Глава 11. Подпрограммы и функции

Учебник по Perl

Содержание [скрыть]

Подпрограммы в языке Perl играют ту же роль, что и функции в языке C или процедуры и функции в языке Pascal. Они выполняют две основные задачи:

- позволяют разбить одну большую программу на несколько небольших частей, делая ее более ясной для понимания;
- объединяют операторы в одну группу для повторного использования.

В языке Perl не различаются понятия «подпрограммам и «функция», эти слова являются синонимами.

Определение подпрограммы

Подпрограмма может быть определена в любом месте основной программы при помощи описания

```
sub name [(proto)] ({block});
```

Здесь:

- *name* имя подпрограммы;
- (proto) прототип, конструкция, используемая для описания передаваемых подпрограмме параметров;
- *{block}* блок операторов, являющийся определением подпрограммы и выполняющийся при каждом ее вызове.

Форма

```
sub name [(proto)];
```

представляет собой предварительное объявление подпрограммы без ее определения. Пользователь, предпочитающий помещать описания всех подпрограмм в конце основной программы, должен при вызове еще не определенной фуНКЦИИ ИСПОЛЬЗОВаТЬ Специальный СИНТаКСИС &лагае ИЛИ *name* (см.раздел 11.2). Если же некоторое имя предварительно объявить в качестве имени функции, то сразу после объявления к этой функции можно обращаться просто по имени без применения специального синтаксиса.

```
#!/usr/bin/perl sub max {
my $maximum = shift @$_;
my $x;
foreach $x (@_) {
$maximum=$x if ($x > $maximum) ;
}
return $maximum } print "Наибольший аргумент=", max(3,5,17,9), "\n";
```

В данном примере функция max () возвращает наибольший из своих аргументов. Об использовании функции ту() и массива @_ будет рассказано ниже.

Данный способ определения подпрограмм не является единственным. Существуют и другие варианты:

- текст подпрограммы может храниться в отдельном файле и загружаться в основную программу при помощи ключевых слов do, require, use;
- строка, содержащая текст подпрограммы, может быть передана в качестве аргумента функции eval о *(см. раздел 10.3);* в этом случае компиляция кода подпрограммы осуществляется при каждом вызове функции eval ();
- анонимную подпрограмму можно определить при помощи ссылки на нее *(см. раздел 9.2.4. Т).*

Применение функции eval () и ссылки на анонимную подпрограмму были рассмотрены ранее.

Конструкция do *filename* вызывает выполнение Perl-программы, содержащейся в файле *filename.* Если файл *filename* недоступен для чтения, функ-ция do возвращает неопределенное значение и присваивает соответствующее значение специальной переменной $!. Если файл *filename* может быть прочитан, но возникают ошибки при его компиляции или выполнении, то функция do возвращает неопределенное значение и помещает в переменную $@ сообщение с указанием строки, содержащей ошибку. Если компиляция прошла успешно, функция do возвращает значение последнего выражения, вычисленного в файле *filename.*

Замечание

> Специальная переменная $! служит для хранения сообщения о последней системной ошибке. Такая ошибка возникает при обращении „к операционной системе с запросом на предоставление некоторой услуги, как, например, создание файла, чтение или запись в него.

Специальная переменная $@ 'используется для хранения сообщения, генерируемого при последнем обращении к функциям eval () или do *filename,*

```
# файл "l.pl":
#!/usr/bin/perl \ do "2.pi"; \ print "ошибка: $@\n" if $@; do "3.pl"; ' . j print "системная ошибка: $!\n" if $!;
# файл "2.pi":
$x=l;
$y=0;
$z=$x/$y;
print "z= $z\n";
```

Perl-программа "i.pi", используя конструкцию do *filename,* пытается выполнить сценарии, содержащиеся в файлах "2.pi" и "3.pi". Первый из них содержит в третьей строке операцию деления на 0, вызывающую появление ошибки во время выполнения программы, а второй вообще не существует. В результате выполнения файла "i .pi" будут выведены следующие сообщения:

```
ошибка: Illegal division by zero at 2.pi line 3. системная ошибка: No such file or directory
```

Ключевые слова use и require используются для включения в текущую программу подпрограмм из других модулей.

(Директивы компилятора use и require рассмотрены в главе 12.)

Вызов подпрограммы

Мы знаем, что принадлежность к тому или иному типу определяется префиксом имени: $x – скалярная переменная, @x – массив, %x – ассоциативный массив. Префиксом функции является символ "&". К любой подпрограмме можно обратиться, указав ее имя с префиксом &:

```
Sname args; Sname(args); Sname;
```

Здесь args обозначает список аргументов подпрограммы. Если список аргументов отсутствует, вместо него используется специальный массив @_.

Если после имени подпрограммы следуют скобки, префикс & можно опустить:

```
name (args); name();
```

Если до обращения к ней подпрограмма была объявлена или импортирована, то скобки также можно опустить:

```
sub name {. . .}; name args;
name;
```

Если подпрограмма вызывается через ссылку на нее, префикс является обязательным:

```
$subref = sub (...}; Ssubref(args);
ssubref;
```

Подпрограмма может быть использована в выражении как функция, возвращающая значение. По умолчанию значением подпрограммы является последнее вычисленное в ней выражение. Его можно изменить, указав явно в качестве аргумента функцию return () в любой точке подпрограммы. Возвращаемое значение может быть скалярной величиной или массивом.

Локальные переменные в подпрограммах

Областью видимости или *областью действия* переменной мы будем называть часть программы, где данная переменная может быть использована. В языке Perl, как мы знаем, нет обязательного явного описания переменных. Точкой определения переменной является место, где она впервые встречается в программе. Область действия большинства переменных ограничена *пакетом.* Исключение составляют некоторые специальные предопределенные глобальные переменные интерпретатора perl. Пакет – это механизм, позволяющий создать свое пространство имен для некоторого отрезка программы (этот отрезок может включать всю программу). Каждый фрагмент кода Perl-лрограммы относится к соответствующему пакету.

(Пакеты рассматриваются в главе 12, а специальные переменные – в главе 14.)

Таким образом, переменная, впервые встретившаяся- в некоторой подпрограмме, становится доступной во всем пакете, к которому эта подпрограмма принадлежит. Любая переменная в Perl по умолчанию считается глобальной, но эта глобальность ограничена рамками пакета. Иногда бывает необходимо ограничить область действия переменной рамками подпрограммы или блока, в которых она определена. Такие переменные называются локальными. В языке Perl существуют два способа описания локальных переменных: при помощи my() и local().

Функция my()

Функция my() используется для объявления одной или нескольких переменных локальными:

```
my EXPR
```

и ограничивает их область действия:

- подпрограммой;

- заключенным в фигурные скобки блоком операторов;
- выражением, переданным на выполнение функции eval () *(см. раздел 10.3);*
- файлом, в зависимости от того, в каком месте вызвана для объявления переменных сама функция my().

Если выражение EXPR содержит список переменных, то он должен быть заключен в скобки:

```
my ($myvar, @mylist, %myhash);
```

Одновременно с объявлением переменные могут быть инициализированы:

```
my $pi = 3.14159;
my ($pi, $exp) = (3.14159, 2.71828);
```

Переменные, объявленные при помощи функции my(), доступны в своей области действия *только* для подпрограмм, определенных в этой области. Для подпрограмм, определенных за ее пределами, они недоступны. Такие переменные называют *лексическими,* а саму область видимости — *лексической* или *статической областью видимости.*

Функция local()

Функция local () также используется для объявления и инициализации переменных:

```
local EXPR;
local ($myvar, @mylist, %myhash);
local $pi = 3.14159;
local ($pi, $exp) = (3.14159, 2.71828);
```

но, в отличие от функции ту о она создает не локальные переменные, а временные значения для глобальных переменных внутри:

- подпрограммы;
- заключенного в фигурные скобки блока операторов;
- выражения, переданного на выполнение функции eval ();
- файла;

в зависимости от того, в каком месте вызвана для объявления переменных сама функция local (). Если функция local () применяется для описания нескольких переменных, они должны быть заключены в скобки. Если глобальная переменная, объявленная при помощи этой функции, ранее встречалась до объявления и имела некоторое значение, то это значение сохраняется в скрытом стеке и восстанавливается после выхода соответственно из подпрограммы, блока, функции eval {} или файла. Переменная, объявленная при помощи функции local (), или, точнее, ее временное значение, доступна для' *любой* функции, вызванной внутри подпрограммы, блока, функции eval о или файла, в

которых сделано объявление. Такую переменную называют *динамической,* а ее область видимости — *динамической областью видимости.* В • названии отражается тот факт, что область видимости переменной динамически изменяется с каждым вызовом функции, получающей доступ к этой переменной.

Функция ту о является относительно новой, она появилась в версии Perl 5. Для создания действительно локальных переменных рекомендуется использовать именно функцию ту о, а не функцию local о. Впрочем, есть несколько исключений. О них мы расскажем ниже.

В следующем примере показано, чем отличаются переменные, объявленные **при помощи функций** my() И local().

```
sub fl{
local ($x) = "aaaa";
my($y) = "bbbb";
print("fl: x = $x\n");
print("fl: y='$y\n\n");
f 2 (} ;
print("fl: x = $x\n");
print("fl: y = $y\n\n");
} • ' ' ' sub f2{
print("f2: x = $x\n");
print("f2: y=$y\n\n");
$x = "cccc";
$y = "dddd";
print("f2: x = $x\n");
print("f2: y=$y\n\n");
I
```

Результатом выполнения данного примера будет следующий вывод:

```
II л = аааа
f. y = bbbb ,
2: x — аааа c2: y =
f2: x = cccc £2: y = dddd
fl: x = cccc fl: y = bbbb
```

Как видно из приведенного результата, функция f2 () не имеет доступа к переменной $y, объявленной при помощи функции ту о внутри функции fi (), и, напротив, имеет доступ к переменной $x, объявленной внутри fl () **при помощи функций** local ().

Передача параметров

Информация в подпрограмму и обратно передается через параметры (аргументы). Для передачи параметров в подпрограмму используется специальный массив @_. Все параметры запоминаются в элементах массива $_ [o], $_ [1] и т. д. Такой механизм позволяет передавать в подпрограмму произвольное количество параметров.

Массив @_ является локальным для данной подпрограммы, но его элементы — это псевдонимы действительных скалярных параметров. Изменение элемента массива @_ вызывает изменение соответствующего действительного параметра.

В языках программирования различают передачу параметров *по ссылке* и *по значению.* При передаче параметров по значению подпрограмма получает копию переменной. Изменение копии внутри подпрограммы не влияет на ее оригинал. При передаче параметров по ссылке подпрограмма получает доступ к самой переменной и может ее изменять.

Передача параметров через специальный массив @_ фактически является передачей параметров по ссылке. В языке Perl можно реализовать передачу параметров по значению, если внутри подпрограммы при помощи функции ту о объявить локальные переменные и присвоить им значения фактических параметров из массива @_, как это сделано в следующем примере.

```
#!/usr/bin/perl
# Передача в подпрограмму параметров по значению sub f {
my($x, $y) = @_; return (++$x * –$y); }
$val = f ^lib-print "Значение (9+1) * (11-1) равно $val.\n"; $x = 9; $У = 11;
$val = f($x,$y);
print "Значение ($x+1) * ($y-1) равно $val.\n"; print "Значение \$x остается равным $x, а \$y равным $y.\n";
```

Результат выполнения:

```
Значение (9+1) * (11-1) равно 100.
Значение (9+1) * (11-1) равно 100.
Значение $x остается равным 9, а $y равным 11.
```

Передача по ссылке параметров-массивов

Итак, подпрограмма получает и возвращает параметры через специальный массив @_. Если параметр является массивом или хеш-массивом, его элементы также сохраняются в массиве параметров @_. При передаче в подпрограмму нескольких параметров-массивов или хеш-массивов они утрачивают свою целостность. Иными словами, после записи параметров-массивов (хеш-массивов) в массив @_ из него невозможно выделить отдельный параметр-массив (хеш-массив): все параметры в массиве @_ хранятся единой "кучей". Для сохранения при передаче в подпрограмму целостности массива или хеш-массива существуют два основных подхода.

11.4.1.1. Использование типа *typeglob*

Первый подход, более старый, заключается в использовании внутреннего типа данных, называемого typeglob. Принадлежность к типу typeglob обо-

значается префиксом "*". Префикс "*" можно рассматривать как метасимвол, вместо которого может стоять любой из префиксов "$", "@", "%", "&", обозначающих тип данных "скаляр", "массив", "хеш-массив", "функция" соответственно. Интерпретатор преобразует переменную типа typeglob, например, *abc, в *скалярную величину.* Эта величина является ссылкой на гнездо в таблице символов, содержащее элементы, *разных* типов с *одинаковым* именем abc, и представляет любой из этих элементов. Например, запись *abc обозначает всю совокупность, а также любую из следующих переменных: скаляр $abc, массив @abc, хеш %abc, функция sabc.

(Таблицы символов обсуждаются в главе 12.)

Передача в подпрограмму вместо параметра-массива или хеш-массива соответствующей переменной типа typeglob является имитацией передачи параметра-массива (хеш-массива) по ссылке с сохранением его целостности. Рассмотрим следующий пример.

```
sub doublargs {
local(*mylist, *myhash) = @_;
foreach $item (@mylist) { $item *= 2;
}
foreach $key (keys %myhash) { $myhash{$key} *= 2;
} }
@somelist= (1,2,3); /^~~- """"~\ %somehash=("one"=>5, "two"=>15, "three"=>20); print
"начальные значения:\n\@somelist=@somelist\n"; foreach $key (keys %somehash) {
print "\$somehash{$key}=$somehash{$key} ";
}
print "\n";
doublargs(*somelist,*somehash);
print "итоговые значения:\n\@somelist=@somelist\n";
foreach $key (keys %somehash) {
print "\$somehash{$key}=$somehash{$key} "; } print "\n";
```

Подпрограмма doubiargs принимает на вход массив и хеш-массив и изменяет их элементы, умножая на 2. Вместо массива и хеш-массива в подпрограмму передаются соответствующие переменные типа typeglob, которые легко выделить из массива @_, так как фактически они являются скалярами. Обратите внимание на применение функции local. Использовать вместо нее функцию ту здесь нельзя. Переменная типа typeglob не может быть локальной, она представляет несколько одноименных переменных разных типов из таблицы символов. Далее возникает вопрос, каким образом изменение в подпрограмме массива @myiist влияет на изменение фактического параметра gsomeiist. Дело в том,

что операция присваивания вида *x = *y создает синоним *x для гнезда таблицы символов *y, так что осуществление операции над $x, @x, %x эквивалентно осуществлению этой операции над $y, @y, %y. В результате присваивания

```
local(*mylist, *myhash) = @_;
```

создается псевдоним *myiist для *someiist, поэтому все изменения элементов массива @myiist внутри подпрограммы эквивалентны изменениям элементов массива @someiist. Все сказанное справедливо и для хеш-массивов %myhash и %somehash. Результат подтверждает корректность передачи массива и хеш-массива по ссылке:

начальные значения:

```
@somelist=l 2 3
$somehash{one}=5 $somehash{three}=20 $somehashftwo)=15
```

итоговые значения:

```
@somelist=2 4 6
$somehash{one}=10 $somehash(three}=40 $somehash{two}=30
```

11.4.1.2. Использование ссылок

Второй, более новый способ передачи массивов в подпрограмму заключается в том, чтобы вместо собственно массивов или хеш-массивов передавать ссылки на них. Ссылка является скалярной величиной и ее легко выделить в массиве параметров @_. Внутри подпрограммы остается только применить к ссылке операцию разыменования для того, чтобы получить доступ к фактическому параметру. Поскольку ссылки появились только в версии Perl 5, то этот способ является относительно новым. При помощи ссылок предыдущий пример можно записать в следующем виде,

```
sub doublparms {
ray ($listref, $hashref) = @_;
foreach $item (@$listref} { $item *= 2;
} .
foreach $key (keys %$hashref) { $$hashref{$key} *= 2;
} }
@somelist=(1,2,3) ;
%somehash=("one"=>5, "two"=>15, "three"=>20); print "начальные
значения:\@somelist=@somelist\n"; foreach $key (keys %somehash) {.
print "\$somehash{$key}=$somehash{$key} "; }
print "\n";
doublparms(\@somelist,\%somehash); print "итоговые значения:\n\@somelist=@somelist\n"; foreach
```

```
$key (keys %somehash) {
print "\$somehash{$key}=$somehash($key} "; } print "\n";
```

Здесь для описания локальных переменных использована функция ту. Как мы выяснили ранее в этой главе, применение функции ту в подобном случае реализует передачу параметров по значению. Другими словами, их изменение внутри подпрограммы не влияет на фактические параметры. Каким же образом в данном случае осуществляется передача массива и хеш-массива по ссылке? Дело в том, что по значению передаются только *ссылки,* указывающие на фактические параметры: массив @someiist и хеш-массив %somehash. Используя операции разыменования внутри подпрограммы, мы получаем доступ непосредственно к массиву @someiist и хеш-массиву %somehash, и изменяем их элементы. В результате выполнения данного сценария будет выведено:

начальные значения:

```
@somelist=l 2 3
$somehash{one}=5 $somehash{three}=20 $somehash{two}=15
```

итоговые значения:

```
@somelist=2 4 6
$somehash{one}=10 $somehash{three)=40 $somehash{two}=30
```

В каких случаях функцию *local* нельзя заменить функцией *my*

В следующих случаях функция local () является незаменимой. П Присваивание временного значения глобальной переменной. В первую очередь это относится к некоторым предопределенным глобальным переменным, таким как $ARGV, $_ и т. д. Рассмотрим пример.

```
#!/usr/bin/perl $/ = under"; @ARGV = ("a"); $_ = <>;
print "Первое значение области ввода \$_= ", split,"\n"; {
local 0ARGV = ("aa"); local $_ = <>;
print "Второе значение области ввода \$_= ", split,"\n"; }
{
local 8ARGV = ("aaa"); local $_ = <>;
@fields = split;
print "Третье значение области ввода \$_= ", split, "\n";
}
print "Восстановленное значение области ввода \$_= ", split,"\n";
```

Пусть имеются три файла

```
"a": "aa": "aaa":
1111 1111 1111 2222 2222 2222 3333 3333 3333
```

```
aaaa bbbb cccc dddd eeee ffff gggg hhhh iiii
```

В приведенной программе используются специальные глобальные переменные $/, $_ И @ARGV.

Специальная переменная $/ содержит значение разделителя входных записей, которым по умолчанию является символ новой строки. Присваивание этой переменной неопределенного значения позволяет при помощи одной операции ввода <> считать весь файл, а не только первую строку.

Специальная переменная $_ используется по умолчанию для сохранения вводимых данных, если в операции ввода соответствующий параметр не указан. Она также используется по умолчанию в качестве аргумента функции split о, если в ней не задана строка, подлежащая разбиению на отдельные строки.

Массив @ARGV содержит аргументы командной строки самой программы. Если при вызове программы ей будет передано имя файла, то оно будет сохранено в массиве @ARGV. Операция ввода о применяется к файлам, переданным в программу в качестве аргументов командной строки, т. е. к файлам, имена которых хранятся в массиве @ARGV. В нашем примере программа вызывается без аргументов, поэтому имя входного файла "а" задается внутри программы прямой записью в массив @ARGV. Первая операция ввода о, следовательно, осуществляется из файла "а". Далее следуют два блока операторов, заключенных в фигурные скобки. В каждом из них при помощи функции local () создаются временные значения для глобальных переменных @ARGV и @_. В первом блоке данные считываются из файла "аа" и сохраняются в качестве временного значения глобальной переменной $_, во втором — из файла "ааа" и также сохраняются в качестве следующего временного значения переменной $_. По выходе из второго блока глобальная переменная $_ восстанавливает свое первоначальное значение. В результате выполнения данной программы будет напечатано:

```
Первое значение области ввода $_= 1111111111llllaaaabbbbcccc
Второе значение области ввода $_= 222222222222ddddeeeeffff
Третье значение области ввода $_= 3333333333;33gggghhhhiiii
Восстановленное значение области ввода $_= lilllllllllllaaaabbbbcccc
```

Создание локального дескриптора файла, каталога или локального псевдонима для функции.

В следующем примере функция local о используется для создания локального дескриптора файла внутри блока операторов.

```
#!/usr/bin/perl
open(FILEHANDLE,">b");
print FILEHANDLE "Новая строка в файл 'b'\n";
{
local *FILEHANDLE;
open(FILEHANDLE,">bb");
```

```
print FILEHANDLE "Новая строка в файл 'bb'\n";
close FILEHANDLE; }
{
local *FILEHANDLE;
open(FILEHANDLE,">bbb");
print FILEHANDLE "Новая строка в файл 'bbb'\n";
close FILEHANDLE; > J
print FILEHANDLE "Еще одна строка в файл 'b'\n"; close FILEHANDLE;
```

В результате выполнения данного сценария в текущем каталоге будут созданы файлы:

```
"b":
Новая строка в файл 'b'
Еще одна строка в файл 'b'
"bb":
Новая строка в файл 'bb'
"bbb":
Новая строка в файл 'bbb'
```

Заметьте, что во время выполнения операций с файлами "bb" и "bbb" файл "b" остается открытым.

Аналогичным образом может быть определено локальное имя для функции.

```
#!/usr/bin/perl
# функция NumberOfArgs() возвращает число своих параметров sub NumberOfArgs {
return $#_ + 1;
) ' . . ' print "NumberOfArgs: число параметров=", NumberOfArgs(1,2,3,4),"\n"; {
local *Numbers = *NumberOf Args;
print "Numbers: число параметров=", Numbers (1, 2, 3} , "\n"; } {
local *N = \SNumberOfArgs;
print "N: число параметров=", N(1,2), "\n"; }
```

Результат выполнения:

```
NumberOfArgs: число параметров=4 Numbers: число параметров=3 N: число параметров=2
```

Временное изменение элемента массива или хеш-массива.

В следующем примере внутри блока операторов временно изменяется значение одного элемента глобального хеш-массива %ENV, содержащего значение переменной $PATH, входящей в состав среды интерпретатора UNIX shell.

```
tt!/usr/bin/perl
print "значение переменной среды \$PATH:\n$ENV{PATH}\n"; {
local $ENV{PATH} = "/home/mike/bin"; I -print "временное значение переменной среды \$PATH:
$ENV{PATH}\n";
}
print "прежнее значение переменной среды \$PATH:\n$ENV{PATH}\n";
```

Результат будет выведен в следующем виде: значение переменной среды $PATH:

```
/sbin: /usr/sbin: /usr/bin: /bin: /usr/XHR6/bin: /usr/local/bin: /opt/bin
временное значение переменной среды $PATH: /home/mike/bin
прежнее значение переменной среды $PATH:
/sbin: /usr/sbin: /usr/bin: /bin: /usr/XHR6/bin: /usr/local/bin: /opt/bin
```

Прототипы

Встроенные функции Perl имеют определенный синтаксис: имя, число и тип параметров. Прототипы позволяют накладывать ограничения на синтаксис функции, объявляемой пользователем. Прототип представляет собой запись, которая состоит из заключенного в скобки списка символов, определяющих количество и тип параметров подпрограммы. Например, объявление

```
sub func ($$) {
1
```

определяет функцию func о с двумя скалярными аргументами. Символы для обозначения типа аргумента приведены в табл. 11.1.

Таблица 11.1. *Символы, используемые в прототипах для задания типа аргумента*

Символ	Тип данных
$	Скаляр
@	Массив
%	Ассоциативный массив
&	Анонимная подпрограмма
*	Тип typeglob

Запись вида \char, где char – один из символов табл. 11.1, обозначает что при вызове подпрограммы имя фактического параметра должно обязательно начинаться с символа char. В этом случае в подпрограмму через массив параметров @_ передается *ссылка* на фактический параметр, указанный при ее вызове. Обязательные параметры в прототипе отделяются от необязательных точкой с запятой.

В табл. 11.2 в качестве примера приведены объявления пользовательских функции *nybud,itin(),* синтаксис которых соответствует синтаксису встроенных функций *buil tin ().*

Таблица 11.2. *Примеры прототипов*

Объявление	Обращение к функции
sub mylink ($$)	mylink $old, $new
sub myvec ($$$)	myvec $var, $offset, 1
sub myindex ($$;$)	myindex Sgetstring, "substr"
sub mysyswrite ($$$;$)	mysyswrite $buf, 0, length ($buf) - $off, vOf f
sub myreverse (@)	myreverse $a, $b, $c
sub my join ($@j	myjoin ":", $a, $b, $c
sub mypop (\@)	mypop garray
sub mysplice (\@$$@)	mysplice Sarray, @array, 0, @pushme
sub mykeys (\%)	mykeys %{$hashref}
sub myopen (*;$)	myopen HANDLE, $name
sub mypipe (**)	mypipe READER, WRITER
sub mygrep (s@)	mygrep { /pattern/ } $a, $b, $c
sub myrand ($)	myrand 42

```
sub mytime ()                   mytime
```

Следует иметь в виду, что проверка синтаксиса, задаваемого при помощи прототипа, не осуществляется, если подпрограмма вызвана с использованием префикса &: ssubname.

Рекурсивные подпрограммы

Язык Perl допускает, чтобы подпрограмма вызывала саму себя. Такая подпрограмма называется *рекурсивной.* При написании рекурсивных подпрограмм следует иметь в виду, что все переменные, значения которых изменяются внутри подпрограммы, должны быть локальными, т. е. объявленными при помощи функций ту (} или local (). В этом случае при каждом вызове подпрограммы создается Новая копия переменной. Это позволяет избежать неопределенности и замещения текущего значения переменной ее значением из следующего вызова подпрограммы.

Рекурсивные подпрограммы следует применять осторожно. Многие алгоритмы, являющиеся по сути итеративными, можно реализовать при помощи рекурсивной подпрограммы. Однако такая подпрограмма окажется неэффективной по времени выполнения и потребляемым ресурсам памяти. Вместе с тем, существуют задачи, решить которые можно только при помощи рекурсивных алгоритмов. В этом случае применение рекурсивных подпрограмм является не только вполне оправданным, но и необходимым. Одной из таких задач, которую операционная система решает постоянно, является рекурсивный просмотр дерева каталогов. Рассмотрим пример рекурсивной Perl-подпрограммы tree (), которая делает то же самое: просматривает дерево каталогов, начиная с каталога, заданного параметром подпрограммы, и выводит список файлов, содержащихся в каждом подкаталоге.

```
sub tree {
local (*ROOT);
my ($root)=$_[01;
opendi.r ROOT, $root;
my (@filelist) = readdir ROOT;
closedir ROOT;
for $x (gfilelist) {
if ($x ne "." and $x ne ".."){ $x=$root."/".$x; print " $x\n" if (-f $x); if (-d $x) {
print "$x:\n";
tree($x); } } } }
```

Здесь использованы встроенные подпрограммы Perl opendiro, ciosediro, readdiro, применяемые соответственно для открытия каталога, его закрытия и чтения содержимого. Подпрограмма tree о рекурсивно просматривает каталог, переданный ей в качестве параметра, и выводит имена вложенных подкаталогов и содержащихся в них файлов в следующем виде:

```
/home/httpd/cgi-bin: /home/httpd/html:
/home/httpd/html/index.html /home/httpd/html/manual:
/home/httpd/html/manual/LICENSE
/home/httpd/html/manual/bind. html
/home/httpd/html/manual/cgi_path.html
```

Вопросы для самоконтроля

1. Какие формы обращения к подпрограмме вы знаете?
2. Что такое область видимости переменной?
3. Как ограничить область видимости переменной?
4. Чем отличаются переменные, объявленные при помощи функции ту о от переменных, объявленных при помощи функции local () ?
5. Каким образом данные передаются в подпрограмму и из подпрограммы?
6. Что такое передача параметров по ссылке и по значению?
7. Какой тип данных называется typegiob?
8. Как осуществить передачу по ссылке параметра-массива?
9. В каких случаях функция local о не может быть заменена функцией my ()?
10. Что такое прототип?
11. Какие значения будут иметь переменные $x, eiisti, @iist2 после выполнения программы

```
#!/usr/bin/perl
$x = 0;
Slistl = (1, 2, 3);
@list2 = funcO ; sub func {
local ($x);
$x = 1;
@listl = (4, 5, 6); ___
} "X
\
```

Упражнения

1. Напишите подпрограмму, которая выводит пронумерованный список своих аргументов.

2. Напишите подпрограмму, которая выводит пронумерованный список своих аргументов в обратном порядке.

3. Напишите подпрограмму, которая подсчитывает число символов из стандартного ввода и выводит результат.

4. Напишите подпрограмму, которая выводит свои параметры-массивы в обратном порядке по элементам.

5. Напишите подпрограмму, которая для двух своих параметров-массивов осуществляет взаимный обмен элементов с одинаковыми индексами.

6. Одной из известных задач, для решения которых применяется рекурсия, является задача о Ханойских башнях.

7. На плоскости установлены три стержня: a, b, c (рис. 11.1).

На стержень a нанизаны n дисков, расположенных по возрастанию диаметра. Необходимо переместить диски со стержня a на стержень c, используя стержень b и соблюдая следующие ограничения: можно перемещать только один диск одновременно, диск большего диаметра никогда не может находиться на диске меньшего диаметра.

Напишите подпрограмму, которая описывает последовательность переноса дисков в ходе решения задачи, выводя сообщения вида:

Перенос диска со стержня a на стержень c.

Рис 11.1. Задача о Ханойских башнях