МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ СТРОИТЕЛЬНЫЙ
УНИВЕРСИТЕТ

А.М. Купфер, Ж.И. Мсхалая, Ю.В. Осипов

ФОРТРАН

Учебное пособие

Москва 2004

УДК 681.3.06

Купфер А.М., Мсхалая Ж.И., Осипов Ю.В. Фортран. Учебное пособие. Под ред. А.М. Купфера/ Моск. госуд. стр. ун-т. – М.: 2001 –87 с.

ISBN 5-7264-0230-8

Пособие предназначено для студентов 1-го и 2-го курса МГСУ всех факультетов. Содержащийся в нем материал соответствует учебным программам курса «Информатика». Изложение основ языка Фортран сопровождается большим количеством примеров. Рассмотрены особенности реализации программ в различных средах (WATFOR-77, Microsoft Fortran PowerStation).

Научный редактор А.М. Купфер

Рецензент
доц. канд. техн. наук Ю.М. Плотинский (МГУ им. М.В. Ломоносова)

ISBN 5-7264-0230-8

1. ВВЕДЕНИЕ

1.1. Общие сведения

Электронно-вычислительная техника широко применяется в сфере строительного производства. На стадии проектирования использование компьютеров позволяет привлечь для расчета зданий, сооружений и их элементов современный математический аппарат. Это делает расчеты на прочность научно обоснованными, надежными и, как правило, более экономичными. ЭВМ позволяют рассчитывать графики возведения объектов, завоза материалов и изделий, определять оптимальные маршруты перевозок грузов, распределение по объектам строительной техники и оборудования. В производстве строительных изделий вычислительные машины определяют наилучшие параметры технологических процессов, контролируют работу оборудования, запасы сырья и сбыт готовой продукции.

1.2. Устройство ЭВМ

Компьютер – это устройство для хранения, передачи и обработки информации в цифровой (числовой) форме.

Основным устройством каждой ЭВМ является *процессор*, где выполняются арифметические операции; процессор же управляет работой остальных устройств компьютера.

Для хранения информации ЭВМ имеет память. Как правило, у компьютера память двух типов. Первый тип, по традиции называемый ОЗУ (оперативное запоминающее устройство), – это память сравнительно небольшого объема и огромного быстродействия. В ОЗУ хранится информация для задач, решаемых в данный момент. Второй тип памяти, называемый ВЗУ (внешнее запоминающее устройство) имеет практически неограниченный объем, но работает гораздо медленнее, чем оперативная память. Чаще всего в качестве внешней памяти используются накопители на *жестких* магнитных дисках (винчестерах), дискетах (сменных *гибких* магнитных дисках) и компакт-дисках.

Для того, чтобы задать компьютеру исходную информацию и получить результаты расчета, используются *устройства ввода и вывода*. Информация для ввода должна быть подготовлена в машиночитаемой форме. Например, можно набирать информацию на клавиатуре с контроль-

2. НАЧАЛЬНЫЕ ПОНЯТИЯ ЯЗЫКА ФОРТРАН

2.1. Алфавит языка Фортран. Простой пример программы

В Фортране, как и в большинстве других алгоритмических языков для записи программ, используются латинские буквы, обычные цифры, специальные символы: + (плюс), – (минус), * (звездочка, например, как знак умножения), / (косая черта, или слэш), . (десятичная точка), , (запятая), ( ) (круглые скобки), ' (апостроф). Транслятор языка Фортран не различает заглавные и строчные буквы. Поэтому при записи программ можно пользоваться как "большими" буквами, так и "малыми".

В стандарте языка буквы заглавные, но современные компьютеры позволяют использовать малые, что мы и рекомендуем.

Пробел мы будем иногда обозначать через ␣, а чаще просто через отсутствие символа.

> В записи программы пробелы не играют никакой роли и используются для оформления ее текста.

Каждая инструкция программы начинается с новой строки и располагается *правее* 6-й позиции и не далее 72-й, например:

```
|    |a=3
|    |b=a+14
```

Вертикальными прямыми условимся отмечать именно эту 6-ю позицию (позиция – место символа в строке).

Показанный фрагмент программы не отличается от обычной записи формул, например, в математике. Сходная ситуация имеет место и в большинстве программ на Фортране. Вот простой, но важный пример.

Пример 2.1. Составить программу вычисления выражения

$$z = \sin x + y,$$

где $x = 4{,}3a - 2b$, $y = 5ab$ при $a = 3$; $b = -7$.

Программа имеет вид:

```
|    |a=-3
|    |b=7
|    |x=4.3*a-2*b
|    |y=5*a*b
|    |z=sin(x)+y
|    |print *,'z=', z
|    |end
```

Замечания:

– в числах ставится десятичная точка, а не запятая: 4.3, а не 4,3;

– *аргументы* в записи функций заключаются в *скобки*;

– в каждой строке записывается законченное действие – оператор;

– некоторые операторы начинаются с *ключевых слов*; здесь встречаются **print** * (печать) и **end** (конец текста).

Ключевые слова в данном пособии выделяются полужирным шрифтом. Так же мы поступаем и с некоторыми символами (круглые скобки, точки, запятые), если они входят в качестве неотъемлемых элементов в те или иные конструкции Фортрана, но, скажем, *не в состав записи чисел и формул*. Выделяем мы и названия специальных клавиш. Порядок следования операторов понятен: чтобы вычислить *z*, нужно сначала найти *x* и *y*, а для этого необходимо знать *a* и *b*.

Оператор

```
print *,'z=', z
```

выводит на экран компьютера результат в форме z=число. Можно не писать 'z='; тогда оператор

```
print *,z
```

напечатает только число.

В случае успешного выполнения программы после ее текста будет напечатано

```
z=            -104.0192000.
```

С точки зрения оформления результаты выглядят неидеально. Возможности улучшения формы вывода будут рассмотрены в дальнейшем.

Вопросы компьютерной реализации программ рассмотрены в Приложениях 1 и 2.

А сейчас в заключение коснемся некоторых проблем *ввода*.

2.2. Исходные данные. Ввод с клавиатуры

Исходными данными в примере 2.1 являются значения *a* и *b*. Они внесены непосредственно в текст программы. Таким образом, наша программа решает только *одну* конкретную задачу. Простое изменение данных требует вмешательства в *текст*, т.е. программу нужно переделывать. Правда, в данном примере изменения в тексте невелики (как и сама программа), однако в других случаях (данных много, программа сложная, к тому же связанная с другими программами) это может быть неприем-

обязательно начинать с позиции 7. Иногда фрагменты длинных операторов для удобного зрительного восприятия тоже выделяются пробелами.

Позиции 1-5. Некоторые операторы могут быть снабжены *меткой*, по которой ЭВМ находит нужное место в программе. Метка представляет собой целое число без знака от 1 до 99999 и записывается в позициях 1-5. В этих позициях метку можно разместить любым образом, так как пробелы и незначащие (левые) нули в этих позициях транслятором не воспринимаются.

Так, следующие записи равносильны:

`76¬¬¬; ¬¬¬¬76; ¬¬76¬; 7¬¬¬¬6; ¬7¬6¬` и т.д.

В качестве меток пользователь, составляющий программу, может выбирать *любые целые числа* от 1 до 99999; в программе метки могут следовать *в любом порядке*.

Позиция 6. В этой позиции ставится символ продолжения. Обычно в позиции 6 стоит пробел. Однако, если некоторый оператор не умещается на одной строке, то его можно продолжить на следующие строки; при этом в шестой позиции второй и последующих строк ставится *любой символ*, отличный от пробела и нуля.

Часто в качестве символа продолжения используется звездочка (*). Мы рекомендуем ставить >. Оператор можно разорвать (для продолжения на другой строке) в любом месте; если разрыв пришелся на знак арифметической операции, то на второй строке этот знак *не повторяется*:

```
      a+                                  
   >  b          то же, что          a+b
```

Позиции 73-80. Эти позиции транслятором не обрабатываются, и программист может написать там любую информацию для себя (например, проставить номера строк) или оставить позиции чистыми. Отметим, что рис.3.1 относится только к самой программе; при записи числовых данных используются все 80 позиций строки.

Замечание. На самом деле исходные данные можно располагать и правее 80-й позиции в строке. Однако это делает неудобным их отображение на экране и может привести к ошибкам.

3.3. Комментарии

Если у некоторой строки программы в первой позиции стоит латинская буква c или символ "*", то данная строка является *комментарием* и

транслятором не обрабатывается. Комментарии используются для записи пояснений к программе и записываются в любых позициях от 2-й до 80-й; при этом в них можно использовать любые символы клавиатуры дисплея, даже не входящие в алфавит Фортрана (например, русские буквы).

3.4. Бесформатный ввод простых переменных

Оператор бесформатного ввода имеет вид:

read *, список ввода

Список – это набор элементов, разделенных запятыми.

Элементами *этого* списка являются имена переменных, которые отделяются друг от друга запятыми; например,

```
read *,a,b,c
read *,d,e и т. д.
```

Выполнение оператора **read** заключается в том, что числовые данные, вводимые с клавиатуры на экран дисплея, записываются в память ЭВМ – в ячейки, отведенные для переменных, перечисленных в списке оператора **read**. Дойдя до очередного оператора **read**, компьютер *приостанавливает* выполнение программы и *ждет*, когда пользователь наберет на клавиатуре предусмотренные данные. Такую ситуацию называют еще *прерыванием*.

Чтобы узнать, какие данные вводить, надо предусмотреть выдачу запросов на экран *раньше*, чем программа дойдет до соответствующего оператора **read** (т.е. разработать *интерфейс пользователя*). Для этого перед каждым оператором **read** надо ставить, например, оператор **print** с запросом примерно такого вида:

```
print *,'введите a,b,c'
```

или в "содержательной форме"

```
print *,'введите начальные плотность,
> давление, температуру'
```

Данные для оператора **read** записываются в одной или нескольких строках. Числа отделяются друг от друга *одним или несколькими пробелами*, запятой (хуже), либо тем и другим вместе (не рекомендуем).

Закончив набор строки данных, нажимаем **Enter**, и программа продолжит работу.

При этом оператор **read** выбирает числа из строки данных слева направо и в таком же порядке присваивает эти значения переменным из списка ввода.

Пример 3.1. Пусть программа выполняет оператор

```
read *,n,c,d
```

Если набрать на клавиатуре

```
-3 0.6 -15,
```

то переменная *n* получит значение –3, переменная *c* будет равна 0.6, а *d* станет равной –15.

Если данные в строке кончились, а переменные в списке остались, то оператор **read** автоматически начинает считывать данные из следующей строки и т.д. Такая ситуация возникнет, если мы нажмем **Enter** раньше времени, т.е. не закончив ввод всех чисел для данного оператора **read**. Ничего страшного не случится, произойдет переход на новую строку, и компьютер будет ждать остальных данных. Правда, при этом предыдущие данные раньше "уйдут" с экрана вверх.

Лишние данные в строке прочитаны не будут.

3.5. Бесформатный вывод. Оператор PRINT

Оператор бесформатного вывода имеет вид

print *, список вывода

Как и раньше, элементы в списке разделяются запятыми. Эти элементы могут быть двух типов: 1) имена переменных и 2) символьные *константы* (любой набор символов, заключенный в апострофы; см. разд. 3.4, а также разд. 12.1). Выполняется оператор **print** следующим образом: на экран дисплея выводятся значения переменных, а также содержание строк.

Пример 3.2. Пусть в ходе предыдущих вычислений *переменные a, c, e* получили значения $a = 12.5$, $c = -1.25$, $e = 125.5$. Тогда после выполнения оператора

```
print *,'a=',a,' e=', e, ' c=', c
```

на экран дисплея будет выведена следующая строка:

```
a=  12.500000000 e=  125.500000000 c=  1.2500000000
```

Оператор **print** имеет следующие особенности:

1)при переходе к оператору **print** он начинает печатать значения переменных списка *с новой строки.*

2) выводимые числа имеют 9 цифр после десятичной точки;

3) оператор **print** * не позволяет управлять расположением чисел на строке. Он заполняет строку полностью, после чего переходит на следующую.

Пример 3.3. Переменные *a, b, c, d, e* имеют те же значения, что и в примере 3.2. Тогда оператор

```
print *,'a=',a,' b=',b,c,d,e
```

выведет на экран дисплея 2 строки:

```
a=    12.500000000 b=    1.250000000     13.6000000
     14.700000000       1.470000000
```

Обратите внимание на пробел(ы) между апострофом и *b=*.

Этим мы предотвращаем примыкание *b* к значению *a*.

3.6. Операторы останова STOP и PAUSE

Действие оператора **stop** заключается в прекращении работы программы с последующим выходом из нее. Отметим, что такая остановка все равно произойдет, когда компьютер дойдет до конца выполняемой программы. Поэтому в программах оператор **stop** применяется редко.

Оператор **pause** применяется, если нужно на время приостановить выполнение программы, например, прочитать промежуточные результаты. Выполнение возобновляется после нажатия любой клавиши. В система MS FORTRAN выдается сообщение "Нажмите любую клавишу", правда, по-английски.

3.7. Ввод данных из файла. Оператор открытия файла OPEN

Рассмотренный способ ввода данных в программу называется *диалоговым*, другими словами, работа программы протекает в *интерактивном режиме*. Преимущества этого способа понятны: возможность принимать решения в ходе выполнения программы, удобный интерфейс с пользователем, минимальные требования к подготовке пользователя.

Недостатки диалогового ввода – как водится, продолжение достоинств. При новом запуске программы приходится *все данные вводить заново*. Данных может быть много: вспомним, в конце концов, что компьютер предназначен для *хранения* и *обработки* информации. Обрабатывают информацию предназначенные для этого программы, а хранить ее

можно только в *файлах*. Поэтому весьма желательно, чтобы программа могла считывать данные *из других файлов*. Такая возможность в Фортране предусмотрена.

Оператор чтения данных из файла имеет вид:

read (номер файла, *) список .

Чтобы компьютер знал, какой файл имеется в виду, нужно указать, какому файлу присваивается данный номер. Такое указание записывается с помощью оператора открытия файла **open**. Этот оператор может выполнять и другие функции, но мы ограничимся вышеназванной.

Выглядит оператор открытия файла так:

open (номер, **file** = 'имя')

Номер может быть константой или переменной целого типа. В последнем случае переменная должна предварительно получить нужное значение (любым допустимым в Фортране способом).

Некоторые номера зарезервированы для данных, обрабатываемых внешними устройствами. Например, 5 – клавиатура, 6 – экран. Поэтому лучше выбирать номера, начиная, скажем, с 8 и далее. Полнота имени (т.е. маршрута) зависит от положения открываемого файла относительно программы. Полного имени будет достаточно всегда. Если файл расположен в текущем каталоге, то достаточно указать собственно имя и расширение (если есть). Для файла исходных данных часто выбирают расширение dat.

Файл для исходных данных создается так же, как и программный. В отличие от программы, у файла исходных данных можно использовать все 80 позиций.

Пример 3.4. Рассмотрим фрагмент программы:

```
open (8,file='pet1.dat')
read (8,*) a,b,c
read (8,*) d,e
. . . . . .
```

Пусть данные в файле pet.dat записаны так:

```
12.5, 1.25, -1.25, -12.5
0.125 125.5
```

После выполнения первого оператора **read** переменная *a* получит значение 12.5, переменная *b* – значение 1.25, переменная *c* – значение

–1.25. Если нужно, чтобы было $b = -1.25$, а $c = 1.25$, то можно или изменить строку данных или записать оператор **read** так:

```
read (8,*) a,c,b
```

Следует запомнить следующие особенности оператора **read**:

1) если числовых данных меньше, чем переменных в списке, то на данном операторе **read** выполнение программы будет прекращено;

2) если числовых данных больше, чем переменных в списке, то "лишние" числа не будут читаться и использоваться;

3) для каждого оператора **read** следует набирать числа с *новой строки данных* (даже если на предыдущей строке не все числа прочитаны).

Первые две ситуации возникают вследствие ошибок программиста; часто это связано с тем, что не учитывается третье правило. Так, в примере 3.4 оператор

```
read (8,*) d,e
```

начинает выбирать числа из второй строки данных и присваивает переменной *d* значение 0.125, а переменной *e* – значение 125.5.

Особенности выполнения оператора **read** иллюстрирует следующий пример.

Пример 3.5. Рассмотрим фрагмент программы:

```
open (9,file='nik2.dat' )
read (9,*) a,b,c
read (9,*) d,e
read (9,*) f
. . . . . .
```

Строки данных:

```
12.5 1.25
13.6 -1.36
14.7 1.47
```

В общей сложности в списках 6 переменных, а в строках данных 6 чисел. Но оператор

```
read (9,*) a,b,c
```

присвоит переменой *a* значение 12.5, переменной *b* значение 1.25, перейдет на вторую строку данных и присвоит переменной *c* значение 13.6. Число –1.36 останется непрочитанным; однако оператор

```
read (9,*) d,e
```

начнет чтение со следующей строки и присвоит *d* значение 14.7, *e* – значение 1.47. На операторе

```
read (9,*) f
```

выполнение программы будет прервано, так как данных больше нет.

3.8. Печать результатов в файл

Это действие является обратным к чтению данных из файла. Файл для печати открывается аналогично. Сам оператор печати имеет вид

write (номер, *) список.

В остальном печать происходит, как было описано выше.

Заметим, что вся старая информация, если она была в файле, предварительно стирается.

Часто бывает желательно иметь возможность при новом запуске программы *дописывать* информацию в файл *без потери* старой. В этом случае нужно при открытии файла указать значение параметра доступа ***access***, равное *append* (добавление). Общий вид оператора открытия файла может быть таким:

open (номер, **file** = 'имя', **access** = 'append')

Теперь новая информация будет печататься после старой.

Пример 3.6. Здесь файл pet1.res будет пополняться при каждом запуске программы.

```
open (9,file='pet1.res',access='append')
write (9,*) a,b,c
```

3.9. Закрытие файлов. Оператор close

Количество файлов, которое может быть открыто, определяется настройкой операционной системы. Обычно их оказывается достаточно. Все же ненужные ранее открытые файлы рекомендуется закрывать, т.е., отключать от программы. Это осуществляется с помощью оператора **close** (закрыть).

Он имеет следующий вид:

close (номер),

где номер должен соответствовать одному из открытых файлов.

Если понадобится, файл можно открыть снова.

3.10. Начало и конец текста программы на Фортране

Текст программы начинается с выполнения первого шага, предписываемого алгоритмом решения задачи, или с описаний некоторых переменных, встречающихся в программе.

Неисполняемые операторы – описания переменных и функций будут рассмотрены в разд. 4, 8, 15.

Первым исполняемым шагом алгоритма часто является ввод исходных данных в память ЭВМ, т.е. оператор **read** (но это не обязательно). Перед этим оператором ввода полезно поместить вывод на экран строки-подсказки. Например, перед оператором

```
read *,a,b,c
```

записать оператор

```
print *,'Задайте значения a,b,c'
```

Если такого "приглашения" не сделать, то программист должен помнить, когда и какие переменные ему следует вводить.

Последним оператором любой программы является неисполняемый оператор **end**. Этот оператор не выполняет никаких действий, а просто указывает на окончание текста программы или одной из подпрограмм (разд. 14, 15). В правильно работающей программе должен быть один оператор **end**.

После записи на экране оператора **end** можно, как уже было показано в разд. 1, запускать программу на выполнение.

4. КОНСТАНТЫ И ПРОСТЫЕ ПЕРЕМЕННЫЕ ЦЕЛОГО И ВЕЩЕСТВЕННОГО ТИПА. ПРАВИЛА ЗАПИСИ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ. ОПЕРАТОР ПРИСВАИВАНИЯ. ОСОБЕННОСТИ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ В ФОРТРАНЕ

4.1. Целые константы

В математике целые числа являются частью множества вещественных чисел. В Фортране термины "целый" и "вещественный" используются для обозначения двух различных типов констант (чисел).

Целая константа – это последовательность цифр без десятичной точки, снабженная знаком или без него. Для хранения целой константы в памяти ПЭВМ, как правило, используется 4 байта, т.е. 32 двоичных разряда, один из которых используется для хранения знака числа. Поэтому

целая константа не может превышать $2^{31}-1 \approx 2 \cdot 10^9$. Константа без знака или со знаком + (плюс) воспринимается как положительная, со знаком минус – как отрицательная.

Примеры целых констант: 125, –1251, +1997, 0.

4.2. Вещественные константы с фиксированной точкой

Эти константы состоят из целой части, точки и дробной части, например: 125.4567, –1983.11, 0.0 и т.д. Правило знаков для вещественных констант то же, что и для целых (знак "минус" обязателен, знак "плюс" – нет). Отсутствующую целую или дробную части можно не писать, но точка должна быть обязательно: 2.00 = 2.0 = 2.; 0.01 = .01; +0.25 = .25; –0.25 = – .25; 0.0 = 0. = .0.

4.3. Вещественные константы с плавающей точкой

Для расширения диапазона представляемых в памяти компьютера чисел используется *экспоненциальная* форма. В этой форме число X представляется в виде $X = Y \cdot 10^d$, где Y – число с фиксированной точкой или целое, d – целое число; при этом Y называется мантиссой, а d – порядком. В простейшем случае (по умолчанию) мантисса имеет 7 десятичных цифр; порядок может быть от –38 до +38. Таким образом, в экспоненциальной форме можно представлять числа в диапазоне примерно от 10^{-38} до 10^{+38}. Другое название этого типа данных – *константы с плавающей точкой*.

Вещественная константа с плавающей точкой в языке Фортран имеет вид:

мантисса **E** порядок,

где *мантисса* – вещественное число с фиксированной точкой или целая константа; *порядок* – целая константа (одно- или двузначная). Разделитель **E** (или **e**) обязателен.

В константе с экспонентой мантисса может отсутствовать (тогда сама константа равна нулю).

Пример 4.1. Число 0.000015 можно представить в виде $0.000015 = 0.15 \cdot 10^{-4}$, следовательно, в экспоненциальной форме это число можно записать как 0.15E–04.

Заметим, что представление числа в форме с экспонентой неоднозначно; то же самое число можно записать как 0.15E–04 = 1.5E–05 =

= 15.0e–06 = 0.015e–03 и т.д. программист может выбрать любой вариант. Однако при печати константа с экспонентой *нормализуется*. Это означает, что порядок d выбирается так, чтобы мантисса Y принадлежала промежутку $0.1 \le |Y| < 1$. Нормализованная форма числа есть 0.15E–04.

Пример 4.2. При написании порядка можно опускать один нуль и знак + (плюс), т.е. следующие записи эквивалентны: 1.25E+01 = = 1.25E01 = 1.25E1 = 12.5E0 = 12.5E00; нормализованная форма этого числа есть 0.125E+02.

4.4. Целые и вещественные переменные

Для обозначения переменных в Фортране используются не только буквы (как обычно в математике), но и *комбинации латинских букв, цифр, символов подчеркивания и $*, состоящие не более чем из 6 символов и начинающиеся не с цифры. Такие комбинации называются *именами* (или *идентификаторами*). Имя переменной может быть связано с ее ролью в данной задаче (переменные dlina, nomer), но это не обязательно.

Здесь рассматриваются только *простые* переменные, применяющиеся для обозначения скалярных величин целого и вещественного типа.

Переменная является целой, если ее имя начинается с одной из букв: *i, j, k, l, m, n*. Если имя переменной начинается с любой другой буквы, символа подчеркивания или $, то эта переменная – вещественная.

Примеры переменных целого типа: *il86, jessy, kiev, lcd, mi5*.

Примеры вещественных переменных: *apollo, soyuz, dlina, x, y*.

Целые переменные и константы используются, как правило, для нумерации и перечисления каких-то объектов или событий в программе. Арифметические действия с ними *не всегда* выполняются по общепринятым правилам (см. разд. 4.7). Вещественные переменные и константы используются для вычислений

4.5. Правила записи арифметических выражений

Арифметические действия обозначаются в Фортране следующим образом:

+ сложение,
– вычитание,
* умножение,
/ деление,

** — возведение в степень.

Например, выражение $b^2 - 4ac$ при записи по правилам Фортрана примет вид: *b**2-4*a*c*.

Порядок действий в Фортране обычный: если нет скобок, то сначала выполняется возведение в степень, затем (слева направо) умножение и деление и, наконец, (слева направо) сложение и вычитание.

Для более сложных формул и изменения порядка действий используются круглые скобки (их число практически не ограничено). В этом случае, как обычно, сначала выполняются действия в "самых внутренних" скобках и т.д.

Если рядом стоят два *возведения в степень*, то они выполняются справа налево: *a**b**c* означает то же, что *a***(*b***c*). Нельзя ставить рядом два знака арифметических операций: если нужно *x* умножить на –1, то это записывается с использованием скобок: *x*(–1).

При переводе арифметических формул на Фортран надо быть весьма внимательным. Наиболее часто встречаются такие ошибки, как пропуск знака умножения (запись 4*a* вместо 4**a*) и механическая замена длинной черты дроби на косую черту ($\frac{b}{2a}$ надо записывать в виде *b*/(2**a*) или как *b*/2/*a*; выражение *b*/2**a* означает совсем иное: $\frac{b}{2}a$).

Допускается соединять знаками арифметических операций переменные и константы разных типов: целые и вещественные. При этом в момент выполнения арифметического действия целое число будет преобразовано (перекодировано) в вещественную форму. Поэтому арифметическое действие 2**a* будет выполняться на микроскопическое время *дольше*, чем действие 2 **a*. Результат смешанной арифметической операции будет иметь вещественный тип.

Таким образом, только в случае, когда все переменные и константы, входящие в выражение, целые, получаем арифметическое выражение *целого типа*.

Пример 4.3. Определим порядок действий и тип результата при вычислении выражения.

(*a***b*)**0.5+*c***2,
(*i***j*)**0.5+*k***2.

В каждой из формул сначала выполняется умножение в скобках, затем – два возведения в степень и, наконец, два сложения. Результат первого выражения будет вещественным. Результат второго выражения также будет иметь вещественный тип, так как в формуле участвует одна вещественная константа 0.5.

4.6. Оператор присваивания

Арифметический оператор присваивания в Фортране имеет вид:

переменная = арифметическое выражение

В состав *арифметического выражения* могут входить числа, переменные, знаки операций, функции (см. ниже).

Выполняется этот оператор следующим образом: вычисляется значение арифметического выражения, стоящего справа, и результат присваивается переменной, записанной слева от знака = (знака присваивания).

Пример 4.4. Нахождение суммы трех чисел.

```
read *,a,b,c
s=a+b+c
print *,s
end
```

Если $a = 12.5$, $b = 1.25$, $c = 13.6$ (как в примере 3.2), то будет напечатано значение s, равное 27.35.

Операторы присваивания можно использовать вместо оператора **read** для первоначального присвоения переменным числовых значений. Это становится неудобным, если таких присваиваний больше двух-трех.

В результате выполнения оператора

```
y=x
```

значение x записывается в y; при этом старое значение y стирается, а содержимое x сохраняется.

Важный частный случай оператора присваивания.

После выполнения оператора

```
x=x+a
```

значение переменной x увеличивается на a.

Пример 4.5. Рассмотрим фрагмент программы

```
x=1.5
```

```
      y=x
      x=x+4
      z=2*x
```

Нетрудно видеть, что после выполнения этих четырех операторов значения переменных x, y, z будут такими: $x = 5.5$, $y = 1.5$, $z = 11.0$.

Пример 4.6. Нахождение корней квадратного уравнения. Пусть дискриминант положителен, а старший коэффициент отличен от нуля.

Программа снабжена комментариями.

```
c     ввод исходных данных
      read *,a,b,c
c     нахождение дискриминанта
      d=b**2-4.*a*c
c     нахождение корней
      x1=(-b-d**0.5)/(2.*a)
      x2=(-b+d**0.5)/(2.*a)
c     печать результатов
      print *, x1,x2
      end
```

При переносе на другую строку операторы Фортрана можно разрывать в любом месте; если перенос пришелся на знак арифметической операции, то на второй строке он не повторяется.

Пример 4.7. Использование символа продолжения. Программа та же, что и в примере 4.6.

```
c     ввод
c     исходных данных
      read
     * *,a,b,c
      d=b*
     * *2-4.*a*c
      x1=(-b-d**0.
     * 5
     * )/(2.*a)
      . . . . . .
```

Видно, что использование звездочки в 6-й позиции для продолжения ухудшает "читаемость" программы. Можно, как уже было рекомендовано, ставить >.

Заметим, что пример 4.7 носит несколько искусственный характер. Обычно перенос оператора на следующую строку используется для "длинных" операторов, которые не умещаются на одной строке в позициях 7-72. Можно посоветовать выявлять при переносах структуру выражения, например, располагать на соседних строках числитель и знаменатель.

4.7. Особенности выполнения арифметических операций в Фортране

На результат арифметических операций сказывается то, что все числа хранятся в памяти компьютера с конечным числом значащих цифр. В силу этого деление зачастую нельзя выполнить точно (напомним пример 1./3.=0.333333...), а при сложении, вычитании и умножении пропадают младшие разряды результата. При этом следует иметь в виду, что во всех случаях в Фортране происходит не округление, а *отбрасывание* (усечение) младших разрядов результата, не умещающихся в ячейке памяти.

Возведение в степень (операция **) производится по-разному в зависимости от показателя степени. Если показатель – целая переменная, целая константа или арифметическое выражение целого типа, то возведение в степень производится путем последовательного умножения.

$$a**3 \rightarrow a*a*a,$$
$$b**(-2) \rightarrow 1./(b*b).$$

Если же показатель имеет вещественный тип, то для выполнения возведения в степень транслятор использует формулу $Y^A = e^{A \cdot \ln Y}$. Следовательно, возведение в степень с вещественным показателем определено только при $Y > 0$. Например, нельзя вычислить выражение $y**(1./3)$ при $y = -8$, хотя вообще-то $\sqrt[3]{-8} = -2$. По той же причине при $x = 0$ нельзя вычислить $x**2.$, но можно вычислить $x**2$ или $x*x$.

Арифметическая операция с целыми переменными или константами выполняется так, что результатом является целое число. Это существенно при делении целых чисел и при возведении целого числа в отрицательную степень (в остальных случаях результат и так целый).

В Фортране при делении целых чисел и возведении целого числа в целую отрицательную степень *дробная часть результата отбрасывается* (без учета знака).

Пример 4.8. Результаты выполнения операций с целыми числами в Фортране:

4/3 = 1; 3/4 = 0; 2**(–2) = 0;

–5/2 = –2; 5/(–2) = –2; (–1)**(–3)= –1.

В сложном арифметическом выражении каждая операция выполняется в соответствии с типом ее аргументов; так же определяется и тип результата. Например, если $x = 2.$, $k = 3$, $m = 5$, то оператор

```
y=x*k/m
```

присвоит переменной y значение 1.2, так как 2.*3 = 6., 6./5 = 1.2. В то же время оператор

```
z=k/m*x
```

присвоит переменной z значение 0, так как 3/5 = 0, 0*2. = 0

Если в левой части оператора присваивания стоит целая переменная, а справа – вещественное выражение, то после вычисления выражения его дробная часть (без учета знака) будет отброшена.

Например, при $x = 2.$, $k = 3$, $m = 5$ оператор

```
l=x*k/m
```

присвоит переменной l значение 1 (сравните с примером 4.8).

Если в левой части оператора присваивания стоит вещественная переменная, а в правой – выражение целого типа, то получившийся целый результат будет переведен (перекодирован) в вещественное число.

Например, при $j = 2$, $k = 3$, $m = 5$ оператор $w = j*k/m$ присвоит переменной w значение 1.0

4.8. Явное описание типов переменных

Автоматическое назначение типов по первым буквам имен не всегда удобно. Например, традиционным является обозначение длины через l, а массы через m, они вовсе не обязаны быть целыми. Наоборот, часто величины r, s, t по смыслу – целые (например, индексы). В таких случаях программист может сам задать тип нужных величин с помощью *явного описания*. Для этого служат операторы

real (вещественный) и **integer** (целый).

В нашей ситуации можно написать:

```
real l,m
integer r,s,t
```

Такие операторы являются неисполняемыми. Они ставятся в начале программы.

Заметим, что современный подход к программированию рекомендует *все* переменные описывать *явно*.

Явное описание типа позволяет присваивать переменным *начальные значения* до начала выполнения программы. Для этого после имени переменной в операторе явного описания типа в наклонных черточках указывается значение этой переменной. Например:

```
real pi/3.14/,eps/1e-6/,i/1./
integer itermax/30/,n/100/,zero/0/
```

Замечание. В Фортранс имеется оператор **data**, который выполняет подобное присваивание без явного описания типа, но в современной литературе его использовать не рекомендуется.

4.9. Имена для констант. Оператор parameter

Некоторые конструкции Фортрана требуют обязательного присутствия *констант*, а не переменных. Это может вызывать ряд неудобств, особенно при изменении программы. Оператор **parameter** позволяет обозначить константу именем – точно так же, как переменную. Вид этого оператора следующий:

parameter (имя1 = константа1, имя2 = константа2...).

Попытка изменить значение такой именованной константы будет воспринято как ошибка. Именованную константу нельзя также использовать в операторах **format**. Оператор **parameter** является невыполняемым.

Пример 4.9.

```
parameter (n=3)
```

5. МАТЕМАТИЧЕСКИЕ ФУНКЦИИ. ОПЕРАТОР-ФУНКЦИЯ

5.1. Математические функции

Для вычисления часто встречающихся математических функций в языке Фортран предусмотрены специальные алгоритмы. Такие функции

называют *встроенными*. Пользователь должен знать закрепленные за ними стандартные обозначения. Вот некоторые из них:

Функция	Обозначение в Фортранс
$y = \sqrt{x}$	`sqrt(x)`
$y = e^x$	`exp(x)`
$y = \ln x$	`Alog(x)`
$y = \lg x = \log_{10} x$	`alog10(x)`
$y = \sin x$	`sin(x)`
$y = \cos x$	`cos(x)`
$y = \mathrm{tg}\, x$	`tan(x)`
$y = \arcsin x$	`asin(x)`
$y = \arccos x$	`acos(x)`
$y = \mathrm{arctg}\, x$	`atan(x)`
$y = \lvert x \rvert$	`abs(x)`

Аргумент должен быть обязательно заключен в скобки.

Аргументом может быть константа, переменная или даже арифметическое выражение. Важно лишь, чтобы (для всех указанных функций, кроме *abs*) аргумент имел *вещественный тип*. Значение перечисленных выше функций (кроме *abs*) также будет иметь вещественный тип. Аргумент функции *abs* может быть как вещественным, так и целым. Значение этой функции имеет тот же тип, что и аргумент. Для нахождения абсолютной величины целых чисел можно использовать также стандартную функцию *iabs*(*x*), аргумент которой должен иметь целый тип.

Пример 5.1. Из выражений sin(2.), sin(*a*), sin(2), sin(2**a*) неверным является третье (аргумент функции sin – целая константа). В остальных, верных случаях аргументом являются: вещественная константа, вещественная переменная, арифметическое выражение вещественного типа.

При вычислении стандартных функций должны соблюдаться некоторые разумные ограничения. Так, аргумент функции *sqrt* должен быть больше или равен 0, а аргумент функций *alog*, *alog10* должен быть больше 0. Аргумент функции *exp* должен быть меньше 88.7 (иначе результат выйдет за пределы представимых в памяти ЭВМ чисел).

Стандартные математические функции могут входить в арифметическое выражение и сами быть аргументом для других функций. Например, |sinx| записывается на Фортране как *abs*(*sin*(*x*)). При этом в сложных арифметических выражениях значения элементарных функций вычисляются до возведения в степень, умножения (деления) и сложения (вычитания) – если иное не предписано скобками.

Пример 5.2. Определим порядок действий при вычислении выражения

sin(*x***2)**2 + sin(*x*)**2 + *si* / *nx* + *sinx***2.

Вначале вычисляется выражение в скобках: *x***2, далее – два значения синуса: sin(*x***2) и sin(*x*); затем производится возведение в степень: sin(*x*)**2 и *sinx***2 (заметим, что *sinx* - новая переменная, не имеющая отношения к функции sin); после этого – деление *si* / *nx* и, наконец, три сложения.

Пример 5.3. Запись формул на Фортране:

а) $b^4 - 4ac$	*b***2–4**a***c*;
б) $\sin^2 3x + \ln^3 4y$	sin(3.**x*)**2+log(4.**y*)*3;
в) $\dfrac{\sqrt{x+y}}{\sqrt{x}+\sqrt{y}}$	sqrt(*x*+*y*) / (sqrt(*x*)+sqrt(*y*));
г) $c^2+e^{-2}+e^{a\,b}$	exp(2.)+exp(–2.)+exp(*a*–*b*).

5.2. Оператор-функция

Зачастую в различных местах программы необходимо производить вычисления по одной и той же формуле, т.е. вычислять значения одной и той же функции.

В Фортранс пользователь имеет возможность представить эту формулу в виде новой функции, дав ей «свое» имя. При этом запись станет более краткой, а программа – более наглядной. Для этой цели можно использовать *оператор-функцию*.

Рассмотрим пример. Пусть в различных местах программы встречаются вычисления по формулам $\sqrt{a^2+1}$, $\sqrt{b^2+c^2}$, $\sqrt{(d+u)^2+p^2}$. Эти три формулы – конкретные значения функции sqrt(*x***2+*y***2) при подходящих значениях *x*, *y*. Это выражение можно оформить как оператор-функцию:

```
f(x,y)=sqrt(x**2+y**2)
```

Конечно, имя может быть любым из не использованных в программе.

Различают *описание* и *вычисление* оператор-функции. В общем случае описание оператор-функции имеет вид:

имя функции(список аргументов) = арифметическое выражение.

В рассмотренном выше примере имя – *f*, список аргументов – *x*, *y*; арифметическое выражение – sqrt(*x***2+*y***2). Имя оператор-функции составляется по общим правилам составления имен в Фортране; напомним,что использовать это имя в той же программе для иных целей нельзя. Имя определяет, каков будет тип значений оператор-функции: целый или вещественный. Аргументами, или параметрами, оператор-функции могут служить простые переменные целого или вещественного типа.

В арифметическом выражении (в правой части), кроме аргументов оператор-функции, могут использоваться и *иные* константы или переменные.

Параметры, участвующие в описаниии оператор-функции, называются *формальными параметрами*; они показывают, *как* функция зависит от своих аргументов, но не определяют, при *каких* конкретных значениях аргументов эта функция будет вычисляться.

Описание оператор-функции помещается в начале программы, до *первого исполняемого оператора*, но *после* операторов описаний (если они есть).

В программе может быть несколько оператор-функций. Может встретиться ситуация, когда в формуле, например, для вычисления функции *h*, используются другие оператор-функции (например, *f* и *g*). В этом случае описание оператор-функции *h* должно следовать после описания функций *f* и *g*.

Пример 5.4. В начале программы записаны операторы

```
f(x,y)=sqrt(x**2+y**2)
g(a)=2.*atan(a/b)
h(x,y,d)=f(x,y)+g(d)+sin(x)
read *,n
. . . . . .
```

Здесь оператор-функция *f* зависит от двух аргументов: *x* и *y*. Оператор-функция *g* зависит от аргумента *a*, но при ее вычислении используется также константа 2. и переменная *b* основной программы.

Отметим, что в данном случае описания функций *f* и *g* могут идти в любом порядке. Оператор-функция *h*, при вычислении которой используются функции *f* и *g*, описана после функций *f* и *g*.

5.3. Правила использования операторов-функций

В том месте программы, где нужно проводить вычисления по формуле, заложенной в оператор-функцию, она просто ставится в правую часть оператора присваивания с нужными аргументами, которые могут быть константами, переменными или выражениями. Аргументы здесь называют фактическими параметрами. Конечно, предварительно нужно позаботиться, чтобы фактические параметры уже получили значения. То же самое относится и к другим переменным, *не* являющимся параметрами, но входящим в описание оператор-функции.

Фактических параметров должно быть столько же, сколько и формальных. Фактическими параметрами могут быть переменные, константы и арифметические выражения.

При этом фактические аргументы должны соответствовать формальным *по типу*, т.е. если первый формальный параметр есть *вещественная* переменная, то первым фактическим параметром должно быть выражение *вещественного* типа, и т.д.

Отметим существенное ограничение: с помощью оператор-функций можно программировать только вычисления, которые записываются в *одном* операторе присваивания.

Пример 5.5. Арифметические выражения, приведенные в разд. 5.2, можно вычислить с помощью операторов

```
f(x,y)=sqrt(x**2+y**2)
Z=f(a,1.)
W=f(b,c)
v=f(d+u,p)
```

6. ОПЕРАЦИИ СРАВНЕНИЯ. ЛОГИЧЕСКИЕ СВЯЗКИ И ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ. ОПЕРАТОРЫ ПЕРЕДАЧИ УПРАВЛЕНИЯ

6.1. Операции сравнения

В Фортране для обозначения понятий "больше", "меньше" и т.д. используются следующие четырехсимвольные обозначения:

Операция сравнения	Обозначение в Фортране
> (больше)	**.gt.** (от англ. greater than)
≥ (больше или равно)	**.ge.** (greater or equal)
< (меньше)	**.lt.** (less than)
≤ (меньше или равно)	**.le.** (less or equal)
= (равно)	**.eq.** (equal)
≠ (не равно)	**.ne.** (not equal)

Таким образом, условие $a \geq b$ по правилам Фортрана запишется так:

a **.ge.** b.

Сравнивать можно арифметические выражения и целого, и вещественного типа.

6.2. Логические связки и логические выражения

Арифметическое выражение (например, $a + b$) в зависимости от значений переменных a и b принимает некоторое числовое значение. Если же рассмотреть *сравнение* (например, a **.gt.** b), то в зависимости от того, чему равны a и b, оно будет или верным, или неверным. Таким образом, мы пришли к понятию *логического выражения*, принимающего одно из двух возможных значений: **.true.** (истинно) и **.false.** (ложно), в зависимости от значений входящих в него числовых переменных.

Логическое выражение может состоять из нескольких сравнений, соединенных между собой знаками логических операций, или логическими связками

Обозначения логических операций таковы:

Операция	Обозначение в Фортране

"и"	**.and.**
логическое "или"	**.or.**
логическое отрицание	**.not.**

Рассмотрим вначале два примера.

Пример 6.1. Пусть требуется вычислить величину $\sqrt{x} + \ln(1-x)$. Прежде, чем производить вычисления, нужно убедиться, что x принадлежит области определения данной функции, которая выделяется неравенствами $x \geq 0$ и $1-x > 0$. В данном случае необходимо, чтобы выполнялось и первое условие и второе. В таких ситуациях используется логическая операция **.and.**, проверка принадлежности x к области определения записывается теперь как проверка *одного* условия:

x **.ge.** 0 **.and.** *x* **.lt.** 1

дважды за- штрихованная часть плоскости. В общем случае правила истинности и ложности сравнений A и B, соединенных знаками логических операций, таковы

A	*B*	*A* **.and.** *B*	*A* **.or.** *B*	**.not.** *A*
T	T	T	T	F
F	T	F	T	T
T	F	F	T	F
F	F	F	F	T

Пример 6.2. Отрицанием выражения *y* **.le.** *x* является выражение not. *y* **.le.** *x* или *y* **.gt.** *x*

Таким образом, при отрицании одного сравнения знак неравенства меняется на противоположный.

Отметим, что одинаковые логические операции вычисляются слева направо; в сложных выражениях для указания порядка логических операций можно использовать скобки. Рассмотрим еще один пример, в котором встречаются все три логические операции.

Пример 6.3. Треугольник на плоскости *Oxy* (рис.6.1, *незаштрихованный*) с вершинами A (0; 0), B (1; 0), C (0; 1) можно задать системой неравенств

$$\begin{cases} x > 0, \\ y > 0, \\ y < 1 - x. \end{cases}$$

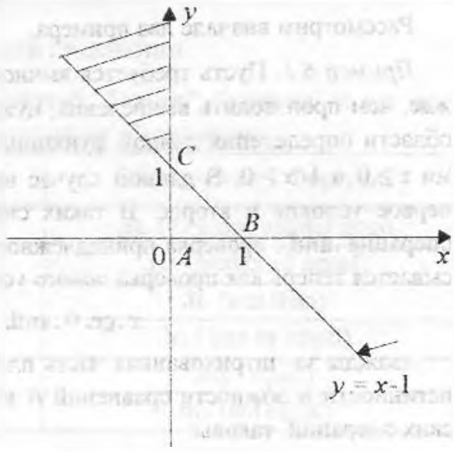

Рис.6.1

Поэтому точка (x, y) принадлежит треугольнику, если истинно выражение *x* **.ge.** 0 **.and.** *y* **.ge.** 0 **.and.** *x*+*y* **.le.** 1. Точка (x, y) не принадлежит треугольнику, если истинно выражение **.not.** (*x* **.ge.** 0 **.and.** *y* **.ge.** 0 **.and.** *x*+*y* **.le.** 1.) или, истинно выражение *x* **.lt.** 0 **.or.** *y* **.lt.** 0 **.or.** *x*+*y* **.gt.** 1

При отрицании взаимно заменяются связки **.and.** и **.or.**

Заметим,что заштрихованный на рис.6.2 угол (включая границы) выделяется условием *x* **.le.** 0 **.and.** *x*+*y* **.ge.** 1

6.3. Логические переменные

Логическое выражение принимает одно из двух значений: **.true.** или **.false.** Это значение можно запомнить с помощью *логической переменной.*

Имя логической переменной может начинаться с любой буквы; в начале программы все логические переменные должны быть описаны оператором **logical**, который имеет вид

logical список переменных.

Логическая переменная может встретиться в левой части оператора присваивания, тогда справа должно стоять логическое выражение, например, *a* = *x* .**ge.** 0. В этом случае переменная a получает то же значение (true или false), что и сравнение *x* .**ge.** 0. Допустимы присваивания вида:

```
b=.true.
d=.false.
c=a.and.x.lt.1,
```

т.е. логическую переменную можно использовать в логическом выражении (*a* .**and.** *x* .**lt.** 1). Заметим, что в итоге переменная *c* получила значение логического выражения *x* .**ge.** 0 .**and.** *x* .**lt.** 1.

Логические переменные удобно использовать, чтобы не переписывать несколько раз сложные логические выражения.

6.4. Оператор безусловной передачи управления GOTO

Операторы Фортрана обычно выполняются в том порядке, в каком они написаны, т.е. сверху вниз (это неявно предполагалось во всех рассмотренных ранее примерах).

Для изменения порядка выполнения служат *операторы передачи* управления, или операторы перехода. При этом операторы, которым передается управление, снабжаются *метками* (напомним, что метка – это число от 1 до 99999, записанное в позициях 1-5 любой строки; программист, выбирая метки, должен руководствоваться только тем, чтобы никакие два оператора не имели одинаковую метку).

Безусловная передача управления осуществляется с помощью оператора **goto**. Этот оператор имеет вид **goto** *n*, где *n* – метка оператора, которому передается управление (т.е. того оператора, который выполняется следующим). В программе оператор с меткой *n* обязателен. Оператор **goto** обычно используется в сочетании с другими операторами управления, либо для программирования выхода из цикла и его завершения (разд. 7).

6.5. Условный оператор IF

Оператор **if** позволяет выбрать (изменить) порядок дальнейшего выполнения в зависимости от указаний программиста (как правило, связанных с результатами предыдущих вычислений). Рассмотрим различные формы оператора условной передачи управления. В простейшем случае он имеет вид:

if (*L*) *B*

D (продолжение программы),

где *L* – логическое выражение, *B* – любой выполняемый оператор, кроме другого оператора **if** и оператора **do** (разд. 7) (т.е. *B* может быть одним из уже рассмотренных операторов: **read**, **print**, присваивания, **stop**, **goto**). Выполняется оператор **if** следующим образом:

1) проверяется выражение *L*;

2) если *L* истинно, выполняется оператор *B*;

3) если *L* ложно, то выполняется следующий по порядку оператор *D* (т.е. оператор *B* игнорируется).

Заметим, что если *B* не изменяет порядка выполнения операторов, то в случае истинности *L* после выполнения *B* оператор *D* *также будет выполнен*.

Пример 6.4. Рассмотрим выполнение фрагмента программы

```
   2|  |if(x.lt.1.4) x=x+0.1
    |  |x=x+0.2
```

Пусть к моменту выполнения оператора с меткой 2 переменная *x* имеет значение 1.5; тогда сравнение ложно, управление передается следующему оператору, и x увеличивается на 0.2; итак, *x* получает значение 1.7. Если же переменная *x* имела значение 1.3, то сравнение истинно, *x* увеличивается на 0.1, а затем управление также передается следующему оператору и *x* увеличивается еще на 0.2; в итоге *x* получает значение 1.6.

Заметим, что оператор *B*, выполняющийся в случае истинности *L*, может быть *только один*. Больше возможностей предоставляет другая форма условного оператора управления, *структурный оператор* **if**, общий вид которого

if (*L*) **then**

. . . . . .

else

. . . . . .

end if

D (продолжение программы).

При этом строка **else** может отсутствовать, а может иметь форму

else if(*L1*) **then**

Выполняется структурный оператор **if** следующим образом:

1) проверяется выражение *L*;

2) если L истинно, выполняются операторы, записанные между **if** и **else**, после чего (если не было другой передачи управления) выполняется оператор D;

3) если L ложно, то выполняются операторы, записанные между **else** и **end if**, после чего (если не было другой передачи управления) выполняется оператор D; если **else** отсутствует, то сразу выполняется оператор D.

Заметим, что каждый оператор **if**(L) **then** (но не **else if**... **then**!) должен заканчиваться строкой **end if**. Передавать управление внутрь структуры **if** ... **end if** нельзя

Пример 6.5. Рассмотрим выполнение фрагмента программы

```
2   if(x.lt.1.4) then
    x=x+0.1
    y=2.*x+3.5
    end if
    x=x+0.2
```

Пусть к моменту выполнения оператора с меткой 2 переменная x имеет значение 1.5; тогда сравнение ложно, управление передается следующему за **end if** оператору, и x увеличивается на 0.2; итак, x получает значение 1.7, а переменная y значения не получает. Если же переменная x имела значение 1.3, то сравнение истинно, x увеличивается на 0.1, переменная y получает значение 6.3 затем управление также передается следующему за **end if** оператору и x увеличивается еще на 0.2; в итоге x получает значение 1.6.

Пример 6.6. Рассмотрим следующий фрагмент программы

```
2   if(x.lt.1.4) then
    x=x+0.1
    y=2.*x+3.5
    else
    x=x+0.2
    y=2.*x+3.5
    end if
    z=x+y
```

Пусть к моменту выполнения оператора с меткой 2 переменная x имеет значение 1.5; тогда сравнение ложно, управление передастся следующему за **else** оператору, и x увеличивается на 0.2, т.е. x получает значение 1.7, а переменная y значение 6.9. Затем переход на продолжение программы, и переменная z получает значение 8.6. Если же переменная x имела значение

1.3, то сравнение истинно, x увеличивается на 0.1, переменная y получает значение 6.3 затем управление также передается следующему за **end if** оператору, и переменная z получает значение 7.7.

Заметим, что структурный оператор **if** позволяет выполнять несколько операторов как в случае истинного, так и ложного выражения L.

При использовании **else if**(L) **then** следует иметь в виду, что дальнейшие строки аналогичны соответствующим строкам в операторе **if** ... **then.**

Пример 6.7. Пусть к началу выполнения следующего фрагмента программы переменная x имеет значение 1.5.

```
if(x.lt.1.4) then
x=x+0.1
y=2.*x
else if(x.gt.2.) then
x=x+0.2
y=2.*x
else
x=x+0.3
y=2.*x
end if
z=x+y
```

Так как условие x **.lt.** 1.4 ложно, то проверяется условие x **.gt.** 2, что тоже ложно; поэтому x увеличивается на 0.3 и y получает значение 3.6, а z – значение 5.4. Если строки

```
else
x=x+0.3
y=2.*x
```

отсутствуют, то сразу должен выполняться оператор $z = x+y$; но в пределах данного фрагмента значение y не определяется, поэтому значения z указать нельзя.

Рекомендуем читателю проверить результаты работы, например, при $x = 1.3$ и $x = 3$.

Пример 6.8. Проверить, принадлежит ли точка с вводимыми координатами x, y области, ограниченной линиями $y = x^2$, $y = 3–x^2$ (рис.6.3). Если да, то выполнить действия $u = x+y$, $v = 0$ и выдать сообщение о попадании. В противном случае положить $u = 0$, $v = x–y$ и напечатать "промах". В обоих случаях вывести на печать u и v.

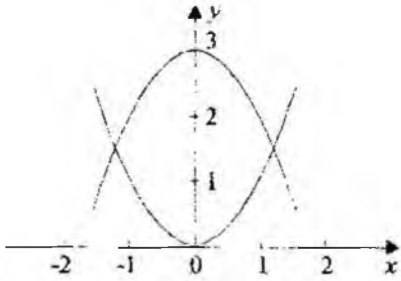

Рис.6.3

```
logical d,d1,d2
print *,'Введите координаты точки'
read *,x,y
d1=y.ge.x**2
d2=y.le.3-x**2
d=d1.and.d2
if(d) then
u=x+y
v=0
print *,'промах'
else
u=0
v=x-y
print *,'попадание'
endif
print *,'u=',u,' v=',v
print *,'Значение логич. выражения', d
end
```

Конечно, здесь можно было бы обойтись и без логических переменных и написать **if** (*y* **.ge.** *x***2. .and.** *y* **.le.** 3–*x***2), но польза от их применения очевидна

6.6. Вычисляемый оператор GOTO

Этот оператор позволяет организовать переход на одну из меток своего списка в зависимости от значения некоторой переменной.

Его общий вид:

goto (список меток) переменная

Управление передается на метку, место которой равно значению переменной.

Пример 6.9.

```
m=2
goto(2,36,7) m
```

Управление будет передано оператору с меткой 36, так как она стоит на 2-ом месте.

Оператор, следующий за вычисляемым **goto**, должен иметь метку.

6.7. Передача управления из оператора READ

При чтении информации из файла могут возникнуть некоторые трудности, связанные с тем, что количество элементов в файле может быть неизвестным, и в то же время требуется прочесть их все.

Эффективно решить эту задачу можно, разместив в операторе **read** управляющую информацию. Соответствующая конструкция имеет вид

read (n, *, end = метка).

Тогда, дойдя до конца файла, компьютер перейдет к указанной метке.

Аналогично обрабатывается ситуация ошибки при чтении из файла:

read (n, *, err = метка).

7. ПРОСТЕЙШИЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ. НЕКОТОРЫЕ ЧАСТО ВСТРЕЧАЮЩИЕСЯ АЛГОРИТМЫ

7.1. Программирование линейных участков алгоритма

Операторы Фортрана, соответствующие линейному участку алгоритма, записываются друг за другом. Примеры линейных участков алгоритмов приведены в разделе 4.6.

7.2. Программирование разветвлений

Общая схема программирования разветвления:

if (проверка) **then**
операторы, выполняемые
в случае "истина"
else
операторы, выполняемые
в случае "ложь"

end if

Пример 7.1. Составить программу нахождения максимального из двух чисел; найденный максимум присвоить переменной z и вывести на печать.

```
read *,x,y
if(x.ge.y) then
z=x
else
z=y
end if
print *,z
end
```

По тексту программы примера 7.1 легко понять, что она делает. Однако для этой задачи можно дать и такое решение:

```
z=x
if(y.gt.z) z=y
```

Пользуясь вторым способом, запишем содержательную часть программы для выбора максимального из трех чисел x, y, t:

```
z=x
if(y.gt.z) z=y
if(t.gt.z) z=t
```

Здесь такое использование условного оператора оказалось удобным, так как в случае истинности должен выполняться только один оператор. Если это не так, применяется общая схема.

Пример 7.2. Выбрать наибольшее и наименьшее из значений переменных x, y и вычислить $z = \max\{x, y\}+2\min\{x, y\}$.

```
read *,x,y
if(x.gt.y) then
u=x
v=y
else
u=y
v=x
end if
z=u+2.*v
print *,z
end
```

Общая схема несколько упрощается, если одна из ветвей пустая. При этом надо так сформулировать условие, чтобы пустой оказалась ветвь "ложь" (при этом не будет **else**).

Пример 7.3. Для заданного x вычислить и напечатать значение $y = \mathrm{sqrt}(x) + \ln(1-x)$. Вычисление y и печать нужно производить только, если x принадлежит области допустимых значений, т.е. истинно выражение x **.ge.** 0 **.and.** x **.lt.** 1. (см. пример 6.1).

```
      read *,x
      if (x.ge.0.and.x.lt.1.) then
      y=sqrt(x)+alog(1-x)
      print *,y
      end if
      end
```

7.3. Программирование циклов

Пример 7.4 (организация цикла) Составить программу, которая вычисляет значения функции $y = 1/x$ при $x = 1.1, 1.2, \ldots, 2.1$ и печатает значения аргумента и функции.

При программировании цикла выделяются операторы: начального задания параметра цикла ($x = 1.1$), изменения параметра цикла ($x = x+0.1$), проверка момента выхода из цикла (**if**). В разд. 8 эти три оператора будут заменены одним оператором **do**.

```
      x=1.1
    1 y=1./x
      print *,x,y
      x=x+0.1
      if(x.le.2.11) goto 1
      end
```

Замечание. Для сравнения с x выбрано число 2.11, поскольку из-за ошибок округления число 2.1 можно "проскочить".

Проверку иногда полезно производить вначале, сразу после задания параметра цикла; тогда программа примет вид:

```
      x=1.1
    1 if(x.le.2.11)then
      y=1./x
```

```
      print *,x,y
      x=x+0.1
      goto 1
      end if
      end
```

Пример 7.5 (накопление суммы). Составить программу вычисления суммы $s = 1+2+3+...+n$.

```
      read *,n
      s=0.
      k=1
2     s=s+k
      k=k+1
      if(k.le.n) goto 2
      print *,s
      end
```

При каждом прохождении цикла к содержимому ячейки *s* прибавляется очередное слагаемое; поэтому до первого выполнения цикла переменной *s* должно быть присвоено нулевое значение.

Пример 7.6 (вычисление факториала). Составить программу вычисления величины $n! = 1*2*3*...*n$.

```
      read *,n
      f=1.
c     Здесь накапливается не сумма, а произве-
c     дение, поэтому первоначально переменной f
c     присвоено значение 1.
      k=1
1     f=f*k
      k=k+1
      if(k.le.n) goto 1
      printf *,f
      end
```

Иногда этот же прием используется для вычисления степени *x**n*.

```
      y=1.
      k=1
1     y=y*x
      k=k+1
      if(k.le.n) goto 1
```

Пример 7.7 (переключатель). Составить программу вычисления суммы $q = 1^2 - 2^2 + 3^2 - 4^2 \ldots + 99^2 - 100^2$.

Отличие этой задачи от примера 7.5 состоит в том, что слагаемые попеременно то добавляются, то вычитаются. Переключатель ±1 можно реализовать так: если некоторой переменной *z перед циклом* присвоить значение 1., а *в цикл вставить оператор* $z = -z$, то при каждом прохождении цикла содержимое ячейки *z* будет попеременно равно +1 или –1. С учетом этого программу можно составить так:

```
      q=0.
      k=1
      z=1.
2     q=q+z*k**2
      z=-z
      k=k+1
      if(k.le.100) goto 2
      print *,q
      end
```

Пример 7.8. Вычислить значение $\sin x$ с помощью конечного отрезка ряда: $\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^n x^{2n+1}}{(2n+1)!}$.

Заметим, что если некоторое слагаемое обозначить $v = \frac{x^k}{k!}$, то следующее можно получить, умножив v на $(-x)^2$ и разделив на $((k+1)*(k+2))$ (при этом $k = 1, 2, \ldots, 2n+1$). Отсюда получаем программу:

```
      read *,x,n
      s=0.
      k=2
      v=x
1     s=s+v
      v=-v*x**2/(k*(k+1))
```

```
k=k+2
if(k.le.2*n+1) goto 1
print *,s
end
```

В этом примере используются все описанные ранее приемы.

8. МАССИВЫ, ИХ ОПИСАНИЕ И РАЗМЕЩЕНИЕ В ПАМЯТИ. ОПЕРАТОРЫ ЦИКЛА DO И WHILE. ПРАВИЛА ИСПОЛЬЗОВАНИЯ ОПЕРАТОРОВ ЦИКЛА

8.1. Массивы

Ранее мы рассматривали *простые переменные*, используемые для вычисления скалярных величин.

Массив – это упорядоченный набор переменных, имеющий имя. Элемент массива называется также *индексированной переменной*. Индексы пишутся в скобках; если индексов несколько, то они разделяются запятыми.

Пример 8.1. Индексированные переменные $a(1)$, $a(2)$, $a(3)$, $a(4)$ образуют одномерный массив a. Переменные $b(1, 1)$, $b(2, 1)$, $b(1, 2)$, $b(2, 2)$, $b(1, 3)$, $b(2, 3)$ образуют двумерный массив b.

Одномерные и двумерные массивы аналогичны математическим понятиям вектора и матрицы; так, массив b представляет собой как бы матрицу из 2 строк и 3 столбцов.

Размерность массива не должна превышать 7 (т.е. элемент массива не может иметь более 7 индексов). Индексы всегда идут от начального до максимального значения подряд. Так, тройка элементов $a(1)$, $a(3)$, $a(4)$ массивом не является.

В качестве индексов могут использоваться константы, переменные и арифметические выражения. Обычно индексы – это выражения целого типа, но можно использовать выражения и вещественного типа.

Употреблять вещественные индексы надо осторожно, поскольку всякий раз дробная часть индекса будет отбрасываться. Например, операторы

```
x=1.2
a(3*x-1)=0
```

допустимы: значение 0 получит элемент $a(2)$ массива a, ибо $3*1.2 - 1 = 2.6$.

8.2. Описание массивов в программе

Каждый массив имеет размер, или длину: массив a из примера 8.1 состоит из 4 элементов, а массив b – из 6. Всякий массив, встречающийся в программе, должен быть *описан*. Причина этого проста: транслятор должен знать заранее размер памяти, отводимой для всех элементов данного массива, а это зависит от его длины. Описание массивов производится, например, с помощью неисполняемого оператора **dimension**

Например:

dimension имя массива (границы значений индексов), ...

Если массивов несколько, то их описания разделяются запятыми; если указана лишь одна граница, то это – максимальное значение индекса, нижняя граница которого равна 1.

Например, оператор

```
dimension a(4),b(2,3)
```

описывает массивы из примера 8.1. Оператор

```
dimension a(4),c(4),d(3,2)
```

описывает два одномерных массива a и c и двумерный массив d, состоящий, как и массив b, из 6 элементов, но *расположенных по-иному*. Оператор

```
dimension a(-1:0,2)
```

описывает двумерный массив a, состоящий из 4 элементов: $a(-1, 1)$, $a(0, 1)$, $a(-1, 2)$, $a(0, 2)$, т.е. первый индекс изменяется от –1 до 0, а второй – от 1 (так как нижняя граница не указана) до 2.

> Операторов *описания массивов* может быть несколько; их следует ставить в начале программы, раньше *оператор-функций* (если они есть). Каждый массив должен быть описан один раз.

В качестве значений индексов в операторе **dimension** разрешается использовать *только целые константы*, в том числе *именованные* (см. разд. 4.10). Несущественное исключение из этого правила изложено в разд. 12.

Это не всегда удобно. Например, в некоторой задаче надо найти корни заданного уравнения и занести их в массив, но ведь заранее неизвестно, сколько корней удастся найти. В тех случаях, когда заранее неизвест-

но, сколько элементов массива будет использоваться, приходится в операторе **real** описывать указанный массив "с запасом".

8.3. Размещение массивов в памяти. Ввод и вывод массива по имени. Присваивание массивам начальных значений

Память ЭВМ устроена линейно (как одна огромная строка); для массива выделяется линейный участок памяти, в котором подряд располагаются его элементы. Так, в одномерном массиве а элементы расположены следующим образом: $a(1)$, $a(2)$, $a(3)$ и т.д.

Что касается многомерных массивов, то они располагаются в памяти так, что быстрее всего изменяется первый индекс, затем второй и т.д. *В двумерном случае это не согласуется с привычным упорядочением по строкам, когда быстрее меняется второй индекс.*

Пример 8.2. Массив $b(1{:}2, 1{:}3)$ располагается в памяти так, как указано в примере 8.1. Массив $g(1{:}2, 1{:}2, 1{:}2)$ из 8 элементов располагается в следующем порядке: $g(1, 1, 1)$, $g(2, 1, 1)$, $g(1, 2, 1)$, $g(2, 2, 1)$, $g(1, 1, 2)$, $g(2, 1, 2)$, $g(1, 2, 2)$, $g(2, 2, 2)$.

Отдельные элементы массива (т.е. переменные с индексами) можно вводить и выводить так же, как и простые переменные. Например, оператор

```
print *,k,a(1),a(2),a(3),a(4),b(1,2)
```

напечатает значения переменной k, четырех элементов одномерного массива a и элемента $b(1, 2)$ двумерного массива b.

Можно ввести и вывести массив целиком. Это делается при помощи операторов вида

read *, имя массива
print *, имя массива

При этом ввод и вывод элементов массива производится в том порядке, в котором элементы расположены в памяти. Будет введено (выведено) столько элементов, сколько их было объявлено в описании массива.

Пример 8.3. Рассмотрим программу

```
dimension b(2,3)
read *,b
end
```

Пусть для ввода задана строка

```
1. 2. 3. 4. 5. 6.
```

При выполнении оператора **read** элементам массива *b* будут присвоены значения: *b*(1, 1) = 1., *b*(2, 1) = 2., *b*(1, 2) = 3., *b*(2, 2) = 4., *b*(1, 3) = 5, *b*(2, 3) = 6.

Если несколько идущих подряд вводимых величин (например, элементов массива) имеют одно и то же значение, то в строке данных можно использовать повторитель. Он имеет вид

k*d,

где k – целая константа – число повторений, d – значение вводимой величины.

Например, если всем элементам одномерного массива *a* длиной 4 нужно присвоить значение 1.5, то строка данных для оператора

```
read *,a
```

может иметь вид 4*1.5. В разд. 13 изложено, как можно вводить (выводить) массивы в ином порядке и не полностью.

8.4. Оператор цикла do

Этот оператор позволяет несколько раз повторять участок программы при меняющемся значении целой или вещественной переменной, которая называется *счетчиком*, или *параметром* цикла.

Оператор **do** имеет вид:

1) **do** метка *i* = *n1*, *n2*, *n3*

или

2) **do** *i* = *n1*, *n2*, *n3*.

Здесь *i* – простая переменная (параметр цикла); *n1*, *n2*, *n3* – простые переменные или константы или арифметические выражения, которые обозначают:

n1 – начальное значение параметра цикла;

n2 – конечное значение параметра цикла;

n3 – шаг изменения значения параметра цикла.

Правая часть оператора цикла показывает, *сколько раз* выполняется цикл, а именно, цикл выполняется при *i* = *n1*, *i* = *n1*+*n3*, *i* = *n1*+2**n3* и т.д., пока $|i-n1| \le |n2-n1|$. Если шаг *n3* равен 1, то его можно не писать; *n3* = 0 не допускается. Левая часть первой формы оператора цикла показывает, какая *часть* программы повторяется: при заданных значениях параметра цикла выполняются операторы, начиная со следующего за оператором

do; последним выполнятся оператор с указанной в **do** меткой. При использовании оператора **do** во второй форме после последнего оператора необходимо написать **end do**. Эта повторяющаяся часть программы иногда называется телом цикла.

Пример 8.4. Рассмотрим использование оператора **do** в программе примера 7.6 (вычисление факториала):

```
      read *,n
      f=1
      do 7 k=1,n
    7 f=f*k
      print *,f
      end
```

Оператор **do** заменил три оператора: первоначальное присвоение ($k = 1$), увеличение счетчика ($k = k+1$), проверку условия (**if**).

Пример 8.5. Воспользуемся второй формой оператора цикла для вычисления значений функции $y = 1/x$ при $x = 1.1, 1.2, \ldots, 2.1$ и печати значений аргумента и функции (см. пример 7.4):

```
      do x=1.1,2.1,0.1
      y=1./x
      print *,x,y
      end do
      end
```

Заметим, что в версиях Фортрана после 95 года допускаются только *целые* параметры циклов.

8.5. Правила использования оператора DO

1. Если оператор **do** оказывается в теле другого оператора цикла, то его тело должно полностью содержаться внутри этого "внешнего" цикла.

2. Если структурный оператор **if** оказывается внутри цикла, то связанный с ним оператор **end if** также должен находиться внутри этого цикла (и наоборот, если оператор **do** оказывается внутри блока **if**, то тело цикла должно целиком содержаться внутри этого блока).

3. Если шаг изменения параметра цикла положителен, а начальное значение счетчика больше конечного, то цикл не выполняется *ни разу* (то же относится к случаю, когда при отрицательном шаге начальное значение счетчика меньше конечного).

4. Не разрешается изменять внутри цикла значение его параметра. После выхода из цикла сохраняется последнее значение параметра, увеличенное на n3.

В данном примере будет напечатано число 13.

```
      do 4 k=1,12
      . . . . . . . . . . . .
      . . . . . . . . . . . .
4
      Print *,k
6
```

5. Последним оператором цикла, имеющим метку, могут быть только операторы:

а) присваивания, **read**, **print**, **call** (см. разд. 14), **continue**;

б) логический оператор **if**, в правой части которого стоит один из операторов, перечисленных в *а*).

6. В современной литературе чаще используется *вторая* форма записи оператора цикла, т.е. с окончанием **end do**.

8.6. Фиктивный (пустой) оператор CONTINUE

Этот оператор не производит никаких действий; однако он может иметь метку и быть последним оператором цикла. Можно порекомендовать ставить метку всегда на *этот* оператор. Это уменьшает вероятность ошибок при разработке и отладке программы и снимает все вопросы о последних операторах цикла.

Пример 8.6. Задан одномерный массив *a* из 4 элементов. Все положительные элементы этого массива напечатать, а затем увеличить на 1.3.

Если все элементы массива неположительны, то в программе нет выполняемых операторов, поэтому нельзя поставить метку для обозначения конца цикла.

Решение 1. Используя оператор **continue**, запишем программу:

```
      real a(4)
      read *,a
      do 2 k=1,4
      if(a(k).le.0) goto 2
      print *,a(k)
      a(k)=a(k)+1.3
2     continue
```

```
end do
end
```

Приведенное решение характерно для старых версий Фортрана, не имевших структурного оператора **if**...**end if**. Его использование в сочетании с оператором **do**...**end do** позволяет запрограммировать практически любой циклический алгоритм без использования меток. При этом повышается наглядность программы. Такой способ программирования циклов принят в современной литературе. Приводим соответствующее решение этого же примера.

Решение 2.

```
real a(4)
read *,a
do k=1,4
if(a(k).gt.0) then
print *,a(k)
a(k)=a(k)+1.3
end if
end do
end
```

Отметим, что требование минимизации количества меток иногда представляется чересчур категоричным; к примеру, программа с большим количеством циклов при прочих равных условиях будет, по мнению авторов, более наглядна, если заканчивать циклы оператором **continue** с меткой.

8.7. Оператор цикла WHILE

Оператор цикла **do** можно использовать, когда заранее известно число повторений цикла. Однако бывают случаи, когда это не так. В примере 7.8 при вычислении $\sin x$ мы задавали количество членов ряда, поэтому легко переписать эту программу с оператором **do**. Если проводить вычисления до тех пор, пока очередной член ряда не станет достаточно малым, то проверка окончания цикла по счетчику затруднительна (очевидно, что число членов ряда в этом случае зависит от значения x). Эту задачу можно решить с помощью оператора **while**, который имеет вид:

while (логическое выражение).

Пока логическое выражение истинно, выполняются операторы, начиная со следующего за оператором **while** и кончая оператором **end while**; если значение логического выражения "ложь", то управление передается оператору, следующему за **end while.**

Пример 8.7. Вычислить значение sin *x*, пользуясь приведенным в примере 7.8 представлением sin *x*; вычисления закончить, когда абсолютная величина очередного члена разложения станет меньше 0.001.

Решение 1.

```
      read *,x
      s=0.
      k=2
      v=x
      while(abs(v).lt.0.001)
      s=s+v
      v=-v*x**2/(k*(k+1))
      k=k+2
      end while
      print *,s
      end
```

Ошибки при использовании оператора **while** часто приводят к зацикливанию, поэтому нужно следить за тем, чтобы переменные в условии этого оператора правильно менялись бы в процессе работы цикла.

Решение 2 (без использования оператора **while**).

```
      read *,x
      s=0.
      k=2
      v=x
      do k=2,10000
      if(abs(v).lt.0.001) goto 5
      s=s+v
      v=-v*x**2/(k*(k+1))
      end do
5     continue
      print *,s
```

```
    end
```

В этой программе максимальное количество повторений цикла не превосходит 10000, что предотвращает возможность зацикливания. Нормальной ситуацией здесь является "досрочный" выход из цикла (см. также раздел 8.8).

8.8. Передача управления в цикле

На рис.8.1 цикл изображен прямоугольником. Сплошными стрелками отмечены допустимые передачи управления, а штриховыми – недопустимые.

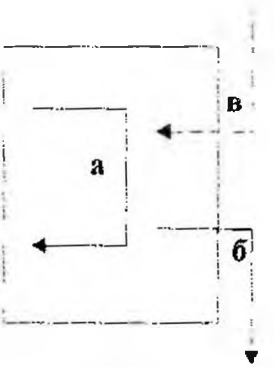

Рис.8.1

Разрешены передачи управления внутри цикла (*а*) и из цикла на оператор вне его области (*б*). Запрещается передача управления внутрь цикла (*в*).

Внутри области цикла могут быть другие операторы **do** (или **while**). Тогда последний оператор внутреннего цикла должен стоять раньше последнего оператора внешнего; при этом помеченный последний оператор может быть общим для обоих циклов **do**, а оператор **end do** (**end while**) может заканчивать только один цикл.

Пример 8.8. Вложенные циклы.

а) цикл по *m* лежит внутри цикла по *k*:

б) цикл по *i* и цикл по *j* имеют общий по-

следний оператор (не рекомендуется):

```
      do 1 k=1,n1          do k=1,n1          do 3 i=1,m
      . . . . . .          . . . . . .        . . . . . .
       do 2 m=1,n2          .                  do 3 j=1,n
        . . . . . .          do m=1,n2          . . . . . .
  2     . . . . . .            . . . . . .      . . . . . .
                              .
                               end do
  1   . . . . . .            end          do    . . . . . .
                              3
```

При "досрочном" выходе из цикла последнее значение его параметра сохраняется.

Например, операторы

```
      do k=1,12
      if(k.eq.11) goto 6
      . . . . . . . . . . . .
      end do
  6   print *,k
```

напечатают число 11.

9. НЕКОТОРЫЕ ЦИКЛИЧЕСКИЕ АЛГОРИТМЫ

9.1. Операторы цикла в задачах без массивов

Пример 9.1. Накопление суммы. Составим программу вычисления суммы $s = 1+2+3+ \ldots +n$ (см. пример 7.5).

```
read *,n
s=0
do k=1,n
s=s+k
end do
print *,s
end
```

Пример 9.2. То же для суммы $s = 1+3+5+ \ldots +1001$.

```
s=0
do 5 k=1,1001,2
s=s+k
end do
end
```

Пример 9.3. То же для суммы $q = 1^2 - 2^2+3^2 \ldots - 100^2$ (см. пример 7.7).

```
q=0
z=1
do k=1,100
q=q+z*k**2
z=-z
end do
print *,q
end
```

Пример 9.4. Табулирование функции. В примерах 7.4, 8.6 составляется таблица значений функции $y = 1/x$ для $x = 1.1, 1.2, \ldots, 2.1$. Составим программу вычисления значений функции $y = \ln x/x$ для $x = 2.5, 2.6, \ldots, 9.5$.

```
do x=2.5,9.51,0.1
y=log(x)/x
print *,x,y
end do
end
```

Заметим, что конечное значение параметра цикла взято "с запасом". Это связано с тем, что вещественные числа представляются в ЭВМ приближенно, и, если последнее значение параметра цикла окажется "чуть больше", чем 9.5, то соответствующее значение y вычислено не будет.

Пример 9.5. Итерационный процесс. Для приближенного вычисления $\sqrt{A}$ можно использовать тот факт, что последовательность

$$x_0 = A,\ x_1 = 0.5(x_0 + A/x_0),\ x_2 = 0.5(x_1 + A/x_1),\ \ldots,\ x_{n+1} = 0.5(x_n + A/x_n),\ \ldots$$

сходится к $\sqrt{A}$ при $n \to \infty$. Программа должна вычислить несколько приближений x_k, вычисления следует прекратить, когда разность между двумя соседними приближениями станет меньше заданной точности: $|x_k - x_{k-1}| < \varepsilon$.

Заметим, что номера приближений нам не нужны: на очередном шаге используется предыдущее (x_{k-1}) и новое (x_k) приближения. На следующем шаге приближение x_k становится "предыдущим" и вычисляется приближение x_{k+1} и т.д.

```
      read *,A,eps,n
      x=A
      do k=1,n
      y=0.5*(x+A/x)
      v=abs(y-x)
      if(v.lt.eps) goto 44
      x=y
      end do
44    print *,'y=',y,' v=',v
      stop
      end
```

В программе через *x* обозначено "предыдущее" приближение, через *y* – "новое"; *n* – максимальное число приближений, *eps* – заданная точность. К оператору с меткой 44 мы попадем в двух случаях: когда достигнута требуемая точность (*eps*) или выполнены все *n* приближений, но точность не достигнута.

Программа печатает последнее найденное приближение *y* и модуль разности между последним и предпоследним приближениями *v*. В данном случае одна из ветвей алгоритма выходит из цикла до его завершения.

9.2. Оператор цикла в задачах с одномерными массивами

Проще всего написать программу в случае, когда внутри цикла нет разветвлений.

Пример 9.6. Заданы массивы $x(100)$ и $y(100)$. Найти

$$s = x(1)*y(1)+x(2)*y(2)+...+x(100)*y(100).$$

```
dimension x(100),y(100)
read *,x,y
s=0
do i=1,100
s=s+x(i)*y(i)
end do
print *,'s=',s
end
```

В примерах 9.7-9.10 разветвление целиком лежит в цикле, причем одна из ветвей пустая.

Пример 9.7. Найти максимальный элемент массива $a(1:100)$.

```
dimension a(100)
read *,a
w=-1.e20
do i=1,100
if(a(i).gt.w) w=a(i)
end do
print *,'max=',w
end
```

Здесь значение w на i-ом шаге цикла равно значению максимального из уже просмотренных i-1 элемента. Начальное значение w играет роль компьютерной "минус бесконечности". Иногда, особенно в учебной литературе, полагают w равным $a(1)$. При этом, если, например, элементы массива сложно вычисляются, то такой метод приведет программиста к необходимости писать лишние операторы, а компьютер – к лишним вычислениям. Кроме того, неестественным будет "нарушение симметрии" (перебор массива со 2-го элемента). Если все же есть сомнения в применимости такого алгоритма в каких-то случаях, можно, конечно, взять w равным *первому элементу*, как в следующем примере.

Пример 9.8. Найти номер максимального элемента массива $b(1:90)$.

```
parameter (n=90)
dimension b(n)
read *,b
w=b(1)
do i=1,n
if(w.lt.b(i)) then
k=i
w=b(i)
end if

end do
print *,'номер максимального элемента ',k
end
```

Пример 9.9. Найти сумму положительных элементов массива $c(1{:}80)$.

```
dimension c(80)
read *,c
s=0
do i=1,80
if(c(i).gt.0) s=s+a(i)
end do
print *,'s=',s
end
```

Пример 9.10. Найти число положительных элементов массива $d(1{:}70)$.

```
dimension d(70)
read *,d
kp=0
do i=1,70
if(d(i).gt.0) kp=kp+1
end do
print *,'kp=',kp
end
```

В примере 9.11 разветвление расположено в цикле, причем есть и ветвь "да", и ветвь "нет".

Пример 9.11. Все положительные элементы массива $r(1{:}50)$ умножить на 2, а все отрицательные – разделить на 2. Измененный массив r напечатать.

```
dimension r(50)
read *,r
do i=1,50
if(r(i).gt.0) then
r(i)=r(i)*2.0.
else
r(i)=r(i)/2.0
end if
end do
print *,'измененный массив:',r
end
```

В следующем примере благодаря регулярному чередованию номеров удается обойтись без разветвлений.

Пример 9.12. Все элементы массива $g(1{:}40)$ с четными номерами увеличить на 1.5, а все элементы с нечетными номерами уменьшить на 2.5. Измененный массив g напечатать.

```
dimension g(40)
read *,g
do i=1,39,2
g(i)=g(i)-2.5
g(i)=g(i)+1.5
end do
print *,'измененный массив',g
end
```

Наконец, в примерах 9.13, 9.14 одна из ветвей выходит из цикла (т.е. выполнение цикла в этом случае прекращается до того, как просмотрен весь массив). Подобное уже встречалось в примере 9.5.

Пример 9.13. Найти и напечатать первый по порядку положительный элемент массива $p(1{:}24)$.

Здесь возможен случай, когда положительных элементов нет. Тогда после просмотра всего массива выполнение программы прекращается (без вывода результатов).

```
      dimension p(24)
10    read *,p
      do m=1,24
      if(p(m).gt.0) goto 10
      end do
      goto 15
      print *,p(m)
15    stop
      end
```

Рекомендуется, чтобы оператор stop был в программе единственным.

Пример 9.14. Найти и напечатать сумму первых N положительных элементов массива $q(1{:}36)$.

```
      dimension q(36)
      read *,q,N
      s=0
      k=0
      do j=1,36
      if(q(j).ge.0) then
      s=s+q(j)
      k=k+1
      end if
      if(k.ge.N) goto 44
      end do
44    continue
      print *,'s=',s,' k=',k
      end
```

Многие более сложные задачи являются комбинацией простых алгоритмов, подобных приведенным в примерах 9.6-9.14. Не нужно стараться, чтобы программа все выполняла за "один проход" по массиву; иногда это и невозможно.

Пример 9.15. Найти среднее арифметическое положительных элементов массива $h(1{:}30)$; после этого все отрицательные элементы заменить найденным средним.

Здесь возможен случай, когда положительных элементов нет; тогда n равно нулю. Об этой ситуации выдается сообщение, и вторая часть программы не выполняется.

```
      dimension h(30)
      read *,h
      s=0
      n=0
      do i=1,30
      if(h(i).gt.0) then
      s=s+h(i)
      n=n+1
      end if
      end do
      if(n.eq.0) then
      print *,'положительных элементов нет'
      goto 15
      end if
c     продолжение программы
      s=s/n
      do i=1,30
      if(h(i).lt.0) h(i)=s
      end do
      print *,h
   15 stop
      end
```

Из более сложных задач рассмотрим алгоритм упорядочения одномерного массива.

Пример 9.16. Упорядочить массив $e(1{:}60)$ в порядке возрастания, т.е. так переставить его элементы, чтобы стало

$$e(1) \le e(2) \le e(3) \le \ldots \le e(60).$$

Идея алгоритма состоит в том, что если два соседних элемента $e(i-1)$ и $e(i)$ не упорядочены, т.е. $e(i) > e(i-1)$, то их надо поменять местами. Это делается следующим линейным алгоритмом:

```
      v=e(i-1)
      e(i-1)=e(i)
      e(i)=v
```

Просмотрев все пары соседей, от $e(1)$-$e(2)$ до $e(59)$-$e(60)$ (т.е. от $i = 2$ до $i = 60$), мы по крайней мере передвинем максимальный элемент массива в ячейку $e(60)$. Теперь надо упорядочивать массив от $e(1)$ до $e(59)$ и т.д. Поэтому просмотр надо повторить 59 раз (массив из одного элемента упорядочивать не надо):

```
dimension e(60)
read *,e
do k=1,59
do i=2,60
if(e(i-1).gt.e(i)) then
v=e(i-1)
e(i-1)=e(i)
e(i)=v
end if
end do
end do
print *,e
end
```

В приведенной программе просматривается весь массив, в том числе и уже упорядоченные "самые правые" элементы. Этого можно избежать (и добиться экономии вычислений), записав заголовки циклов так:

```
do k=1,59
do i=2,61-k
```

9.3. Операторы цикла в задачах с двумерными массивами

В примере 9.16 нам впервые встретились вложенные циклы: "цикл в цикле". В задачах на двумерные массивы вложенные циклы используются постоянно.

Наиболее простой случай – когда массив рассматривается целиком, причем порядок просмотра несущественен.

Пример 9.17. Найти максимальный элемент массива $a(1{:}20, 1{:}10)$.

Сравнивая эту программу с программой примера 9.7, видим, что вся разница – в очевидном вложении цикла во внешний.

```
dimension a(20,10)
read *,a
s=-1.e20
do i=1,20
do j=1,10
if(a(i,j).gt.s) s=a(i,j)
end do
end do
print *,s
stop
end
```

В более сложных ситуациях порядок просмотра может определяться условиями задачи.

Пример 9.18. Дан двумерный массив r(1:14, 1:11). Составить программу, которая находит сумму положительных элементов каждой строки, заполняет этими суммами массив v (1:14) и печатает те элементы массива v, которые не превосходят 12.5.

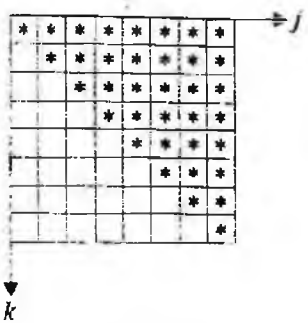

Рис.9.1

```
dimension r(14,10),v(14)
read *,r
do i=1,14
v(i)=0
do j=1,11
if(r(i,j).gt.0) v(i)=v(i)+r(i,j)
end do
if(v(i).le.12.5) print *,v(i)
end do
end
```

Наконец, отметим случай, когда двумерный массив просматривается не целиком.

Пример 9.19. Дан двумерный массив z(1:8,1:8). Среди элементов этого массива лежащих выше главной диагонали и на ней (заполненная звездочками область на рис.9.1), найти число отрицательных элементов.

Мы видим, что в данном случае индекс k меняется от 1 до 8; при фиксированном k индекс j меняется от k до 8.

Теперь мы знаем, как построить циклы; остальная часть программы почти такая же, как в примере 9.10.

```
dimension z(8,8)
read *,z
n=0
do k=1,8
do j=k,8
```

```
      if(z(k,j).lt.0)
>     n=n+1
      end do
      end do
      print *,z
      end
```

Можно изменить порядок просмотра массива *z* (не по строкам, а по столбцам); тогда содержательная часть программы примет вид:

```
      n=0
      do j=1,8
      do k=1,j
      if(z(k,j).lt.0) n=n+1
      end do
      end do
```

Для выделения нужной части массива *z* можно использовать возможности логического оператора **if**:

```
      dimension z(8,8)
      read *,z
      n=0
      do k=1,8
      do j=1,8
      if(j.ge.k.and.z(k,j).lt.0) n=n+1
      end do
      end do
      print *,z
      end
```

9.4. Неявные циклы

Дополнительные возможности дает использование *неявного цикла* ввода (вывода). Неявный цикл записывается прямо в операторе **read** или **print** и имеет вид:

(список переменных, $i = n1, n2, n3$),

Таким образом, оператор

```
      read *,(a(i),i=1,6)
```

введет первые шесть элементов одномерного массива *a* (независимо от того, какова длина этого массива, описанная оператором **dimension**).

Если заменить **read** на **print**, то компьютер напечатает 6 элементов *в одну строку*.

Заметим, что операторы ввода (вывода) можно заключать и в обычные циклы. Так, операторы

```
do i=1,6
read *,a(i)
end do
```

также вводят шесть первых элементов массива *a*. В этом случае оператор **read** * фактически выполняется шесть раз. При каждом выполнении данные берутся с *новой* строки: поэтому шесть введенных чисел должны быть записаны на шести строках. При замене **read** на **print** будет напечатано 6 элементов, каждый – с *новой* строки, т.е., 6 строк по одному лементу, что, очевидно, не всегда желательно.

Оператор

```
print c,(b(k),k=2,8,2)
```

напечатает значения следующих переменных: *b*(2), *c*, *b*(4), *c*, *b*(6), *c*, *b*(8), *c*, так что переменная *c* выводится 4 раза.

Конструкция *i* = *n*1, *n*2, *n*3 подчиняется тем же правилам, что и обычный оператор **do** (см. разд. 8).

Печать в неявном цикле ведется в одну строку (по умолчанию) с возможным переносом, если информация не умещается в строке.

Один оператор ввода (вывода) может содержать несколько неявных циклов, в том числе вложенных друг в друга.

Это позволяет вводить (выводить) многомерные, например, двумерные массивы не целиком и в нужном порядке.

Заметим, что чаще всего двумерный массив хранит элементы некоторой матрицы.

Для ввода первых *m* строк и *n* столбцов матрицы рекомендуется сочетание явного и неявного циклов:

```
Do i=1,m
```

Компьютер напечатает

k=¬39¬x=12.437

Обращаем внимание на пробел между апострофом и *x* в списке вывода, который воспроизводится в строке печати, предотвращая «слипание» 39 и *x*.

Если вместо запятой между описателями поставить обратный слэш /, то произойдет переход на новую строку.

Пример 10.3.

```
      |  |  print 99,'k=',k,' x=',x
   99 |  |  format(a,i3/a,f6.3)
      |  |
```

Будет напечатано

```
k=¬39
¬x=12.437
```

10.2. Вывод чисел с помощью описателя E

Описатель **f** удобен, когда речь идет о не очень больших и не очень маленьких (по абсолютной величине) числах, в других ситуациях нужно использовать описатели **e** или **g**.

Описатель **e** имеет вид: **e***w.d*, где *w* и *d* – целые константы без знака, причем желательно выбирать $w \geq d+7$. При выводе число печатается в *нормализованной форме* с экспонентой (см. разд. 4.4) и занимает *w* позиций на экране; при этом *d* позиций отводится на дробную часть.

Пример 10.4. Пусть $A=6.02\cdot10^{23}$, $h=6.626\cdot10^{-34}$. Тогда операторы

```
      |  |  Print 9,A,h
    9 |  |  format(e10.3,e12.4)
```

напечатают

¬0.602E+24¬¬0.6626E-33

10.3. Вывод чисел с помощью описателя G

совмещает достоинства описателей **f** и **e**. Его общий вид **g***w.d*, где как всегда *w* – общее число позиций.

Если есть возможность, то число представляется по формату

f(*w* – 4).*m*,4**x**,

причем *m* подбирается так, чтобы обеспечить выдачу *d значащих* цифр. Описание 4**x** означает 4 пробела (см. также разд. 11).

Если такой возможности нет, то число выводится в формате **e***w.d*.

Пример 10.5. По формату g10.3 число 12.745 будет выведено как

¬¬12.7¬¬¬¬

10.4. Форматный ввод из файла и вывод в файл

Оператор ввода из файла с номером *n* по формату с указанной меткой имеет вид

read (*n*, метка) список переменных.

Для печати в файл синтаксис аналогичен:

write (*n*, метка) список переменных

Отличие от бесформатного ввода-вывода (см. разд. 3.7 и 3.8) в том, что вместо звездочки после номера файла ставится метка оператора **format** (точнее – ссылка на нее).

10.5. Понятие о файлах прямого доступа

При работе с файлами часто возникает необходимость получить доступ программы к определенному месту файла для чтения или печати. Такую возможность обеспечивают файлы *прямого доступа*. При этом файл разбивается на нумерованные *записи*. Под записью можно понимать набор полей, описанных одним оператором **format**.

Изложим коротко на примере некоторые правила работы с такими файлами.

Пример 10.6. В результате работы данной программы в первой строке файла pet res будет напечатано число 4, во второй – число 3.

```
      open (8,file='pet1.res',access='dirt',
     >form='formatted',recl=10)
      a=3
      b=4
      write(8,fmt=900,rec=1)a
      write(8,fmt=900,rec=2)a
      write(8,fmt=900,rec=1)b
  900 format(5x,f6.3)
      End
```

Здесь direct (прямой) – значение переменной **access** (доступ), formatted (форматный) – значение переменной **form**, 10 – длина записи (количество символов) **recl**.

В операторе печати **fmt** – метка оператора **format**, **rec** – номер записи.

В операторах прямого доступа ввод и вывод всегда форматные.

10.6. Форматный ввод-вывод массивов

Сначала надо решить, какие описатели будут использоваться. Это зависит от порядка исходных данных, результатов и их точности. В этом разделе для определенности полагаем, что (с учетом пробелов между числами) для целых переменных достаточно описателя i6, а для вещественных – f7.1.

От того, какой повторитель поставить перед описателями, зависит, сколько чисел уместится на одной строке. Ее длина при вводе и выводе на экран – 80 символов.

Пример 10.7.

```
      print *,(a(i),i=1,6)
    1 format (1x,10f7.1)
```

Здесь формат выбран «с запасом» – из 10 полей используются всего 6.

Ввод и вывод должны быть наглядными. Например, двумерные массивы небольшого размера (до 10 элементов в строке) должны вводиться и выводиться так, чтобы одной строке данных (результатов) соответствовала ровно одна строка массива.

Сложнее обстоит дело, если размеры вводимой (выводимой) части массива могут меняться. При вводе в этом случае лучше пользоваться оператором **read** *; тогда в строке данных можно записать столько чисел, сколько нужно. Выводить же массивы и их части лучше с форматом.

Для двумерных массивов, как уже рекомендовалось, удобно сочетать явный и неявный циклы. Дадим пример форматного вывода части массива *a*(50, 100).

Пример 10.8.

```
      do i=1,m
      print 900,(a(i,j),j=1,n)
      end do
  900 format(1x,10f7.1)
```

Каждая новая строка массива будет печататься с новой строки. Если строка массива не поместится в строке печати, то произойдет автоматический переход на следующую строку, и формат начнет просматриваться заново.

Можно применить и вложенные неявные циклы (см. также разд. 10.4).

Пример 10.9.

```
      print 900,((a(i,j),j=i,n),i=1,m)
  900 format(1x,10f7.1)
```

Эта запись компактнее, но требует подбора формата (приведенный пример годится для n=10;при других n строки будут перепутаны).

11. ПОДПРОГРАММЫ

11.1. Назначение подпрограмм. Типы подпрограмм в Фортране

Одно из достоинств Фортрана - возможность разделения сложных программ на отдельные блоки, которые программируются независимо. Такие блоки называются подпрограммами. Алгоритм решения задачи может состоять из *основной* программы и вызываемых ею подпрограмм.

В языке Фортран имеется два типа подпрограмм: подпрограммы-функции (**function**) и подпрограммы-процедуры (**subroutine**). Они во многом похожи, но отличаются способом обращения (вызова) и результатами работы.

Заметим, что в литературе встречается и другая терминология, например, в [1] процедурой называется *любая* подпрограмма, а *подпрограммой* называется только **subroutine**.

11.2. Размещение подпрограммы. Работа подпрограммы

Подпрограммы можно располагать в любом порядке относительно друг друга и основной программы, но *не внутри текста*. Чаще основную программу располагают вначале. Подпрограммы могут располагаться даже в разных файлах.

В нужном месте текста *вызывающей* программы (подпрограммы) записывается *обращение*, или *вызов* подпрограммы. В этом месте выполнение вызывающей программы будет прервано, и начнет выполняться подпрограмма. После окончания работы подпрограммы работа вызывающей программы продолжается с того места, где она была прервана.

Текст каждой подпрограммы завершается оператором **end**.

Выполнение подпрограммы может быть прервано оператором **return** – возврат (в вызывающую программу или подпрограмму).

11.3. Самостоятельность программных единиц

Основная программа и подпрограммы являются независимыми программными единицами. Это означает, что имена переменных, описания и размеры массивов, метки, форматы, оператор-функции – действуют только в одной программе или подпрограмме (там, где они описаны).

Несколько небольших подпрограмм отладить проще, чем одну большую программу.

Независимость программных единиц позволяет, например, поручить составление программы нескольким людям. В этом случае каждый из них может не думать об остальных программных единицах. Нужно лишь позаботиться об их *стыковке*, что касается лишь небольшого числа параметров (на "входе" и "выходе").

11.4. Подпрограмма-функция

Подпрограмма-функция используется, как правило, в том случае, когда вычисление некоторой функции *не укладывается в один оператор Фортрана* (т.е. нельзя обойтись оператор-функцией). Результатом работы подпрограммы-функции является одно (*возвращаемое*) значение.

Текст подпрограммы-функции начинается с заголовка вида:

function имя (список аргументов).

Аргументами (параметрами) подпрограммы-функции могут быть: простые переменные, массивы, имена других подпрограмм. Заметим, что имя самой функции может быть только *простой* переменной.

Параметры, перечисленные в заголовке функции, являются *формальными*: они показывают лишь *как* результат работы подпрограммы зависит от аргументов. При каждом использовании подпрограммы будут заданы *фактические* значения параметров. Конечно, внутри программы могут использоваться и другие переменные, не указанные в заголовке.

Имя функции используется в тексте программы как обычная простая переменная. Это переменная в ходе работы подпрограммы получает некоторое значение; оно и будет *результатом*, который передается в вызывающую программу.

Пример 11.1. Рассмотрим подпрограмму

```
function sign(x)
```

```
if(x.gt.0) z=1
if(x.eq.0) z=0
if(x.lt.0) z=-1
sign=z
end
```

Здесь имеется один формальный параметр – переменная вещественного типа *x*. Внутри программы используется также переменная *z*.

В результате работы данной программы переменная sign принимает значение, определяемое знаком аргумента *x*.

Из этой программы не видно, чему конкретно равно значение sign в результате ее работы: переменная *x* является лишь *формальным* аргументом. При обращении к подпрограмме sign переменная *x* будет иметь конкретное *фактическое* значение; им и определится результат.

Для вызова подпрограммы типа **function** ее имя (с фактическими переменными) должно появиться в некотором выражении (например, в правой части оператора присваивания).

Заметим, что использование подпрограммы-функции в вызывающей программе не отличается по виду от использования оператор-функции.

Пример 11.2. Рассмотрим программу

```
x=2.5
z=2.*sign(2.-x)
print *,z
end
```

Будет напечатано значение переменной *z*, равное –2. Отметим, что переменная *z* основной программы *не имеет никакого отношения* к переменной *z* из функции sign.

В рассмотренных примерах аргументами подпрограмм были простые переменные. *Массив* также может быть аргументом подпрограммы-функции.

Если формальный параметр – имя *массива* (массив описывается обычным образом в самой подпрограмме), то фактический параметр может быть *массивом или индексированной переменной*. Этот массив должен быть описан в вызывающей программе.

Если формальный параметр – имя функции, то фактический – имя подпрограммы типа **function** или **subroutine**.

В подпрограммах разрешается описывать массивы переменной длины, которая *обязательно* должна быть одним из аргументов.

Пример 11.3. Эта подпрограмма-функция вычисляет в массиве из n элементов номер максимального элемента.

```
Function nomax(a,n)
Dimension a(n)
w=-1.e20
do i=i,20
if(a(i).gt.w) then
w=a(i)
nomax=i
endif
End do
```

Обращение к подпрограмме *nomax* может иметь вид

```
dimension t(40)
read*,m
read *,(t(i),i=1,m)
k=nomax(t,m)
print *,k
end
```

В данном случае будут обработаны первые *m* элементов массива t.

11.5. Подпрограмма-процедура

Подпрограмма-процедура (**subroutine**) используется, как правило, в тех случаях, когда результат работы подпрограммы *нельзя выразить одним числом*. Надо заметить, что для этого вида подпрограмм нет стандартного русского термина. В литературе их часто называют просто подпрограммами или подпрограммами **subroutine**.

Подпрограмму-процедуру считают наиболее общим видом подпрограмм, поскольку оператор-функции и подпрограммы типа **function** всегда можно заменить процедурами. Однако то же самое можно сказать и о **function** (об этом см. разд. 15.4).

Текст подпрограммы-процедуры начинается с заголовка

subroutine (список параметров).

Имени процедуры не присваивается никакого значения, поэтому вопрос о типе не возникает.

Остальные правила оформления процедуры те же, что и для **function**.

Результаты работы процедуры передаются в вызывающую программу через параметры.

По виду и действию процедура напоминает известную в математическом анализе *неявную функцию*.

Обращение к подпрограмме типа **subroutine** производится с помощью оператора **call** (вызов). Он имеет вид:

call имя (список фактических параметров).

Фактические параметры должны соответствовать по количеству и типу формальным параметрам.

Пример 11.4. Составим подпрограмму-процедуру для вычисления полярных координат точки (r, φ) по заданным декартовым координатам (x, y). Для простоты считаем $x > 0$.

```
Subroutine polar(x,y,r,fi)
r=sqrt(x**2+y**2)
fi=atan(y/x)
end
```

Иногда формальные параметры процедуры разделяют по их назначению. В рассмотренном примере параметры *x* и *y* – *входные*: они переносят из вызывающей программы в процедуру исходную информацию (данные). Результат работы процедуры – значения переменных *r* и *fi*, которые называют *выходными* параметрами. Можно сказать, что результаты работы процедуры передаются в вызывающую программу через выходные параметры.

Деление параметров на входные и выходные весьма условно. Один и тот же параметр может выполнять обе эти роли.

Пример 11.5. Составить подпрограмму-процедуру, вычисляющую сумму положительных элементов массива *a* (из n элементов) и заменяющую отрицательные элементы этого массива нулями.

```
Subroutine massiv(a,s,n)
Dimension a(n)
s=0
do i=1,20
if(a(i).gt.0) s=s+a(i)
if(a(i).lt.0) a(i)=0
End do
End
```

12. ПОДПРОГРАММЫ. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

В настоящем разделе рассмотрены ситуации, когда в качестве параметров подпрограмм выступают массивы или имена других программных единиц

12.1. Массивы переменной длины в подпрограммах

Использовать двумерные и иные многомерные массивы переменной длины надо крайне осмотрительно. Если имя массива входит в число параметров, то при вызове подпрограммы совмещаются *первые элементы* массива вызываемой и вызывающей программ (т.е. формального и фактического массивов); а далее массив заполняет (без пропусков) отрезок памяти нужного размера. Заметим, что размерности (число индексов) этих массивов могут не совпадать.

Эти довольно странные особенности объясняются тем, что любой массив при обработке преобразуется в одномерный и компьютер *"не помнит"*, сколько было в массиве строк и столбцов.

Пример 12.1. Пусть в основной программе имеется оператор

```
Dimension a(10,10)
```

Процедура *s* начинается операторами

```
Subroutine s(b,n,k)
Dimension b(n,k)
```

Тогда обращение

```
Call s(a,8,8)
```

будет работать не с первыми 8 строками и 8 столбцами, а с *первыми 64 элементами* массива *a*. На рис.12.1 схематически изображено: слева – то, что мы хотели получить; справа – та часть массива, которая на самом деле будет обрабатываться программой *s*.

Отметим, что оператор

```
Call s(a,10,8)
```

будет работать так, как задумано: 10 строк и 8 столбцов массива *a* – это то же самое, что его первые 80 элементов.

а)

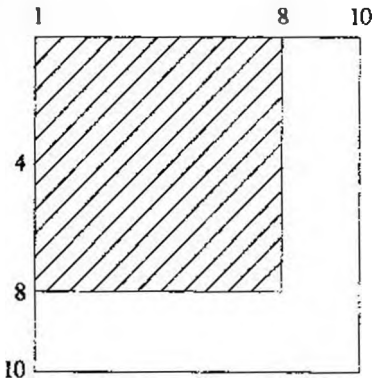

Рис.12.1 *а)*

б)

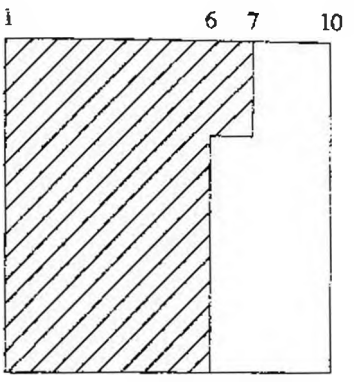

Рис. 12.1 *б)*

Как же добиться желаемой ситуации рис.12.1*а*?

Рекомендуемый ниже метод позволяет решить эту проблему и к тому же раз и навсегда *забыть*, что автоматическое упорядочение массива ведется "по столбцам".

Нужно написать промежуточную подпрограмму *sm*, которая формирует массив из *m* строк и *n* столбцов.

Соответствующий набор программ может иметь вид:

```
c           |  Основная программа
            |  Dimension a(10,10)
            |  Read *,m,n
            |  Call sm(a,m,n)
            |  Stop
            |  End
            |
c           |
c           |  Подпрограмма формирования массива v(m,n)
c           |
            |  Subroutine sm(v,m,n)
            |  Dimension v(m,n)
            |  Do i=1,m
            |  Do j=1,n
            |  . . . . . .
c           |  Определение элемента v(i,j)
            |  . . . . . .
            |  End do
```

```
      End do
      Call s(v,m,n)
      End
```

Подпрограмма *s* остается без изменений.

Здесь мы полагаем, что массив *вычисляется*. Если он *вводится*, внутренний цикл по *j* в подпрограмме *sm* лучше сделать неявным.

12.2. Внешние функции. Оператор EXTERNAL. Вычисление интегралов.

Теперь рассмотрим случай, когда параметром подпрограммы служит имя другой подпрограммы.

Прежде всего отметим, что правила Фортрана не допускают, чтобы подпрограмма обращалась сама к себе прямо или косвенно (через другие подпрограммы). Другими словами, в Фортране запрещена *рекурсия*.

Предположим, что основная программа вызывает функцию *f*, которая в свою очередь вызывает функцию *r*. В момент обращения к функции *f* основная программа должна использовать информацию, что *r* – имя *внешней* функции, а не какой-то переменной внутри основной программы.

Для использования в качестве параметров имена внешних подпрограмм объявляются (перечисляются) внутри вызывающей программы с помощью оператора **external** (внешний). Этот неисполняемый оператор записывается в начале программы среди прочих описаний: операторов **dimension** и оператор-функций.

Пример 12.2. Подпрограмма mass заменяет нулями отрицательные элементы одномерного массива (см. пример 14.6).

```
      Subroutine mass(a,n,out)
      Dimension a(n)
    1 Do i=1,n
      If(a(i).lt.0) a(i)=0
      End do
      Call out(a,n)
      End
```

Подпрограммы *out1* и *out2* предназначены для вывода результатов; они различаются форматом.

Процедура *out1* печатает массив "в столбик".

```
      Subroutine out1(a,n)
      Dimension a(n)
      Print 900,(a(i),i=1,n)
900   Format (f8.1)
      End
```

Процедура *out2* печатает массив *a* "в строчку", по 10 чисел на строке.

```
      Subroutine out2(a,n)
      Dimension a(n)
      Print 900,(a(i),i=1,n)
900   Format (10f8.1)
      End
```

В основной программе при обращении к процедуре mass необходимо указать *фактическое имя процедуры* в операторе **external** например,

```
      External out1
      Call mass(b,20,out1)
```

Пусть параметром является имя подпрограммы-функции (**function**).

Пример 12.3. Для нахождения максимума произвольной функции g(x) можно написать подпрограмму

```
      Function amax(a,b,g,h)
      Do x=a,b,h
      U=g(x)
      If (amax.lt.u)amax=u
      Enddo
      End
```

Основная (вызывающая) программа может иметь для *конкретной* функции f(x) вид

```
      External f
      Read *, a,b,h
      Z=amax(a,b,f,h)
      Print *,z
      End
```

В качестве f(x) можно взять любую подпрограмму-функцию.

ПРИЛОЖЕНИЯ

Приложение 1. Подготовка программ в MS DOS (WATFOR-77)

Чтобы действительно получить результат поставленной задачи, нужно *ввести в компьютер текст программы и запустить ее на выполнение*. Эти операции на современных компьютерах выполняются достаточно просто.

Мы будем предполагать, что работа ведется в системе WATFOR-77 с использованием операционной оболочки Norton Commander (сокращенно NC). Далее описывается рекомендуемая последовательность действий.

Итак, на экране изображена стандартная таблица NC.

Выберем сначала нужный *диск*.

Для выбора диска нажимаем клавиши **Alt+F1** (левая панель) или **Alt+F2** (правая панель).

Плюс набирать не нужно!

Появится небольшое окно с отображением всех дисков. С помощью клавишей со стрелками устанавливаем курсор на нужный диск (например, диск E) и нажимаем клавишу **Enter**.

Затем создадим *каталог* для хранения программ. Для студентов в имени каталога указывается факультет, курс и группа.

Пример: PGS-1-4

Для создания каталога нажимаем клавишу **F7**

В появившемся окне набираем по образцу имя каталога и нажимаем **Enter**. В таблице появится имя каталога (оно всегда выводится заглавными буквами) *с поставленным на него курсором*.

Войдем в каталог, другими словами, сделаем его *текущим*; для этого достаточно нажать **Enter**.

Теперь – самое главное: создаем *файл*, в который будет записана программа. Рекомендуется для имени файла использовать начало своей фамилии с указанием номера программы (лабораторной работы).

Файлы могут иметь *расширение*. Оно присоединяется к имени после *точки*. Программы на Фортране должны иметь расширение **for**.

Для создания файла нажимаем клавиши **Shift+F4.**

В новом появившемся окне набираем имя файла, например, для фамилии Петров и лабораторной работы №1 – pet1.for (ни в коем случае не используйте в *имени* точки!)

Неконтролируемое использование точек в именах файлов может привести к потере информации!

Нажав **Enter**, мы наконец получим чистую область экрана для набора программы. Теперь можно набирать *текст программы*.

Для быстрого позиционирования курсора рекомендуется в начале каждой строки нажимать клавишу **Tab** – табуляция. Тогда курсор будет автоматически перемещаться в 9-ю позицию. Отсюда можно начинать набор строки, т.к. эта позиция *правее* 6-й.

Конечно, *никаких вертикальных линий в 6-й позиции набирать не нужно*.

В конце каждой строки нажимаем **Enter**.

Не забудьте нажать **Enter** в конце последней строки (**end**).

В процессе набора программы рекомендуется время от времени нажимать клавишу **F2** (запись на диск). Тогда набранная часть файла будет надежно сохранена.

Закончив набор программы, нажимаем **F2** и затем **Esc**.

Произойдет выход в стандартную таблицу NC. Курсор будет стоять на имени созданного нами файла.

Для запуска программы *на выполнение* теперь достаточно нажать **Enter**.

Текст программы с результатами (если будут получены!) автоматически запишется в одноименный файл с расширением *lst* (листинг, распечатка). В нашем примере это будет файл *pet1.lst*.

Для просмотра файла на экране нажимаем **F3**.

Если в программе были ошибки (с точки зрения системы), то в этом файле будут сообщения о них с указанием соответствующего места в программе, *а результатов не будет*.

Для исправления ошибок вызываем *снова* файл *pet1.for*. Всякое изменение файла называется *редактированием*.

Для редактирования файла нажимаем **F4**.

Внеся в файл необходимые изменения, снова запускаем его на выполнение.

Не следует редактировать файл с расширением *lst* – при этом не меняется файл с программой!

Приложение 2. Объединение программных файлов

Самым простым является случай, когда весь исходный текст программы содержится в одном файле. Однако часто бывает необходимо включить в нее готовые тексты, содержащиеся в каких-то других файлах и даже других каталогах – стандартные программы из соответствующих библиотек, некоторые вспомогательные файлы с описанием констант и переменных и т.п. Прямое копирование таких файлов в основную программу может сделать ее слишком громоздкой. В современных компьютерных средах существуют различные возможности связывания файлов без их «физического» объединения.

П2.1. Связывание исходных текстов в системе WATFOR-77.

В этой системе единственным средством объединения файлов является *метакоманда* **include**, не являющаяся оператором языка. Она обычно записывается перед началом программы или после заключительного оператора **end**. В строке метакоманда идет сразу после знаков *$, стоящих в позициях 1-2.

Общий вид метакоманды include такой:

*$**include** имя присоединяемого файла (путь).

Чтобы избежать печати ненужных текстов при компиляции, используется метакоманда **nolist**; отменяет ее действие метакоманда **list**.

Пример П2.1. Пусть нужно добавить к исходному тексту программы две подпрограммы – *eigen.for* и *nroot.for*, хранящиеся в библиотеке стандартных программ SSP на диске D. Тогда конец программы будет выглядеть следующим образом:

```
      | | end
*$nolist
*$include d:\SSP\eigen.for
*$include d:\SSP\nroot.for
```

Пробелы ставятся только в указанных местах!

П2.2. Связывание исходных текстов MS FORTRAN.

Можно связывать исходные тексты, используя в основной программе оператор **include** – присоединить. Его общий вид:

include 'имя файла (путь)'.

Таких операторов может быть несколько. Каждый присоединяет свой файл. Эти операторы обычно размещают в начале программы.

Кроме того, можно использовать и метакоманды.

Приложение 3. Работа в среде FORTRAN PS4 для WINDOWS 95

В данном приложении рассматривается работа с более новой версией Фортрана – Fortran Power Station 4.0.

П3.1. Вход в среду

При любой настройке WINDOWS-95 войти в среду Fortran Power Station 4.0 можно путем следующих действий:

– в меню рабочего стола выбрать пункт *Программы*;

– в развернувшемся меню выбрать пункт *Fortran PowerStation 4.0*;

– в новом открывшемся меню выбрать *Microsoft Developer Studio*.

Произойдет загрузка среды и выход в основное меню.

П3.2. Создание проекта

Для этого:

– в меню *File* выбираем пункт *New*;

– в открывшемся меню выбираем пункт *Project Workspace* и нажимаем кнопку OK;

– в новом меню выбираем *Console Application* или, если предполагается использовать графику, *Standard Graphic Application*;

– в окошечке *Name* пишем имя файла (проекта) без расширения (в нашем примере – *pet1*); затем нажимаем кнопку *Create* (создать); происходит возврат в основное меню; левое окно откроется; в нем появится значок (папка) проекта и его имя с добавлением *files* .

Теперь можно подготовить текст программы и включить его в проект.

П3.3. Создание файла с исходным текстом программы.

– меню *File* выбираем *New* (*Новый*);

– в открывшемся меню выбираем пункт *Text* (текстовый файл); при возврате в главное меню откроется правое окно;

– в меню *File* выбираем *Save As* (*Сохранить как*);

– в появившемся диалоговом окне набираем имя файла с расширением *for*: на экране появится рабочая область для набора программы. Характерный признак рабочей области – *вертикальная зеленая черта*, проходящая по 6-й позиции.

Имя программы, конечно, не обязательно должно совпадать с именем проекта. Мы назвали программу *p1.for*.

П3.4. Добавление файлов в проект.

Для этого:

– в меню *Insert* (*Вставка*) выбираем *Files into Project* (*Файлы в проект*);

– в диалоговом окне находим и отмечаем нужные файлы и нажимаем кнопку *Add* (*Добавить*);

– если файл с текстом программы не включить в проект, она выполняться не будет.

П3.5. Запуск программ.

– проще всего запустить программу на компиляцию и компоновку нажатием комбинации клавиш **Alt+F8**;

– обнаруженные компилятором ошибки (*Errors*) и сделанные системой предупреждения (*Warnings*) появятся в нижнем окне;

– на строке с указанием ошибки или предупреждения можно выполнить двойной щелчок мышью, и система отметит место ошибки в тексте программы.

Если результаты больше не нужны, нужно *обязательно* нажать любую клавишу для снятия задачи. Если программные файлы или проекты уже созданы, то после входа в среду нужно в меню *File* выбрать пункт *Open* (*Открыть*) и в появившемся диалоговом окне обычным образом провести поиск.

Можно запускать программы и с помощью меню *Build* (*Собрать*), выбрав последовательно пункты *Compile* (*Компилировать*), *Build* и *Execution* (*Выполнение*) или сразу *Build* и затем *Execution*.

Заметим, что предупреждения не препятствуют запуску программы на выполнение. При наличии ошибок программа выполняться не будет.

Приложение 4. Подпрограмма как обобщение понятия функции

В таблице представлены различные способы оформления вычисления функции

$$z = \sin(xy) + \cos x$$

на языке Фортран.

Наименование	Описание	Использование (обращение, вызов)
Комбинация стандартных функций	нет	`z=sin(x*y)+cos(x)`
Оператор-функция	В начале программы. После операторов описания типов и массивов (если есть) `w(u,v)=sin(u*v)+cos(u)`	`z=w(x,y)`

Подпрограмма-функция	Отдельная программная единица.
	`Function w(u,v)`
	`w=sin(u*v)+cos(u)`
	`end`
		`z=w(x,y)`

Подпрограмма-процедура	Отдельная программная единица
	`subroutine g(u,v,w)`
	`w=sin(u*v)+cos(u)`
	`end`
		`call g(x,y,z)`

ЛИТЕРАТУРА

1. Белецки Я. Фортран 77. – М.: Высшая школа, 1991.
2. Катцан Г. Язык Фортран-77. – М.: Мир, 1986.
3. Новые возможности языка Фортран. Метод. указ./Сост. П.В.Уманец. –М.: МИСИ, 1986.
4. Разветвления и циклы в Фортране-77 Метод. указ./Сост. А.М.Купфер. –М.: МИСИ, 1992
5. Самохин А.Б., Самохина А.С. Фортран и вычислительные методы. – М.: Русина, 1994.
6. Соловьев П.В. Фортран для персонального компьютера. – М.: Арист, 1991.
7. Основы работы на персональном компьютере и элементы программирования на алгоритмических языках.. Методические указания / Сост В.Н. Сидоров и др. – М.: МГСУ, 2001.
8. Элементы программирования и использование стандартного программного обеспечения. Методические указания / Сост. А.Б Золотов и др. – М.: МГСУ, 2001.