остановка вычисления

Напишите программу, реализующую этот алгоритм.

6.5 Волна периода $T$ может быть определена функцией

$$f(t)=\begin{cases}1, & если \quad 0<t<T\\ -1 & если \quad -T<t<0\end{cases}$$

Ряд Фурье для $f(t)$ имеет вид

$$\frac{4}{\pi}\sum_{k=0}^{\infty}\frac{1}{2k+1}\sin\left[\frac{(2k+1)\pi t}{T}\right]$$

Интересно узнать, сколько членов требуется для хорошего приближения к этой бесконечной сумме. Принимая $T$ = 1, напишите программу для вычисления и отображения суммы $n$ членов ряда для $t$ от 0 до 1 с шагом, скажем, 0,1. Запустите программу для разных значений $n$, например. 1, 3, 6 и т.д.

6.6 Вертикальное уменьшение скорости дрейфового течения может быть рассчитано с помощью теории Экмана [10]. Решение зависит от скорости ветра и географической широты места. Вектор скорости течения рассматривается как сумма проекций вектора ($u$, $v$) на взаимно ортогональные оси $x$ и $y$. Исходные уравнения записываются в следующем виде:

Проекции вектора течения

$$u=Ve^{-az}\cos(45°-az)$$
$$v=Ve^{-az}\sin(45°-az)$$

где

$$V=\frac{\tau}{\sqrt{2\rho_w\mu\omega\sin(\varphi)}},\quad a=\sqrt{\frac{\rho_w\omega\sin(\varphi)}{\mu}}$$

Написать программу и выполнить расчет проекций вектора течения $u$ и $v$ на глубинах $z$ с шагом $\Delta z = 0{,}1D$ при

$$D=\pi\sqrt{\frac{\mu}{\rho_w\omega\sin(\varphi)}}$$

для условий $\rho_w$ = 1.25×10$^3$ кг/м$^3$, $\mu$ = 0,001 мПа/с, $\varphi$ = 55° с.ш., $\tau$ = 2,6×10$^{-3}W_a$, $W_a$ = 10 м/с.

6.7 Если популяция растет по логистической модели, то ее размер $X(t)$ в момент времени $t$ определяется формулой

$$X(t) = \frac{KX_0}{(K - X_0)e^{-rt} + X_0}$$

где $X_0$ – начальный размер в момент времени $t = 0$, $r$ – скорость роста, $K$ – несущая способность среды. Напишите программу, которая будет вычислять и печатать значения $X(t)$ за период в 200 лет. Возьмем $X_0 = 2$, $r = 0{,}1$ и $K = 1000$. Поэкспериментируйте с различными значениями $K$ и посмотрите, сможете ли вы интерпретировать $K$ биологически.

# Глава 7 Возникающие ошибки

Программы редко запускаются правильно с первого раза, даже у опытных программистов. На компьютерном жаргоне ошибка в программе называется ошибкой. История состоит в том, что мотылек замкнул два термоэлектронных клапана в одном из первых компьютеров. На поиски этого первобытного (обугленного) «жука» ушло несколько дней. Процесс обнаружения и исправления таких ошибок называется отладкой. Существует четыре типа ошибок:

• ошибки компиляции
• ошибки во время выполнения
• логические ошибки
• ошибка округления.

В этой главе мы рассмотрим виды ошибок, которые могут возникнуть при программировании, которое мы делаем до сих пор.

## 7.1 Ошибки компиляции

Ошибки компиляции — это ошибки в синтаксисе и построении, такие как орфографические ошибки, которые обнаруживаются компилятором во время компиляции — процесса, в ходе которого ваша программа транслируется в машинный код. Они являются наиболее частым типом ошибок. Компилятор печатает сообщения, которые могут быть полезными или бесполезными, когда он сталкивается с такой ошибкой.

Как правило, существует три вида ошибок компилятора:

- *Обычные ошибки* — компилятор попытается продолжить компиляцию после возникновения одной или нескольких таких ошибок, например,

  `missing ENDIF statements`
  (отсутствующие операторы ENDIF)

- *Фатальные ошибки* — компилятор не будет пытаться проводить дальнейшую компиляцию после обнаружения фатальной ошибки, т.е.

  `Program too complicated–too many strings`
  (Слишком сложная программа — слишком много строк)

- *Предупреждения* — это не совсем ошибки, они предназначены для информирования вас о том, что вы сделали что-то

необычное, что впоследствии может вызвать проблемы, например.

`Expression in IF construct is constant`
(Выражение в конструкции IF является константой)

или что вы использовали устаревшую функцию, например,

`Non-integer DO control variables are obsolescent`
(Нецелочисленные управляющие переменные DO устарели)

Существует большое количество сообщений об ошибках компилятора, которые будут перечислены в руководстве пользователя, прилагаемом к вашему конкретному компилятору. Поскольку компилятор не так умен, как вы, сообщения об ошибках иногда могут быть довольно бесполезными и даже вводить в заблуждение. Некоторые распространенные примеры приведены ниже.

`Inappropriate use of symbol` *X* `at line` *N*
(Неуместное использование символа *X* в строке *N*)

Имя *X* использовалось для представления более чем одного объекта, скорее всего, имени программы, а также переменной. Дублированное вхождение будет в строке *N*.

`Implicit type for` *X* `at line` *N*
(Неявный тип для *X* в строке *N*)

Переменная *X* не была объявлена явно после оператора `IMPLICIT NONE`. Таким образом будут обнаружены орфографические ошибки в объявленных переменных.

`Syntax error at line` *N*
(Синтаксическая ошибка в строке *N*)

Это одно из самых раздражающих сообщений, в котором содержится множество ошибок, например.

```
G = 9,8                    ! запятая вместо десятичной точки

IF (A = 0) X = 1           !=  вместо == в логическом выражении

IF (A == 0) THEN X = 1     ! неправильное использование THEN в
                           !  простом IF или неправильное размещение
                           !оператора после THEN

X = 1 + (2 * 3             ! непарные скобки
```

Возникает вопрос, почему компилятор не может быть немного более конкретным.

```
Symbol X referenced but not set at line N
```

Это полезное сообщение предупреждает вас о том, что программа не присвоила *X* никакого значения. Однако указанный номер строки, по-видимому, относится к последней строке программы.

Любопытно, однако, что следующее кодирование работает без ошибок:

```
IMPLICIT NONE

REAL X

X = X + 1
```

Результатом является мусор (1.4209760E+14 с компилятором FTN90), потому что, конечно, *X* не определен. Кажется, что компилятор считает, что, поскольку X появляется в левой части присваивания, он должен быть определен!

Ошибок компилятора существует гораздо больше, и вы, вероятно, уже сами обнаружите большое их количество. С опытом вы постепенно научитесь замечать свои ошибки.

## 7.2 Ошибки во время выполнения (Run-time Errors)

Если программа успешно скомпилирована, она запустится. Ошибки, возникающие на этом этапе, называются ошибками времени выполнения и неизменно являются фатальными, т.е. программа «вылетает». Сообщение об ошибке, например,

```
Floating point division by zero
```
(Деление величины с плавающей запятой на ноль)

или

```
Floating point arithmetic overflow
```
(Арифметическое переполнение величины с плавающей запятой)

генерируется. Последнее встречается довольно часто. Это происходит, например, при попытке вычислить действительное выражение, которое слишком велико, или когда `SQRT` имеет отрицательный аргумент, или когда аргумент `LOG` неположителен.

Некоторые компиляторы имеют интерактивные средства отладки, с помощью которых вы можете, например, выполнять программу строка за строкой, пока не найдете строку, в которой возникает ошибка времени выполнения, или где вы можете пометить строку в коде и выполнить ее до этой точки. Эти средства чрезвычайно

полезны, особенно для отладки больших программ; вам следует выяснить и использовать то, что предлагает ваш компилятор в этой строке.

*Перехват ошибок*

Fortran 90 имеет средства для перехвата и обработки определенных ошибок во времея выполнения, таких как ошибки ввода/вывода (например, попытка чтения за концом файла или из несуществующего файла). Они обсуждаются позже, когда мы будем иметь дело с расширенным вводом-выводом и обработкой файлов.

Ошибки в логике

Это ошибки в реальном алгоритме, который вы используете для решения проблемы, и их труднее всего найти; программа запускается, но дает неверные ответы! Еще хуже, если вы не понимаете, что ответы неверны. Следующие советы могут помочь вам проверить логику.

- Попробуйте запустить программу для некоторых особых случаев, когда вы знаете ответы.
- Если вы не знаете никаких точных ответов, попробуйте использовать свое понимание проблемы, чтобы проверить, соответствуют ли ответы правильному порядку величин.
- Попробуйте работать с программой вручную (или используйте средства отладки), чтобы увидеть, сможете ли вы определить, где что-то идет не так.

## 7.3 Ошибка округления

Иногда программа будет давать численные ответы на задачу, которые необъяснимо отличаются от того, что мы знаем как правильное математическое решение. Это может быть связано с ошибкой округления, которая возникает из-за конечной точности, доступной на компьютере, например, два или четыре байта на переменную вместо бесконечного числа.

Запустите следующий фрагмент программы:

```
X = 0.1

DO
  X = X + 0.001
  PRINT*, X
```

```
    IF (X == 0.2) EXIT
  END DO
```

Вы обнаружите, что вам нужно разбить программу, чтобы остановить, например, *ctrl-break* на ПК. $X$ никогда не имеет точного значения 0,2 из-за ошибки округления. На самом деле $X$ не соответствует значению 0,2 примерно на $10^{-9}$ , что можно увидеть, каждый раз выводя $X - 0{,}2$. Было бы лучше заменить предложение `EXIT` на

```
  IF (X > 0.2) EXIT
```

или

```
  IF (ABS(X - 0.2) < 1E-6) EXIT
```

В общем, всегда лучше проверять «равенство» двух реальных выражений таким образом, например,

```
IF(ABS(A - B) < 1E-6) PRINT*, 'A practically equal to B'
```

Ошибку округления можно уменьшить (хотя и не устранить полностью) за счет использования действительного вида с более высокой точностью, чем по умолчанию, например,

```
  REAL(KIND = 2) A, B
```

Ошибку округления также можно уменьшить путем математической перестановки задачи. Если известное квадратное уравнение записать в менее знакомой форме

$$x^2 - 2ax + e = 0$$

два решения могут быть выражены как

$$x_1 = a + \sqrt{a^2 - e}$$
$$x_2 = a - \sqrt{a^2 - e}$$

Если $e$ очень мало по сравнению с $a$, второй корень выражается как разность между двумя почти равными числами, и значительная значимость теряется. Например, принимая $a = 5\times10^6$ и $e = 1$, получаем $x_2 = -9{,}42\times10^{-9}$ с FTN90. Однако второй корень также может быть выражен математически как

$$x_2 = \frac{e}{a + \sqrt{a^2 - e}} \approx \frac{e}{2a}$$

Использование этой формы дает $x_2 = 10^{-7}$, что является более точным.

**Резюме**

- Ошибки компиляции — это ошибки в синтаксисе (кодировании).
- Во время работы программы возникают ошибки выполнения (времени выполнения).
- Ошибки ввода/вывода могут быть перехвачены во время выполнения.
- Средства отладки могут использоваться для работы с программой, оператор за оператором.
- Логические ошибки — это ошибки в алгоритме, используемом для решения задачи.
- Ошибка округления возникает из-за того, что компьютер может хранить числа только с конечной точностью. Это уменьшается, но не обязательно устраняется за счет использования вещественных чисел с более высокой точностью, чем по умолчанию.

**Упражнения**

7.1 Отношение Ньютона

$$\frac{f(x+h)-f(x)}{h}$$

может использоваться для оценки первой производной $f'(x)$ функции $f(x)$, если $h$ достаточно мало. Напишите программу, вычисляющую отношение Ньютона для функции $f(x) = x^2$ в точке $x = 2$ (точный ответ: 4) для значений $h$, начинающихся с 1 и каждый раз уменьшающихся в 10 раз. Эффект ошибки округления становится очевидным, когда $h$ становится «слишком маленьким», то есть меньше примерно $10^{-6}$.

7.2 Эта задача, демонстрирует еще одну численную задачу, называемую плохой обусловленностью, где небольшое изменение коэффициентов приводит к большому изменению решения. Показать, что решение системы

$$x + 5.000y = 17.0$$

$$1.5x + 7.501y = 25.503$$

есть $x = 2$, $y = 3$, используя программу для упражнения 5.5 с точностью по умолчанию. Теперь измените член в правой части второго уравнения на 25,501, изменение примерно на одну часть на 12 000, и

заметьте, что получается совершенно другое решение. Также попробуйте изменить этот член на 25,502, 25,504 и т.д. Если коэффициенты подвержены экспериментальным ошибкам, решение снова бессмысленно. Один из способов предвидеть такую ошибку — провести анализ чувствительности коэффициентов: изменить их все по очереди на один и тот же процент и посмотреть, как это повлияет на решение.

# Глава 8 Подпрограммы и модули

В главе 4 мы видели, что логика нетривиальной задачи может быть разбита на отдельные подпрограммы (или процедуры), каждая из которых выполняет определенную, четко определенную задачу. Часто бывает так, что такие подпрограммы могут использоваться множеством разных «основных» программ, а по сути разными пользователями одной и той же компьютерной системы. Fortran 90 позволяет реализовать эти подпрограммы как функции и подпрограммы, независимые от основной программы. Примерами являются процедуры для выполнения статистических операций, или для сортировки элементов, или для нахождения наилучшей прямой линии через набор точек, или для решения системы дифференциальных уравнений.

Подпрограммы могут быть внутренними или внешними. Полезные процедуры могут быть собраны вместе в виде библиотек. Такие коллекции называются модулями. Основные программы (то есть все, что мы видели до сих пор), внешние подпрограммы и модули называются программными единицами.

По сути, внутренняя подпрограмма содержится в другом программном модуле и, следовательно, компилируется с ним, в то время как внешняя подпрограмма нет — фактически она компилируется отдельно. Важное различие между двумя типами подпрограмм заключается в том, что внутренняя подпрограмма может использовать имена объектов, объявленных программным модулем, который ее содержит, тогда как внешняя подпрограмма не содержится в другом программном модуле.

Сначала разберемся с внутренними подпрограммами.

## 8.1 Внутренние подпрограммы

Существует два типа подпрограмм: функции и подпрограммы. Сначала мы рассмотрим функции. Мы уже видели, как использовать некоторые встроенные функции, предоставляемые Fortran 90, такие как `SIN`, `COS`, `LOG` и т.д. Вы можете написать свои собственные функции, которые будут использоваться таким же образом в программе. Прежде чем мы подробно обсудим правила, мы рассмотрим несколько примеров.

*Снова метод Ньютона*

Метод Ньютона можно использовать для решения общего уравнения $f(x) = 0$ путем повторения присваивания

$x$ становится как $x - \dfrac{f(x)}{f'(x)}$

где $f'(x)$ – первая производная от $f(x)$, пока $f(x)$ не приблизится достаточно близко к нулю.

Предположим, что $f(x) = x^3 + x - 3$. Затем $f'(x) = 3x^2 + 1$. Программа, представленная ниже, использует метод Ньютона для решения этого уравнения, начиная с $x = 2$ и останавливается либо тогда, когда абсолютное значение $f(x)$ меньше $10^{-6}$, либо, скажем, после 20 итераций. Он использует две функции: `F(X)` для $f(x)$ и `DF(X)` для $f'(x)$.

```
PROGRAM Newton
! Solves f(x) = 0 by Newton's method

IMPLICIT NONE
INTEGER  ::  Its       = 0        ! iteration counter
INTEGER  ::  MaxIts    = 20       ! maximum iterations
LOGICAL  ::  Converged = .false.  ! convergence flag
REAL     ::  Eps       = 1e-6     ! maximum error
REAL     ::  X         = 2        ! starting guess

DO WHILE (.NOT. Converged .AND. Its < MaxIts)
  X = X–F(X) / DF(X)
  PRINT*, X, F(X)
  Its = Its + 1
  Converged = ABS( F(X) ) <= Eps
END DO

IF (Converged) THEN
  PRINT*, 'Newton converged'
ELSE
  PRINT*, 'Newton diverged'
END IF

CONTAINS
  FUNCTION F(X)
  ! problem is to solve f(x) = 0
    REAL F, X
    F = X ** 3 + X–3
  END FUNCTION F

  FUNCTION DF(X)
```

```
    ! first derivative of f(x)
      REAL DF, X
      DF = 3 * X ** 2 + 1
    END FUNCTION DF
  END PROGRAM Newton
```

Стоит обратить внимание на то, что есть два условия, которые остановят цикл DO: либо сходимость, либо завершение 20 итераций. В противном случае программа может работать бесконечно.

*Вращение координатных осей*

Функции особенно полезны, когда арифметические выражения, которые могут стать длинными и громоздкими, необходимо многократно вычислять. Хорошим примером является вращение декартовой системы координат. Если такую систему повернуть против часовой стрелки на угол в радианы, новые координаты ($x'$, $y'$) точки относительно повернутых осей задаются формулой

$$x' = x\cos(a) + y\sin(a)$$

$$y' = -x\sin(a) + y\cos(a)$$

где ($x$, $y$) – его координаты до поворота осей. Для определения новых координат можно использовать следующие функции:

```
  FUNCTION Xnew( X, Y, A )
    REAL XNew, X, Y, A
    Xnew = X * COS( A ) + Y * SIN( A )
  END FUNCTION Xnew

  FUNCTION YNew( X, Y, A )
    REAL YNew, X, Y, A
    YNew =-X * SIN( A ) + Y * COS( A )
  END FUNCTION Ynew
```

Внутренние функции

Поскольку функции и подпрограммы очень похожи, общие функции описываются с помощью ссылки на них как на подпрограммы. Большинство следующих правил применимы также к внешним подпрограммам, если не указано иное. Все внутренние подпрограммы размещаются между оператором `CONTAINS` и оператором `END` основной программы. Подпрограммы выглядят почти как основная программа, за исключением их заголовков и операторов

END. Внутренние подпрограммы не могут содержать другие подпрограммы и, следовательно, сами по себе могут не иметь оператора CONTAINS.

Общий синтаксис внутренней функции:

```
FUNCTION Name( [список аргументов] )
  [операторы описания]
  [исполняемые операторы]
END FUNCTION [Name]
```

Выражение

```
FUNCTION Name( список аргументов )
```

называется оператором функции, заголовком или объявлением. Обратите внимание, что если в основной программе есть оператор IMPLICIT NONE (а правильный стиль программирования настаивает на том, чтобы он был), имя функции и аргументы должны быть объявлены с типом. Хотя это можно сделать в основной программе, рекомендуется объявлять имя функции и аргументы в самом теле функции.

Поскольку значение связано с именем функции, это значение должно быть присвоено имени функции в теле функции. Следует иметь в виду, что когда имя функции появляется в левой части оператора присваивания, его аргументы должны быть опущены, например,

```
YNew =–X * SIN( A ) + Y * COS( A )
```

в функции YNew выше. Это означает, что функция возвращает полученное значение вызывающей или хост-программе. Значение функции также может быть возвращено с помощью предложения RESULT.

Внутренняя подпрограмма автоматически имеет доступ ко всем объектам хоста. Таким образом, переменные, объявленные в основной программе, являются глобальными в том смысле, что они доступны во всей области действия основной программы. Эта область включает все внутренние подпрограммы. Иногда это может привести к серьезным ошибкам, поэтому внутренние подпрограммы следует использовать только для довольно небольших задач, характерных для их конкретного хоста. Более общие процедуры должны быть написаны как внешние подпрограммы.

Поскольку внутренняя подпрограмма в некотором смысле объявлена в своем хосте, это правило области видимости также подразумевает, что внутренние подпрограммы известны друг другу,

т.е. они могут вызывать друг друга. Однако, если переменная повторно объявлена в подпрограмме, эта подпрограмма больше не имеет доступа к исходной переменной с тем же именем, объявленной в хосте.

*Аргументы*

Аргументы в операторе подпрограммы, например, *X*, *Y* и *A* в

```
FUNCTION YNew( X, Y, A )
```

являются фиктивными аргументами. То есть они существуют только с целью определения функции. Они представляют собой общие переменные одного типа. Следовательно, не обязательно использовать одни и те же имена при вызове или вызове подпрограммы. Например, `YNew` может быть вызван в основной программе оператором

```
YNew( U, V, Pi/2 )
```

Можно представить, что значения фактических аргументов `U`, `V` и `Pi/2` копируются в фиктивные аргументы `X`, `Y` и `A` соответственно. Говорят, что фактические аргументы передаются подпрограмме. Фактический аргумент, который является именем переменной, должен иметь параметры типа и типа того же типа, что и соответствующий ему фиктивный аргумент. Точный способ передачи аргументов обсуждается ниже.

Вы должны знать, что если значение фиктивного аргумента изменяется внутри функции, это изменение «копируется обратно» в соответствующий фактический аргумент в основной программе, если это переменная. Это нежелательный побочный эффект функции, и его следует избегать в интересах звукового программирования. Если вы хотите изменить фактический аргумент, правильным транспортным средством является подпрограмма.

Локальные и глобальные переменные

Следующая программа содержит внутреннюю функцию `Fact(N)`, которая вычисляет *n*! Что-то не так. Посмотрите, сможете ли вы вручную решить, какими будут первые несколько строк вывода, прежде чем запускать его.

```
PROGRAM Factorial
IMPLICIT NONE
INTEGER I
```

```
DO I = 1, 10
  PRINT*, I, Fact(I)
END DO

CONTAINS
  FUNCTION Fact( N )
    INTEGER Fact, N, Temp
    Temp = 1
    DO I = 2, N
      Temp = I * Temp
    END DO
    Fact = Temp
  END FUNCTION
END
```

Проблема в том, что `I` — *глобальная* переменная, т.е. имя `I` представляет одну и ту же переменную внутри и вне функции. Факт сначала вызывается, когда `I = 1`, что является первым записанным значением. Это значение передается фиктивному аргументу функции *N*. Тот же самый `I` получает теперь начальное значение 2 циклом `DO` внутри `Fact`, но поскольку оно больше `N`, цикл `DO` не выполняется, так что все еще есть значение 2, когда `Fact` возвращается для печати в основной программе. Однако теперь `I` увеличивается до 3 в цикле `DO` в основной программе, что является значением, которое оно имеет, когда происходит второй вызов `Fact`. Таким образом, `Fact` никогда не вычисляется для четного значения `I`. Все это является следствием того, что переменная `I` является *глобальной*. Проблема решается переобъявлением `I` в функции, чтобы сделать ее *локальной*. Вы должны взять за правило объявлять все переменные, используемые в подпрограммах. Таким образом, вы никогда не сможете непреднамеренно использовать глобальную переменную в неправильном контексте. Если вам нужно получить информацию в подпрограмму от ее хоста, самый безопасный способ сделать это — использовать фиктивные аргументы. Когда существует большое количество такой информации, которая должна использоваться многими подпрограммами, лучшим решением будет объявить глобальные переменные в модуле, а подпрограммам, которым нужен доступ к этим переменным, использовать модуль. Использование `IMPLICIT NONE` в основной программе особенно важно при наличии внутренних подпрограмм. Это заставляет вас объявлять все локальные переменные и фиктивные аргументы, что обеспечивает хороший стиль программирования.

Функция без аргументов должна иметь пустые круглые скобки, например. `Fung()`.

### Оператор **RETURN**

Обычный выход из подпрограммы происходит в ее операторе `END`. Однако иногда бывает удобно выйти из других точек. Это можно сделать с помощью оператора `RETURN`. Следует избегать чрезмерного использования `RETURN`, так как это очень легко приводит к «спагетти» (неструктурированному коду).

### *Операторные функции*

В более старых версиях Фортрана функция могла быть определена в одной строке, например,

```
F(X) = X ** 3 + X - 3
```

Эта форма поддерживается Fortran 90, но не рекомендуется, поскольку она не соответствует многим общим правилам для подпрограмм.

### Внутренние подпрограммы

Подпрограммы очень похожи на функции. Различия:

- С именем подпрограммы не связано никакое значение, поэтому ее нельзя объявлять.
- Подпрограмма вызывается оператором `CALL`.
- В определении и операторе `END` используется ключевое слово `SUBROUTINE`.
- У подпрограммы не должно быть никаких аргументов, и в этом случае имя пишется без круглых скобок, например.

```
CALL PLONK
```

Следующая программа выводит строку с подпрограммой `PrettyLine`, имеющей два фиктивных аргумента. `Num` – количество печатаемых символов; `Symbol` – это код ASCII символов, которые должны быть напечатаны.

```
IMPLICIT NONE
CALL PrettyLine( 5, 2 )
```

```
CONTAINS
  SUBROUTINE PrettyLine( Num, Symbol )
    INTEGER I, Num, Symbol
    CHARACTER*80 Line
    DO I = 1, Num
      Line(I:I) = ACHAR( Symbol )
    END DO
    PRINT*, Line
  END SUBROUTINE
END
```

В следующем примере показано, как можно использовать фиктивные аргументы для передачи информации вызывающей программе — в этом случае их значения меняются местами:

```
IMPLICIT NONE
REAL A, B

READ*, A, B
CALL SWOP( A, B )
PRINT*, A, B

CONTAINS
  SUBROUTINE SWOP( X, Y )
    REAL Temp, X, Y
    Temp = X
    X = Y
    Y = Temp
  END SUBROUTINE
END
```

На самом деле происходит то, что фактические аргументы A и B передаются по ссылке. В результате значения фактических аргументов передаются фиктивным аргументам, а любые изменения, внесенные в фиктивные аргументы, копируются обратно в фактические аргументы. Таким образом, информация может быть возвращена из подпрограммы. Это может иметь неприятные побочные эффекты. Вы можете непреднамеренно изменить фиктивный аргумент внутри подпрограммы — после возврата в вызывающую программу соответствующий фактический аргумент также будет изменен, что может привести к катастрофическим последствиям. В Фортране 90 есть способ предотвратить это, называемый намерением аргумента.

Поскольку Temp в приведенном выше примере объявлен в подпрограмме, он локален для нее и недоступен для основной программы. Таким образом, общий синтаксис внутренней подпрограммы таков:

```
SUBROUTINE Name[( список аргументов )]
```

```
                    [операторы описания]
                    [исполняемые операторы]
          END SUBROUTINE [Name]
```

## 8.2 Основная программа

Каждая полная программа должна иметь одну и только одну основную программу, которая имеет вид

```
PROGRAM name
  [операторы описания]
  [исполняемые операторы]
[CONTAINS
  внутренние подпрограммы]
END [PROGRAM [name]]
```

Если последний оператор перед оператором CONTAINS не приводит к переходу (а он не должен), управление внутренними подпрограммами переходит к оператору END, и программа останавливается. Другими словами, внутренние подпрограммы выполняются только в том случае, если они правильно вызваны оператором CALL в случае подпрограммы или обращением к ее имени в случае функции.

## 8.3 Внешние подпрограммы

Внешняя подпрограмма находится в отдельном от основной программы файле. Обычно он выполняет определенную задачу и является внешним, чтобы быть доступным для многих различных вызывающих программ. За исключением заголовка и оператора END, подпрограмма внешне идентична основной программе:

```
SUBROUTINE name[( список аргументов )]
  [операторы описания]
  [исполняемые операторы]
[CONTAINS
  внутренние подпрограммы]
END [SUBROUTINE [name]]
```

или

```
FUNCTION name([список аргументов ])
  [операторы описания]
  [исполняемые операторы]
[CONTAINS
  внутренние подпрограммы]
END [FUNCTION [name]]
```

Обратите внимание на небольшие, но существенные различия между внешними и внутренними подпрограммами:

- внешние подпрограммы могут сами содержать внутренние подпрограммы (которые будут доступны только внешней подпрограмме хоста); внутренние подпрограммы не могут содержать дополнительных внутренних подпрограмм
- ключевое слово FUNCTION или SUBROUTINE является необязательным в операторе END внешней подпрограммы, но обязательным во внутреннем операторе END подпрограммы.

Обратите также внимание на то, что, поскольку внешняя подпрограмма находится в отдельном от основной программы файле, ее необходимо компилировать отдельно. В компиляторе промежуточный тип машинного кода, называемый перемещаемым двоичным файлом, создается в файле с расширением .OBJ. Это, в свою очередь, должно быть связано с вызывающей программой с помощью специальной программы, называемой компоновщиком, в результате чего получается EXE-версия основной программы. В вашем руководстве по компилятору будет подробно описано, как это сделать. После окончательной отладки внешнюю подпрограмму никогда не нужно перекомпилировать, только компоновать. Это предотвратит трату времени на компиляцию снова и снова, что имело бы место, если бы это была внутренняя подпрограмма.

В качестве примера перепишем внутреннюю подпрограмму SWOP из раздела 8.1 как внешнюю подпрограмму. Основная программа (в одном файле) становится

```
IMPLICIT NONE
EXTERNAL SWOP
REAL A, B

READ*, A, B
CALL SWOP( A, B )
PRINT*, A, B

END
```

и внешняя подпрограмма (в отдельном файле) имеет вид

```
SUBROUTINE SWOP( X, Y )
  REAL Temp, X, Y
  Temp = X
  X = Y
  Y = Temp
END SUBROUTINE
```

Оператор EXTERNAL обсуждается ниже. Теперь вы должны попробовать скомпилировать, скомпоновать и запустить этот пример. Если вам нужно более одной внешней подпрограммы в одном и том же файле, вы должны использовать.

Оператор **EXTERNAL**

Если вы случайно использовали имя встроенной подпрограммы для внешней подпрограммы, компилятор по умолчанию предположил бы, что вы ссылаетесь на внутреннюю подпрограмму, поэтому ваша внешняя подпрограмма будет недоступна. Вы можете подумать, что знаете имена всех встроенных подпрограмм и что этой проблемы не возникнет. Однако у вас могут возникнуть проблемы при переносе кода на другую установку, поскольку стандарт позволяет компиляторам предоставлять дополнительные встроенные подпрограммы.

Чтобы избежать этой проблемы, имена всех внешних подпрограмм должны быть указаны в операторе EXTERNAL, который должен стоять после любого оператора USE или IMPLICIT. Такое имя внешней подпрограммы гарантирует, что она связана как внешняя подпрограмма, и делает любую внутреннюю подпрограмму с таким именем недоступной. Эта практика настоятельно рекомендуется.

## 8.4 Интерфейсные блоки

Чтобы компилятор правильно генерировал вызовы подпрограмм, ему необходимо знать определенные сведения о подпрограмме: имя, количество и тип аргументов и т.д. Этот набор информации называется *интерфейсом* подпрограммы. В случае встроенных подпрограмм, внутренних подпрограмм и подпрограмм модулей интерфейс всегда известен компилятору и называется *явным*.

Однако, когда компилятор генерирует вызов внешней подпрограммы, эта информация недоступна, и говорят, что интерфейс подпрограммы *неявный*.

Ранее видели, что оператора EXTERNAL в вызывающей программе достаточно, чтобы предоставить компилятору имя внешней подпрограммы, и это позволяет ее найти и скомпоновать. Однако интерфейс по-прежнему является неявным, и в более сложных случаях (например, с необязательными или ключевыми аргументами, как обсуждается ниже) для удовлетворительного интерфейса требуется дополнительная информация. В Fortran 90 есть механизм, называемый

*интерфейсным блоком*, который позволяет создавать явный интерфейс.

Общая форма блока интерфейса, который должен быть помещен в область действия вызывающей программы, выглядит следующим образом:

```
INTERFACE
  тело интерфейса
END INTERFACE
```

где *тело интерфейса* может быть точной копией заголовка подпрограммы, объявлений ее аргументов и результатов, а также ее оператора END. Однако имена аргументов могут быть изменены, и могут быть включены другие спецификации (например, для локальной переменной), но не операторы DATA или FORMAT или внутренние подпрограммы.

Вызывающую программу для внешней подпрограммы SWOP выше можно переписать с блоком интерфейса следующим образом:

```
IMPLICIT NONE

INTERFACE
  SUBROUTINE SWOP( X, Y )
    REAL X, Y
  END SUBROUTINE
END INTERFACE

REAL A, B

READ*, A, B
CALL SWOP( A, B )
...
```

Более общий доступ к блокам интерфейса может быть обеспечен с помощью модулей. Блоки интерфейса также используются для перегрузки нескольких имен подпрограмм одним «общим» именем.

*Когда использовать интерфейсные блоки*

Всегда можно использовать интерфейсный блок для внешней процедуры. Однако бывают ситуации, когда его необходимо использовать. Мы столкнемся с большинством этих ситуаций позже; они собраны здесь для удобства поиска.

Блок интерфейса необходим для вызова внешней процедуры в следующих случаях:

- когда процедура вызывается с ключевым словом и/или отсутствующим аргументом;

- когда вызывается процедура, определяющая или перегружающая оператор или присваивание;
- когда вызываемая процедура является функцией со значением массива или символьной функцией, которая не является ни постоянной, ни предполагаемой длиной;
- когда вызываемая процедура имеет фиктивный аргумент, который является массивом предполагаемой формы, указателем или целью;
- когда процедура является фиктивным или фактическим аргументом;
- когда процедура вызывается с использованием универсального имени.

## 8.5 Программные модули

Напомним, что существует три типа программных модулей: основная программа, внешняя подпрограмма и модуль. Модуль отличается от подпрограммы двумя важными моментами:

- модуль может содержать более одной подпрограммы (называемой подпрограммами модуля);
- модуль может содержать операторы объявлений и спецификаций, которые доступны для всех модулей программы, использующих модуль.

Модули также собираются отдельно.

В следующем примере подпрограмма `SWOP` размещена в модуле `MyUtils`. Для иллюстрации модуль также объявляет реальный параметр `Pi`. В основной программе теперь есть инструкция `USE MyUtils`, которая делает модуль доступным для нее. Таким образом, имя `Pi` известно основной программе. Измененная основная программа:

```
USE MyUtils

IMPLICIT NONE
REAL A, B

READ*, A
B = Pi
CALL SWOP( A, B )
PRINT*, A, B

END
```

в то время как модуль (снова сохраненный в отдельном файле)

```
MODULE MyUtils
  REAL, PARAMETER :: Pi = 3.1415927
CONTAINS
  SUBROUTINE SWOP( X, Y )
    REAL Temp, X, Y
    Temp = X
    X = Y
    Y = Temp
  END SUBROUTINE SWOP
END MODULE MyUtils
```

Обратите внимание, что оператор EXTERNAL теперь не является ни необходимым, ни фактически правильным, поскольку SWOP больше не является внешней подпрограммой — технически теперь это подпрограмма модуля.

Общая форма модуля

```
MODULE name
  [операторы объявления]
[CONTAINS
  подпрограммы модуля]
END [MODULE [name]]
```

Доступ к нему осуществляется с помощью оператора USE в модуле хост-программы, который должен предшествовать всем остальным операторам:

```
USE module-name
```

*Подпрограмма модуля* имеет точно такую же форму, как и внешняя подпрограмма, за исключением того, что в операторе END должны присутствовать ключевые слова FUNCTION или SUBROUTINE. Он имеет доступ ко всем другим объектам в модуле, включая все переменные, объявленные в нем. Обратите внимание, что подпрограмма модуля может содержать свои собственные внутренние подпрограммы. Теперь мы столкнулись со всеми тремя типами программных модулей. Вложение внутренних, внешних и модульных подпрограмм относительно этих программных модулей показано на рис. 8.1.

Модуль может использовать через USE другие модули, хотя он не может обращаться к себе косвенно через цепочку операторов USE в разных модулях. Стандарт не требует заказа модулей. Однако при разработке библиотек модулей вы должны попытаться спроектировать

иерархию, чтобы последующие модули использовали только более ранние.

Поскольку модуль может содержать *операторы объявления*, которые доступны для всех программных модулей хоста, глобальные переменные могут быть объявлены таким образом для использования всеми хостами, обращающимися к модулю. Эта функция особенно полезна для создания более сложных объявлений, например объявлений для производных типов, доступных глобально. В частности, интерфейсные блоки могут быть сгруппированы в модули. В качестве примера снова рассмотрим внешнюю подпрограмму `SWOP` из раздела 8.3. Его интерфейс может быть в модуле `MyMod`,

```
MODULE MyMod

INTERFACE
  SUBROUTINE SWOP( U, V )
    REAL U, V
  END SUBROUTINE
END INTERFACE

END MODULE
```

доступ к которому можно получить с помощью оператора `USE MyMod` в вызывающей программе.

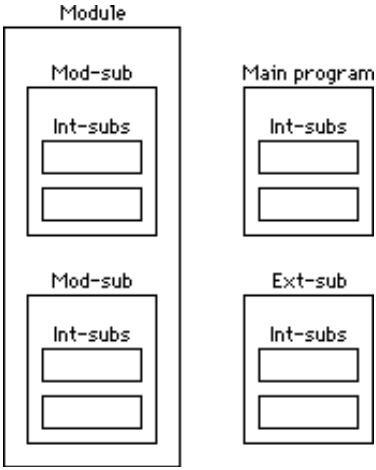


Рисунок 8.1 Подпрограммы и программные блоки (Mod-sub: модульная подпрограмма; Int-subs: внутренняя подпрограмма; Ext-sub: внешняя подпрограмма)

### Оператор **USE**

Мы видели, что оператор USE разрешает доступ к сущностям в модуле. Существуют две специальные формы оператора USE, влияющие на режим доступа. Может быть неудобно (или невозможно) использовать имя конкретного объекта в модуле. Например, каждый из двух независимо написанных модулей может иметь подпрограмму с одинаковым именем. Объект модуля может иметь длинное и громоздкое имя. Объекты модуля могут быть переименованы для использования в основной программе под новым именем. Например, если модуль YourMod имеет подпрограмму или переменную с именем YourPlonk, ее можно переименовать для использования под именем MyPlonk следующим образом:

```
USE YourMod, MyPlonk => YourPlonk
```

Общая форма

```
USE module-name, rename-list
```

где каждый элемент в *rename-list* имеет форму

```
new-name /=>/ original-name
```

В списке может появиться любое количество переименований. Это использование на самом деле не рекомендуется и должно использоваться только в крайнем случае. Было бы лучше использовать текстовый редактор, чтобы изменить оригинальные имена.

Другой способ повлиять на доступ к модулю — использовать предложение ONLY в операторе USE. Например, оператор

```
USE YourMod, ONLY : X, Y
```

разрешает доступ только к объектам X и Y модуля. Элементы, следующие за двоеточием, также могут быть переименованы. Каждый доступный модуль должен отображаться в отдельном операторе USE.

### Атрибуты **PUBLIC** и **PRIVATE**

Как мы видели, все объекты в модуле по умолчанию доступны для любого программного модуля, использующего этот модуль. Однако доступ можно ограничить, указав переменную с атрибутом PRIVATE в ее объявлении:

```
REAL, PRIVATE :: X
```

В качестве альтернативы переменная или подпрограмма могут быть указаны с этим атрибутом в отдельном операторе:

```
PRIVATE X, SWOP
```

Это означает, что объекты X и SWOP недоступны вне модуля. Однако они по-прежнему доступны внутри модуля. Атрибут PUBLIC (по умолчанию) может быть указан аналогичным образом. Оператор PUBLIC или PRIVATE без списка объектов подтверждает или сбрасывает значение по умолчанию. Итак, операторы

```
PRIVATE
PUBLIC SWOP
```

делают все объекты в модуле как PRIVATE по умолчанию, кроме SWOP.

## 8.6 Область действия

Областью действия метки или имени является набор строк, в которых это имя или метка могут использоваться однозначно. Правила области действия для меток и имен отличаются.

### Область действия меток

Единственное допустимое использование меток в этом пособии — это операторы FORMAT. Однако метки также можно использовать в сочетании с пресловутым оператором GOTO и для завершения цикла DO, хотя такие практики не рекомендуются.

*Единица области действия*

*Единица области действия* — это одна из следующих единиц.

- определение производного типа;
- тело интерфейса подпрограммы; за исключением любых определений производных типов и тел интерфейсов, содержится в нем; или же
- Программный модуль или подпрограмма, за исключением определений производных типов, тел интерфейсов и содержащиеся в нем подпрограммы.

Каждая подпрограмма, внутренняя или внешняя, имеет свой собственный независимый набор меток. Так, например, одна и та же метка может использоваться в основной программе и нескольких ее

внутренних подпрограммах без двусмысленности. Таким образом, областью действия метки является основная программа или подпрограмма, исключая любые содержащиеся в ней внутренние подпрограммы.

### Область действия имен

Область действия имени, объявленного в программе, простирается от оператора PROGRAM до оператора END и не распространяется ни на какие внешние подпрограммы, которые могут быть вызваны. Имя, объявленное в подпрограмме, имеет область действия от заголовка подпрограммы до ее оператора END. Из этого следует, что имя, объявленное в основной или внешней подпрограмме, имеет область действия во всех содержащихся в нем подпрограммах.

Область действия имени, объявленного во внутренней подпрограмме, распространяется только на эту внутреннюю подпрограмму и не распространяется на другие внутренние подпрограммы. Из этого следует, что область действия имени, объявленного в программе или подпрограмме, не включает внутренние подпрограммы, в которых оно повторно объявлено.

Область действия имени внутренней подпрограммы, а также количество и тип ее аргументов распространяется на всю содержащую программу или подпрограмму (и, следовательно, на все остальные внутренние подпрограммы). Область действия имени, объявленного (где имя теперь является именем переменной или подпрограммы) в модуле, распространяется на любые программные единицы, которые используют модуль, но, очевидно, исключает любые внутренние подпрограммы, в которых имя (в случае переменной) объявляется повторно.

Описав основные элементы области действия, полезно уточнить концепцию, определив *единицу области действия*. Некоторые выводы следуют из этого определения:

- Объекты, объявленные в разных единицах области действия, всегда разные, даже если они имеют одинаковые имена и свойства.
- В пределах *единицы области действия* каждый именованный объект данных, подпрограмма, производный тип, именованная конструкция и группа списка имен должны иметь отличное имя, за исключением общих имен подпрограмм.
- Имена программных модулей являются глобальными, поэтому программные модули в одной и той же программе могут не

иметь одинаковых имен, а также не могут иметь имена каких-либо объектов, объявленных в программном модуле.

Существует отдельное правило области действия для переменной DO подразумеваемого (или скрытого) DO; он распространяется только на подразумеваемый DO.

## 8.7 Общие имена подпрограмм: перегрузка

Вы, возможно, задавались вопросом, увидев, как фактические и фиктивные аргументы должны точно совпадать по типу и количеству, как некоторые из встроенных функций могут принимать аргументы более чем одного типа. Например, аргумент `ABS` может быть целым, действительным или даже комплексным! Ответ состоит в том, чтобы использовать хитрый прием, предоставляемый Fortran 90: перегрузку.

*Перегрузка* – это общая возможность, предоставляемая многими современными языками программирования. В этом контексте это возможность вызывать несколько разных подпрограмм с одним и тем же общим именем. В принципе, подпрограммы с разными именами пишутся для разных типов аргументов; их конкретные имена затем перегружаются одним общим именем для всех них. Родовое имя называется; компилятор решает, какое конкретное имя вызывать за кулисами в соответствии с типом фактических аргументов. Рассмотрим снова внешнюю подпрограмму `SWOP(X, Y)`. Он принимает только реальные аргументы. Мы можем заставить его принимать целочисленные аргументы несколькими способами.

Один из них заключается в написании двух отдельных внешних подпрограмм, `SwopReals` и `SwopIntegers`, с вещественными и целочисленными аргументами соответственно, например.

```
SUBROUTINE SwopReals( X, Y )
    REAL X, Y, Temp
    Temp = X
    X = Y
    Y = Temp
END SUBROUTINE SwopReals
```

и

```
SUBROUTINE SwopIntegers( X, Y )
    INTEGER X, Y, Temp
    Temp = X
    X = Y
    Y = Temp
END SUBROUTINE SwopIntegers
```

Поскольку это должны быть внешние подпрограммы, каждая из них должна находиться в отдельном файле и компилироваться отдельно. Затем перегрузку можно выполнить в основной программе с помощью оператора INTERFACE, в котором указывается общее имя (SWOP) и тела интерфейса для двух перегруженных подпрограмм:

```
PROGRAM Main

INTERFACE SWOP
  SUBROUTINE SwopReals( X, Y )
    REAL X, Y, Temp
  END SUBROUTINE SwopReals
  SUBROUTINE SwopIntegers( X, Y )
    INTEGER X, Y, Temp
  END SUBROUTINE SwopIntegers
END INTERFACE

REAL A, B
INTEGER I, J
...
CALL SWOP( A, B )
CALL SWOP( I, J )
...
```

Конкретное имя может совпадать с общим именем, если это более удобно. Общее имя может совпадать с другим доступным общим именем, и в этом случае все подпрограммы с этим общим именем вызываются через него. Таким образом, встроенные функции могут быть расширены для приема аргументов производного типа.

Если вы хотите перегрузить подпрограмму модуля, интерфейс уже является явным, поэтому некорректно указывать тело интерфейса. Вместо этого вы должны включить оператор

```
MODULE PROCEDURE procedure-names
```

в интерфейсный блок, где имена процедур — это процедуры (подпрограммы), которые необходимо перегрузить. Таким образом, если бы подпрограммы SwopReals и SwopIntegers были определены в модуле, блок интерфейса был бы

```
INTERFACE SWOP
  MODULE PROCEDURE SwopReals, SwopIntegers
END INTERFACE
```

Этот интерфейсный блок может быть размещен в самом модуле. Попытайтесь выполнить этот подход. Позже мы увидим, как перегрузить встроенные операторы и присваивание, чтобы распространить эти операции на производные типы.

## 8.8 Макеты

Большая программа будет иметь много подпрограмм. Планировать и кодировать их все перед компиляцией — значит напрашиваться на неприятности. Полезным приемом является использование макетов, которые определяют имена подпрограмм, но ничего не делают (сначала). Затем заполняйте макеты по одной, компилируя после каждой заливки. Таким образом, намного проще отловить все ошибки компилятора. Вы можете изначально определить подпрограммы как внутренние, со всеми объявленными локальными переменными, перемещая их в модули по мере их завершения. Это избавляет от необходимости перекомпилировать все по мере заполнения все большего количества макетов.

```
PROGRAM BigOne

IMPLICIT NONE

CALL FIRST
CALL LAST

CONTAINS
  SUBROUTINE FIRST
    PRINT*, 'First here'
  END SUBROUTINE

  SUBROUTINE LAST
    PRINT*, 'Last here'
  END SUBROUTINE
END PROGRAM BigOne
```

Возможно, сейчас он мало что делает, но, по крайней мере, компилируется!

## 8.9 Рекурсия

Многие математические функции (и более общие процедуры) могут быть определены рекурсивно, то есть в терминах более простых случаев самих себя. Например, факториальная функция может быть определена рекурсивно как

$$n! = n*(n - 1)!$$

учитывая что 1! определяется как 1.

Чтобы реализовать это определение как функцию, необходимо, чтобы функция вызывала сама себя. Обычно в Fortran 90 это невозможно. Однако, если перед заголовком функции стоит ключевое

слово RECURSIVE, функция может вызвать сама себя. Если вызов прямой (т.е. имя функции встречается в теле определения функции), в заголовок функции необходимо добавить предложение RESULT, чтобы использовать другое (локальное) имя для функции. Это показано ниже, где функция Factorial определена рекурсивно:

```
IMPLICIT NONE
INTEGER I

I = 10
PRINT*, I, Factorial(I)

CONTAINS
  RECURSIVE FUNCTION Factorial( N ) RESULT (Fact)
    INTEGER Fact, N
    IF( N == 1 ) THEN
      Fact = 1
    ELSE
      Fact = N * Factorial( N-1 )
    END IF
  END FUNCTION
END
```

Пункт RESULT необходим, поскольку имя Factorial может не отображаться в левой части присваивания. Обратите внимание, что имя Factorial не должно быть объявлено в операторе INTEGER: достаточно объявления Fact.

*Рекурсия* – сложная тема, хотя она кажется обманчиво простой. Вы можете задаться вопросом, как на самом деле работает рекурсивная функция Factorial. Важно различать вызываемую функцию и выполняемую. Когда происходит первоначальный вызов, *N* имеет значение 10, поэтому оценивается предложение ELSE в Factorial. Однако значение Factorial(9) на данном этапе неизвестно, поэтому делается копия всех операторов в функции, которые нужно будет выполнить после того, как значение Factorial(9) станет известно. Ссылка на Factorial(9) заставляет Factorial вызывать себя, на этот раз со значением 9 для *N*. Снова оценивается предложение ELSE, и снова Fortran 90 обнаруживает, что ему неизвестно значение Factorial(8). Таким образом, делается еще одна (другая) копия всех операторов, которые необходимо будет выполнить после того, как значение Factorial(8) станет известно. Таким образом, при каждом вызове Factorial создаются отдельные копии всех операторов, которые еще предстоит выполнить. Наконец, компилятор находит значение *N* (1), для которого ему известно

значение Factorial, так что, наконец, он может начать выполнять (в обратном порядке) кучу операторов, которые были запружены в памяти.

Это обсуждение иллюстрирует, что рекурсию следует использовать осторожно. Хотя он идеально подходит для использования в таком случае, он может потреблять огромное количество компьютерной памяти и времени.

Следует отметить, что предложение RESULT может использоваться необязательно, когда функция не определена рекурсивно — другими словами, никогда не бывает неправильно иметь предложение RESULT. Здесь это показано с нерекурсивной версией Factorial:

```
IMPLICIT NONE
INTEGER I

DO I = 1, 10
  PRINT*, I, Factorial( I )
END DO

CONTAINS
  FUNCTION Factorial( N ) RESULT (Fact)
    INTEGER Fact, I, N
    Fact = 1
    DO I = 2, N
      Fact = I * Fact
    END DO
  END FUNCTION
END
```

Когда подпрограмма вызывает саму себя напрямую, необходимо также использовать ключевое слово RECURSIVE. В приведенном ниже примере Factorial переписывается как рекурсивная подпрограмма. Это немного более тонко.

```
IMPLICIT NONE
INTEGER F, I

DO I = 1, 10
  CALL Factorial( F, I )
  PRINT*, I, F
END DO

CONTAINS
  RECURSIVE SUBROUTINE Factorial( F, N )
    INTEGER F, N
    IF (N == 1) THEN
      F = 1
```

```
    ELSE
      CALL Factorial( F, N-1 )
      F = N * F
    END IF
  END SUBROUTINE
END
```

В первой версии этой программы, по ошибке было поставлено *N* вместо *N*-1 в качестве аргумента рекурсивного вызова. Следовательно, программа не могла закончиться, так как `Factorial` всегда вызывался со значением *N*, равным 10. «Управляющий оператор», `F` = 1, никогда не мог быть выполнен. Имейте в виду!

Тонкость в этом примере заключается в том, чтобы определить, следует ли размещать оператор `F = N * F` до или после рекурсивного вызова `Factorial`. Попробуйте запустить программу с `F = N * F` перед вызовом `Factorial`. Отличие в том, что теперь при каждом вызове выполняется `F = N * F`, а не делается копия. В результате последнее выполнение устанавливает `F` в 1, и это его значение при возврате. В упражнениях есть еще примеры рекурсивных функций. Рекурсивная подпрограмма используется для реализации алгоритма быстрой сортировки.

**Резюме**

- Большие программы следует разбивать на подпрограммы (процедуры) для выполнения более простых задач.
- Подпрограммы могут быть внутренними или внешними.
- Внешние подпрограммы компилируются отдельно от основной программы.
- Подпрограммы модуля – это подпрограммы, собранные в отдельно скомпилированные модули (библиотеки).
- Программный модуль – это основная программа, внешняя подпрограмма или модуль.
- Внутренние подпрограммы содержатся в программных блоках.
- Подпрограммы состоят из функций или подпрограмм.
- Функция возвращает значение, которое имеет тип.
- Функция вызывается ссылкой на ее имя.
- Имя подпрограммы не имеет значения.
- Подпрограмма вызывается оператором `CALL`.
- Фиктивные аргументы используются в описании подпрограммы, фактические аргументы – при ее вызове.

- Компилятор предоставляет явные интерфейсы для встроенных, внутренних и модульных подпрограмм.
- Оператора EXTERNAL в вызывающей программе достаточно для связи с внешней подпрограммой в обычных обстоятельствах.
- Интерфейсный блок всегда может использоваться для предоставления явного интерфейса внешней подпрограмме; бывают ситуации, когда его необходимо использовать.
- Модуль может содержать объявления, операторы спецификаций и подпрограммы.
- Объекты в модуле, указанном с помощью атрибута PRIVATE, могут быть доступны только внутри модуля. Сущности по умолчанию общедоступны.
- Областью действия имени или метки является набор строк, в которых они могут использоваться однозначно. Правила области действия для меток и имен отличаются.
- Фактические аргументы могут передаваться подпрограммам по ссылке или по значению. Аргумент, переданный по ссылке, может быть изменен при возврате. Константы или выражения передаются по значению. Круглые скобки вокруг имени переменной делают его выражением.
- Фиктивные аргументы могут быть указаны с атрибутом INTENT: IN (не могут быть изменены в подпрограмме), OUT (фактический аргумент должен быть переменной, чтобы его можно было изменить) или INOUT (фактический аргумент должен быть переменной). Все фиктивные аргументы должны иметь намерение.
- Фиктивные аргументы функции должны иметь намерение IN.
- Фиктивные аргументы могут быть указаны OPTIONAL. Требуемые аргументы могут быть предоставлены в списке аргументов ключевого слова. Внешние подпрограммы с необязательными аргументами должны иметь явный интерфейс, предоставляемый телом интерфейса.
- Интерфейс может использоваться для перегрузки определенных имен подпрограмм общим именем.
- При разработке больших программ следует использовать макеты (пустые подпрограммы).
- Подпрограммы могут вызывать сами себя напрямую, если они объявлены как RECURSIVE. Рекурсивная функция должна

иметь предложение RESULT в своем заголовке, чтобы возвращать свое значение.

**Упражнения**

8.1 Напишите программу, использующую отношение Ньютона

$$\frac{f(x+h)-f(x)}{h}$$

для оценки первой производной $f(x) = x^3$ при $x = 1$, используя последовательно меньшие значения $h$: 1, $10^{-1}$, $10^{-2}$ и т. д. Используйте функциональную подпрограмму для $f(x)$ .

8.2 Первые три полинома Лежандра — это $P_0(x) = 1$, $P_1(x) = x$ и $P_2(x) = (3x^2–1)/2$. Существует общая рекуррентная формула для полиномов Лежандра, по которой они определяются рекурсивно:

$$(n + 1)P_{n+1}(x) – (2n +1)xP_n(x) + NP_{n–1}(x) = 0$$

Задайте рекурсивную функцию Фортрана $P(N, X)$ для генерации полиномов Лежандра, учитывая форму $P_0$ и $P_1$. Используйте вашу функцию для вычисления $P(2, X)$ для нескольких значений $X$ и сравните свои результаты с результатами, полученными с использованием аналитической формы $P_2(x)$, приведенной выше.

# Глава 9 Массивы

В реальных программах нам часто нужно одинаково обрабатывать большие объемы данных, например. чтобы найти среднее значение набора чисел, или отсортировать список чисел или имен, или проанализировать набор результатов тестов учащихся, или решить систему линейных уравнений. Чтобы избежать чрезвычайно неуклюжей программы, где могут потребоваться сотни имен переменных, мы можем использовать индексированные переменные или массивы. Их можно рассматривать как переменные с компонентами, скорее как векторы или матрицы. Они записываются обычным образом, за исключением того, что нижние индексы заключаются в круглые скобки после имени переменной, например `X(3)`, `Y(I + 2 * N)`.

Fortran 90 обладает чрезвычайно мощным набором функций массивов.

## 9.1 Среднее и стандартное отклонение

Чтобы проиллюстрировать основные принципы, давайте вычислим выборочное среднее и стандартное отклонение набора из $N$ наблюдений. Среднее значение определяется как

$$\overline{X} = \frac{1}{N}\sum_{i=1}^{N} X_i \qquad (9.1)$$

где $X_i$ – $i$ наблюдение. Стандартное отклонение $s$ определяется как

$$s^2 = \frac{1}{N-1}\left(X_i - \overline{X}\right)^2 \qquad (9.2)$$

Следующая программа вычисляет эти две величины из данных, считанных из дискового файла DATA. Первый элемент данных в файле — это значение $N$. За ним следуют ровно $N$ наблюдений — все в отдельных строках.

```
IMPLICIT NONE
INTEGER              :: I, N
REAL                 :: Std = 0
REAL, DIMENSION(100) :: X
REAL                 :: XBar = 0
OPEN (1, FILE = 'DATA')
```

```
READ (1, *) N
DO I = 1, N
  READ (1, *) X(I)
  XBar = Xbar + X(I)
END DO

XBar = XBar / N

DO I = 1, N
  Std = Std + (X(I) - XBar) ** 2
END DO

Std = SQRT( Std / (N - 1) )
PRINT*, 'Mean: ', XBar
PRINT*, 'Std deviation: ', Std
```

Попробуйте это с некоторыми примерами данных (каждое число в отдельной строке):

10 5.1 6.2 5.7 3.5 9.9 1.2 7.6 5.3 8.7 4.4

Вы должны получить среднее значение 5,76 и стандартное отклонение 2,53 (с точностью до двух знаков после запятой). Атрибут `DIMENSION(100)` в операторе объявления типа для массива *X* выделяет 100 ячеек памяти с именами `X(1)`, `X(2)`, ..., `X(100)`. Однако пример данных выше состоит только из 10 чисел, поэтому используются только первые 10 расположений. Обратите внимание, что значение *N* должно быть прочитано первым (и должно быть правильным), прежде чем значения *N* могут быть прочитаны.

После завершения `READ` область памяти, в которой хранится массив, выглядит так:

| | | | | |
|---|---|---|---|---|
| *X*(1) | *X*(2) | *X*(3) | ... | *X*(10) |
| 5.1 | 6.2 | 5.7 | ... | 4.4 |

Теперь, когда данные надежно сохранены в массиве, их можно использовать снова, просто сославшись на имя массива *X* с номером элемента, например *X*(3). Таким образом, сумма первых двух элементов может быть вычислена как

```
SUM = X(1) + X(2)
```

Это средство необходимо для вычисления *s* в соответствии с приведенной выше формулой — все данные должны быть прочитаны для вычисления среднего значения, а среднее значение должно быть вычислено до того, как все данные будут повторно использованы для вычисления *s*.

Честно говоря, есть еще один способ вычисления стандартного отклонения, который не требует использования массива:

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N} X_i^2 - N\overline{X}^2 \qquad (9.3)$$

В качестве упражнения перепишите программу без массива, считывая все данные в одну переменную *X*.

Размещать каждое значение данных на отдельной строке в файле довольно громоздко — это требуется для отдельного выполнения каждого `READ(1, *) X(I)` в цикле `DO`. Фортран позволяет использовать подразумеваемый (скрытый) `DO` для чтения или печати всего массива или его части. Просто замените всю конструкцию `DO` в программе на

```
READ (1, *) ( X(I), I = 1, N )
```

Давайте отметим, что синтаксис требует круглых скобок вокруг подразумеваемого DO.

## 9.2 Основные правила и обозначения

*Массив* – это наш первый пример составного объекта в Fortran 90, то есть объекта, который может иметь более одного значения. Массивы могут быть довольно сложными образованиями. Здесь упоминаются только основы; более продвинутые функции будут представлены позже.

Оператор

```
REAL, DIMENSION(10) :: X
```

объявляет *X* массивом (или списком) с 10 действительными элементами, обозначаемыми *X*(1), *X*(2), ..., *X*(10). Количество элементов в массиве называется его размером (в данном случае 10).

Каждый элемент массива является скаляром (однозначным). На элементы массива ссылаются с помощью нижнего индекса, указанного в круглых скобках после имени массива. Нижний индекс должен быть целочисленным выражением — его значение должно находиться в пределах диапазона, определенного в объявлении массива. Таким образом `X(I+1)` является действительной ссылкой на элемент *X*, как указано выше, если `(I+1)` находится в диапазоне 1–10. Ошибка компилятора возникает, если нижний индекс выходит за допустимые пределы.

По умолчанию массивы имеют нижнюю границу 1 (наименьшее значение, которое может принимать индекс). Однако вы можете иметь любую нижнюю границу, которая вам нравится:

```
REAL, DIMENSION(0:100) :: A
```

объявляет, что *A* имеет 101 элемент, от *A*(0) до *A*(100). Должна быть указана верхняя граница; если нижняя граница отсутствует, по умолчанию она равна 1.

Массив может иметь более одного измерения. Границы каждого измерения должны быть указаны:

```
REAL, DIMENSION(2,3) :: A
```

`A` – двумерный массив. Количество элементов в измерении называется размером в этом измерении. `A` имеет размер 2 в первом измерении и размер 3 во втором измерении (и общий размер 6). Фортран допускает до семи измерений. Количество измерений массива — это его ранг, а последовательность размеров – его форма. Форма *A*(2, 3) или (2×3) в матричных обозначениях. Скаляр считается имеющим нулевой ранг. В этой главе мы сосредоточимся в основном на одномерных массивах. Двумерные массивы более уместно обсуждать в контексте матриц.

Атрибут `DIMENSION` является необязательным. Он предоставляет необязательную форму `DIMENSION` по умолчанию для переменных, за именами которых не следует форма:

```
REAL, DIMENSION(5) :: A,B(2,3),C(10) ! only A is (1:5)
INTEGER I(10), K(4,4), L(5)          ! all different
```

Индекс массива можно использовать в качестве управляющей переменной в цикле DO для генерации элементов массива. Следующий код присваивает первые пять четных целых чисел десяти элементам *X* (предполагается, что они объявлены правильно):

```
DO I = 1, 5
  X(I) = 2 * I
END DO
```

Тот же эффект может быть достигнут несколькими различными способами с помощью конструктора массива:

```
X = (/ 2, 4, 6, 8, 10 /)
```

или

```
X = (/ (I, I = 2, 10, 2) /)
```

Константный массив может быть объявлен таким образом с атрибутом `PARAMETER`. Весь массив может быть прочитан:

```
READ*, X
```

Разумеется, должно быть указано точное количество значений данных. Целому массиву может быть присвоено скалярное значение:

```
X = 1
```

Это частный случай присваивания массива, с которым мы еще встретимся позже.

*Чтение неизвестного объема данных*

Подразумеваемый (скрытый) DO вместе со спецификатором IOSTAT в READ обеспечивает аккуратный способ чтения неизвестного количества данных в массив, где указан только максимальный размер массива:

```
INTEGER, PARAMETER   :: MAX = 100
REAL, DIMENSION(MAX) :: X

OPEN (1, FILE = 'DATA')
READ( 1, *, IOSTAT = IO ) ( X(I), I = 1, MAX )

IF (IO < 0) THEN
  N = I - 1
ELSE
  N = MAX
END IF

PRINT*, ( X(I), I = 1, N )
```

Данные могут быть расположены в любом формате во входном файле. Обратите внимание, что I на единицу больше числа прочитанных значений: оно увеличивается в подразумеваемом DO до того, как будет обнаружено условие конца файла. Обратите также внимание, что при нормальном выходе из подразумеваемого DO его значение будет MAX+1.

## 9.3 Массивы как аргументы подпрограммы

Массив может быть передан в качестве аргумента подпрограмме несколькими способами. Самый аккуратный способ показан в следующей программе, где вычисление среднего и стандартного отклонения отнесено к подпрограмме Stats:

```
IMPLICIT NONE

INTEGER              :: I, N
REAL, DIMENSION(100) :: X
REAL                 :: Std = 0
```

```
REAL                  :: XBar = 0

OPEN (1, FILE = 'DATA')
READ (1, *) N, (X(I), I = 1, N)
CALL Stats( X, N, XBar, Std )
PRINT*, 'Mean:          ', XBar
PRINT*, 'Std deviation: ', Std

CONTAINS
  SUBROUTINE Stats( Y, N, YBar, S )
    REAL, DIMENSION(:), INTENT(IN) :: Y
    REAL, INTENT(INOUT)            :: S, YBar
    INTEGER, INTENT(IN)            :: N

    INTEGER  I

    YBar = 0; S = 0

    DO I = 1, N
      YBar = Ybar + Y(I)
    END DO

    YBar = YBar / N

    DO I = 1, N
      S = S + (Y(I) - YBar) ** 2
    END DO

    S = SQRT( S / (N - 1) )
  END SUBROUTINE
END
```

Компилятор требует, чтобы формы фактических и фиктивных аргументов совпадали. Описание

```
REAL, DIMENSION(:), INTENT(IN) :: Y
```

фиктивного аргумента делает его предполагаемой формой, т.е. принимает любую форму, заданную фактическим аргументом. Полный синтаксис для измерения в этом случае:

[*lower bound*]:

Нижняя граница по умолчанию равна 1, если она не указана. Обратите внимание, что форма передается, а не ограничивается. Итак, в этом примере

```
REAL, DIMENSION(0:), INTENT(IN) :: Y
```

приведет к `X(I)`, соответствующему `Y(I-1)`. Если нижние границы в объявлении фактического и фиктивного аргументов совпадают, соответствие будет точным. Обратите внимание, что если `Stats`

скомпилирована отдельно как внешняя подпрограмма, в вызывающей программе должен быть предоставлен явный интерфейс, например.

```
INTERFACE
  SUBROUTINE Stats( Y, N, YBar, S )
    INTEGER, INTENT(IN)            :: N
    REAL, DIMENSION(:), INTENT(IN) :: Y
    REAL, INTENT(INOUT)            :: S, YBar
  END SUBROUTINE
END INTERFACE
```

## 9.4 Распределяемые (динамические) массивы

Программа в разделе 9.2 считывает неизвестное количество данных в массив. Однако должен быть объявлен максимальный размер массива. В программах, которые требуют много памяти, это может быть расточительно. Более эффективным решением, невозможным в более ранних версиях Фортрана, является использование динамической памяти. Все типы переменных, которые мы видели до сих пор, были статическими переменными, хотя это никогда не упоминалось. Это означает, что когда переменная объявляется, компилятор присваивает ей определенный адрес в памяти (с фиксированным объемом памяти), и она остается там, пока работает программа. Напротив, куски динамической памяти используются только при необходимости, пока выполняется программирование, а затем отбрасываются. Зачастую это более эффективный способ использования памяти. Переменная указывается как динамическая с атрибутом ALLOCATABLE. В частности, одномерный массив может быть указан следующим образом:

```
REAL, DIMENSION(:), ALLOCATABLE :: X
```

Хотя его ранг указан, он не имеет размера до тех пор, пока не появится в операторе ALLOCATE, таком как

```
ALLOCATE( X(N) )
```

Когда он больше не нужен, его можно освободить:

```
DEALLOCATE( X )
```

тем самым освобождая используемую память. В следующем фрагменте программы показано, как использовать выделяемые массивы, как называются эти звери, для чтения неизвестного количества данных, которые, к сожалению, должны предоставляться по одному элементу в строке из-за того, как работает READ.

```
REAL, DIMENSION(:), ALLOCATABLE :: X, OldX
```

```
REAL      A
INTEGER   IO, N
ALLOCATE( X(0) )           ! size zero to start with?
N = 0
OPEN( 1, FILE = 'DATA' )
DO
  READ(1, *, IOSTAT = IO) A
  IF (IO < 0 ) EXIT
    N = N + 1
    ALLOCATE( OldX( SIZE(X) ) )
    OldX = X           ! entire array can be assigned
    DEALLOCATE( X )
    ALLOCATE( X(N) )
    X = OldX
    X(N) = A
    DEALLOCATE( OldX )
END DO
PRINT*, ( X(I), I = 1, N )
...
```

Хотелось бы иметь возможность увеличивать размер X для каждого прочитанного значения. Однако, прежде чем X может быть выделен с большим размером, он должен быть освобожден, что приведет к потере всех предыдущих считанных данных. Таким образом, другой динамический массив, OldX, должен быть использован, чтобы позаботиться об этом.

Обратите внимание на следующие важные особенности:

- массив может иметь нулевой размер — часто это удобно;
- могут быть назначены целые массивы;
- массив, выделенный в данный момент, не может быть выделен повторно;
- освобождаемый массив должен быть выделен в данный момент.

У вас может возникнуть соблазн написать это как подпрограмму. Однако макет, не являющийся ALLOCATABLE, может не иметь атрибута ALLOCATABLE.

## 9.5 Сортировка списка

Пузырьковая сортировка

Одним из стандартных применений массивов является сортировка списка чисел, скажем, в порядке возрастания. Основная идея заключается в том, что несортированный список считывается в массив.

Затем числа упорядочиваются с помощью процесса, который, по существу, проходит по списку много раз, меняя местами последовательные элементы в неправильном порядке, пока все элементы не окажутся в правильном порядке. Такой процесс называется сортировкой пузырьком, потому что меньшие числа поднимаются в начало списка, как пузырьки воздуха в воде. (На самом деле, в версии, показанной ниже, наибольшее число «опустится» в конец списка после первого прохода, что действительно делает его сортировкой «пузырьковой».) Существует много других методов сортировки, которые могут можно найти в большинстве учебников по информатике (один из них, `Quick Sort`, приведен в следующем разделе). Как правило, они более эффективны, чем пузырьковая сортировка, но их преимущество в том, что это самый простой метод для программирования. Структурный план пузырьковой сортировки выглядит следующим образом:

1. Инициализировать $N$ (длина списка)
2. Прочитайте список $X$
3. Повторить $N$–1 раз со счетчиком $K$:
   Повторить $N$–$K$ раз со счетчиком $J$:
   Если $X_j > X_{j+1}$, то поменять местами содержимое $X_j$ и $X_{j+1}$
4. Распечатайте список $X$, который теперь отсортирован.

В качестве примера рассмотрим список из пяти чисел: 27, 13, 9, 5 и 3. Сначала они считываются в массив $X$. Часть памяти компьютера для этой задачи представлена в табл. 9.1. В каждом столбце отображается список во время каждого прохода. Штрих в строке указывает на изменение этой переменной во время прохода, когда программа работает вниз по списку. Количество тестов, выполненных на каждом проходе, также показано в таблице. Работайте с таблицей вручную вместе со структурным планом, пока не поймете, как работает алгоритм.

Таблица 9.1: Память во время пузырьковой сортировки

| | **1st pass** | **2nd pass** | **3rd pass** | **4th pass** |
|---|---|---|---|---|
| `X(1):` | 27/13 | 13/9 | 9/5 | 5/3 |
| `X(2):` | 13/27/9 | 9/13/5 | 5/9/3 | 3/5 |
| `X(3):` | 9/27/5 | 5/13/3 | 3/9 | 9 |
| `X(4):` | 5/27/3 | 3/13 | 13 | 13 |
| `X(5):` | 3/27 | 27 | 27 | 27 |
| | 4 tests | 3 tests | 2 tests | 1 test |

Алгоритмы сортировки сравниваются путем подсчета количества выполняемых ими тестов, так как это занимает большую часть времени выполнения при сортировке. На K-м проходе пузырьковой сортировки имеется ровно $N–K$ тестов, поэтому общее количество тестов равно [1 + 2 + 3 + точки + ($N$–1) = $N$($N$–1)/2] (приблизительно $N^2$/ 2 для больших $N$). Таким образом, для списка из пяти чисел требуется 10 тестов, а для 10 чисел — 45 тестов. Необходимое компьютерное время увеличивается пропорционально квадрату длины списка.

В приведенной ниже программе используется подпрограмма `Bubble_Sort` для сортировки 100 случайных чисел. Он немного отличается от плана структуры выше, что сделает $N$–1 проходов, даже если список отсортирован до последнего прохода. Поскольку большинство реальных списков частично отсортированы, имеет смысл после каждого прохода проверять, не были ли сделаны какие-либо свопы. Если их нет, список необходимо отсортировать, чтобы можно было исключить ненужные (и, следовательно, тратящие время) тесты. В подпрограмме логическая переменная `Sorted` используется для определения момента сортировки списка, а вместо этого внешний цикл кодируется как недетерминированный цикл `DO WHILE`.

```
IMPLICIT NONE
INTEGER, PARAMETER  :: N = 100 ! size of list
REAL, DIMENSION(N)  :: List    ! list to be sorted
INTEGER I                      ! counter
REAL R                         ! random number

DO I = 1, N
  CALL Random_Number( R )
  List(I) = INT(N*R + 1) !random integers in range 1-N
END DO

PRINT 10, List                 ! print unsorted list
10  FORMAT( 13F6.0 )

CALL Bubble_Sort( List )       ! sort

PRINT 10, List                 ! print sorted list

CONTAINS
  SUBROUTINE Bubble_Sort( X )
    REAL, DIMENSION(:),INTENT(INOUT) :: X ! list
    INTEGER :: Num                     ! size of list
    REAL Temp                          ! temp for swop
    INTEGER J, K         ! counters
    LOGICAL Sorted       ! flag to detect when sorted

    Num = SIZE(X)
```

```
      Sorted = .FALSE.          ! initially unsorted
      K = 0                     ! count the passes

      DO WHILE (.NOT. Sorted)
        Sorted = .TRUE.         ! they could be sorted
        K = K + 1           ! another pass
        DO J = 1, Num-K     ! fewer tests on each pass
          IF (X(J) > X(J+1)) THEN   ! are they in order?
            Temp = X(J)             ! no ...
            X(J) = X(J+1)
            X(J+1) = Temp
            Sorted = .FALSE.        ! a swop was made
          END IF
        END DO
      END DO

    END SUBROUTINE
  END
```

`Bubble_Sort` записывается здесь как внутренняя подпрограмма, но будет выполняться как внешняя подпрограмма или в модуле `eject`.

### Быстрая сортировка

Попробуйте отсортировать 1000 чисел с помощью пузырьковой сортировки. Это занимает довольно много времени. Сортировка 10 000 чисел (не немыслимая проблема) заняла бы примерно в 100 раз больше времени. Знаменитый алгоритм быстрой сортировки, изобретенный C.A.R. Хоар в 1960 году [11] намного быстрее. Он основан на подходе «разделяй и властвуй»: чтобы решить большую проблему, разбейте ее на более мелкие подзадачи и разбейте каждую подзадачу таким же образом, пока они не станут достаточно маленькими для решения. Как кто-то заметил: «В каждой проблеме есть меньшая проблема внутри, ожидающая выхода наружу». Как разбить нашу задачу сортировки на управляемые подзадачи? Что ж, взгляните на следующий список:

| *Number:* | 19 | 30 | 14 | 28 | 8 | 32 | 72 | 41 | 87 | 33 |
|---|---|---|---|---|---|---|---|---|---|---|
| *Postion:* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Значение 32 в позиции 6 имеет особое свойство. Все значения слева от него меньше 32, а все значения справа от него больше 32. Говорят, что значение 32 разделяет задачу сортировки на две подзадачи: левую подзадачу и правую подзадачу. Каждая из них может быть отсортирована отдельно, потому что никакое значение в левой подзадаче никогда не может попасть в правую подзадачу, и наоборот. Кроме того, значение 32 находится в правильной позиции в правой

подзадаче — там это наименьшее значение. Вы можете подумать, что это был счастливый случай, когда 32 с самого начала аккуратно разбили список. Прелесть алгоритма в том, что для любого списка мы всегда можем создать раздел с самым левым значением без особых трудностей.

Посмотрите теперь на перестановку списка:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 32 | 19 | 41 | 14 | 28 | 8 | 72 | 30 | 87 | 33 |
| $L$ | $L_1$ | | | | | | | $R_1$ | $R$ |

Крайние концы помечены $L$ и $R$. Мы собираемся разбить список со значением 32, в настоящее время в позиции $L$. Мы определяем счетчики $L_1$ и $R_1$ с начальными значениями, как показано. Теперь идея состоит в том, чтобы переместить $L_1$ вправо, убедившись, что

- каждое значение слева от позиции $L_1$ *leq* значение раздела
- а затем переместить $R_1$ влево, при этом убедившись, что
- каждое значение справа от позиции $R_1$ > значения раздела.

Это приводит нас к такой ситуации:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 32 | 19 | 41 | 14 | 28 | 8 | 72 | 30 | 87 | 33 |
| $L$ | | $L_1$ | | | | | $R_1$ | | $R$ |

Что теперь? Кажется, наступил тупик. Но нет, просто поменяйте местами значение в позиции $L_1$ (41) со значением в позиции $R_1$ (30):

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 32 | 19 | **30** | 14 | 28 | 8 | 72 | **41** | 87 | 33 |
| $L$ | | $L_1$ | | | | | $R_1$ | | $R$ |

Теперь мы можем продолжить перемещение $L_1$ и $R_1$ в соответствии с указанными выше правилами, пока не доберемся до этой сцены:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 32 | 19 | 30 | 14 | 28 | 8 | 72 | 41 | 87 | 33 |
| $L$ | | | | | $R_1$ | $L_1$ | | | $R$ |

Однако сейчас ситуация иная. $L_1$ и $R_1$ пересеклись, значит, мы должны найти точку разбиения: она находится в позиции $R_1$. Все, что осталось сделать, это поменять местами значения в $L$ (32) и $R_1$ (8), что даст нам секционированный массив с 32 в качестве раздела:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **8** | 19 | 30 | 14 | 28 | 32 | 72 | 41 | 87 | 33 |
| $L$ | | | | | $R_1$ | $L_1$ | | | $R$ |

Теперь мы можем разделить любую задачу с ее крайним левым значением. Таким образом, полученные подзадачи могут быть

разделены таким же образом. Мы просто продолжаем разбивать подзадачи до тех пор, пока в подзадачах не будет только 1 элемент, который необходимо отсортировать!

Для этого типа алгоритма «разделяй и властвуй» была создана рекурсия. Следующая программа реализует это рекурсивно.

```
IMPLICIT NONE
INTEGER, PARAMETER  :: N = 100   ! size of list
REAL, DIMENSION(N)  :: List      ! list to be sorted
INTEGER I                        ! counter
REAL R                           ! random number

DO I = 1, N
  CALL RANDOM_NUMBER( R )
  List(I) = INT(N*R + 1) !random integers in range 1-N
END DO

PRINT 10, List              ! print unsorted list
10  FORMAT( 13F6.0 )

CALL Quick_Sort( List, 1, N )    ! quick sort now

PRINT 10, List                   ! print sorted list

CONTAINS
  RECURSIVE SUBROUTINE Quick_Sort( X, L, R )
    REAL, DIMENSION(:), INTENT(INOUT) :: X    ! list
    INTEGER, INTENT(IN)   :: L,R ! left, right bounds
    INTEGER L1, R1                            ! etc

    IF (L < R)THEN
      L1 = L
      R1 = R

      DO
       DO WHILE(L1< R.and.X(L1) <= X(L)) !shift L1 right
          L1 = L1 + 1
       END DO

       DO WHILE(L< R1.and.X(R1) >= X(L)) !shift R1 left
          R1 = R1 - 1
        END DO

        IF (L1 < R1) CALL Swop( X(L1), X(R1) ) ! swop
        IF (L1 >= R1) EXIT   ! crossover - partition
      END DO

     CALL Swop(X(L),X(R1)) ! partition with X(L) at R1
    CALL Quick_Sort(X,L,R1-1) !now attack left subproblem
    CALL Quick_Sort(X,R1+1,R)  !on't forget right subproblem
    END IF
```

```
  END SUBROUTINE Quick_Sort
  SUBROUTINE Swop( A, B )
    REAL, INTENT(INOUT) :: A, B
    REAL                :: Temp
    Temp = A
    A = B
    B = Temp
  END SUBROUTINE Swop
END
```

Обратите внимание, что подкачка реализована как подпрограмма. Если вы перепишете `Quick_Sort` как внешнюю подпрограмму, `Swop` может оказаться внутри нее. Вам следует попробовать поработать с программой вручную с образцом массива на рисунках. Попробуйте быструю сортировку на 1000 номеров. Вы должны быть впечатлены! Было доказано, что для быстрой сортировки требуется примерно $N\log_2 N$ сравнений, в отличие от сортировки пузырьком $N^2/2$. Возможно, вам будет интересно узнать, что быстрая сортировка сильно замедляется, если список уже почти отсортирован (попробуйте применить ее к отсортированному списку). Однако в этом случае он будет работать так же быстро, если вы выберете значение рядом с серединой подзадачи для значения раздела, а не самое левое значение. Удачной сортировки!

## 9.6 Дополнительные функции массива

Fortran 90 имеет несколько новых мощных функций массивов, которые идеально подходят для приложений численного анализа. Они обобщены в этом разделе.

### Конструкторы массивов

Одномерный постоянный массив может быть создан из списка значений элементов, заключенных между разделителями (/ и /). Например,

```
(/ 2, 4, 6, 8, 10 /)
```

представляет собой массив первого ранга из пяти элементов.

Общая форма конструктора массива:

(/список значений конструктора массива/)

где каждое значение является либо выражением, либо подразумеваемым DO формы

(список значений, переменная = expr1, expr2 [,expr3] )

Параметры подразумеваемого DO работают так же, как и параметры DO. Например.

```
(/ 2, 4, (I, I = 4, 10, 2) /)
```

такой же как

```
(/ 2, 4, 6, 8, 10 /)
```

Необязательный параметр expr3 иногда называют шагом в контексте подразумеваемого DO. Подразумеваемый DO может быть вложен в другой, что делает

```
(/ ((I, I = 1, 2), J = 1, 3) /)
```

такой же как

```
(/ 1, 2, 1, 2, 1, 2 /)
```

Если список пуст, создается массив нулевого размера. Область действия подразумеваемой переменной DO ограничена подразумеваемым DO — она не повлияет на значение другой переменной с тем же именем где-либо еще в единице области видимости конструктора.

Константный массив с рангом больше единицы может быть создан из конструктора массива с помощью встроенной функции RESHAPE. Например.

```
RESHAPE( SOURCE = (/1,2,3,4,5,6/), SHAPE = (/2,3/) )
```

образует матрицу (2 на 3)

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Разделы массива

На подмассив, называемый разделом, можно ссылаться, указав диапазон индексов, например.

```
X(I:J)     ! массив 1 ранга размером J-I+1
Y(I, 1:J) ! массив 1 ранга размером J (например, I-я строка
             ! матрицы с J столбцами)
X(2:5, 8:9) ! массив 2 ранга размером 4×2
```

Раздел массива технически является массивом и может появляться в операторах, где массив уместен, хотя на его отдельные элементы нельзя ссылаться напрямую. Поэтому мы не можем написать `/X( I:J )(2)/` для ссылки на второй элемент секции `X(I:J)`. Лучше напишите `X(I+1)` (поскольку первый элемент, естественно, `X(I)`). Одна или обе границы в разделе могут быть опущены, и может использоваться шаг, отличный от 1:

```
A(J, :)       ! все элементы строки J
A(J, 1:K:3 ) ! элементы 1, 4, 7, ... строки J
```

Нижний индекс раздела может быть даже одномерным массивом целочисленного типа. Например, кодирование

```
REAL, DIMENSION(10) :: A = (/ (I, I = 2, 20, 2) /)
INTEGER, DIMENSION(5) :: B = (/ 5, 4, 3, 2, 1 /)
PRINT*, A(B)
```

производит вывод

```
10.00 8.00 6.00 4.00 2.00
```

Нижний индекс такого типа называется индексом вектора. Элементы индекса вектора могут быть в любом порядке. Например,

```
A( (/ 3, 5, 1, 2 /) )
```

является разделом *A* с элементами *A*(3), *A*(5), *A*(1) и *A*(2) в указанном порядке. Некоторые значения в нижнем индексе вектора могут повторяться. Это называется разделом «многие-один», так как более одного элемента раздела отображается на один элемент массива. Например,

```
A( (/ 3, 5, 1, 5 /) )
```

является разделом, в котором элементы 2 и 4 ссылаются на A(5). Раздел «многие-один» не может отображаться слева от оператора присваивания. Если раздел массива с индексом вектора является фактическим аргументом подпрограммы, он считается выражением и не может быть изменен подпрограммой. Поэтому он может не иметь намерения `OUT` или `INOUT`.

### *Инициализация массивов с помощью* **DATA**

Весь или часть массива может быть инициализирована в операторе `DATA`. Существует ряд возможностей, например.

```
REAL, DIMENSION(10) :: A, B, C(3,3)
```

```
DATA A / 5*0, 5*1 /            ! первые 5 – 0,
последние 5 – 1
DATA B(1:5) / 4, 0, 5, 2, -1 / ! только часть 1:5
DATA ((C(I,J), J= 1,3), I=1,3) / 3*0, 3*1, 3*2 / !
по строкам
```

### Выражения массива

Встроенный оператор может работать как с массивом, так и со скаляром, чтобы создать выражение массива. Когда унарный встроенный оператор действует на массив, он действует на каждый элемент массива. Например, -*X* меняет знак каждого элемента массива *X*. Когда бинарная встроенная операция применяется к паре массивов одинаковой формы (одинакового ранга и размеров), операция применяется к соответствующим элементам операндов. Результатом операции является массив той же формы. Один из операндов может быть скаляром, и в этом случае он используется в операции над каждым элементом операнда массива (считается, что скаляр был «транслирован» в массив той же формы, что и операнд массива). Например, при объявлении

```
REAL, DIMENSION(10) :: X, Y
```

ниже приведены примеры выражений массива:

```
X + Y          ! результат имеет элементы X(I) + Y(I)
X * Y          ! результат имеет элементы X(I) * Y(I)
X * 3          ! результат имеет элементы X(I) * 3
X * SQRT( Y ) ! результат имеет элементы X(I) *
SQRT(Y(I))
X == Y         ! результат имеет элементы .TRUE.
               ! если X(I) == Y(I), и .FALSE. иначе
```

Следует отметить, что когда массив является аргументом элементарной функции, функция работает с каждым элементом массива. Два массива одинаковой формы *подобны*. Скаляр совместим с любым массивом. Так же обратите внимание на то, что бинарные операции применяются к соответствующим позициям экстента, а не к соответствующим нижним индексам. Например.

```
X(2:5) + Y(4:7)
```

имеет значения элементов

```
X(I) + Y(I+2), I = 2, 3, 4, 5.
```

Назначение массива

Выражение массива (включая скалярное выражение) может быть присвоено переменной массива той же формы. Каждый элемент выражения присваивается соответствующему элементу целевого массива. Опять же, соответствие осуществляется по положению внутри экстента, а не по значению нижнего индекса. Например,

```
REAL, DIMENSION(10)  :: X, Y
REAL, DIMENSION(5,5) :: A, C
X = Y            ! оба с рангом 1 с одинаковым размером
Y = 0            ! Y весь нули
X = 1 / X       ! заменить каждый элемент X его обратным
X = COS( X )   ! замените каждый элемент X его косинусом
X(1:5) = Y(4:8)   ! у обоих ранг 1 с размером 5
A(I,1:J) = C(K,1:J) !строка K матрицы C, присвоенная
                    !строке I матрицы A
```

Эти средства чрезвычайно полезны в численных процедурах, таких как редукция Гаусса. Если выражение справа от присваивания массива ссылается на часть массива слева, выражение полностью вычисляется до начала присваивания. Как, например,

```
X(1:3) = X(2:4)
```

приводит к тому, что каждый элемент `X(I), I = 1, 2, 3`, имеет значение, которое `X(I+1)` имело до начала присваивания.

*Конструкция* **WHERE**

Конструкция `WHERE` может использоваться для выполнения операции только с определенными элементами массива, например.

```
WHERE (A > 0)
  A = LOG( A ) !журнал всех положительных элементов ELSEWHERE
  A = 0    !все неположительные элементы равны нулю END WHERE
```

Оператор `ELSEWHERE` является необязательным. Конструкция аналогична `IF-THEN-ELSE`. Выражение в круглых скобках после ключевого слова `WHERE` является выражением логического массива и может быть просто логическим массивом. Иногда его называют маской. Существует соответствующий оператор `WHERE`:

```
WHERE (A /= 0) A = 1 / A !
```

заменить ненулевые элементы на обратные

*Функции со значением массива*

Функция может иметь значение массива. Если это внешняя функция, ей нужен интерфейсный блок.

*Массив, обрабатывающий встроенные функции*

Существует ряд встроенных функций, которые относятся конкретно к массивам.

`ALL(X)` возвращает значение `.TRUE.` только если все элементы логического массива `X` истинны.
`ANY(X)` возвращает значение `.TRUE.` если какой-либо элемент логического массива `X` истинен. В противном случае возвращается `.FALSE.`
`SUM(X)` и `PRODUCT(X)` возвращают сумму и произведение элементов целочисленного, действительного или комплексного массива `X` соответственно.

Во всех этих случаях `X` может быть выражением массива:

```
INTEGER, DIMENSION(5,5) :: A
REAL X(3), Y(3)
...
IF(ANY(A > 0)) A=1 !если какой-то элемент > 0 заменить все на 1
IF(ALL(A == 0)) A=-1 !если все элементы = 0, заменить все на -1
Dot = SUM( X * Y )    ! скалярное произведение X и Y
```

**Резюме**

- Массивы удобны для представления и обработки больших объемов данных.
- Массив представляет собой набор индексированных переменных с одинаковыми именами.
- Члены массива называются элементами.
- Количество элементов в массиве равно его размеру. Размер может быть нулевым.
- Верхняя и нижняя границы индексов массива задаются с помощью атрибута `DIMENSION` в операторе спецификации типа массива.
- Нижний индекс массива не может выходить за пределы, указанные параметром `DIMENSION`.
- Нижний индекс массива может быть любым допустимым числовым выражением (при необходимости округленным).

- Количество измерений массива является его рангом. Массив может иметь до семи измерений. Скаляр имеет нулевой ранг.
- Количество элементов в измерении массива является экстентом измерения.
- Последовательность экстентов массива — это его форма.
- Массив может быть передан как фактический аргумент подпрограмме. Фиктивный аргумент должен иметь ту же форму. Если соответствующий фиктивный аргумент является массивом предполагаемой формы, он примет форму фактического аргумента.
- Динамическая переменная (которая может быть массивом) указывается с помощью атрибута `ALLOCATABLE`, и ей может быть выделена память во время работы программы. Память может быть освобождена позже.
- Фиктивный аргумент не может быть размещен.
- Константа массива первого ранга может быть сформирована с помощью конструктора массива.
- Подразумеваемый `DO` может использоваться в конструкторе массива.
- Раздел массива является подмассивом.
- Нижний индекс раздела, заданный целочисленным выражением первого ранга, является индексом вектора.
- Массивы одинаковой формы совместимы.
- Сечение соответствует любому массиву такой же формы.
- Скаляр совместим с любым массивом.
- Выражения массива могут быть сформированы из соответствующих массивов.
- Выражения массива могут быть присвоены соответствующим массивам.
- Когда элементарная встроенная функция принимает аргумент массива, функция применяется к каждому элементу массива.
- Конструкция `WHERE` управляет операциями над элементами массива по логической маске.

**Упражнения**

9.1 Если `Num` – целочисленный массив с атрибутом `DIMENSION(100)`, напишите строки кода, которые поместят первые 100 положительных целых чисел (1, 2, ..., 100) в элементы `Num(1)`, ...,

`Num(100)`; поместить первые 50 положительных целых чисел (2, ..., 100) в элементы `Num(1)`, ..., `Num(50)`;

9.2 Назначьте целые числа в обратном порядке, т.е. назначьте 100 для `Num(1)`, 99 для `Num(2)` и т.д.

9.3 Напишите программу, которая считывает 10 чисел в массив и выводит среднее значение, а также число в массиве, наиболее удаленное по абсолютной величине от среднего.

9.4 Разработайте структурный план для задачи записи всех простых чисел, меньших 1000 (1 и 2 обычно считаются простыми числами, генерация простых чисел, и, вероятно, их придется решать отдельно). Напишите программу. *Подсказка*: используйте массив для хранения простых чисел по мере их нахождения.

9.5 В эксперименте делается $N$ пар наблюдений ($X_i$, $Y_i$). Наилучшая прямая линия наименьших квадратов, которую можно провести через эти точки (используя метод наименьших квадратов), имеет точку пересечения $A$ на оси $x$ и наклон $B$, где

$$B = \left(S_1 - S_2 S_3 / N\right) / \left(S_4 - S_2^2 / N\right)$$
$$A = S_3 / N - S_2 B / N$$

и

$$S_i = \sum X_i Y_i, \quad S_2 = \sum X_i, \quad S_3 = \sum Y_i, \quad S_4 = \sum X_i^2$$

Коэффициент корреляции R определяется выражением

$$R = \frac{NS_1 - S_2 S_3}{\sqrt{NS_4 - S_2^2}\sqrt{NS_5 - S_3^2}}$$

где $S_5$ = sum $Y_i^2$. ($R$ = 1 подразумевает идеальную линейную зависимость между Xi и Yi. Этот факт можно использовать для проверки вашей программы.) Все суммирования находятся в диапазоне от 1 до N. Наблюдения хранятся в текстовом файл. Неизвестно, сколько наблюдений. Напишите программу для чтения данных и вычисления $A$, $B$ и $R$. *Подсказка*: вам не нужны массивы!

9.6 Если набор точек ($X_i$, $Y_i$) соединен прямыми линиями, значение $Y$, соответствующее значению $X$, лежащему на прямой между $X_i$ и $X_{i+1}$, определяется выражением

$$Y = Y_i + \left(X - X_i\right)\frac{Y_{i+1} - Y_i}{X_{i+1} - X_i}$$

Этот процесс называется линейной интерполяцией. Предположим, что нелинейная интерполяция содержит более 100 наборов пар данных, хранящихся в порядке возрастания $X_i$ в текстовом файле. Напишите программу, которая будет вычислять интерполированное значение $Y$ при произвольном значении $X$, введенном с клавиатуры. Предполагается, что $X$ находится в диапазоне, охватываемом данными. Обратите внимание, что данные должны быть отсортированы в порядке возрастания относительно значений $X_i$. Если бы это было не так, пришлось бы сначала их рассортировать.

# Заключение

Представленное учебное пособие должно помочь студенту не только научиться программировать простые, но порой очень нужные для оперативной обработки информации программы, а в большей степени пособие позволяет достаточно свободно читать программные коды, написанные другими специалистами, и разрабатывать свои, как инженерные, так и научные программные комплексы. В данном случае, речь идет о формировании у студента навыков разработки алгоритмов решения той или иной задачи. Приведенные в пособии примеры дают возможность более полно понимать проблемы оптимизации вычислительного процесса, так необходимой для работы с большими массивами данных.

На протяжении всего пособия основное внимание уделялось написанию четких, последовательных программ для решения интересных задач. Программа, написанная много времени назад, хотя и может делать то, что требуется, будет трудна для понимания, когда вы уделите ей снова внимание через месяц или два. Поэтому серьезные программисты уделяют немало внимания тому, что называется стилем программирования, чтобы сделать свои программы более понятными и читабельными как для себя, так и для других потенциальных пользователей. Это может вас несколько раздражать, если вы впервые начинаете программировать, потому что вам, естественно, не терпится побыстрее приступить к работе. Но небольшое дополнительное внимание к макету вашей программы принесет огромные дивиденды в долгосрочной перспективе, особенно когда дело доходит до отладки.

Можно привести некоторые советы о том, как улучшить свой стиль программирования.

- Вы должны свободно использовать комментарии, как в начале программного модуля или подпрограммы, чтобы кратко описать, что они делают, и любые специальные методы, которые могли быть использованы, так и на протяжении всего кода, чтобы представить различные логические разделы. Любые ограничения на размер и тип данных, которые могут использоваться в качестве входных данных, должны быть четко указаны в комментариях (например, максимальные размеры массивов).
- Значение каждой переменной должно быть кратко описано в комментарии при ее объявлении. Вы должны систематически

объявлять переменные, например, в алфавитном порядке по типам.

- Подпрограммы должны располагаться в алфавитном порядке, по крайней мере, с одной пустой строкой между ними.
- Пустые строки должны свободно использоваться для разделения разделов кодирования (например, структуры цикла до и после).
- Кодирование внутри структур (циклов, решений и т.д.) должно иметь отступ в несколько столбцов, чтобы они выделялись.
- В заявлениях следует использовать пробелы, чтобы сделать их более читабельными, например: по обе стороны от операторов и знаков равенства; после запятых
- Однако в сложных выражениях местами могут быть опущены пробелы, что может сделать структуру более понятной.
- Операторы FORMAT должны быть сгруппированы вместе.
- Оператор GOTO никогда не должен использоваться ни при каких обстоятельствах.
- Вы должны стараться не ломать конструкции посередине, т.е. с помощью CYCLE или EXIT.
- Следует избегать заявлений, которые генерируют предупреждение об устаревании — они вполне могут исчезнуть во время пересмотра следующего стандарта.

## Список использованной литературы

1 Walter S. Brainerd Guide to Fortran 2003 Programming. – London: Springer-Verlag, 2009. – 357 p.
2 Бартеньев О.В. Современный Фортран. – М.: ДИАЛОГ-МИФИ, 2000. – 449 с.
3 Берков Н. А., Беркова Н.Н.Алгоритмический Язык Фортран 90: Учебное пособие. – М: МГИУ, 1998 г. –96с.
4 М. Меткалф, Дж. Рид Описание языка программирования Фортран 90. – М.: Мир, 1995. – 302 с.
5 S.J. Chapman Fortran 95/2003 for Scientists and Engineers. The McGraw-Hill Companies, 2008. – 987 p.
6 Прохоренок Н.Ф., Дронов В.А. Python 3. Самое необходимое. – СПб.: БХВ-Петербург, 2016. – 459 с.
7 Л. Ромальо Python, К вершинам мастерства. – М.: ДМК Пресс, 2016. – 768 с.
8 У. Маккинли Python и анализ данных. – М.: ДМК Пресс, 2015. – 482 с.
9 D.E. Knuth The Art of Computer Programming. Volume 2: Seminumerical Algorithms. Addison-Wesley, 1981. – 498 p.
10 Жуков Л.А. Общая океанология. – Л.: Гидрометеоиздат, 1976. – 376 с.
11 C.A.R. Hoare Quicksort // The Computer Journal, Volume 5, Issue 1, 1962, pp. 10–16

*Учебное издание*

**Чанцев Валерий Юрьевич**,
доцент, канд. геогр. наук

Язык программирования ФОРТРАН
для решения океанологических задач

Часть 1

*Печатается в авторской редакции.*

Подписано в печать 28.07.2022. Формат 60×90 1/16.
Гарнитура Times New Roman. Печать цифровая.
Усл. печ. л. 9, 75. Тираж 10 экз. Заказ № 1266.
РГГМУ, 192007, Санкт-Петербург, Воронежская, ул., д. 79.