Глава 8. Процедуры и функции

Подпрограмма – поименованная или иным образом идентифицированная часть программы, содержащая описание определённого набора действий. Однажды описанная, подпрограмма может быть многократно вызвана из разных частей программы.

Википедия

Подпрограммы (процедуры и функции) являются очень важным инструментом в языках программирования. Они предназначены для реализации свойства *структурности* алгоритма. По существу, подпрограммы (subprograms, subroutines) описывают подзадачи, на которые разлагается алгоритм.

В большинстве языков программирования предусмотрены средства, позволяющие чётко определить для подпрограмм их входные и выходные данные. Сначала мы будем изучать подзадачи-процедуры, а потом рассмотрим, чем от них отличаются подзадачи-функции.[1]

8.1. Процедуры

Существует множество вещей, с которыми мы свыкаемся, не понимая их.

Айзек Азимов. «Конец Вечности»

Для вызова процедур предназначен оператор процедуры. Надо, однако чётко понимать, что перед использованием этого оператора где-то в другом месте программы (выше по тексту или в другом модуле) обязательно должно располагаться описание этой процедуры, видимое из точки вызова. У оператора процедуры и у описания процедуры в языках программирования сложные синтаксис и семантика. Сначала рассмотрим синтаксис описания процедур в стандарте Паскаля.

```
<описание процедуры> ::= <заголовок процедуры><блок><конец процедуры>
<конец процедуры> ::=  ;
```

Как видно, описание процедуры очень похоже на описание всей программы на Паскале. Основной частью описания программы и описания процедуры является блок, а вместо точки в конце программы используется точка с запятой в конце процедуры. Соответственно, заголовок программы заменяется заголовком процедуры. Это не случайно, так как процедура описывает *подзадачу*, структура которой должна соответствовать структуре всей задачи. Более того, по свойству структурности описание процедуры может включать в себя (разлагаться) на другие процедуры и функции и т.д. Эта возможность реализуется в блоке, который может включать в себя свой раздел процедур и функций.

<заголовок процедуры> ::= **`procedure`** <имя>[(<формальные параметры>)];

Заголовок процедуры (procedure header) содержит имя процедуры и, возможно, список формальных параметров (formal parameters), иногда говорят список формальных аргументов (formal arguments). В стандарте Паскаля у подпрограммы обязательно есть имя, нет *безымянных* (анонимных) подпрограмм,[2] в более сложных языках есть и *безымянные* (анонимные) подпрограммы.

Список формальных параметров заключается в круглые скобки, а если параметров нет, то и круглые скобки опускаются. Другими словами, в стандарте Паскаля *пустые* круглые скобки запрещены. В некоторый других языках (например, в Фортране и С) при отсутствии параметров остаются (обязательные) пустые круглые скобки. Оба этих подхода имеют свои достоинства и недостатки, о которых мы сейчас говорить не будем. В языке Free Pascal в подпрограммах без параметров можно (но не обязательно) использовать пустые круглый скобки.

---

[1] В некоторых языках (например, в языках C, Haskel и др.) процедур нет, есть только функции.

[2] Почти во всех языках у процедуры одно имя или, как говорят, одна точка входа (entry point), но есть языки, которые допускают у процедуры несколько точек входа (несколько имён), мы про это говорить не будем.

Забавно, но в языке C *пустые* круглые скобки у описания функции означают не отсутствие параметров, а произвольное число фактических параметров, все их функция игнорирует. Такое «странное» поведение функций языка C объясняется особыми соглашениями о связях между основной программой и функциями, его Вы будете изучать в курсе по архитектурам ЭВМ.

Формальные параметры задают (при хорошем программировании) все входные и выходные данные процедуры. В частности, в процедуру могут передаваться параметры, которые, в свою очередь, тоже являются процедурами и функциями. К сожалению, механизм такой передачи в стандарте Паскаля признан неудачным, и все конкретные реализации (и наш Free Pascal), реализуют другой механизм такой передачи. Исходя из этого, передачу процедур и функций в стандарте Паскаля мы рассматривать не будем, поэтому синтаксис формальных параметров существенно упрощается.

```
<формальные параметры> ::= <секция>{;<секция>}...
<секция> ::= [var]<имя>{,<имя>}...:<имя типа>
```

Каждый формальный параметр обязательно имеет имя и тип, причем допускаются только *именованные* типы, скоро мы поймём, почему так сделано. Здесь учащиеся регулярно делают ошибки, пытаясь указать *безымянные* типы, например, `1..100` или `array[1..N] of char`.[1] Для удобства программирования все однотипные параметры можно объединять в одну секцию, например, вместо `x: integer; y: integer;` писать `x,y: integer;` (а вот, скажем, язык C такого не допускает).

Перед секцией может стоять (а может и не стоять) служебное слово **var**, такие секции имеют разную семантику, которую мы вскоре рассмотрим. Итак, каждый параметр имеет имя и число параметров процедуры равно количеству таких имён во всех секциях. Заметим, что по сути формальные параметры описываются так же, как и переменные (да фактически они и являются переменными), они часто и называются *формальными переменными*.

Таков синтаксис описания процедуры, теперь рассмотрим синтаксис оператора процедуры:

```
<оператор процедуры> ::= <имя>[(<фактические параметры>)]
<фактические параметры> ::= <выражение>{,<выражение>}...
```

Итак, каждый фактический параметр является выражением, число таких выражений и задаёт число фактических параметров (actual parameters). Далее начинаются сложности. Дело в том, что и описание процедуры и оператор этой процедуры по отдельности могут быть синтаксически правильными, а вот вместе уже неправильными.

Мы уже сталкивались с такой ситуацией, например, при описании оператора присваивания, который состоял из левой части (чаще всего переменной переменной) и правой части (выражения). Для правильности всего оператора присваивания мы накладывали на его синтаксис семантический фильтр, говорили, что тип выражения должен соответствовать типу переменной. Аналогично и для общей синтаксической правильности описания процедуры и оператора процедуры должно соблюдаться соответствие между формальными и фактическими параметрами, т.е. фактические параметры должны *соответствовать* формальным (а не наоборот!). Должно выполняться три вида соответствия: в числе, в типе и в порядке следования. Рассмотрим эти соответствия подробно.

3. Соответствие в порядке следования. Во всех книгах по программированию это соответствие описано третьим, но, по хорошему, как мы здесь и сделали, его надо рассматривать первым. Оно определяет так называемое *позиционное* соответствие: первый фактический параметр соответствует первому формальному, второй фактический – второму формальному (слева-направо) и т.д. Необходимость явного указания на такое соответствие (positional parameters) заключается в том, что существуют языки (например, Ada, Python), в которых есть так называемые ключевые параметры (keyword parameters). В этом случае фактический параметр содержит явное имя (ключ) формального параметра, которому он будет соответствовать. Таким образом, фактические параметры при вызове процедуры могут идти в любом порядке. В большинстве языков используется только позиционное соответствие фактических и формальных параметров.

1. Соответствие в числе. Число выражений-фактических параметров должно точно совпадать с числом имён формальных параметров (во всех секциях).

---

[1] В языке Free Pascal в качестве исключений допускаются безымянные типы, о них мы расскажем далее.

В Паскале такое соответствие должно соблюдаться только для процедур пользователя, для стандартных подпрограмм его может и не быть. Например, в стандартной процедуре ввода можно писать любое число фактических параметров: `read(x,y,z,...)`. Было бы «неприлично» требовать от программиста писать вместо этого `read(x); read(y); read(z);...` Здесь, однако, Паскаль вызывает «сам себя» и всегда может «договориться», как определить настоящее число параметров (например, можно передать скрытый дополнительный параметр, задающий число остальных параметров). А вот когда процедуру пишет программист Иванов, а её вызывает программист Петров, это становится проблемой. В некоторых языках (например, в языке C) нельзя обойтись без функций с переменным числом параметров.[1] Соответственно там должны быть громоздкие и ненадежные механизмы работы с переменным числом параметров. Ярким примером является так называемый *форматный* ввод/вывод в языке C, когда значение первого фактического параметра определяет число остальных параметров. Вам придётся ознакомится с этим при изучении языка C. Кроме того, переменное число параметров часто используется в макроопределениях Ассемблера (это тема отдельного курса по архитектурам ЭВМ).

2. Соответствие в типе. Тип фактического параметра должен *соответствовать* типу своего формального параметра (а не наоборот!). Это соответствие сложное и устанавливается по разному, в зависимости от того, стоит ли перед секцией формальных параметров служебное слов **`var`**, или не стоит.

а). Секция без **`var`**. В этом случае требуется *соответствие по присваиванию* между формальным параметром и выражением–фактическим параметром (нужно вспомнить, что это такое). Такая передача параметра называется передачей по значению (pass by value).

б). Секция с **`var`**. Такая передача параметра называется передачей по ссылке (pass by reference), скоро мы поймём смысл этого названия. Здесь для соответствия должны выполняться два требования.

1). Фактический параметр может быть не любым выражением, а только переменной. Скоро мы поймём, почему так должно быть.

2). Тип формального параметра должен совпадать с типом фактического параметра. Так как формальный параметр должен иметь *именованный* тип, то совпадать они могут только тогда, когда имя типа фактического параметра либо совпадает с именем типа формального параметра, либо эти имена объявлены как *идентичные*.

Язык Free Pascal позволяет создавать «хитрые» совместимые по присваиванию, но не идентичные типы (alias type), например:

```
type int1=integer; int2=type integer;
var a: integer; b: int1; c: int2;
procedure P(❶ var x: int1); begin ... end;
    P(a); // всё хорошо
    P(b); // всё хорошо
    P(c); // ОШИБКА, несоответствие типов !
```

При передаче параметра по значению (без ❶ **`var`**) такой ошибки не возникает, есть совместимость по присваиванию.

Как видим, соответствие в типе предъявляет очень жёсткие требования к описанию процедуры и оператору процедуры. Это призвано повысить надёжность программы, так как через параметры в процедуру поступают входные данные и возвращаются результаты работы, и это должно строго контролироваться синтаксисом языка. Таким образом, Паскаль уже на этапе компиляции выявляет большинство ошибок взаимодействия между основной программой и её процедурами. Заметим, что многие языки предъявляют значительно менее строгие требования к передаче параметров в подпрограммы.

---

[1] Для продвинутых читателей. Подпрограммы с переменнывм числом параметров называются вариадическими (variadic). В некоторых языках (например, в языке C) без них не обойтись, так как там нет стандартного ввода/вывода, это функции пользователя (правда, обычно из стандартной библиотеки, написанной опытными программистами).

Перейдём теперь к *семантике* взаимодействия оператора процедуры с описанием этой процедуры, т.е. к правилам выполнения оператора процедуры. Это выполнение включает в себя несколько этапов.

1. Поиск описания процедуры.
2. Передача фактических параметров на место формальных.
3. Выполнение блока процедуры.
4. Возврат из процедуры на следующий после вызова оператор.

Подробно разберём выполнение каждого из этапов.

1. Поиск описания процедуры. Для каждого *оператора* процедуры где-то (выше по тексту программы или в другом модуле) обязательно должно находится *описание* процедуры с этим именем. Если такого описания нет, то фиксируется ошибка «Не описано имя процедуры». В то же время допускается, чтобы выше по тексту программы было *несколько* описаний этого имени.

Теперь необходимо вспомнить, что при выполнении алгоритма исполнитель всё время переходит от одного шага к другому. Таким образом, в каждый момент времени существует текущая точка выполнения программы.[1] И вот здесь мы приходим к важному понятию области видимости (scope) имени. Оказывается, что исполнитель, находясь в некоторой точке программы, может просто не видеть некоторых участков программы (и, в частности, описанных там имён ⚠).

Эту ситуацию можно образно сравнить с одним из видов японского искусства, так называемым *садом камней*. В саду японского храма Рёан-дзи в беспорядке расположены 15 больших камней. Прогуливаясь по извилистой тропинке в этом саду, человек всегда видит только 14 камней, а один скрыт за остальными камнями. Пройдя далее по тропинке, можно увидеть другие 14 камней, но один по прежнему скрыт.

Области видимости тесно связаны с *блоками* программы, о чём мы будем подробно говорить далее. Для синтаксически правильной программы требуется, чтобы из каждой точки выполнения исполнитель видел выше по тексту программы ровно одно описание используемого имени, а остальные должны быть скрыты.[2] Понятно, что программист, просматривая текст программы, должен чётко понимать, какие имена видны из каждой точки выполнения.

Выполнение остальных шагов оператора процедуры (2, 3 и 4) лучше проводить на примере. Рассмотрим следующую простую программу.

```
program My_prog(input,output);
  var ❶ a,b: integer;
  procedure P(x: integer; var y: integer);
    var ❸ a,c: integer;
    begin ❹ a:=-1; c:=-2; x:=4; y:=a+b end;
begin ❷ a:=1; b:=2; P(a+2,b); write(a,b)
end.
```

Здесь блок всей программы показан более светлым, а блок процедуры показан более тёмным прямоугольниками. В нашем примере выполнение программы `My_prog` начинается с раздела переменных ❶, при этом эти переменные порождаются. При порождении им отводится место в памяти ЭВМ, таким образом, у переменных появляются ссылки (адреса) их расположения в памяти. Говорят, что в этот момент производится связывание (binding) имён переменных с их ссылками. Будем условно считать, что каждая переменная занимает некоторую «ячейку» памяти, адрес которой задаётся целым числом.[3] Например, пусть порождается некоторая переменная `a`, мы изобразим это так:

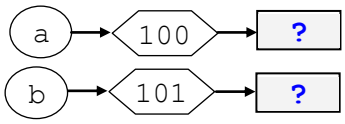
Здесь показано, что переменная с именем `a` размещена в 100-й ячейке памяти (её ссылка равна 100), и сразу после порождения не она имеет никакого конкретного значения. Тогда, после обработки раздела переменных всей программы `var ❶a,b: integer;` у нас будет такая картинка:

---

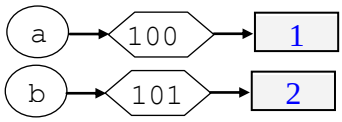
[1] В архитектуре ЭВМ этому соответствует значение счётчика адреса машинной команды.

[2] В современных языках есть важное исключение из этого правила, см. разд. 8.7.

[3] Обычно переменная занимает один или несколько подряд расположенных байт памяти ЭВМ, адресом переменной считается адрес первого из этих байт.

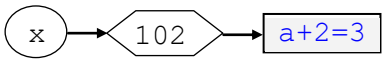


Затем начинает выполняться раздел операторов программы, после выполнения операторов присваивания ❷`a:=1; b:=2;` мы получим:

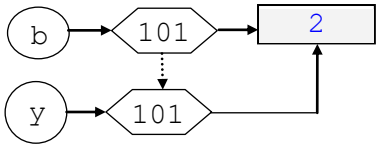


Теперь начинает выполняться оператор процедуры `P(a+2,b)`. Первый фактический параметр `a+2` передаётся на место формального параметра `x` (передача по значению), а второй фактический параметр `b` на место формального параметра `y` (передача по ссылке). Часто говорят, что производится *связывание* (binding) формальных параметров с фактическими.

При передаче первого параметра по значению *порождается* новая переменная `x` и ей присваивается значение фактического параметра:

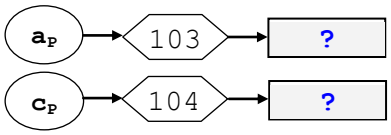


При передаче второго параметра по ссылке порождается переменная `y` и ей отводится то же самое место в памяти, которое занимает фактический параметр – *переменная* `b`.



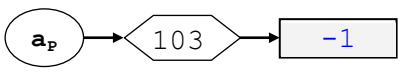
Теперь у переменной `b`, расположенной по адресу `101`, появилось новое, как говорят, альтернативное имя (alias) `y`, так что все операции с `y` «на самом деле» проводятся с переменной `b`. Как видим, при передаче по значению копируется значение фактического параметра, а при передаче по ссылке – его ссылка (у нас это действие показано как пунктирная стрелка).

Далее по семантике выполняется блок процедуры (закрашен в программе более тёмным). Сначала в блоке расположен раздел переменных, поэтому порождаются переменные с именами ❸`a,c: integer;`. Будем обозначатьэти переменные с индексом `P` как a_P и c_P, им отводятся новые ячейки памяти с неопределёнными значениями:

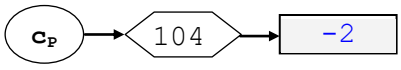


Теперь в программе существуют две переменные с именем `a`, одна описана в блоке всей программы, а вторая – в блоке процедуры (вспомним знаменитый японский сад камней, правда у нас не 15, в всего два камня 🙂).

И вот теперь начинает выполняться раздел операторов процедуры. Оператор ❹ `a:=-1` видит выше по тексту два описания переменной с именем `a`. По правилам, описание некоторого имени во *внутреннем* блоке делает невидимым описание этого имени во *внешнем* блоке, говорят, что внутреннее описание экранирует (заслоняет) внешнее описание.[1] Таким образом, будет выполняться присваивание a_P`:=-1`.



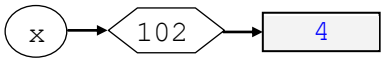
Далее выполняется оператор `c:=-2`, выше есть только одно описание имени `c`, поэтому будет:



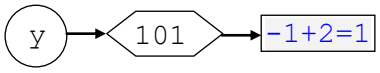
---

[1] Для продвинутых читателей. Вообще говоря, здесь ситуация более сложная. Внутреннее имя экранирует внешнее только когда у них разные так называемые *сигнатуры*, см. разд. 8.7.

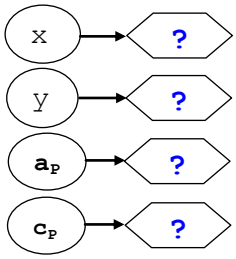
Следующий оператор присваивания `x:=4` тоже выполнится просто:



И, наконец, последний оператор в процедуре `y:=a+b` выполнится как $y:=a_P+b=-1+2=1$:



После выполнения раздела операторов начинается этап выхода из процедуры. На этом этапе все переменные, которые были порождены при входе в процедуру, должны быть уничтожены. Занимаемая ими память ЭВМ объявляется свободной и может быть в дальнейшем использована для размещения новых переменных [i] [см. сноску в конце главы]. Получается следующая картинка:



Как мы ранее говорили, переменные, которые порождаются при входе в подпрограмму, и уничтожаются при выходе, принадлежат к автоматическому классу памяти. Вне подпрограмм у этих переменных из атрибутов остаётся лишь имя, тип и класс памяти, у них нет ни ссылок, ни значений.

Отметим такое важное свойство нашего языка программирования. Когда переменная *не существует*, то её имя *невидимо* (вскоре мы покажем, что обратное неверно). Это делает невозможным все попытки программиста читать и писать в несуществующие переменные, что увеличивает надёжность программ. Это, однако, распространяется только на переменные статического и автоматического классов. Для переменных динамического класса памяти (мы будем их подробно изучать далее) это, к сожалению, уже не так. Переменные этого класса порождаются и уничтожаются только по прямым указанием программиста, поэтому возможно обращение как к ещё не порождённым, так и к уже уничтоженным переменным, что влечёт позникновение в программах тяжёлых семантических ошибкок.

Дадим теперь важное определение. Имя называется локальным (local name) *в некоторой точке* (работы исполнителя), если это имя описано выше в том же блоке, где находится указанная точка, или является *формальным параметром*, иначе имя называется глобальным (global name). Как следует из определения, свойство быть локальным или глобальным зависит от текущей точки выполнения программы, одно и то же имя из одной точки может смотреться как локальное, а из другой – как глобальное. Снова рассмотрим в качестве примера нашу предыдущую простую программу, указав в ней несколько точек выполнения, помеченных значками ❶, ❷ и т.д.

```
program ❺ My_prog(input,output);
var a,b: integer;
procedure P(x: integer; var y: integer);
  var a,c: integer;
  begin ❶ a:=-1; c:=-2; x:=4; ❷ y:=a+b end;
begin ❸ a:=1; b:=2; P(a+2,b); ❹ write(a,b)
end.
```

В точке ❶ имя `a` является локальным, оно описано выше в блоке процедуры (для наглядности ранее на картинках мы обозначали это имя как a_P). В точке ❷, имя `y` является локальным (это формальный параметр), а имя `b` наоборот, является глобальным, оно описано в объемлющем блоке всей программы. В точке ❸ имя `a` является локальным, а имя a_P невидимым из данной точки. Про невидимые имена бессмысленно спрашивать у исполнителя, являются ли они для него локальными или глобальными, он их просто не видит!

Рассмотрим теперь точку ❹, в этой точке имя `write` будет глобальным, так как является *стандартным* именем, которые программист в своей программе не описывает. Аналогично, имя всей программы ❺ `My_prog` и имена потоков ввода/вывода `input` и `output` тоже являются глобаль-

ными по отношению к программе. Считается, что все стандартные имена описаны в специальном блоке с именем **system**, в который вкладывается блок всей программы:

```
{ Блок system }
const Maxint=32767; var input: text;
procedure write(...
program My_prog(input,output);
var a,b: integer;
procedure P(x: integer; var y: integer);
  var a,c: integer;
  begin a:=-1; c:=-2; x:=4; y:=a+b end;
begin a:=1; b:=2; P(a+2,b); write(a,b)
end.
```

Считается, что в блоке **system** как-то описаны все стандартные имена (`write`, `abs`, `sin`, `Maxint` и т.д.). Таким способом стандартные имена становятся видимыми в программе. Их, однако, можно *переопределить*, если описать во внутренних блоках такое же имя, по общим правилам оно экранирует стандартное имя, делая его невидимым.

Впрочем, в языке Free Pascal к экранированным именам можно обратиться, используя *префикс* модуля, в котором описаны эти имена (тогда говорят о *квалифицированном имени*), например:

```
const
  Write=10; { теперь write имя константы 10 🐷 }
begin
  system.Write(Write); { печать константы Write=10 }
```

Таким образом можно увидеть имена внутри другого модуля, экранированные такими же локальными именами, например:

```
const Maxint=-1; { теперь Maxint=-1 }
type
  integer=shortint;
  Natural=0..system.Maxint;
var x: system.integer;
```

В языке Free Pascal существуют и так называемые типизированные константы. Это «странные» константы, значения которых можно менять, более уместно называть их переменными с начальными значениями.[1] Описывая такие «константы», программист явно задаёт тип этой константы (и, в частности, её длину в байтах), например:

```
const Nb: byte=2;     { Беззнаковая двойка длиной в байт }
      N64: int64=2;   { Знаковая двойка длиной 8 байт }
{$R-} Nb: byte=-2;    { Nb=254 ⚠ }
{$R+} Nb: byte=-2;    { ОШИБКА Range Check Error }
```

В языке Free Pascal допускаются и типизированные константы сложных типов,[2] например:

```
const Comp: record re,im: real end=(re:0; im:1);
      Arr: array[1..4] of byte=(1,2,3,4);
      Stc: set of char=['+','-','0'..'9'];
```

Иногда программисту хочется «сидеть на двух стульях». Ему нужно, чтобы переменная была видимой (т.е. могла использоваться) только внутри процедуры, но при выходе из этой процедуры не уничтожалась. И пусть при повторном входе в эту же процедуру переменная имеет то же значение, что у неё было при последнем выходе. Это позволит процедуре иметь свою «личную» переменную, которая сохраняет данные между вызовами (например, ведёт счётчик вызовов процедуры, и об этом

---

[1] Если Вас смущает, что такие «константы» можно менять, то можно запретить это директивой `{$J-}` (синоним `{$WriteableConst OFF}`).

[2] Константы сложных типов невозможно сделать не типизированными, так как по внешнему виду такой константы часто невозможно опеределить её тип.

счётчике «не знает» главная программа). Для этого тоже можно использовать типизированные константы, например:

```
procedure P(x: integer);
const
  N=100; { обычная локальная константа }
{ далее две типизированные константы y и z }
 ❶ y: shortint=-1;
 ❶ z: array[1..3] of byte=(1,2,3);
var
  a: longint; { обычная локальная переменная }
{ локальная переменная b с начальным значением 'A' }
❷ b: char='A';
begin y:=y+1; ... end;
```

Локальная переменная ❷ b порождается при каждом входе в процедуру и ей присваивается начальное значение 'A', при выходе из процедуры эта переменная уничтожается. А вот *переменные* с начальными значениями ❶ y и z видны только в процедуре P, они порождаются до начала счёта программы с указанными начальными значениями, далее они существуют и *между вызовами*. У этих переменных локальная область видимости и глобальная область существования ⚠. На будущее: при *рекурсивном вызове* процедуры P существует только один экземпляр таких переменных. Отметим, что в языке C такие локальные переменные обозначаются ключевым словом **static**.

Итак, семантика процедур весьма сложная 😕. Какие это даёт преимущества программисту? Во-первых, внутри подпрограмм появляется, как говорится, новое пространство имён. Другими словами, внутри подпрограмм программист может использовать любые имена, независимо от того, используются ли эти имена в других частях программы. Во-вторых, это экономия памяти: когда подпрограмма заканчивается, все её локальные переменные уничтожаются, освобождая память, которая может быть использована для размещения порождаемых переменных другой подпрограммы.

В языке Free Pascal при включённом режиме работы {$**mode** OBJFPC} можно не передавать в подпрограммы последние из списка фактических параметров, если при описании для них указаны так называемые значения по умолчанию, например:

```
{$mode OBJFPC}
  var a: Mas;
procedure P(var x: Mas; N: integer=100);
  begin ... end;
begin P(a,200); ❶ P(a); { Это P(a,100) }
```

Здесь при вызове процедуры ❶ P опущен второй параметр (длина массива), по умолчанию он получает в процедуре значение 100. Многия языки (например, Ada, C++, Python, PHP, Ruby, последние версии языка Fotran и другие) допускают передачу параметров по умолчанию.

Задавать значения по умолчанию можно только для параметров, переданных по значению. После изучения этого раздела объясните, почему это так (по крайней мере постарайтесь) ⚠.

Разберём теперь прагматику передачи параметров. Когда программист должен передавать параметр по значению, а когда по ссылке?

При передаче по значению, в подпрограмме создаётся новая переменная и туда передаётся копия фактического параметра, и даже если фактический параметр сам является переменной, то в подпрограмме эту переменную нельзя изменить (испортить). Другими словами, если мы не предполагаем в подпрограмме изменять значение параметра, то из соображений надёжности его следует передавать только по значению. Единственное противопоказание – большой объём фактического параметра, тогда снять с него копию становится весьма трудным делом. Например, пусть параметр является массивом из миллиона чисел. При передаче такого массива по значению необходимо завести в подпрограмме новый массив на миллион чисел и скопировать старый массив в новый. Отметим, что в этом случае возможен и аварийный останов программы, если уже нет необходимого свободного объёма памяти для снятия копии с фактического параметра. Исходя из этого, передачу больших параметров рекомендуется делать всё-таки по ссылке, даже если это уменьшает надёжность программы (увеличивает риск испортить фактический параметр). [ii] [см. сноску в конце главы]

В языке Free Pascal есть альтернативный способ передачи больших параметров, которые нельзя менять внутри процедуры. Этот способ совмещает в себе эффективность передачи параметра по ссылке с высокой надёжностью передачи по значению. Рассмотрим пример:

```
const N=1000000;
type Mas=array[1..N] of longint;
procedure P(var x: Mas; const y: Mas);
```

Здесь служебное имя **const** предписывает передавать большие параметры *по ссылке*, а маленькие – *по значению*, но при этом всегда стараться не допускать изменения этих параметров внутри процедуры. Другими словами, внутри процедуры оператор `x:=y` допустим, а операторы `y:=x` или `y[1]:=0` будут вызывать синтаксические ошибки. Для принудительной передачи *маленьких* константных параметров по ссылке нужно вместо **const** указывать служебное слово **constref**.

Заметим, что автор Паскаля Никлаус Вирт не включил этот, на первый взгляд очень хороший, способ передачи параметров в стандарт языка. Это сделано, чтобы не внушать программистам ложное чувство безопасности. Действительно, прямо изменить такой параметр внутри процедуры `P` невозможно. Можно, однако, выполнить оператор `Q(y)` и передать этот параметр в другую процедуру уже по ссылке (с **var**), и в процедуре `Q` спокойно менять `y`  . Отметим, что в языке Free Pascal такая защита становится совсем эфемерной, например:

```
procedure P(const x: integer);
  var y: ↑integer;
begin y:=@x; y↑:=1 { x испорчен ☹ } end;
```

Кроме того, в языке Free Pascal подпрограмма может принимать по ссылке и бестиповый параметр, т.е. ссылку на переменную любого типа. С такими переменными можно работать, только указывая их тип с помощью операции явного преобразования типа, например:

```
procedure P(var x,y); { безтиповые x и y}
  type Mas=array[1..1000] of real;
  var i: integer;
begin
  for i:=1 to 1000 do
{ 🡻🡻 приказываем считать x – массивом }
      Mas(x)[i]:=longint(y);
{ 🡻🡻 приказываем считать var x: byte; y: char; }
  byte(x):=ord(char(y)) { мистика 🕮 }
end;
```

В этом примере программист приказывает считать параметр `x` массивом типа `Mas`, а параметр `y` переменной типа `longint`, затем приказывает считать `x` переменной типа `byte`, а параметр `y` переменной типа `char`.

В языке Free Pascal существует модификация передачи параметра по ссылке, это так называемые выходные **out** параметры, считается, что при входе в подпрограмму их значения не определены, например:

```
procedure P(var x: byte; out y: byte);
begin
  write(x); x:=1; { Всё хорошо }
  x:=y; write(y); { ОШИБКА, значение y не определено }
  y:=1;           { Всё хорошо }
end;
```

Итак, вызов процедуры требует от исполнителя выполнения ряда вспомогательных действий: передача параметров, порождение и уничтожение локальных переменных, возврат на следующий оператор. Эти действия не связаны непосредственно с работой алгоритма, но требуют на этапе счёта программы определённых затрат машинного времени.

Во многих языках существует возможность перенести затраты по вызову процедур с этапа выполнения на этап компиляции программы. Для этого используются так называемые встраиваемые

(inline) или листовые (leaf) подпрограммы. Например, в языке Free Pascal применение директивы {$**inline** ON} рекомендует использование встраиваемых подпрограмм при указании в них так называемого модификатора **inline**,[1] например:

```
procedure P(var x: real; y: real); inline;
   var z: real;
begin z:=sin(y); x:=3*z+y+1 end;
```

При этом блок процедуры, со всеми операторами из её тела, уже настроенными на конкретные фактические параметры, подставляются вместо вызова этой процедуры. Таким образом, на место вызова `P(a,a+b)` подставятся операторы

```
Create(y,z); { Порождаются локальные y и z }
y:=a+b; z:=sin(y); a:=3*z+y+1;
Delete(y,z); { Уничтожаются локальные  y и z }
```

Скорость счёта возрастает, так как не надо передавать параметры, однако тратится много машинной памяти, так как теперь тело процедуры замещает каждый оператор вызова этой процедуры.

8.1.1. Объявления подпрограмм

Реальность бесконечно сложна для нашего познания. Мы должны упрощать.
Олдос Хаксли
«О дивный новый мир»

Как уже говорилось, любое имя пользователя перед его применением должно быть как-то описано выше по тексту программы. Для подпрограмм это иногда сделать невозможно, например:

```
procedure P(x: char); begin Q(x);... end;
procedure Q(x: char); begin P(x);... end;
```

Какую бы процедуру мы ни описали первой, всё равно будет ошибка, они перекрёстно вызывают друг на друга. Для выхода из этой ситуации в языки программирования (и не только в Паскаль) добавлено не описание, а объявление подпрограмм,[2] например для нашего случая надо написать:

```
procedure Q(x: char); forward;
procedure P(x: char); begin Q(x);... end;
procedure Q(x: char); begin P(x);... end;
```

Как видим, объявление процедуры Q содержит только её заголовок и так называемый модификатор **forward**, который говорит, что ниже по тексту обязательно будет «настоящее» описание этой процедуры.

8.2. Функции

Функции используются для наведения порядка в хаосе алгоритмов.
Бьёрн Страуструп, создатель C++

Функции, как и процедуры, предназначены для структуризации алгоритма. В отличие от процедуры, функция должна дополнительно возвращать в точку вызова главный результат своей работы. С точки зрения языков программирования вызов функции является выражением и,следовательно, имеет значение, а вызов процедуры является оператором. Функцию пользователя, как и процедуру, сначала надо описать (задав при этом имя функции), а потом можно вызывать с помощью указателя функции, который является частным случаем выражения.

---

[1] В языке Free Pascal (как и в других языка) у процедур и функций много модификаторов, задающих их дополнительные свойства Вскоре мы познакомимся с модификатором **forward**. Модификаторы бывают и у переменных, в разделе 9.2 мы рассмотрим модификатор `absolute`.

[2] В принципе, можно обойтись и без предварительного объявления процедур и функций, а использовать так называемый процедурный тип (см. разд. 8.4), но это более сложный путь, мы это рассматривать не будем.

В некоторых языках можно использовать так же *безымянные* так называемые λ-функции (лямбда-функции), они «одноразовые», так как вызываются только один раз сразу после описания. Впрочем, некоторые языка (например, Python) позволяют присваивать λ-функции имя, и тогда единственным отличием от «обычной» функции является возможность описать λ-функцию в любом месте программы (например, внутри выражения, прямо перед вызовом этой функции). Эти функции не имеют своих локальных переменных (кроме параметров), они в качестве локальных используют переменные того блока, в котором они описаны и выполняются.

Итак, синтаксис описания функции:

```
<описание функции> ::= <заголовок функции><блок><конец функции>
<конец функции> ::= ;
```

Как и процедура, функция определяет подзадачу, её описание очень походит на описание всей программы на Паскале.

```
<заголовок функции> ::=
    function <имя>[(<формальные параметры>)]:<имя типа результата>;
```

Формальные параметры функции описываются точно так же, как и в процедуре. В конце заголовка добавляется обязательный тип результата, который возвращает функция в точку своего вызова, замещая собой указатель функции. Например, при вызове `sin(x)` вычисленное значение этой тригонометрической функции подставляется на место указателя функции `sin(x)`.

Обратите внимание, что функции в Паскале могут возвращать величины только *именованных типов*. Таким образом, функция не может вернуть, величину безымянного типа, например, ограниченного типа `1..100` или перечислимого типа `(Red,Yellow,Green)`. Это позволяет строго контролировать тип выражения, в состав которого входит вызов функции.

Как уже говорилось, вычисленное функцией значение возвращается в точку вызова. Это накладывает ограничение на *размер* этого результата, в стандарте Паскаля допускаются только скалярные (простые) и ссылочные результаты. Другими словами функция может вычислить и вернуть что-то небольшое (одно целое или вещественное число, один символ, одну ссылку и т.д.).

Таким образом, функцию следует использовать, если у подпрограммы один результат небольшого размера, а если результатов несколько, или они большие по размеру (например, массивы), то следует использовать подзадачу-процедуру.

Как Вы узнаете в курсе по архитектуре ЭВМ, для возврата значения функции обычно используется регистр – быстродействующая ячейка памяти ЭВМ небольшого размера. В языке С функция может возвратить массив, но это «обман», так как в этом языке значением массива считаются не все его элементы (как в Паскале), а только ссылка на начало массива. В языке Free Pascal функция может возвратить значение любого именованного типа (например, строки, массива или записи). В этом случае такая функция при вызове получает по ссылке дополнительный (невидимый, служебный) параметр, содержащий адрес памяти, куда надо поместить ответ. Отметим, что так же поступает и язык С.

Любая функция обязана возвратить результат своей работы, поэтому в теле функции Паскаля должен быть хоть один оператор присваивание особого вида:

<имя функции> := <выражение>

Тип этого выражения должен быть совместим по присваиванию с типом функции, а значение выражения и задаёт возвращаемый функцией результат. Когда нет ни одного такого оператора то фиксируется синтаксическая ошибка, а когда каких операторов несколько, то функция возвращает значение последнего такого выполненного оператора. Разумеется, наличия такого оператора ещё не гарантирует, что функция обязательно выработает какой-то результат, например, для функции с именем `F` в операторе

```
if x<0 then F:=0
```

хотя формально и присутствует оператор `F:=0`, но он может и не выполнится, в этом случае, как мы догадываемся, результат работы функция *не определён*, это семантическая ошибка в программе.[1]

В отличие от многих других языков программирования, в стандарте Паскаля нет оператора возврата из подпрограмм, возврат происходит *автоматически* при достижении конца раздела операторов **end** в блоке этой подпрограммы. Впрочем, в языке Free Pascal это «исправлено», как и в языке C, добавлен оператор возврата из функции `exit`.

Такой же оператор `exit` есть и для немедленного выхода из процедур. На самом деле это просто «замаскиро́ванный» оператор перехода **goto**, который передаёт управление на *пустой оператор*, стоящий перед последним **end** блока. Аналогично работает оператор `exit` и в основной программе. Подобным же образом, оператор `break` немедленно выходит из тела цикла, это тоже «замаскиро́ванный» **goto**. В наших примерах мы такие операторы использовать не будем.

И, наконец, оператор `halt`, по аналогии с операторов `exit`, всегда делает **goto** на конечный **end**. основной программы, завершая её работу. Этот оператор полезен, если в нём указать целочисленный параметр, например, `halt(4)`. Значение этого параметра становится кодом завершения программы (`%errorlevel%`), он позволяет судить, правильно завершилась вызванная программы, или нет. Обычно ноль является «хорошим» кодом возврата, а остальные – плохими. Например, таким образом программа-компилятор сигнализирует, успешно ли закончился перевод программы на язык машины, или в программе пользователя были ошибки, которые не позволили это сделать. Обычный выход на последний **end.** возвращает `%errorlevel%=0`, а, например, при `halt(4)` будет `%errorlevel%=4`.

Важно также понять следующее. В операторе присваивания, задающим значение функции, левая часть в стандарте Паскаля не является «настоящей» переменной. Имени функции можно присваивать новое значение, но оттуда нельзя читать старое значение. Образно можно сказать, что имя функции является переменной, открытой на запись, но закрытой на чтение 🙁, как упоминалось при описании процедур в языке Free Pascal, это выходной (`out`) параметр. Таким образом, например, оператор `F:=F+1` будет скорее всего синтаксической ошибкой.

Так происходит потому, что в стандарте Паскаля запрещены пустые круглые скобки при отсутствии параметров у подпрограмм. Тогда, например, для оператора `F:=F+1` невозможно определить, является ли `F` в правой части переменной со старым значением функции, либо (рекурсивным) вызовом этой же функции. В тех языках, где разрешены пустые круглые скобки, этой проблемы нет. Например, в Фортране оператор присваивания `F=F+F()+1` чётко различает, что `F` является переменной со значением функции, а `F()` – вызовом этой функции. В языке Free Pascal тоже по умолчанию принят режим работы, в котором оператор `F:=F+F()+1` будет правильным.

Рассмотрим теперь синтаксис вызова (указателя) функции.

<указатель функции> ::= <имя>[(<фактические параметры>)]

Фактические параметры для функции точно такие же, как и для процедуры. Обратите внимание, что в Паскале возможны функции без параметров и без пустых круглых скобок (например, уже знаковые Вам стандартные функции `eoln` и `eof`).

Чтобы лучше понять отличие процедур от функций, введём следующее определение. Состоянием программы (program state) в текущей точке выполнения является (кроме самой этой текущей точки) значение всех её существующих в данный момент переменных.[2] Это своеобразный «моментальный снимок» памяти переменных программы, запомнив состояние программы можно надолго прервать её

---

[1] Из курса по архитектуре ЭВМ Вы потом узнаете, что в этом случае возвращается (случайное) значение определённого регистра процессора ЭВМ.

[2] Сюда включаются и значения особых файловых переменных (файловых дескрипторов), каждый из которых определяет состояние одного из потоков ввода/вывода программы. С этими переменными мы познакомимся при изучении сложной структуры данных – файлов. Кроме того, у программы могут быть и особые, так называемые переменные окружения, с помощью которых программа общается с «внешним миром». В курсе по архитектурам ЭВМ Вы познакомитесь и с важными регистровыми переменными.

выполнение и затем возобновить его с прерванной точки. Операторы программы могут изменять значения переменных и, следовательно, состояние программы [iii] [см. сноску в конце главы].

Например, если после выполнения оператора процедуры состояние программы не изменилось, то он эквивалентен пустому оператору. Рассмотрим пример такой «пустой» процедуры:

```
procedure P(x: integer);
  var y: real;
begin y:=x+1; x:=2+trunc(y) end;
```

Как видим, в этой процедуре меняются значение только *локальных* переменных, которые уничтожаются при выходе из этой процедуры. Таким образом, если процедура не меняет значение ни одной нелокальной переменной, то она «ничего не делает».[1] А вот для функции это уже не так, функция обязана вырабатывать некоторое значение, являющееся результатом её работы. Таким образом, даже если сама функция и не меняет состояние программы, но её возвращаемое значение участвует в вычислении выражения, что может изменить это состояние.

Итак, функция обязана возвращать вычисленный результат, это значение в Паскале необходимо как-то использовать, его нельзя просто «выбросить». Например, возвращаемый результат функции `f(x)`, можно использовать так:

`y:=f(x);` `write(f(x));` **`if`** `f(x)>0` **`then`** ... и т.д.

А вот, например, написать `a:=0; f(x);` будет в стандарте Паскаля синтаксической ошибкой.

Есть языки, в которых каждое выражение считается ещё и оператором. Например, в языке C можно писать в программе такие «операторы»

`sin(x); 2+x; 123;` и т.д.

При этом эти выражения вычисляются, а результаты вычислений просто выбрасываются. Кроме того, так как компилятор обычно работает в режиме оптимизации, то такие выражения и вызовы таких функций просто удаляются из программы, как не оказывающие никакого влияния на алгоритм. Исключение составляют выражения, в которые входят переменные класса памяти **`volatile`** (см. раздел 3.3.3) и функции, у которых может быть так называемый побочный эффект, о котором будем говорить далее.

К сожалению, *по умолчанию* в языке Free Pascal включён режим `{$X+}` и тоже можно вызывать функцию как оператор, например `a:=0; f(x);`. Это, однако, можно запретить, выдав директиву `{$X-}`. А вот такие «операторы» `a+1; 11`; и т.д. к счастью, будут в языке Free Pascal ошибочными всегда.

8.3. Использование процедур и функций

Глубоко ошибается тот, кто думает, что изделиями программиста являются программы, которые он пишет. Программист обязан изготовлять заслуживающие доверия решения и представлять их в форме убедительных доводов... А текст написанной программы является лишь сопроводительным материалом, к которому эти доказательства применимы.

Эдгар Дейкстра

Подпрограммы являются очень мощным инструментом языков программирования, и важно научиться пользоваться этим инструментом правильно. Рассмотрим это на простой задаче. Пусть задана целая константа `N>0`. Необходимо ввести три вектора `A`, `B` и `C`, в каждом по `N` вещественных чисел. Обозначим через `(P,Q)` скалярное произведение векторов (dot product) этих векторов, т.е.

---

[1] Замечание на будущее. *Нелокальная* переменная не обязательно является *глобальной*, потом мы познакомимся с динамическими переменными, которые не являются ни локальными, ни глобальными. Они, однако, тоже являются для процедуры *нелокальными* (ну, как всё запутано 😉).

$$(P,Q)=\sum_{i=1}^{N} P[i]*Q[i]$$

Необходимо вычислить величину `S=(X,X)-(Y,Z)`, где в качестве каждого из векторов X, Y и Z выступает один из введённых векторов A, B или C. Конкретное значение для вектора X (будет ли это A, B или C) определяется по следующему правилу. В каждом векторе находится максимальное значение, обозначим их как Amax, Bmax и Cmax, далее, за вектор X берется тот из векторов A, B или C, для которого этот максимум *самый маленький*. В качестве векторов Y и Z берутся два оставшихся вектора, скалярное произведение коммутативно, так что неважно, какой из них будет Y, а какой Z.

В качестве «учебного» предусловия программы будем предполагать, что при вводе данных «все будет хорошо», т.е. в стандартном входном потоке находятся `3*N` правильных лексем вещественных чисел, а при вычислении скалярных произведений и их разности никогда не произойдёт выход за границу допустимых значений.

Далее необходимо выполнить спецификацию задачи. Ясно, что особым будет случай, когда сразу два или три введённых вектора имеют одинаковый максимум. Давайте в этом случае отдавать предпочтение вектору с «меньшим» именем, т.е. вектор A будет предпочтительнее, чем B, а B чем C.

На следующем этапе необходимо разбить задачу на подзадачи, каждая из которых является законченным шагом алгоритма. Это можно сделать по-разному. Например, в качестве таких подзадач можно выбрать 1) ввод вектора, 2) вычисление максимальной величины вектора и 3) вычисление итоговой разности S. В последней подзадаче при её дальнейшей детализации можно было бы выделить подзадачу вычисления скалярного произведения, но для простоты мы не будем этого делать. Теперь перейдём к следующему этапу разработки программы. Будем записывать подзадачи в виде процедур и функций.

Первая подзадача ввода вектора возвращает введённый массив, поэтому должна быть реализована в виде процедуры. Вторая и третья подзадачи, наоборот, имеют один небольшой результат – вещественное число, поэтому их следует реализовывать в виде функций. Первую процедуру надо вызвать из главной программы три раза для массивов A, B и C (и именно в таком порядке). Вторую функцию надо тоже вызвать три раза (уже в любом порядке). Ниже приводится текст программы для решения этой задачи, важные для понимания места, которые будут поясняться, отмечены значками ❶, ❷ и т.д.

```
const ❸ N=1000000;
type Vec=array[1..N] of real;
var  A,B,C: Vec; Amax,Bmax,Cmax,S: real;
procedure Vvod(var X: Vec; ❶ N: integer);
  var i: integer;
begin for i:=1 to N do ❹ read(X[i]) end;
```

```
function Max(var X: Vec; ❶ N: integer): real;
  var i: integer; M: real;
begin ❷ M:=X[1];
  for i:=2 to N do if X[i]>M then M:=X[i];
  Max:=M
end;
```

```
function RSP(var X,Y,Z: Vec; ❶ N: integer): real;
  var i: integer; Rez: real;
begin ❷ Rez:=0.0;
  for i:=1 to N do Rez:=Rez+X[i]*X[i]-Y[i]*Z[i];
  RSP:=Rez
end;
begin { раздел операторов главной программы }
  Vvod(A,N); Vvod(B,N); Vvod(C,N);
  Amax:=Max(A,N); Bmax:=Max(B,N); Cmax:=Max(C,N);
  if (Amax<=Bmax) and (Amax<=Cmax)
    then { X это A } S:=RSP(A,B,C,N)
    else if (Bmax<=Cmax)
            then { X это B } S:=RSP(B,A,C,N)
```

```
            else { X это C } S:=RSP(C,A,B,N);
  writeln('S=',S)
end.
```

Обсудим эту программу. В стандарте Паскаля в точках ❷ необходимо использовать вспомогательную локальную переменную, так как одним именем функции, как уже рассказывалось, не обойтись.

В языке Free Pascal, как уже говорилось, функцию `Max` можно писать проще, вспомогательная переменная ❷ не нужна:

```
function Max(var X: Vec; N: longword): double;
  var i: longword;
begin Max:=-inf; { это -1.0/0.0 т.е. -∞ ! }
  for i:=1 to N do
    if X[i] > Max then Max:=X[i]
end;
```

Далее у учащихся обычно возникает вопрос, зачем в подпрограмму передавать параметр ❶ `N`, задающий длину вектора. Действительно, если этот параметр опустить, то внутри подпрограмм становится видимой одноимённая глобальная константа ❸ `N`, и всё будет работать по-прежнему. Для того чтобы понять желательность этого параметра, необходимо сначала сформулировать один из основных принципов в разработке подзадач, оформленных в виде подпрограмм.

Как известно, в большинстве языков программирования высокого уровня, тела подпрограмм являются блоками, так что извне всё внутри них, включая имена формальных параметров, становится невидимыми. Это одно из проявлений принципа инкапсуляции (encapsulation), который в данном случае заключается в том, что внутренняя реализация подпрограмм невидима извне и может быть изменена при смене реализации подзадачи (например, при исправлении ошибки или для оптимизации алгоритма).

> Для хорошо написанных подпрограмм, однако, должно быть верным и обратное: они не должны сами (по собственной инициативе) использовать в своей работе ничего из «внешнего мира», т.е. все свои входные данные они получают в качестве параметров, и все результаты возвращают тоже через свои параметры (и значение функции). Это позволяет минимизировать и стандартизировать *связи* между подзадачей и остальной программой.

Для приведённого примера подпрограммы не должны знать, как в основной программе задаётся длина вектора. Это позволяет, например, свободно изменять в программе имя константы, задающей длину вектора, или вызывать процедуру ввода в виде `Vvod(X,N div 2)`, чтобы ввести только первую половину вектора и т.д. При необходимости можно пойти ещё дальше и передавать в подзадачу начальный и конечный индексы вектора:

```
procedure Vvod(var X: Vec; K,N: integer);
```

и записать цикл ввода в виде

```
for i:=K to N do read(X[i])
```

Это позволит осуществить ввод данных в любую часть вектора. Запись подзадач в таком обобщенном виде позволит использовать их и в других программах, что называется повторным использованием (reuse) реализованного программного обеспечения.

В точке ❹ учащиеся часто делают характерную ошибку, они используют вместо оператора `read(X[i])` оператор `readln(X[i])`. При этом для задачи делается дополнительная спецификация, что в потоке `input` каждое вводимое число располагается в отдельной строке, что, конечно, ниоткуда не следует и во многих задачах не выполняется.

Язык Free Pascal позволяет наряду с обычными массивами использовать и так называемые динамические массивы, например:

```
const N=1000;
type
  VS=array[1..N] of real; {статический массив}
```

```
  VD=array of real;       {динамический массив}
var
  X: VS; Y: VD;
  Z: array of array of real; {динамическая матрица}
```

Размер динамического массива при описании не задаётся, он порождается пустым (не содержит ни одного элемента). Таким образом, переменная `Y` является не самим массивом (как переменная `X`), а только (как в языке C) ссылкой на начало массива, причём `Y` порождается со значением **`nil`** (см. главу 12). Перед использованием этого массива программист должен породить его, задавая нужное число элементов оператором

```
SetLength(Y,1000);   { Y[0..999] }
SetLength(Z,20,30); { Z[0..19,0..29] }
```

При первом выделении памяти под динамический массив, она очищается (нулями). Элементы динамического массива всегда нумеруются, начиная с нуля. Затем в процессе счёта можно выделить массиву больше памяти, чем он занимал ранее, например, `SetLength(Y,2000)`. При этом сначала выделяется новая область памяти бóльшего размера, она очищается нулями, затем массив `Y` копируется из старой области памяти в новую, а лишь потом старая область памяти уничтожается (объявляется свободной и может быть использована для размещения других переменных).

Таким образом, это не «настоящее» изменение размера переменной, а порождение новой переменной бóльшего размера. Это весьма трудоёмкая операция! Разумеется, если уменьшить размер массива, например, `SetLength(Y,500)`, то новая область памяти не выделяется, просто помечается, что изменился её размер. Возможно принудительное уничтожение динамического массива оператором `SetLength(Y,0)` (есть более короткий способ `Y:=nil`).

Ранее упомянутые стандартные функции `low` и `high` возвращают начальный (нижний) и конечный (верхний) индексы любого массива (элементы которого считаются пронумерованными с нуля), например:

```
low(X)=0     high(X)=999     low(Y)=0   high(Y)=499
high(Z)=19   high(Z[3])=29
```

Полезной является также стандартная функция `length`, возвращающая число элементов массива, например:

```
length(X)=1000   length(Y)=500
length(Z)=20     length(Z[15])=30
```

Любопытно, что динамическая матрица может иметь строки разной длины 🙄, например:

```
SetLength(X[4],40); SetLength(X[7],50); ...
```

Таким образом, двумерный динамический массив больше похож не на прямоугольную матрицу, а на «текст» из строк переменной длины, некоторые из которых пустые. Отметим, что такой же «фокус» можно сделать и в языке C.

Динамический массив можно сделать формальным параметром подпрограмм, в этом случае он называется параметром – открытым массивом, например:

```
procedure Vvod(var x: array of real);
```

Здесь тот редкий случай, когда язык Free Pascal позволяет использовать формальные параметры безымянного типа. На место такого формального параметра – открытого массива можно передавать любые одномерные вещественные массивы, например:

```
Vvod(X); Vvod(Y); Vvod(X[100..300]);
Vvod(Y[200..400]); Vvod(Z[3]); Vvod(Z[6][5..20]);
```

Как видно, фактическим параметром открытого массива может быть не весь фактический массив, в только его часть (сегмент).

В некоторых языках над выбранным сегментом массива можно проводить преобразования, например, в языке Fortran-95 можно использовать так называемый индексный триплет, он задаёт нижнюю и верхнюю границы и шаг выборки элементов (с символа ! в Фортране начинается комментарий):

```
real x(100)  ! var x: array[1..100] of real;
x(10:20)     ! x[10..20]
x(20:30:3)   ! x[20,23,26,29]  { с шагом 3 }
```

```
x(40:44:-1) ! x[44,43,42,41,40] { в обратном порядке }
```

Далее, в языке Free Pascal можно не передавать в подпрограмму длину массива явно, а использовать для определения этой длины стандартные функции `low` и `high`. Так, у нас будет:

```
Vvod(X);           { low(X)=0, high(X)=999 }
Vvod(Y);           { low(Y)=0, high(Y)=499 }
Vvod(X[100..300]); { low(X)=0, high(X)=200 }
Vvod(Y[200..300]); { low(Y)=0, high(Y)=100 }
```

Кроме того, можно узнать, и как описаны соответствующие типы массивов в программе:

```
low(VS)=1, high(VS)=1000, low(VD)=0, high(VD)=-1
```

Это позволяет, например, принимать в подпрограмме «хитрые» решения, скажем, фиксировать ошибку, если на обработку в подпрограмму передано менее 20% «настоящей» длины массива. При этом цикл ввода следует писать в виде

```
for i:=low(X) to high(X) do read(X[i])
```

или

```
for i:=0 to length(X)-1 do read(X[i])
```

Заметим, что для пустого массива `low(X)=0` и `high(X)=-1`, так что цикл не выполниться ни одного раза.

Так и рекомендуется работать в языке Free Pascal с массивами в подпрограммах.

Будьте осторожны, динамический массив всегда передаётся в подпрограмму по ссылке, даже если мы уберём у параметра **var**! Отсутствие **var**, однако, приводит к тому, что фактический параметр может быть массивом-константой, например:

```
procedure P(x: array of integer);
begin ... end;
  var i: integer;
begin P([1,4,3*i+1]); ... end.
```

Язык Free Pascal позволяет использовать особые параметры подпрограмм, так называемые массивы констант **array of const**. Фактический параметр такого типа является массивом переменной длины, элементами которого являются (константные) [1] значения разных скалярных типов, например, процедуру с заголовком

```
procedure P(X: array of const);
```

можно вызывать как

```
var x: longint; s: string='Привет!';
P([x+1,succ('#'),s,'Hellow',1<2,...])
```

Как и для динамического массива, максимальный индекс элемента возвращает функция `high(X)`, для пустого массива это `-1`. Каждый элемент такого массива считается полем записи с вариантами (см. разд. 9.2):

```
TVarRec=record
  case VType: Ptrint of
    vtInteger : (VInteger: Longint);
    vtBoolean : (VBoolean: Boolean);
    vtChar    : (VChar: Char);
    vtExtended: (VExtended: PExtended);
    vtString  : (VString: PString);
    vtPointer : (VPointer: Pointer);
    vtVariant : (VVariant: PVariant);
    vtInt64   : (VInt64: PInt64);
    ...
```

[1] Название типа может ввести в заблуждение: это не массив констант, а массив элементов, которые нельзя менять. Впрочем, некоторые из этих элементов ссылочного типа, и, хотя саму ссылку менять и нельзя, но можно изменить значение переменной, на которую указывает эта ссылка ⚠.

```
end;
```

Таким образом, тип каждого элемента массива равен `X[i].VType`, а значение (или ссылка на значение) равно имени поля записи с вариантами соответствующего типа. Например, пусть процедура `Print` выводит значения всех элементов своего параметра-массива:

```
procedure Print(X: array of const);
  var i: integer;
begin
  for i:=0 to High(X) do
    case X[i].VType of
      vtInteger: Writeln(VInteger);
      vtBoolean: Writeln(VBoolean);
      vtChar:    Writeln(VChar);
      vtString:  Writeln(VString↑);
      ...
    end;
end;
```

8.4. Передача процедур и функций как параметров

Зная четыре параметра, я могу создать слона, а зная пять – заставить его размахивать хоботом.

Джон фон Нейман

Разберём теперь как передавать процедуры и функции в качестве параметров в подпрограммы, такие подпрограммами называются подпрограммами *высших порядков* (higher-order functions). По существу, передача подпрограмм эта передача ссылочных значений, хотя слово **var** и не пишется (см. глава 12), так как у каждая процедуры и функции есть ссылка на её месторасположение (адрес начала) в памяти ЭВМ.[1] В языке Free Pascal для этого предусмотрен специальный процедурный тип данных. Этот тип определяется, как заголовок процедуры или функции, но без имени, например:

```
type Mas=array[1..1000000] of real;
     Func=function (x: integer; y: char): real;
     Proc=procedure (var x: Mas; n: integer);
     ProcWithoutParams=procedure;
var  p1,p2: Proc; f: Func;
```

Таким образом, каждая подпрограмма является константой ⚠ (т.е. конкретным значением) своего процедурного типа, например, процедура

```
procedure Zero(var x: Mas; n: integer);
  var i: integer;
begin for i:=1 to n do x[i]:=0 end;
```

является константой ⚠ типа `Proc`. Для присваивания процедурным переменным значения необходимо явно указывать унарную операцию взятия ссылки `@` (или воспользоваться функцией взятия адреса `Addr`):

```
p1:=@Zero { или p1:=Addr(Zero) }
```

Теперь можно, например, описать процедуру с заголовком

```
procedure MyProc(x: integer; y: proc);
```

и вызывать её операторами `MyProc(1,@Zero)` или `MyProc(-2,p1)`. Как обычно, можно передавать параметры процедурного типа по ссылке и по значению, например:

```
type Proc=procedure;
procedure P1; begin Write('P1':3) end;
```

---

[1] Подпрограммы могут располагаться в специальных динамических библиотеках (Dinamic Link Libraries), тогда у программы на них есть только так называемые косвенные ссылки (см. главу 12).

```
procedure P2; begin Write('P2':3) end;
procedure P(var x: proc; y: proc);
begin x; y; x:=@P2; y:=@P1 end;
  var pr1: Proc=@P1; pr2: Proc=@P2;
begin
  P(@P1,@P2); { ОШИБКА, @P1 не переменная }
{ @P2 тоже не переменная, но ошибки нет,
  т.к. она передаётся по значению }
  P(pr1,pr2); { вывод,  ␣␣P1␣P2 }
  P(pr1,pr2); { вывод,  ␣␣P2␣P2 }
```

Кроме того, функция может вырабатывать и значение процедурного типа, например:

```
  type func=function (x,y: integer): integer;
  var a: integer;
  function G(x,y: integer): integer;
  begin G:=x+y end;
  function F(x: integer): func;
  begin if x<2 then F:=@G end;
begin a:=F(1)(3,4) {a=7} end;
```
[1]

При использовании подпрограмм высших порядков могут возникать серьёзные проблемы. Во-первых, для модульной программы производится раздельная компиляция и непонятно как проконтролировать число и типы передаваемых в подпрограмму параметров. Здесь есть и более серьёзные проблемы, в частности, так называемая Фунарг проблема [iv] [см. сноску в конце главы].

8.5. Функции с побочным эффектом

Обычно счастье – это побочный эффект
другой деятельности.

Олдос Хаксли
«О дивный новый мир»

Как мы уже выяснили, главным результатом работы функции является возвращаемое значение. Когда других результатов нет, то состояние программы (вспомните, что это такое ?) после вызова этой функции не изменяется. В этом случае говорят, что функция не имеет побочных эффектов. Таким образом, побочный эффект (side effect) функции – это изменение ей нелокальных переменных, т.е. некоторые переменные изменяют свои значения после возврата из функции [v] [см. сноску в конце главы]. Разумеется, если *процедура* не имеет такого «побочного эффекта», то она вообще ничего не делает, поэтому для неё понятие побочного эффекта не применяется.

Можно говорить и о более общем случае побочного эффекта при вычислении не только функции, но и любого выражения. Выражение имеет побочный эффект, если, кроме вычисления своего значения оно меняет значения переменных, которые продолжают существовать после вычисления этого значения (в частности, производится ввод или вывод данных). Для языка Паскаль побочный эффект могут иметь только входящие в выражения функции, однако в других языках это может быть и не так. Например, в языке C операторы присваивания могут входить в выражения, обеспечивая побочный эффект:

`3*❶(x=2)  ❷-i++` или `f(x)❶+g(y)❷;`

В некоторых языках (например, в языке C) вводится понятие точка следования (sequence point), это такая точка в тексте программы, в которой все побочные эффекты *предыдущих* вычислений уже проявились, а все побочные эффекты *последующих* вычислений ещё отсутствуют. Например, в приведённых выше выражениях ❷ это точки следования, а ❶ нет. Далее вы ещё будем об этом горорить.

---

[1] Для продвинутых читателей. В этом примере функция F возвращает в качестве результата своей работы уже *существующую* функцию, описанную или объявленную где-то выше в программе. Есть, однако, и языки программирования (например, Lisp), функция в котором может возвратить как результата своей работы *новую*, только что построенную функцию, которую потом можно сразу вызывать для выполнения ⚠.

Во всех книгах по программированию говорится, что побочный эффект у функции не желателен, разберёмся, почему это так. В качестве примера рассмотрим простую программу, содержащую функцию с побочным эффектом:

```
var ❶ a: integer;
function F(x: integer): integer;
  begin ❶ a:=a+1; F:=x*a end;
begin a:=1; write(F(a+1)+F(a-1)) end.
```

Как видим, наша функция в качестве побочного эффекта меняет значение глобальной переменной ❶ `a`. Определим, что выведет программа, для этого выполним её трассировку (пошаговое выполнение):

```
a=1; F(a+1=2)={x=2; a=2; F=2*2=4}
a=2; F(a-1=1)={x=1; a=3; F=1*3=3}
write(4+3)=7
```

А теперь обратим внимание, что параметр процедуры `write(F(a+1)+F(a-1))` является суммой двух величин, а операция сложения коммутативна. Другими словами, ничего не должно измениться, если мы напишем вывод в виде `write(F(a-1)+F(a+1))`. Снова сделаем трассировку программы:

```
a=1; F(a-1=0)={x=0; a=2; F=0*2=0}
a=2; F(a+1=3)={x=3; a=3; F=3*3=9}
write(0+9)=9 ⚠
```

Так что же напечатается? Ясно, что вывод будет разный, когда исполнитель сначала вычислит первое слагаемое параметра функции, а затем второе, или же наоборот.[1] Мы уже знаем, что в машинной (дискретной) математике ассоциативный и дистрибутивный законы арифметики не выполняются, т.е. в общем случае `(a+b)+c≠a+(b+c)` и `a*(b+c)≠a*b+a*c`. Машинные команды любого компьютера, однако, всегда соблюдают закон коммутативности (по сложению и умножению). А вот язык программирования, получается, его не выполняет. Это очень плохо, но, по счастью, в Паскале это происходит только для функций с побочным эффектом.

Как уже говорилось ранее, круглые скобки не спасают при одинаковом старшинстве операторов, например, `q:=x+(y+z)` может вычисляться компилятором и как `q:=(x+y)+z`. Единственный способ гарантировать нужный результат, это использовать вре́менную переменную:

```
temp:=y+z; q:=x+temp
```

Аналогично `write(F(a+1)+F(a-1))` надо записать как:

```
t:=F(a+1); write(t+F(a-1))
```

В общем случае побочный эффект может приводить к *неопределённому поведению* программы. Например, если в предыдущей программе добавить

```
y:=F(a+1)+F(a-1); if y=7 then P(x) else Q(x)
```

Здесь неизвестно, какая из двух процедур (`P` или `Q`) будет вызвана.

Кардинальным решением проблемы может быть требование всегда вычислять выражения слева направо, так, например, сделано в языке Java. Это, однако, закрывает много возможностей по оптимизации программы компилятором.

Можно отметить, что в терминах введённого нескольно выше понятия *точки следования* в выражении `F(a+1)+F(a-1)` ❶ в точке ❶ уже проявился весь побочный эффект от вызовов фукции `F`.

Далее, в математике мы привыкли, что переменная изменяет своё значение только тогда, когда некоторый оператор (явно) меняет это значение на новое. Посмотрим, что будет:

```
a:=1; b:=c; write(a)
```

---

[1] Для продвинутых читателей. Разные компиляторы даже с одного и того же языка программирования могут вычислять сумму в разном порядке (слева направо или справа налево). В основном это связано с тем, преобразуют ли они выражения для их последующего вычисления в бесскобочную, так называемую прямую или обратную польскую запись, об этом уже упоминалось ранее.

Любой «нормальный» программист ожидает, что `write(a)` напечатает единицу, но, если в программе допустить побочный эффект функций, то это уже не гарантируется, например:

```
function c: integer; begin a:=a+1; c:=1 end;
```

Другой пример, который часто встречается в задачниках по программированию: что выведет оператор `write(x=x)` ? Требуется привести пример, когда этот оператор выведет логические значение FALSE, а не TRUE.

Интересно, что на наших компьютерах тоже существуют вещественные числа, каждое из которые при сравнении с собой даёт значение **false**! Правда, они и называются «не числами» NaN (Not a Number), о них уже немного рассказывалось в разд. 3.14.2, посвященном вещественному типу данных. Ясно, что проверку вещественной переменной X на значения NaN нельзя проводить обычным сравнением, так как `X=NaN` всегда даст **false**, Не поможет и битовое сравнение (скажем, машинной командой **test**), так как у NaN много разных битовых представлений, например, для 4-байтного вещественного типа single есть около 2^{23} NaN с разной кодировкой! Можно, однако, делать сравнение `X<>NaN`, так как оно даст **false** только, когда в X храниться NaN ⚠.

В языке Free Pascal (при подключённом модуле Math) доступна константа с именем NaN (при печати выводит лексему-имя Nan) и стандартная логическая функция `IsNaN(X)`.

Ещё один пример, фрагмент программы:

```
x:=1; if a<0 then Write(x)
```

вполне может вывести значение, отличное от единицы. Получается, что, глядя на текст программы, становится очень сложно понять алгоритм её работы. Теперь для Вас должно быть понятно, почему функции с побочным эффектом лучше в программах не использовать.

8.6. Рекурсивные процедуры и функции

Любое дело требует больше времени, чем казалось в начале, даже если Вы учитывали при этом закон Хофштадтера.

Закон Хофштадтера

Итерация свойственна человеку, рекурсия божественна.

Питер Дойч, создатель Ghostscript 🤔

Как мы уже знаем, одним из этапов обработки вызова подпрограмм является выполнение блока, входящего в эту подпрограмму. Выполнение блока заканчивается во время выхода из подпрограммы. Когда до выхода из блока подпрограммы производится повторный вход в начало этой же подпрограммы, то говорят о рекурсивном вызове этой подпрограммы.

Различают *прямую* (direct) и *косвенную* (indirect) рекурсию. Прямая рекурсия подразумевает, что подпрограмма непосредственно вызывает сама себя из своего блока. Косвенная рекурсия означает, что вызов производится через цепочку других подпрограмм, например:

P → Q → V → P

Другими словами, подпрограмма P вызывает подпрограмму Q, та вызывает подпрограмму V, а уже V снова вызывает подпрограмму P.

Прямой рекурсивный вызов *процедуры* программист может, при желании, заблокировать, сделав имя процедуры *невидимым* в блоке этой процедуры, например:

```
procedure P;
  var P: char;
begin P:='*'; P { ОШИБКА, имя процедуры не видно }
end;
```

А вот с *функцией* Паскаля такой фокус уже не сработает:

```
function F(x: integer): integer;
  var F: char; { ОШИБКА, в функции есть оператор F:=... }
begin F:=1; x:=F(x) { ОШИБКА компиляции }
```

```
end;
```

Рекурсивный вход в блок подпрограммы до выхода из этого блока приводит к тому, что во время счёта повторно вызываемый блок *вкладывается* внутрь уже работающего блока. Вскоре мы подробно рассмотрим возникающие при этом эффекты.

В качестве примера, во многих книгах по программированию рассматривается задача вычисления факториала целого числа

`Factorial(n)=1*2* ... *n`

Эту задачу следует реализовывать в виде функции, например, с заголовком

```
function Fact(n: integer): integer;
```

По определению, `Factorial(0)=1`. Особым случаем является значение `n<0`, пусть, например, в этом случае функция тоже будет возвращать единицу.

В математике функция факториала расширяется на отрицательные значения аргумента в комплексной плоскости, мы этого, естественно, делать не будем. В стандарте Паскаля можно описать тип аргумента этой функции в виде:

```
type NotNeg=0..Maxint;
function Fact(n: NotNeg): NotNeg;
```

В языке Free Pascal для аргумента этой функции естественно использовать небольшой (2-х байтный) беззнаковый тип `word`, а для результата – самый большой беззнаковый диапазон целых чисел `qword`:

```
function Fact(n: word): qword;
```

Самая простая и эффективная реализация этой функции делается в виде цикла, например, вот в стандарте Паскаля:

```
function Fact(n: integer): integer;
  var i,temp: integer;
begin temp:=1;
  for i:=2 to n do temp:=i*temp;
  Fact:=temp
end;
```

Следуя спецификации, наша функция для параметра `n<2` возвращает значение единица, так как тело цикла не выполняется ни одного раза. Здесь следует также обратить внимание на типичную ошибку учащихся, которые в стандарте Паскаля записывают цикл в виде

`Fact:=1; for i:=2 to n do Fact:=i*Fact;`

Оператор `Fact:=1` вполне допустим, а вот оператор `Fact:=i*Fact` вызовет в стандарте Паскаля синтаксическую ошибку, так как предписывает читать значение из имени функции `Fact`, что в стандарте Паскаля невозможно.

Далее, в математике существует и рекурсивное определение функции факториал (при $n \geq 0$)

`Factorial(0)=1; Factorial(n)=n*Factorial(n-1)`

Вот для такого определения факториала и подходит его реализация в виде рекурсивной функции. Если делать реализацию «в лоб», то получится, например, такая функция:

```
function Fact(n: integer): integer;
begin
  if n<2 then Fact:=1 else Fact:=n*Fact(n-1)
end;
```

Как видно, эта функция описывается очень просто и хорошо соответствует математическому описанию факториала.

Современные компьютеры, там где это возможно, предпочитают алгоритмы с условными операторами сокращённого вида, т.е. вместо оператора **`if ... then ... else`** предпочитают оператор **`if ... then ...`** (причины этого раскрываются в курсе по архитектурам ЭВМ). Исходя из этого лучше реализовать функцию факториала в виде:

```
function Fact(n: integer): integer;
begin Fact:=1;
  if n>1 then Fact:=n*Fact(n-1)
end;
```

Здесь в большинстве случаев выполняется «лишнее» присваивание `Fact:=1`, так как чаще всего `n>1`, однако для компьютера это выгоднее, чем реализовывать дополнительную ветвь условного оператора.

Работу рекурсивной функции факториала рассмотрим на простом примере её вызова оператором вывода `write(Fact(4))`. Трассировка программы приводит к картине последовательного выполнения блоков функции, показанной на рис. 8.1.

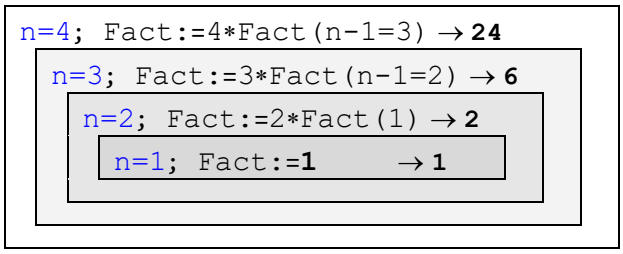


Рис. 8.1. Вложенные блоки функции `Fact`

На этом рисунке видно, как при рекурсивном вызове каждый новый вход в блок порождает и новый комплект локальных переменных (в нашем случае это только формальный параметр `n`).[1] Количество одновременно существующих блоков называется глубиной рекурсии, в нашем примере глубина рекурсии равна четырём. При максимальной глубине рекурсии у нас существуют четыре одноимённых переменных с именем `n`, но, по свойству видимости, находясь в каждом блоке, исполнитель видит только одну (самую внутреннюю) такую переменную, а остальные, хотя и существуют, но невидимы.

Как мы уже знаем, локальные переменные имеют автоматический класс памяти, они порождаются при входе в подпрограмму и автоматически уничтожаются при выходе. Таким образом, при рекурсивном вызове порождается новый комплект локальных переменных, потребляя память ЭВМ, а при возврате из блоков эти переменные уничтожаются, освобождая память.[2]

В качестве следующего примера рассмотрим такую задачу. Надо вводить из потока `input` целые числа, пока не будет введён ноль (признак конца ввода). Потом надо вывести все введённые положительные числа, но в обратном порядке: сначала последнее положительное, потом предпоследнее и т.д.

При осознании задачи надо понять, что в памяти надо обязательно хранить все введённые положительные числа. Невозможно, однако, заранее отвести необходимый объём памяти для их хранения, так как количество таких чисел неизвестно. «Хитрые» учащиеся предлагают здесь создать целочисленный массив максимального (входящего в память) размера. Этот метод не подходит по двум причинам. Каждый компьютер обладает своим размером памяти, на одном всего жалких `0.25` Гб, на другом `2` или `3` Гб.[3] Это, однако, не главное, достаточно немного усложнить задачу и сказать, что надо ввести не одну, а несколько последовательностей данных переменной длины. Ясно, что создать несколько массивов максимально возможного размера нельзя.

Единственным выходом из положения является брать участки памяти в процессе счёта, по мере ввода порций данных. Рассмотрим решение нашей задачи с помощью локальных переменных, которые автоматически порождаются при входе процедуру и уничтожаются при выходе. Будем вызывать процедуру каждый раз, когда нам понадобится новая переменная для размещения вводимых данных:

---

[1] В отличие от лексической (в тексте программы) области видимости (lexical scope) при рекурсивном вызове образуется новая динамическая область видимости (dynamic scope).

[2] Из курса по архитектуре ЭВМ Вы узнаете, что для реализации автоматического класса памяти лучше всего подходит так называемый аппаратный стек компьютера.

[3] В системе программирования на языке Free Pascal в 32-битном режиме работы у программы примерно `1.5` Гб памяти для размещения статических и динамических переменных. В 64-битном режиме может быть доступно `16` Гб памяти и больше, из них до `8` Гб может быть отдано программе пользователя.

```
procedure P;
  var x: integer;
begin
  repeat read(x);
    if x>0 then begin { новая порция }
      P; write(x:4); x:=0 { для выхода из цикла }
    end { else отрицательное число выбрасывается }
  until x=0
end; { конец процедуры P }
begin { основная программа } P end.
```

Итак, при вводе числа больше нуля, мы оставляем его в локальной переменной x и рекурсивно вызываем процедуру P, порождая новую локальную переменную x. При вводе нуля мы начинаем последовательный возврат из процедур, печатая положительные числа в обратном порядке. Например, для ввода

```
1 12 -7 -322 18 -111 -4 4 8 -2 0
```

будут последовательно порождаться локальные переменные (присвоим им индексы для лучшего понимания механизма работы):

x_1=**1** x_2=**12** x_3=-7; -322; **18**

x_4=-111; -4; **4** x_5=**8** x_6=-2; **0**

Далее эти числа будут печататься в обратном порядке (по 4 позиции на число):

```
   8   4  18  12   1
```

Из рассмотренного механизма работы рекурсивного вызова следует, что глубина рекурсии не может быть слишком большой, так как при каждом вызове расходуется память для размещения локальных переменных. На первый взгляд, это совсем маленький объём памяти, для нашей функции Fact это переменная n (обычно 4 или 8 байт). К сожалению, реализация рекурсии на ЭВМ требует, помимо явных локальных переменных, каждый раз размещать в памяти около десятка служебных переменных (невидимых программисту на языке высокого уровня). Необходимость таких переменных Вы поймёте из курса по архитектуре ЭВМ. В итоге получается, что каждый рекурсивный вызов требует выделения не менее 50-ти байт автоматической (стековой) памяти компьютера. Под автоматическую память обычно выделяется небольшой объём, порядка миллиона байт, поэтому максимальная глубина рекурсии составляет примерно 20 000 вызовов.

Для примера вычисления факториала это не критично, так как значение факториала выйдет за представимый диапазон целых чисел qword значительно раньше, чем будет достигнута максимальная глубина рекурсии. Для нашего второго примера количество положительных чисел во вводимой последовательности ограничено примерно 20 000, это приходится учитывать в спецификации задачи.

Итак, для реализации рекурсивного алгоритма следует переформулировать задачу в рекурсивном виде. Рассмотрим, например, задачу поиска суммы элементов вещественного вектора. Для этого, например, можно предложить такую функцию (со «щадящим» предусловием «пусть всё будет хорошо»), реализующую алгоритм

$$Sum = \sum_{i=1}^{N} x[i]$$

```
function Sum(var x: Vec; n: integer): real;
   var i: integer; S: real;
begin S:=0.0;
      for i:=1 to n do S:=S+x[i];
      Sum:=S
end;
```

Теперь переформулируем задачу поиска суммы элементов в рекурсивном виде (мы делаем естественное предусловие n>0):

```
Sum(x,1)=x[1]
Sum(x,n>1)=x[n]+Sum(x,n-1)
```

Тогда получится такая рекурсивная функция поиска суммы элементов вектора:

```
function Sum(var x: Vec; n: integer): real;
begin Sum:=x[1];
      if n>1 then Sum:=x[n]+Sum(x,n-1)
end;
```

При использования открытого массива языка Free Pascal (см. разд 8.3) функцию можно записать в таком виде:

```
function Sum(var x: array of real): real;
  var n: integer;
begin
  n:=high(x); Sum:=0.0; { для пустого массива }
  if n>-1 then { не пустой массив }
    if n=0
      then Sum:=x[0]      { один элемент }
      else Sum:=x[n]+Sum(x[0..n-1])
end;
```

В показанной рекурсивной функции при больших n глубина рекурсии становится неприемлимо большой. Для решения этой проблемы можно модифицировать рекурсивный алгоритм суммирования, используя деление массива пополам:

```
function Sum(var x: array of real): real;
  var k,n: integer;
begin
  k:=high(x); Sum:=0; { для пустого массива }
  if k>-1 then begin { не пустой массив }
    if k=0 then Sum:=x[0] { один элемент }
    else begin
      n:=trunc(k/2);
      Sum:=Sum(x[0..n])+Sum(x[n+1..k])
    end
  end
end;
```

Реализация рекурсивного вызова требует бóльшего времени ЭВМ и объёма памяти автоматического класса. Исходя из этого следует, где только возможно, заменять рекурсию на цикл. Когда рекурсивная подпрограмма вызывает в своём теле сама себя только один рад, то заменить рекурсию на итерацию (цикл) достаточно просто. Существуют, однако, существенно *рекурсивные структуры данных* (например, деревья), обрабатывающие их рекурсивные функции вызывают себя несколько раз. Замена такой рекурсии на цикл очень трудна,[1] в дальнейшем мы изучим такие структуры данных.

Заметим, что в так называемых функциональных языках программирования (самый известный из них старый язык Lisp, а один из последних сложный язык Haskel) циклов может и вообще не быть, вся обработка данных строится на основе рекурсии.

Задания для продвинутых читателей.

1. Объясните, почему *рекурсивную* процедуру нельзя описать как встраиваемую (**inline**).
2. Объясните, как ведёт себя *типизированная константа*, описанная в рекурсивной функции.

8.7. Перегрузка операций, процедур и функций.

Знание некоторых принципов легко возмещает незнание некоторых фактов.
Клод Адриан Гельвеций

[1] Точнее, реализация их в виде цикла требует промоделировать в алгоритме работу механизма рекурсии, что нерационально, так как в языке программирования это уже сделано.

Перегрузка (overload) [1] является одним из видов статического полиморфизма (см. разд. 3.4.2). Мы уже говорили, что, например, операция «плюс» имеет в стандарте Паскаля пять различных смыслов: унарный целый и унарный вещественный, бинарный целый и бинарный вещественный, объединение множеств. Многие языки позволяют программисту добавлять к операциям и свои собственные полиморфные смыслы. Посмотрим, как это делается в языке Free Pascal.

Пусть описан тип комплексных чисел

```
type complex=record re,im: real end;
var  x,y,z: complex;
```

Как мы знаем, операция `x+y` будет вызывать синтаксическую ошибку, но в языке Free Pascal мы можем доопределить операцию «плюс» новым смыслом – сложением комплексных чисел. Для этого надо написать функцию специального вида:

```
operator + (a,b: complex) c: complex;
begin
  c.re:=a.re+b.re; c.im:=a.im+b.im
end;
```

У этой функции нет имени (понятно, что операция + это не имя), поэтому свой результат она возвращает, присваивая его дополнительной (служебной) переменной, указанной в типе результата (у нас это `c: complex`). Теперь в программе можно складывать комплексные числа `z:=x+y`.

Аналогично доопределим операцию «плюс» для сложения массивов:

```
const N=100000;
type Mas=array[1..N] of integer;
var x,y,z: Mas;
operator + (var {можно const} a,b: Mas) c: Mas;
  var i: integer;
begin
  for i:=Low(a) to High(a) do
      c[i]:=a[i]+b[i]
end;
```

Эта функция вырабатывает в качестве результата своей работы массив, адрес этого массива передаётся функции как дополнительный (невидимый) параметр. Вот теперь мы можем складывать массивы `z:=x+y`. Как видим, перегрузка операций позволяет писать более компактные и выразительные программы, так как иначе пришлось бы писать процедуру сложения массивов и вызывать ее как-то так: `PlusMas(x,y,z)`.

Аналогично, можно писать и новые полиморфные подпрограммы. Например, добавим к стандартной функции `abs(x)`, которая в Паскале применяется только для целых и вещественных аргументов, возможность работы и с комплексными числами:

```
function abs(a: complex): real;
begin
  abs:=sqrt(sqr(a.re)+sqr(a.im))
end;
```

Как видим, это самая обычная функция пользователя, суть в том, что её имя не закрывает (экранирует) имя стандартных функций `abs`, а добавляет это имя в область видимости. Теперь для вызова `abs(x)` выше по тексту будут видны три описания функции `abs`:

```
function abs(x: integer): integer;
function abs(x: real): real;
function abs(x: complex): real;
```

---

[1] Ближе по смыслу было бы назвать это переопределением, а ещё лучше доопределением. Английское слово `operator` переводится как «операция» (знак операции), а оператор в языке программирования переводится на английский как `statement`, поэтому в некоторых книгах говорится о «перегрузке операторов», что, конечно, неверно.

и исполнитель Паскаля для конкретного вызова `abs(a)` выберет одну из этих функция, в зависимости от типа фактического параметра.

В теории программирования есть понятие сигнатуры (signature – подпись, иногда говорят прототип – prototype) подпрограммы. Сигнатура включает в себя имя подпрограммы, число и типы всех её параметров (и результата функции). Сигнатуры различаются, если число параметров различно и/или их типы различны, причём соответствующие параметры-значения несовместимы по присваиванию. Перегружаемые подпрограммы должны иметь разные сигнатуры, только тогда они могут располагаться в одной области видимости, например:

```
function ❶ max(x: integer; y: real): integer;
  begin max:=x; if x<y then max:=round(y) end;
function ❷ max(x: real; y: integer): integer;
  begin max:=y; if x>y then max:=round(x) end;
function max(x: char; y: integer): integer;
  begin max:=y; if ord(x)>y then max:=ord(x) end;
function max(x: complex): real;
  begin max:=x.re; if x.im>x.re then max:=x.im end;
function ❸ max: real;
  begin max:=2*Maxint end;
function ❹ max: integer;
  begin max:=Maxint end;
begin { Вызовем эти функции }
  Write(❶ max(1,2.3),❷ max(4.5,6); { Всё хорошо }
{ ОШИБКА, нельзя выбрать между ❶ и ❷,
  ↓↓ их параметры совместимы по присваиванию }
  Write(max(1,1);
  Write(max); { Вызов ❸ первой из описанных 🙄 }
end.
```

Можно перегрузить и стандартную процедуру `write`, хотя здесь есть и свои тонкости. Вот её перегрузка для вывода комплексных значений:

```
procedure write(x: complex);
begin
 ⚠ System.write(x.re:3:2,'+',x.im:3:2,'*i')
end;
```

Тонкость заключается в указании, что в теле процедуры надо вызывать стандартную `write` из модуля `System`. Для доступа к невидимой [1] в данной точке процедуры `write` использовался явный *префикс* области видимости (т.е. имени модуля, в котором это имя описано).

Заметим, что в языке Free Pascal нельзя переопределить *операцию* для стандартных типов, а только для типов, описанных пользователем. Например, для

```
operator + (a,b: string) c: real; { ОШИБКА }
```

будет зафиксирована синтаксическая ошибка, так как Free Pascal «знает» стандартный тип `string` и операцию `+` для этого типа. Можно, однако, переопределить

```
operator + (a: char; b: complex) c: real;
begin c:=ord(a)+b.re+b.im end;
```

Можно перегрузить и все операторы отношений (сравнений) =, <>, >, <, <=, >=, однако выдвигается разумное требование, чтобы результат операций оставался логическим. Отметим, что при перегрузки операции «равно» (=) автоматически перегружается и операция «не равно» (<>) как отри-

---

[1] К сожалению, стандартную процедуру `write` нельзя перегрузить обычным способом (чтобы она осталась видимой), так как у неё необычная сигнатура: *переменное* число параметров, причем на место каждого формального параметра можно передавать фактические параметры разных типов (это хитрая «личная» процедура Паскаля).

цание равенства. К сожалению, *обратное неверно*, так что при желании можно так перегрузить операцию «не равно» так, что будет выполняться

```
(x=y) and (x<>y)  { будет true 😕 }
```

Таким образом, с математической точки зрения будет не выполняться так называемый закон исключённого третьего.

Знак присваивания `:=`, хотя и не является операцией (его нет в таблице старшинства операций!), но тоже можно перегрузить. При этом присваивание (как и в языке C), трактуется как одноместная операция, например:

```
operator := (a: char) b: complex;
begin b.re:=ord(a); b.im:=0.0 end;
```

Перегрузка присваивания добавляет компилятору возможность автоматического преобразования типа, например, после перегрузки:

```
operator := (a: complex) b: real;
begin b:=a.re end;
```

Здесь мы вместе с перегрузкой присваивания добавили новое старшинство в отношения между типами

```
complex ⋘ real
```

Теперь компилятор может автоматически преобразовывать комплексный тип в вещественный, что сделает возможными для комплексной переменной `c`, например, (ошибочные?) выражения вида `sin(c)` или `c+3.14`.

Заметим, что запрет на перегрузку операций для стандартных типов не позволяет делать «неестественные» операции. Например, нельзя переопределить `+` так, чтобы можно было складывать целые и символы, т.е. `1+'A'` будет ОШИБКОЙ всегда, свобода языка C в языке Free Pascal недостижима 😕.

Перегрузка операций и подпрограмм возможна на многих языках программирования, например, Ada, Java, C++ и других. Разновидностью перегрузки подпрограмм является так называемый параметрический (или истинный) полиморфизм (parametric polymorphism). Главная идея заключается в возможности передачи в подпрограмму как параметров не только переменных и подпрограмм, но и типов ⚠, например, как то так:

```
procedere P(var x: <T1>; y: <T2>);
   begin x:=x+y end;
var a: real; b: complex;
    P(a<real>,b<complex>);
```

Заметим, однако, что обычно параметрическими в языках программирования объявляются не подпрограммы, а классы (см. Главу 16).

Вопросы и упражнения

Усердие всё превозмогает!
Козьма Прутков

1. Следует ли использовать подпрограммы, если они вызываются в программе только по одному разу?
2. Почему описание подпрограммы похоже на описание всей программы?
3. Почему хорошо написанные подпрограммы все входные данные должны получать только через свои параметры, и результаты работы возвращать тоже через свои параметры?
4. Что такое соответствие между формальными и фактическими параметрами?
5. Всегда ли в Паскале соблюдается соответствие между числом формальных и фактических параметров?
6. Чем передача параметра по значению отличается от передачи по ссылке?
7. Какие фактические параметры нельзя передать по ссылке?
8. Какие фактические параметры нужно передать по ссылке?
9. Какие фактические параметры нельзя передать по значению?
10. Какие фактические параметры нужно передать по значению?
11. Объясните, почему по ссылке можно передать только переменную, а не любое выражение?

12. Что такое блок?
13. Как определить область видимости конкретного имени?
14. Как определить область существования конкретной переменной?
15. Что такое переменные автоматического класса памяти, как они порождаются и уничтожаются?
16. Почему, если переменная автоматического класса не существует, то к ней нельзя обратиться?
17. Какая переменная называется локальной?
18. Опишите ситуацию, когда при передаче параметра по ссылке программа завершится нормально, а при передаче по значению произойдёт аварийный останов?
19. Чем отличается описание подпрограммы от её объявления?
20. Почему нежелательно, чтобы возвращаемые функцией результаты были слишком большими по размеру?
21. В какой момент подпрограммы завершают свою работу?
22. Что такое инкапсуляция в процедуре?
23. Как передать подпрограмму как параметр?
24. Что такое побочный эффект функции, и почему его следует избегать?
25. Какая процедура называется рекурсивной?
26. Что такое глубина рекурсии, и почему она не может быть слишком большой?
27. Обоснуйте, почему встраиваемая (`inline`) подпрограмма не может быть рекурсивной.
28. Для чего нужна перегрузка операций?
29. Для чего нужна перегрузка подпрограмм?

---

[i] Для продвинутых читателей. Участок освобождённой памяти обычно включается в (односторонний) список свободной памяти (см. разд. 13.3). Следует понять, что это не очередь, а именно список, организованный по возрастанию адресов свободныъ участков в памяти ЭВМ. Это позводит более эффективно бороться с так называемой *фрагментарностью* динамической памяти. Более подробно мы этот вопрос обсуждать не будем.

[ii] Для продвинутых читателей. Итак, в Паскале два способа передачи параметров, по ссылке и по значению. В некоторых языках существует только один способ. Например, в языке С параметры всегда передаются по значению (в С++ это исправлено, добавлена передача по ссылке). А в первых версиях языка Фортран параметры передавались только по ссылке. Для передачи по ссылке в языке С используется передача по значению копии ссылки (так называемого указателя). В первых версиях Фортрана ситуация была хуже, например, чтобы передать в процедуру константу, эту константу приходится записывать в некоторую область памяти, и уже ссылку на эту память передавать как параметр. Таким образом, при передаче константы по ссылке в процедуре можно было изменить значение константы. Могла возникать забавная ситуация, когда после операторов `P(1); write(1)` выводится число 2 🐷.

Вообще говоря, в языках программирования существуют три принципиально разных способов передачи параметров. Кроме передачи по ссылке и по значению, есть ещё передача по имени (pass by name), другие названия: по написанию или по подстановке (pass by substitution). В этом случае строка-написание фактического параметра заменяет в процедуре все имена этого формального параметра (замена строк, как в Нормальных Алгоритмах Маркова). В качестве примера рассмотрим гипотетическое расширение языка Паскаль, в котором есть все три способа передачи параметров. Передачу по значению будем обозначать служебным словом **`value`**, по ссылке привычным **`var`**, а по подстановке словом **`subst`**. Возьмём простую программу:

```
{ гипотетический Паскаль с тремя способами передачи параметров }
var i: integer; A: array[1..2] of integer;
procedure P(value x: integer; var y: integer; subst z: integer);
  begin
    i:=2; A[1]:=2; Write(x:2,y:2,z:2) { Будет вывод  1 2 3 }
  end;
begin i:=1; A[1]:=1; A[2]:=3; P(A[i],A[i],A[i]) end.
```

Проведите тщательную трассировку программы и поймете, как она работает. Передача параметра по подстановке была в одном из первых языков программирования Алголе-60. Этот способ передачи параметров самый гибкий, но его очень трудно реализовать на ЭВМ. В настоящее время он в основном используется в макроязыках, Вы познакомитесь с ним при изучении языка Макроассемблера в курсе по архитектурам ЭВМ, так что это знание Вам ещё понадобится.

[iii] Для продвинутых читателей. Работа всех рассмотренных до этого исполнителей алгоритма заключалась в том, что на каждом шаге работы они производили модификацию *состояния программы* (см. разд. 8.3). Машина Тьюринга меняла символы в клетках ленты и переключала состояния q_i, Нормальные Алгоритмы Маркова меняли входное слово, программа на Паскале изменяла значения переменных и т.д. Можно даже сказать, что на каждом шаге *разрушалось* старое состояние программы, и создавалось новое. Главную лепту в это вносят операторы присваивания, которые явно меняют значение переменных в своих левых частях, однако в состояние программы могут входить и значения так называемых *переменных окружения*, открытых файлов, состояние сетевых ресурсов (облачных хранилищ, сайтов в Интернета и т.д.).

Существуют, однако, и принципмально другие исполнители алгоритма, которые не меняют состояние переменных программы, как это не покажется удивительным на первый взгляд. Основой работы этих исполнителей является использование «чистых» функций (см. нижу сноску [iv]), единственным эффектом этих функций является вычисление своего значения, поймите, что состояние программы (кроме текущей точки выполнения) при этом не меняется! Основанные на этом принципе работы языки так и называются – функциональными алгоритмическими языками (см. сноску [i] к главе 1). В «чистых» функциональных языках нет операторов присваивания, операторов циклов (только рекурсия) и передачи параметров по ссылке. Вся программа является функцией, которая отображает входные данные на результаты работы. Понять эту, как говорят, парадигму программирования непросто, а современные функциональные языки достаточно сложны для изучения. В то же время такие языки предоставляют новые возможности при разработке программного обеспечения. Надеюсь, что Вы вскоре изучите какой-нибуль из таких языков (самым мощным и очень трудным для изучения считается, вероятно, язык Haskell).

[iv] Для продвинутых читателей. В процедурном типе бесконечное (счётное) множество значений, а в «обычных» типах только конечное. Видимо, это была одна из причин, по которой Н. Вирт и не стал вводить такие типы в стандарт Паскаля. Главная же проблема состоит именно в передаче подпрограммы как параметра (а также возможности функции возвратить подпрограмму в качестве своего значения). Как мы уже знаем, каждая подпрограмма является блоком, и, одновременно, выполняется в некотором *объемлющем* блоке, в частности, может использовать (глобальные) переменные этого объемлющего блока. В Паскале (в отличие, скажем, от языка C) одна подпрограмма может быть при описании вложена в другую, следовательно должна выполняться в блоке этой объемлющей подпрограммы.

Однако, если локальную подпрограмму передать как параметр в глобальную подпрограмму и вызвать там, то она дожна выполняться уже в блоке этой глобальной подрограммы, а не в том, в котором она описана (создана) ⚠. Это называется Фунарг проблемой (Funarg problem), Funarg есть сокращение от «функциональный аргумент», т.е. параметр-подпрограмма. Впервые эта проблема была обнаружена в языке Lisp, её весьма сложно решить. Например, вот эта же проблема в языке Free Pascal:

```
{$modeswitch nestedprocvars}
type
  proc=procedure;
{ ❻ proc=procedure is nested; }
var ❶ x,y: integer; xQ: proc;
```

```
procedure R;
  var ❸ x,y: integer;
begin
  x:=5; y:=6;
  ❸ xQ
end;
```

```
procedure P(var z: proc);
  var ❷ x,y: integer;
  procedure Q;
  begin
    Writeln(❷ x+y)
  end;
begin
  x:=3; y:=4;
  Q; {Вывод 3+4=7}
```

```
   z:=❹ @Q
 { z=xQ:=@Q }
end;
```

```
begin
  x:=1; y:=2;
  P(xQ); {xQ=@Q}
  R;
{ Вызов Q в блоке процедуры R, вывод 5+6=11 }
  ❺ xQ; { Вызов Q в блоке основной программы, вывод 1+2=3 }
end.
```

При первом вызове процедуры Q из процедуры P будет, естественно, вывод ❷ x+y=3+4=7. В точке ❹ переменной z присваивается адрес процедуры Q, локальной (и видимой) только внутри процедуры P. Обычно Free Pascal такого не позволяет, но мы разрешили это директивой {$**modeswitch** nestedprocvars} и можно описать процедурный тип как

❻ **type** proc=**procedure is nested**;

Такое описание способа работы с процедурой Q приводят к тому, что её вызов в точке ❸ xQ приказывает считать, что она по-прежнему выполняется внутри блока процедуры P, а не внутри блока процедуры R. Поэтому второй вывод будет опять ❷ x+y=7, а вовсе не ❸ x+y=11. Можно сделать заключение, что при взятии адреса в точке ❸ xQ этот адрес неявно связывается со ссылкой на контекст, в котором должна выполняться подпрограмма.

Много хуже обстоит дело в точке ❺, где тоже производится вызов процедуры Q, но блока процедуры P уже не существует, так как мы вышли из этой процедуры. Т.е. переменные ❷ x,y не существуют 🙁, и вывод x+y=???. Да и неясно, что с точки зрения программиста должна вывести процедура Q: ❷ x+y или ❶ x+y ? Эта проблема в языке Free Pascal не решена. Хорошей задачей является понять, что в стандарте Паскаля такая ситуация невозможна.

Возможны следующие подходы к вопросу о том, в каком контексте должна выполняться вызываемая подпрограмма:

1. Подпрограммы выполняется в контексте точки её вызова, это называется теневое связывание (shallow binding). В точке ❺ нашего примера будет вывод 1+2=3.
2. Подпрограммы выполняется в контексте своего описания, это называется глубокое связывание (deep binding). В точке ❺ нашего примера будет вывод x+y=???. Более подробно о том, что конкретно будет выводиться в этом случае в точке ❺ Вы узнаете из курса по архитектурам ЭВМ, когда будет рассматриваться выполнение подпрограммы в системе так называемых стековых кадров (фреймов).
3. Контекст выполнения передаётся вместе с подпрограммой как (дополнительный) пареметр, это называется специальное связывание (ad hoc binding). В точке ❺ нашего примера будет вывод, зависящий от переданного контекста.

Для полного решения этой проблемы в языках программирования используется особый приём, называемый замыканием (closure), суть которого заключается в том, что используется специальное связывание и локальная подпрограмма хранит (где-то в куче) и свой контекст т.е. последнее содержимое своего объемлющего блока. При вызове такой подпрограммы ей передаётся дополнительный служебный параметр – указатель на контекст того блока, в котором она выполнялась последний раз. Разумеется, это обходится весьма дорого. В языках Free Pascal и C этот механизм не реализован, однако во многих языках (Lisp, C#, JavaScript, PHP и другие) это сделано. На первый взгляд кажется, что в языке C этой проблемы не будет, так как в этом языке нет *вложенных* функций (хотя некоторые *расширения* языка их допускают), однако там есть так называемые безымянные блоки, так что проблема остаётся.

Итак, связывание – это объединение подпрограммы с видимыми из неё именами того блока, в котором находится эта подпрограмма. В качестве иллюстрации этого механизма рассмотрим гипотетическую версию языка Free Pascal, в котором есть такое специальной связывание. Рассмотрим программу:

```
type Func=function (x: integer): integer;
var  F1,F2: Func;
```

```
function P(x: iRef): Func;
{ Будет замыкание функции F с локальной x функции P }
  function F(y: integer): integer;
  begin ❶ F:=x+y end;
begin P:=@F { P возвращает функцию F } end;
```

```
begin
```

```
  F1:=P(1); { Замыкание F при x=1 }
  F2:=P(2); { Замыкание F при x=2 }
  Writeln(F1(3)); { Печать F(x+y=1+3=4 }
  Writeln(F2(4)); { Печать F(x+y=2+4=6 }
end.
```

Здесь в точку ❶ при вызове `P(1)` функция `F` «видит» `x=1`, а при вызове `P(2)` «видит» `x=2`, при этом функция `P` возвращает замыкание функции `F` как результат своей работы. Этот результат запоминается в функциональных переменных `F1` и `F1`, и всюду далее функция `F1` «замкнута» с `x=1`, а функция `F2` «замкнута» с `x=2`. Обратите внимание, что при использовании функций `F1` и `F1` локальные переменные процедуры `P` (и, в частности, параметр `x`) уже не существуют ⚠.

Необходимо понять, что Фунарг проблема имеет корни в самой возможности хранить в ссылочных переменных адреса переменных автоматического класса памяти, например

```
type iRef=↑integer;
var xRef: iRef;
procedure P(var a: iRef);
  var ❶x: integer;
begin x:=1; a:=@x;
{ В стандарте Паскаля так нельзя ! }
end;
begin P(xRef); Write(xRef↑); {???}
{ xRef ссылается на уничтоженную
  локальную переменную ❶x процедуры P }
end;
```

Теперь Вы должны понять, почему Вирт запретил в стандарте Паскаля операцию взятия адреса `@x`.

---

v Для продвинутых читателей. В программировании функция называется детерминированной (название перекликается с основным свойством алгоритма), если для одинаковых значений своих фактических параметров она всегда возвращает одинаковый результат. Ясно, что функция может быть не детерминированной только тогда, когда часть своих входных данных она как-то получает не из фактических параметров, а из «внешнего мира», т.е. читая значения нелокальных переменных, делая ввод данных или получая эти данные путём вызова других подпрограмм.

Как уже говорилась «хорошая» процедура все свои входные данные получает через параметры и все результаты своей работы выдаёт тоже только через параметры. Ясно, что такие требования нужно предъявлять и к «хорошим» функциям. Функция называется чистой, если она детерминированная и не имеет побочного эффекта. Эти два свойства независимы друг от друга (математики говорят *ортогональны*), т.е. функция может быть детерминированной, но иметь побочный эффект, и наоборот.

Большинство языков позволяют писать чистые функции и это является хорошим стилем программирования. Плохая идея, например, написать функцию с заголовком

```
function sincos(x: real; var cos: real):real;
```

которая возвращает значение синуса угла `x` в качестве главного результата, а значение косинуса – через переменную `cos`. По прагматике такую подзадачу надо реализовывать в виде процедуры

```
procedure sincos(x: real; var sin,cos: real);
```

Отметим, что именно такая процедура реализована в модуле `Math` языка Free Pascal.

Объясните, почему формальные параметры процедуры и функции программист может «спокойно» называть именами стандартных функций `sin` и `cos`.