Лабораторная работа № 5

Сценарии языка Perl

***Цель работы*:** изучение языка **Perl** для построения сценариев, используемых в командном интерпретаторе **BASH** ОС Linux и в сценариях гипертекстовых страниц.

Продолжительность работы - 4 ч.

Основы создания Perl-программы

Practical Extraction and Report Language, сокращенно называемый **Perl**, представляет собой интерпретируемый язык, предназначенный для написания сценариев. Он реализован в виде ядра, к которому удобно добавлять новые функции. В настоящее время с помощью **Perl** можно проверять сетевые соединения, контролировать взаимодействие между процессами, управлять базами данных, выполнять команды системного администрирования, использовать для создания сценариев на **web**-узлах.

Любой оператор языка **Perl** может быть вызван из командной строки, например, оператор print с его данными:

$ print "Hello word":

$ perl -e 'print "Hello word.";'

В этом случае используется команда **perl** с опцией **-e**, текст оператора заключается в одинарные кавычки.

Более длинный сценарий **Perl** можно также записывать в одинарных кавычках, но удобнее оформлять отдельным файлом. Файлы, содержащие команды **Perl**, должны иметь расширение **.pl**. Простейшая программа, написанная на языке **Perl** и запомненная с расширением **.pl**, может выглядеть таким образом:

```
#!/usr/local/bin/perl
# программа, выводящая на печать текст
print 'Hello world.';          # Печать текста
```

Каждая **Perl**-программа начинается с первой строки, прописывающей путь к команде **perl**, но может быть другой путь в конкретной системе. Первая строка начинается со знаков **#!**:

#!/usr/local/bin/perl

Прежде чем начать писать сценарии на языке **Perl**, посмотрите, в каком каталоге Вашей файловой системы он находится. Возможно, это **/usr/bin/perl**.

Далее следует стандартный комментарий, используемый в любых shell-сценариях и вставляемый в программу через символ #. Программа не реагирует на строки с символом # за исключением первой строки. Если комментарий располагается на нескольких строках, то в начале каждой строки должен быть поставлен символ **#.** Каждый оператор **Perl** должен заканчиваться точкой с запятой.

Функция **print** выводит некоторую информацию. В примере, приведенном выше, она печатает последовательность символов **Hello world**. И конечно, строка закачивается знаком точка с запятой.

Запуск программы

Чтобы подготовить текст программы, можно использовать любой текстовый редактор. Удобным является редактор **Emacs**. После того как текст программы подготовлен и запомнен в файл (с расширением **.pl**), задайть этому файлу право на выполнение, используя команду установки полномочий, аналогично тому, как это делалось для **shell**-сценариев:

$ chmod u+x *progname*

Далее запустите программу на выполнение в среде **Perl**:

$ perl progname

$ ./progname

$ progname

Весь сценарий **Perl** проверяется перед выполнением программы, сведения об ошибках выдаются на экран с указанием соответствующих номеров строк. Но иногда сообщения недостаточно четки или вообще отсутствуют. Поэтому лучше выполнять программу **Perl** с опцией указания замечаний:

$ perl -w progname

На экране будут отображаться замечания и другие полезные сообщения до выполнения программы.

Язык **Perl** синтаксически больше похож на язык **C++**, чем на **shell**-сценарии. В **Perl** в конце строки, как и в **C++**, используется знак точки с запятой, также схоже большинство управляющих символов. Например, одинаково обозначаются символ перехода на новую строку **\n** и символ табуляции **\t**. Но, в отличие от **C++**, в языке-интерпретаторе **Perl** в управляющих структурах не допускается применение одинарных операторов. Набор операторов должен быть заключен в фигурные скобки **{ }**.

Чтобы запустить программу с отладчиком, следует использовать команду **perl** с опцией **-d**:

$ perl -d progname

После выполнения первой компиляции, вносятся необходимые исправления, и выполняется следующая компиляция. Так делается до тех пор пока все ошибки не будут исправлены. Если ошибок нет в очередной, может быть первой, версии компиляции, то после короткой паузы на компиляцию программа выполняется довольно быстро.

Скалярные переменные

Скалярные переменные - основные переменные в **Perl** (**scalar variable**). Они поддерживают оба типа переменных - строковые и числовые. Признаком переменной в программе является знак **$** перед ее именем. Тип переменной зависит от ее использования. Переменные не объявляются. Если переменные используются в арифметических выражениях или им присваиваются числовые значения, то среда **Perl** считает эти переменные числовыми. Все остальные переменные считаются строковыми.

Пример 1. Установить значение переменной **$priority** равным 9:

```
$priority = 9;
```

Пример 2. Perl поддерживает числа как строковые переменные, они должны быть помещены в одинарные кавычки:

```
$priority = '9';
```

Пример 3. Строковые переменные могут быть записаны как последовательность символов, взятых в одинарные кавычки, например:

```
$priority = 'high';
```

или:

```
$default = '0009';
```

Perl также использует специальные переменные среды, которые могут состоять из знака подчеркивания, букв, чисел, но не начинающихся с чисел. Также в **Perl** различаются прописные и строчные символы в переменных. Так, переменная **$a** и переменная **$A** являются различными.

Присваивания и операции

Perl использует все обычные арифметические операторы языка **C**:

```
$a = 1 + 2;        # Добавить 1 к 2 и сохранить в $a
$a = 3 - 4;        # Вычесть 4 из 3 и сохранить в $a
$a = 5 * 6;        # Умножение 5 на 6
$a = 7 / 8;        # Деление 7 на 8 дает 0.875
$a = 9 ** 10;      # Возведение 9 в десятую степень
$a = 5 % 2;        # Остаток от деления 5-ти на 2
++$a;              # Инкремент $a и помещение результата в $a
$a++;              # Помещение значения в переменную $a и  увеличение $a на 1
--$a;              # Уменьшение $a и помещение результата в $a
$a--;              # Помещение в $a и уменьшение $a на 1
```

Для строковых переменных **Perl** использует следующие операции:

```
$a = $b . $c;      # Конкатенация $b и
$a = $b x $c;      # Повторение строки $b указанное в $c число раз (строка может состоять из одного символа)
```

При присвоении **Perl** использует:

```
$a = $b;           # присвоение $b переменной $a
$a += $b;          # Добавляет $b к $a
$a -= $b;          # Вычитает $b из $a
$a .= $b;          # Присоединяет $b к $a
```

Заметим, что когда **Perl** присваивает величину **$b** переменной **$a: $a = $b**, то он делает копию **$b** и после этого выполняет присвоение в **$a**. Если через некоторое время изменится значение переменной **$b**, то оно не станет значением переменной **$a**.

Используя команду:

```
$ man perlop
```

Вы можете получить подробную информацию о команде **Perl**.

Пример 1. Печать **apples** и **pears** с использованием конкатенации:

```
$a = 'apples';
$b = 'pears';
$ print $a 'и' $b'
```

печатает **$a' и '$b**;

Пример 2. Если записать команду:

```
$ print '$a и $b';
```

то печатается посимвольно текст **$a и $b**.

Пример 3. Если использовать двойные кавычки вместо одинарных, то напечатается как в первом примере:

```
$ print "$a и $b";
```

Дело в том, что двойные кавычки дают возможность интерполировать информацию в них, включая переменные.

Условия

Perl поддерживает управляющие структуры - **if/then/else** состояния и операции сравнения. Пример условной управляющей структуры **if/else**:

```
if ($a)
{
    print "The string is not empty\n";
}
```

```
else
{
    print "The string is empty\n";
}
```
Пустая строковая переменная рассматривается как ошибка.

Пример условной управляющей структуры **if/elsif /else**:

```
if (!$a)                    # The ! is the not operator
{
    print "The string is empty\n";
}
elsif (length($a) == 1)     # If above fails, try this
{
    print "The string has one character\n";
}
elsif (length($a) == 2)     # If that fails, try this
{
    print "The string has two characters\n";
}
else                        # Now, everything has failed
{
    print "The string has lots of characters\n";
}
```

Переменные массивов

В **Perl** используется еще один вид переменных - переменные массивов (**array variable**), *представляющие список скалярных переменных*. Переменной, обозначающей массив, присваивается список значений. Массивы помечаются символом **@** перед именем массива:

```
food  = ("apples", "pears", "eels");
@music = ("whistle", "flute");
```

Переменная **@food**, переменная **@music** состоит из 2-х элементов. К элементам массива можно обращаться по отдельности, нумерация начинается с 0.

К элементам массива можно обращаться по индексу, заключенному в квадратные скобки, например, обращение ко второму элементу массива **@food:**

```
$food[2]
```

Присвоения в массиве

Сформируем новый массив двумя способами, используя предыдущий **@music = ("whistle", "flute");** и новые элементы. Следующие два выражения являются эквивалентными:

```
@moremusic = ("organ", @music, "harp");

@moremusic = ("organ", "whistle", "flute", "harp");
```

Для обработки массивов используются различные функции и свойства массивов.

1. Функция **push** предназначена для добавления новых элементов в конец массива. Простейший путь добавить элемент в массив следующий:

```
push(массив, список значений);
```

Например, **eggs** добавляется в конец массива **@food**:

```
push(@food, "eggs");
```

Чтобы добавить два или более элементов в массив используется один из следующих способов:

```
push(@food, "eggs", "lard");

push(@food, ("eggs", "lard"));

push(@food, @morefood);
```

Функция **push** возвращает длину нового списка.

2. Функция **pop** используется для удаления последнего элемента массива:

```
pop(массив);
```

Например, функция **pop** из начального списка возвращает **eels** и теперь **@food** имеет два элемента массива **@food**:

```
push(@food, "eels");
$grub = pop(@food);          # Теперь $grub = "eels"
```

3. Функция **shift** добавляет элемент в начало массива:

```
shift (массив, список значений);
```

4. Функция **unshift** удаляет элемент из начала массива:

```
unshift (массив);
```

5. Сортировка элементов массива в прямом и обратном порядке выполняется функциями соответственно **sort**(массив) **reverse**(массив).

6. Запись длины массива в скалярную переменную.

Строка **$f = @food;** выдает длину **@food**, но следует заметить, что **$f ="@food";** возвращает список строковых данных с пробелами между каждыми элементами. Пробелы могут быть удалены изменением специальной строковой переменной **$**. Это одна из многочисленных специальных переменных **Perl**. Элементы массивов можно присваивать скалярным переменным:

```
($a, $b) = ($c, $d);          # То же самое, что и $a=$c; $b=$d;
($a, $b) = @food;
# $a и $b являются первым и вторым элементами @food.
($a, @somefood) = @food;
# $a первый элемент массива @food,
# @somefood список остальных элементов массива.
# (@somefood, $a) = @food;
# @somefood это @food и $a неопределен.
```

7. Определение текущего числа элементов массива с использованием специальной переменной, обозначаемой таким образом: **$#имя_массива**, соответствующей индексу последнего элемента списка. Элементы массива имеют разный тип: строковый, числовой, строковый.

Чтобы определить текущее число элементов массива **@food** используется следующее выражение **$#food**, чтобы вывести на экран количество элементов массива используется команда:

```
$ print "$#food"
print @food;                  # Просмотр самого массива
print "@food";          # текущее количество элементов массива
$ @food = ("pure", 12, 3.1415, " fresh bunchings") | print @food;
```

Поддержка файлов

Perl поддерживает работу с файлами. Здесь рассмотрены основные операторы для работы с файлами, приведены примеры.

Пример простой программы в **Perl**, использующей файлы. Эта программа используется в ОС Linux и ОС UNIX как команда **cat** при чтении из входного потока и выводе на стандартный вывод (экран).

```
#!/usr/local/bin/perl
#  Программа открывает файл пароля, читает из него,
#  печатает его и закрывает его снова.
$file = '/etc/passwd';
# имя файла, содержащееся в переменной
open(INFO, $file);
# Открытие файла $file с дескриптором INFO
@lines = <INFO>;
# чтение файла из дескриптора в массив
close(INFO);                  # Закрытие файла
print @lines;                 # Печать массива
```

Функция **open** открывает файл для чтения, первый параметр называется **дескриптором** файла (**filehandle**), он позволяет **Perl** ссылаться на файл в будущем. Второй параметр - переменная, в которой находится имя файла. Имя файла может быть указано явно, тогда используются двойные кавычки. Например, **open(INFO, "file1");** следовательно, **$file = "file1".** Чтобы прочитать информацию из открытого файла, нужно указать его дескриптор между символами **< >**.

Функция **close** используется в **Perl** для завершения работы с файлом. Функция **open** использует следующие установки прав доступа к файлам:

```
open(INFO, $file);
# Открытие файла для чтения
open(INFO, ">$file");
# Открытие файла для записи
open(INFO, ">>$file");
# Открытие файла для добавления
#данных в конец файла
open(INFO, "<$file");
# Открытие файла для чтения
```

Если файл уже открыт для записи, то строковую информацию можно записать в него, используя дескриптор файла **INFO**, как показано в следующей записи: **print INFO "This line goes to the file.\n";**

Если используется стандартный ввод, который обычно производится с клавиатуры, и стандартный вывод (обычно экран), то записи осуществляются следующим образом соответственно:

```
open(INFO, '-');              # открыть стандартный ввод
open(INFO, '>-');             # открыть стандартный вывод
```

В вышеописанной программе информация читается из файла INFO.

В описании **@lines = <INFO>;** файл представлен дескриптором файла, заданным в угловых скобках, в массиве **@lines**. Выражение **<INFO>** читает переменную в файл за один шаг, т.к. чтение осуществляется в контексте

переменной массива. Если переменную **@lines** заменить скалярной переменной **$lines**, тогда только одна следующая строка будет прочитана в файл. В обоих случаях каждая строка полностью сохраняется с признаком конца строки в конце.

Циклические структуры

Язык **Perl** использует следующие циклические структуры: **foreach**, **for**, **while**, **do-while** и **until**.

foreach

Структура **foreach** используется для обработки списков и массивов, последовательно просматриваются аргументы **foreach,** следующие за этой функцией после переменной и выделенные скобками:

```
foreach $morsel (@food)      # обращается к каждому элементу
                             # и записывает в переменную $morsel
{
    print "$morsel\n";       # Печать элементов массива
    print "OK\n";            # OK
}
```

Действия, выполняемые в фигурных скобках, повторяются каждый цикл. Переменной **$morsel** последовательно присваиваются значения элементов массива **@food** до тех пор, пока есть элементы.

Операторы сравнения

```
$a == $b          # $a равно $b (числовое сравнение)?
$a != $b          # $a не равно $b(числовое сравнение)?
$a eq $b          # $a равно $b (строковое сравнение)?
$a ne $b          # $a не равно $b (строковое сравнение)?
```

Можно использовать логическое **and**, **or** и **not**:

```
($a && $b)              # $a и $b верно?
($a || $b)              # Хотя бы одно из $a и $b верно?
!($a)                   # $a неверно?
```

Цикл for

Структура цикла **for** в **Perl** похожа на структуру **for** в **C**. Ее вид следующий:

```
for (initialise; test; inc)
{
     first_action;
     second_action;
     etc
}
```

В первой строке сначала выполняется инициализация переменных (**initialise**), выполняется сравнение (**test**), далее увеличение значения переменной на 1 (**inc**).

Пример печани чисел от 0 до 9.

```
for ($i = 0; $i < 10; ++$i)          # Start with $i = 1
                                     # Выполнять пока $i < 10
                                     # увеличение $i на 1 до повторения
{
    print "$i\n";
}
```

Операторы циклов while и until

Ниже представлена программа, иллюстрирующая чтение входной информации с клавиатуры (пароль) и продолжающейся до тех пор, пока не будет введен правильный пароль:

```
#!/usr/local/bin/perl
print "Password? ";                  # Пароль?
$a = <STDIN>;                        # Ввод пароля
chop $a;
                    # удалить последний символ переменной
while ($a ne "fred")          # Пока ввод неправильный...
{
   print "sorry. Again? ";           # Спросить пароль снова
   $a = <STDIN>;                     # Ввести пароль снова
   chop $a;                  # Chop off newline again
}
```

Блок в фигурных скобках выполняется до тех пор пока не будет введен правильный пароль. Из стандартного ввода можно читать до открытия файла. Когда пароль введен в переменную **$a**, в ее конец добавляется символ начала новой строки. Функция **chop** удаляет последний символ переменной (имеет строковый тип).

```
#!/usr/local/bin/perl
@week = (“monday”,”tuesday”,”wednesday”,
                    ”thursday”,”freday”,”saturday”,”sunday”);
print(“Какой по счету сегодня день недели?”);
```

```
$day = <STDIN>;
chop $day;
print (“Today is a ”, @week[$day]);
```

Помимо операторов цикла **while** или **until** можно использовать другую полезную технику на основе оператора **do** с проверкой условия в конце цикла.

```
#!/usr/local/bin/perl
do
{
    "Password? ";          # Спрашивается пароль
    $a = <STDIN>;          # Вводится пароль
    chop $a;               # Убирается символ newline
}
while ($a ne "fred")
# Выполнять до тех пор пока неверный пароль
```

Контрольное задание

1. Перепишите программу **Hello world** таким образом, чтобы в переменной **a** был записан печатаемый текст, а в переменной **b** - управляющий символ перехода на новую строку. Используйте двойные кавычки, но не используйте оператор конкатенации.

2. Выполнить любое задание по условным операторам, использующим **if/elsif /else,** из лабораторной работы 4, используя язык **Perl**.

3. Проверьте каждый из изученных способов печати на созданном Вами массиве.

4. Создайте небольшой массив и отсортируйте его элементы в прямом и обратном порядке.

5. Структура **until** в **Perl** похожа на структуру **until** в **C**. Перепишите программу, использующую оператор **while**, применяя **until**. Выполнение блока повторять до тех пор, пока выражение не станет верно, в **while** использовалась проверка на неравенство.

6. Посчитать значение факториала числа 8.

7. Модифицируйте программу раздела о поддержке файлов так, чтобы она печатала в конце каждой строки **символ@.**

8. Составьте свои программы на основе следующих примеров-фрагментов сценариев(**sub** - подпрограмма):

Пример 1. Определение и вызов процедуры.

```
sub marine {
    $n += 1; #Глобальная переменная $n
    print “Hello, sailor number $n!\n”;
}
&marine;   # содержит сообщение Hello, sailor number 1!
&marine;   # содержит сообщение Hello, sailor number 2!
&marine;   # содержит сообщение Hello, sailor number 3!
&marine;   # содержит сообщение Hello, sailor number 4!
```

Пример 2. Программа, использующая подпрограмму сложения двух переменных.

```
sub sum_of_fred_and_barney {
    print “Вы вызвали процедуру sum_of_fred_and_barney!\n”;
    $fred + $barney;          #Возвращаемое значение
}
$fred = 3;
$barney = 4;
$c = &sum_of_fred_and_barney;      # $c получает 7
print “\$c равно $c.\n”;
$d = 3 * &sum_of_fred_and_barney; #$d получает 21
print “\$d равно $d.\n”;
```

Пример 3. «Последнее вычисляемое значение»

```
sub larger_of_fred_and_barney {
    if ($fred > $barney)       {
            $fred;
    } else     {
            $barney;
    }
}
$fred = 3;
$barney = 4;
$c = &larger_of_fred_and_barney; # $c получает 4
print “\$c равно $c.\n”;
```

Пример 4. Формирование интервала чисел.

```
sub list_of_fred_and_barney {
    if ($fred < $barney)        {
# Формирует интервал чисел от $fred до $barney (по возрастанию)
            $fred..$barney;
    } else    {
# Формирует интервал чисел от $fred до $barney (по убыванию)
            reverse $barney..$fred;
    }
}
$fred = 11;
$barney = 6;
@c = &list_of_fred_and_barney;
# Массив @c получает значения 11, 10, 9, 8, 7, 6
```

Пример 5. Работа с аргументами.

```
sub max {
    if ($_[0] > $_[1]) {
            $_[0];
    } else    {
            $_[1];
    }
}
$c = &max(10, 15); # Запишет 15
$d = &max(5, 4, 10);
# Проигнорирует третий параметр и запишет 5
```

Пример 6. Усовершенствованная процедура &max.

```
$maximum =  &max(3, 5, 10, 4, 6);
sub max {
        my($max_so_far) = shift @_;
        #присваиваем индивидуальной переменной $max_so_far
        #значение первого параметра и удаляем его из массива
    foreach (@_) {
            If ($_ > $max_so_far) {
                $max_so_far = $_;
            }
    }
    $max_so_far;
}
```

Пример 7. Рекурсивная процедура просмотра дерева каталогов.

```
sub tree {
    local (*ROOT);
    my ($root)=$_[01;
    opendir 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);
                }
            }
    }
}
```

Пример 8. Нахождение наименьшего общего кратного НОК и наибольшего общего делителя НОД двух чисел.

```
sub nod {
    my($tmp);
    $tmp =  $_[1] % $_[0];
    if ($_[0] != 0) {
    nod($tmp, $_[0]);
    } else    {
    $_[1];
    }
}
sub nok {
    my($tmp) = nod($_[0], $_[1]);
    $_[1] * ($_[0]- $_[0] % $tmp) / $tmp;
}
$a = $ARGV[0];
$b = $ARGV[1];
$c = nod($a, $b);
$d = nok($a, $b);
print “НОК $a и $b = $d, а НОД $a и $b = $c”;
```

Пример 9. Секретный код.

```
my $secret = int(1 + rand 100);
# “секретным считается число $secret.\n”;
while (1) {
print “Пожалуйста, введите число из диапазона от 1 до 100: ”;
    chomp(my $guess = <STDIN>);
    if ($guess =~ /quit|exit|^\s*$/i) {
            print “Вы сдались. Секретным числом было $secret.\n”;
            last;
    } elsif ($guess < $secret) {
            print “Слишком малое. Попробуйте еще раз!\n”;
    } elsif ($guess == $secret) {
            print “Вы угадали!\n”;
            last;
    } else    {
            print “Слишком большое. Попробуйте еще раз!\n”;
    }
}
```

Пример 10.

```
#!/usr/bin/perl
package Staff;
sub new {
my ($class, $data) = @_;
my $self = $data;
bless $self, $class;
return $self;
}
sub setdata {
my ($self,$data) = @_;
for $i (keys %$data) {
$self->{$i}=$data->{$i};
}
return $self;
}
sub showdata {
my $self = shift;
my @keys = @_ ? @_ : sort keys %$self;
foreach $key (@keys) {
print "\t$key => $self->{$key}\n";
}
return $self;
```

Лабораторное задание и порядок выполнения работы

1. Изучить материал, выполняя рекомендуемые примеры и задания.
2. Выполнить контрольное задание, используя свои файлы и каталоги. Возможны добавления в сценарии.
3. Кратко законспектировать материал по новым командам.

Оформить отчет и защитить работу.

Требования к отчету

Отчет должен содержать:

1. Краткие сведения о работе.
2. Описание команд и сценариев, выполненных в данной работе (с Вашими именами файлов и каталогов).