#### Министерство образования и науки Российской Федерации

#### Федеральное государственное бюджетное образовательное учреждение

#### высшего профессионального образования

#### «Владимирский государственный университет

#### имени Александра Григорьевича и Николая Григорьевича Столетовых»

#### (ВлГУ)

Кафедра информационных систем   
и программной инженерии

Х.М. Салех, Д.И. Левковский

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ИНФОРМАЦИОННЫХ СИСТЕМ

Методические указания к лабораторным работам

Владимир 2013

Содержание

[Лабораторная работа №1 3](#_Toc377147466)

[Лабораторная работа №2 15](#_Toc377147469)

[Лабораторная работа №3 20](#_Toc377147471)

[Лабораторная работа №4 48](#_Toc377147473)

[Лабораторная работа №5 98](#_Toc377147475)

[Лабораторная работа №6 115](#_Toc377147477)

[Лабораторная работа №7-8 144](#_Toc377147479)

лАБОРАТОРНАЯ РАБОТА №1

знакомство с операционной системой linux. BASH и shell

**1. Цель работы**

Познакомиться с ОС Linux. Изучить основы работы с BASH и языком программирования SHELL.

**2. Общие сведения**

***2.1 Потоки ввода и вывода данных***

Логически все файлы в системе Linux организованы в непрерывный поток байтов. Любой файл можно свободно копировать и добавлять к другому файлу, так как все файлы организованы одинаково.

Эта логическая организация файлов распространяется на операции ввода и вывода. Данные в операциях ввода и вывода организованы аналогично файлам. Данные, вводимые с клавиатуры, направляются в поток данных, организованный как непрерывная совокупность байтов. Данные, выводимые из команды или программы, также направляются в поток, организованный как непрерывная совокупность байтов. Входной поток данных в ОС Linux называется стандартным вводом, а выходной поток данных - стандартным выводом.

Поскольку стандартный ввод и стандартный вывод организованы так же, как файл, они свободно могут взаимодействовать с файлами. В ОС Linux широко используется переадресация, которая позволяет перемещать данные в файлы и из файлов.

*Переадресация стандартного вывода: > и >>*

Когда выполняется команда ОС Linux, дающая какую-либо выходную информацию, эта информация направляется в поток данных стандартного вывода. По умолчанию в качестве пункта назначения данных стандартного вывода используется какое-либо устройство, в данном случае экран. Устройства, такие как клавиатура и экран, тоже рассматриваются как файлы. Они принимают и отправляют потоки байтов, имеющих такую же организацию, как и файлы байтового потока. Экран - это устройство, на котором отображается непрерывный поток байтов. По умолчанию стандартный вывод посылает свои данные на экран, где они отображаются.

Для направления стандартного вывода в файл, а не на экран, необходимо использовать оператор переадресации вывода: >. С помощью операции переадресации создается новый файл-адресат. Если он уже существует, то система заменит его содержимое данными стандартного вывода. Для того чтобы этого не произошло, можно установить для shell режим noclobber:

set -o noclobber

В этом случае операция переадресации существующего файла выполнена не будет. Отменить режим noclobber можно, поставив после оператора переадресации восклицательный знак:

cat file1 >! file2

Хотя оператор переадресации и имя файла ставятся после команды, перенаправление стандартного потока выполняется не после выполнения команды, а до него. С помощью оператора переадресации создается файл и переадресация организуется до того, как начинают поступать данные со стандартного вывода. Если файл уже существует, он будет разрушен и заменен новым файлом под тем же именем. Команда, генерирующая выходные данные, выполняется только после создания файла переадресации.

Если пользователь попытается использовать одно и то же имя для входного файла команды и переадресованного файла, возникнет ошибка. Так как операция переадресации выполняется первой, входной файл, поскольку он существует, разрушается и заменяется файлом с тем же именем. Когда команда начинает выполняться, она обнаруживает пустой входной файл. Для добавления стандартного вывода к существующему файлу служит оператор переадресации >>.

*Переадресация стандартного ввода: < и <<*

Многие команды ОС Linux могут принимать данные со стандартного ввода. Сам стандартный ввод получает данные из устройства или из файла. По умолчанию в качестве устройства для стандартного ввода используется клавиатура. Символы, набираемые на клавиатуре, подаются на стандартный ввод, который затем направляется в команду.

Во многих Linux-системах применяется метод буферизации строк. При буферизации строк, информация посылается на стандартный ввод только после того, как пользователь ввел всю строку.

Стандартный ввод можно переадресовать так же, как и стандартный вывод. Стандартный ввод может приниматься не с клавиатуры, а из файла. Оператор переадресации стандартного ввода: <. Кроме этого для переадресации стандартного ввода применяется механизм "файл здесь":

cat << 'слово-признак конца ввода'

> 'текст'

> 'текст'

> 'слово-признак конца ввода'

Операции переадресации стандартного ввода и стандартного вывода можно объединять.

Переадресация и пересылка по каналу стандартного потока ошибок: >&, 2>.

При выполнении команд иногда происходят ошибки. Например, пользователь указал неверное количество аргументов или возникла какая-то системная ошибка. Когда возникает ошибка, система выдает специальное сообщение. Как правило, такие сообщения об ошибках отображаются на экране вместе со стандартным выводом. ОС Linux, однако, различает стандартный вывод и сообщения об ошибках. Сообщения об ошибках помещаются еще в один стандартный байтовый поток, который называется стандартным потоком ошибок (диагностики).

Так как сообщения об ошибках направляются в поток, отдельный от стандартного вывода, то в случае переадресации стандартного вывода в файл они все равно появляются на экране.

Стандартный поток ошибок можно переадресовать так же, как и стандартный вывод. Например, сообщения об ошибках можно сохранить в файле для справок. Как и в случае стандартного вывода, пунктом назначения стандартного потока ошибок по умолчанию является экран, однако с помощью специальных операторов переадресации его можно переадресовать в любой файл или устройство.

Для переадресации стандартного потока ошибок в shell предусмотрена специальная возможность. Все стандартные байтовые потоки в операциях переадресации можно обозначать номерами (дескрипторами). Номера 0, 1 и 2 обозначают соответственно стандартный ввод, стандартный вывод и стандартный поток ошибок. Оператор переадресации вывода, >, по умолчанию действует на стандартный вывод, 1. Чтобы переадресовать стандартный поток ошибок, нужно поставить перед оператором переадресации вывода цифру 2.

Стандартный поток ошибок можно дописать в файл, используя цифру 2 и оператор добавления: >>. Для того чтобы переадресовать и стандартный вывод, и стандартный поток ошибок, нужны две операции переадресации и два файла.

В BASH можно ссылаться на стандартный поток по его номеру со знаком "&": &1 обозначает стандартный вывод. Это обозначение можно использовать в операции переадресации для того, чтобы сделать стандартный вывод файлом назначения. Операция переадресации 2>&1 переадресует стандартный поток ошибок на стандартный вывод. В результате стандартный вывод становится файлом назначения для стандартного потока ошибок. Операция переадресации 1>&2 переадресует стандартный ввод в стандартный поток ошибок. По умолчанию входным потоком операции >& является стандартный поток ошибок, а выходным потоком - стандартный вывод. Поэтому, если его использовать в команде, все сообщения о ошибках будут перенаправляться на стандартный вывод.

*Программные каналы: |*

Иногда возникают ситуации, когда нужно передать данные из одной команды в другую. Другими словами, необходимо послать стандартный вывод одной команды на стандартный ввод другой, а не в файл. Для образования такого соединения в ОС Linux используется так называемый канал. Оператор канала, | (вертикальная черта), помещенный между двумя командами, связывает их стандартные потоки. Стандартный вывод одной команды становится стандартным вводом другой. Выходная информация команды, стоящей перед оператором канала, передается в качестве входной в команду, стоящую за оператором канала.

Программные каналы можно объединять с другими средствами shell, например со специальными символами, проводя таким образом специализированные операции.

Если операция переадресации позволяет просто направлять выходную информацию в файл, то каналы обеспечивают ее пересылку в другую команду Linux. Следует помнить о различии между файлом и командой. Файл - это носитель данных. Вы можете хранить или читать из него данные. Команда - это программа, которая исполняет инструкции. Команда может читать данные из файла и сохранять данные в файле, но во время выполнения ее нельзя рассматривать как файл. По этой причине операция переадресации выполняется с файлами, а не с командами. В процессе переадресации данные посылаются из программы в файл, а не в другую программу. Пунктом назначения операции переадресации могут быть только файлы, но не программы.

Можно, тем не менее, смоделировать процесс конвейерной пересылки с помощью нескольких операций переадресации. Выходная информация одной команды посылается в файл. Команда, записанная в следующей строке, использует этот файл как переадресованный ввод.

Каналы работают со стандартным выводом команды независимо от того, что подается на этот вывод. Пересылаться по каналу из одной команды в другую может cодержимое целого файла и даже нескольких файлов.

Стандартным вводом, посылаемым по каналу в команду, можно более эффективно управлять с помощью аргумента стандартного ввода, **-**. Дефис, используемый в команде в качестве аргумента, обозначает стандартный ввод.

Допустим, нужно напечатать файл с именем его каталога в начале списка. Команда pwd выдает имя каталога, а команда cat выдает содержимое файла. В данном случае команде cat нужно использовать в качестве входной информации и файл, и стандартный ввод, пересланный по каналу из команды pwd. Команда cat будет иметь два аргумента: стандартный ввод, обозначенный дефисом, и имя выводимого на печать файла.

pwd | cat - file | lpr

*Каналы и переадресация: команда tee*

Для того, чтобы переадресовать стандартный вывод в файл и одновременно воспроизвести эту информацию на экране необходимо использовать команду tee. Команда tee копирует стандартный вывод в файл. В качестве аргумента она использует имя нового файла, в который копируется стандартный вывод. Это все равно что скопировать содержимое стандартного вывода и один его экземпляр переадресовать в файл, а другой отправить дальше (часто - на экран).

Переадресацией в сочетании с каналами необходимо пользоваться осторожно. Переадресация стандартного вывода задает файл назначения для него. Стандартный вывод записывается и сохраняется в этом файле. После записи никакой информации для пересылки по каналу в другую команду не остается. Переадресация может производиться в конце последовательности программных каналов, но не внутри этой последовательности.

***2.2 Программирование в BASH-shell***

*Shell*

Командный интерпретатор в среде UNIX выполняет две основные функции:

* представляет интерактивный интерфейс с пользователем, т.е. выдает приглашение, и обрабатывает вводимые пользователем команды;
* обрабатывает и исполняет текстовые файлы, содержащие команды интерпретатора (командные файлы).

В последнем случае, операционная система позволяет рассматривать командные файлы как разновидность исполняемых файлов. Соответственно различают два режима работы интерпретатора: интерактивный и командный.

Существует несколько типов оболочек в мире UNIX. Две главные - это «Bourne shell» и «C shell». Bourne shell (или просто shell) использует командный синтаксис, похожий на первоначально для UNIX. В большинстве UNIX-систем Bourne shell имеет имя /bin/sh (где sh сокращение от «shell»). C shell использует иной синтаксис, чем-то напоминающий синтаксис языка программирования Си. В большинстве UNIX-систем он имеет имя /bin/csh.

В Linux есть несколько вариаций этих оболочек. Две наиболее часто используемые, это Новый Bourne shell (Bourne Again Shell) или «Bash» (/bin/bash) и Tcsh (/bin/tcsh). Bash - это развитие прежнего shell с добавлением многих полезных возможностей, частично содержащихся в C shell.

Поскольку Bash можно рассматривать как надмножество синтаксиса прежнего shell, любая программа, написанная на sh shell должна работать и в Bash. Tcsh является расширенной версией C shell. При входе в систему пользователю загружается командный интерпретатор по умолчанию. Информация о том, какой интерпретатор использовать для конкретного пользователя находится в файле /etc/passwd.

Возможно, вам захочется выполнить сценарий, написанный для одного из shell Linux, в то время как вы работаете в другом. Предположим, вы работаете в TCSH-shell и хотите выполнить написанный в BASH сценарий, содержащий команды этого (второго) shell. Сначала нужно с помощью команды sh перейти в BASH-shell, выполнить сценарий, а затем вернуться в TCSH. Эту процедуру можно автоматизировать, поставив первыми в сценарии символы #! и указав после них путевое имя программы нужного shell в вашей системе. Shell всегда изучает первые символы сценария и на их основании делает вывод о том, к какому типу shell этот сценарий относится - BASH, PDKSH или TCSH. Если первый символ - пробел, это сценарий BASH-shell или PDKSH-shell. Если первый символ - знак #, это сценарий TCSH-shell. Если первые символы - #!, то shell читает указанное за ними имя программы. После символов #! всегда должно следовать путевое имя программы нужного shell, по которому можно идентифицировать его тип. Если вы запускаете сценарий из shell, отличного от того, который указан в первой строке запускаемого сценария, то будет вызван shell, указанный в первой строке, и в нем выполнится ваш сценарий. В такой ситуации одного пробела или знака 41 для указания того, что это сценарий BASH или TCSH, бывает недостаточно. Такая идентификация работает только в собственных сценариях этих shell. Чтобы обозначить сценарий другого shell, необходимо поставить символы #! и путевое имя. Например, если поставить в начало первой строки сценария hello комбинацию символов #!/bin/sh, то этот сценарий можно будет выполнять непосредственно из TCSH-shell. Сначала сценарий осуществит переход в BASH, выполнит его команды, а затем вернется в TCSH (или в тот shell, из которого он выполнялся). В следующем примере сценарий hello содержит команду #!/bin/sh. Пользователь выполняет сценарий, находясь в TCSH-shell.

*Командные файлы*

Командный файл в Unix представляет собой обычный текстовый файл, содержащий набор команд Unix и команд Shell. Для того чтобы командный интерпретатор воспринимал этот текстовый файл, как командный необходимо установить атрибут на исполнение.

$ echo “ps –af” > commandfile

$ chmod +x commandfile

$ ./commandfile

В представленном примере команда echo “ps -af ” > commandfile создаст файл с одной строкой “ps –af”, команда chmod +x commandfile установит атрибут на исполнение для этого файла, команда ./commandfile осуществит запуск этого файла.

*Переменные shell*

Имя shell-переменной - это начинающаяся с буквы последовательность букв, цифр и подчеркиваний. Значение shell-переменной - строка символов.

Например: Var = “String” или Var = String

Команда echo $Var выведет на экран содержимое переменной Var т.е. строку “String”, на то что мы выводим содержимое переменной указывает символ “$”. Так команда echo Var выведет на экран просто строку «Var».

Еще один вариант присвоения значения переменной Var = ‘набор команд Unix’. Обратные кавычки говорят о том, что сначала должна быть выполнена заключенная в них команда), а результат ее выполнения, вместо выдачи на стандартный выход, приписывается в качестве значения переменной.

CurrentDate = `date`

Переменной CurrentDate будет присвоен результат выполнения команды date. Можно присвоить значение переменной и с помощью команды “read”, которая обеспечивает прием значения переменной с (клавиатуры) дисплея в диалоговом режиме.

echo “Введите число”

read X1

echo “вы ввели -” $X1

Несмотря на то, что shell-переменные в общем случае воспринимаются как строки, т. е. “35” - это не число, а строка из двух символов “3” и “5”, в ряде случаев они могут интерпретироваться иначе, например, как целые числа.

Разнообразные возможности имеет команда “expr”.

x=7

y=2

rez=’expr $x + $y’

echo результат=$rez

выдаст на экран результат=9

*Параметры командного файла*

В командный файл могут быть переданы параметры. В shell используются позиционные параметры (т.е. существенна очередность их следования). В командном файле соответствующие параметрам переменные (аналогично shell-переменным) начинаются с символа “$”, а далее следует одна из цифр от 0 до 9. При обращении к параметрам перед цифрой ставится символ доллара “$” (как и при обращении к переменным):

$0 соответствует имени данного командного файла;

$1 первый по порядку параметр;

$2 второй параметр и т.д.

Поскольку число переменных, в которые могут передаваться параметры, ограничено одной цифрой, т.е. 9-ю ("0", как уже отмечалось имеет особый смысл), то для передачи большего числа параметров используется специальная команда "shift".

Команда “set” устанавливает значения параметров. Это бывает очень удобно. Например, команда “date” выдает на экран текущую дату, скажем, “Mon May 01 12:15:10 2002”, состоящую из пяти слов, тогда set ‘date echo $1 $3 $5’ выдаст на экран “Mon 01 2002”.

*Программные структуры*

Как во всяком процедурном языке программирования в языке shell есть операторы. Ряд операторов позволяет управлять последовательностью выполнения команд. В таких операторах часто необходима проверка условия, которая и определяет направление продолжения вычислений.

*Команда test*

Команда test проверяет выполнение некоторого условия. С использованием этой (встроенной) команды формируются операторы выбора и цикла языка shell. Два возможных формата команды:

test условие

или

[ условие ]

В shell используются условия различных "типов".

Условия проверки файлов:

-f file файл "file" является обычным файлом;

-d file файл "file" - каталог;

-с file файл "file" - специальный файл;

-r file имеется разрешение на чтение файла "file";

-w file имеется разрешение на запись в файл "file";

7

-s file файл "file" не пустой.

Условия проверки строк:

str1 = str2 строки "str1" и "str2" совпадают;

str1 != str2 строки "str1" и "str2" не совпадают;

-n str1 строка "str1" существует (непустая);

-z str1 строка "str1" не существует (пустая).

Условия сравнения целых чисел:

x -eq y "x" равно "y",

x -ne y "x" неравно "y",

x -gt y "x" больше "y",

x -ge y "x" больше или равно "y",

x -lt y "x" меньше "y",

x -le y "x" меньше или равно "y".

То есть в данном случае команда "test" воспринимает строки символов как целые (!) числа. Поэтому во всех остальных случаях "нулевому" значению соответствует пустая строка. В данном же случае, если надо обнулить переменную, скажем, "x", то это достигается присваиванием "x=0". Сложные условия реализуются с помощью типовых логических операций:

! (not) инвертирует значение кода завершения.

-o (or) соответствует логическому "ИЛИ".

-a (and) соответствует логическому "И"

*Управляющие структуры*

С помощью управляющих структур пользователь может осуществлять контроль над выполнением Linux-команд в программе. Управляющие структуры позволяют повторять команды и выбирать для выполнения команды, необходимые в конкретной ситуации. Управляющая структура состоит из двух компонентов: операции проверки и команд. Если проверка считается успешной, то выполняются команды. Таким образом, с помощью управляющих структур можно принимать решения о том, какие команды следует выполнять.

Существует два вида управляющих структур: циклы и условия. Цикл используется для повторения команд, тогда как условие обеспечивает выполнение команды при соблюдении некоторых критериев. В BASH-shell используются три вида циклических управляющих структур (while, for и for-in) и две условные управляющие структуры (if и case).

Управляющие структуры while и if - это структуры общего назначения. Они используются, например, для выполнения итераций и принятия решений на основании различных проверок. Управляющие структуры case и for - более специализированные. Структура case представляет собой модифицированную форму условия if и часто используется для построения меню. Структура for - это цикл ограниченного типа. Здесь обрабатывается список значений, и на каждой итерации цикла переменной присваивается новое значение.

В управляющих структурах if и while проверка построена на выполнении Linux-команды. Все команды ОС Linux после выполнения выдают код завершения. Если команда выполнена нормально, выдается код 0. Если по какой-либо причине команда не выполняется, то выдается положительное число, обозначающее тип отказа. Управляющие структуры if и while проверяют код завершения Linux-команды. Если код - 0, выполняются действия одного типа, если нет - другого.

*Дополнительная информация*

Во-первых, обязательно обратитесь к man bash.

Во-вторых, краткая памятка по командам, которые вам могут понадобиться:

pwd Вывести текущую директорию.

hostname Вывести или изменить сетевое имя машины.

whoami Ввести имя под которым я зарегистрирован.

date Вывести или изменить дату и время. Например, чтобы установить дату и время равную 2000-12-31 23:57, следует выполнить команду: date 123123572000

time Получить информацию о времени, нужного для выполнения процесса + еще кое-какую информацию. Не путайте эту команду с date. Например: Я могу определить как много времени требуется для вывода списка файлов в директории, набрав последовательность: time ls

who Определить кто из пользователей работает на машине.

rwho –a Определение всех пользователей, подключившихся к вашей сети. Для выполнения этой команды требуется, чтобы был запущен процесс rwho. Если такого нет - запустите setup "setup" под суперпользователем.

finger [имя\_пользователя] Системная информация о зарегистрированном пользователе. Попробуйте: finger root

uptime Количество времени прошедшего с последней перезагрузки.

ps –a Список текущих процессов.

top Интерактивный список текущих процессов отсортированных по использованию

cpu.uname Вывести системную информацию.

free Вывести информацию по памяти.

df –h (=место на диске) Вывести информацию о свободном и используемом месте на дисках (в читабельном виде).

du / -bh | more (=кто сколько занял) Вывод детальной информации о размере файлов по директориям начиная с корневой (в читабельном виде).

cat /proc/cpuinfo Системная информация о процессоре. Заметьте, что файла в /proc директории - не настоящие файлы. Они используются для получения информации, известной системе.

cat /proc/interrupts Используемые прерывания.

cat /proc/version Версия ядра Linux и другая информация

cat /proc/filesystems Вывести используемые в данный момент типы файловых систем.

cat /etc/printcap Вывести настройки принтера.

lsmod (как root) Вывести информацию о загруженных в данный момент модулях ядра.

set | more Вывести текущие значения переменных окружения.

echo $PATH Вывести значение переменной окружения "PATH". Эта команда может использоваться для вывода значений других переменных окружения. Воспользуйтесь командой set, для получения полного списка.

grep … Поиск вхождения регулярного выражения в строки заданного файла (потока).

**3. Задания для выполнения**

1. Вывести любое сообщение с помощью команды echo, перенаправив вывод:

* в несуществующий файл с помощью символа >;
* в несуществующий файл с помощью символа >>;
* в существующий файл с помощью символа >;
* в существующий файл с помощью символа >>.

Объяснить результаты.

1. Переадресовать стандартный ввод для команды cat на файл.
2. Вывести сообщение с помощью команды echo в канал ошибок. Создать файл myscript:

#!/bin/sh

echo stdout

echo stderr>&2

exit 0

Запустить его:

* без перенаправления (sh myscript);
* перенаправив стандартный вывод в файл, просмотреть содержимое файла (sh myscript > file1);
* перенаправить стандартный канал ошибок в существующий и несуществующий
* файлы с помощью символов > и >> (а тут и дальше уже сами :) );
* перенаправив стандартный вывод в файл 1, стандартный канал ошибок - в файл 2; перенаправив стандартный вывод и стандартный канал ошибок в файл 3;
* перенаправив стандартный вывод в файл 4 с помощью символа >, а стандартный
* канал ошибок в файл 4 с помощью символа >>.

Объяснить результаты.

1. Вывести третью строку из последних десяти строк отсортированного в обратном порядке файла /etc/group.
2. Подсчитать при помощи конвейера команд количество блочных устройств ввода-вывода, доступных в системе.
3. Написать скрипт, выводящий на консоль все аргументы командной строки, переданные данному скрипту. Привести различные варианты запуска данного скрипта, в том числе без непосредственного вызова интерпретатора в командной строке.
4. Написать скрипт согласно индивидуальному заданию. Номер варианта согласовать с преподавателем.

**4. Варианты индивидуальных заданий.**

1. Написать командный файл, реализующий меню из трех пунктов: 1-ый пункт – ввести пользователя и вывести на экран все процессы, запущенные данным пользователем; 2-ой пункт - показать всех пользователей, в настоящий момент, находящихся в системе; 3-ий пункт – завершение.
2. Написать командный файл, реализующий меню из трех пунктов: 1-ый пункт - вывести всех пользователей, в настоящее время, работающих в системе; 2-ой пункт – послать сообщение пользователю, имя пользователя, терминал и сообщение вводятся с клавиатуры; 3-ий пункт – завершение.
3. Написать командный файл, реализующий меню из трех пунктов: 1-ый пункт - показать все процессы пользователя, запустившего данный командный файл; 2-ой пункт – послать сигнал завершения процессу текущего пользователя (ввести PID процесса); 3-ий пункт – завершение.
4. Написать командный файл, посылающий сигнал завершения процессам текущего пользователя. Символьная маска имени процесса вводится с клавиатуры.
5. Написать командный файл подсчитывающий количество определенных процессов пользователя (Ввести имя пользователя и название процесса).
6. Реализовать Меню из двух пунктов: 1-ый пункт – определить количество запущенных данным пользователем процессов bash (предусмотреть ввод имени пользователя); 2-ой пункт – завершить все процессы bash данного пользователя.
7. Реализовать Меню из трех пунктов: 1-ый пункт поиск файла в каталоге <Имя файла> и <Имя каталога> вводятся пользователем; 2-ой пункт – копирование одного файла в другой каталог - <Имя файла> и <Имя каталога> вводятся; 3-ий пункт – завершение командного файла.
8. Написать командный файл который в цикле по нажатию клавиши выводит информацию о системе, активных пользователях в системе, а для введенного имени пользователя выводит список активных процессов данного пользователя.
9. Реализовать командный файл который при старте выводит информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе – запрашивает имя пользователя и выводят список всех процессов bash запущенных данным пользователем.
10. Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего командный файл пользователя, тип операционной системы, IP-адрес машины.
11. Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям (исключая пользователя, запустившего данный командный файл) сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего командный файл пользователя, тип операционной системы, список загруженных модулей.
12. Реализовать командный файл, который при старте выводит информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе – запрашивает имя пользователя и выводят список всех терминалов, на которых зарегистрирован этот пользователь.
13. Реализовать командный файл, который выводит: дату, информацию о системе, текущий каталог, текущего пользователя, настройки домашнего каталога текущего пользователя, далее в цикле выводит список активных пользователей – запрашивает имя пользователя и выводит информацию об активности данного пользователя.
14. Реализовать командный файл, который выводит: дату в формате день – месяц – год – время; информацию о системе в формате: имя компьютера: версия ОС: IP адрес: имя текущего пользователя: текущий каталог; настройки домашнего каталога текущего пользователя; основные переменные окружения. Далее в цикле выводит список активных пользователей – запрашивает имя пользователя и выводит информацию об активности введенного пользователя.
15. Реализовать командный файл, реализующий символьное меню (в цикле):

1 Пункт: вывод полной информации о файлах каталога: ввести имя каталога для отображения;

2 Пункт: изменить атрибуты файла: файл вводится с клавиатуры по запросу, атрибуты, которые требуются установить, тоже вводятся. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов;

3 Пункт: выход: при старте командный файл выводит информацию об имени компьютера, IP-адреса, и список всех пользователей, зарегистрированных в данный момент на компьютере.

1. Реализовать командный файл, реализующий символьное меню (в цикле):

1 Пункт: вывод полной информации о файлах каталога: ввести имя каталога для отображения;

2 Пункт: создать командный файл: файл вводится с клавиатуры по запросу, далее изменяются атрибут файла на исполнение, затем вводится с клавиатуры строка, которую будет исполнять командный файл. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов и запустить созданный командный файл;

3 Пункт: выход: при старте командный файл выводит информацию об имени компьютера, IP-адреса, и список всех пользователей, зарегистрированных в данный момент на компьютере.

1. Реализовать командный файл, реализующий символьное меню:

1 Пункт: работа с информационными командами (реализовать все основные информационные команды);

2 Пункт: копирование файлов: в этом пункте выводится информация о содержимом текущего каталога, далее предлагается интерфейс копирования файла: ввод имени файла и ввод каталога для копирования. По выполнению пункта выводится содержимое каталога, куда был скопирован файл и выводится содержимое скопированного файла;

3 Пункт: выход.

1. Написать скрипт с использованием цикла for, выводящий на консоль размеры и права доступа для всех файлов в заданном каталоге и всех его подкаталогах (имя каталога задается пользователем в качестве первого аргумента командной строки).
2. Написать скрипт, находящий в заданном каталоге и всех его подкаталогах все файлы, владельцем которых является заданный пользователь. Имя владельца и каталог задаются пользователем в качестве первого и второго аргумента командной строки. Скрипт выводит результаты в файл (третий аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.
3. Написать скрипт поиска одинаковых по их содержимому файлов в двух каталогах, например, Dir1 и Dir2. Пользователь задает имена Dir1 и Dir2 в качестве первого и второго аргумента командной строки. В результате работы файлы, имеющиеся в Dir1, сравниваются с файлами в Dir2 по их содержимому. На экран выводятся число просмотренных файлов и результаты сравнения.

лАБОРАТОРНАЯ РАБОТА №2

Программирование в BASH

**1. Цель работы**

Изучить основные shell-оболочки, используемые в операционных системах класса Unix. Получить навыки создания BASH-скриптов в ОС Linux.

**2. Общие сведения**

*Более полные сведения по теме «Программирование в BASH» можно получить, изучив «BASH Programming - Introduction HOW-TO» и ознакомившись с man страницами для bash.*

Программирование в BASH заключается в написании shell-скриптов, в соответствии с синтаксисом языка BASH. Такие скрипты обычно имеют расширение sh и для получения возможности запуска должны иметь соответствующие атрибуты.

*Скрипт-файл имеет следующую структуру:*

#!/bin/bash

...

где первая команда указывает имя shell-оболочки, а далее следует текст программы.

Пример простейшего BATH-скрипта, выводящего на экран сообщение приведен ниже.

#!/bin/bash

echo Hello World!

Ниже кратко рассмотрены основные моменты языка BASH.

*Переменные*

В BATH-скриптах можно использовать переменные аналогично другим языкам программирования. Типы данных отсутствуют. Переменная может представлять собой число, символ или строку символов. Нет необходимости объявлять переменную, поскольку присвоение ей значения автоматически создаёт её.

Переменные делятся на локальные и глобальные. Локальные создаются с использованием ключевого слова local. Примеры:

|  |  |
| --- | --- |
| **Использование глобальной переменной** | **Использование локальной переменной** |
| #!/bin/bash  STR="Hello World!"  echo $STR | #!/bin/bash  HELLO=Hello  function hello  {  local HELLO=World  echo $HELLO  }  echo $HELLO  hello  echo $HELLO |

Пример скрипта с переменными:

#!/bin/bash

d="Текущая дата: "$(date +%m)-$(date +%d)-$(date +%Y)

echo $d

*Чтение пользовательского ввода с помощью read*

Использование read является одним из способов ввода с клавиатуры пользовательских данных. Например:

#!/bin/bash

echo "Введите, пожалуйста, Ваше имя и фамилию"

read FN LN #FN - First Name - имя; LN - Last Name - фамилия

echo "Hi! $LN, $FN !"

*Арифметические вычисления*

Арифметические вычисления в BASH-скриптах выполняются при помощи следующих конструкций:

echo $((1+1))

echo $[1+1]

При необходимости использования более сложной математики, следует прибегать к помощи bc для вычисления арифметических выражений. Например,

echo 3/4|bc –l

*Условные операторы*

Условные операторы предоставляют возможность организации ветвлений в программах. Существует несколько форм условных операторов.

Элементарная конструкция оператора if в bash выглядит следующим образом:

if [выражение]; then

# операторы 1

fi

Если выражением внутри квадратных скобок истинно, то выполняются операторы1, в противном случае выполняется следующий за fi оператор.

Более сложная конструкция if записывается в следующем виде:

if [выражение]; then

#операторы1

else

#операторы2

fi

Если выражением внутри квадратных скобок истинно, то выполняются операторы1, в противном случае выполняется операротор2. Ниже приведен пример, иллюстрирующий использование условного оператора:

#!/bin/bash

echo "Введите выражение 1: "

read T1

echo "Введите выражение 2: "

read T2

if [ "$T1" = "$T2" ]; then

echo выражение вычислилось как истина

else

echo выражение вычислилось как ложь

fi

*Циклы for, while и until*

Цикл for немного отличается от аналогов в других языках программирования. Прежде всего, он предоставляет возможность выполнять последовательные действия над "словами" в строке.

Цикл while выполняет фрагмент программы, если тестируемым выражением является истина; и останавливается при условии, если им является ложь (или внутри исполняемого кода встречается явно заданное прерывание цикла).

Цикл until практически идентичен циклу while. Отличие заключается только в том, что код выполняется при условии, если проверяемым выражением является ложь.

Примеры использования операторов организации цикла.

#!/bin/bash

for i in $( ls ); do

echo item: $i

done

В приведенном примере конструкция “i in $(ls)” позволяет заносить на каждой итерации цикла в переменную i различные значения, содержащиеся в $(ls).

Пример цикла while:

#!/bin/bash

COUNTER=0

while [ $COUNTER -lt 10 ]; do

echo The counter is $COUNTER

let COUNTER=COUNTER+1

done

Пример цикла until:

#!/bin/bash

COUNTER=20

until [ $COUNTER -lt 10 ]; do

echo COUNTER $COUNTER

let COUNTER-=1

done

*Функции*

При программировании в BASH можно использовать функции аналогично любому другому языку программирования. Объявление функции имеет следующий синтаксис:

function my\_func

{

# код функции

}.

Вызов функции осуществляется по ее имени.

Пример функций с параметрами:

#!/bin/bash

function print

{

echo $1

}

print Hello

print World

*Создание простых меню*

BASH содержит средства для создания простейших меню. Для этого используется директива select. Ее синтаксис аналогичен синтаксису директивы for. Ниже приведен пример создания простейшего меню.

#!/bin/bash

OPTIONS="Print Quit"

select opt in $OPTIONS; do

if [ "$opt" = "Quit" ]; then

exit

elif [ "$opt" = "Print" ]; then

echo Select "Print"

else

clear

echo Bad option

exit

fi

done

*Использование командной строки*

После вызова BASH программы переменные %1, %2 и т.д. содержат соответственно первый второй и т.д. Параметры командной строки введенные пользователем. Ниже приведен пример использования командной строки.

#!/bin/bash

if [ "$2" != "" ]; then

echo Параметр 1: "$1"

echo Параметр 2: "$2"

else

echo Необходимы два параметра

fi

**3. Задания для выполнения**

Разработать bash-скрипт в соответствии с вариантом задания.

**4. Варианты индивидуальных заданий.**

1. Разработать программу вычисления суммы, среднего арифметического, максимума и минимума последовательности чисел вводимых с клавиатуры. Количество чисел последовательности задается пользователем. Результат выводится в файл.
2. Разработать программу копирования содержимого первого файла во второй. Имена файлов вводятся в командной строке.
3. Разработать программу, отыскивающую в текущем каталоге файл с заданным именем. Имя искомого файла вводится в командной строке.
4. Разработать программу, производящую архивирование одного из подкаталогов домашнего каталога пользователя, в файл, имя которого содержит текущую дату. Архивный файл должен помещаться в /tmp.
5. Разработать программу, осуществляющую поиск всех файлов соответствующих маске (поиск начинается с корневого каталога) и помещающих их список в файл. Маска и имя выходного файла передаются в программу через командную строку.
6. Разработать программу вычисления факториала *n!*. Величина *n* вводится в командной строке.
7. Разработать программу, создающую в каталоге /tmp подкаталог, имя которого содержит текущую дату и осуществляющую копирование в него файлов соответствующих маске и содержащихся в каталоге, заданном путем. Маска и путь вводятся в командной строке.
8. Разработать программу, производящую копирование заданного в командной строке файла, в десять файлов, имена которых генерируются автоматически.
9. Разработать программу записи в файл из вводимой с клавиатуры последовательности чисел, лишь тех, которые входят в диапазон, заданный с клавиатуры.
10. Разработать программу копирующую, удаляющую или переименовывающую файлы в текущем каталоге в соответствии с маской. Маска вводится в командной строке. Выбор операции осуществляется с использованием меню.

лАБОРАТОРНАЯ РАБОТА №3

Разработка CGI-программ на языке Perl

**1. Цель работы**

Приобрести навыки обработки данных на Web-сервере с помощью программ на языке Perl.

**2. Общие сведения**

***2.1 Состав рабочего места***

*Оборудование:* IBM-совместимый персональный компьютер (ПК).

*Программное обеспечение:* операционная система Windows, Web-браузер Internet Explorer версии 5.5 и выше, Web-сервер (Apache или IIS), инструментальные средства языка Perl.

***2.2 Технология работы с Perl***

Для написания программы на языке Perl можно использовать любой текстовый редактор.

Текст программы содержит операторы языка Perl и комментарии. Комментарий в Perl начинается с символа "**#**".

Признаком окончания оператора является символ "**;**". Между элементами операторов должен быть хотя бы один пробел. Кроме того, разделителями могут служить также табуляторы и символы перехода на новую строку.

Первая строка программы Perl должна иметь следующий вид:

**#!** *путь-к-интерпретатору-Perl*

например,

#! /usr/bin/perl

Хотя этот оператор имеет смысл только при работе под управлением UNIX (в среде Windows он игнорируется), имеет смысл указывать его для обеспечения переносимости программы в другую операционную систему.

Текст программы может сохраняться с произвольным именем, однако в Windows желательно (но не обязательно) сохранять его с расширением *pl*, например, *hello.pl*.

Поскольку Perl является языком интерпретирующего типа, при запуске команды

**perl** *имя-файла*

программа сразу начинает выполняться. Пример запуска программы:

**perl hello.pl**.

ИнтерпретаторPerl не обращает особого внимания на то, как выглядит код, лишь бы он работал. Однако имеется возможность ука­зать интерпретатору Perl, чтобы тот более внимательно анализировал исходный код. На­пример, вывод информации о предупреждениях (опция **-w**), включенная в командную строку или в первую строку програм­мы, позволяет избежать многих неприятных ошибок. Интерпретатор Perl может предупреждать об использовании неопределенных переменных, однократном использовании имен и т.п.

Кроме ключа **-w**, в интерпретаторе Perl имеются и другие средства для вывода дополнительных предупреждений во время компиляции. Для этого ис­пользуется оператор **use strict**.

Оператор **use strict** называется *директивой интерпретатора.* Эта директива указывает Perl, что необходимо отслеживать различные ситуации, в которых потенциально возможны ошибки, и выводить сообще­ния об ошибках времени выполнения с информацией о текущем файле и блоке.

***2.3 Переменные в языке Perl***

В языке Perl существует три типа данных и соответственно им три типа переменных:

* скалярные переменные;
* массивы скалярных переменных;
* ассо­циативные массивы скалярных переменных (или хэш - массивы).

*Скалярный тип* является базовым типом, на основании которого строятся более сложные структуры данных. Скалярная переменная содержит одно значение, которое может быть числом, строкой или ссылкой на другую пе­ременную. В Perl каждый скаляр (или переменная скалярного типа) может содержать от­дельное слово, запись, документ, строку текста, символ или ссылку на другую пе­ременную. Скаляры Perl могут со­держать также *литеральные данные,* т.е. данные, значение которых не изменяется во время выполнения программы. В некоторых языках программирования для такого рода данных используются термины *константа* или *литерал.*

Perl содержит два различных типа скалярных литералов: *числовые и строковые литералы*.

*Числовые литералы* могут быть целыми числами, числами с десятичной точкой и числами, записанными в экспоненциальной форме. Числовые литералы могут быть записаны, помимо десятичной формы, также в восьмеричной, шестнадцатеричной и двоичной формах. Признаком того, что число представлено в восьмеричной форме, является **0** впереди числа. Признаком записи в шестнадцатеричной форме являются символы "**0X**" или "**0x**" впереди числа. И, наконец, признаком записи в двоичной форме является наличие "**0B**" или "**0b**" перед числом.

*Строковые литералы* в Perl могут содержать любое количество данных, а также могут быть пустыми. Строковые литералы заключаются в одинарные (**' '**), или двойные кавычки (**" "**), например, **'abc'**, **"привет"**.

Если внутри кавычек нужно поместить символ кавычки, перед ним необхо­димо поставить символ обратной косой черты (обратный слеш). Этот символ, используемый внутри строко­вого литерала, говорит Perl, что идущий за ним символ не должен восприниматься как управляющий, например, **'don\'t'**. То же самое относится и к двойным кавычкам.

Основное отличие между одинарными и двойными кавычками заключается в том, что строка в одинарных кавычках является литералом в чистом виде. Поэтому ее содержимое никак не интерпретируется Perl. В строках, заключенных в двойные кавыч­ки, могут находиться имена переменных и *последовательности управляющих символов.* Для хранения скалярных данных в Perl используются *скалярные переменные*.

Особенностью языка Perl является то, что переменные в программе не нужно явно описывать перед их использованием. Тип переменной определяется начальным символом, предшествующим ее имени. Признаком скалярной переменной является начальный символ "**$**". За символом доллара, называемым *идентификатором типа*, следует имя переменной. Имена переменных в Perl подчиняются следующим правилам:

* имена переменных должны состоять из символов латинского алфавита (**a**–**z** и **A**–**Z**), цифровых символов или символа подчеркивания, причем первый символ переменной не может быть цифрой;
* имена переменных чувствительны к регистру, т.е. прописные и строчные буквы являются разными символами.

Кроме того, в Perl зарезервированы имена некоторых односимвольных переменных. Такие переменные, как **$\_**, **$"**, **$/**, **$2** и **$$**, называются специальными и их не следует использовать как обычные переменные в Perl-программах.

Так переменная **$\_** используется в выражениях, если явно не указана другая переменная.

Для неинициализиро­ванных переменных Perl использует значение, принятое по умолчанию. Если пере­менная используется как число (например, в математическом выражении), ее значе­ние по умолчанию – **0** (нуль), если переменная используется как строка (т.е. почти во всех остальных случаях), используется "", или пустая строка.

***2.4 Простейший ввод-вывод в Perl***

Оператор **<STDIN>** читает вводимые с клавиатуры символы до тех пор, пока пользо­ватель не нажмет клавишу **Enter**. Строка текста, возвращаемая оператором **<STDIN>**, содержит символ перево­да строки, введенный пользователем при нажатии клавиши **Enter**. Обычно не жела­тельно, чтобы в конце введенной строки находился символ перевода строки, т.е. тре­буемая строка должна содержать лишь текст. Для удаления символа перевода строки можно воспользоваться следующей функцией:

**chomp** *имя-переменной*

Функция **chomp** удаляет в строке-аргументе завершающий символ перевода строки. Он также возвращает количество удаленных символов – обычно это 1 или 0, если ничего не удалено.

После окончания ввода с терминала надо оповестить Perl, что ввод данных завершен. Для этого нужно набрать символ конца файла (EOF). Конкретное значение этого символа зависит от операционной системы. В UNIX таким симво­лом является **Ctrl+D**, помещенный в начале строки. В MS-DOS или Windows признаком конца файла являются два идущих подряд символа **Ctrl+Z**, которые могут располагаться в любом месте текста.

Вывод данных на дисплей производится с помощью функции:

**print** *переменная-или-литерал*

Форма записи больших фрагментов Web-страницы с помощью функции **print** является неудобной и поэтому в CGI-программах часто используется другая конструкция, заимствованная из командного интерпретатора UNIX shell и называемая "here document" ("документ здесь") или встроенным документом (inline document), имеющая следующий синтаксис:

**<<***метка*;

*строки-текста*

*метка*

Текст, заключенный между строкой **<<***метка* и сле­дующим вхождением идентификатора *метка*, расположенного в отдельной строке, трактуется как заключенный в двойные кавычки. Такая конструкция позволяет внутри себя использовать символы, которые при заключении в обычные двойные кавычки необходимо маскировать символом "**\**", напри­мер, сами двойные кавычки **"**, символы **@**, **$**, **%**. Между символами **<<** и *меткой* не должно быть пробела, а закры­вающий конструкцию идентификатор *метка* должен располагаться в отдель­ной строке и не заканчиваться точкой с запятой.

***2.5 Арифметические, строковые и логические операции Perl***

Для присваивания в Perl используется оператор при­сваивания "**=**". Этот оператор помешает значение выражения, находящегося в правой части, в переменную, указанную в левой части.

Операции обычно делятся на унарные операции, выполняемые над одним операндом и бинарные, выполняемые над двумя операндами.

К унарным арифметическим операциям относятся операции "**+**" и "**-**", а также операции автоинкремента и автодекремента: "**++**" и "**--**". По своему действию эти операции аналогичны соответствующим операциям в языках C, C++, Java и JavaScript, т.е. увеличивают или уменьшают значение операнда на **1**.

Кроме указанных унарных операций, в Perl определены так называемые *именованные унарные операции*, например:

* **int(5.6234)** – возвращает целую часть аргумента (5);
* **cos(50)** – косинус 50 радиан (0.964966);
* **rand(5)** – возвращается случайное число в диапазоне от 0 до указанного в аргументе; если аргу­мент отсутствует— возвращается случай­ное число в диапазоне от 0 до 1.

Полный список именованных операций находится в разделах **perlop** и **perlfunc** справочного руководства.

Бинарные арифметические операции в языке Perl такие же, как и в других языках программирования: "**+**" (сложение), "**-**" (вычитание), "**\***" (умножение), "**/**" (деление) и "**%**" (остаток от деления). Кроме этого, в Perl определена операция "**\*\***" (возведение в степень).

Язык Perl может оперировать не только с числами, но и со строками. Некоторые именованные унарные операции для строк приведены ниже:

* **length("String1")** – длина строки (**7**);
* **lc("ABC")** – переводит все алфавитные символы строки в нижний регистр (**"abc"**);
* **uc("r21")** – переводит все алфавитные символы строки в верхний регистр (**"R21"**).

Следует отметить, что операции **lc** и **uc** будут выполняться только для строк, в которых используются символы латинских букв.

Бинарным строковым оператором является оператор конкатенации "**.**". Этот оператор берет строку, находящуюся слева от него, и строку справа и возвращает строку, объединяющую две строки.

Операция повторения **x** позволяет повторить первый оператор (количество повторений задается вторым операндом), например, результатом выполнения операции **"0"x3** будет строка **"000"**.

Операции сравнения скалярных величин определены отдельно для числовых: "**>**" (больше чем), "**<**" (меньше чем), "**>=**" (больше или равно), "**<=**" (меньше или равно), "**==**" (равно) и "**!=**" (неравно) и отдельно для строковых значений: "**gt**" (больше чем), "**lt**" (меньше чем), "**ge**" (больше или равно), "**le**" (меньше или равно), "**eq**" (равно) и "**ne**" (неравно).

В Perl определены также следующие логические операции: **!** или **not** (отрицание), **&** (побитовое И), **|** (Побитовое ИЛИ), **^** (побитовое исключающее ИЛИ), **>>** (побитовый сдвиг вправо), **<<** (побитовый сдвиг влево), **&&** или **and** (логическое И), **||** или **or** (логическое ИЛИ).

Следует отметить, что в языке Perl нет специального типа для представления логических данных, но, в то же время, есть логические операции, выполняемые над скалярными вели­чинами. При выполнении логических операций над данными скалярного типа следует руководствоваться следующими правилами:

* Любая строка, кроме пустой строки **""** и строки **"0"**, соответствует логическому значению «истина».
* Любое число, кроме **0**, соответствует значению «истина».
* Любая ссылка соответствует значению «истина».
* Любое неопределенное значение соответствует значению «ложь».

Так как логические значения «истина» и «ложь» не имеют однозначной записи, то логические операции **&&** и **||** в качестве результата возвращают последнее скалярное значение, вы­численное в результате применения операции.

Как и в других производных от C языках, в Perl есть комбинированные операции присваивания, записываемые следующим образом:

*переменная операция* **=** *выражение*

Эта форма записи эквивалентна следующему оператору:

*переменная* **=** *переменная операция выражение*

например,

**$x+=3;**

эквивалентно записи

**$x = $x +3;**

В качестве знаков операций могут использоваться все приведенные выше арифметические и логические операции, а также операция конкатенации.

Как и в других языках, в Perl определено старшинство или приоритетность операций (этот порядок может быть изменен с помощью скобок).

В порядке убывания приоритета операции располагаются следующим образом:

|  |
| --- |
| **\*\*** |
| **\* / % x** |
| **+ - .** |
| **<< >>** |
| **< > <= >= lt gt le ge** |
| **== != eq ne** |
| **&** |
| **|** |
| **&&** |
| **||** |

Язык Perl позволяет использовать в выражениях и строки и числа. При этом, в зависимости от ситуации, интерпретатор выполняет автоматическое преобразование чисел в строковое представление и наоборот. Ниже приведено несколько правил, которыми руко­водствуется программа-интерпретатор:

* если из строки можно выделить число, Perl использует число;
* если число используется в строковом выражении, Perl преобразовывает число в строку;
* если в выражении строка используется там, где должно быть число, Perl использует число **0**.

***2.5 Операторы передачи управления Perl***

***2.5.1 Оператор условия***

Для управления условным выполнением операторов Perl обычно используется оператор if. Синтаксис этого оператора приведен ниже:

if(*выражение*) # Если выражение истинно...

блок-1 # выполняется этот блок кода.

else

*блок-2* # В противном случае выполняется этот блок.

где блок-1 и *блок-2* – один или несколько операторов Perl. В случае нескольких операторов, блок, как и во всех производных от C языках, должен быть заключен в фигурные скобки. Оператор else и следующий за ним блок может быть опущен (в этом случае, когда выражение ложно, выполняется оператор, следующий за блок-1).

Существует еще одна разновидность оператора if, с помощью которой можно прове­рить значения нескольких выражений и выполнить код, соответствующий истинному условию:

if (*выражение-1*) # Если выражение-1 истинно...

блок-1# выполняется этот блок кода.

elseif (*выражение-2*) # Иначе, если выражение-2 истинно...

блок-2 # выполняется этот блок кода.

else

блок-3 # Если ни одно из выражений не истинно,

# выполняется этот блок.

Данный оператор выполняется следующим образом: если выражение, помеченное как *выражение-1,* истинно — выполняется блок кода блок-1*.* Иначе управление передается оператору elseif и проверяется значение *выражения-2.* Если ни *выражение-1,* ни *выражение-2* не являются истинными, выполняется блок-3*.*

Оператор if может иметь другой синтаксис. Если внутри блока имеется только одно выражение, оно может предшествовать if. Так, вместо кода

if(*условное-выражение*) {*выражение*;}

можно написать:

*выражение* if(*условное-выражение*);

***2.5.2 Операторы цикла***

Оператор **while** повторяет блок кода до тех пор, пока некоторое выражение истинно. Вот синтаксис этого оператора:

**while(***выражение***)**

*блок*

Когда интерпретатор Perl встречает оператор **while**, проверяется выполнение условия. Если *выражение* истинно, выполняется *блок* кода. После выполнения всего блока повторно вычисляется значение *выражения,* если оно истинно, блок повторяется, например:

**$counter=0;**

**while ($counter < 10)**

**{**

**$counter++;**

**}**

Оператор **for** является более сложной, но в то же время более универсальной конструкцией для организации циклов в Perl. Его синтаксис выглядит так:

**for (***инициализация***;** *условие***;** *инкремент***)**

*блок*

Когда Perl встречает оператор **for**, выполняется следующая последователь­ность действий:

* вычисляется выражение *инициализации.*
* вычисляется выражение, задающее условие окончания цикла. Если оно ис­тинно, то выполняется *блок* кода.
* после выполнения блока производится приращение счетчика и снова проверяется условие. Если оно по-прежнему истинно, блок кода выполняется повторно. Этот процесс продолжается до тех пор, пока не перестает выпол­няться условие.

***2.5.3 Операторы last и next***

Кроме операторов **for** и **while**, управляющих порядком выполнения бло­ков, в Perl имеются операторы для управления программой *внутри* самих блоков.

Одним из таких операторов является **last**. С его помощью можно выйти из внутреннего выполняемого блока цикла (этот оператор аналогичен оператору **break** в C).

Кроме **last**, в Perl существует также оператор **next**. Он завершает текущую итерацию цикла и передает управление в начало цикла (так же, как оператор **continue** в C).

***2.5.4 Метки***

Язык Perl позволяет помечать блоки и некоторые операторы циклов (**for** или **while**). Для этого перед блоком или оператором помещают специальный идентификатор, который называется *меткой,* например:

**MYBLOCK: { }**

Имена меток следуют тем же правилам, что и имена переменных, за небольшим исключением: в отличие от имен перемен­ных, метки не должны иметь символов **%**, **$**, **@**. Кроме того, имена меток не должны совпадать с зарезервированными словами Perl. Хорошим стилем считается использование в именах меток только прописных букв.

Метку можно указывать в качестве аргумента в таких операторах, как **last** и **next**. Это позволяет досрочно завершить выполнение любого блока операторов.

***2.5.5 Оператор выхода из программы***

Оператор **exit** завершает выполнение программы, после чего управление возвращается операционной системе вместе со специальным *кодом завершения.* Обычно, если программа закончилась успешно, коду завершения присваивается значение **0**, например:

**exit 0;**

Ненулевое значение коду завершения можно присвоить в случае тех или иных аварийных ситуаций в программе, например, отсутствует файл, из которого необходимо произвести чтение данных.

***2.6 Списки и массивы***

***2.6.1 Списки***

Множества объектов в Perl реализованы с помощью *списков данных.* Списки данных могут быть представлены тремя способами: с использованием простых списков, массивов и ассоциативных массивов.

**Список** представляет собой последовательность имен скаляров, разделенных запятыми. Вся последовательность заключена в круглые скоб­ки. Каждый отдельный скаляр называется *элементом списка.*

Если список должен состоять только из обычных строк, можно обойтись без кавычек и запятых путем использования перед списком оператора **qw**. Каждый элемент может быть отделен от соседних элементов символом пробела, табуляторами или символом перевода строки.

Если слева от оператора присваивания находится больше переменных, чем справа, избыточные переменные получают значение **undef**. Если спра­ва больше элементов, чем слева, избыточные элементы справа игнорируются.

Для работы со списками литералов в Perl имеется **оператор диапазона***.* В операторе диапазона задается верхняя и нижняя границы диапазона, разделенные парой точек "**..**". Для того чтобы в списке представить не­сколько диапазонов, используются несколько операторов диапазона, разделенные запятой. Оператор диапазона можно применять не только к числам, но и к строкам.

***2.6.2 Инициализация массива***

Для хранения списка в виде переменной используется **массив**. Признаком переменной, обозначающей массив, является начальный символ имени (префикс) "**@**". Имена переменных-массивов в Perl подчиняются тем же правилам, что и имена остальных переменных.

В отличие от других языков программирования, мас­сив в Perl не нужно заранее объявлять и указывать его размерность. Создать новый массив и поместить в него элементы можно с помощью присвоения имени массива списку.

Для того, чтобы получить доступ к содержимому всего массива, надо использовать его имя в двойных кавычках.

***2.6.3 Доступ к элементам массива***

Доступ к отдельным элементам массива осуществляется посредством их ин­дексов. Индекс элементов массива, как и в других языках, производных от C, начинается с **0** и с каждым элементом увеличивается на **1**.

Для доступа к определенному элементу используется следующий синтаксис:

**$***имя-массива***[***индекс***]**

Язык Perl может оперировать с подгруппой внутри массива, которая называется **сечением** *(slice).* Сечение массива обозначается символом "**@**", свидетельствующим о том, что это группа элементов и квадратными скобками с перечислением индивидуальных элементов мас­сива (диапазоны индексов массива указывать нельзя).

В языке Perl можно добавлять новые элементы к массиву при помощи присваивания значений. При этом размер массива автоматически увеличивается.

Часто требуется определить размер массива, или индекс его последнего элемента. Подобная задача возникает при добавлении или удалении элементов массива. В Perl для решения этой задачи предусмотрено несколько способов.

Первый способ – это использова­ние специальной переменной **$#***имя-массива.* Она возвращает последний допустимый индекс массива.

Другой способ нахождения размера массива – это использование имени массива в скалярном контексте (т.е. там, где в выражении ожидается скаляр), например:

**$size=@boys;**

Переменная **$size** теперь содержит количество элементов массива **@boys**.

При работе с массивами можно также использовать отрицательные индексы, которые соответствуют элементам, расположенным с конца массива.

Для выполнения операций над множеством элементов массива в Perl имеется оператор **foreach**, который имеет следующую форму:

*имя переменной* **(***список-или-массив***)**

*блок*

При выполнении этого оператора для каждого значения, принимаемого переменной *имя-переменной* из *списка-или-массива*, выполняется последовательность в *блоке*. Параметр *список-или-массив* может быть списком скалярных значений, разделенных запятыми, или переменной, представляющей массив.

***2.6.4 Функции для работы с массивами***

Язык Perl располагает набором функций для работы с массивами. Ниже в таблице приведены наиболее употребительные из этих функций:

| Определение | Действие |
| --- | --- |
| **splice @ARRAY, OFFSET [, LENGTH [, LIST]]** | Удаляет **LENGTH** элементов из массива **@array**, начиная с элемента с ин­дексом **OFFSET**, и заменяет их элементами списка **LIST**. Возвращает уда­ленные элементы. Если количество удаляемых элементов **LENGTH** не ука­зано, удаляются все элементы, начиная с элемента **OFFSET**. |
| **shift @ARRAY** | Осуществляет сдвиг элементов массива **@ARRAY** влево с отбрасыванием первого элемента. Возвращает значение удаленного первого элемента массива. |
| **unshift @ARRAY, LIST** | Добавляет к началу массива **@ARRAY** список элементов **LIST**. Возвращает число элементов в новом массиве. |
| **pop @ARRAY** | Удаляет последний элемент массива **@ARRAY** и возвращает его значение. |
| **push @ARRAY, LIST** | Добавляет в конец массива **ARRAY** элементы из списка **LIST**. |
| **sort @ARRAY** | Сортирует элементы массива **@ARRAY** и возвращает отсортированный мас­сив. |
| **reverse @ARRAY** | Возвращает массив **@ARRAY** с элементами, расположенными в обратном порядке. |
| **join EXPR, @ARRAY** | Объединяет в одну строку элементы массива **@ARRAY**, используя в качестве разделителя полей значение **EXPR**, и возвращает эту строку. |
| **split /EXPR/, $STRING** | Преобразует строку **$STRING** в список, причем разделителем служит символ **EXPR** строки. Возвращает список значений. Если задан пустой символ-разделитель (**//**), строка будет преобразована в список посимвольно. |

***2.7 Ассоциативные массивы Perl***

Ассоциативный массив или **хэш-массив** – это неупорядоченное множество пар скалярных величин «ключ-значение».

Как и массивы, хэши представляют собой коллекцию скаляров. Разница между ними состоит в том, что доступ к элемен­там хэша осуществляется по имени, а не с помощью числовых индексов, как в масси­вах. Элементы хэша состоят из двух частей – *ключа* и *значения.* Ключ идентифициру­ет элемент хэша, а значение содержит данные, связанные с этим ключом. Такая взаи­мосвязь описывается термином *пара ключ-значение.*

Примерами хэш-подобных структур данных являются: ин­вентарные книги, медицинские карточки, телефонные счета, дисковая файловая сис­тема, музыкальные коллекции на компакт-дисках, библиотечные каталоги и многое другое.

Поскольку хэш-массив представ­ляет собой неупорядоченное, а, значит, ненумерованное множество элемен­тов, то обращение к отдельному элементу осуществляется не через его ин­декс, которого нет, а через ключ, ассоциированный с этим элементом. Имя переменной, обозначающей хэш-массив, содержит в качестве префикса символ "**%**". Инициализацию хэш-массива можно выполнить, если задать список пар элементов "*ключ*", "*значение*", например:

**%phones = ("123-45-67", "Саша", "765-43-21", "Петя",**

**"543-21-76", "Ира");**

Для получения списка всех ключей или всех значений ассоциативного мас­сива можно использовать функции **keys** и **values**.

Для доступа к отдельному элементу хэш-массива необходимо задать имя элемента и значение ключа, например:

**$search\_name = %phones{"123-45-67"};**

Часто нужно извлечь отдельный элемент хэш-массива не по ключу, а по его значению. Для этого существует специальный метод доступа, называемый *инверсией хэша.* Он заключается в том, что ключи и значения меняются местами, т.е. все ключи становятся значениями, а все значения становятся ключами, например:

**%ByNames = reverse %phones;**

Для проверки существования ключа в Perl есть специальная функция **exists**. Функция **exists** прове­ряет наличие указанного ключа в хэш-массиве и возвращает либо истинное значение (если ключ существует), либо ложное (в противном случае).

Другая операция – удаление ключей из хэш-массива. Для удаления одного ключа можно воспользоваться функцией **delete**.

***2.8 Функции Perl***

Практически все языки программирования содержат *функции.* Функция – это фрагмент кода, вызываемый по имени и возвращающий некоторое значение. Также практически все языки программирования имеют в своем составе так называемые **встроенные функции**.

Язык Perl, как и другие языки, предоставляет возможность написания собственных функций. Определенные пользователем функции называются в Perl **подпрограммами***.* Как и встроенные функции, они вызываются по имени и возвращают значение.

В Perl также реализована концепция *области видимости.* Область видимости определяет набор переменных, доступных программе в определенный момент времени. Благодаря этой концепции можно создавать функции, полностью независимые от остальной части программы. Корректно написанные функции могут быть повторно использованы и в других программах.

***2.8.1 Создание и вызов подпрограмм***

Для создания пользовательских подпрограмм в Perl используется следующий синтаксис:

**sub** *имя-подпрограммы*

**{**

*оператор-1***;**

....................

*оператор-N***;**

**}**

При вызове подпрограммы запоминается место, в котором произошел вызов, выпол­няет операторы подпрограммы и затем возвращается в основную программу к месту вызова. Подпрограммы Perl могут быть вызваны в любом месте основной программы или же из других подпрограмм.

***2.8.2 Возврат значений из подпрограмм***

Подпрограмма, как и функции, операторы и выражения Perl, имеет значение. Оно называется **возвращаемым значением***.* Возвращаемое значение подпрограммы – это значение последнего вычисленного в подпрограмме выражения или значение, явно возвращаемое оператором **return**.

Подпрограммы могут возвращать не только скаляры, но и массивы или хэш-массивы.

***2.8.3 Аргументы функций***

Подпрограмма, Значения, передаваемые функции и изменяющие ее поведение, называются **аргументами**. Для передачи функции аргументов можно использовать следующий синтаксис:

*имя-функции***(***арг1***,** *арг2***,** *арг3***);**

*имя-функции* *арг1***,** *арг2***,** *арг3***;**

**&***имя-функции***(***арг1***,** *арг2***,** *арг3***);**

Вторая форма (без скобок) может быть использована, только если интерпретатор Perl уже встречал определение этой функции.

В подпрограмме доступ к аргументам осуществляется посредством специальной переменной **@\_**.

Для доступа к индивидуальным аргументам можно использовать индексы массива **@\_**, как и в случае любого другого массива.

Переменная **@\_** содержит реальный список исходных аргументов подпрограммы. Ее изменение или изменение любого ее элемента приводит к модификации соответствующих переменных, поэтому функция не должна изменять значения передаваемых ей аргументов, если только это не делается специально.

Аргументы подпрограмм не обязательно должны быть скалярами. Можно передавать в подпрограмму массив или хэш-массив, однако, учитывая правила присвоения значений элементам массива, нельзя передавать в качестве аргументов два массива, поскольку все значения будут присвоены первому массиву, а второй массив останется пустым. Поэтому функции можно передавать только один массив, причем, если передаются и массив и скаляры, то надо поместить массив или хэш-массив последним в списке аргументов.

***2.8.4 Область видимости***

Язык Perl позволяет использовать одни и те же имена для обозначения различных пере­менных в большой программе. По умолчанию, переменные Perl видимы в основной программе и в подпрограммах. Это так называемые **глобальные переменные***.*

Допустим, вам нужно создать переменную, относящуюся только к данной функции. Для этого следует воспользоваться оператором **my**.

Внутри подпрограммы переменная, объявленная с помощью оператора **my** является **личной переменной***.* Другие подпрограммы или функции программы не имеют доступа к этой переменной. Та часть программы, в которой можно использовать данную переменную, называ­ется **областью видимости** переменной.

Личные переменные после окончания подпрограммы аннулируются, т.е. занятая ими память освобождается.

Можно объявить переменные даже с меньшей областью видимости, чем целая подпрограмма. Для этого следует поместить оператор **my** в блок. Это может быть либо основной блок подпрограммы, либо какой-нибудь другой блок.

В Perl подпрограмма может вызывать саму себя, т.е. допустимы **рекур­сивные подпрограммы***.*

***2.9 Работа со строками в Perl***

В Perl входит несколько функций работы со скалярами как строками.

Если необходимо определить, содержится ли указанная строка внутри другого скаляра, можно использовать функцию **index**. Ее синтаксис выглядит следующим образом:

**index** *строка*, *подстрока*

**index** *строка***,** *подстрока***,** *начальная-позиция*

Функция **index** начинает просмотр *строки* слева и ищет в ней *подстроку.* Во второй версии функции в параметре *начальная-позиция* задается позиция в строке, с которой начинается поиск *подстроки* (в первой версии поиск начинается с начала строки). Функция возвращает позицию, в которой найдена *подстрока* (позиции в строке нумеруются, начиная с нуля). Если *подстрока* не найдена, функция **index** возвращает значение **-1**.

Функция **rindex** работает точно так же, как **index**, за исключением того, что поиск начинается с крайнего правого символа строки и проводится в левую сторону. Синтаксис этой функции выглядит следующим образом.

**rindex** *строка, подстрока*

**rindex** *строка, подстрока***,** *начальная-позиция*

Функция **substr** возвращает подстроку указанной строки и имеет следующий синтаксис:

**substr** *строка***,** *смещение*

**substr** *строка***,** *смещение***,** *длина*

В первой версии функция возвращает подстроку, начинающуюся с заданного индекса – *смещения* в строке и до конца строки, во второй версии возвращается подстрока указанной *длины* (если в результате происходит выход за пределы строки, то просто берется подстрока до кон­ца строки). Если задано отрицательное значение параметра *смещение,* то функция **substr** начинает отсчет справа.

Если задано отрицательное значение параметра *длина,* то функция **substr** возвращает подстроку от начальной позиции и до конца строки, за исключением последних символов, количество которых определяется параметром *длина*.

Функцию **substr** можно использовать также в левой части оператора присваивания. Такая конструкция позволяет указать, какие символы в скаляре должны быть за­менены. В этом случае в качестве первого аргумента функции **substr** следует задать скалярную переменную, а не строку символов, которой будет присваиваться значение.

***2.10 Регулярные выражения в Perl***

***2.10.1 Синтаксис регулярных выражений***

Регулярное выражение представляет собой текстовый шаблон, по которому ведется поиск нужных символов в строке.

Шаблон заключается в ограничители, в качестве которых в Perl может быть использован любой символ, например, в выражении **/abc/** последовательность символов "**abc**" является шаблоном, а символ "**/**" ограничителем. Обычно в качестве ограничителя используется символ "**/**", за исключением тех случаев, когда он встречается в шаблоне. Поэтому в дальнейшем в тексте для краткости в качестве ограничителя будет применяться символ "**/**", хотя можно было использовать и другой символ, например символ "**:**" или "**\***". Если символ "**/**" встречается в шаблоне, перед ним необходимо поставить обратный слеш.

В качестве ограничителей могут использоваться также круглые ("**(**" и "**)**"), квадратные ("**[**" и "**]**"), фигурные ("**{**" и "**}**") и угловые ("**<**" и "**>**") скобки, например, **{abc}**.

Символ или символы перед начальным ограничителем определяют тип оператора, для регулярного выражения. Для некоторых типов операторов, наряду с символьным обозначением оператора, используются так называемые **пользовательские** (customary) обозначения операторов.

В Perl определены следующие типы операторов для регулярных выражений:

* **q**/*строка*/ – строковый литерал, в котором не производятся никакие замены;
* **qq**/*строка*/ – строковый литерал, в котором могут производиться замены;
* **qw**/*список*/ – строковый литерал, содержащий элементы списка, разделенные пробелами;
* **qx**/*строка*/ – строковый литерал, содержащий команду;
* **m**/*шаблон*/ – поиск соответствий;
* **s**/*шаблон*/*значение-замены*/ – поиск и замена символов по заданному шаблону поиска и последовательности замены;
* **qr**/*строка*/ – возвращает строку как шаблон;
* **tr**/*список-поиска*/*список-замены*/ – поиск и замена символов из списка шаблонов поиска на соответствующие элементы списка замены (вместо имени **tr** может использоваться его синоним **y**).

**Оператор q/строка/** представляет собой строковый литерал, в котором не производятся никакие замены, за исключение специальных символов, например, "**\n**". Пользовательское обозначение этого оператора – заключение строки в одиночные апострофы.

Пользовательским обозначением **оператора qq/строка/** является заключением строки в двойные апострофы.

**Оператор qw/список/**, возвращает список.

При выполнении **оператора qx/строка/** в строке производится поиск и замена идентификаторов, а затем полученная строка выполняется как команда. Пользовательским обозначением этого оператора служит заключение строки в обратные апострофы (символ "**`**").

**Оператор m/шаблон/** выполняет поиск соответствий по заданному шаблону. Ее пользовательским обозначением является заключение шаблона в ограничители "**/**", т.е. если в качестве ограничителей используются символы "**/**", указатель типа оператора – символ **m** можно опустить, т.е. записи **m/abc/** и просто **/abc/** являются эквивалентными.

Строка, в которой выполняется поиск, должна быть предварительно присвоена переменной **$\_**.

В скалярном контексте оператор **m** возвращает значение **1** (или истина), если указанная последовательность есть в строке, и пустая строка (или ложь) в противном случае. В контексте списка возвращается список из одного единственного элемента, значение которого равно **1**, если последовательность есть в строке, или пустой список – в противном случае.

Для того, чтобы выполнять поиск в произвольной строке, в Perl введены **операторы связывания** "**=~**" и "**!~**".

Оператор *идентификатор-переменной* **=~ m/***шаблон***/** связывает переменную со строкой поиска, т.е. заданный поиск будет выполняться не в строке **$\_**, а в указанной строке. Этот оператор возвращает значение истина, если заданная последовательность была найдена в строке и ложь – в противном случае.

Оператор "**!~**" отличается от оператора "**=~**" тем, что он возвращает значение ложь, если строка поиска найдена в указанной строке, и значение истина – в противном случае.

**Оператор s/шаблон/значение-замены/** осуществляет поиск и замену в строке. Как и для оператора поиска, строка, в которой выполняется поиск, должна быть предварительно присвоена переменной **$\_**, либо строку для поиска или замены можно определить с помощью одной из операций связывания. Оператор возвращает количество замен, сделанных в строке или пустую строку, если не было сделано ни одной замены.

**Оператор qr/строка/** возвращает значение строки заданной в качестве параметра как шаблон в регулярном выражении. Это значение затем можно использовать как шаблон в операторах **m** и **s**.

**Оператор транслитерации tr/***список-поиска***/***список-замены***/** выполняет поиск в строке элементов, указанных в первом списке, и заменяет их на соответствующие элементы из второго списка. По умолчанию поиск и замена выполняются в строке, находящейся в переменной **$\_**, либо переменная для поиска и замены определяется с помощью операции связывания.

В *списке-поиска* и в *списке-замены* можно указывать также диапазон символов. Если количество символов в *списке-замены* меньше, чем в *списке-поиска*, то все лишние символы в *списке-поиска* заменяются на последний символ в *списке-замены*.

Если в *списке-поиска* и/или в *списке-замены* необходимо указать переменные, то необходимо вычислить выражение **tr** с использованием встроенной функции **eval**.

***2.10.2 Вспомогательные переменные и опции в регулярных выражениях***

Помимо переменной **$\_**, при работе с регулярными выражениями можно использовать следующие **вспомогательные переменные**:

**$&** – содержит найденное значение шаблона;

**$`** – содержит подстроку исходной строки перед найденным шаблоном;

**$'** – содержит подстроку исходной строки после найденного шаблона.

После последнего ограничителя символа для некоторых типов регулярных выражений можно задать одну или несколько односимвольных **опций**, уточняющих выполняемую операцию.

Опции, допустимые в Perl, приведены в таблице:

| Опция | Действие | В каких типах регулярных выражений используется |
| --- | --- | --- |
| **c** | Не обнуляет позицию поиска при не найденном соответствии шаблона и символов строки, если включена опция **g**. | **m** |
| **c** | Для поиска будет использован не сам список, а его дополнение | **tr** |
| **d** | Удаление найденных, но не замененных символов | **tr** |
| **e** | Вычисление значения замены как выражения. | **s** |
| **g** | Глобальный поиск соответствий, т.е. нахождение всех соответствий. | **m**, **s** |
| **i** | Поиск соответствий без учета регистра (для букв). | **m**, **qr**, **s** |
| **m** | Строка рассматривается как последовательность строк, т.е. учитывается символ "**\n**" как символ конца строки. | **m**, **qr**, **s** |
| **o** | Компиляция шаблона только один раз (по умолчанию, поскольку шаблон может содержать переменные, каждый раз перед выполнением очередной операции, производится компиляция шаблона). | **m**, **qr**, **s** |
| **s** | Строка рассматривается как последовательность символов (символы "**\n**" игнорируются как символы конца строки). | **m**, **qr**, **s** |
| **s** | Если идущие друг за другом символы строки преобразуются к одному и тому же символу, в преобразованной строке сохраняется только один символ. | **tr** |
| **x** | Использование расширенных регулярных выражений. | **m**, **qr**, **s** |
| **U** | Преобразование в соответствии с кодировкой UTF-8. | **tr** |
| **C** | Преобразование в 8-битовый символ. | **tr** |

***2.10.3 Метасимволы***

Внутри шаблона большинство символов соответствуют сами себе, однако следующие символы при их использовании внутри шаблона имеют специальные значения:

**^ $ ( ) \ | @ [ { ? . + \***

Эти символы называются **метасимволами** и именно их использование обеспечивает всю мощь и гибкость регулярных выражений.

Если метасимвол необходимо использовать в шаблоне как обычный символ (например, для поиска), необходимо перед ним поставить обратный слеш – "\".

Если в шаблоне необходимо использовать символ "**\**", то он обозначается символами "**\\**".

Если в шаблоне необходимо указать символы перевода строки, возврата каретки, табулятора и новой страницы, используются специальные символы "**\n**", "**\r**", "**\t**" и "**\f**". Можно также задать любой символ ASCII в виде "**\x***код*", где код – шестнадцатеричный код символа (в диапазоне от "**\x00**" до "**\xff**").

В регулярных выражениях можно объединять несколько шаблонов, так чтобы найденная строка соответствовала хотя бы одному из них. Для решения по­добной проблемы служит **операция альтернации***,* которая в регулярных выражениях задается символом "**|**".

**Операция группировки элементов**, т.е. заключение группы элементов в круглые скобки, позволяет рассматривать данную группу элементов как один элемент.

Метасимвол точка "**.**" внутри регулярного выражения точка соот­ветствует любому одиночному символу, кроме символа перевода строки.

**Квантификаторы**– это метасимволы, используемые для указания количественных отношений между символами в шаблоне и в искомой строке. Квантификатор может быть поставлен после одиночного символа или после группы символов.

Простейшим квантификатором является метасимвол "**+**". Он означает, что идущий перед ним символ соответствует нескольким идущим подряд таким символам в строке поиска. Количество символов может быть любым (максимально большим в рамках соответствия шаблону), но должен присутствовать хотя бы один символ.

Действие метасимвола "**\***" похоже на действие метасимвола "**+**". Метасимвол "**\***" указывает, что идущий перед ним символ встречается нуль или более раз.

Метасимвол "**?**" указывает, что предшествующий ему символ должен встречаться либо один раз, либо не встречаться вообще.

Если необходимо указать точно количество повторений символа, можно воспользоваться конструкцией

*pat***{***n,m***}**

Здесь *n* – минимально допустимое количество повторений, *m* – максимально допустимое количество повторений, a *pat* – символ или группа символов, для которых указывается количество повторений. Один из параметров *n* или *m* можно опустить.

В регулярных выражениях часто используют сочетание метасимволов "**.\***". Ему соответствуют любые символы.

Для поиска в регулярных выражениях можно задавать также **классы символов**, заключенные в квадратные скобки. Во время поиска все символы в классе рассматриваются как один символ. Внутри класса можно задавать диапазон символов (когда такой диапазон имеет смысл), по­мещая дефис между границами диапазона.

Если первым символом класса является знак вставки "**^**", то значение выражения инвертируется. Другими словами, такому классу соответствует любой символ, *не* вхо­дящий в класс, например, шаблону

**/[^A-Z]/**

будут соответствовать любые символы, кроме символов верхнего регистра.

Так как в классах символы "**]**", "**^**" и "**-**" имеют специальное значение, для их использо­вания в классе существуют определенные правила. Литерал "**^**" не должен быть первым символом класса. Перед литералом "**]**" должен стоять символ обратной косой черты, например

**/[abc\]]/**.

Для помещения в класс символа "**-**" достаточно либо поставить его на первую позицию, либо поместить перед ним символ обратной косой черты.

В Perl имеются сокращения для некоторых часто используемых классов.

Наиболее употребительные сокращения приведены в таблице:

| Сокращение | Описание |
| --- | --- |
| **\w** | Символ, встречающийся в словах (латинский алфавит), то же, что и  **[a-zA-Z0-9]** . |
| **\W** | Символ, не встречающийся в словах, инверсия **\w**. |
| **\d** | Цифра, то же, что и **[0-9]** . |
| **\D** | Нецифровой символ. |
| **\s** | Символ пробела, то же, что и **[ \t\f\r\n]** . |
| **\S** | Символ, не являющийся пробельным символом. |
| **\b** | Соответствие границе слова. |
| **\B** | Соответствие не границе слова. |
| **\A** | Соответствие только в начале строки. |
| **\Z** | Соответствие только в конце строки или перед символом "**\n**". |
| **\z** | Соответствие только в конце строки. |

Последняя группа метасимволов – это **анкеры**. С их помощью можно указать, в каком месте строки (в начале или в конце) должно быть найдено соответствие с шаблоном.

Первый из этих анкеров – символ вставки "**^**". Этот символ, помещенный в начале регулярного выражения, говорит о том, что соответствие шаблону должно быть най­дено в начале строки.

Символ доллара "**$**", помещенный в конец регуляр­ного выражения, говорит о том, что соответствие шаблону должно быть найдено в конце строки.

Если в регулярных выражениях Perl используются скобки, части искомой строки, соответствующие фрагментам в скобках, запоминаются в специальных переменных **$1** (первый фрагмент в скобках), **$2** (второй фрагмент в скобках), **$3** (третий фрагмент в скобках) и т.д.

Следует отметить, что переменные **$1**, **$2** и т.д. модифицируются при каждом успешном поиске, независимо от использования в регулярном выражении скобок. Кроме того, значения этих переменных устанавливаются тогда и только тогда, когда строка полностью соответствует шаблону*.*

***2.10.4 Функции grep и man***

Поиск в массиве по шаблону — одна из наиболее распространенных операций Perl. Например, необходимо узнать, в каких элементах массива строк встречается определенная подстрока. Для операций такого рода можно использовать функцию **grep**, для которой определены следующие две формы вызова:

**grep** *выражение***,** *список*

**grep** *блок***,** *список*

Функция **grep** проходит каждый элемент *списка* и выполняет для него указанное *выражение* или *блок.* Внутри *выражения* или *блока* в качестве очередного элемента списка выступает переменная **$\_**. Если выражение истинно, данный элемент возвращается функцией **grep**.

Родственная **grep** функция **map** имеет аналогичный синтаксис:

**map** *выражение***,** *список*

**map** *блок***,** *список*

Функция **map** проходит каждый элемент *списка*, присваивает ему значение **$\_** и выполняет для переменной **$\_** указанное *выражение* или *блок.* Функция возвращает список, состоящий из элементов, являющихся результатом выполнения *выражения* или *блока* для соответствующих элементов исходного списка.

***2.11 Работа с файлами в Perl***

Практические все операции по обработке данных на сервере выполняются с использованием файлов.

***2.11.1 Открытие и закрытие файлов***

Каждый раз, когда необходимо получить доступ к файлу на диске, необходимо создать новый дескриптор и открыть его. Для открытия дескрипторов, используется функция **open**:

**open** *дескриптор-файла, режим-и-путь*

Первый аргумент функции **open** определяет дескриптор файла, второй аргумент указывает режим доступа и путь к файлу.

Возможные режимы доступа приведены в таблице:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Режим | Чтение | Запись | Добавление | Создается при необходимости | Старые данные теряются |
| **<** или не указан | Да | Нет | Нет | Нет | Нет |
| **>** | Нет | Да | Нет | Да | Да |
| **>>** | Нет | Да | Да | Да | Нет |
| **+<** | Да | Да | Нет | Нет | Нет |
| **+>** | Да | Да | Нет | Да | Да |
| **+>>** | Да | Да | Да | Да | Нет |

Путь указывает, какой файл необходимо открыть. При указании пути в Windows предпочтительнее указывать в качестве разделителя элементов пути не символ "**\**", а символ "**/**", например, " **d:/work/myfile.txt**".

Если не указан полный путь, функция **open** попытается открыть файл в текущем каталоге. При успешном открытии файла функция **open** возвращает ненулевое значение (истина), при неудачном — возвращается **undef** (ложь), например:

Функция **die** останавливает выполнение программы и выводит сообщение об ошибке:

**Died at** *имя-сценария* **line** *номер-строки*

Здесь *имя-сценария* – название программы на Perl, *номер-строки* – номер строки, в которой встретилась функция **die**. Функции **die** и **open** часто используются вместе следующим образом:

**open(MYTEXT, "novel.txt") || die;**

Программа или открывает файл, или прекращает свое выполнение. Если **open** завершилась неудачно, возвратив ложное значение, вступает в действие логический оператор ИЛИ "**||**" (иногда используется другая форма записи этого оператора – **or**). В результате будет вычисляться аргумент, находящийся в правой части оператора (в данном случае – функция **die**). При удачном выполнении функ­ции **open** правая часть логического выражения не вычисляется.

В Perl предусмотрена специальная переменная **$!**, содержащая сообщение об ошибке, возникшей при выполнении последней системной операции (например, опе­рации дискового ввода-вывода). В числовом контексте конструкция **$!** возвращает номер ошибки. В строковом контексте переменная **$!** возвращает сообщение операционной системы об ошибке.

Иногда нужно вывести в программе предупреждающее сообщение и продолжить ее выполнение. Для генерации предупреждений служит функция **warn**, аналогичная **die**, за исключением того, что выполнение программы продолжается.

Для закрытия деск­риптора используется функция **close**:

**close** *дескриптор-файла*

Если попытаться открыть файл, указав в качестве параметров функции **open** один из уже открытых дескрипторов, то вначале этот дескриптор закрывается, а затем по­вторно открывается.

***2.11.2 Чтение данных из файла***

Для чтения информации из файла дос­таточно поместить его дескриптор в угловые скобки и присвоить это значение пере­менной.

Для чтения информации из файла удобно использовать цикл **while**. Если в цикле **while** вместо условного выражения используется угловой оператор, Perl автоматически присваивает введенную строку специальной переменной **$\_** и повторяет цикл, пока файл не закончится:

**while(<MYFILE>)**

**{**

**print $\_;**

**}**

При этом в операторе **while** выполняется присваивание введенной строки переменной **$\_** и проверка признака достижения *конца файла.* Такое поведение реализовано только для цикла **while** и лишь тогда, когда условное выражение состоит из углового оператора.

Следует отметить, что в прочитанных из файла данных, кроме самого текста, содер­жатся также символы конца строки (так же как при вводе с клавиатуры). Поэтому, в случае, если нужен только текст, необходимо использовать функцию **chomp**, позволяющую удалить символы конца строки.

В контексте списка угловой оператор читает файл целиком и присваивает его списку. Каждая строка файла присваивается соответствующему элементу списка или массива.

***2.11.3 Запись данных в файл***

Для записи данных в файл сначала нужно открыть сам файл для записи. Наиболее часто используются открытие файла со следующими режимами:

**open(***дескриптор***, ">***путь***")**

**open(***дескриптор***, ">>***путь***")**

Символ "**>**" перед путем говорит Perl, что в файл должны быть за­писаны новые данные. При этом уже имеющиеся данные в файле уничтожаются, и ука­занный дескриптор открывается для записи. Во втором форме символы "**>>**" также указывают, что файл открывается для записи, но если файл уже существует, новые данные дописываются в конец файла.

Функция **close** сообщает операционной системе, что запись в файл завершена, и данные должны быть помещены в место своего постоянного хране­ния на диске.

После того как файл открыт для записи, вывод в него данных выполняется с помощью функции **print**, которая уже использовалась для отображения данных на экране дисплея. Если перед списком в этой функции задать дескриптор файла, то вызов функции примет следующий вид:

**print** *дескриптор***,** *список*

Здесь параметр *дескриптор —* это дескриптор файла, открытого для записи, а *список —* список элементов, которые нужно вывести в файл.

***2.12. Прямой доступ к интерфейсу CGI в Perl***

В зависимости от метода данные формы передаются в CGI-программу или через стандартный ввод (**POST**), или через переменную среды **QUERY\_STRING** (**GET**). Помимо этих данных CGI-программе доступна и другая информация, поступившая от клиента в заголовках запроса или предоставленная Web-сервером. Эта информация сохраняется в переменных среды операционных систем UNIX или Windows. В таблице перечислены переменные среды, обычно используемые в CGI:

| Переменная среды | Описание |
| --- | --- |
| **GATEWAY\_INTERFACE** | Версия CGI, которую использует сервер. |
| **SERVER\_NAME** | Доменное имя или IP-адрес сервера. |
| **SERVER\_SOFTWARE** | Имя и версия программы-сервера, отвечающей на за­прос клиента (например, Apache 1.3). |
| **SERVER\_PROTOCOL** | Имя и версия информационного протокола, который был использован для запроса (например, HTTP 1.0). |
| **SERVER\_PORT** | Номер порта компьютера, на котором работает сервер (по умолчанию 80). |
| **REQUEST\_METHOD** | Метод, использованный для выдачи запроса (**GET**, **POST**). |
| **PATH\_INFO** | Дополнительная информация о пути. |
| **PATH\_TRANSLATED** | Та же информация, что и в переменной **PATH\_INFO** с префиксом, задающим путь к корневому каталогу дерева Web-документов. |
| **SCRIPT\_NAME** | Относительное маршрутное имя CGI-сценария (напри­мер, **/cgi-bin/program.pl**). |
| **DOCUMENT\_ROOT** | Корневой каталог дерева Web-документов. |
| **QUERY\_STRING** | Строка запроса — информация, переданная в составе URL запроса после символа "**?**". |
| **REMOTE\_HOST** | Имя удаленной машины, с которой сделан запрос. |
| **REMOTE\_ADDR** | IP-адрес удаленной машины, с которой сделан запрос. |
| **REMOTE\_USER** | Идентификационное имя пользователя, посылающего запрос |
| **CONTENT\_TYPE** | Медиа-тип данных запроса, например, "text/html". |
| **CONTENT\_LENGTH** | Количество байт в теле запроса, переданных в CGI-программу через стандартный ввод. |
| **HTTP\_HOST** | Имя компьютера, на котором работает сервер |
| **HTTP\_FROM** | Адрес электронной почты пользователя, направившего запрос |
| **HTTP\_ACCEPT** | Список медиа-типов, которые может принимать клиент |
| **HTTP\_USER\_AGENT** | Web-браузер, которым клиент пользуется для выдачи запроса |
| **HTTP\_REFERER** | URL-адрес документа, на который клиент указывал пе­ред обращением к CGI-программе |
| **DATE\_GMT** | Текущая дата и время по Гринвичу. |
| **DATE\_LOCAL** | Текущие местные дата и время. |
| **DOCUMENT\_NAME** | Имя файла (но не каталога) документа, запрошенного пользователем. |
| **DOCUMENT\_URI** | Путь к документу, запрошенному пользователем и закодированным для URL. |
| **LAST\_MODIFIED** | Дата и время последнего изменения документа, запрошенного пользователем. |

CGI-программа на языке Perl имеет доступ к переменным среды через автоматически создаваемый интерпретатором хэш-массив **%ENV**, к элементам которого можно обратиться по ключу, совпадающему с именем переменной среды.

Данные формы поступают в CGI-программу в закодированном виде, поэто­му в качестве первого шага обработки CGI-сценарий должен выполнить де­кодирование полученной информации. При пересылке данных методом **GET** данные формы, как видно из приведенного выше примера, присваиваются переменной среды **QUERY\_STRING**, при пере­даче методом **POST** — передаются в программу через стандартный ввод и то­же могут быть присвоены некоторой внутренней переменной. Таким обра­зом, декодирование данных сводится к следующей последовательности ма­нипуляций со строкой:

* замена каждой группы "**%hh**", состоящей из шестнадцатеричного ASCII-кода "**hh**" с префиксом "**%**", на соответствующий ASCII-символ;
* замена символов "**+**" пробелами;
* выделение отдельных пар *имя***=***значение*, разделенных ограничителем "**&**";
* выделение из каждой пары *имя***=***значение* имени и значения соответст­вующего поля формы.

Вывод CGI-программы в интерфейс CGI выполняется с помощью функции **print**, которая направляет данные в стандартный выводной поток, причем первой должна быть выведена строка заголовка протокола HTTP (поле "**Content-type**"), как в следующем примере:

**#!/usr/bin/perl**

**# Создание заголовка HTTP**

**print "Content-type:text/html\n\n";**

**# Создание заголовка Web-страницы**

**print "<HTML><HEAD><TITLE>HelloWorld Page",**

**"</TITLE></HEAD><BODY>";**

**# Создание заголовка первого уровня**

**print "<H1>Hello, world</H1>";**

**# Создание окончания Web-страницы**

**print "</BODY><HTML>";**

С использованием конструкции "here document" ("документ здесь") предыдущий пример можно записать следующим образом:

**print<<TEST\_WEB\_PAGE;**

**Content-type:text/html\n\n**

**<HTML><HEAD><TITLE>HelloWorld Page</TITLE></HEAD><BODY>**

**<H1>Hello, world</H1>**

**</BODY><HTML>**

**TEST\_WEB\_PAGE**

***2.13. Использование модуля CGI в Perl***

Для написания CGI-программ для доступа к интерфейсу CGI можно использовать модуль CGI, входящий в состав стандартной поставки Perl. Этот модуль включается в программу с помощью оператора **use**:

**use CGI;**

***2.13.1. Стили программирования CGI-программ***

При создании CGI-программ с помощью модуля CGI можно использовать один из двух стилей программирования: объектно-ориентированный и стиль, ориентированный на использование функций.

При использовании объектно-ориентированного стиля создаются один или несколько объектов CGI, а затем для создания элементов страницы используются объектные методы применительно к созданным объектам. Каждый объект CGI формируется со списком именованных параметров, передаваемых Web-сервером CGI-программе. Можно изменять объекты, сохранять их в файле или базе данных, а затем восстанавливать их из файла или базы данных.

Например, простая Web-страница, содержащая приветствие "**Hello, world**" может быть создана с помощью объектно-ориентированного стиля с помощью следующей последовательности операторов:

**#!/usr/bin/perl -w**

**use CGI; # Загрузка модуля CGI**

**$q = new CGI; # Создание нового объекта CGI**

**print $q->header, # Создание заголовка HTTP**

**# Создание заголовка Web-страницы**

**$q->start\_html('HelloWorld Page'),**

**# Создание заголовка первого уровня**

**$q->h1('Hello, world'),**

**$q->end\_html; # Создание окончания Web-страницы**

При использовании стиля, ориентированного на функции, для получения данных из форм, формирования Web-страницы и других действий вызываются функции модуля CGI. Так, та же страница, созданная с помощью стиля, ориентированного на функции, выглядит следующим образом:

**#!/usr/bin/perl**

**# Загрузка стандартных функций CGI**

**use CGI qw/:standard/;**

**print header, # Создание заголовка HTTP**

**# Создание заголовка Web-страницы**

**start\_html('HelloWorld Page'),**

**# Создание заголовка первого уровня**

**h1('Hello, world'),**

**end\_html; # Создание окончания Web-страницы**

По своим функциональным возможностям эти стили не отличаются друг от друга, поэтому в дальнейшем будет использоваться стиль, ориентированный на функции, как более простой для использования.

При использовании стиля, ориентированного на функции, необходимо при подключении модуля CGI определить функции или наборы функций, которые будут импортированы из модуля в программу с помощью следующего синтаксиса:

**use CGI** *список-функций***;**

Однако более часто используются наборы функции, импортируемые по их групповому имени. Перед именем набора ставится префикс "**:**". Наиболее часто используются набор **:standard**, в который входят функции обработки форм и дескрипторов HTML, а также функции, связанные с интерфейсом CGI и набор **:all**, в который входят все функции модуля CGI.

***2.13.2. Функции модуля CGI***

В модуле CGI определено множество функций, которые можно разбить на следующие категории:

* функции управления Web-браузером;
* функции для обработки параметров;
* информационные функции;
* функции для записи дескрипторов.

Функция управления **header** информирует Web-браузер о том, данные какого типа он будет получать. Для этого используется параметр **Content-Type**. По умолчанию функция **header** посылает Web-браузеру заголовок **Content-Type** типа **text/html**. Чтобы указать другой тип получаемых данных, используется ключ **-type**, например:

**print header (-type => image/gif);**

В качестве параметра при вызове функции **header** может быть задано любое имя параметра заголовка HTTP, в котором впереди добавлен символ "**-**", а все символы дефиса "**-**" заменены символами подчеркивания "**\_**". Кроме того, при вызове можно указать несколько параметров, разделенных запятыми, например:

**print header(-Content\_length=>3002, -Server=>$server\_name);**

Для обработки параметров в модуле CGI используются функции **param** и **escape**. Функция **param** без параметров возвращает **имена параметров**, переданных CGI-программе (с помощью метода **GET** или **POST**). Если у функции **param** задан аргумент, она возвратит **значение параметра**, имя которого задано в качестве аргумента, например, **param(′username′)** возвратит значение **username**, введенное в форме или заданное в запросе. При этом будут, в случае необходимости, выполнены преобразования шестнадцатеричных представлений в символьные представления. Если параметр, указанный в аргументе функции **param**, не используется в форме, функция возвращает значение **undef**.

Если функция **param** декодирует переданные из формы или запроса данные, то функция **escape** выполняет обратную операцию – кодирование данных для передачи. Такая задача иногда возникает, когда в выводном потоке CGI-программы должен быть сформирован запрос. Следует отметить, что при использовании модуля CGI функция **escape** обычно недоступна для программы, и в случае необходимости ее приходится указывать в операторе **use** в явном виде.

В модуле CGI раз­работан целый набор функций, предназначенных для получения информации о типе Web-браузера и Web-сервера.

Однако следует отметить, что большинство описанных в этом разделе функций зависит от значений, которые предоставляются Web-сервером или посылаются Web-браузером в сеансе связи с сервером через протокол HTTP, т.е. Web-браузеры и Web-серверы не всегда сообщают точные сведения о себе.

Некоторые наиболее распространенные функции представлены в таблице:

| Функция | Описание |
| --- | --- |
| **referer** | Возвращает URL гиперссылки, с помощью которой будет осуществлен переход к этой странице. |
| **user\_agent** | Возвращает строку, идентифицирующую тип Web-браузера, который запросил страницу (например, браузеры Netscape, IE или Lynx). |
| **remote\_host** | Возвращает либо имя компьютера (host), либо IP-адрес системы, которая запросила страницу. Получаемое значение зависит от того, как сконфигурирован ваш Web-сервер и доступно ли для него имя удален­ного компьютера (hostname). |
| **script\_name** | Возвращает имя данной CGI-программы в виде части URL (например, **/cgi-bin/foo.pl**). |
| **server\_name** | Возвращает имя сервера, на котором работает CGI-программа |
| **virtual\_host** | Возвращает имя виртуального Web-сервера, который использовался для выполнения данной CGI-программы. Эта функция отличается от функции **server\_name**, поскольку зачастую один сервер может управлять несколькими Web-узлами. Функция **virtual\_host** возвращает имя конкретного узла, который был затребован. |

И, наконец, последнюю, самую большую группу функций составляют функции, формирующие дескрипторы языка HTML. Как правило, их имена совпадают с наименованиями соответствующих дескрипторов HTML, например **h1**, или эти функции формируют начальные и конечные дескрипторы, например **start\_table** и **end\_table**.

При программировании с использованием модуля CGI можно дать следующие рекомендации:

1. Для управления программой, работы с параметрами и получения информации о Web-браузере и Web-сервере рекомендуется использовать функции модуля CGI.
2. Программирование дескрипторов HTML, вставок сценариев, листов стилей и апплетов выполнять вручную, без использования соответствующих функций модуля CGI.

***2.13.3. Использование файлов DBM***

При программировании на Perl использование DBM-файлов является самым простым способом хранения структурированных данных. Файлы DBM, обычно используемые в UNIX, обладают одним замечательным свойством – в программах на Perl их можно напрямую связать с хэш-массивом. При этом чтение и запись DBM-файлов сводится к простым операциям с хэш-массивами, которые рассматривались в 2.10.

Хотя DBM-файлы можно использовать и при написании программ обработки данных в локальном режиме, они наиболее подходят при создании несложных баз данных на Web-сервере с доступом к ним с помощью CGI-программ, написанных на Perl.

Чтобы связать хэш-массив с DBM-файлом, в Perl используется функция **dbmopen**, синтак­сис которой выглядит следующим образом:

**dbmopen (***хэш***,** *имя-файла***,** *права-доступа***)**

В результате выполнения этой функции указанный хэш-массив подключается к файлу DBM. Параметр *имя-файла* на самом деле определяет два файла на жестком диске: *имя-файла***.pag** и *имя-файла***.dir**. Они используются Perl для хранения данных в иерархическом виде и быстрого доступа к ним. Эти файлы не являются текстовыми, поэтому их нельзя редактировать с помощью обычного текстового редактора.

Третий параметр функции **dbmopen** определяет права доступа, которые назначаются двум файлам DBM при их создании (задаются в виде восьмеричного числа). Например, значение кода прав доступа, равное **0666**, обеспечивает дос­туп по чтению и записи к файлам DBM для всех пользователей данного ком­пьютера; значение **0644** позволяет читать и записывать данные владельцу файла, в то время как для остальных пользователей обеспечивается только режим чтения. При работе в системе Windows данный параметр не играет особой роли, поскольку в ней не предусмотрены средства управления доступом, поэтому в Windows обычно используется значение **0666**.

Функция **dbmopen** возвращает истинное значение, если операция подключения хэш-массива к файлу DBM прошла успешно, например:

**dbmopen(%hash, "dbafile", 0644) ||**

**die "Ошибка при открытии DBM-файла: $!";**

После выполнения этого оператора устанавливается связь хэш-массива **%hash** с DBM-файлом **dbmfile**. Для хранения хэш-массива на диске Perl создает два файла: **dbmfile.pag** и **dbmfile.dir**. Если в последующих операторах значение элементов хэш-массива будет измене­но (как показано ниже на примере), Perl автоматически обновит соответствующие DBM-файлы:

**$hash{'155-1515'}="Иванов";**

**$hash{'442-1139'}="Петров";**

Обращение к элементам хэш-массива автоматически приводит к считыванию информации из DBM-файла, например:

**print $hash{'442-1139'};**

Чтобы разорвать связь хэш-массива **%hash** с DBM-файлом, используется функция **dbmclose**, например,

**dbmclose(%hash);**

После выполнения этой функции элементы хэш-массива "**155-1515**" и "**442-1139**" останутся в DBM-файле. В результате при следующем запуске программы и связывании хэш-массива **%hash** с DBM-файлами значение указанных элементов хэш-массива будет восстановлено.

С хэш-массивами, связанными с DBM-файлами, можно выполнять те же операции, что и с обычными хэш-массивами, в том числе использовать функции **keys**, **values** и **delete**. Чтобы очистить хэш-массив (и соответственно DBM-файл), надо присвоить ему пустой список, например:

**%hash=();**

Чтобы инициализировать хэш-массив и соответствующий ему DBM-файл, после выпол­нения функции **dbmopen** надо присвоить ей нужные значения в списке.

**3. Задания для выполнения**

Разработать perl-скрипт в соответствии с вариантом задания.

**4. Варианты индивидуальных заданий.**

1. В двух предложениях найти и распечатать совпадающие слова.
2. В предложении найти слово, состоящее из наибольшего количества гласных букв.
3. Распечатать слова, из которых состоит предложение, в порядке возрастания длины слова.
4. В предложении найти все однокоренные слова. Корень слова также задан с клавиатуры.
5. Удалить в предложении все повторные вхождения слов и распечатать получившееся предложение.
6. Найти в предложении слово, состоящее из наибольшего количества разных букв.
7. Выделить из предложения слова, содержащие две или более одинаковые буквы.
8. Напечатать слова, входящие в предложение, в алфавитном порядке.
9. Удалить из предложения все предлоги. Список предлогов задан во втором предложении.
10. Все буквы каждого слова в предложении записать в обратном порядке и распечатать получившееся предложение.
11. Во всех словах предложения удалить гласные буквы и распечатать по лучившееся предложение.
12. Дополнить каждое слово предложения пробелами так, чтобы его длина была равна длине самого длинного слова. Распечатать получившееся предложение.
13. Исходное предложение, состоящее из четырех слов, каждое из которых состоит из четырех букв, например, "Было утро, рано петь." распечатать в виде:

Бурп ытае, лрнт оооь.

Первое слово состоит из первых букв слов, второе - из вторых, и так далее.

1. С клавиатуры введены две строки. Одна строка - предложение на русском языке, а вторая (под ней) состоит из пробелов и знаков "минус" или "равно" (будем считать это подчеркиванием). Выделите (распечатайте) подчеркнутые слова. Например:

От перемены мест слагаемых сумма не меняется.

---- =======

1. Проверьте на совпадение два предложения. Количеством пробелов между словами пренебрегать. Знаки препинания - учитывать.
2. Удалить в предложении все слова, имеющие символы цифр и распечатать получившееся предложение.
3. Предложение состоит из записи символов цифр, например:

123434 53423 2344 6564.

Распечатать слова в порядке возрастания.

лАБОРАТОРНАЯ РАБОТА №4

Регулярные выражения в Perl

**1. Цель работы**

Получить навыки работы с регулярными выражениями на языке Perl.

**2. Общие сведения**

***2.1 Основные определения***

*Регулярные выражения в perl (regexp)* одна из самых мощных его возможностей. Они позволяют сопоставлять текст с указанным шаблоном, разбивать текст в массив по шаблону, производить замену текста по шаблону и многое другое. Также иногда регекспами называются операторы поиска и замены.

Оператор **q(text)** заменяет строку **text** на строку, заключенную в одинарные кавычки (например, если в **q(text)** поставить символ **q(text\n)**, то напечатает **text\n**, т.е. **\n** это два символа, подобно **print 'amam $file'** напечатает **amam $file**). В данном случае почти все специальные символы не будут интерпретироваться внутри **q()**, исключая **'\':**

**$some=q(Don't may be);**

Оператор **qq~text~;** (вместо значка **~** можно ставить например знак **|**) позволяет работать со строками и многострочными текстами. Используя этот оператор, можно выводить целые куски html-кода и писать в этом коде имена скалярных переменных.

Оператор **qw("text")** разбивает строку на массив слов:

**@mass=qw("я вышел погулять и увидел как через реку строят новый мост");**

**#хотя с настроенной локалью будет работать и**

**@mass=qw(я вышел погулять и увидел как через реку строят новый мост);**

**for(@mass){print $\_,"\n"}**

Оператор **qr/pattern/** ключи- **imosx** работает подобно регулярному выражению **s/.../.../:**

**$rex=qr/my.STRING/is;**

**s#$rex#foo#;**

**#тоже самое, что и**

**s/my.STRING/foo/is;**

Результат может использоваться подобно вызову подпрограммы (см. **perldoc perlop Regex quote like operator**):

**$re=qr/$pattern/;**

**$string=~/foo${re}bar/;**

**$string=~$re;**

**$string=~/$re/;**

Ключи **imosx** стандартные (см. ниже).

Оператор **qx/STRING/** работает как системная команда, подобно **$output = `cmd 2>$1`;**. Программа, иллюстрирующая использование данного оператора:

**#!/usr/bin/perl**

**qx[dbfdump --fs="\x18" --rs="\x19" pdffile.dbf >pdffile.txt];**

Файл **pdffile.dbf**содержит *memo-поля* (memo-поле содержит ссылку подобно функции **seek** на текст в файле с расширением **\*.fpt**). Он принимает разрешения FoxBASE4 и дампит файлы со встроенными memo-полями в текстовый вид. Таким образом получилается вытащить информацию из файла memo-типа \*.fpt.

Допустим, используя команду **$perl\_info = qx(ps $$);**, мы выводим информацию о текущем процессе запущенного скрипта (каждая запущенная программа в UNIX имеет свой собственный уникальный идентификатор, который содержится во встроенной переменной **$$** - достаточно уникальное число, можно использовать почти как счетчик случайных чисел). Если сказать **$shell\_info = qx'ps $$';** то выведет информацию о самом ps. Т.е. скобки осуществляют своеобразное экранирование от двойной кавычки.

В перл есть три основных оператора, работающих со строками:

**m/.../ - проверка совпадений (matching),**

**s/.../.../ - подстановка текста (substitution),**

**tr/.../ - замена текста (translation).**

Оператор **m/.../** анализирует входной текст и ищет в нем подстроку, совпадающую с указанным шаблоном (он задан регулярным выражением). Оператор **s/.../.../**выполняет подстановку одних текстовых фрагментов вместо других при помощи регулярных выражений. Оператор **tr/.../.../**заменяет выходной текст, но при этом он не использует регулярные выражения, осуществляя замену посимвольно.

Оператор **m/шаблон/** - поиск подстроки по определенному шаблону. Например

**print "$1 г.\n" while m!((\d){4})!g**

найдет и выведет все даты в переменной **$\_**. В шаблоне не важно, что будет его ограничителем. Например, при поиске гиперссылок, которые зачастую содержат символы **/**, разумнее пользоваться не **/**, а, например, **#** или **!** как символами ограничителями. В таком случае шаблон будет более прост для понимания другим программистам и немного короче. В perl оператор **m/.../** используется очень часто, и поэтому используется сокращение без начальной буквы **m**. Если начальная буква есть, то в качестве символов ограничителей можно использовать любой другой символ.

Для оператора **m/pattern/** есть 6 параметров (**gimsxo**):

**m/foo/g** говорит компилятору найти все **foo** в тексте, в то время как **m/foo/** найдет только первое вхождение подстроки **foo** в строке **$\_**. В строке **$\_** содержится обычный текст, как и в переменной **$text$,** **$\_** такая же переменная, только существует всегда и вводится, когда не определена специально другая по умолчанию.

Например, можно сказать

**for (@mass){print $\_,"\n"}**

или

**for $elem (@mass){print $elem,"\n"}.**

Эти две строчки делают одно и то же, но в первом случае запись короче, да и зачастую бывает удобно использовать переменную **$\_**, например, когда нужно выделить при помощи регулярного выражения определенные данные, пользуясь перебором массива (функция **map**):

**@res=map{/(\d\d\d\d)/} split /\s/, $texts;**

что эквививалентно коду

**push @res, $1 while m!((\d){4})!g; #(в данном случае $\_=$texts)**

или что эквивалентно конструкции

**foreach(split /\s/, $texts){**

**push @res, $1 if(/(\d\d\d\d)/g)**

**}**

Следующий параметр **m/foo/i**, говорит о том, что не нужно учитывать регистр при поиске по подстроке.

Параметр **m/foo/s** говорит о том, что строка, по которой производится поиск, состоит из одной строчки.

Например, нужно выцепить все url картинок из странички www.astronomynow.com, чтобы сделать локальное зеркало этой странички, и пользователи могли с интересом читать последние новости астрономии:

**#!/usr/bin/perl -wT**

**use LWP::Simple;**

**$page=get "http://www.astronomynow.com";**

**&getlink($page);**

**sub getlink{**

**local $\_=$\_[0];**

**push(@res, "http://$2")**

**while m{SRC\s\*=\s\*(["'])http://(.\*?)\1\s\*(.\*?)WIDTH="100"HEIGHT="100"(.\*?)>}igs**

**}**

В подпрограмме заводится при помощи функции **local** переменная, видимая только в области действия подпрограммы. Этой переменной присваивается значение переменной **$page**, в которой содержится текст выкачанной **Simple.pm** странички.

Можно сделать немного по-другому. Сохранить скачанную страничку в файл на диск и выполнить следующее:

**$/="\001";**

**open F, "<page.html"; $page=<F>; close F;**

**&getlink($page);**

**...**

Встроенная переменная **$/** содержит символ разделителя входных записей. Это может быть перевод каретки или, при **upload far'ом** на сервер файлов в не ASCI виде, она приобретают на конце строчки хитрый символ **^M**.

Если **$/** переопределить, то можно свободно пользоваться дескрипторами открытия файлов для просмотра многострочного текста (**m/pattern/s**). Например, когда открывается файл при помощи функции **open F**, **"<file.txt"; @mass=<F>**, то присваивая дескриптор F массиву в массиве появятся строчки, разделенные символом, содержащимся в **$/**.

Переопределив **$/** можно запросто написать:

**open F, "<file.txt"; $mass=<F>**

и в переменной **$mass** будет содержаться многострочный текст с точки зрения человека, но программа будет видеть этот текст как одну строку, и по тексту можно будет запросто пройтись поиском **m/pattern/igs** и выделить все необходимые подстроки.

Параметр **m/foo/o** говорит о том, что шаблон нужно компилировать только один раз. Если оператор используется в сочетании с операциями привязки **=~** и отрицание **!~**, то строкой, в которой ведется поиск, является переменная, стоящая слева от операции привязки. В противном случае поиск ведется в строке **$\_**.

Оператор **s!pattern!substring!** - поиск в строке по шаблону **pattern** и замена найденного текста на **substring**. Как и для оператора **m/.../**, косую черту можно не ставить, пригоден любой символ, который не находится в противоречии с заданным выражением. Не рекомендуется использовать в качестве ограничителей **?** и **'**.

**s!/usr/local/etc/!/some/where/else! - заменяет путь.**

**s(/usr/local/etc/)(/some/where/else)g - заменяет все встречающимеся пути до файла.**

параметры: **egimsxo e** - указывает, что **substring** нужно вычислить.

Например, нужно переделать все **escape** последовательности. Для этого вызывается соответствующая подпрограмма:

**$text =~ s/(&.\*?;)/&esc2char($1)/egs;**

т.е. из регулярного выражения происходит вызов подпрограммы.

**g** - заменить все одинаковые компоненты, а не один, как в отсутствии ключа g.

**i** - не учитывать регистр.

**m** - строка, в которой происходит поиск, состоит из множества строк.

**s** - строка, в которой происходит поиск, состоит из одной строки.

**x** - сложный шаблон, т.е. можно писать не в строчку, а для упрощения понимания разбивать шаблон на несколько строк, примеры об этом ниже.

**o** - компилировать шаблон один раз.

Допустим нужно сделать поисковик, который ходит по директориям на сервере, но некоторые директории типа **/cgi-bin/** и т.п. индексировать нельзя. Объявляем переменную, которая будет содержать регулярное выражение, в данном случае перечисление или **img** или **image** или **temp** или **tmp** или **cgi-bin**:

**$no\_dir = '(img|image|temp|tmp|cgi-bin)';**

Ключи регулярного выражения **m#$no\_dir$#io** говорят о том, что компилировать содержимое **$no\_dir** нужно только один раз (ключ **o**) и не учитывать регистр (ключ **i**).

Оператор **tr/выражение1/выражение2/**, ключи **cds.**

Смысл: замена **выражения1** на **выражение2**. Если указан ключ **с**, то это инверсия **выражения1**, т.е. *в выражение один не входят содержащиеся в нем символы*. Если указан ключ **d**, то значит *стереть замененные символы*. Если указан ключ **s**, то значит *заменить многочисленные повторяющиеся символы на одиночный символ*.

Оператор **y/выражение1/выражение2/(ключи cds)**, равносилен оператору **tr**.

Например, в поисковой системе нужно приводить запрос в нижний регистр, чтобы не зависеть от настроек локали:

**$CAP\_LETTERS = '\xC0-\xDF\xA8';**

**$LOW\_LETTERS = '\xE0-\xFF\xB8';**

**$code = '$html\_text =~ ';**

**$code .= "tr/A-Z$CAP\_LETTERS/a-z$LOW\_LETTERS/";**

**$down\_case = eval "sub{$code}";**

***2.1.1 Одиночные символы***

В регулярном выражении любой символ соответствует самому себе, если только он не является *метасимволом* со специальным значением (такими метасимволами являются **\, |, (, ), [,{, \*, +, ^, $, ? и .**). В следующем примере проверяется, не ввел ли пользователь команду "**quit**" (и если это так, то прекращаем работу программы):

**while(<>){**

**if(m/quit/){exit;}**

**}**

Правильнее проверить, что введенное пользователем слово "**quit**" не имеет соседних слов, изменяющих смысл предложения. (Например, программа выполнит заведомо неверное действие, если вместо "**quit**" пользователь введет команду "**Don't quit!**".) Это можно сделать с помощью метасимволов **^** и **$**. Чтобы сравнение было нечувствительно к разнице между прописными и заглавными буквами, используем модификатор **i**:

**while (<>)**

**{if (m/^quit$/i) {exit;} }**

Кроме обычных символов perl определяет *специальные символы*. Они вводятся с помощью обратной косой черты (escape-последовательности) и также могут встречаться в регулярном выражении:

* **\077**- восьмеричный символ;
* **\а** - символ BEL (звонок);
* **\с[** - управляющие символы (комбинация Ctrl + символ, в данном случае это управляющий символ ESC);
* **\d** - соответствует цифре;
* **\D** - соответствует любому символу, кроме цифры;
* **\е** - символ escape (ESC);
* **\Е** - конец действия команд \L, \U и \Q;
* **\f** - символ прогона страницы (FF);
* **\1** - следующая литера становится строчной (lowercase);
* **\L** - все последующие литеры становятся строчными вплоть до команды \Е;
* **\n** - символ новой строки (LF, NL);
* **\Q** - вплоть до команды \Е все последующие метасимволы становятся обычными символами;
* **\r** - символ перевода каретки (CR);
* **\s** - соответствует любому из "пробельных символов" (пробел, вертикальная или горизонтальная табуляция, символ новой строки и т. д.);
* **\S** - любой символ, кроме "пробельного";
* **\t** - символ горизонтальной табуляции (НТ, TAB);
* **\u** - следующая литера становится заглавной (uppercase);
* **\U** - все последующие литеры становятся заглавными вплоть до команды \E;
* **\v** - символ вертикальной табуляции (VT);
* **\w** - алфавитно-цифровой символ (любая буква, цифра или символ подчеркивания);
* **\W** - любой символ, кроме букв, цифр и символа подчеркивания;
* **\x1B** - шестнадцатеричный символ.

B at также можете "защитить" любой метасимвол, то есть заставить perl рассматривать его как обыкновенный символ, а не как команду, поставив перед метасимволом обратную косую черту **\**. Обратите внимание на символы типа **\w**, **\d** и **\s**, которые соответствуют не одному, а любому символу из некоторой группы. Также заметьте, что один такой символ, указанный в шаблоне, соответствует ровно одному символу проверяемой строки. Поэтому для задания шаблона, соответствующего, например, слову из букв, цифр и символов подчеркивания, надо использовать конструкцию **\w+**, как это сделано в следующем примере:

**$text = "Here is some text."**

**$text =~ s/\w+/There/;**

**print $text;**

**There is some text.**

***2.1.2 Классы символов***

Символы могут быть сгруппированы в *классы*. Указанный в шаблоне класс символов сопоставляется с любым из символов, входящим в этот класс. *Класс* - это совокупность символов, заключенных в квадратные скобки **[** и **]**. Можно указывать как отдельные символы, так и их диапазон (диапазон задается двумя крайними символами, соединенными тире). Например, следующий код производит поиск гласных:

**$text ="Here is the text.";**

**if ($text =" /[aeiou]/) {print "Vowels: we got 'em.\n";}**

**Vowels: we got 'em.**

Другой пример: с помощью шаблона **[A-Za-z]+** (метасимвол **+** означает утверждение: "один или более таких символов") ищется и заменяется первое слово:

**$text = "What is the subject.";**

**$text =" s/[A-Za-z]+/Perl/;**

**print $text;**

**Perl is the subject;**

Если требуется задать минус как символ, входящий в класс символов, перед ним надо поставить обратную косую черту **\-**. Если сразу после открывающей квадратной скобки стоит символ **^**, то смысл меняется на противоположный. А именно, этот класс сопоставляется любому символу, кроме перечисленных в квадратных скобках. В следующем примере производится замена фрагмента текста, составленного не из букв и не из пробелов:

**$text = "perl is the subject on page 493 of the book.";**

**$text =- s/[a-Za-z\s]+/500/;**

**print $text;**

**perl is the subject on page 500 of the book.**

***2.1.3 Альтернативные шаблоны***

Вы можете задать несколько альтернативных шаблонов, используя символ **|** как разделитель. Альтернативные шаблоны позволяют превратить процедуру поиска из однонаправленного процесса в разветвленный: если не подходит один шаблон, perl подставляет другой и повторяет сравнение, и так до тех пор, пока не иссякнут все возможные альтернативные комбинации. Например, следующий фрагмент проверяет, не ввел ли пользователь "**exit**", "**quit**" или "**stop**":

**while (<>){**

**if(m/exit|quit|stop/){exit;}**

**}**

Чтобы было ясно, где начинается и где заканчивается набор альтернативных шаблонов, их заключают в круглые скобки - иначе символы, расположенные справа и слева от группы шаблонов, могут смещаться с альтернативыми шаблонами.

В следующем примере метасимволы **^** и **$** обозначают начало и конец строки и отделяются от набора альтернативных шаблонов с помощью скобок:

**while (<>){**

**if(m/^(exit|quit|stop)$/){exit;}**

**}**

Альтернативные варианты перебираются слева направо. Как только найдена первая альтернатива, для которой выполняется совпадение с шаблоном, перебор прекращается. Участки шаблона, заключенные в круглые скобки, выполняют специальную роль при выполнении операций поиска и замены. Если символ **\** находится в квадратных скобках, он интерпретируется как обычный символ. Поэтому если вы используете конструкцию шаблона вида **[Tim|Tom|Tam],** то она будет эквивалентна классу символов **[Tioam|]**. Точно так же большинство других метасимволов и команд, специфичных для регулярных выражений - в частности, квантификаторы и мнимые символы, описанные в двух последующих разделах, - внутри квадратных скобок превращаются в обычные символы или escape-последовательности текстовых строк.

***2.1.4 Квантификаторы***

Квантификаторы указывают на то, что тот или иной шаблон в строке может повторяться определенное количество раз. Например, можно использовать квантификатор **+** для поиска мест неоднократного повторения подряд латинской буквы **е** и их замены на одиночную букву **е**:

**$text = "Hello from Peeeeeeeeeeeeeeerl.";**

**$text =~ s/e+/e/:**

**print $text;**

**Hello from perl.**

***2.1.5 Мнимые символы***

В perl имеются символы (метасимволы), которые соответствуют не какой-либо литере или литерам, а означают выполнение определенного условия (поэтому в английском языке их называют assertions, или *утверждениями*). Их можно рассматривать как мнимые символы нулевого размера, расположенные на границе между реальными символами в точке, соответствующей определенному условию:

* **^** - начало строки текста;
* **$** - конец строки или позиция перед символом начала новой строки, расположенного в конце;
* **\b** - граница слова;
* **\В** - отсутствие границы слова;
* **\А** - "истинное" начало строки;
* **\Z** - "истинный" конец строки или позиция перед символом начала новой строки, расположенного в "истинном" конце строки;
* **\z** - истинный конец строки;
* **\G** - граница, на которой остановился предыдущий глобальный поиск, выполняемый командой **m/.../g**,
* **(?= шаблон)** - после этой точки есть фрагмент текста, который соответствует указанному регулярному выражению;
* **(?! шаблон)** - после этой точки нет текста, который бы соответствовал указанному регулярному выражению;
* **(?<= шаблон)** - перед этой точкой есть фрагмент текста, соответствующий указанному регулярному выражению;
* **(?<! шаблон)** - перед этой точкой нет фрагмента текста, соответствующего указанному регулярному выражению.

Например, чтобы выполнить поиск и замену слова, используя метасимволы границы слов, необходимо воспользоваться следующим скриптом:

**$text = "Here is some text.";**

**$text = s~/\b([A-Za-z)+)\b/There/;**

**print $text;**

**There is some text.**

Perl считает границей слова точку, расположенную между **\w** и **\W**, независимо от того, в каком порядке следуют эти символы. В следующем примере выводится сообщение о том, что пользователь ввел слово "**yes**", при условии, что оно единственное, что ввел пользователь. Для этого шаблон включает мнимые символы начала и конца строки:

**while (<>) {**

**if (m/^yes$/) {**

**print "Thank you for being agreeable.\n";**

**}**

**}**

Приведенный выше пример требует комментария. Прежде всего, бросается в глаза наличие двух групп метасимволов для начала и конца строки. В большинстве случаев они означают одно и то же, так как обычно символы новой строки (то есть **\n**), встречающиеся внутри текстового выражения, не рассматриваются как вложенные строки. Однако если для команды **m/.../**или **s/.../.../**указан модификатор **m**, то текстовое выражение будет рассматриваться как многострочный текст, в котором границами строк выступают символы новой строки **\n**. В случае многострочного текста метасимвол **^** сопоставляется с позицией после любого символа новой строки, а не только с началом текстового выражения. Точно также метасимвол **$** - это позиция перед любым символом новой строки, расположенным внутри текстового выражения, а не обязательно конец текстового выражения или же позиция перед концевым символом **\n**. Однако метасимвол **\A** - начало текстового выражения, а метасимвол **\Z** - конец текстового выражения или позиция перед концевым символом **\n**, даже если в текстовом выражении имеются вложенные символы **\n** и при выполнении операции поиска или замены указан модификатор **m**. Метасимвол точка (**.**) соответствует любому символу, кроме символа новой строки **\n**. Независимо от того, задан ли модификатор **m**, она не будет сопоставляться ни c внутренними, ни с концевыми символами **\n**. Единственный способ заставить точку рассматривать **\n** как обычный символ - использовать модификатор **s**.

Отсюда понятна разница между метасимволами **\Z** и **\z**. Если в качестве текстового выражения используется результат чтения входного потока данных, то с большой вероятностью данное выражение заканчивается символом **\n**, за исключениeм того варианта, когда программа предусмотрительно "отщипнула" его с помощью функции **chop** или **chomp**. Метасимвол **\Z** игнорирует концевой символ **\n,** если он случайно остался на месте, рассматривая обе ситуации как "**конец строки**". В отличие от него метасимвол **\z** оказывается более пунктуальным и рассматривает концевой символ **\n** как неотъемлемую часть проверяемого текстового выражения, если только пользователь не позаботился об удалении этого символа.

Отдельно следует остановиться на метасимволе **\G**. Он может указываться в регулярном выражении только в том случае, если выполняется глобальный поиск (то есть если команда **m/.../** имеет модификатор **g**). Метасимвол **\G**, указанный в шаблоне, соответствует точке, на которой остановилась предыдущая операция поиска.

***2.1.6 Ссылки на найденный текст***

Иногда нужно сослаться на подстроку текста, для которой получено совпадение с некоторой частью шаблона. Например, при обработке файла, HTML может потребоваться выделять фрагменты текста, ограниченные открывающими и закрывающими метками HTML (например, **<А>** и **</А>**). В начале уже приводился пример, в котором выделялся текст, ограниченный метками HTML **<А>** и **<B>**. Следующий пример позволяет выделять текст, расположенный между любыми правильно закрытыми метками:

**$text = "<А>Here is an anct1or.</А>";**

**if($text=~m%<([A-Za-z]+)>[\w\s\.]+</\1>%i){**

**}**

Вместо косой черты в качестве ограничителя шаблона использован другой символ. Это позволяет использовать символ косой черты внутри шаблона без предшествующей ему обратной косой черты. Каждому фрагменту шаблона, заключенному в круглые скобки, соответствует определенная внутренняя переменная. Переменные пронумерованы, так что на них можно ссылаться внутри шаблона, поставив перед номером обратную косую черту (\1, \2, \3,...). На значения переменных можно ссылаться внутри шаблона, как на обычный текст, поэтому </\1> соответствует </\A>, если открывающей меткой служит <А>, и, если открывающей меткой служит. Эти же самые внутренние переменные можно использовать и вне шаблона, ссылаясь на них как на скаляры с именами $1, $2, $3,..., $n:

**$text = "I have 4 apples.";**

**if ($text =- /(\(\d+)/) {**

**print "Here Is the number of apples: $1.\n";**

**Here is the number of apples: 4.**

Каждой паре скобок внутри шаблона после завершения операции поиска будет соответствовать скалярная переменная с соответствующим номером. Это можно использовать при выделении нужных для последующей работы фрагментов анализируемой строки. В следующем примере мы изменяем порядок трех слов в текстовой строке с помощью команды **s/.../.../**:

**$text = "I see you.";**

**$text=-s/^(\w+) \*(\w+) \*(\w+)/$3 $2 $1/;**

**print $text;**

**you see I.**

Переменные, соответствующие фрагментам шаблона, нумеруются слева направо с учетом вложенности скобок. Например, после следующей операции поиска будут проинициализированы шесть переменных, соответствующих шести парам скобок:

**$text = "ABCDEFGH";**

**$text =- m/(\w(\w)(\w))((\w)(\w))/;**

**print "$1/$2/$3/$4/$5/$6/";**

**ABC/B/C/DE/D/E**

Кроме переменных, ссылающихся на найденный текст, можно использовать специальные переменные perl.

***2.2 Функции, использующие регулярные выражения***

Фактически, есть три функции, которые в качестве разделителя могут использовать регулярные выражение: **split**, **grep**, **map** и еще можно воспользоваться специальными операторами **...** и **..** и используемыми совместно с ними условиями **if**, **unless** и просто логическими операторами.

***2.2.1 Split***

Если необходимо разделить данные из **STDIN** по нужному разделителю, то можно воспользоваться локализацией **$/**:

**sub example\_local{**

**local $/ = undef;**

**@mass= split /pattern/, <>;**

**return 1;**

**}**

**print scalar(@mass);**

Можно разделять данные из файла и так:

**undef $/;**

**@res=split /pattern/, <F>;**

**что эквивалентно:**

**while (<F>) {push @asdf, split}**

После **split** можно ставить вместо запятой и стрелочку:

**@mass = split /(\d){4}/ => $file;**

В функции сплит можно воспользоваться максимальным квантификатором **\***, который в том числе и **0** символов, позволит разделить строку на символы, которых там нет (в силу того, что **\*** это **0** и более символов), т.е. посимвольно:

**@ruru = split /\001\*/ => "lalalalalala";**

#массив @ruru будет содержать элементы по одной букве.

Если строка состоит из нескольких строк, то можно поставить разделителем и символ начала новой строки:

**$str = "asdf\nghjk\nqwer\n";**

**@lines = split /^/ => $str;**

Вобщем, в split можно вставлять любой поиск по шаблону.

***2.2.2 Grep***

Функция **grep** также позволяет заполнять массив значениями. Например, нужно получить список расширений файлов в заданной директории:

**while(<$dir/\*.\*>){push @files, $\_} #читаем директорию**

**@test = grep { s|.\*/(.\*?)\.(.\*)|$2| } @files; #оставляем в директории только расширения файлов**

можно использовать признак четности для занесения в массив:

**@test1=qw(1 2 3 4 5 6 7 8 9);**

**@evens = grep($\_%2 == 1) @test1;**

Или более сложное регулярное выражение для вытаскивания всех e-mail адресов из текстовой странички:

**@mass=grep{s/(.\*) ([\w+\-\.]+\@[\w\-\.]+\.\w{2,3})(.\*)/$2/ig} split /\n/, $test;**

Здесь используется укороченная запись:

**@mass=grep {/pattern/} split /\n/, $test;**

которая эквивалента записи из двух строчек:

**@uuu=split /\n/, $test;**

**@mass=grep {/pattern/} @uuu;**

***2.2.3 Map***

Функция **map** похожа по своей работе на обычное условие **if**, допустим нужно разделить записи на блоки, разделенные четырьмя пробелами:

**@probel = map m!\s{4}!, split /\n/, $test;**

***2.2.3 Other***

Вывод строк из заданного интервала для данной строки:

**if(/pattern1/i .. /pattern2/i){...}**

**#истинность первого оператора включает конструкцию, а второго е выключает.**

**if($nomer1 .. $nomer2){...}**

... не возвратит истину, в отличии от .., если условия выполняются в одной строке.

**if(/pattern1/i ... /pattern2/i){...}**

**if($nomer1 ... $nomer2){...}**

для многострочного файла

**print -ne 'print if 3 .. 15' file.txt**

выведет строки файла с 3 по 15 строчку, та же самая операция но немного по другому:

**open F, "<file";**

**while(<F>){**

**print if(3 .. 15)**

**}**

или с какой-нибудь начальной и конечной разметкой. Например, есть вспомогательный файл шаблонов (просто различные виды html, в зависимости от действия пользователя) для разных определенны случаев, которые нужны исходя из контекста программы:

**open F, "<file";**

**while(<F>){**

**print if(/<!--begin welcome-->/i ... /<!--end welcome-->/i)**

**}**

Такая конструкция позволяет выводить куски многострочного html кода (для однострочного нужно ставить оператор **..**). Условия в таких операторах можно ставить и разнотипными

**$file=qr/2345/;**

**while(<F>){**

**print if(/^$/ .. 10); #увидим, что находится от пустой до 10-й строки**

**print if(/^\001/ .. /$file/); #выведет все, что после нуля и до того что задано qr**

**}**

Программа чтения почтовых адресов из mbox или sent-mail:

**while(<F>){**

**next unless /^From:?\s/i .. /^$/;**

**while (/([^<>(,;)\s]+\@[^<>(,;)\s]+)/)g{**

**print "$1\n" unless $test{$1}++;**

**}**

**}**

запускается ./regex.pl /root/mail/sent-mail и выводит каждый емейл по одному разу.

***2.3 Использование встроенных переменных***

* **$'** - подстрока, следующая за совпадением с шаблоном поиска, можно только читать;
* **$&** - совпадение с шаблоном поиска при последней операции поиска или замены. В отличии от переменной **$\_**, эту переменную переопределять как вздумается нельзя;
* **$`** - подстрока, расположенная перед совпадение разрешается только чтение;
* **$^R** - результат вычисления утверждения в теле шаблона для последнего вычисления шаблона, если в нем идет счет или вызывается внешняя программа:

**$qwer="lala";**

**$qwer=~ /x(?{$var=5})/;**

**print $^R;**

**5**

* **$n** - n-ный фрагмент совпадения

**print "$1 $2 $3\n" if(/^(\d)(\w)(\W)$/);**

* **\n** - n-ный фрагмент совпадения, вызываемый в самом шаблоне, например поиск гиперссылок:

**/a href=(['"])(.\*?)\1>/**

* **$+** - фрагмент совпадения в шаблоне, который в нем был последним в круглых скобках. Разрешается только чтение $+;
* **$\*** - разрешает выполнять поиск в многострочных файлах, булева переменная, если она взведена в 1, то символы шаблона поиска **^** и **$** сопоставляются позициям перед и после внутренних символов новой строки, если 0, то от начала текста и до конца текста:

**$kim="lala\nfa\eti\nzvuki...";**

**$kim=~~ /^eti/; #совпадение не нашлось**

**$\*=1;**

**$kim=~~ /^eti/; #совпадение нашлось**

Например, нужно занести в массив только цифры из строчки "**12@#34@@#@@###34@@##67##@@#@#@34**":

**$\_='12@#34@@#@@###34@@##67##@@#@#@34';**

**s/@/#/g;**

**s/(#)\1+/$1/g;**

**print join /\n/, split /#/, $\_;**

* **@-** - спецмассив, который содержит начальную позицию найденного слова;
* **@+** - массив, содержащий позицию последнего найденного слова.

Регулярное выражение **s/(#)\1+/$1/g;** использует повторение переменной **$1** (квантификатор **+**) и если оно есть, то заменяет все подряд идущие **#** между цифрами на одну **#**, содержащуюся в**$1** (переменная **$1** существует, если часть шаблона или шаблон указать в круглых скобках).

Допустим нужно определить, все ли цифры числа различны. Попробуем найти хотя бы одно повторяющееся число:

**if(/(\d).\*(?=\1)/g){**

**print "по крайней мере одна цифра $1 различна\n";**

**}**

Выражение берет 1-ю цифру и ищет совпадения со всеми остальными, если есть, то говорит, что найдено и заканчивает работу. Регулярное выражение берет первое число при помощи **(\d)** и начинает его сравнивать со всеми остальными числами при помощи **.\*(?=\1)**. Если первое число в строке уникально, регулярное выражение начнет сопоставлять второе число со всеми восемью оставшимися числами. Если и второе число в строке уникально, то берется третье число и сравнивается со всеми остальными. И т.д., если совпадение было найдено, то регулярное выражение возвращает **true** и заканчивает свою работу, даже если в строке еще есть повторяющиеся числа. Чтобы можно было просмотреть все повторяющиеся числа, можно воспользоваться модификацией предыдущего кода:

**$\_ = '2314152467';**

**my @a = m/(\d)(?=\d\*\1)/g ;**

**if (@a){**

**print join(',',@a)," - Repeat\n";**

**}**

**else{**

**print "Ok\n" ;**

**}**

Этот усовершенствованный код работает до тех пор, пока не будут найдены все совпадения, если таковые вообще есть.

В perl 5.6 вводятся переменные **@-** и **@+**, комбинация которых может заменять переменные **$`**, **$&**, и **$'**. После совпадения шаблона переменная **$-[0]** содержит начало соответствия текста шаблону, а переменная **$+[0]** содержит конец соответствия текста шаблону. В начале поиска обе являются нулями. Это значит, что можно вычислить значения **$`**, **$&**, и **$'**:

**$do = substr($stroka, 0, $-[0]);**

**$sovpalo = substr($stroka, $-[0], $+[0] - $-[0]);**

**$posle = substr($stroka, $+[0]);**

Например:

**$test="11-231234";**

**$test=~/\d{2}-\d{6}/;**

**print "$-[0], $+[0]";**

**0, 9**

Соответственно переменные **$#-** и **$#-** указывают размерность массивов **@-** и **@+**. Переменная **$^N**.

***2.4 Как работают регулярные выражения***

Регулярные выражения, использующие квантификаторы, могут порождать процесс, который называется перебор с возвратом (**backtracking**). Чтобы произошло совпадение текста с шаблоном, надо построить соответствие между текстом и всем регулярным выражением, а не его частью. Начало шаблона может содержать квантификатор, который поначалу срабатывает, но впоследствии приводит к тому, что для части шаблона не хватает текста или возникает несоответствие между текстом и шаблоном. В таких случаях perl возвращается назад и начинает построение соответствия между текстом и шаблоном с самого начала, ограничивая "жадность" квантификатора (именно поэтому процесс и называется "перебор с возвратом"). Перечислим квантификаторы perl:

* **\*** - ноль или несколько совпадений;
* **+** - одно или несколько совпадений;
* **?** - ноль совпадений или одно совпадение;
* **{n}**- ровно n совпадений;
* **{n,}**- по крайней мере n совпадений;
* **{n,m}**- от n до m совпадений.

Например, квантификатор **+** соответствует фразе "один или несколько" и является жадным. Рассмотрим пошагово принцип перебора с возвратом на примере квантификатора **+**:

**'aaabc' =~/a+abc/;**

a+ сразу в силу жадности совпадает с тремя а:

**(aaa)bc,**

но после **aaa** не следует строка "**abc**", а следует "**bc**". Поэтому результат - **failed** поэтому анализатор должен откатиться назад и вернуть с помощью **a+** два **a**: **(aa)abc,**т.е. на втором шаге шаблон найдет совпадение.

Рассмотрим пример работы еще одного жадного квантификатора **\*** (ноль или несколько совпадений):

**amxdemxg /.\*m/**

Сначала будет найдена вся строка **abcdebfg** в силу жадности **.\***, потом квантификатору нужно будет найти сравнение с буквой **m**, произойдет ошибка. Квантификатор **.\*** отдаст одну букву и его содержимое будет уже **amxdemx**. На конце снова нет буквы **m**. Будет отдана еще одна буква и снова не будет найдено совпадение со всем шаблоном и наконец квантификатор **.\*** будет содержать подстроку **amxde**, за которой уже стоит символ **m**. И поиск на этом и закончится не смотря на то, что в строке **amxdemxg** содержится не одна буква **m**. Потому и говорят, что квантификаторы обладают жадностью, т.е. находят максимально возможное совпадение.

Допустим нужно найти совпадение:

**$uu="How are you? Thanks! I'm fine, you are ok??";**

**$uu=~s/.\*you//;**

**print $uu;**

Квантификатор **.\*** оставит текст "**are ok??**", а вовсе не **"? Thanks! I'm fine, you are ok??**". Если же поставить ограничитель **?**, который вместе со знаком квантификатора означает максимально возможное совпадение

**$uu="How are you? Thanks! I'm fine, you are ok??";**

**$uu=~s/.\*you//;**

**print $uu;**

то переменная **$uu** будет содержать текст **"? Thanks! I'm fine, you are ok??**".

Предположим нужно найти совпадения типа **network workshop**, т.е. перекрытия.

**$u='network';**

**$m='workshop';**

**print "перекрытие $2 найдено: $1$2$3\n" if("$u $m" =~/^(\w+)(\w+) \2(\w+)$/);**

**$1** сразу берет все слово в **$u**, но дальше идет еще один максимальный квантификатор **(\w+)**, которому тоже чего-то надо и он забирает из переменной **\1** букву **k** (причем только одну):

**#!/usr/bin/perl**

**$uu="asdfg asdf";**

**$uu=/(\w+)(\w+)\s(\w+)(\w+)/;**

**print "$1 $2##$3 $4";**

**asdf g##asd f**

Далее пошаговая работа regex выглядит примерно так:

**1: 'networ''k'=> '\sk' совпадает ли с '\sworkshop' falure**

**2: 'netwo''rk'=> '\srk' совпадает ли с '\sworkshop' falure**

**3: 'netw''ork'=> '\sork' совпадает ли с '\sworkshop' falure**

**4: 'net''work'=> '\swork' совпадает ли с '\sworkshop' ok**

и в результате программа выдаст:

**перекрытие work найдено: networkshop**

Данный регексп не сработает, если

**$u='networkwork';**

**$m='workshop';**

шаблон найдет перекрытия **workwork**, а не **work**. Чтобы этого избежать, нужно сделать минимальным **\1: /^(\w+?)(\w+) \2(\w+)$/**

Квантификатор действует только на предшествующий ему элемент шаблона. Например, конструкция **\d{2}[a-z]+** будет соответствовать последовательности из одной или нескольких строчных латинских букв, начинающейся с двух цифр, а не последовательности, составленной из чередующихся цифр и букв. Для выделения группы элементов, на которую действует квантификатор, используются круглые скобки: **(\d{2}(a-z])+).**

***2.5 Логические операции в регулярных выражениях***

В регулярных выражениях perl есть синтаксические выражение, позволяющие в шаблонах использовать простые логические конструкции:

* **(?= шаблон)** - после этой точки есть фрагмент текста, который соответствует указанному регулярному выражению;
* **(?! шаблон)** - после этой точки нет текста, который бы соответствовал указанному регулярному выражению;
* **(?<= шаблон)** - перед этой точкой есть фрагмент текста, соответствующий указанному регулярному выражению;
* **(?<! шаблон)** - перед этой точкой нет фрагмента текста, соответствующего указанному регулярному выражению;
* **(?#текст)** - комментарий. Текст комментария игнорируется;
* **(?:шаблон)** или **(?модификаторы:шаблон)** - группирует элементы шаблона. В отличие от обычных круглых скобок, не создает нумерованной переменной. Например, модификаторы не будет делать различия между строчными и заглавными буквами, однако область действия этого модификатора будет ограничена только указанным шаблоном;
* **(?=шаблон)** - "заглядывание вперед". Требует, чтобы после текущей точки находился текст, соответствующий данному шаблону. Такая, конструкция обрабатывается как условие или мнимый символ, поскольку не включается в результат поиска. Например, поиск с помощью команды **/w+(?=\s+)/** найдет слово, за которым следуют один или несколько "пробельных символов", однако сами они в результат не войдут;
* **(?!шаблон)** - случай, противоположный предыдущему. После текущей точки не должно быть текста, соотносимого с заданным шаблоном. Так, если шаблон **w+(?=\s)** - это слово, за которым следует "пробельный символ", то шаблон **w+(?!\s)** - это слово, за которым мет "пробельного символа".
* **(?<=шаблон)** - заглядывание назад. Требует, чтобы перед текущей точкой находился соответствующий текст. Так, шаблон **(?<=\s)w+** интерпретируется как слово, перед которым имеется пробельный символ (в отличие от заглядывания вперед, заглядывание назад может работать только с фиксированным числом проверяемых символов).
* **(?<!шаблон)** - отрицание предыдущего условия. Перед текущей точкой не должно быть текста, соотносимого с заданным шаблоном. Соответственно, от команды **/(?<!\s)w+/** требуется найти слово, перед которым нет пробельного символа.
* **(?{код})** - условие (мнимый символ), которое всегда выполняется. Сводится к выполнению команд perl в фигурных скобках. Вы можете использовать эту конструкцию, только если в начале сценария указана команда **use re 'eval'**. При последовательном соотнесении текста и шаблона, когда perl доходит до такой конструкции, выполняется указанный код. Если полного соответствия для оставшихся элементов найти не удалось, то при возврате левее данной точки шаблона вычисления, проделанные с локальными переменными, откатываются назад. (Условие является экспериментальным. В документации, прилагаемой в perl, можно найти довольно детальное рассмотрение (с примерами) работы этого условия и возможных трудностей в случае его применения.)
* **(?>шаблон)** - "независимый" или "автономный" шаблон. Используется для оптимизации процесса поиска, поскольку запрещает "поиск с возвратом". Такая конструкция соответствует подстроке, на которую налагается заданный шаблон, если его закрепить в текущей точке без учета последующих элементов шаблона. Например, шаблон **(?>а\*)аb** в отличие от **a\*ab** не может соответствовать никакой строке. Если поставить в любом месте шаблон **а\***, он съест все буквы а, не оставив ни одной шаблону **ab**. (Для шаблона **а\*аb**"аппетит" квантификатор **\*** будет ограничен за счет работы поиска с возвратами: после того как на первом этапе не удастся найти соответствие между шаблоном и текстом, perl сделает шаг назад и уменьшит количество букв **а**, захватываемых конструкцией **а\***.)
* **(?(условие)шаблон-да|шаблон-нет)** или **(?(условие)шаблон-да)** - условный оператор, который подставляет тот или иной шаблон в зависимости от выполнения заданного условия. Более подробно описан в документации perl.
* **(?модификаторы)** - задает модификаторы, которые локальным образом меняют работу процедуры поиска. В отличие от глобальных модификаторов, имеют силу только для текущего блока, то есть для ближайшей группы круглых скобок, охватывающих конструкцию, Например, шаблон **((?i)text)** соответcтвует слову "**text**" без учета регистра.

Поиск повторяющихся слов в регулярном выражении осуществляется при помощи т.н. обратных ссылок. Выше уже был приведен пример их использования для выбирания всех адресов рисунков с [www.astronomynow.com](http://www.astronomynow.com):

**m{SRC\s\*=\s\*(["'])http://(.\*?)\1\s+(.\*?)WIDTH="100" HEIGHT="100"(.\*?)>}igs**

**(["'])** - найти либо " либо ' либо ничего, т.к. **src=http://** может быть без кавычек. Как только был найдено что-либо из этих трех позиций, через минимальное количество символов (регулярное выражение **(.\*?))** символов оно заносится в специальную переменную **\1**, которая вне **m/.../** может быть вызвана как **$1**(в **s/.../.../** она вызывается в его левую половину как **$1**). Дальше после **\*.gif|\*.jpg|\*.bmp** и т.д. должен обязательно идти хотя бы один пробел **\s+**, т.к. браузеры воспримут подстроку **src=file.gifborder=0** как файл картинки с расширением **gifborder=0**. Поэтому данное регулярное выражение вполне исправно работает, хотя оно было сделано для сайта, где в **img src**ставится полный адрес, т.е. начинающийся с **http://**Для других сайтов придется выстраивать полные пути в ссылках используя **base href**, если есть или его **url**. Если нужно найти какое-то по счету совпадение шаблона в строке, то это реализуется примерно так:

**while($str=~/WHAT/g){$n++}**

**$n++ while $str=~/WHAT/g;**

**$n++ while $str=~/(?=WHAT)/g;#для перекрывающихся совпадений**

**for($n=0; $n=~/WHAT/g; $n++){}**

Каждое кратное совпадение

**(++$n % 6) == 0;**

Нужное Вам совпадение:

**$n=($str=~/WHAT/gi)[6]; #допустим шестое**

Или каждое четное совпадение

**@mass=grep{$n++ %2==0} /WHAT/gi;**

для нечетного нужно написать внутри grep: **$n++ %2==1**Логические операции внутри регулярных выражений. Если нужно найти последнее совпадение, то можно воспользоваться отрицанием опережающей проверки **(?!WHAT)**:

**m#PATTERN(?!.\*PATTERN)$#**

т.е. найти какой-то **PATTERN**, при этом не должно найтись что-то еще **(.\*)** и **PATTERN**, т.е. результат - последнее совпадение;

Минимальные квантификаторы **\*?, +?, ??,{}?**

Допустим нужно найти двойку, перед которой не стоит 3 или пробел:

**print "$1\n" while m%2(?![3\s])gm%;**

используется условие по отрицанию, **A(?!B)**: найти А, перед которым не находится В. Чтобы найти двойку, за которой стоит 3 или пробел **(\s)**, то можно воспользоваться:

**print "$1\n" while m%2(?=[3\s])gm%;**

или

**print "$1\n" while m%2(?![^3\s])gm%;**

где используется **^, [^3\s]**, который значит следующее: в класс символов, которые нужно найти, не входят 3 и пробел, или другими словами найти все кроме 3 и **\s**.

Допустим, существует HTML-документ, в котором произвольное число вложенных таблиц **[&lt;table>.\*</table>]**. Требуется "вырезать" по очереди самые вложенные таблицы (не содержащие внутри **[<table>.\*</table>]**), и, соответственно, выводить. И так - рекурсивно до конца вырезать изнутри всю таблицу. Ниже представлена программа, реализующая эту задачу при помощи логического оператора **(?!...)**:

**#!/usr/bin/perl -wT**

**$file=qq|s<table>aaa bbb**

**<table>cc<table>ccc**

**<table> 2<table>bb</table> <table>cc</table> </table></table>cc**

**</table>**

**ddd</table>d**

**|;**

**print $file;**

**&req($file);**

**sub req {**

**if($file=~m%(<table>((?!.\*<table>).\*?)</table>)%igs){**

**$file=~s%(<table>((?!.\*<table>).\*?)</table>)%%igs;**

**print "Virezali --$1--";**

**&req($file);**

**}**

**return $file;**

**}**

Продолжаем рассматривать логические операторы в регулярных выражениях на операторах типа **OR**, **AND** или **NOT**.

Регексп истинен, если **/AM|BMA/** или **/AM/ || /BMA/**и если есть перекрытие типа **/BMAM/**. Также и **/AM/ && /BMA/**:

**/^(?=.\*AM)(?=.\*BMA)/s**

Выражение истинно если **/AM/** и **/BMA/**совпадают при перекрытии, которое не разрешено:

**/AM.\*BMA|BMA.\*AM/s**

Выражение истинно, если шаблон **/ABC/**не совпадает:

**!~/ABC/**

Или

**/^(?:(?!ABC).)\*$/s**

Выражение истинно, если **ABC**не совпадает, а **VBN** совпадает:

**/(?=^(?:(?!ABC).)\*$)VBN/s**

Несовпадение можно проверить несколькими способами:

**unless($str =~ /MMM/){...}**

**if(!($str =~ /MMM/)){...}**

**if($str !~ /MMM/){...}**

Для обязательного совпадения в двух шаблонах:

**unless ($str !~ /MMM/ && $str !~ /BBB/){...}**

**#или**

**if ($str =~ /MMM/ && $str =~ /BBB/){...}**

Хотя бы в одном

**unless ($str !~ /MMM/ || $str !~ /BBB/){...}**

**#или**

**if ($str =~ /MMM/ || $str =~ /BBB/){...}**

Регулярные выражения - основа работы с операторами **m/.../** и **s/.../.../**, так как они передаются последним в качестве аргументов. Разберемся, как устроено регулярное выражение **\b([A-Za-z)+)\b**, осуществляющее поиск отдельных слов в строке:

**$text = "Perl is the subject.";**

**$text =~/\b([A-Za-z]+)\b/;**

**print $1;**

Выражение **\b([A-Za-z]+)\b** включает в себя группирующие метасимволы **(** и **)**, метасимвол границы слова **\b**, класс всех латинских букв **[A-Za-z]**(он объединяет заглавные и строчные буквы) и квантификатор **+**, который указывает на то, что требуется найти один или несколько символов рассматриваемого класса. Поскольку регулярные выражения, как это было в предыдущем примере, могут быть очень сложными, разберем их по частям. В общем случае регулярное выражение состоит из следующих компонентов:

Совпадение с любым символом

В perl имеется еще один мощный символ - а именно, точка (**.**). В шаблоне он соответствует любому знаку, кроме символа новой строки. Например, следующая команда заменяет в строке все символы на звездочки (использован модификатор **g**, обеспечивающий глобальную замену):

**$text = "Now is the time.";**

**$text =~ s/./\*/g;**

**print $text;**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

А что делать, если требуется проверить совпадение именно с точкой? Символы вроде точки (конкретно, **\|()[{^$\*+?.**), играющие в регулярном выражении особую роль) называются, как уже было сказано выше, *метасимволами*, и если вы хотите, чтобы они внутри шаблона интерпретировались как обычные символы, метасимволу должна предшествовать обратная косая черта. Точно так же обратная косая черта предшествует символу, используемому в качестве ограничителя для команды **m/.../, s/.../.../** или **tr/.../.../**, если он встречается внутри шаблона и не должен рассматриваться как ограничитель. Рассмотрим пример:

**$line = ".Hello!";**

**if ($1ine =- m/\./) {**

**print "Shouldn't start a sentence with a perlod!\n";**

**}**

**Shouldn't start a sentence with a perlod!**

Если нужно найти самый короткий текстовый фрагмент **/QQ(.\*?)FF/**в "**QQ ff QQ ff FF**", однако оно найдет "**ff QQ ff**". Шаблон всегда находит левую строку минимальной длины, которая соответствует всему шаблону, т.е. это вся строка в этом примере. Для правильного шаблона нужно воспользоваться логическими операторами в регулярных выражениях: **/QQ((?:(?!QQ).)\*)FF/**, т.е. сначала **QQ**, потом не **QQ**, потом **FF**.

Конструкции **(?<=шaблoн)**и **(?<!шаблон)**работают только с шаблонами, соответствующими фиксированному числу символов. Иными словами, в шаблонах, указываемых для **(?<=...)** и **(?<!...)**, не должно быть квантификаторов.

Эти условия полезны, если нужно проверить, что перед определенным фрагментом текста или после него находится нужная строка, однако ее не требуется включать в результат поиска. Это бывает необходимо, если в коде используются специальные переменные **$&** (фрагмент, для которого найдено соответствие между текстом и регулярным выражением), **$`**(текст, предшествующий найденному фрагменту) и **$'** (текст, следующий за найденным фрагментом). Более гибким представляется применение нумерованных переменных **$1, $2, $3, ...** в которые заносятся отдельные части найденного фрагмента.

В следующем примере ищется слово, за которым следует пробел, но сам пробел не включается в результат поиска:

**$text = "Маrу Tom Frank ";**

**while ($text =~ /\w+(?=\s)/g) {print $& . "\n";}**

**Маrу**

**Tom**

**Frank**

Того же результата можно добиться, если заключить в круглые скобки интересующую нас часть шаблона и затем использовать ее как переменную **$1**:

**$text = "Mary Tom Frank ";**

**while ($text =~ /(\w+)\s/g) {**

**print $1 . "\n";**

**}**

**Маrу**

**Tom**

**Frank**

Следует четко понимать, что вы имеете в виду, когда используете то или иное условие. Рассмотрим следующий пример:

**$text="Mary+Tom";**

**if($text=~m|(?!Mary\+)Tom|){**

**print "Tom is without Mary!\n";**

**}**

**else{**

**print "Tom is busy...\n";**

**}**

Вопреки нашим ожиданиям, perl напечатает: **Tom is without Mary!**Это произойдет по следующей причине. Пробуя различные начальные точки входной строки, от которой начинается сопоставление шаблона и текста, pеr1 рано или поздно доберется до позиции, расположенной прямо перед именем "**Tom**". Условие **(?!Маry\+)**требует, чтобы после текущей точки не находился текст **\*Маry+"**, и это условие для рассматриваемой точки будет выполнено. Далее, perl последовательно проверяет, что после текущей точки следуют буквы "**Т**", "**o**" и "**m**", и это требование также в силе (после проверки условия **(?!Маry\+)**текущая точка остается на месте). Тем самым найдено соответствие между подстрокой "**Тоm**" и шаблоном, поэтому команда поиска возвращает значение истина.

Регулярное выражение **(?!Mary\+)....Tom**, резервирующее четыре символа под текст **"Маry+**", для приведенного выше случая выведет то, что требовалось, но выдаст ошибочный ответ, если перед именем "**Тоm**" нет четырех символов:

**$text="O, Tom! ";**

**if($text =~ m|(?!Mary\+)....Tom|){**

**print "Tom is without Mary!\n";**

**}**

**else{**

**print "Tom is busy...\n";**

**}**

**Tom is busy...**

Наконец, если более точно сформулировать, чего требуется, получится нужный результат:

**$text="Mary+Tom";**

**if($text=~m|(?<!Mary\+)Tom|){**

**print "Tom is without Mary!\n";**

**}**

**else{**

**print "Tom is busy...\n";**

**}**

**Tom is busy...**

Вспомнить и написать про строчку вида

**push @mass, $li unless($li=~m/(([2 .. 12]).\*?1995)|(([6 .. 12]).\*?2001)|/)**

**; perldoc perlop [0-9.]**

Модификаторы команд **m/.../** и **s/.../.../**

В perl имеется несколько модификаторов, используемых с командами **m/.../** и **s/.../.../**:

* **i** - игнорирует различие между заглавными и строчными буквами.
* **s** - метасимволу "точка" разрешено соответствовать символам **\n**.
* **m** - разрешает метасимволам **^** и **$** привязываться к промежуточным символам **\n**, имеющимся в тексте. Не влияет на работу метасимволов **\А**, **\Z** и **\z**.
* **х** - игнорирует "пробельные символы" в шаблоне (имеются в виду "истинные" пробелы, а не метасимволы **\s** и пробелы, созданные через escape-последовательности). Разрешает использовать внутри шаблона комментарии.
* **g** - выполняет глобальный поиск и глобальную замену.
* **с** - после того как в скалярном контексте при поиске с модификатором **g** не удалось найти очередное совпадение, не позволяет сбрасывать текущую позицию поиска. Работает только для команды **m/.../** и только вместе с модификатором **g**.
* **о** - запрещает повторную компиляцию шаблона при каждом обращении к данному оператору поиска или замены, пользователь, однако, должен гарантировать, что шаблон не меняется между вызовами данного фрагмента кода.
* **е** - показывает, что правый аргумент команды **s/.../.../** - это фрагменты выполняемого кода. В качестве текста для подстановки будет использовано возвращаемое значение - возможно, после процесса интерполяции.
* **ee** - показывает, что правый аргумент команды **s/.../.../** - это строковое выражение, которое надо вычислить и выполнить как фрагмент кода (через функцию **eval**). В качестве текста для подстановки используется возвращаемое значение - возможно, после процесса интерполяции

**Особенности работы команд m/.../ и s/.../.../.**

До сих пор мы рассматривали регулярные выражения, используемые в качестве шаблонов для команд **m/.../**и **s/.../.../,** и не особо интересовались, как работают эти команды. Настало время восполнить пробелы.

Команда **m/.../**ищет текст по заданному шаблону. Ее работа и возвращаемое значение сильно зависят от того, в скалярном или списковом контексте она используется и имеется ли модификатор **g** (глобальный поиск).

Команда **s/.../.../**ищет прототип, соответствующий шаблону, и, если поиск оказывается успешным, заменяет его на новый текст. Без модификатора замена производится только для первого найденного совпадения, с модификатором **g** выполняются замены для всех, совпадений во входном тексте. Команда возвращает в качестве результата число успешных замен или пустую строку (условие ложь **false**), если ни одной замены сделано не было. В качестве анализируемого текста используется **$\_** (режим по умолчанию) или выражение, присоединенное к шаблону с помощью оператора **=~** или **!~**. В случае поиска (команда **m/.../**) конструкция, расположенная слева от операторов **=~** или **!~**, может и не быть переменной. В случае замены (команда **s/.../.../**) в левой части должна стоять скалярная переменная, или элемент массива, или элемент хэша, или же команда присвоения одному из указанных объектов.

Вместо косой черты в качестве ограничителя для аргументов команд **m/.../** и **s/.../.../** можно использовать любой символ, за исключением "пробельного символа", буквы или цифры. Например, в этом качестве можно использовать символ комментария, который будет работать как ограничитель:

**$text="ABC-abc";**

**$text =~ s#B#xxx#ig;**

**print $text;**

**AxxxC-axxxc**

В качестве ограничителей не стоит использовать вопросительный знак и апостроф (одинарную кавычку) - шаблоны, с такими ограничителями обрабатываются специальным образом. Если команда **m/.../** использует символ косой черты в качестве разделителя, то букву **m** можно опустить:

**while (defined($text = <>))**

**{ if ($text =~/^exit$/i) {exit;} }**

Если в качестве ограничителя для команды **m/.../** используется вопросительный знак, то букву **m** также можно опустить. Однако шаблоны, ограниченные символом **?**, в случае поиска работают особым образом (независимо от наличия или отсутствия начальной **m**). А именно, они ведут себя как триггеры, которые срабатывают один раз и потом выдают состояние ложь (**false**), пока их не взведут снова, вызвав функцию **reset** (она очищает статус блокировки сразу всех конструкций **?...?**, локальных для данного пакета). Например, следующий фрагмент сценария проверяет, есть ли в файле пустые строки:

**while (<>)**

**if (?^$?) {print ."There is an empty line nere.\n";} continue {**

**reset if eof; #очистить для следующего файла**

**}**

Диагностическое сообщение будет напечатано только один раз, даже если в файле присутствует несколько пустых строк. Команда поиска с вопросительным знаком относится к подозрительным командам, а потому может не войти в новые версии perl. В качестве ограничителей можно также использовать различные (парные) конструкции скобок:

**while (<>){**

**if(m/^quit$/i){exit;}**

**if(m(^stop$)i){exit;}**

**if(m[^end$]i) {exit;}**

**if(m{^bye$}i) {exit;}**

**if (!1)<^ехit$>i) {exit;}**

**}**

В случае команды **s/.../.../**и использования скобок как ограничителей для первого аргумента, ограничители второго аргумента могут выбираться независимо:

**$text =~ "Perl is wonderful";**

**$text =~ s/is/is very/;**

**$text =~ s[wonderful]{beautiful};**

**$text =~ s(\.)/!/;**

**print $text;**

**Perl is very beautiful!**

Предварительная обработка регулярных выражений

Аргументами команд **m/.../**и **s/.../.../**являются регулярные выражения, которые перед началом работы интерполируются подобно строкам, заключенным в двойные кавычки В отличие от текстовых строк, для шаблона не выполняется интерполяция имен типа **$**), **$|** и одиночного **$** - perl считает, что такие конструкции соответствуют метасимволу конца строки, а не специальной переменной. Если же в результате интерполяции шаблон поиска оказался пустой строкой, perl использует последний шаблон, который применялся им для поиска или замены.

Если вы не хотите, чтобы perl выполнял интерполяцию регулярного выражения, в качестве ограничителя надо использовать апостроф (одиночную кавычку), тогда шаблон будет вести себя, как текстовая строка, заключенная в апострофы. Однако, например, в случае команды замены **s/.../.../** с модификатором е или ее (их работа описывается чуть дальше) для второго аргумента будет выполняться интерполяция даже в том случае, если он заключен в апострофы.

Если вы уверены, что при любом обращении к команде поиска или замены шаблон остается неизменным (например, несмотря на интерполяцию, скалярные переменные внутри шаблона не будут менять своего значения), то можно задать модификатор о. Тогда perl компилирует шаблон в свое внутреннее представление только при первой встрече с данной командой поиска или замены. При остальных обращениях к команде будет использовать откомпилированное значение. Однако, если внезапно изменить значение переменных, задействованных в шаблоне, perl этого даже не заметит.

Команда замены **s/.../.../** использует регулярное выражение, указанное в качестве второго аргумента, для замены текста. Поскольку оно обрабатывается (интерполируется) после того, как выполнена очередная операция поиска, в нем можно, в частности, использовать временные переменные, созданные на этапе поиска. В следующем примере мы последовательно заменим местами пары слов, заданных во входном тексте, оставив между ними по одному пробелу:

**$text = "One Two Three Four Five Six";**

**$text =- s/(\w+)\s\*(\w+)/$2$1/g;**

**Two One Four Three Six Five**

Однако perl допускает и более сложные способы определения заменяющего текста. Так, если для команды **s/.../.../** указать модификатор е, то в качестве второго аргумента надо указать код, который необходимо выполнить (например, вызвать функцию). Полученное выражение будет использовано как текст для подстановки. При этом после вычисления текстового значения, но перед его подстановкой будет выполнен процесс интерполяции, аналогичный процессу интерполяции текстовых строк, заключенных в двойные кавычки. Еще более сложная схема реализуется, если задан модификатор ее. В этом случае второй аргумент команды **s/.../.../** - это строковое выражение, которое сперва надо вычислить (то есть интерполировать), затем выполнить в качестве кода (вызвав встроенную функцию **eval**) и только после второй интерполяции полученный результат подставляется вместо найденного текста.

Работа команды **m/.../** в режиме однократного поиска В скалярном контексте и без модификатора **g** команда **m/.../** возвращает логическое значение - целое число 1 (истина **(true)**), если поиск оказался успешным, и пустую строку **""** (ложь **(false)**), если нужный фрагмент текста найти не удалось. Если внутри шаблона имеются группы элементов, заключенные в круглые скобки, то после операции поиска создаются нумерованные переменные **$1**, **$2**, **...**, в которых содержится текст, соответствующий круглым скобкам. В частности, если весь шаблон заключить в круглые скобки, то в случае успешного поиска переменная **$1** будет содержать текст, соотнесенный с шаблоном. После успешного поиска можно также использовать специальные переменные **$&**, **$'**, **$'** и **$+**

**$text = "---one---two---three---";**

**$scalar = ($text =' m/(\w+)/);**

**print "Result: $scalar ($1).";**

**Result: 1 (one).**

Если вы используете команду **m/.../** в списковом контексте, то возвращаемое значение сильно зависит от того, есть ли группы из круглых скобок в вашем шаблоне. Если они есть (то есть если создаются нумерованные переменные), то после успешного поиска в качестве результата будет получен список, составленный из нумерованных переменных **($1**, **$2**,**...**):

**$text = "---one, two, three---";**

**array = ($text ='m/(\w+),\s+(\w+),\s+(\w+)/);**

**print join "=", array;**

**one=two=three.**

В отличие от ранних версий, perl 5 присваивает значения нумерованным переменным, даже если команда поиска работает в списковом контексте:

**$text = "---one, two, three--- ";**

**($Fa, $Fb, $Fc) = ($text=-m/(\w+),\s+(\w+),\s+(\w+)/);**

**print "/$Fa/$Fb/$Fc/\n";**

**print "$1=$2=$3.\n";**

**/one/two/three/**

**one=two::three.**

Если же в шаблоне нет групп, выделенных круглыми скобками, то в случае успешного поиска возвращается список, состоящий из одного элемента - числа 1. При неудачном поиске независимо от того, были ли в шаблоне круглые скобки, возвращается пустой список:

**$text = "---one, two, three--- ";**

**@array = ($text=~ m/z\w+/);**

**print "Result: /", @array, "/\n";**

**print "Size: ", $#array+1, ".\n";**

**Result://**

**Size: 0.**

Обратите внимание на разницу между пустым и неопределенным списками.

**Работа команды m/.../ в режиме глобального поиска**

Команда **m/.../** работает иначе, если указан модификатор **g**, задающий глобальный поиск всех вхождений шаблона по всему тексту. Если оператор используется в списковом контексте и в шаблоне есть группы круглых скобок, то в случае удачного поиска возвращается список, состоящий из всех найденных групп, расположенных друг за другом:

**$text = "---one---two-~-three---";**

**@array = ($text =~m/(-(\w+))/);**

**print "Single: [", join(", ", array),"].\n";**

**@array = ($text =~m/(-(\w+))/g);**

**print "Global: [", join(", ", array),"].\n";**

**Single: [-one, one].**

**Global: [-one, one, -two, two, -three, three].**

Если же в шаблоне нет групп круглых скобок, то оператор поиска возвращает список всех найденных прототипов шаблона, то есть ведет себя так, как если бы весь шаблон был заключен в круглые скобки:

**$text = "---one---two---three--";**

**@array = ($text =~m/\w+/);**

**print "Result: (", join(", ", @array), ").\n";**

**Result: (one, two, three).**

В случае неудачного поиска, как и в предыдущих вариантах, возвращается пустой список. В скалярном контексте и с модификатором **g** комaндa **m/.../**ведет себя совершенно особым образом. Специальная переменная **$\_** или переменная, стоящая слева от оператора **=~** или **!~**, при поиске с модификатором **g**получает дополнительные свойства - в нее записывается последнее состояние. При каждом последующем обращении к данному фрагменту кода поиск будет продолжаться с того места, на котором он остановился в последний раз. Например, следующая команда подсчитывает количество букв **х** в заданной строке текста:

**$text = "Here is texxxxxt.";**

**$counter = O;**

**while ($text =~ m/x/g){**

**print "Found another x.\n";**

**$conter++;**

**print "Total amount = $counter.\n";**

**Found another х.**

**Found another х.**

**Found another x.**

**Found another x.**

**Found another x.**

**Total amount = 5.**

Состoяние (точнее, позиция) поиска сохраняется даже в случае перехода к следующему оператору поиска, имеющему модификатор **g**. Неудачный поиск сбрасывает значение в исходное состояние, если только для команды **m/.../** не указан модификатор с (то есть команда должна иметь вид **m/.../gc**). Изменение текстового буфера, для которого выполняется поиск, также сбрасывает позицию поиска в исходное состояние. В следующем примере из текстовой строки последовательно извлекаются и выводятся пары имя/значение до тех пор, пока строка не закончится:

**$text = "X=5; z117e=3.14l6; temp=lQ24;";**

**$docycle = 1; $counter = 0;**

**while ($docycle) {**

**undef $name; undef $value;**

**if ($text =~ m/(\w+)\s\*=\s\*/g) {$name = $1;}**

**if ($text =~ m/([\d\.\\*\-]\*)\s\*;/g) {$value = $1;}**

**if (defined($name) and defined($value)) {**

**print "Name=$name, Value=$value.\n";**

**$counter++,**

**}else{**

**$docycle = 0;**

**}**

**}**

**print "I have found $conter values.\n";**

**Name=X, Value=5.**

**Name=z117e, Value=3.1416.**

**Name=temp, Value=1024.**

**I have found 3 values.**

Позиция, на которой остановился поиск, может быть прочитана и даже переустановлена с помощью встроенной функции perl **pos**. В шаблоне на текущую позицию поиска можно ссылаться с помощью метасимвола **\G**. В следующем примере из строки последовательно извлекаются буквы **p**, **o** и **q** и выводится текущая позиция поиска:

**$index = 0;**

**$\_ = "ppooqppqq";**

**while ($index++ < 2) {**

**print "1: '";**

**print $1 while /(o)/gc; print "', pos=", pos, "\n";**

**print "2: '";**

**print $1 if /\G(q)/gc; print "', pos=";' pos, "\n";**

**print "3: '";**

**print while /(p)/gc; print "', pos=",pos, "\n";**

**}**

**1: 'oo', pos=4;**

**2: 'q', pos=7;**

**3: 'pp', pos=4;**

**1: '', pos=7;**

**2: 'q', pos=8;**

**3: '', pos=8;**

В документации perl приводится основанный на этом механизме интересный пример последовательного лексического разбора текста. В нем каждая последующая команда поиска очередной лексической единицы начинает выполняться с того места, где завершила свою работу предыдущая (страница руководства **perlop**, раздел "Regexp Quote-Uke Operators", описание команды **m/PATTERN/**).

**Замена строк с помощью команды tr/.../.../**

Кроме команд **m/.../ и s/.../.../**строки можно обрабатывать с помощью команды **tr/.../.../**(она же - команда **у/.../.../**):

**tr/список1/список2/модификаторы;**

**у/список1/список2/модификаторы;**

В отличие от **m/.../**и **s/.../.../**, эта команда не использует шаблоны и регулярные выражения, а выполняет посимвольную замену, подставляя в текст вместо литер из первого списка соответствующие им литеры из второго списка. Например, в следующем случае производится замена литер "**i**" на "**о**":

**$text = "My name is Tim.";**

**$text =~ tr/i/o/; print $text;**

**My name is Tom.**

В качестве списков используются идущие друг за другом символы, не разделяемые запятыми (то есть это скорее строки, чем списки). В отличие от шаблонов команд **m/.../**и **s/.../.../**, аргументы команды tr/.../.../ не интерполируются (то есть подстановки значений вместо имен переменных не происходит), хотя escape-последовательности, указанные внутри аргументов, обрабатываются правильно. Подобно **m/.../** и **s/.../.../**, команда **tr/.../.../** пo умолчанию работает с переменной **$\_**:

**while (<>){**

**tr/iI/jJ/;**

**print;**

В качестве списков можно указывать диапазоны символов - как, например, в следующем фрагменте кода, заменяющем строчные буквы на заглавные:

**$text = "Here is the text.";**

**$text =~ tr/a-z/A-Z/;**

**print $text;**

**HERE IS THE TEXT.**

Как и в случае **m/.../** u **s/.../.../**, команда **tr/.../.../** не требует использовать именно знаки косой черты в качестве ограничителей. Можно использовать практически любой символ, отличный от "пробельных", букв и цифр, а также парные скобочные конструкции.

Команда **tr/.../.../** возвращает число успешных замен. В частности, если не было сделано никаких замен, она возвращает число ноль. Это позволяет, например, подсчитать с помощью команды **tr/.../.../** количество вхождений буквы **х** в строку **$text**, не меняя содержимого этой переменной:

**$text = "Here is the text.";**

**$xcount = ($text =~tr/x/x/);**

**print $xcount;**

**1**

Если у команды **tr/.../.../** нет модификаторов (см. далее раздел "Модификаторы команды tr/.../.../"), то ее аргументы при обычных условиях должны быть одинаковой длины. Если второй аргумент длиннее первого, то он усекается до длины первого аргумента - так, команда **tr/abc/0-9/** эквивалентна команде **tr/abc/012/**. Если первый аргумент длиннее второго и второй не пуст, то для второго аргумента необходимое число раз повторяется его последний символ - так, команда **tr/O-9/abc/** эквивалентна команде **tr/0123456789/abcccccccc/**. Если же второй, аргумент пуст, то команда **tr/.../.../** подставляет вместо него первый аргумент.

Как легко заметить, если второй аргумент пуст, то (при отсутствии модификаторов) команда **tr/.../.../** не производит никаких действий, а возвращаемое ею значение равно числу совпадений между первым аргументом и обрабатываемым текстом. Например, следующая команда подсчитывает количество цифр в строке:

**$text = "Pi=3.1415926536, е=2.7182";**

**$digit\_counter=($text =~ tr/0-9//);**

**print $digit\_counter;**

**16**

Команда **tr/.../.../** работает без рекурсии, просто последовательно заменяет символы входного текста. Например, для замены заглавных букв на строчные, и наоборот, достаточно выполнить команду:

**$text = "MS Windows 95/98/NT";**

**$text =" tr/A-Za-z/a-zA-Z/;**

**print $text;**

**ms WINDOWS 95/98/nt**

Если в списке, указанном в качестве первого аргумента, есть повторяющиеся символы, то для замены используется первое вхождение символа:

**$text = "Billy Gates";**

**$text =~ tr/ttt/mvd/;**

**print $text;**

**Billy Games**

Модификаторы команды **tr/.../.../**

Команда **tr/.../.../**допускает использование следующих модификаторов:

* **d**- удаляет непарные символы, не выравнивая аргументы по длине.
* **с** - в качестве первого аргумента использует полный список из 256 символов за вычетом указанных в списке символов.
* **s** - удаляет образовавшиеся в результате замены повторяющиеся символы.

Если указан модификатор **d**, a первый аргумент команды длиннее второго, то все символы из первого списка, не имеющие соответствия со вторым списком, удаляются из обрабатываемого текста. Пример: удаляем строчные латинские буквы и заменяем пробелы на слэши:

**$text = "Here is the text.";**

**$text =~ tr[ a-z][/]d;**

**print $text;**

**H///.**

Наличие модификатора **d**- единственный случай, когда первый и второй аргументы не выравниваются друг относительно друга, В остальных вариантах второй аргумент либо усекается, либо последний символ в нем повторяется до тех пор, пока аргументы не сравняются, либо, если второй аргумент пуст, вместо второго аргумента берется копия первого.

Если указан модификатор **с**, то в качестве первого аргумента рассматриваются все символы, кроме указанных. Например, заменим на звездочки все символы, кроме строчных латинских букв:

**$text = "Here is the text,";**

**$text =' tr/a-z/\*/c;**

**print $text;**

**\*ere\*is\*the\*text\***

Если указан модификатор **s**, то в случае если замещаемые символы образуют цепочки из одинаковых символов, они сокращаются до одного. Например, заменим слова, состоящие из латинских букв, на однократные символы косой черты:

**$text = "Here is the text.";**

**$text ="tr(A-Za-z)(/)s;**

**print $text;**

**/ / / /.**

Без модификатора s результат был бы другим:

**$text = "Here is the text.";**

**$text =' tr(A-Za-z)(/);**

**print $text;**

**//// // /// ////.**

***2.6 Примеры***

**1. Заменить множественные пробелы и нетекстовые символы на одиночные пробелы:**

**$text = "Here is the text."**

**$text =~ tr[\000-\040\177\377][\040]s;**

**print $text;**

**Here is the text.**

**2. Сократить удвоенные, утроенные и т.д. буквы;**

**$text = "Here is the texxxxxxt.";**

**$text =~ tr/a-zA-Z/s;**

**print $text;**

**Here is the text.**

**3. Пересчитать количество небуквенных символов:**

**$xcount=($text =~ tr/A-Za-z//c);**

**4. Обнулить восьмой бит символов, удалить нетекстовые символы:**

**$text =- tr{\200-\377}{\000-\l77};**

**$text =~ tr[\000-\037\177][]d;**

**5. Заменить нетекстовые и 8-битные символы на одиночный пробел:**

**$text =~ tr/\021-\176/ /cs;**

**6. Поиск отдельных слов**

Чтобы выделить слово, можно использовать метасимвол **\S** соответствующий символам, отличным от "пробельных":

**$text = "Now is the time.";**

**$text =- /(\S+)/;**

**print $1;**

**Now**

Однако метасимвол **\S** соответствует также и символам, обычно не используемым для идентификаторов. Чтобы отобрать слова, составленные из латинских букв, цифр и символов подчеркивания, нужно использовать метасимвол **\w**:

**$text = "Now is the time.";**

**$text =~ /(\w+)/;**

**print $1;**

**Now**

Если требуется включить в поиск только латинские буквы, надо использовать класс символов:

**$text = "Now is the time.";**

**$text =~ /([A-Za-z]+)/;**

**print $1;**

**Now**

Более безопасный метод состоит в том, чтобы включить в шаблон мнимые символы границы слова:

**$text = "How is the time.";**

**$text=~/\b([A-Za-z]+)\b/;**

**print $1;**

**Now**

1. **Привязка к началу строки**

Началу строки соответствует метасимвол (мнимый символ) **^**. Чтобы шаблон к началу строки, надо задать этот символ в начале регулярного выражения. Например, вот так можно проверить, что текст не начинается с точки:

**$line = ".Hello!";**

**if($line=~m/^\./){**

**print "Shouldn't start a sentence with a period!\n";**

**}**

**Shouldn't start a sentence with a period!**

Чтобы точка, указанная в шаблоне, не интерпретировалась как метасимвол перед ней пришлось поставить обратную косую черту.

1. **Привязка к концу строки**

Чтобы привязать шаблон к концу строки, используется метасимвол (мнимый символ) **$**. В нашем примере мы используем привязку шаблона к началу и к концу строки, чтобы убедиться, что пользователь ввел только слово "**exit**":

**while(<>){**

**if(m/"exlt$/) {exit;}**

**}**

1. **Поиск чисел**

Для проверки того, действительно ли пользователь ввел число, можно использовать метасимволы **\d** и **\D**. Метасимвол **\D** соответствует любому символу, кроме цифр. Например, следующий код проверяет, действительно ли введенный текст представляет собой целое значение без знака и паразитных пробелов:

**$test = "Hello!";**

**if($text =~ /\D/){**

**print "It is not a number.\n";**

**}**

**It is not a number.**

To же самое можно проделать, использовав метасимвол **\d**:

**$text = "333";**

**if($text =~ /^\d+$/){**

**print "It is a number.\n";**

**}**

**It is a number.**

Вы можете потребовать, чтобы число соответствовало привычному формату. То есть число может содержать десятичную точку, перед которой стоит по крайней мере одна цифра и, возможно, какие-то цифры после нее:

**$text= "3,1415926";**

**if($text =~ /^(\d+\.\d\*|\d+)$/){**

**print "It is a number.\n";**

**}**

**It is a number.**

Кроме того, при проверке можно учитывать тот факт, что перед числом может стоять как плюс, так и минус (или пустое место):

**$text = "-2.7182";**

**if ($text =~ /^([+-]\*\d+)(\.\d\*|)$/) {**

**print "It is a number.\n";**

Поскольку плюс является метасимволом, его надо защищать обратной косой чертой. Однако внутри квадратных скобок, то есть класса символов, он не может быть квантификаторам. Знак "минус" внутри класса символов обычно играет роль оператора диапазона и поэтому должен защищаться обратной косой чертой. Однако в начале или в конце шаблона он никак не может обозначать диапазон, и поэтому обратная косая черта необязательна. Наконец, более строгая проверка, требует, чтобы знак, если он присутствует, был только один:

**$text = "+0.142857142857142857";**

**if ($text =~ /^(+|-|)\d+(\.\d\*\)$/) {**

**print "It is a number.\n";**

**}**

**It is a number.**

Альтернативные шаблоны, если они присутствуют, проверяются слева направо. Перебор вариантов обрывается, как только найдено соответствие между текстом и шаблоном. Поэтому, например, порядок альтернатив в шаблоне **(\.\d\*|)** мог бы стать критичным, если бы не привязка к концу строки. Наконец, вот как можно произвести проверку того, что текст является шестнадцатеричным числом без знака и остальных атрибутов:

**$text = "1AO";**

**unless (ftext =~ m/^[a-fA-F\d]+$/) {**

**print "It is not a hex number, \n";**

**}**

1. **Проверка идентификаторов**

С помощью метасимвола **\w** можно проверить, состоит ли текст только из букв, цифр и символов подчеркивания (это те символы, которые perl называет словесными (**word characters**)):

**$text="abc";**

**if($text=~/^\w+$/){**

**print "Only word characters found. \n";**

**}**

**Only word characters found.**

Однако, если вы хотите убедиться, что текст содержит латинские буквы и не содержит цифр или символов подчеркивания, придется использовать другой шаблон:

**$text = "аbс";**

**if($text=~ /^[A-Za-z]+$/)**

**{ print "Only letter characters found.\n";}**

**Qnly letter characters found.**

Наконец, для проверки, что текст является идентификатором, то есть начинаетcя с буквы и содержит буквы, цифры и символы подчеркивания, можно использовать команду:

**$text = "X125c";**

**if($text=~ /^[A-Za-z]\w+$/)**

**{ print "This is identifier.\n";}**

**This is identifier.**

1. **Как найти множественные совпадения**

Для поиска нескольких вхождений шаблона можно использовать модификатор g. Следующий пример, который мы уже видели ранее, использует команду **m/.../** с модификатором g для поиска всех вхождений буквы x в тексте:

**$text="Here is texxxxxt";**

**while($text=~m/x/g){**

**print "Found another x.\n";**

**}**

**Found another x.**

**Found another x.**

**Found another x.**

**Found another x.**

**Found another x.**

Модификатор g делает поиск глобальным. В данном (скалярном) контексте perl помнит, где он остановился в строке при предыдущем поиске. Следующий поиск продолжается с отложенной точки. Без модификатора **g** команда **m/.../** будет упорно находить первое вхождение буквы **х**, и цикл будет продолжаться бесконечно.

В отличие от команды **m/.../** команда **s/.../.../** с модификатором **g** выполняет глобальную замену за один раз, работая так, будто внутри нее уже имеется встроенный цикл поиска, подобный приведенному выше. Следующий пример за один раз заменяет все вхождения **х** на **z**:

**$text = "Here is texxxxxt.";**

**$text =~ s/x/z/g;**

**print $text;**

**Here is tezzzzzt.**

Без модификатора **g** команда **s/.../.../** заменит только первую букву **х**. Команда **s/.../.../** возвращает в качестве значения число сделанных подстановок, что может оказаться полезным:

**$text= "Here is texxxxxt.";**

**print (text =~ s/x/z/g)**

**5**

1. **Поиск нечувствительных к регистру совпадений**

Вы можете использовать модификатор **i**, чтобы сделать поиск нечувствительным к разнице между заглавными и строчными буквами. В следующем примере программа повторяет на экране введенный пользователем текст до тех пор, пока не будет введено **Q**, или **q** (сокращение для **QUIT** или **quit**), после чего программа прекращает работу:

**while(<>){**

**chomp;**

**unless (/^q$/i){**

**print**

**}**

**else {**

**exit;**

**}**

**}**

1. **Выделение подстроки**

Чтобы получить найденную подстроку текста, можно использовать круглые скобки в теле шаблона. Если это более удобно, можно также использовать встроенную функцию substr. В следующем примере мы вырезаем из текстовой строки нужный нам тип изделия:

**$record = "Product number:12345**

**Product type: printer**

**Product price: $325";**

**if($record=~/Product type:\s\*([a-z]+)/i){**

**print "The product's type Is^$1.\n";**

**}**

**product's type is printer.**

1. **Вызов функций и вычисление выражений при подстановке текста**

Используя для команды **s/.../.../**модификатор е, вы тем самым показываете, что правый операнд (то есть подставляемый текст) - это то выражение perl, которое надо вычислить. Например, с помощью встроенной функции perl **uc** (**uppercase**) можно заменить все строчные буквы слов строки на заглавные:

**$text = "Now is the time.";**

**$text=~ s/(\w+)/uc($1)/ge;**

**print $text;**

**NOW IS THE TIME.**

Вместо функции **uc($l)** можно поместить произвольный код, включая вызовы программ.

1. **Поиск n-го совпадения**

С помощью модификатора **g** перебираются все вхождения заданного шаблона. Но то делать, если нужна вполне определенная точка совпадения с шаблоном, например, вторая или третья? Оператор цикла **while** в сочетании с круглыми cкобками, выделяющими нужный образец, поможет вам:

**$text = "Name:Anne Nanie:Burkart Name:Glaire Name: Dan";**

**while ($text =~ /Name: \s\*(\w+)/g){**

**++$match;**

**print "Match number $match is $1.\n";**

**}**

**Match number 1 is Anne**

**Match number 2 is Burkart**

**Match number 3 is Claire**

**Match number 4 is Dan**

Этот пример можно переписать, используя цикл **for**:

**$text = "Name:Anne Name:Burkart Name:Ciaire Name:Dan";**

**for ($match = 0;**

**$text =~ /Name:\s\*(\w+)/g;**

**print "Match number ${\match} is $1.\n")**

**{}**

**Match nuwber 1 Is Anne**

**Match number 2 is Burkart**

**Match number 3 is Claire**

**Match number 4 is Dan**

Если же вам требуется определить нужное совпадение не по номеру, а по содержанию (например, по первой букве имени пользователя), то вместо счетчика **$match** можно анализировать содержимое переменной **$1**, обновляемой при каждом найденном совпадении. Когда требуется не найти, а заменить второе или третье вхождение текста, можно применить ту же схему, использовав в качестве тела цикла выражение perl, вызываемое для вычисления заменяющей строки:

**$text = "Name:Anne Name:Burkart Name:Claire Name:Dan";**

**$match =0;**

**$text =~ s/(Name:\s\*(\w+))/ # начинается код perl**

**if (++$match == 2) # увеличить счетчик**

**{"Name:John ($2)"}# вернуть новое значение**

**else {$1} # оставить старое значение**

**/gex;**

**print $text;**

**Name:Anne Name:John (Burkart) Name:ClaireName:Dan**

В процессе глобального поиска при каждом найденном совпадении вычисляется выражение, указанное в качестве второго операнда. При его вычислении увеличивается значение счетчика, и в зависимости от него в качестве замены подставляется либо старое значение текста, либо новое. Модификатор **х** позволяет добавить в поле шаблона комментарии, делая код более прозрачным. Обратите внимание, что нам пришлось заключить весь шаблон в круглые скобки, чтобы получить значение найденного текста и подставить его на прежнее место полностью.

1. **Как ограничить "жадность" квантификаторов**

По умолчанию квантификаторы ведут себя как "жадные" объекты. Начиная с текущей позиции поиска, они захватывают самую длинную строку, которой может соответствовать регулярное выражение, стоящее перед квантификатором. Алгоритм перебора с возвратами, используемый perl, способен ограничивать аппетит квантификаторов, возвращаясь назад и уменьшая длину захваченной строки, если не удалось найти соответствия между текстом и шаблоном. Однако этот механизм не всегда работает так, как хотелось бы. Рассмотрим следующий пример. Мы хотим заменить текст "**That is**" текстом "**That's**". Однако в силу "жадности" квантификатора регулярное выражение **".\*is"** сопоставляется фрагменту текста от начала строки и до последнего найденного "**is**":

**$text = "That is some text, isn't it?";**

**$text =~ s/.\*is/That's/;**

**print $texts;**

**That'sn't it?**

Чтобы сделать квантификаторы не столь жадными, а именно заставить их захватывать минимальную строку, с которой сопоставимо регулярное выражение, после квантификатора нужно поставить вопросительный знак. Тем самым квантификаторы принимают следующий вид:

* **\*?** - ноль или несколько совпадений,
* **+?** - одно или несколько совпадений,
* **??** - ноль совпадений или одно совпадение,
* **{n}?** - ровно n совпадений,
* **{n,}?** - по крайней мере n совпадений,
* **{n,m}?** - совпадений по крайней мере n, но не более, чем m.

Обратите внимание, что смысл квантификатора от этого не меняется; меняется только поведение алгоритма поиска. Если в процессе сопоставления шаблона и текста прототип определяется однозначно, то алгоритм поиска с возвратами увеличит "жадность" такого квантификатора точно так же, как он ограничивает аппетит собрата. Однако если выбор неоднозначен, то результат поиска будет другим:

**$text = "That is some text, isn't it?";**

**$text =~ s/.\*?is/That's/;**

**print $texts;**

**That's some text, isn't it?**

1. **Как удалить ведущие и завершающие пробелы**

Чтобы отсечь от строки начальные "пробельные символы", можно использовать, следующую команду:

**$text = " Now is the time.";**

**$text =~ s/^\s+//;**

**print $texts;**

**Now is the time.**

**Чтобы отсечь "хвостовые" пробелы, годится команда:**

**$text = "Now is the time. ";**

**$text =~ s/\s+$//;**

**print $texts;**

**Now is the time.**

Чтобы отсечь и начальные, и хвостовые пробелы лучше вызвать последовательно эти две команды, чем использовать шаблон, делающий отсечение ненужных пробелов за один раз. Поскольку процедура сопоставления шаблона и текста достаточно сложна, на эту простую операцию может уйти гораздо больше времени, чем хотелось бы.

Например, в тексте нужно найти текст, находящийся между открывающим и закрывающим тегом:

**$text="<a>blah-blah</a>";**

**if($text=~m!<([a|b])>(.\*?)/\1!ig){**

**print "$2\n";**

**}**

найдет все слова, стоящие между тегами <a></a> и <b></b>.

В регулярных выражениях присутствует своя семантика: быстрота, торопливость и возврат. Если квантификатор **\*** совпадает во многих случаях, то в результате будет выведен наибольший по длине результат. Это жадность. Быстрота: поиск старается найти как можно быстрее. **"Text"=~/m\*/**, по смыслу символов m нет, но в результате будет возвращено значение 0. Т.е. формально 0 и более символов.

**$test="aaooee ooaao";**

**$test=~s/o\*/e/;**

**print $test;**

**eaaooee ooaao**

потому что 1 элемент строки - 0 и более символов.

Если добавить квантификатор **g**, то результат будет таким: **eaeaeeeeee eeaeaee**, т.к строка содержит 13 мест, где может встречается **o**, в том числе и пустых.

Модификаторы:

* **/i** игнорировать регистр;
* **/x** игнорировать пропуски в шаблоне и разрешить комментарии;
* **/g** модификатор разрешающий выполнение поиска/замены везде, где это возможно;
* **/gc** не сбрасывается позиция при неудачном поиске;
* **/s** разрешается совпадение. с **\n**, игнорируется **$\***;
* **/m** разрешить совпадение **^** и **$** для начала и конца строки во внутренних переводах строк;
* **/o** однократная компиляция;
* **/e** правая часть **s///** представляет собой выполняемый код;
* **/ee** правая часть **s///** выполняется, после чего возвращаемое значение интерпретируется снова.

при вызове **use locаle**учитываются локальные настройки. Модификатор **/g** может заполнить массив значений **@nums = m/(\d+)/g;**но это сработает для не накладывающихся совпадений. Чтобы поймать совпадения нужно воспользоваться оператором **?=...** Если ширина = 0, то механизм поиска остался на прежнем месте. Найденные данные остаются внутри скобок. Если есть модификатор **/g**, то текущая позиция остается прежней, но происходит перемещение на один символ вперед.

**$numbers="123456789";**

**@one=$numbers=~/(\d\d\d)/g;**

**@two=$numbers=~/(?=(\d\d\d))/g;**

**print "@one \n"; print "@two \n";**

Модификаторы **m** и **s** нужны для поиска последовательностей символов, содержащих перевод строки. При **s** точка совпадает с **\n** и игнорируется **$\***. **m** делает совпадающими **^** и **$** до и после **\n**. Правая часть выполняется как программный код: perl **-i -n -p -e** **'s/(.)/lc($1)/g' \*.html** приводит все литеры во всех файлах **\*.html** текущей директории к нижнему регистру.

1. **Встроенные переменные в regex.**

**$1, $2, $3, $4, ..., $n ...** содержат ссылки на найденный текст, только в том случае если regex был в круглых скобках:

**s%<f(.\*?)><(.\*?)"><(.\*?)">%$1 $2 $3%g;**

внутри regex можно использовать переменные типа **\1, \2, \3, \4, ... \n, ...**

**s/a href=(["'])(.\*?)\1>/$2/g**

найдет все урл, заключенные в двойные, одинарные и вообще без кавычек, находящиеся в документе.

для **/(a.\*b)|(mumu)/** в переменной **$+** содержится **$1** или **$2**.

**$&** содержит полный текст совпадения при последнем поиске.

**$'** и **$`** содержатся строки до и после совпадения

Если нужно скопировать и сделать подстановку, то нужно действовать примерно так:

**($at = $bt) =~ s!m(.\*?)o!! #для строк**

**for(@mass1 = @mass2){s/umka/maugli/} #для массивов**

**$u = ($m=~s/a/b/g); #поменять $m и занести в $u число замен.**

Если нужно выцепить только алфавитные символы, с учетом настроек locale, то регексп примерно такой: **/^[^\W\d\_]+$/**в нем учитываются все не алфавитные символы, не цифры и не подчеркивания (для случая "ванька-встанька"), символ отрицания в группе **[] - ^**, т.е. найти все, что не **[\W\d\_]**, можно было написать и скажем так **!~m/^(\W|\d|\_)\*/**.

Для упрощения понимания сложных регулярных выражений можно воспользоваться их комментированием. Иногда правда можно только по виду регулярного выражения определить зачем оно предназначено:

**$mmm{$1} = $2 while ($nnn =~ /^([^:]+):\s+(.\*)$/m);**

читаем регулярное выражение:

нужно найти в файле все что до двоеточия не двоеточие и все что после двоеточия(включая возможные повторения после первого **: .\*?: .\*?: .\*?:**, потому что была найдена первая позиция: выделить все что не есть двоеточие до первого двоеточия)

Что это может быть, вполне вероятно, что оно нужно для составления статистики писем, выцепление заголовка письма и его названия из mbox в хеш. По крайней мере это регулярное выражение подходит для данной задачи.

***2.7 Рабочие программы, использующие регулярные выражения***

В принципе регулярные выражения это вовсе не вещь в себе, хотя иногда и может встретится задача, фактически полностью реализуемая при помощи regex. Ниже приведены программы, иллюстрирующие использование регулярных выражений:

* + 1. ***Выделение чисел в математической записи***

Пример использования логических условий для нахождения любых чисел в том числе и в общепринятой математической записи:

**#!/usr/bin/perl**

**$\_=qq~**

**1234**

**34 -4567**

**3456**

**-0.35e-0,2**

**56grf45**

**-.034 E20**

**-.034 e2,01 -,045 e-,23**

**-,034 e201 3e-.20**

**-,045 e-,23 e-0.88**

**4 E-0.20**

**22**

**E-21**

**-0.2 w 4 3**

**345**

**2 ^-,3**

**~;**

**print "$1\n" while**

**m%(([+-]?(?=\d|[\.,]\d)\d\*([\.,]\d\*)?((\se|e|\s?\^)**

**([-+]?\d\*[,\.]?)\d+)?)|([+-]?e[+-]?\d\*[,.]?\d+))%gxi;**

программа исправно выводит все числа. Разберем регулярное выражение

**m%(([+-]?(?=\d|[\.,]\d)\d\*([\.,]\d\*)?((\se|e|\s?\^)**

**([-+]?\d\*[,\.]?)\d+)?)|([+-]?e[+-]?\d\*[,.]?\d+))%gxi;**

в переменной **$1** содержится то, что регулярное выражение находит в результате, т.е. **m%(...)%gmi. m%((что-то)|([+-]?e[+-]?\d\*[,.]?\d+))%gmi** нужно для того, чтобы находить числа вида e-20 или E21(так в математике обозначают десятку в какой-то степени, например e-0,20 = 10-0,20 или E20 = 1021). Рассмотрим левое регулярное выражение "что-то" для чисел вида неe20 или E21:

**([+-]?(?=\d|[\.,]\d)\d\*([\.,]\d\*)?((\se|e|\s?\^)([-+]?\d\*[,\.]?)\d+)?)**

**[+-]?** - есть ли в перед числом знак + или -. ? - если вообще есть что-то, находящееся внутри впереди стоящего [...]. Выкинем проверку знака, регексп сократится до

**(?=\d|[\.,]\d)\d\*([\.,]\d\*)?((\se|e|\s?\^)([-+]?\d\*[,\.]?)\d+)?**

Рассмотрим regex **(?=\d|[\.,]\d)\d\***логический оператор **(?=B)**требует, чтобы перед числом было B. В данном случае B представляет из себя regex **\d|[\.,]\d Regex \d|[\.,]\d** значит, что перед каждым числом должно быть что-то либо просто число, либо число, перед которым стоит либо запятая, либо точка, т.е. находим все числа вида ,2 .2 или просто числа 2 (2 выбрано для примера, может быть и 3). Далее скобка закрывается и идет **\d\***, т.е. число вида, 2 точно пройдет, а вот число вида ,223 не пройдет. Да и regex **(?=\d|[\.,]\d)**говорит о том, что нужно найти только одну цифру после запятой. Для остальных цифр и нужен квантификатор **\d\***, который значит любое количество цифр, в том числе и ноль, т.е. оно работает и для числе вида .2 или ,2 Далее идет регулярное выражение **([\.,]\d\*)?** которое говорит о том, есть ли вообще точка и запятая (здесь всю полную строчку в принципе можно усовершенствовать) и число \d\*(в том числе и его отсутствие, ведь квантификатор **\*** значит любой символ в том числе и ноль). Отбрасывая все что было выше от этого большого регулярного выражения остается строчка:

**((\se|e|\s?\^)([-+]?\d\*[,\.]?)\d+)?**

Эта строчка отвечает за поиск в строке **$\_** математических обозначений степеней типа e201, E,20(число в степени 0,20 например a-0,20) и т.д. но только для подстрок вида -,034 e201. Заметьте, что в конце стоит знак вопроса, т.е. если степенное обозначение вообще существует. **(\se|e|\s?\^)** есть ли числа вида -,034 e201 или -,034e201 и числа в "компьютерной" записи вида 2 ^-,3 = 2-0,3, т.е. этим регекспом мы разрешили пользователю ставить или не ставить пробел при указании степени и разрешили писать значек **^** с пробелом перед ним (если есть). Далее идет выражение **([-+]?\d\*[,\.]?)**, которое говорит о том, что степень может быть с + или - (типа e,-23 где юзер забыл поставить нолик, а на самом деле хотел написать a-0,23). Дальше идет цифра **\d\*** (а может и не идет, т.к. квантификатор то **\***). Потом идет либо точка либо запятая (причем тут негласно введено ограничение на использование запятой/точки, после e, если степень дробная или вообще есть, точка или запятая должна быть, иными словами не имеет смысла написать -2,34e-,23, хотя юзер на самом деле хотел написать число-2,34-0,23). Наконец мы добрались до конца: идет **\d+**, но тут уж, пользователь, будь добр напиши хотя бы одно число, т.к. квантификатор **+**, а не **\*** после **\d**. Т.е. наложили своего рода ограничения здравого смысла, можно просто написать 2, а можно написать и 2e,- что суть бессмыленно. И еще, **m%(что-то)%igm**стоит квантификатор **i**, который разрешает **e** быть и заглавным и квантификатор **x**, который разрешает разносить регулярное выражение на несколько строк.

Итак, регулярным выражением

**m%(([+-]?(?=\d|[\.,]\d)\d\*([\.,]\d\*)?((\se|e|\s?\^)**

**([-+]?\d\*[,\.]?)\d+)?)|([+-]?e[+-]?\d\*[,.]?\d+))%gxi;**

были предусмотрены числа степенного порядка, просто числа, числа со знаком, нецелые числа вида ,3 (которое есть 0,3 или 0.3), ошибки пользователя при вводе чисел (типа -.034 e2,01 хотя надо бы писать либо -,034 e2,01 либо -.034 e2.01 хотя по смыслу перед точками и запятыми нужно ставить нули, но мы предусмотрели и это) и числа в "компьютерном" представлении.

Конечно, данное регулярное выражение не претендует на абсолютную работу, т.к. оно успешно не работает на подстроках вида -,045 e -,23 e-0.88 считая -,045 отдельным числом, а -,23 возводит в степень e-0.88, хотя по идее должно было бы быть два числа -,045 e -,23 и e-0.88, в таком случае еще одно ограничение пользователю: если хочется, чтобы степенные числа понимались корректно (для этой программы), то нельзя ставить пробел перед степенью e.

* + 1. ***Облегчение поиска работы***

Допустим, вы оказались без работы, развалилась ваша фирма или еще какая-нибудь причина. Вам требуется найти новую. Для упрощения этой задачи есть следующий скрипт, который находит по нужной позиции (веб программирование, зарплата от 200$ и т.д.) с www.job.ru все заявки за последние 10-15 дней, точнее емайлы, куда нужно слать резюме, что значительно убыстряет поиск работы (имея базы адресов легче разослать одно и то же резюме, используя нехитрый список рассылки):

**#!/usr/bin/perl -wT**

**$url0="http://www.job.ru/cgi/list1.cgi?GR\_NUM=";**

**$url1="%31&TOPICID=9&EDUC=2&TP=&Gr=&SEX=&AGEMIN=23&AGEMAX=&MONEY=200&CDT=";**

**$url2="&LDAY=99&ADDR=%ED%CF%D3%CB%D7%C1&KWORD=&KW\_TP=AND";**

**use LWP::Simple;**

**foreach($i=1; $i<=57; $i++){#57 число листаемых страниц**

**$plus.="%31%2B";**

**$test=$url0.$plus.$url1.$url2,"\n";**

**@mass=grep{s/(.\*) ([\w+\-\.]+\@[\w\-\.]+\.\w{2,3})(.\*)/$2/ig} split /\n/, get "$test";**

**$test.=join "\n", @mass;**

**$test.=\n";**

**}**

**@un=grep{!$test{$\_}++} split /\n/, $test;**

**print join "\n", @un;**

**print "\nВы можете отправлять по вашей специальности $#un резюме\n";**

Что делает эта программа? Она составляет GET запрос из параметров, которые скрыты в hidden полях навигации по результатам запроса на www.job.ru. Программа при помощи Simple.pm отправляет запрос на сервер и как бы листает странички с поиском. Критерий ваших профессиональных навыков составлен в GET-запросе и осталось только разослать почту(для этого можно написать список рассылки) по адресам, которые выдала программа. Разберем регулярное выражение для вытаскивания почтового адреса из текущей странички

**s/(.\*) ([\w+\-\.]+\@[\w\-\.]+\.\w{2,3})(.\*)/$2/ig.**

**[\w+\-\.]\@** - найти все, что содержит буквы, тире и точки до символа **@**, ведь почтовый адрес по спецификации может быть вида aa.ss-ss@chto-to.ru. То же самое после символа **@ -[\w\-\.]+** далее может быть точка **\.** и любая буква от 2 до 3 символов **\w{2,3}**, т.е. окончание, самый верхний домен .com, .ru, .cz и т.д. Далее регулярное выражение состоит из трех классов скобок**(.\*)** - переменная **$1**, **([\w+\-\.]+\@[\w\-\.]+\.\w{2,3})** переменная **$2** и все остальное в **(.\*)** - **$3**. Пробел перед **$2** стоит потому, что так устроен html, отдаваемый пользователю поиском по базе предложений о работе www.job.ru. Нам нужно содержимое **$2**, в котором находится e-mail работодателя. Пишем его во вторую часть **s/** наш **regex/$2/ig**. Квантификатор **i** нужен для того, чтобы не различать регисты Vasya@pupkin.ru и vasya@pupkin.ru, квантиикатор **g** задействова на тот случай, если работодатель указывает 2 адреса, по которым нужно высылать резюме. На 23 августа 2001 года на 20 часов 10 минут программа выдала 410 e-mail адресов (пролистав за 3-4 минуты 57 страниц), где вас ждут, как потенциального сотрудника.

Остается написать скрипт почтовой рассылки по e-mails, выданным данным скриптом. Но это в другой главе.

Примером выше был получен список email адресов. Теперь необходимо проверить, действительно ли существуют домены, на которых заведены такие пользователи (примитивная - но проверка).

**#!/usr/bin/perl**

**use Socket; #загрузить inet\_addr**

**s{ #**

**( #Сохранить имя хоста в $1**

**(?: #Группирующие скобки**

**(?! [-\_] ) #ни подчеркивание, ни дефис**

**[\w-] + #кусок имени хоста**

**\. #и точка домена**

**)+ #повторить несколько раз**

**[A-Za-z] #следующий символ - буква**

**[\w-]+ #домен верхнего уровня**

**) #конец записи $1**

**}{ #Заменить следующим:**

**"$1" . #исходн часть + пробел**

**(($addr = gethostbyname($1)) #Если имеется адрес**

**? "[" . inet\_ntoa($addr). "]"#отформатировать**

**: "[???]" #иначе пометить как сомнительный**

**)**

**}gex**

Переписываем исходную программу с учетом вышеприведенного кода:

**#!/usr/bin/perl -wT**

**$url0="http://www.job.ru/cgi/list1.cgi?GR\_NUM=";**

**$url1="%31&TOPICID=9&EDUC=2&TP=&Gr=&SEX=&AGEMIN=23&AGEMAX=&MONEY=200&CDT=";**

**$url2="&LDAY=99&ADDR=%ED%CF%D3%CB%D7%C1&KWORD=&KW\_TP=AND";**

**use Socket;**

**use LWP::Simple;**

**foreach($i=1; $i<=57; $i++){**

**$plus.="%31%2B";**

**$test=$url0.$plus.$url1.$url2,"\n";**

**@mass=grep{s/(.\*) ([\w+\-\.]+\@[\w+\-\.]+\.\w{2,3})(.\*)/$2/ig} split /\n/, get "$test";**

**$test1.=join "\n", @mass;**

**$test1.="\n";**

**}**

**@res=split /\n/, $test1;**

**@un=grep{!$test{$\_}++} @res;**

**foreach $file(@un){**

**$file=~s/(.\*)\@(.\*)/www\.$2/;**

**=pod**

**$file=~s{((?:(?![-\_])[\w-]+\.)+[A-Za-z][\w-]+)}**

**{"$1".(($addr=gethostbyname($1))?"[".inet\_ntoa($addr)."]":"[???]")}gex;**

**print $file,"\n" if($file !~/\?\?\?/);**

**=cut**

**$file=~s{**

**(**

**(?:**

**(?![-\_])**

**[\w-]+**

**\.**

**)+**

**[A-Za-z]**

**[\w\_]+**

**)**

**}{**

**"$1".**

**(($addr = gethostbyname($1))**

**? "[".inet\_ntoa($addr)."]"**

**: "[???]"**

**)**

**}gex;**

**print $file,"\n" if($file !~/\?\?\?/);**

**}**

Между строчками можно комментировать целые куски кода.

**=pod**

**$file=~s{((?:(?![-\_])[\w-]+\.)+[A-Za-z][\w-]+)}**

**{"$1".(($site=gethostbyname($1))?"[".inet\_ntoa($site)."]":"[???]")}gex;**

**print $file,"\n" if($file !~/\?\?\?/);**

**=cut**

Эта программа успешно удалила некоторые из адресов, которые Socket.pm показались подозрительными. Если научится читать сложные регулярные выражения, то можно написать полный регексп е-mail адресов, который занимается тем, что выделяет адреса в точности с соответствующим RFC (занимает это регулярное выражение несколько страниц).

Ключи, которые использовались в вышеприведенном регулярном выражении

**g** - глобальная замена

**е** – выполнение

**x** - улучшенное форматирование.

Если написать это регулярное выражение в одну строчку, то оно вряд ли там поместится:

**s{((?:(?![-\_])[\w-]+\.)+[A-Za-z][\w-])}#здесь силовой перевод каретки**

**{"$1".(($addr=gethostbyname($1))?"[".inet\_ntoa($addr)."]":"[???]")}gex**

Разберем один интересный момент в данном регекспе:

**s/regex/условие?да:иначе/**

Тут проявляется, пожалуй, одна из действительно сильнейших особенностей regex, возможность в одном регулярном выражении избежать многострочных условий с циклом. В приведенном примере работает все примерно так: Если **$addr=gethostbyname($1)** - да, то ставить ip-адрес **(inet\_ntoa($addr))**, если нет (не откликнулся сервер, сбой на линии и т. д.) то отметить этот урл как подозрительный **[???]**. В принципе в программе ничего человеку делать не нужно, т.к. подозрительные отметаются условием **print $file,"\n" if($file !~/\?\?\?/);** Общее время работы программы 10-15 минут.

* + 1. ***Простое решение для зеркала новостной ленты***

Допустим, нужно сделать зеркало какой-либо зарубежной новостной ленты вместе с загрузкой картинок с удаленного сервера, чтобы не ждать по несколько минут отображения содержимого полностью загруженной большой таблицы. Приведенный скрипт запускается при помощи crontab каждые 5 часов:

**#!/usr/bin/perl -w**

**$/="\001";**

**print "content-type: text/html\n\n";**

**$dir="/var/www/docs/html/news/images";**

**$imgurl="http://www.qwerty.ru/news/images";**

**use LWP::Simple;**

**use LWP::UserAgent;**

**$page=get "http://www.astronomynow.com";**

**$page=~s/face="(.\*?)"//igs;**

**&getimg($page);**

**$page=~s!/images/grafix/listdot.gif!../../listdot.gif!igs;**

**$page=~s!/images/grafix/spacer.gif!../../spacer.gif!igs;**

**$page=~s!images/grafix/spacer.gif!../../spacer.gif!igs;**

**if($page=~m!<TABLE WIDTH="400" BORDER="0" CELLPADDING="0" CELLSPACING="0">(.\*?)</TD></TR></TABLE>!igsm){**

**$file=$1;**

**&getlink($page);**

**foreach $names(@res){**

**$names=~s|.\*/||ig;**

**$file=~s|src="http://(.\*?)$names"|src=$imgurl/$names|igs;**

**}**

**$html=qq~**

**<TABLE BORDER="0" CELLPADDING="0"**

**CELLSPACING="0">**

**$file**

**</TD></TR></TABLE>~;**

**}**

**open F, ">$dir/news.txt";**

**print F $html or die "\n\n\n ERROR: $!\n\n\n";**

**close F;**

**sub getimg{**

**&getlink($\_[0]);**

**foreach $img(@res){**

**my $res = LWP::UserAgent->new->request(new HTTP::Request GET => $img);**

**if ($res->is\_success) {**

**$img=~s|.\*/||;**

**open (ABC, ">$dir/$img") or die "\n\n\nERROR: $!\n\n\n";**

**binmode(ABC);**

**print ABC $res->content; close ABC or die "\n\n\nERROR: $!\n\n\n";**

**} else {**

**print $res->status\_line;**

**}**

**}**

**return @res;**

**}**

**sub getlink{**

**local $\_=$\_[0];**

**push(@res, "http://$2")**

**while m{SRC\s\*=\s\*(["'])http://(.\*?)\1\s\*(.\*?)WIDTH="100" HEIGHT="100"(.\*?)>}igs;**

**return @res;**

**}**

* + 1. ***Вывод результатов поиска***

Предположим, есть необходимость подсветить результаты поиска в файлах, подобно тому, как это делает поисковик **aport**. Данное регулярное выражение позволяет реализовать эту красивую функцию для поисковика, но оно имеет очень большой минус, при обработке текста машина начинает тормозить, но мы рассмотрим этот регексп из общих соображений:

**$sn=4;**

**{**

**local $\_=$description1;**

**print "...$1<font color=red>$3</font>$4..."**

**while(m/(([\s,\.\n^]\*\w\*){$sn})(\s\*$query\s\*)(([\s,\.\n^]\*\w+){$sn})/ig);**

**}**

**$\_="";**

Исходная задача состоит в следующем: вывести по 4 слова спереди и сзади результата поиска, причем так, чтобы если слово находится первым, то будет видно 4 слова позади него. В точности такое же условие и для последнего слова.

Соответственно из вида регекспа понятно, что разделителями слов могут быть символы **[\s,\.\n^]\***, в том числе и символ перевода каретки **^**. Комбинация **(\d\d\d){$sn}** значит, что нужно найти 3 цифры три раза.

**3. Задания для выполнения**

Разработать perl-скрипты в соответствии с вариантом задания. Для второго задания из каждого варианта осуществить вывод результата в файл.

**4. Варианты индивидуальных заданий.**

1. Задание 1. Даты три номерных знака автомашин. Найти номерной знак, содержащий буквы "МОН" и вывести его на печать. Если такого знака среди заданных нет, то напечатать соответствующее сообщение.

Задание 2. Даны x1, x2, …, x10. Определить:



1. Задание 1. Даны три марки автомашин. Определить, есть ли среди них марка "ВАЗ2101". Вывести соответствующее сообщение.

Задание 2. Даны y1, y2, …, y9. Определить:



1. Задание 1. Даны четыре слова одинаковой длины. Напечатать сообщение о наличии или отсутствии одинаковых слов и это слово.

Задание 2. Даны x1, x2, …, x5 ; y1, y2, …, y8. Определить:



1. Задание 1. Даны четыре фамилии. Определить, есть ли среди них фамилия ИВАНОВ. Напечатать соответствующее сообщение.

Задание 2. Даны массивы L1, …, L7 и Y1, …, Y5. Определить:



1. Задание 1. Даны наименования трёх вузов. Определить, есть ли среди них МАДИ. Напечатать соответствующее сообщение.

Задание 2. Даны α1, α2, …, α8; γ1, γ2, …, γ5. Определить:



1. Задание 1. Даны наименования четырех факультетов. Определить, имеется ли среди них ДСФ и вывести об этом соответствующее сообщение.

Задание 2. Даны x1, x2, …, x8 ; y1, y2, …, y8. Определить:



1. Задание 1. Даны три слова. Определить и вывести слова, которые состоят из пяти букв.

Задание 2. Даны а1, а2, …, а6. Определить:



1. Задание 1. Даны три слога, каждый из двух букв и слово из 6 букв. Составить из слогов возможные слова и определить, получается ли заданное слово. Вывести соответствующее сообщение.

Задание 2. Даны x1, x2, …, x10; y1, y2, …, y5. Определить:



1. Задание 1. Дано слово из пяти букв. Сколько раз встречаются идущие подряд буквы "НН"?

Задание 2. Даны x1, x2, …, x10; l1, l2, …, l10. Определить:



1. Задание 1. Даны номерные знаки 4 автомашин. Определить, имеются ли среди них одинаковые, вывести их или сообщение, что таких - нет.

Задание 2. Даны l1, l2, …, l8; n1, n2, …, n5; m1, m2, …, m5. Определить:



1. Задание 1. Даны номерные знаки трёх автомашин. Определить, есть ли среди них знак   
   "МНЗ 2754" и напечатать об этом сообщение.

Задание 2. Даны m1, m2, …, m7. Определить:



1. Задание 1. Даны три фамилии Р1, Р2, Р3 и три имени М1, М2, М3 соответственно фамилиям. Определить, есть ли среди них фамилия и имя Р4 М4.

Задание 2. Даны β1, β2, …, β8; γ1, γ2, …, γ7 . Определить:



1. Задание 1. Даны три английских слова А1, А2, А3 и их русских перевод Р1, Р2, Р3. Напечатать перевод заданного английского слова А4 (А4 = А1 или А2 или А3).

Задание 2. Даны x1, …, x5; y1, …, y5; z1, …, z5. Определить:



1. Задание 1. Дано слово длиной в 4 символа. Определить, является ли первый или последний символ слова буквой "А". Напечатать соответствующие сообщения.

Задание 2. Даны x1, …, x10; α1, …, α10; a, b. Определить:



1. Задание 1. Даны три одинаковых слова, но в одном из них может быть допущена ошибка. Вывести соответствующее сообщение и слово с ошибкой.

Задание 2. Даны x1, …, x7; *l*1, …, *l*5; a, b. Определить:



1. Задание 1. Даны три фамилии. Определить, есть ли среди них фамилии, начинающиеся на букву 'М’. Вывести найденные фамилии. Если таковых нет, вывести соответствующее сообщение,

Задание 2. Даны С1, … , С9 ; d1, … , d9 ; f. Определить:



.

1. Задание 1. Даны три слова одинаковой длины, начинающиеся на буквы 'А', 'B', 'K' в любом порядке следования. Вывести их в алфавит­ном порядке.

Задание 2. Даны b1, ... , b10; d1, ... , d7; A. Определить:

.

1. Задание 1. Даны четыре слова. Напечатать слово, имеющее максималь­ную длину.

Задание 2. Даны массивы α1, … , α8 ; β1 , … , β8. Определить:



;  .

1. Задание 1. Дано слово длиной 8 символов. Определить, содержит ли оно слог "кн". Вывести это слово или сообщение.

Задание 2. Даны α1, ... , α10 ; b. Определить:

S =  ; di = S – (αi - b).

1. Задание 1. Дано слово длиной пять символов. Определить, есть ли в нём буква 'М’ или буква ‘Н’. Вывести соответствующее сообщение.

Задание 2. Даны a1, ... , a12 ; b1, ... , b10. Определить:

S1 =



;

.

S2 =



1. Задание 1. Задана запись одного оператора. Проверить наличие симво­ла точка с запятой. Вывести соответствующие сообщения.

Задание 2. Даны массивы R1, ... , R9 ; f1, ... , f9 ; d1, ... , d7. Определить :

Si =



1. Задание 1. Дано слово из четырех символов. Сколько раз встречается в нем заданный символ? Вывести соответствующее сообщение.

Задание 2. Даны массивы d1, ... , d12 ; f1, ... , f8 и переменная Т. Определить:



Ki = S – di .

S =



;

1. Задание 1. Предложение описано символьной переменной заданной длиной. Определить, встречается ли запятая?

Задание 2. Даны массивы P1, ... , P12 ; q1, ... , q12.Определить:



di =

.

1. Задание 1. Даны два слова, одинаковых по значению, в одном из них сделана ошибка. Определить, в какой позиции ошибочный символ.

Задание 2. Даны массивы f1, … , f14 ; l1, … , l7.

Определить



.



;

S1 =

1. Задание 1. В строке символов определить наличие гласных букв.

Задание 2. Даны массивы t1, … , t20; V1, ... , V20. Определить:

; ; .

1. Задание 1. Дан номерной знак автомобиля в виде строки символов. Опре­делить, имеется ли в нём сочетание цифр 92. Выдать соответ­ствующее сообщение.

Задание 2. Даны массивы α1, ... , α15 ; γ1, ... ,γ7.

Определить



1. Задание 1. Даны три фамилии. Определить фамилии, начинающиеся с гласной буквы. Сделать соответствующее сообщение.

Задание 2. Дан массив K1, ... , K10. Определить:

.

1. Задание 1. В слове из четырех букв определить номер позиции, в которой находится буква «а» (два способа!).

Задание 2. Дан массив M1, ... , M20.Определить:



лАБОРАТОРНАЯ РАБОТА №5

сервер сценариев Windows Scripting Host. методы объектов wscriptshell и wshnetwork

**1. Цель работы**

Овладеть навыками работы сервером сценариев Windows, изучить возможности сервера сценариев, режимы выполнения сценариев и методы объекта WscriptShell иWshNetwork.

**2. Общие сведения**

***2.1 Основные положения***

Долгое время для выполнения однотипных задач в среде Windows и DOS служили командные (пакетные) BAT-файлы. Основным их недостатком были примитивный DOS-интерфейс – отсутствие интерактивности, и довольно ограниченные возможности по работе с WINDOWS (трудность работы в сети, с ярлыками, с реестром и т.д.).

Ситуация изменилась, когда Microsoft разработала Сервер Сценариев (Windows Scripting Host), который должен служить для автоматизации работы с повторяющимися процессами. Сам Windows Scripting Host не является языком как таковым, он только представляет свойства и методы для работы в Windows, которые могут использоваться другими языками сценариев. Наиболее удобными и предназначенными для этого явились ранее разработанные самой Microsoft языки сценариев Visual Basic Scripting Edition (VBScript) и JScript.

Раньше языки VBScript и JScript по своим возможностям были очень близки к Visual Basic for Applications - они также могли быть вызваны только из MS Internet Explorer и нескольких других программ Microsoft, которые их поддерживали.

С появлением Windows Scripting Host появилась возможность создавать для них отдельные сценарии, которые можно запускать и без Internet Explorer.

Также преимуществом Windows Scripting Host является то, что для запуска сценариев требуется мало памяти и то, что файлы сценариев могут быть практически любого размера (содержать десятки тысяч строк).

Еще одним преимуществом Windows Scripting Host является отсутствие среды разработки - не нужны компиляторы, редактирование сценариев может производиться в любом текстовом редакторе, способном работать с текстовыми файлами.

Сервер сценариев предназначен для автоматизации повторяющихся задач и во многом, по сравнению с обычными языками программирования, обладает достаточно скромными возможностями. Но по сравнению с пакетными файлами DOS он обладает более широкими возможностями. Такими как:

- вывод сообщений на экран;

- запуск других программ;

- работать с сетевыми дисками;

- устанавливать принтеры;

- работать с переменными окружения;

- работать с реестром.

В Windows 2000 и последующих версиях Windows Scripting Host установлен по умолчанию. Отключить его использование можно только удалением ассоциаций с его файлами. Установленный Windows Script Host поддерживает несколько видов файлов: vbs, vbe, js, jse, wsf, wsc и wsh. Все они (кроме vbe и jse) являются простыми текстовыми файлами и могут редактироваться в любом текстовом редакторе.

Файлы .vbs и .js являются файлами, написанными на языке сценариев MS Visual Basic Script и MS JScript соответственно.

Файлы vbe и jse– это vbs и js-файлы зашифрованные с помощью программы MS Script Encoder.

Файлы с расширением .wsf – это файлы, содержащие XML-разметку для работы с WSH.

Файлы wsc - Windows Script Components (WSC) позволяют упаковывать сценарии для использования их в качестве СОМ-компонентов. По сути, это те же wsf-файлы, еще и содержащие COM-компоненты.

Файлы wsh являются файлами настроек Сервера Сценариев.

***2.1.1 Запуск сценариев (WScript.exe и CScript.exe)***

Для запуска сценариев, в составе Windows Scripting Host служат файлы WScript.exe (диалоговый режим) и CScript.exe (режим командной строки). Они находятся в каталоге C:\WINDOWS\system32\.

WScript.exe служит для запуска сценариев из Windows. Используя его, вы можете запускать сценарии подобно обычным приложениям Windows. Вот несколько способов:

1. Запускать сценарий, как обычное приложение двойным щелчком мыши, выделить и нажать Enter и т.д.

2. Ввести имя файла сценария и путь к нему в окне в окне “Выполнить” (RUN) меню “Пуск” (Start).

3. Ввести в строку окна “Выполнить” WScript.exe и имя файла сценария (с указанием пути к нему). При этом вы можете использовать параметры запуска WScript.exe.

CScript.exe - это версия Windows Scripting Host, которая используется для запуска сценариев из командной строки.

***Синтаксис:*** CScript [параметры] имя\_файла.расширение [аргументы]

Для запуска сценариев, как с помощью CScript.exe так и с WScript.exe, можно использовать следующие параметры командной строки:

|  |  |  |
| --- | --- | --- |
| Параметр | Версия WSH | Описание |
| //B | 1.0 | Пакетный режим (подавляется вывод информации, запросов и сообщений об ошибках) |
| //D | 2.0 | Включить активную отладку |
| //E:*язык* | 2.0 | Указать язык сценария для исполнения файла |
| //H:CScript | 1.0 | Заменить исполняемый сервер сценариев на CScript.exe |
| //H:WScript | 1.0 | Заменить исполняемый сервер сценариев на WScript.exe |
| //I | 1.0 | Диалоговый режим (противоположный //B) (по умолчанию) |
| //Job:xxxx | 2.0 | Выполнить задание xxxx WSF-файла |
| //Logo | 1.0 | Отображать заставку (по умолчанию) |
| //Nologo | 1.0 | Не выводить заставку |
| //S | 1.0 | Запомнить параметры текущей командной строки для данного пользователя |
| //T:nn | 1.0 | Задать время исполнения сценария в секундах |
| //X | 2.0 | Выполнить сценарий под отладчиком |
| //U | 2.0 | Применять кодировку Unicode для перенаправленного консольного ввода-вывода |

*Например:*

Cscript //T:10 Пример1.vbs

Отвести на выполнение сценария Пример1.vbs 10 секунд времени.

Попробуем сделать используемым сервером сценариев по умолчанию CScript.exe.

Для этого введем в командной строке: wscript //h:cscript

С этого момента, все сценарии будут запускаться в режиме командной строки. Для многих сценариев выводящих информацию это будет несколько неудобно.

Попробуем снова запустить сценарий VBScript Пример1.vbs.

Прочитать надпись практически невозможно, поскольку окно быстро “промелькнет” на экране.

Чтобы снова сделать сервером по умолчанию WScript.exe, введем в командную строку:

*wscript //h:wscript*

И снова запускаемые файлы сценариев по умолчанию запускаются в диалоговом режиме.

Какой способ лучше, нельзя сказать однозначно. У каждого есть свои плюсы и минусы. В пользу WScript.exe говорит его интерактивность. В пользу CScript.exe – незаметность для пользователя. Хотя для незаметности лучше всего включить пакетный режим с параметром //b.

***2.1.2 Объекты Windows Scripting Host***

В работе WSH используются 9 объектов: WScript (не путать с WScript.exe), WshArguments, WshEnvironment, WshNetwork, WshShell, WshShortcut, WshSpecialFolders, WshUrlShortcut и FileSystemObject.

Объект **WScript** является главным объектом **Windows Script Host**. Он служит для создания объектов и выполняет служебные задачи, связанные с ними, содержит сведения о сервере сценариев и о запущенных сценариях.

Объект **WshArguments** служит для работы с аргументами окружения

**WshEnvironment** – работает переменными окружения.

**WshNetwork** – используется при работе с сетевым окружением: содержит информацию для сети о данном компьютере, позволяет подключать сетевые принтеры и диски.

**WshShell** – служит для работы с переменными окружения Windows, запускает другие программы, работает с реестром и т.д.

**WshShortcut** – создает ярлыки.

**WshSpecialFolders** – используется для доступа к специальным папкам Windows, таким как меню Пуск, Рабочий стол, Мои документы и т.д.

**WshUrlShortcut** – еще один объект для создания ссылок, но обладающий более ограниченными возможностями, чем WshShortcut.

Особняком стоит **FileSystemObject** объект. Как таковой он не является объектом WSH и дочерним объектом **WScript**, но занимает важное место в создании сценариев используясь для работы с файлами.

Объект **TextStream** используется для работы с содержанием текстовых файлов.

Из всех вышеперечисленных объектов можно выделить 4 главных: **WScript**, **WshShell**, **WshNetwork** и **FileSystemObject**. Остальные же являются их объектами, созданными для удобства, дублируя некоторые их возможности.

Перед использованием всех объектов (кроме **WScript**) нужно создать их экземпляр. Для этого используется метод **CreateObject**, объекта **WScript**.

Например, объект **WshShell** создается следующим образом:

Set WshShell = CreateObject(“WScript.Shell”)

**Свойства объекта WScript**

Объект **WScipt** содержит информацию о сервере сценариев и о самих, исполняемых файлах сценариев.

**Name** – выводит надпись: “Сервер сценариев”

Пример

*WScript.Echo WScript.Name*

**FullName** – возвращает используемый сервер сценариев (CScript.exe или WScript.exe) и полный путь к нему.

Результат будет типа:

C:\WINDOWS\WSCRIPT.EXE

**Path** – возвращает путь к папке с файлами сервера сценариев (CScript.exe и WScript.exe).

Если Windows находится в папке Windows, то результат будет:

C:\WINDOWS

**Version** – показывает версию установленного сервера сценариев. Обратите внимание, что свойство Version возвращает не версию языка Windows Script Host, а версию его интерпретатора.

Например

WScript.Echo WScript.Version

Выдаст результат: (5.0 или, 5.1), где 5.0 соответствует версии Windows Scriptinh Host 1.0, а 5.1. версии 2.0.

**ScriptName** – выдает имя исполняемого файла сценария.

**Timeout**

Свойство **Timeout** устанавливает время, по истечении которого сценарий завершает свою работу.

Пример

WScript.Timeout = 5

WScript.Echo "Сценарий завершит работу через 5 секунд"

**SrdErr**, **StdIn**, **StdOut** - методы, предназначенные для ввода-вывода информации в режиме командной строки (CScript.exe).

**Interactive**

Свойство **Interactive** показывает, используется ли диалоговый режим (WScript.exe), возвращая логический результат. А также может устанавливать или отключать диалоговый режим. При значении false - интерактивный режим отключается, т.е. диалоговые окна не могут использоваться.

Синтаксис:

*WScript.Interactive[ = True|False]*

**Методы объекта WScript**

**CreateObject** – создает экземпляр объекта ActiveX.

Синтаксис:

*object.CreateObject(strProgID[,strPrefix])*

где

**object** – объект WScript.

**StrProgID** – класс к которому принадлежит объект.

Например, создадим объект **WshShell**

Set WshShell = WScript.CreateObject(“WScript.Shell”)

**ConnectObject** – позволяет подключить исполняемый сценарий к существующему объекту, его событиям.

**DisconnectObject** – отключается от объекта, с которым был соединен сценарий методом ConnectObject.

**GetObject** – получает объект, который уже создан и находится в другом файле.

**Echo** – выводит диалоговое окно с сообщением пользователю. При использовании CScript.exe выводит строку с текстом.

**Sleep** – переводит сценарий в неактивное состояние, на заданное время (в миллисекундах), после чего продолжает его работу.

**Quit** – завершает работу сценария. Необязателен.

Windows Scripting Host имеет два вида диалоговых окон: простое (метод **Echo**) и управляющее (метод **Popup**)

**Echo**

Метод **Echo** объекта **WScript** отображает сообщение в диалоговом окне, если используется WScript.exe, или выводит строку с текстом, если используется CScript.exe, по своим возможностям дублируя команду echo bat-файлов.

Синтаксис следующий:

*object.Echo [[Arg1] [,Arg2] [,Arg3] ... ]*

где

*object* - объект WScript.

*Arg1, Arg2, Arg3 ...* – данные, которые должны быть выведены на экран

Для перевода строки используется константа vbCrLf.

Например

Wscript.Echo "Это", vbCrLf, "пример"

Wscript.Echo "Это"& vbCrLf& "пример"

**Popup**

Управляющее окно **Popup** имеет те же возможности вывода информации, что и окно созданное с помощью метода **Echo**, но вдобавок ко всему обладает дополнительными возможностями, расширяющими его возможности и сферу его применения. Метод **Popup**, является методом объекта **WshShell**, и для его использования должен быть создан объект **WshShell**.

Синтаксис:

*intButton = object.Popup(strText,[WaitSec],[strTitle],[natType])*

где

*object* - объект WshShell

*strText* - само сообщение в данном окне

*WaitSec* - время (в секундах), по истечении которого окно закроется

*strTitle* - заголовок окна. Если отсутствует, то заголовок окна будет по умолчанию "Сервер сценариев".

*natType* - параметр определяющий картинку и кнопку в данном окне.

|  |  |
| --- | --- |
| Значение | Кнопки |
| 0 | OK |
| 1 | OK и Отмена(Cancel) |
| 2 | Стоп(Abort), Повтор(Retry), и Пропустить(Ignore) |
| 3 | Да(Yes), Нет(No), и Отмена(Cancel) |
| 4 | Да(Yes) и Нет(No) |
| 5 | Повтор(Retry) и Отмена(Cancel) |

Параметры определяющие рисунок:

|  |  |
| --- | --- |
| Значение | Рисунок |
| 16 | Важное сообщение |
| 32 | Вопрос |
| 48 | Предупреждение |
| 64 | Информация |

При закрытии окно popup возвращает значение intButton, которое содержит информацию о том, какая кнопка была нажата.

Значения выбранных кнопок:

|  |  |
| --- | --- |
| Значения | Нажатая кнопка |
| 1 | OK |
| 2 | Отмена (Cancel) |
| 3 | Стоп (Abort) |
| 4 | Повтор (Retry) |
| 5 | Пропустить (Ignore) |
| 6 | Да (Yes) |
| 7 | Нет (No) |

Интересной особенностью окна popup является его возможность закрываться по истечении заданного времени.

Dim Interval, WshShell

‘устанавливаем время через которое окно закроется:

Interval = 5

‘создаем объект WshShell для метода Popup:

Set WshShell = CreateObject(“WScript.Shell”)

WshShell.Popup “Это окно закроется через 5 секунд”,Interval,\_

“Окно Popup”, 48

В сценариях VBScript, кроме использования диалоговых окон Windows Script Host может использовать свои собственные диалоговые окна: MsgBox (окно вывода информации) и InputBox (окно ввода информации). Параметры, используемые в данных функциях, аналогичны применяемым в языке VBA.

***2.2 Методы и свойства объекта WshShell***

В работе Объект **WshShell** служит для работы с переменными окружения Windows, специальными папками, запускает другие программы, создает ярлыки и т.д. Для его использования его необходимо создать методом CreateObject.

Set WshShell = CreateObject(“WScript.Shell”)

***2.2.1 Информация о системе***

Объект **WshShell** Свойство Environment объекта WshShell, позволяет работать с системными переменными окружения. Для этого создается объект WshEnvironment

Синтаксис:

*WshShell.Environment([strType])*

где

*strType* – может принимать параметры "System", "User", "Volatile" или "Process" .

Значение “Volatile" используется для работы с данными, передаваемыми другими программами, а "System", "User" и "Process" используются для работы с системными данными.

Таблица системных данных среды, доступных с помощью значений "System", "User" и "Process"

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Значение | Описание | Где присутствует | | |
| System | ыser | Process |
| NUMBER\_OF\_PROCESSORS | Количество процессоров на данном компьютере | X | - | X |
| PROCESSOR\_ARCHITECTURE | Тип процессора | X | - | X |
| PROCESSOR\_IDENTIFIER | Расширенные данные о процессоре | X | - | X |
| PROCESSOR\_LEVEL | Поколение процессора | X | - | X |
| OS | Операционная система | X | - | X |
| COMSPEC | Путь к файлу командной строки (cmd.exe или command.com) | X | - | X |
| HOMEPATH | Каталог по умолчанию для пользователей |  |  | X |
| HOMEDRIVE | Первый локальный диск (обычно С:) | - | - | X |
| PATH | Заданные системные пути | X | X | X |
| PATHEXT | Исполняемые файлы (.exe , .com и т.д.) | X | - | X |
| SYSTEMDRIVE | Диск на котором находится директория с операционной системой | - | - | X |
| SYSTEMROOT | Директория Windows | - | - | X |
| WINDIR | Директория Windows | X | - | X |
| TEMP или TMP | Папка для временных файлов | - | X | X |

Объект Environment, как и все коллекции WSH, имеет свойство Count, в котором хранится число элементов коллекции, и метод Item.

Следующий пример выводит на экран некоторые системные данные VBScript:

'создаем объект WshShell:

Set WshShell = WScript.CreateObject("WScript.Shell")

'создаем объект Environment со значением PROCESS:

Set WshProEnv = WshShell.Environment("PROCESS")

SysInfo = "Системные параметры компьютера:" + Chr(10)+ Chr(10)

SysInfo = SysInfo + "Процессоров: " + \_

WshProEnv("NUMBER\_OF\_PROCESSORS") + Chr(10)

SysInfo = SysInfo + "Директория Windows: " + \_

WshProEnv("WINDIR") + Chr(10)

SysInfo = SysInfo + "Временная папка: " + WshProEnv("TEMP") + Chr(10)

MsgBox SysInfo

***2.2.2 Запуск приложений***

С помощью метода **Run** объекта **WshShell** мы можем запускать другие приложения.

Синтаксис:

*object.Run (strCommand, [intWindowStyle], [bWaitOnReturn])*

где

*object* - объект **WshShell**

*strCommand* - исполняемая команда

**Особенностью синтаксиса данного метода является то, что при использовании всех параметров скобки ( ) опускаются.**

Необязательные параметры:

*intWindowStyle* - стиль окна запускаемого приложения

bWaitOnReturn - может принимать значение **true** или **false**, указывающее, следует ли сценарию дожидаться окончания выполнения запущенного приложения. **true** - выполнение сценария приостанавливается до тех пор, пока не будет закрыто запущенное приложение. **false** - не дожидаться.

Значения, принимаемые параметром *intWindowStyle*:

|  |  |
| --- | --- |
| Значение | Описание |
| 0 | Запуск в скрытом виде |
| 1 | Обычный размер окна, если окно свернуто или развернуто на весь экран, то ему возвращается исходный вид и положение на экране |
| 2 | Запуск в свернутом виде |
| 3 | Развернутое на весь экран |
| 4 | Запуск в обычном размере, в неактивном состоянии (без фокуса) |
| 8 | Обычный размер в неактивном состоянии, но в фокусе остается запустившее приложение |

**Метод Run не может вызвать приложение, если оно расположено в папке, путь к которой содержит пробелы. В этом случае путь надо заключить в тройные кавычки.**

В качестве примера, с помощью метода Run запустим программу Блокнот, с загруженным вызывающим сценарием.

Dim WshShell

'Создаем объект WshShell:

Set WshShell = WScript.CreateObject("WScript.Shell")

'Вызываем Блокнот и добавляем к нему в качестве параметра

'путь исполняемого сценария:

WshShell.Run ("%windir%\notepad " & WScript.ScriptFullName)

***2.3 Специальные папки Windows***

Для получения пути к специальным папкам Windows типа Мои документы, Рабочий стол и т.д. используется свойство **SpecialFolders** объекта **WshShell**.

Синтаксис:

*object.SpecialFolders(objWshSpecialFolders)*

где

*object* - объект WshShell;

*objWshSpecialFolders* – специальная папка.

|  |  |
| --- | --- |
| *objWshSpecialFolders* | Название папки |
| Desktop | Рабочий стол |
| Favorites | Избранное |
| Fonts | Шрифты |
| MyDocuments | Мои документы |
| NetHood | Пустая папка, используемая в качестве шаблона для сетевой среды. |
| PrintHood | Принтеры |
| Programs | Программы, меню “Пуск” |
| Recent | Раздел просмотренных документов, меню “Пуск” |
| SendTo | Отправить |
| StartMenu | Меню “Пуск” |
| Startup | Автозагрузка |
| Templates | Шаблоны |

Кроме вышеперечисленных папок в Windows 2000 доступны:

AllUsersDesktop, AllUsersStartMenu, AllUsersPrograms, и AllUsersStartup.

Для перебора всех элементов коллекции можно применять операторы цикла For или For Each ...In, указав имя коллекции WshShell.SpecialFolders. Свойство Count содержит количество элементов в коллекции.

В качестве примера составим сценарий, узнающий адрес папки “Рабочего стола”.

Dim WshShell, DesktopPath

Set WshShell = CreateObject("WScript.Shell")

'получаем путь к рабочему столу:

DesktopPath = WshShell.SpecialFolders("Desktop")

MsgBox "Адрес рабочего стола: " + DesktopPat

***2.4 Создание ярлыков***

Для создания ярлыков используется метод **CreateShortcut**

Синтаксис:

*object.CreateShortcut(strPathname)*

где

*object* - объект WshShell

*strPathname* - имя и полный путь к ресурсу, для которого создается ярлык (например, это может быть файл, интернет-адрес и т.д.).

Для демонстрации метода **CreateShortcut** создадим на нашем рабочем столе ярлык блокнота.

В самом процессе создания ярлыка можно выделить несколько этапов:

1. Собственно создание ярлыка.

2. Установка его свойств.

3. Сохранения ярлыка с установленными свойствами

Начнем с первого пункта создадим ярлык на рабочем столе

Dim WshShell, DesktopPath, Nshortcut

'Создаем объект WshShell:

Set WshShell = CreateObject(“WScript.Shell”)

'получаем путь к нашему рабочему столу:

DesktopPath = WshShell.SpecialFolders(“Desktop”)

'создаем ярлык:

Set NShortcut = WshShell.CreateShortcut(DesktopPath & “\Блокнот.lnk”)

На втором этапе мы должны установить следующие свойства ярлыка, такие как:

**TargetPath** – имя ресурса на который ссылается ярлык.

**WindowStyle** – стиль запускаемого окна.

Может принимать следующие значения:

1 – открыть как обычное окно;

3 – раскрыть на весь экран;

7 – запустить в свернутом виде.

**HotKey** – устанавливает клавиши быстрого вызова

**IconLocation** – расположение иконки ярлыка, файлы .exe и .dll могут содержать несколько значков, номера которых начинаются с 0. Можно использовать библиотеки значков, например Moricons.dll или Shell32.dll, находящиеся в папке Windows или во вложенной папке \System.Можно также присвоить путь для .bmp .ico-файла.

**Description** – всплывающая подсказка (комментарий)

**WorkingDirectory** – устанавливает рабочий каталог, который приложение использует для хранения временных и других файлов.

И, на самом последнем этапе создания ярлыка, мы сохраняем установленные свойства и сам ярлык методом **Save**.

Таким образом, наш сценарий полностью будет иметь следующий вид:

VBScript:

Dim WshShell, DesktopPath, Nshortcut

'Создаем объект WshShell:

Set WshShell = CreateObject(“WScript.Shell”)

'получаем путь к нашему рабочему столу:

DesktopPath = WshShell.SpecialFolders(“Desktop”)

'создаем ярлык:

Set NShortcut = WshShell.CreateShortcut(DesktopPath & “\Блокнот.lnk”)

'С помощью функции API "%windir%" получаем путь к папке

'WINDOWS и Блокноту и помещаем полученный путь в

'TargetPath:

NShortcut.TargetPath = "%windir%\notepad.exe"

'Устанавливаем обычный стиль окна:

NShortcut.WindowStyle = 1

'Клавиши быстрого вызова:

NShortcut.Hotkey = "CTRL+ALT+f"

'Путь к файлу иконки:

NShortcut.IconLocation = "notepad.exe, 0"

'Устанавливаем всплывающую подсказку:

NShortcut.Description = "Ярлык блокнота"

'Устанавливаем в качестве рабочей папки - Рабочий стол:

NShortcut.WorkingDirectory = DesktopPath

'Сохраняем настройки и сам ярлык:

NShortcut.Save

Кроме ярлыков к локальным ресурсам можно создавать ярлыки Интернета. Создадим ярлык адреса компании Microsoft.

Dim WshShell, UrlLink

Set WshShell = WScript.CreateObject("WScript.Shell")

Set UrlLink = WshShell.CreateShortcut("Microsoft.URL")

UrlLink.TargetPath = "http://www.microsoft.com"

UrlLink.Save

***2.5 Работа с ресурсами локальной сети***

**Работа с локальной сетью**

При работе с локальной сетью наиболее часто решаются следующие задачи:

* организация общих сетевых ресурсов;
* подключение пользователей к ресурсам;
* организация резервного копирования с одного компьютера в сети на другой.

Для доступа к сетевым ресурсам используются **сетевые пути** в следующем формате:

\\Имя\_компьютера\Имя\_ресурса[\Имя\_папки\...\Имя\_файла]

Например, сетевым именем Server1 имеется общедоступный принтер с сетевым именем Epson, то путь к нему указывается так: \\Server1\Epson.

Если на этом же компьютере имеется общедоступная папка Programs, то путь к ней выглядит как \\Server1\Programs, а путь к файлу в этой папке, например, так: \\Server1\Programs\Corel\Coreldrv.exe

Для работы с сетевой папкой, расположенной на сервере K2S, необходимо задать путь:

[\\K2S\Имя\_папки](file:///\\K2S\Имя_папки)

1. ***Объект WshNetwork***

Объект **WshNetwork** используется для работы с сетью, установки конфигурации сетевого окружения - а именно для управления сетевыми дисками и принтерами.

Через объект **WshNetwork** можно получить информацию о локальном компьютере, подключаться к дискам и принтерам в сети, устанавливать принтер по умолчанию и отключаться от сетевых дисков и принтеров.

Для его использования объект нужно создать.

Set WshNetwork = CreateObject("WScript.Network")

Для получения информации о компьютере, можно получить его сетевые атрибуты: имя пользователя, имя компьютера и его домен.

Свойства объекта **WshNetwork**:

|  |  |
| --- | --- |
| Свойство | Описание |
| ComputerName | Имя компьютера |
| UserDamain | Домен |
| UserName | Имя пользователя |

1. ***Сетевые диски***

Для работы с сетевыми дисками Windows Script Host предоставляет несколько методов, которые позволяют получать сведения о подключенных сетевых дисках, подключать сетевые диски и отключаться от них.

У объекта **WshNetwork** имеется методы **EnumNetworkDrives и EnumNetworkPrinterConnections,** с помощью которых можно создать коллекции, содержащие сведения обо всех подключенных к локальной сети сетевых дисках и сетевых принтерах. Эти коллекции устроены следующим образом: первым элементом коллекции является буква диска или название порта, вторым – сетевое имя ресурса, с которым связан этот диск или принтер. Та же последовательность сохраняется для всех элементов коллекции.

**EnumNetworkDrives** - возвращает список подключенных сетевых дисков.

Синтаксис:

*objDrives = object.EnumNetworkDrive*

где

*object* - объект WshNetwork

*objDrives* - переменная, которой присваивается ссылка на коллекцию сетевых дисков

Dim WshNetwork, Drives

Set WshNetwork = WScript.CreateObject("WScript.Network")

Set Drives = WshNetwork.EnumNetworkDrives

WScript.Echo "Подключены сетевые диски: "

For i = 0 to Drives.Count - 1 Step 2

WScript.Echo "Диски " & Drives.Item(i) & " = " & Drives.Item(i+1)

Next

**MapNetworkDrive** - назначает заданную букву сетевому диску.

Синтаксис:

*WshNetwork.MapNetworkDrive strLocalName, strRemoteName, [bUpdateProfile], [strUser], [strPassword]*

где

*strLocalName* - назначаемая буква сетевого диска

*strRemoteName* - удаленное имя

Необязательные параметры:

*bUpdateProfile* - логическая величина определяющая сохранять ли сделанную настройку в пользовательской конфигурации.

*strUser, strPassword* - вы можете указать имя и пароль для доступа к диску.

'создаем объект WshNetwork:

Set WshNetwork = CreateObject("WScript.Network")

'назначаем сетевой ресурс Server\PublicFiles как сетевой диск Z: WshNetwork.MapNetworkDrive "Z:", "\\Server\PublicFiles"

**RemoveNetworkDrive** - отключает сетевой диск

Синтаксис:

*WshNetwork.RemoveNetworkDrive strName, [bForce], [bUpdateProfile]*

где:

*strName* - имя диска

Необязательные параметры:

*bForce* - логический параметр. Если принимает значение true, то сетевой диск отключается, даже если он в настоящий момент используется данным компьютером.

*bUpdateProfile* - логический параметр, указывающий, сохранить ли сделанную настройку в пользовательском профиле.

VBScript:

'создаем объект WshNetwork:

Set WshNetwork = WScript.CreateObject("WScript.Network")

'подключаем сетевой диск Z:

WshNetwork.MapNetworkDrive "Z:", "\\Server\PublicFiles"

'отключаем сетевой диск Z:

WshNetwork.RemoveNetworkDrive "Z:"

При работе с сетевыми ресурсами могут возникать так называемые **ошибки выполнения** или **исключения.**  Но это не те ошибки, когда вы забыли поставить закрывающую скобку, запятую или ошиблись в имени переменной. Это ошибки, возникающие в некоторых ситуациях во время выполнения сценария и зависящие от некоторых не регулярно возникающих обстоятельств. Например, для того, чтобы подключить сетевой диск к устройству Z: , нужно быть уверенным, что такая буква не использована ранее (иначе произойдет ошибка).

Режим обработки исключительных ситуаций в VBScript включается с помощью оператора **On Error Resume Next**. Если после этого при выполнении какого-либо оператора в сценарии произойдет ошибка, то выполнение передастся следующему оператору в тексте.

Для анализа ошибок используется объект **Err**. Это стандартный объект VBScript, содержащий информацию о так называемых **run-time errors** (ошибках времени выполнения). Свойство **Source** указывает на источник ошибки. Метод **Clear** очищает свойства объекта, удаляя сведения об ошибке. Помимо Source и Clear у объекта есть и другие свойства и методы, но реально может понадобиться, пожалуй, свойство **Number** и **Description**. Number возвращает код произошедшей ошибки, а Description ее описание.

Для отмены режима обработки исключений служит оператор **On Error Goto 0**.

Пример сценария с обработкой исключений :

' Создаем объект WshNetwork

Set WshNetwork = WScript.CreateObject("WScript.Network")

' Создаем объект WshShell

Set WshShell = WScript.CreateObject("WScript.Shell")

' Включаем обработку ошибок времени выполнения

On Error Resume Next

' Отключаем сетевой диск

WshNetwork.RemoveNetworkDrive "Z:"

If Err.Number<>0 Then

Info="Ошибка при отключении диска " & Drive & vbCrLf & \_

"Код ошибки: " & err.number & vbCrLf &+ \_

"Описание: " & err.description

WshShell.Popup Info,0,"Отключение сетевого диска",vbCritical

Else

' Все в порядке

Info="Диск " & Drive & " отключен успешно"

WshShell.Popup Mess,0,"Отключение сетевого диска",vbInformation

End If

1. ***Работа с принтерами***

Подобно работе с сетевыми дисками, WSH предоставляет для работы с сетевыми принтерами по сути те же методы.

**EnumPrinterConnections** – возвращает список подключенных принтеров

Синтаксис:

*objPrinters = object.EnumPrinterConnections*

где

*object* - объект WshNetwork

*objPrinters* - ссылка на коллекцию принтеров

**AddPrinterConnection** – подключает сетевой принтер.

*WshNetworkt.AddPrinterConnection strLocalName, strRemoteName* *[,bUpdateProfile [,strUser][,strPassword]*

где

*strLocalName* – назначаемое имя подключенному принтеру

*strRemoteName* - удаленное имя

Необязательные параметры:

*bUpdateProfile* - логическая величина (true или false),определяющая сохранять ли сделанную настройку в пользовательской конфигурации.

*strUser*, *strPassword* - вы можете указать имя и пароль для доступа к диску.

Set WshNetwork = CreateObject("WScript.Network")

WshNetwork.AddPrinterConnection "LPT1", “Server\\LaserJet1”

**RemovePrinterConnection** – отключат сетевой принтер.

Синтаксис:

*WshNetwork.RemovePrinterConnection strName, [bForce], [bUpdateProfile]*

где:

*strName* - имя диска

Необязательные параметры:

*bForce* - логический параметр. Если принимает значение true, то сетевой принтер отключается, независимо от того, используется он в настоящий момент или нет.

*bUpdateProfile* - логический параметр, указывающий, сохранить ли сделанную настройку в пользовательском профиле.

Set WshNetwork = CreateObject("WScript.Network")

WshNetwork.RemovePrinterConnection "LPT1:"

Кроме методов, подобных методам для работы с сетевыми дисками Windows Scripting Host для работы с принтерами предоставляет еще несколько.

**AddWindowsPrinterConnections** – устанавливает новый принтер в Windows.

Синтаксис:

*WshNetwork.AddWindowsPrinterConnection(strPrinterPath)*

где

*strPrinterPath* - путь к принтеру

Set WshNetwork = CreateObject("WScript.Network")

PrinterPath= "\\printserv\DefaultPrinter"

WshNework.AddWindowsPrinterConnection(PrinterPath)

**SetDefaultPrinter** – устанавливает принтер по умолчанию

Синтаксис:

*object.SetDefaultPrinter strPrinterName*

где

*object* - объкт **WshNetwork**

*strPrinterName* - удаленное имя принтера

Set WshNetwork = WScript.CreateObject("WScript.Network")

‘создаем объект WshNetwork

WshNetwork.AddPrinterConnection "LPT1:", “\\Server\\Prnt1”

‘подключаем к системе сетевой принтер

WshNetwork.SetDefaultPrinter "LPT1:"

‘задаем его принтером по умолчанию

**3. Задания для выполнения**

* 1. Создайте с помощью Блокнота файл, выводящий сообщение. Например, введите текст: WScript.Echo "Это первый сценарий"
  2. Сохраните файл с любым из допустимых расширений (vbs).
  3. Запустите сценарий на выполнение в диалоговом режиме и режиме командной строки.
  4. Задайте настройки для сценария, для чего выполните следующие действия. Щелкните на нем правой кнопкой мыши и в окне свойств файла выберите вкладку «Сценарий». Измените настройки на этой вкладке, поставив или сняв любой флажок, чтобы кнопка «Вернуть установки по умолчанию» стала доступна. После этого щелкните OK. Появился файл с расширением wsh.
  5. В дальнейшем, если необходимо использовать измененные настройки, нужно вместо файла с расширением .vbs запускать файл с расширением .wsh.
  6. Откроем файл с расширением .wsh с помощью Блокнота. Там будет примерно следующий текст:

**[ScriptFile]**

**Path=C:\Мои документы\Пример1.VBS**

**[Options]**

**Timeout=10**

**DisplayLogo=1**

Параметр Path в разделе [ScriptFile] содержит путь к файлу, для которого используется wsh-файл. Настройки в разделе [Options] – это сами настройки, ради которых, собственно и создан файл настройки.

Timeout – определяет время отведенное для выполнения сценария.

DisplayLogo – отвечает за вывод эмблемы Windows Script Host, при запуске в командном режиме. Если изменить его на 0, то эмблема отображаться не будет.

BatchMode – включает/выключает пакетный режим. Если ему присвоить значение 1, то сценарий будет выполнятся в пакетном режиме – без вывода информации на экран и сообщений об ошибках.

* 1. Самостоятельно выполните приведенные в разделе 2 примеры.
  2. По индивидуальному варианту задания создайте файлы сценариев, которые выполняют необходимые операции.

**4. Варианты индивидуальных заданий.**

Необходимо выполнить следующие операции:

1. Выводит сведения о системе (см. таблицу).

|  |  |
| --- | --- |
| ***№ варианта*** | ***Сведения о системе*** |
| 1 | Операционная система, Директория Windows |
| 2 | Расширенные данные о процессоре, Поколение процессора |
| 3 | Заданные системные пути, директория Windows |
| 4 | Тип процессора, Операционная система |
| 5 | Количество процессоров на данном компьютере, Заданные системные пути |
| 6 | Диск на котором находится директория с операционной системой, Каталог по умолчанию для пользователей |
| 7 | Первый локальный диск, поколение процессора |
| 8 | Путь к файлу командной строки, папка для временных файлов |
| 9 | Тип процессора, заданные системные пути |
| 10 | Заданные системные пути, поколение процессора |
| 11 | Каталог по умолчанию для пользователей, Операционная система |
| 12 | Расширенные данные о процессоре, Операционная система |

1. Запускает приложение (на Ваш выбор) из пункта меню Стандартные с заданными параметрами (см. таблицу).

|  |  |  |
| --- | --- | --- |
| ***№ варианта*** | ***Стиль окна запускаемого приложения*** | ***Дожидаться окончания запущенного приложения*** |
| 1 | Обычный размер в неактивном состоянии, но в фокусе остается запустившее приложение | Да |
| 2 | Развернутое на весь экран | Да |
| 3 | Запуск в скрытом виде | Нет |
| 4 | Запуск в свернутом виде | Нет |
| 5 | Запуск в скрытом виде | Да |
| 6 | Развернутое на весь экран | Да |
| 7 | Запуск в обычном размере (без фокуса) | Нет |
| 8 | Развернутое на весь экран | Да |
| 9 | Запуск в скрытом виде | Нет |
| 10 | Развернутое на весь экран | Да |
| 11 | Запуск в обычном размере (без фокуса) | Нет |
| 12 | Развернутое на весь экран | Да |

1. Выводит сведения о путях ко всем специальным системным папкам, информацию о их количестве на данном компьютере. Использовать цикл для работы с семейством WshShell.SpecialFolders.
2. Создает ярлык для какого-либо приложения, разместив его в заданной папке, с заданным стилем окна (см. таблицу). Установить горячие клавиши, иконку, всплывающую подсказку. В качестве рабочего каталога задать каталог \Temp, для определения точного пути к нему использовать переменную окружения %Temp%.

|  |  |  |
| --- | --- | --- |
| ***№ варианта*** | ***Папка для размещения ярлыка*** | ***Стиль окна*** |
| 1 | Рабочий стол | На весь экран |
| 2 | Избранное | Свернутое |
| 3 | Мои документы | На весь экран |
| 4 | Избранное | Обычное |
| 5 | Рабочий стол | Свернутое |
| 6 | Мои документы | Обычное |
| 7 | Избранное | Свернутое |
| 8 | Рабочий стол | На весь экран |
| 9 | Мои документы | На весь экран |
| 10 | Избранное | Свернутое |
| 11 | Рабочий стол | На весь экран |
| 12 | Избранное | Обычное |

1. Создает ярлык для Web-узла, разместив его произвольной папке.
2. Выводит информацию о локальном компьютере, на котором работает студент (имя пользователя и имя компьютера.).
3. Выводит наименования подключенных сетевых дисков и их количество.
4. Выводит сведения о сетевых принтерах и их количество. Если сетевых принтеров несколько, запросить у пользователя, какой из них следует сделать используемым по умолчанию.
5. Выполняет попытку подключения сетевого диска с заданным пользователем именем и параметрами. Если подключение прошло успешно, перед завершением работы сценария предложить отключить сетевой диск. Организовать обработку исключительных ситуаций.

лАБОРАТОРНАЯ РАБОТА №6

Windows Scripting Host

**1. Цель работы**

Овладеть умением работать с командными . js файлами, изучить назначение команд.

**2. Общие сведения**

До Windows 98 практически не было средств автоматизации средств работы с файлами, то – есть выполнения рутинных операций. Конечно, ряд операций можно было выполнить с помощью старых пакетных (BAT) файлов MS DOS в окне командной строки. Но в командных файлах нельзя поддерживать диалоговые окна и сообщения. Также средства программирования весьма ограничены. Windows Script Host (WSH), позволяет писать на языках сторонних организаций - VBScript и JScript, на базе которого был создан автономный сервер сценариев на уровне ОС. Его можно запускать из командной строки. Языки VBScript и JScript имеют доступ к объектам WSH и любым другим объектам автоматизации (Automation).

Сценарий на Jscript имеет расширение .js. Для его запуска можно использовать командный интерпретатор script . exe , либо wscript . exe . В качестве параметра для них указывается файл со скриптом.

Создание диалогового окна. Для создания окна используется метод *Echo()* объекта Wscript

*WScript . Echo (" Hello World !");*

Пользователь может передать сценарию аргументы – строки, разделенные пробелами. Если один из аргументов содержит несколько строк, разделенных пробелами, его заключают в кавычки.

Прием параметров из командной строки.

Объект WScript.Arguments представляет массив аргументов. Вывод - c помощью метода Echo() объекта Wscript

*var objArgs=WScript.Arguments;*

*var text="Arguments \n";*

*var i;*

*for (i=0;i<objArgs.length;i++)*

*text=text+objArgs(i)+" ";*

*text=text+"\n";*

*WScript.Echo(text);*

***2.1 Синтаксис языка***

***Инструкции***

Код JScript **инструкции** состоит из одного или более символов в строке. Новая строка начинает новую инструкцию, но хорошим стилем является объявление конца инструкции явно. В JScript для этого используется точка с запятой (;).

*aBird = "Robin";*

*var today = new Date();*

Группа JScript-инструкций, заключенная в фигурные скобки ({}), называется блоком. Блоки инструкций используются как функции и условные выражения. В следующем примере первая инструкция определяет функцию, которая состоит из блока пяти инструкций. Последние три инструкции, которые не окружены фигурными скобками - блоком не являются.

*function convert(inches) {*

*feet = inches / 12; // Эти пять инструкций - блок.*

*miles = feet / 5280;*

*nauticalMiles = feet / 6080;*

*cm = inches \* 2.54; meters = inches / 39.37;*

*}*

*km = meters / 1000; // Эти инструкции блоком не являются.*

*kradius = km;*

*mradius = miles;*

***Комментарии***

**Комментарием** в JScript является текст, расположенный после двойного слэша (//) до конца строки. Многострочный комментарий начинается слэшем со знаком умножения (/\*), и кончается их обратной комбинацией (\*/).

*aGoodIdea = "Comment your code thoroughly."; // Однострочный комментарий.*

*/\* Это многострочный комментарий.*

*Инструкция объявляет значение переменной aGoodIdea.*

*Значение, которое содержится между кавычками, называется константой.*

*Константа явно и непосредственно содержит информацию.*

*Кавычки - не являются частью константы. \*/*

*// Это еще один вариант многострочного комментария*

*// Каждая строка начинается с двойного слэша*

*// В некоторых случаях такой комментарий очень удобен*

*var extendedIdea = aGoodIdea + " You never know when you'll have to figure out what it does.";*

***Специальные символы***

JScript позволяет использовать специальные символы, которые включаются в строки, но их невозможно напечатать непосредственно. Каждый из этих символов начинается с наклонной черты влево. Наклонная черта влево - ESC-символ, применяемый для того, чтобы сообщить интерпретатору, что следующий символ является специальным.

|  |  |
| --- | --- |
| Обозначение | Описание |
| \b | Удаление одного символа слева |
| \f | Конец страницы |
| \n | Перевод строки (новая строка) |
| \r | Возврат каретки |
| \t | Табуляция (Ctrl-I) |
| \’ | Одинарная кавычка - апостроф |
| \” | Двойная кавычка |
| \\ | Наклонная черта влево – обратный слэш |

Обратите внимание, что, потому что обратный слэш непосредственно означает ESC-символ, и его одиночный символ в сценариях не допустим. Если Вы хотите вывести обратный слэш, используйте два его символа вместе (\\).

*document.write('Путь к файлу C:\\webstuff\\mypage\\gifs\\garden.gif.');*

*document.write('The caption reads, "After the snow of \'97. Grandma\'s house is covered."');*

Вы можете использовать эти управляющие последовательности, чтобы управлять форматированием текста внутри тэгов <PRE> и <XMP>, а иногда в всплывающих окнах alert, prompt и confirm.

***Выражения***

JScript выражения можно разделить на Логические или Числовые . Выражения содержат некоторые особенности, к примеру, символ "+" означает " добавить к...". Любая допустимая комбинация значений, переменных, операторов, и других выражений является выражением.

*var anExpression = "3 \* (4 / 5)";*

*var aSecondExpression = "Math.PI \* radius \* 2";*

*var aThirdExpression = aSecondExpression + "%" + anExpression;*

*var aFourthExpression = "(" + aSecondExpression + ") % (" + anExpression + ")";*

***Объявление переменных***

Необязательно, но считается хорошим стилем программирования, объявление переменной перед использованием. Это делается с помощью иструкции **var** . Вы обязаны применять инструкцию **var** при объявлении локальной (local)переменной внутри функции. В остальных случаях объявление инструкции **var** перед применением в сценариях рекомендуется.

Примеры объявления переменных:

*var mim = "A man, a plan, a canal, Panama!"; // Строковый тип*

*var ror = 3; // Целый числовой тип.*

*var nen = true; // Boolean или логический тип.*

*var fif = 2.718281828 // Числовой тип.*

***Имена переменных***

JScript различает регистр в имени переменной: *myCounter* отличается от *MYCounter* . На практике присваивания имен требуется соблюдать следующие правила:

* Первым символом может быть буква любого регистра, или символ подчеркивания (\_), или знак доллара ($).
* Следующими символами могут быть буквы, символы подчеркивания, цифры и знаки доллара.
* Именами переменных не могут служить reserved word - зарезервируемые слова.

Несколько примеров правильных имен:

* \_pagecount
* Part9
* Number\_Items

Некоторые неправильные имена:

* 99Balloons // Первый символ - цифра.
* Smith&Wesson // Амперсанд(&) не разрешено применять в именах.

Если необходимо объявить и инициализировать переменную, но без присваивания определенного значения, можно применить значение **null** .

*var zaz = null;*

*var notalot = 3 \* zaz; // В данном случае notalot равен 0.*

Если переменная объявлена, а значение не присвоено, она существует, но значение не определено - undefined.

*var godot;*

*var waitingFor = 1 \* godot; // waitingFor имеет значение NaN, так как значение переменной godot не определено.*

Разрешается объявление переменной неявно - без инструкции **var** . Однако, в выражениях применять необъявленные переменные не допускается.

*lel = ""; // Переменная lel объявлена неявно.*

*var aMess = vyv + zez; // Ошибка, так как vyv and zez не существуют.*

***Типы данных***

***Что такое типы данных в JScript?***

В Microsoft JScript используется шесть типов данных. Основные - числа, строки, объекты, логический. Остальные два - **null** (NULL) **undefined** (Неопределенный).

***Строковый тип данных***

Строки объявляются при помощи двойных кавычек или апострофов. Строки могут быть объектами JScript, но это - частный случай с особыми свойствами. Несколько примеров строк:

*"The cow jumped over the moon."*

*'"Avast, ye lubbers!" roared the technician.'*

*"42"*

Строка может состоять из нуля или более символов unicode. Когда количество символов равно нулю, это называется пустой строкой ("").

***Числовой тип данных***

JScript поддерживает числа как целые, так и с плавающей точкой (запятой). Целые могут быть положительные, 0 (нуль) и отрицательными; a числа с плавающей точкой (запятой) представляются с десятичной точкой, либо в "e"-формате, означающим "10 в степени" в научном описании. Это соответствует IEEE 754 стандарту для представления чисел. Наконец, существуют специальные представления чисел:

* **NaN** , Not A Number или НеЧисло
* Плюс-Бесконечность
* Минус-Бесконечность
* Положительный 0
* Отрицательный 0

Целые числа могут быть представлены по основанию 10 (decimal-десятичные), 8 (octal-восьмеричные) или 16 (hexadecimal-шестнадцатиричные).

Восьмеричные числа отображаются с первым символом "0", состоят из чисел от 0 до 7. Если число начинается с "0", но содержат числа "8" и (или) "9", оно является десятичным. Указание в восьмеричном числе символа "e" (или "E") является ошибкой.

Шестнадцатиричные ("hex", 16-ричные) числа начинаются с "0x" (или "0X") и содержат цифры от 0 до 9, буквы от A до F (или от a до f). Символ "e" считается шестнадцатиричной цифрой, и форма числа не является экспоненциальной. Буквы A-F используются как шестнадцатиричные цифры, означаются цифры от 10 до 15 по основанию 10. Так, 0xF означает 15, а 0x10 эквивалентно 16.

8- и 16- ричные числа могут быть отрицательными, но не дробными. Число, начинающееся с "0" и содержащее десятичную точку, является десятичным с плавающей точкой; Если число начинается с "0x" или "00" содержит десятичную точку, все символы, расположенные справа от нее игнорируются.

Примеры чисел:

.0001, 0.0001, 1e-4, 1.0e-4 // Четыре приведенных числа с плавающей точкой равны друг другу.

3.45e2 // Число с плавающей точкой - равно 345.

42 // Целое число.

0377 // Восьмиричное число, эквивалентно 255.

00.0001 // 8-ричное число не может содержать плавающую точку, эквивалентно 0.

0378 // 8-ричное число, равное 378.

0Xff // 16-ричное, эквивалентное 255.

0x37CF // 16-ричное, эквивалентное 14287.

0x3e7 // 16-ричное, эквивалентное 999. 0x3.

45e2 // 16-ричное число не может содержать дробную часть, эквивалентно 3.

***Логический или булев тип***

Возможные значения типа - **true** и **false** . Это - специальные значения, не равные соответственно 1 и 0.

**Предупреждение.** При сравнении, любое выражение, равное 0, считается ложным - false, а любое выражение, равное числу, отличным от 0 будет правдой(верным) - true. Следующее выражение возвращает true: (false == 0)

***Undefined - тип не определен***

Значение undefined имеет переменная после ее объявления и до присвоения ей какого-либо определенного значения.

***Тип Null***

Переменная типа **null** - не имеет никакого определенного значения.

***Преобразование типов***

Так как JScript - язык с нестрогим контролем типов, переменные в JScript не имеют строго фиксированного типа. Переменные имеют тип, эквивалентный типу значения, которое они содержат. Однако, в некоторых случаях, необходимо принудительное преобразование переменной в определенный тип. Числа могут быть объявлены как строки, а строки необходимо преобразовать в числовой тип. Для этого применяют конверсионные функции **parseInt()** и **parseFloat()**.

*var theFrom = 1;*

*var theTo = 10;*

*var doWhat = "Count from ";*

*doWhat += theFrom + " to " + theTo + ".";*

После выполнения кода, переменная *doWhat* принимает значение "Count from 1 to 10." Числовой тип преобразовывается в строковый тип.

*var nowWhat = 0;*

*nowWhat += 1 + "10"; // В этом случае, "10" является строкой, // "+=" - оператор конкатенации.*

После исполнения кода, переменная *nowWhat* принимает значение "0110". Нижеследующее объясняет выполнение кода:

1. Посмотрите на типы 1 и "10". "10" - строковый, а 1 - числовой, поэтому число было преобразовано в строку.
2. Оператор **+** над строками, является оператором конкатенации. Результатом является "110".
3. Посмотрите на типы по обе стороны оператора +=. *nowWhat* включает число, и "110" - строку, и поэтому число преобразуется в строку.
4. В данный момент по обе стороны оператора += находятся строки, происходит конкатенация строк. Результатом является "0110".
5. Результат хранится в переменной *nowWhat* .

var nowThen = 0;

nowThen += 1 + parseInt("10");

В данном случае, "+=" является оператором сложения. После выполнения кода, переменная *nowThen* принимает значение 11.

***Операторы языка***

В JScript применяются множество операторов: арифметические, логические, разрядные, присваивания и прочие.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Вычислительные | | Логические | | Разрядные | | Присваивания | | Прочие | |
| Название | Символ | Название | Символ | Название | Символ | Название | Символ | Название | Символ |
| Унарный минус | - | Логическое НЕ | ! | Разрядное НЕ | ~ | Присваивание | = | Удаление | delete |
| Инкремент | ++ | Меньше | < | Поразрядный левый сдвиг | << | Составное присваивание | OP= | Тип | typeof |
| Декремент | -- | Больше | > | Поразрядный правый сдвиг | >> |  |  | Пусто | void |
| Умножение | \* | Меньше или равно | <= | Беззнаковый поразрядный правый сдвиг | >>> |  |  |  |  |
| Деление | / | Больше или равно | >= | Разрядное И | & |  |  |  |  |
| Деление по остатку | % | Равно | == | Разрядное ИСКЛЮЧАЮЩЕЕ ИЛИ | ^ |  |  |  |  |
| Сложение | + | Не равно | != | Разрядное ИЛИ | | |  |  |  |  |
| Вычитание | - | Логическое И | && |  |  |  |  |  |  |
|  |  | Логическое ИЛИ | || |  |  |  |  |  |  |
|  |  | Условное выражение (Тринар) | ?: |  |  |  |  |  |  |
|  |  | Запятая | , |  |  |  |  |  |  |
|  |  | Тождественно | === |  |  |  |  |  |  |
|  |  | Нетождественно | !== |  |  |  |  |  |  |

***Приоритет операторов***

В JScript операторы выполняются в определенном порядке, называемом приоритет операций. Следующий список отражает приоритет операторов от высшего к низшему. Операторы, указанные в одной строке, выполняются слева направо.

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| . [] () | Точка, индексы массивов, вызов функции |
| ++ -- - ~ ! typeof new void delete | Унарные операции, вывод типов данных, создание объектов, неопределенные значения |
| \* / % | Умножение, деление, деление по остатку |
| + - + | Сложение, вычитание, конкатенация строк |
| << >> >>> | Поразрядные сдвиги |
| < <= > >= | Меньше, меньше или равно, больше, больше или равно |
| == != === !== | Равно, неравно, тождественно, нетождественно |
| & | Разрядное И |
| ^ | Разрядное ИСКЛЮЧАЮЩЕЕ ИЛИ |
| | | Разрядное ИЛИ |
| && | Логическое И |
| || | Логическое ИЛИ |
| ?: | Условное выражение |
| = *OP* = | Операторы присваивания |
| , | Запятая |

Круглые скобки используются, чтобы изменить порядок выполнения операторов. Выражение в круглых скобках полностью вычисляется прежде, и его значение используется как остаточный член инструкции.

Оператор с более высоким приоритетом выполняется ранее оператора с низким приоритетом. Например:

z = 78 \* (96 + 3 + 45)

В данном выражении пять операторов: =, \*, (), + и +. Приоритет операторов - следующий: (), \*, +, +, =.

1. Первым вычисляется значение выражения в круглых скобках: К сумме операции 96 и 3 прибавляется 45, общая сумма равна 144.
2. Далее выполняется умножение: произведение 78 и 144 дает результат 11232.
3. Переменной z присваивается значение 11232 .

***Использование условных выражений***

В JScript поддерживаются условные выражения **if** и **if...else** . В выражении **if** проверяется условие, при соответствии этому условию, выполняется написанный разработчиками JScript-код. (В выражении **if...else** выполняется при несоответствии условию другой код.) Простейшая форма оператора **if** может быть написана одной строкой, Но обычно операторы **if** и **if...else** записываются в несколько строк.

Следующий пример демонстрирует синтаксис выражений **if** и **if...else** . В первом примере - простейшая логическая проверка. Если выражение в круглых скобках равно **true** , выражение или блок выражений после **if** выполняется.

*// Функция smash() определена в другом месте кода.*

*if (newShip)*

*smash(champagneBottle,bow); // Логическая проверка newShip на равенство true.*

*// В данном примере, условие выполняется, если оба подусловия равны true.*

*if (rind.color == "ярко-желтый " && rind.texture == "большие и малые пятна") {*

*theResponse = ("Это тыква? <br> ");*

*}*

*// В следующем примере, условие выполнится, если хотя бы одно из подусловий (или оба) равны true.*

*var theReaction = "";*

*if ((lbsWeight > 15) || (lbsWeight > 45)) {*

*theReaction = ("Неплохо! <br>");*

*} else {*

*theReaction = ("Не очень! <br>");*

*}*

***Цикл FOR***

***Описание***

Выполнение блока инструкций до тех пор, пока указанное условие верно.

***Синтаксис***

**for** ( *initialization* ; *test* ; *increment* )

*statement*

Синтаксис инструкции **for** состоит из следующих частей:

|  |  |
| --- | --- |
| **Часть** | **Описание** |
| *initialization* | Выражение ( expression ). Выполняется однажды - перед выполнением цикла. |
| *test* | Логическое выражение ( Boolean expression ). Если *test* равно **true** , выполняется *statement* . Если *test* равно **false** , цикл заканчивается. |
| *increment* | Выражение. Выполняется в конце каждой итерации цикла. |
| *statement* | Инструкция, выполняемая при *test* равном **true** . Может быть составной (compound statement ). |

***Примечание***

Обычно цикл **for** используется тогда, когда он должен быть выполнен определенное количество раз. Это продемонстрировано в следующем примере.

*/\* i установлено равным 0 в начале и увеличивается на единицу в конце каждой итерации.*

*Цикл завершается, когда i становится не менее 10 до итерации цикла. \*/*

*var myarray = new Array();*

*for (i = 0; i < 10; i++) {*

*myarray[i] = i;*

*}*

***Цикл FOR ... IN***

***Описание***

Выполнение инструкции для каждого элемента объекта или массива.

***Синтаксис***

**for** ( *variable* **in** [ *object* | *array* ])

*statement*

Синтаксис инструкции **for** состоит из частей:

|  |  |
| --- | --- |
| **Часть** | **Описание** |
| *variable* | Переменная (variable), содержащая любой элемент *object* . |
| *object,array* | Объект или массив для выполнения итераций. |
| *statement* | Инструкция, исполняемая для каждого элемента *object* . Может быть составной (compound statement). |

***Примечание***

До каждой итерации *variable* присваивается значение следующего элемента *object*. Можно использовать любою из инструкций внутри цикла точно также как элемент *object*.

При выполнении итераций в объекте, нельзя никаким способом определять или управлять порядком, в котором элементы объекта назначены для *variable* .

В следующем примере применяется инструкция **for ... in**:

*function ForStmDemo() { // Создание некоторых переменных.*

*var a, d, i, s = "";*

*d = new ActiveXObject("Scripting.Dictionary"); // Добавление ключей и значений*

*d.Add ("a", "Athens");*

*d.Add ("b", "Belgrade");*

*d.Add ("c", "Cairo"); // Размещение элементов в массиве.*

*a = (new VBArray(d.Items())).toArray(); //Выполнение итераций.*

*for ( i in a ) {*

*s += a[i] + "\n";*

*}*

*return(s);*

*}*

***Цикл WHILE***

***Описание***

Выполнение инструкции до тех пор, пока указанное условие не станет **false** .

***Синтаксис***

**while** ( *expression* )

*statement*

Синтаксис инструкции **while** состоит из частей:

|  |  |
| --- | --- |
| **Часть** | **Описание** |
| *expression* | Логическое выражение (Boolean expression), проверяемое до начала выполнения каждой итерации цикла. Если *expression* равно **true**, цикл выполняется. Если *expression* равен **false**, то происходит выход из цикла. |
| *statement* | Выполняемая инструкция при *expression* равном **true** . Может быть составной (compound statement). |

***Примечание***

Инструкция **while** проверяет *expression* до выполнения цикла. Если *expression* равно **false**, цикл никогда не будет выполнен.

В следующем примере применяется инструкция **while**:

*function BreakTest(breakpoint) {*

*var i = 0;*

*while ( i < 100 ) {*

*if (i == breakpoint)*

*break; i++;*

*}*

*return(i);*

*}*

***Цикл DO ... WHILE***

***Описание***

Выполнение блока инструкций один раз, затем цикл продолжает инсполняться, пока условие не станет **false** .

***Синтаксис***

**do**   
*statement*   
**while** ( *expression* ) ;

Синтаксис инструкции **do...while** состоит из частей:

|  |  |
| --- | --- |
| **Часть** | **Описание** |
| *statement* | Инструкция, исполняемая при *expression* равном **true** . Может быть составной (compound statement). |
| *expression* | Выражение (expression), преобразованное к булеву значению **true** или **false** . При *expression* равном **true**, цикл выполняется снова. Если *expression* равно **false**, происходит выход из цикла. |

***Примечание***

Значение *expression* не проверяется до выполнения первой итерации, это гарантирует, что цикл будет выполнен по крайней мере один раз. Условие проверяется после выполнения каждой итерации.

В следующем примере применена инструкция **do...while**:

*function GetDriveList() {*

*var fso, s, n, e, x;*

*fso = new ActiveXObject("Scripting.FileSystemObject");*

*e = new Enumerator(fso.Drives);*

*s = "";*

*do {*

*x = e.item();*

*s = s + x.DriveLetter;*

*s += " - ";*

*if (x.DriveType == 3)*

*n = x.ShareName;*

*else*

*if (x.IsReady)*

*n = x.VolumeName;*

*else*

*n = "[Drive not ready]";*

*s += n + "<br>";*

*e.moveNext(); }*

*while ( !e.atEnd() );*

*return(s);*

*}*

***Оператор SWITCH***

***Описание***

Выполнение одной или нескольких инструкций, если значение указанного выражения соответствует метке.

***Синтаксис***

**switch (** *expression* **) {**

**case** *label* **:**

*statementlist*

**case** *label* **:**

*statementlist*

...

**default :**

*statementlist*

**}**

Синтаксис инструкции **switch** состоит из частей:

|  |  |
| --- | --- |
| **Часть** | **Описание** |
| *expression* | Оцениваемое выражение. |
| *label* | Идентификатор соответствия для *expression*. Если *label* === *expression*, немедленно начинается выполнение *statementlist* после двоеточия и выполняется либо до инструкции **break**, которая необязательно, либо до конца инструкции **switch** . |
| *statementlist* | Одна и несколько инструкций к выполнению. |

***Примечание***

Используйте предложение **default** для инструкций, которые будут выполнены, если ни одна из меток не совпадет с *expression*. Оно должно размещаться в пределах инструкции **switch** .

Если никакой *label* не соответствует *expression*, a **default** не указан, то ни одна инструкция не будет выполнена.

Выполнение происходит следующим образом:

1. Оценивает *expression* и *label* по порядку до нахождения соответствия.
2. Если значение *label* равно *expression* , выполняется *statementlist* .   
   Выполнение продолжается до инструкции **break** или до окончания инструкции **switch** . Это делает возможным выполнение инструкций нескольких *label* при неиспользовании инструкций **break** .
3. Если нет ни одного *label* , равных *expression* , выполняется вариант **default** .
4. Выполняются инструкции, следующие за окончанием всех блоков в инструкции **switch** .

В следующем примере объект проверяется на тип:

*function MyObject() {*

*...*

*switch ( object.constructor ){*

*case Date : ...*

*case Number : ...*

*case String : ...*

*case MyObject : ...*

*default: ...*

*}*

*}*

***Функции***

***Что такое функция?***

Функции в Microsoft JScript выполняют определенные действия. Они могут возвращать некоторый результат, например результат вычислений или сравнения

Функции исполняют при вызове определенный блок инструкций. Это позволяет однажды определить функцию, а в дальнейшем вызывать ее когда потребуется.

Вы передаете данные функции, включая их в круглые скобки после имени функции. Данные в круглых скобках называются параметрами. В некоторых функциях нет параметров вообще; в некоторых - один параметр; иногда параметров несколько.

В JScript имеется два вида функций: встроенные и определяемые.

***Специальные встроенные функции***

Язык JScript включает несколько встроенных функций. Некоторые из них позволяют обрабатывать выражения и специальные символы, и преобразовывать строки в числовые значения.

Например, **escape()** и **unescape()** используются для перевода кода HTML и спецсимволов, то есть символов, которые нельзя непосредственно размещать в тексте. К примеру, символы "<" и ">", применяются для обозначения HTML-тэгов.

Функция **escape** принимает в параметре спецсимволы, а возвращает ESC-код символа. Каждый код ESC-код начинается со знака процента (%) и последующим двухзначным числом. Функция **unescape** является инверсионной. В качестве параметра вводится ESC-код, а возвращает символ.

Еще одна встроенная функция - **eval()** , которая выполняет любое правильное математическое выражение, представленное в виде строки. Функция **eval()** имеет один аргумент - выражение для выполнения.

*var anExpression = "6 \* 9 % 7";*

*var total = eval(anExpression); // Вычисляет выражение, равно 5.*

*var yetAnotherExpression = "6 \* (9 % 7)";*

*total = eval(yetAnotherExpression) // Вычисляет выражение, равно 12.*

*var totality = eval("Текст."); // Возвращает ошибку.*

***Создание собственных функций***

Вы можете создавать собственные функции и вызывать их, когда потребуется. Определение функции состоит из объявления параметров и блока инструкций JScript .

Функция checkTriplet в следующем примере принимает в качестве параметров длины трех сторон треугольника, и определяет, является ли треугольник прямоугольным, проверяя согласно Пифагорову правилу(триплету). (Квадрат длины гипотенузы прямоугольного треугольника равен сумме квадратов длин катетов.) Функция checkTriplet в ходе проверки вызывает одну из двух других функций.

При проверке значений при входных значениях чисел с плавающей точкой используется "машинное эпсилон" ("epsilon") - очень маленькое число. Из-за погрешности округления в вычислениях с плавающей точкой, без применения этого числа, функция может возвращать неверный результат. "Машинное эпсилон" является максимальной допустимой погрешностью.

*var epsilon = 0.0000000000001; // "Машинное эпсилон".*

*var triplet = false;*

*function integerCheck(a, b, c) { // Функция проверки при целых числах.*

*if ( (a\*a) == ((b\*b) + (c\*c)) ) { // Код проверки.*

*triplet = true;*

*}*

*} // Конец функции проверки при целых числах.*

*function floatCheck(a, b, c) { // Функция проверки при числах с плавающей точкой.*

*var theCheck = ((a\*a) - ((b\*b) + (c\*c))) // Контрольное число.*

*if (theCheck < 0) { // Вычисление модуля контрольного числа.*

*theCheck \*= -1;*

*}*

*if (epsilon > theCheck) { // Сравнение отклонения с допустимой погрешностью!*

*triplet = true;*

*}*

*} // Конец функции проверки при числах с плавающей точкой.*

*function checkTriplet(a, b, c) { // Проверка триплета.*

*//Первым делом, присвоим "a" наибольшее значение из трех входных чисел.*

*var d = 0; // Временная переменная.*

*if (c > b) { // При c > b, меняем местами.*

*d = c;*

*c = b;*

*b = d;*

*} // Иначе - не меняем.*

*if (b > a) { // При b > a, меняем местами.*

*d = b;*

*b = a;*

*a = d;*

*} // Иначе - не меняем.*

*// Сторона "a" является гипотенузой.*

*if (((a%1) == 0) && ((b%1) == 0) && ((c%1) == 0)) { // Проверка чисел - целые ли они?*

*integerCheck(a, b, c); // Функция для целых чисел.*

*} else*

*floatCheck(a, b, c); // Иначе вызываем функцию для чисел с плавающей точкой*

*} // Окончание проверки правила Пифагора (триплета).*

*// Практическая проверка - объявляем три переменные.*

*var sideA = 5;*

*var sideB = 5;*

*var sideC = Math.sqrt(50);*

*checkTriplet(sideA, sideB, sideC); // Функция проверки на триплет.*

***Массивы***

***Индексация массивов***

Массивы в JScript разряжены. Так, если в массиве три значения пронумерованы как 0, 1 и 2, вы можете создать элемент 50, не волнуясь об элементах от 3 до 49. Если массив имеет автоматическую переменную размера, переменная размера установлена как 51, несмотря на размер 4. Конечно, можно создавать массивы без разрывов в нумерации элементов, но это не обязательное условие. Фактически в JScript, массивы вообще могут не иметь нумерации.

В JScript объекты и массивы идентичны друг другу. Настоящая разница не в данных, а скорее в адресации элементов массива или свойств и методов объекта.

***Адресация массивов***

Существует два основных способа для адресации элементов массива. Обычно, для адресации используют индексы. Индексы содержат числовое значение или выражение, которое оценивается как неотрицательное целое. В следующем примере предполагается, что переменная *entryNum* определена, и ей присвоено значение в другом месте в сценарие. theListing = addressBook[entryNum]; theFirstLine = theListing[1]; Метод адресации эквивалентен методу адресации объектов, хотя при адресации объекта, индекс должен быть именем существующего свойства. Если такого свойства нет, возникает ошибка при исполнении кода.

Второй способ адресации массива состоит в том, чтобы создать объект (массив), который содержит свойства, которые пронумерованы числами в цикле. В следующем примере создается два массива, один для имени, второй для адреса, внесенных в список *addressBook* . Каждый из них содержит четыре свойства. Образец *theName* , например, формируется от [Name1] до [Name4] свойств *theListing* , может содержать "G." "Edward" "Heatherington" "IV" или "George" "" "Sand" "".

*theListing = addressBook[entryNum];*

*for (i = 1; i < 4; i++) {*

*theName[i] = theListing["Name" + i];*

*theAddress[i] = theListing["Address" + i];*

*}*

В то же время код мог бы легко быть написан в "dot"-стиле системы обозначений (то есть адресуя *theListing* , *theName* и *theAddress* скорее как объекты, чем матрицы, через точку), но подобное не всегда возможно. Иногда какое-либо свойство не может существовать до времени выполнения, или нельзя его узнать заранее. Например, если массив *addressBook* упорядочен по фамилии вместо нумерации, пользователь будет вероятно вводить названия " на лету", в то время как сценарий функционирует, просматривая людей. Следующий пример показывает применение определений функций в другом месте сценария.

*theListing = addressBook[getName()];*

*theIndivListing = theListing[getFirstName()];*

Это ассоциативная адресация массива, то есть адресация посредством полностью произвольных строк. Объекты в JScript являются ассоциативными массивами. Хотя чаще всего используется "dot"-стиль, это не всегда требуется.

***2.2 Объект***

***Что такое Объект?***

В Microsoft JScript объекты по сути являются совокупностями методов и свойств. Метод - функция, которая выполняется внутри объекта, а свойство - значение или набор значений (в виде матрицы или другого объекта), являющееся частью объекта. В JScript объекты можно разделить на три вида: встроенные, созданные и внешние.

***Объекты и Массивы***

В JScript, обработка объектов и массивов идентична. Вы можете обратиться к любой части объекта (его свойствам и методам) либо по имени, либо по индексу. Нумерация индексов в JScript начинается с нуля. Для удобства работы, частям можно присвоить имена.

Существует несколько вариантов обращений. Следующие выражения эквивалентны.

*theWidth = spaghetti.width;*

*theWidth = spaghetti[3]; // [3] является индексом "width".*

*theWidth = spaghetti["width"];*

При использовании числовых индексов, в обращении по имени нельзя использовать точку (.). Следующее выражение вызывает ошибку.

theWidth = spaghetti.3;

В случаях, когда объект является свойством другого объекта, обращение образуется прямым путем.

*var init4 = toDoToday.shoppingList[3].substring(0,1); // массив shoppingList - свойство toDoToday.*

Так как объекты могут быть свойствами других объектов, возможно создание массивов с более чем одним измерением, которые непосредственно не поддерживаются. Следующий код создает таблицу умножения для значений от 0 до 16.

*var multTable = new Array(17); // Создание оболочки для таблицы*

*for (var j = 0; j < multTable.length; j++) { // Подготовка к заполнению строками*

*var aRow = new Array(17); // Создание строки*

*for (var i = 0; i < aRow.length; i++) { // Подготовка к заполнению строки*

*aRow[i] = (i + " times " + j + " = " + i\*j); // Создание и размещение одного значения*

*}*

*multTable[j] = aRow; // Заполнение таблицы строкой*

*}*

Обращение к каждому элементу производится указанием нескольких индексов

*var multiply3x7 = multTable[3][7];*

Следующее выражение вызовет ошибку.

*var multiply3x7 = multTable[3, 7];*

***Встроенные объекты***

В Microsoft JScript девять встроенных объектов. Это **Array**, **Boolean**, **Date**, **Function**, **Global**, **Math**, **Number**, **Object** и **String**. Каждый из объектов имеет свои методы и свойства, подробно описанные в справочнике языка. Здесь приводится описание объектов.

***Объект Array - Массив***

В JScript обработка массивов происходит подобно обработки объектов. Индекс в массиве, эквивалентный свойству объекта, может быть вызван по порядковому номеру (или по имени, ассоциированному с ним). Создания нового массива происходит с помощью оператора **new** и конструктора **Array()** ( constructor ), подобно приведенному ниже примеру.

*var theMonths = new Array(12) { theMonths[0] = "Jan";*

*theMonths[1] = "Feb";*

*theMonths[2] = "Mar";*

*theMonths[3] = "Apr";*

*theMonths[4] = "May";*

*theMonths[5] = "Jun";*

*theMonths[6] = "Jul";*

*theMonths[7] = "Aug";*

*theMonths[8] = "Sep";*

*theMonths[9] = "Oct";*

*theMonths[10] = "Nov";*

*theMonths[11] = "Dec"; }*

При создании массива, JScript включает в объект **Array** свойство размера ( length ), которое характеризует количество элементов массива. Если массив не имеет элементов, то его размер равен нулю. Если задан номер индекса, размер массива равно ему. В следующем примере

*var theMonths = new Array("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec");*

*var l = theMonths.length;*

*document.write(l);*

размер массива равен 12.

***Объект String - Строка***

В JScript строки являются объектами. Это означает, что в любой можно объявить переменную строкового типа, либо применить строку-константу, что фактически и происходит при создании нового объекта String. Объект **String** имеет встроенные методы, которые можно применить при работе со строками. Одним из методов является **substring**, возвращающий часть строки. Он содержит два аргумента числового типа.

*aString = "0123456789";*

*var aChunk = aString.substring(4, 7); // aChunk равно "456".*

*var aNotherChunk = aString.substring(7, 4); // aNotherChunk равно "456".*

*// Применение метода в созданном ранее массиве:*

*firstLetter = theMonths [5].substring(0,1);*

*// переменной firstLetter присвоено значение "J".*

Еще одно свойство объекта **String** - свойство длины ( **length** ) строки, значение которого соответствует количеству символов в строке, значение 0 соответствует пустой строке. Это значение может быть использовано в вычислениях.

*var howLong = "Hello World".length // Возвращает значение 11, которое присваивается переменной howLong.*

***Объект Math - Математические вычисления***

Объект **Math** включает ряд предопределенных свойств и методов. Свойствами являются определенные числа. Например, число ПИ (приблизительно 3.14159...). Свойство **Math.PI** используется в приведенном ниже примере.

*// Переменная radius является числовым значением.*

*var circleArea = Math.PI \* radius \* radius; // Вычисление площади круга.*

Пример встроенного метода объекта **Math**: вычисление возведения в степень числа ( **pow** ). Приведенный пример использует число ПИ и метод возведения в степень.

*// Вычисление объема сферы в зависимости от радиуса.*

*volume = (4/3)\*(Math.PI\*Math.pow(radius,3));*

***Объект Date – Дата***

Объект **Date** применяется для вывода текущей даты, а также для определения разницы между датами. Все значения методов и свойств предопределены. Вообще объект **Date** позволяет определить день недели; день, месяц, год; а также время в часах, минутах и секундах. Основанием является количество миллисекунд, прошедших с 1-го января 1970 года, 00 ч,00 мин,00.000 сек по GMT. GMT означает "Среднее Время По Гринвичу"; условно UTC, или "Универсальная Координата Времени", являющееся Международным Эталоном Времени.

**Предупреждение.** В JScript для обработки временных величин до полуночи 1-го января 1970 года нельзя применять объект **Date** . Для работы с временными данными до 1970 года необходимо написать собственные функции.

Для создания нового объекта **Date** используется оператор **new** . В следующем примере для текущего года определяется количество прошедших и оставшихся дней.

*/\* В переменной thisIsToday содержится массив для текущего момента. \*/*

*var thisIsToday = new Date();*

*var toDay = new Date();*

*// Текущая дата.*

*// Определение года, месяца и дня.*

*var thisYear = toDay.getYear() + 1900;*

*var thisMonth = theMonths[toDay.getMonth()];*

*var thisDay = thisMonth + " " + toDay.getDate() + "," + (parseInt(toDay.getYear()) + 1900);*

*// Определение количества дней.*

*thisDay = Math.round(Date.parse(thisDay)/8.64e7);*

*// То же самое для дня начала текущего года.*

*var firstDay = "Jan 1, " + thisYear;*

*firstDay = Math.floor(Date.parse(firstDay)/8.64e7);*

*// То же самое для дня конца текущего года.*

*var lastDay = "Dec 31, " + thisYear;*

*lastDay = Math.floor(Date.parse(lastDay)/8.64e7);*

*// Количество дней в году.*

*var daysInYear = (lastDay - firstDay) + 1;*

*// Определение количество прошедших и оставшихся дней.*

*var daysElapsed = thisDay - firstDay;*

*var daysLeft = daysInYear - daysElapsed;*

*// Комментарии.*

*var comment1 = daysElapsed+ " days have elapsed in the year.";*

*var comment2 = "That means there are " + daysLeft + " days left in " + thisYear + ".";*

*// Особые случаи.*

*if (daysElapsed == 0) {*

*comment1 = "Первого Января " + thisYear + " года.";*

*}*

*if (daysElapsed == 1) {*

*comment1 = "Прошел один день.";*

*}*

*if(daysElapsed == daysInYear) {*

*comment1 = thisYear + " закончен.";*

*}*

*if (daysLeft == 0) {*

*comment2 = "Наилучшие пожелания в Новом Году!";*

*}*

*if (daysLeft == 1) {*

*comment2 = "Остался один день в " + thisYear + " году.";*

*}*

*if (daysLeft == daysInYear) {*

*comment2 = "С Новым Годом!";*

*}*

***Объект Number – Число***

В дополнение к особым числовым свойствам (**PI**, например), доступным в объекте **Math**, в Microsoft JScript доступ к некоторым свойствам происходит через объект **Number**.

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| MAX\_VALUE | Наибольшее доступное число - примерно 1.79E+308; положительное или отрицательное. (Значение может изменяться в зависимости от системы.) |
| MIN\_VALUE | Наименьшее доступное число - примерно 2.22E-308; положительное или отрицательное. (Значение может изменяться в зависимости от системы.) |
| NaN | Специальное нечисловое значение, "not a number" - НеЧисло. |
| POSITIVE\_INFINITY | Любое положительное число, большее Number.MAX\_VALUE конвертируется в это значение; представляется как "Inf". |
| NEGATIVE\_INFINITY | Любое отрицательное число, меньшее -Number.MAX\_VALUE конвертируется в это значение; представляется как "-Inf". |

**Предупреждение. Number.NaN** - особое свойство, означающее нечисловое значение - "not a number." Например, деление на нуль возвращает **NaN** . Попытка использовать строку как число также возвращает **Number.NaN** . При сравнении **NaN** не равно ни какому-либо числу, ни самому себе. Следовательно, для проверки на результат **NaN** , не следует использовать сравнение с **Number.NaN** ; для этого применяется функция **isNaN()** .

***Объекты ActiveX***

Основные дополнения к языку JavaScript, сделанные корпорацией Microsoft, связаны с использованием управляющих элементов ActiveX . Эти элементы представляют собой динамические библиотеки, созданные в соответствии со стандартом COM. COM (Component Object Model) — это стандарт создания программных компонентов, рассмотрение которого выходит за рамки нашего справочника. Для наших целей вполне достаточно воспринимать элементы ActiveX как черный ящик, выполняющий определенные действия по запросу вызвавшей его программы.

Появление элементов ActiveX было порождено стремлением Microsoft создать технологию, которая могла бы успешно конкурировать с подключаемыми модулями обозревателей Netscape. Следует признать, что задача была решена исключительно удачно, т. к. в результате появился стандарт разработки платформо-зависимых программных компонентов, реализующих платформо-независимые протоколы и архитектуры.

Все элементы ActiveX при установке в системе Windows регистрируются в ее регистре. При этом в качестве уникального ключа используется GUID данного элемента, т. е. его уникальный 64-разрядный шестнадцатеричный номер.

Объект ActiveXObject используется для запуска объектов ActiveX. Он создается конструктором

*new ActiveXObject(" сервер . класс " [," адрес "?])*

Здесь сервер — имя COM-сервера, класс — имя класса создаваемого объекта, адрес — необязательное имя сетевого сервера, на котором должен быть создан заданный объект (может быть доменным именем или IP-адресом). Создание объекта на удаленном сервере действующими версиями JScirpt пока не поддерживается, т. е. аргумент адрес при вызове данного конструктора игнорируется.

После создания объекта ActiveX (т. е. после запуска соответствующей программы) мы можем обращаться к его методам и свойствам, используя стандартный синтаксис языка JavaScript. Встроенных свойств и методов этот объект не имеет.

Пример: если на компьютере-клиенте установлен Microsoft Word 7.0, то следующий сценарий его запустит, откроет документ Mydoc.doc, сохранит этот документ в текстовом формате и завершит работу MS Word.

*var word = new ActiveXObject("Word.Application"); // запускает MS Word*

*word.Documents.Open("Mydoc.doc"); // открывает документ*

*word.ActiveDocument.SaveAs("Mydoc.txt", 4); // 4 = текстовый формат DOS*

*word.Quit(); // завершает работу MS Word*

Следует помнить, что после запуска объекта ActiveX он остается активным до закрытия окна обозревателя, из которого он был вызван. JScript не содержит стандартного средства для завершения работы этого объекта. Если мы хотим, чтобы объект прекратил работу ранее завершения работы нашего сценария, то должны вызвать соответствующий метод данного объекта (в приведенном примере это метод Quit).

Любопытно, что операция typeof возвращает для свойств и методов объектов ActiveX нестандартное значение "unknown" (видимо из-за того, что все объекты ActiveX являются наследниками COM-интерфейса IUnknown).

***Объект Wscript***

Объект Wscript позволяет скриптам "познать самих себя". Вдобавок этот объект позволяет скриптам запускать приложения и управлять ими. Основные свойства и методы этого объекта перечислены в таблице:

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Arguments | Возвращает указатель на список аргументов командной строки |
| FullName | Возвращает имя исполняемого файла хоста и полный путь к нему (например, C:\Windows\wscript.exe) |
| Name | Выводит замечательную надпись Windows Scripting Host |
| Path | Определяет каталог и путь, содержащие wscript.exe или cscript.exe |
| ScriptFullName | Возвращает полный путь и имя исполняемого в данный момент скрипта |
| ScriptName | То же, что и ScriptFullName, но без пути |
| Version | Возвращает версию установленного Windows Scripting Host |
| Метод Описание |  |
| **Метод** | **Описание** |
| CreateObject | Создает объект по его ProgID |
| ConnectObject | Позволяет подключиться к событиям объекта. В качестве параметра принимает объект, к которому надо подключиться и префикс соответствующих событиям процедур, реализованных в скрипте |
| DisconnectObject | Отключает от объекта, подключенного предыдущим методом |
| Echo | Выводит текстовую строку в cscript - в StdOut, в Wscript - в виде диалогового окна. |
| GetObject | Позволяет получить указатель на объект из файла или объекта, указанного в параметре strProgID. |
| Quit | Завершает скрипт |
| Sleep | Переводит скрипт в неактивное состояние на время, указанное в миллисекундах |

***Объект WsHShell***

Объект WshShell позволяет скриптам устанавливать и конфигурировать другие приложения. Он также дает возможность общаться с пользователем, изменять Реестр и находить папки. Некоторые его свойства и методы перечислены в таблице.

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Environment | Возвращает объект, который позволяет получить значения переменных среды, например  var WSHShell = CreateObject("WScript.Shell");  var WshSysEnv = WshShell.Environment("Process");  WScript.Echo(WshSysEnv("windir")); |
| SpecialFolders | Возвращает полный путь к специальным папкам типа меню Пуск. Вот список доступных папок: AllUsersDesktop, AllUsersStartMenu, AllUsersPrograms, AllUsersStartup, Desktop, Favorites, Fonts, MyDocuments, NetHood, PrintHood, Programs, Recent, SendTo, StartMenu, Startup, Templates. |
| Метод Описание |  |
| **Метод** | **Описание** |
| Run | Запускает приложение. strCommand - исполняемая команда, параметр intWindowStyle позволяет управлять положением и поведением окон, а bWaitOnReturn позволяет указать объекту WshShell, надо ли ожидать окончания выполнения запущенного приложения. |
| Popup | Ещё один способ вывести текст в окне |
| CreateShortcut | Создает ярлыки файлов или URL |
| RegRead | Возвращает значение ключа или записи реестра по его имени |
| RegWrite | Создает новые ключ или запись Реестра или записывает новое значение существующих. |
| RegDelete | Удаляет ключ или запись реестра |
| LogEvent | Позволяет записать сообщение в event log Windows NT или W2K или файл WSH.log в случае Windows 9х |
| ExpandEnvironmentStrings | Позволяет раскрыть строку, содержащую переменные среды |
| SendKeys | Позволяет эмулировать клавиатурный ввод |

Вызов WinShell=WScript.CreateObject(“WScript.Shell”) создает экземпляр объекта WshShell.

Метод Popup() этого объекта создает диалоговое окно с заголовком для вывода текста. Его параметры:

•  Текст – выводимый текст

•  Таймаут – время, пока окно не будет автоматически закрыто. Если параметр равен 0, то окно закрывается пользователем при нажатии кнопки

•  Заголовок – строка

•  Кнопки и пиктограмма – вид пиктограммы и набор кнопок

Пиктограммы и кнопки определяются числами, для наглядности эти значения присваиваются переменным с определенными мнемоническими именами.

Константы видов пиктограмм диалогового окна:

|  |  |
| --- | --- |
| Значение | Описание |
| 16 | Пиктограмма "Стоп" |
| 32 | Пиктограмма "Вопрос" |
| 48 | Пиктограмма "Восклицательный знак " |
| 64 | Пиктограмма "Информация" |

Константы видов кнопок:

|  |  |
| --- | --- |
| Значение | Описание |
| 0 | Кнопка Да |
| 1 | Кнопки Да и Прервать |
| 2 | Кнопки Прервать, Повторить и Игнорировать |
| 3 | Кнопки Да, Нет и Прервать |
| 4 | Кнопки Да и Нет |
| 5 | Кнопки Повторить и Прервать |

Кнопки можно комбинировать, складывая константы

Вывод диалогового окна

*var mbOKCancel=1, mbInfo=64, mbCancel=2,*

*Text="Test simple", Title="My Title",*

*WshShell=WScript.CreateObject("WScript.Shell"),*

*DoIt=WshShell.Popup(Text,0,Title,mbOKCancel+mbInfo);*

*if(DoIt==mbCancel) {*

*WScript.Quit();*

*}*

*WScript.Echo("Sample executed");*

Вывод диалогового окна с приветствием

*var OkOnly=0, Info=64;*

*var cmonth = new Array ("January","February","March","April", "May","June","July","August","September","October","November","December");*

*var cday=new Array("Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday");*

*var name="First ", firstname="S.", title="Welcome, "+firstname+" "+name;*

*var objAdr=WScript.CreateObject("WScript.Shell");*

*var mydate=new Date(), weekday=mydate.getDay(), day=mydate.getDate(),*

*month=mydate.getMonth(),year=mydate.getYear(),*

*text="Today is "+cday[weekday]+" "+cmonth[month]+" "+day+" "+year+"\n"+"Windows has been launched";*

*objAdr.Popup(text,10,title,OkOnly+Info);*

Метод Run() предназначен для вызова программ, в частности, команд MS DOS (являющихся внешними или внутренними утилитами).

Для внешних утилит (например, sort, move и т.д.) аргумент метода – строка с именем утилиты. Для выполнения внутренней утилиты (например, dir, copy и т.д.) необходимо запустить командный процессор и передать ему внутреннюю команду. Для Windows 95/98 эта командная строка имеет вид:

C:\Windows\command.com /k dir.

Параметр /k говорит о том, что окно MS DOS не закрывается.

Для Windows 2000/XP командный процессор cmd.exe.

Чтобы не зависеть от версии Windows, следует использовать переменную окружения COMSPEC, которая содержит полное имя командного процессора

Объект WshArguments служит для доступа к набору аргументов сценария (аргументов командной строки). Его свойство length определяет число аргументов командной строки.

***Объект FileSystemObject***

Объект FileSystemObject (FSO) содержит следующие объекты и коллекции:

|  |  |
| --- | --- |
| **Объект/Коллекция** | **Описание** |
| Drive | Объект. Содержит методы свойства, которые позволяют собирать информацию об устройстве. |
| Drives | Коллекция. Содержит список устройств, связанных с системой физически или логически. |
| File | Объект. Содержит методы и свойства для создания, удаления, копирования или перемещения файла, а также для получения информации о свойствах файла (имя, путь и т.д.). |
| Files | Коллекция. Содержит список всех файлов, содержащихся в папке. |
| Folder | Объект. Содержит методы и свойства для создания, копирования, удаления или перемещения папки, а также для получения информации о свойствах папки (имя, путь и т.д.). |
| Folders | Коллекция. Содержит список всех папок внутри папки. |

***Методы:***

**BuildPath(путь,имя)**

Присоединяет имя к пути файла

**CopyFile (имя копируемого файла или группы файлов, имя каталога или файла копии[,булевский при знак необходимости перекрытия существующего файла при копировании] )**

Копирует один или более файлов из одного положения в другое. По умолчанию последний аргумент равен true

Маска \* может использоваться только в имени исходного файла. Например, можно использовать:

fso = CreateObject("Scripting.FileSystemObject");

fso.CopyFile ("c:\\mydocuments\\letters\\ \* .doc", "c:\\tempfolder\\")

но нельзя использовать:

fso = CreateObject("Scripting.FileSystemObject");

fso.CopyFile ("c:\\mydocuments\\ \* \\R1???97.xls","c:\\tempfolder")

**CopyFolder ( имя копируемой папки, имя каталога копии [, булевский при знак необходимости перекрытия существующего каталога при копировании] )**

Рекурсивно копирует папку из одного положения в другое.

Маска \* может использоваться только в имени исходного файла. Например, можно использовать:

fso = CreateObject ("Scripting.FileSystemObject");

fso.CopyFolder ("c:\\mydocuments\\letters\\ \* ","c:\\tempfolder\\")

но нельзя использовать:

fso = CreateObject ("Scripting.FileSystemObject");

fso.CopyFolder ("c:\\mydocuments\\ \* \\ \* ", "c:\\tempfolder\\")

Если имя исходной папки содержит символы маски или имя каталога – копии заканчивается символом \, то предполагается, что каталог – копия существует, и в него копируются указанные каталоги и подкаталоги. В противном случае предполагается, что имя каталога – копии есть создаваемая папка.

**CreateFolder( имя каталога)**

Создает каталог. Ошибка возникает, если каталог существует

**DeleteFile (полное имя файла )**

Удаляет файл Полное имя файла может содержать символы маски в последней компоненте имени. Ошибка возникает, если файл не существует.

**DeleteFolder ( полное имя папки)**

Удаляет папку и ее содержимое. Полное имя папки может содержать символы маски в последней компоненте имени. Не делается различия между пустыми и не пустыми папками. Ошибка возникает, если папка не существует.

**FileExists( полное имя файла)**

Возвращает true, если файл существует и false в противном случае.

**FolderExists( полное имя папки)**

Возвращает true, если папка существует и false в противном случае.

**GetAbsolutePathName(путь)**

Возвращает полный путь. Предположим, что текущий каталог c:\mydocuments\reports, таблица иллюстрирует поведение метода GetAbsolutePathName

|  |  |
| --- | --- |
| Путь | Результат выполнения метода |
| "c:" | "c:\mydocuments\reports" |
| "c:.." | "c:\mydocuments" |
| "c:\\" | "c:\" |
| "c:\*.\*\\may97" | "c:\mydocuments\reports\\*.\*\may97" |
| "region1" | "c:\mydocuments\reports\region1" |
| "c:\\..\\..\\mydocuments" | "c:\mydocuments" |

**GetDrive(спецификация устройства)**

Возвращает объект Drive, соответствующий аргументу.

Спецификация устройства может быть буквой устройства (c), буквой устройства с двоеточием (c:), буквой устройства с двоеточием и разделителем \ (c:\). Для вызова метода GetDrive обычно используется такой контекст:

*DriveSpec = GetDriveName(GetAbsolutePathName(Path))*

Пример -вывод вложенных папок данной папки.

*var Text="Folders\n\n",*

*wsh=WScript.CreateObject("WScript.Shell");*

*var fso=WScript.CreateObject("Scripting.FileSystemObject");*

*var path=fso.GetSpecialFolder(0);//1-System Folder*

*Text=Text+path+"\n";*

*var oFolders=fso.GetFolder(path);*

*var oSubFolder=new Enumerator(oFolders.SubFolders);*

*for( ; !oSubFolder.atEnd(); oSubFolder.moveNext()) {*

*oFolder=oSubFolder.item();*

*Text=Text+oFolder.Name+"\n";*

*}*

*WScript.Echo(Text);*

Объекты File и Folder имеют свойство Attributes, которое устанавливает или возвращает атрибуты файла или папки object.Attributes [= новое значение атрибутов]

Новое значение атрибутов может быть только одним из следующих значений или их комбинацией:

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Значение** | **Описание** |
| Normal | 0 | Нет атрибутов |
| ReadOnly | 1 | Атрибут может изменяться |
| Hidden | 2 | Атрибут может изменяться |
| System | 4 | Атрибут может изменяться |
| Volume | 8 | Атрибут не может изменяться |
| Directory | 16 | Атрибут не может изменяться |
| Archive | 32 | Атрибут может изменяться |
| Alias | 64 | Ссылка. Атрибут не может изменяться |

Пример использования атрибута:

*function ToggleArchiveBit(filespec) {*

*var fso, f, r, s;*

*fso=new ActiveXObject("Scripting.FileSystemObject");*

*f = fso.GetFile(filespec);*

*if ( f.attributes&&32 ) {*

*f.attributes = f.attributes-32;*

*s = "Archive bit is cleared.";*

*} else {*

*f.attributes = f.attributes+32;*

*s ="Archive bit is set";*

*}*

*return(s);*

*}*

**Пример файла скрипта**

*//input data:*

*// first argument - operation: delete - delete all files with extension followed after delete*

*// sort - sort file followed after sort*

*var mbOk = 0;*

*var mbOkCancel = 1;*

*var mbQuestion = 32;*

*var mbExclamation = 48;*

*var mbInfo = 64;*

*var mbOkOnly = 1;*

*var fso = WScript.CreateObject("Scripting.FileSystemObject");*

*function mySort(l1, l2) {*

*var temp1 = l1.substring(l1.lastIndexOf(" "), l1.length);*

*var temp2 = l2.substring(l2.lastIndexOf(" "), l2.length);*

*if (temp1 == temp2)*

*return 0;*

*else if (temp1 > temp2)*

*return 1;*

*else*

*return -1;*

*}*

*function WriteToFile(filename, text, bool) {*

*var fso1 = WScript.CreateObject("Scripting.FileSystemObject");*

*var f1 = fso1.CreateTextFile(filename, bool);*

*f1.WriteLine(text);*

*f1.Close();*

*}*

*function delFile(fname) {*

*try {*

*fso.DeleteFile(fname);*

*} catch (e) {*

*var strErr = "No Info";*

*if (e.description.length != 0)*

*strErr = e.description;*

*WScript.Echo(strErr);*

*}*

*}*

*var objArgs = WScript.Arguments;*

*if (objArgs.length == 0) {*

*WScript.Echo("Missing arguments. You can use following parametr:\ndelete extensions\nsort name\_of\_file");*

*WScript.Quit();*

*}*

*if (objArgs(i) == "delete") {*

*var i = 1;*

*if (objArgs.length == 1) {*

*delFile("\*.");*

*}*

*for (; i < objArgs.length;i++)*

*delFile("\*." + objArgs(i));*

*} else if (objArgs(i) == "sort") {*

*var inp;*

*if (objArgs.length > 1 )*

*inp = objArgs(1)*

*else*

*inp = "";*

*var bool = true;*

*do {*

*if ( !fso.FileExists(inp)) {*

*WScript.Echo("Invalid filename " + inp);*

*var ie = WScript.CreateObject("InternetExplorer.Application");*

*ie.Visible = 0;*

*ie.navigate("about:blank");*

*inp = ie.Application.Document.Script.prompt("Enter the file name", "");*

*if (inp == null)*

*WScript.Quit();*

*} else*

*bool = false;*

*} while (bool);*

*var textStream = fso.OpenTextFile(inp, 1); //Open file for reading*

*var strings = new Array(textStream.Line); //Create array of file's lines*

*var i = 0;*

*for (; i < textStream.Line; i++) {*

*try {*

*strings[i] = textStream.ReadLine();*

*} catch (e) {*

*var strErr = "no info";*

*if (e.description.length != 0)*

*strErr = e.description;*

*WScript.Echo(strErr);*

*}*

*}*

*textStream.Close();*

*strings.sort(mySort); //Sorting*

*var text = "";*

*for (i = 0; i < strings.length; i++)*

*text += strings[i] + "\n";*

*var wshshell = WScript.CreateObject("WScript.Shell");*

*wshshell.Popup(text, 0, "The result of sorting", mbOk+mbInfo);*

*if (objArgs.length > 2) {*

*if (fso.FileExists(objArgs(2))) {*

*var done = wshshell.Popup("File already exist. Rewrite it?", 0, "File already exist", mbOkCancel + mbQuestion);*

*if (done == mbOkOnly)*

*WriteToFile(objArgs(2), text, true);*

*else*

*wshshell.Popup("File hasn't been created", 0, "File already exist", mbOk + mbExclamation);*

*} else*

*WriteToFile(objArgs(2), text, false);*

*}*

*} else {*

*WScript.Echo("Invalid arguments");*

*}*

**3. Задания для выполнения**

В зависимости от запроса в интерактивном режиме выбирается один из нескольких вариантов действий. Всю информацию, не заданную конкретными значениями, передавать через параметры при вызове командного файла. Предусмотреть реакцию на ошибочные ситуации (отсутствие нужных файлов или присутствие вновь создаваемых).

**4. Варианты индивидуальных заданий.**

1. В зависимости от запроса
   1. отсортировать файлы заданного каталога, созданные до указанной даты, по расширению;
   2. удалить файлы с указанным расширением в указанном каталоге.
2. В зависимости от запроса
   1. удалить подкаталог текущего каталога с указанным именем (если он пуст);
   2. удалить из непустого подкаталога текущего каталога файлы с расширениями PAS,BAS и TXT.
3. В зависимости от запроса
   1. для каждого типа файлов -TXT,INF,ASM - выдать число строк с заданным текстом;
   2. отсортировать файлы заданными именами по убыванию 3-го символа.
4. В зависимости от запроса
   1. создать подкаталог текущего каталога с заданным именем и переместить в него из текущего каталога все файлы с расширением TXT, сделав их скрытыми;
   2. в каждом файле с расширением PAS или C заданного каталога найти и выдать на экран строки с определенным текстом, подсчитать количество таких строк.
5. В зависимости от запроса
   1. подсчитать число файлов заданного каталога, созданных или модифицированных в заданную дату;
   2. выдать на экран список имен файлов текущего каталога, содержащих заданную строку.
6. В зависимости от запроса
   1. объединить три файла с заданными именами в один и скопировать этот файл в заданный каталог;
   2. выдать на экран перечень всех подкаталогов заданного каталога.
7. В зависимости от запроса
   1. заданный файл в заданном каталоге, имеющий содержание: ФИО- 20 символов, адрес- 30 символов, N телефона- 6 символов отсортировать по возрастанию N телефона.
   2. в зависимости от значения параметра отсортировать заданный каталог либо по возрастанию имени, либо по убыванию расширения, либо по возрастанию размера.
8. В зависимости от запроса
   1. удалить из заданного каталога все файлы, имеющие расширение на заданную букву;
   2. выдать на экран содержимое файла с заданным именем заданного каталога, если он не пуст;
   3. переименовать файлы заданного каталога с именами, начинающимися на заданную букву.
9. В зависимости от запроса
   1. скопировать в файл с указанным именем указанного каталога те строки заданного файла текущего каталога, которые содержат заданный текст;
   2. переместить файл с заданным именем из одного каталога в другой.
10. В зависимости от запроса
    1. скопировать заданный файл текущего каталога в указанный каталог;
    2. отсортировать указанный файл указанного каталога по 10 символу, записать в файл с указанным именем;
    3. вывести на экран содержимое файлов указанного каталога с именем на заданную букву, затем эти файлы удалить.
11. В зависимости от запроса
    1. найти в заданном каталоге все файлы в названии которых содержится заданная строка и выдать их содержимое на экран;
    2. создать в заданном каталоге файл с заданным именем, содержащий текущую дату.
12. В зависимости от запроса
    1. заданный файл в текущем каталоге, содержит разделённые пробелами записи вида: дата звонка, телефонный номер вызывающего абонента, телефонный номер вызываемого абонента. Отсортировать данный файл по номеру вызывающего абонента;
    2. скопировать заданный файл из текущего каталога во все его подкаталоги, имена которых начинаются на заданную букву.

лАБОРАТОРНАЯ РАБОТА №7-8

Планирование процессов

**1. Цель работы**

Изучить как происходит планирование исполнения процессов в мультипрограммных вычислительных системах. Рассмотреть преимущества и недостатки различных алгоритмов планирования.

**2. Общие сведения**

Всякий раз, когда приходится иметь дело с ограниченным количеством ресурсов и несколькими их потребителями, занимаются распределением наличных ресурсов между потребителями или, другими словами, планированием использования ресурсов. Такое планирование должно иметь четко поставленные критерии (чего мы хотим добиться распределением ресурсов) и алгоритмы, соответствующие критериям и опирающиеся на параметры потребителей.

***2.1. Уровни планирования***

Существуют два вида планирования в вычислительных системах: планирование заданий и планирование использования процессора. Планирование заданий появилось в пакетных системах после того, как для хранения сформированных пакетов заданий начали использоваться магнитные диски. Магнитные диски, будучи устройствами прямого доступа, позволяют загружать задания в компьютер в произвольном порядке, а не только в том, в котором они были записаны на диск. Изменяя порядок загрузки заданий в вычислительную систему, можно повысить эффективность ее использования. Процедуру выбора очередного задания для загрузки в машину, т. е. для порождения соответствующего процесса, и называется планированием заданий. Планирование использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии ***готовность*** могут одновременно находиться несколько процессов. Именно процедура выбора из них одного процесса, который получит процессор в свое распоряжение, т.е. будет переведен в состояние ***исполнение***, называется планированием процессов. Оба эти вида планирования и есть различные уровни планирования процессов.

Планирование заданий выступает в качестве долгосрочного планирования процессов. Оно отвечает за порождение новых процессов в системе, определяя ее *степень мультипрограммирования*, т. е. количество процессов, одновременно находящихся в ней. Если степень мультипрограммирования системы поддерживается постоянной, т. е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения ранее загруженных. Поэтому долгосрочное планирование осуществляется достаточно редко, между появлением новых процессов могут проходить минуты и даже десятки минут. Решение о выборе для запуска того или иного процесса оказывает влияние на функционирование вычислительной системы на протяжении достаточно длительного интервала времени. Отсюда и проистекает название этого уровня планирования — долгосрочное. В некоторых операционных системах долгосрочное планирование сведено к минимуму или совсем отсутствует. Так, например, во многих интерактивных системах разделения времени порождение процесса происходит сразу после появления соответствующего запроса. Поддержание разумной степени мультипрограммирования осуществляется за счет ограничения количества пользователей, которые могут работать в системе, и человеческой психологии. Если между нажатием на клавишу и появлением символа на экране проходит 20-30 секунд, то многие пользователи предпочтут прекратить работу и продолжить ее, когда система будет менее загружена.

Планирование использования процессора выступает в качестве краткосрочного планирования процессов. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени. Поэтому краткосрочное планирование осуществляется весьма часто, как правило, не реже одного раза в 100 миллисекунд. Выбор нового процесса для исполнения оказывает влияние на функционирование системы до наступления очередного аналогичного события, т. е. в течение короткого промежутка времени, что и обусловило название этого уровня планирования — краткосрочное.

В некоторых вычислительных системах бывает выгодно для повышения их производительности временно удалить какой-либо частично выполнившийся процесс из оперативной памяти на диск, а позже вернуть его обратно для дальнейшего выполнения. Такая процедура в англоязычной литературе получила название *swapping*, что можно перевести на русский язык как перекачка, хотя в профессиональной литературе оно употребляется без перевода — свопинг. Когда и какой из процессов нужно перекачать на диск и вернуть обратно, решается дополнительным промежуточным уровнем планирования процессов — среднесрочным.

***2.2. Критерии планирования и требования к алгоритмам***

Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и ***целями***, которых мы хотим достичь, используя планирование. К числу таких целей можно отнести:

* **Справедливость:** гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не приступал к выполнению.
* **Эффективность:** постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90 процентов.
* **Сокращение полного времени выполнения (turnaround time):** обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.
* **Сокращение времени ожидания (waiting time):** минимизировать время, которое проводят процессы в состоянии ***готовность*** и задания в очереди для загрузки.
* **Сокращение времени отклика (response time):** минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Независимо от поставленных целей планирования желательно также, чтобы алгоритмы обладали следующими ***свойствами***:

* **Были предсказуемыми**. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма планирования не должно приводить, к примеру, к извлечению корня квадратного из 4 за сотые доли секунды при одном запуске и за несколько суток при втором запуске.
* **Имели минимальные накладные расходы, связанные с их работой.** Если на каждые 100 миллисекунд, выделенных процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, использовать не стоит.
* **Равномерно загружали ресурсы вычислительной системы**, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.
* **Обладали масштабируемостью**, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Многие из приведенных выше целей и свойств являются противоречивыми. Улучшая работу алгоритма с точки зрения одного критерия, мы ухудшаем ее с точки зрения другого. Приспосабливая алгоритм под один класс задач, мы тем самым дискриминируем задачи другого класса.

***2.3. Параметры планирования***

Для осуществления поставленных целей разумные алгоритмы планирования должны опираться на какие-либо характеристики процессов в системе, заданий в очереди на загрузку, состояния самой вычислительной системы, иными словами, на параметры планирования. В этом разделе мы опишем ряд таких параметров, не претендуя на полноту изложения.

Все параметры планирования можно разбить на две большие группы: статические параметры и динамические параметры. ***Статические параметры*** не изменяются в ходе функционирования вычислительной системы, ***динамические*** же, напротив, подвержены постоянным изменениям.

К статическим параметрам вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т. п.). Динамические параметры системы описывают количество свободных ресурсов в текущий момент времени.

К статическим параметрам процессов относятся характеристики, как правило, присущие заданиям уже на этапе загрузки:

* Каким пользователем запущен процесс или сформировано задание.
* Насколько важной является поставленная задача, т. е. каков приоритет ее выполнения.
* Сколько процессорного времени запрошено пользователем для решения задачи.
* Каково соотношение процессорного времени и времени, необходимого для осуществления операций ввода-вывода.
* Какие ресурсы вычислительной системы (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т. д.) и в каком количестве необходимы заданию.

Алгоритмы долгосрочного планирования используют в своей работе статические и динамические параметры вычислительной системы и статические параметры процессов (динамические параметры процессов на этапе загрузки заданий еще не известны). Алгоритмы краткосрочного и среднесрочного планирования дополнительно учитывают и динамические характеристики процессов. Для среднесрочного планирования в качестве таких характеристик может выступать следующая информация:

* Сколько времени прошло со времени выгрузки процесса на диск или его загрузки в оперативную память.
* Сколько оперативной памяти занимает процесс.
* Сколько процессорного времени было уже предоставлено процессу.

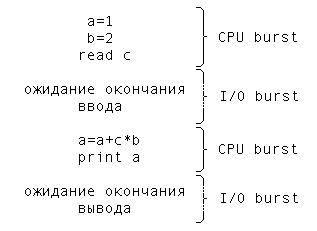


Рис 1. Фрагмент деятельности процесса с выделением промежутков непрерывного использования процессора и ожидания ввода-вывода.

Для краткосрочного планирования нам понадобится ввести еще два динамических параметра. Деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Промежуток времени непрерывного использования процессора носит на английском языке название *CPU burst*, а промежуток времени непрерывного ожидания ввода-вывода – *I/O burst*. На рисунке 1. показан фрагмент деятельности некоторого процесса на псевдоязыке программирования с выделением указанных промежутков. Значения продолжительности последних и очередных CPU burst и I/O burst являются важными динамическими параметрами процесса.

***2.4. Вытесняющее и невытесняющее планирование***

Процесс планирования осуществляется частью операционной системы, называемой планировщиком. Планировщик может принимать решения о выборе для исполнения нового процесса, из числа находящихся в состоянии ***готовность***, в следующих четырех случаях:

1. Когда процесс переводится из состояния ***исполнение*** в состояние ***завершение***.
2. Когда процесс переводится из состояния ***исполнение*** в состояние ***ожидание***.
3. Когда процесс переводится из состояния ***исполнение*** в состояние ***готовность*** (например, после прерывания от таймера).
4. Когда процесс переводится из состояния ***ожидание*** в состояние ***готовность*** (завершилась операция ввода-вывода или произошло другое событие).

В случаях 1 и 2 процесс, находившийся в состоянии ***исполнение***, не может дальше исполняться, и для выполнения всегда необходимо выбрать новый процесс. В случаях 3 и 4 планирование может не проводиться, процесс, который исполнялся до прерывания, может продолжать свое выполнение после обработки прерывания. Если планирование осуществляется только в случаях 1 и 2, говорят, что имеет место *невытесняющее (nonpreemptive) планирование*. В противном случае говорят о *вытесняющем (preemptive) планировании*. Термин “вытесняющее планирование” возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния ***исполнение*** другим процессом.

Невытесняющее планирование используется, например, в MS Windows 3.1 и ОС Apple Macintosh. При таком режиме планирования процесс занимает столько процессорного времени, сколько ему необходимо. При этом переключение процессов возникает только при желании самого исполняющегося процесса передать управление (для ожидания завершения операции ввода-вывода или по окончании работы). Этот метод планирования относительно просто реализуем и достаточно эффективен, так как позволяет использовать большую часть процессорного времени на работу самих процессов и до минимума сократить затраты на переключение контекста. Однако при невытесняющем планировании возникает проблема возможности полного захвата процессора одним процессом, который вследствие каких-либо причин (например, из-за ошибки в программе) зацикливается и не может передать управление другому процессу. В такой ситуации спасает только перезагрузка всей вычислительной системы.

Вытесняющее планирование обычно используется в системах разделения времени. В этом режиме планирования процесс может быть приостановлен в любой момент своего исполнения. Операционная система устанавливает специальный таймер для генерации сигнала прерывания по истечении некоторого интервала времени — *кванта*. После прерывания процессор передается в распоряжение следующего процесса. Временные прерывания помогают гарантировать приемлемые времена отклика процессов для пользователей, работающих в диалоговом режиме, и предотвращают “зависание” компьютерной системы из-за зацикливания какой-либо программы.

***2.5. Алгоритмы планирования***

Существует достаточно большой набор разнообразных алгоритмов планирования, которые предназначены для достижения различных целей и эффективны для разных классов задач. Многие из них могут быть использованы на нескольких уровнях планирования. В этом разделе рассмотрены некоторые наиболее употребительные алгоритмы применительно к процессу кратковременного планирования.

**2.5.1. First-Come, First-Served (FCFS)**

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой *FCFS* по первым буквам его английского названия — First Come, First Served (первым пришел, первым обслужен). Представим, что процессы, находящиеся в состоянии ***готовность***, организованы в очередь. Когда процесс переходит в состояние ***готовность***, он, а точнее ссылка на его PCB, помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование *FIFO* — сокращение от First In, First Out (первым вошел, первым вышел).

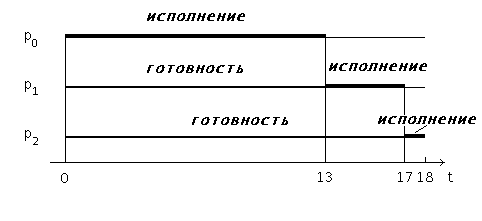
Надо отметить, что аббревиатура FCFS используется для этого алгоритма планирования вместо стандартной аббревиатуры FIFO для механизмов подобного типа для того, чтобы подчеркнуть, что организация готовых процессов в очередь FIFO возможна и при других алгоритмах планирования (например, для Round Robin).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения своего текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди.

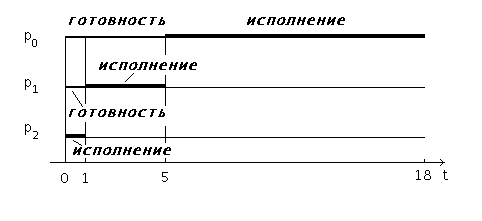
|  |  |  |  |
| --- | --- | --- | --- |
| Процесс | p0 | p1 | p2 |
| Продолжительность очередного CPU burst | 13 | 4 | 1 |

Таблица 1.

Преимуществом алгоритма FCFS является легкость его реализации, в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии ***готовность*** находятся три процесса p0, p1 и p2, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 1. в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода, и что время переключения контекста пренебрежимо мало.

  
Рис 2. Выполнение процессов при порядке p0,p1,p2

Если процессы расположены в очереди процессов готовых к исполнению в порядке p0, p1, p2, то картина их выполнения выглядит так, как показано на рисунке 2. Первым для выполнения выбирается процесс p0, который получает процессор на все время своего CPU burst, т. е. на 13 единиц времени. После его окончания в состояние ***исполнение*** переводится процесс p1, занимая процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p2. Время ожидания для процесса p0 составляет 0 единиц времени, для процесса p1 — 13 единиц, для процесса p2 — 13 + 4 = 17 единиц. Таким образом, среднее время ожидания в этом случае — (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса p0 составляет 13 единиц времени, для процесса p1 — 13 + 4 = 17 единиц, для процесса p2 — 13 + 4 + 1 = 18 единиц. Среднее полное время выполнения оказывается равным (13 + 17 + 18)/3 = 16 единицам времени.

  
Рис 3. Выполнение процессов при порядке p2,p1,p0

Если те же самые процессы расположены в порядке p2, p1, p0, то картина их выполнения будет соответствовать рисунку 3. Время ожидания для процесса p0 равняется 5 единицам времени, для процесса p1 — 1 единице, для процесса p2 — 0 единиц. Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае. Полное время выполнения для процесса p0 получается равным 18 единицам времени, для процесса p1 — 5 единицам, для процесса p2 — 1 единице. Среднее полное время выполнения составляет (18 + 5 + 1)/3 = 6 единиц времени, что почти в 2,7 раза меньше чем при первой расстановке процессов.

Как можно видеть, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Если есть процесс с длительным CPU burst, то короткие процессы, перешедшие в состояние ***готовность*** после длительного процесса, будут очень долго ждать начала своего выполнения. Поэтому алгоритм FCFS практически неприменим для систем разделения времени. Слишком большим получается среднее время отклика в интерактивных процессах.

**2.5.2. Round Robin (RR)**

Модификацией алгоритма FCFS является алгоритм, получивший название Round Robin (Round Robin – это вид детской карусели в США) или сокращенно *RR*. По сути это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить все множество готовых процессов организованным циклически — процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 - 100 миллисекунд (см. рисунок 3.4.). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

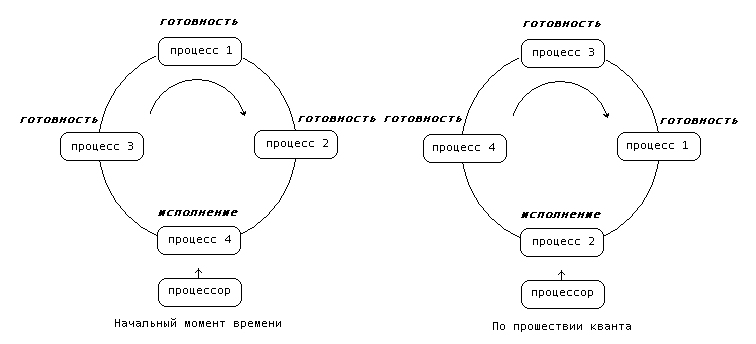


Рис 4. Процессы на карусели.

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии ***готовность***, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта:

* Время непрерывного использования процессора, требующееся процессу, (остаток текущего CPU burst) меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение выбирается новый процесс из начала очереди и таймер начинает отсчет кванта заново.
* Продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Рассмотрим предыдущий пример с порядком процессов p0, p1, p2 и величиной кванта времени равной 4. Выполнение этих процессов иллюстрируется таблицей 2. Обозначение “И” используется в ней для процесса, находящегося в состоянии ***исполнение***, обозначение “Г” — для процессов в состоянии ***готовность***, пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т. е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

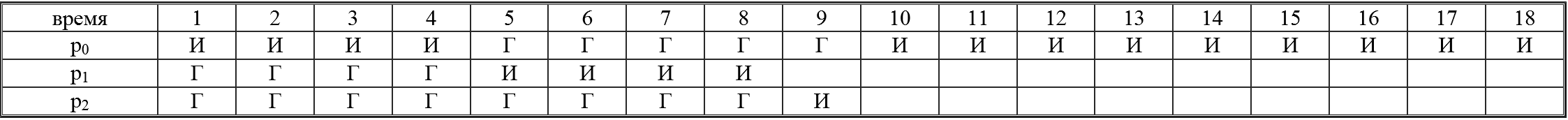


Таблица 2.

Первым для исполнения выбирается процесс p0. Продолжительность его CPU burst больше, чем величина кванта времени, и поэтому процесс исполняется до истечения кванта, т. е. в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p1, p2, p0. Следующим начинает выполняться процесс p1. Время его исполнения совпадает с величиной выделенного кванта, поэтому процесс работает до своего завершения. Теперь очередь процессов в состоянии ***готовность*** состоит из двух процессов p2, p0. Процессор выделяется процессу p2. Он завершается до истечения отпущенного ему процессорного времени, и очередные кванты отмеряются процессу p0 — единственному, не закончившему к этому моменту свою работу. Время ожидания для процесса p0 (количество символов “Г” в соответствующей строке) составляет 5 единиц времени, для процесса p1 — 4 единицы времени, для процесса p2 — 8 единиц времени. Таким образом, среднее время ожидания для этого алгоритма получается равным (5 + 4 + 8)/3 = 5,6(6) единицы времени. Полное время выполнения для процесса p0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p1 — 8 единиц, для процесса p2 — 9 единиц. Среднее полное время выполнения оказывается равным (18 + 8 + 9)/3 = 11,6(6) единицам времени.

Легко видеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для алгоритма FCFS и составляют 2 и 6 единиц времени соответственно.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример c порядком процессов p0, p1, p2 для величины кванта времени равной 1 (см. таблицу 3.). Время ожидания для процесса p0 составит 5 единиц времени, для процесса p1 — тоже 5 единиц, для процесса p2 — 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

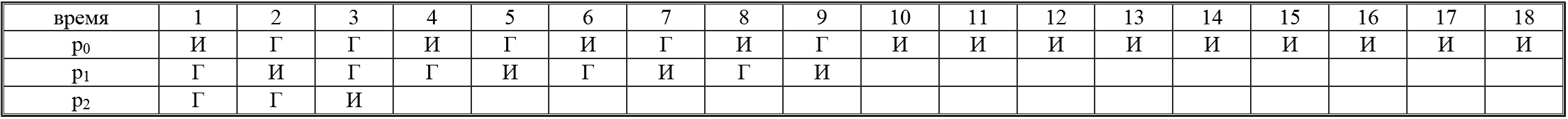


Таблица 3.

При очень больших величинах кванта времени, когда каждый процесс успевает завершить свой CPU burst до возникновения прерывания по времени, алгоритм RR вырождается в алгоритм FCFS. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на своем собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения контекста процессов. В реальных условиях при слишком малой величине кванта времени и, соответственно, слишком частом переключении контекста, накладные расходы на переключение резко снижают производительность системы.

**2.5.3. Shortest-Job-First (SJF)**

При рассмотрении алгоритмов FCFS и RR можно было видеть, насколько существенным для них является порядок расположения процессов в очереди процессов готовых к исполнению. Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы можно было знат время следующих CPU burst для процессов, находящихся в состоянии ***готовность***, то можно бы было выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название “кратчайшая работа первой” или *Shortest Job First* (*SJF*).

SJF алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJF планировании процессор предоставляется избранному процессу на все требующееся ему время, независимо от событий происходящих в вычислительной системе. При вытесняющем SJF планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burst у исполняющегося, то исполняющийся процесс вытесняется новым.

Рассмотрим пример работы невытесняющего алгоритма SJF. Пусть в состоянии ***готовность*** находятся четыре процесса p0, p1, p2 и p3, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 4. Как и прежде, будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода, и что время переключения контекста пренебрежимо мало.



Таблица 4.

При использовании невытесняющего алгоритма SJF первым для исполнения будет выбран процесс p3, имеющий наименьшее значение очередного CPU burst. После его завершения для исполнения выбирается процесс p1, затем p0 и, наконец, p2. Вся эта картина изображена в таблице 5.

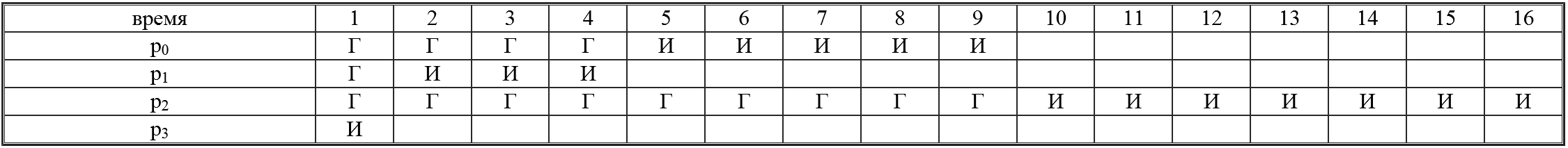


Таблица 5.

Видно, что среднее время ожидания для алгоритма SJF составляет (4 + 1 + 9 + 0)/4 = 3,5 единицы времени. Легко посчитать, что для алгоритма FCFS при порядке процессов p0, p1, p2, p3 эта величина будет равняться (0 + 5 + 8 + 15)/4 = 7 единицам времени, т. е. будет в 2 раза больше, чем для алгоритма SJF. Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) алгоритм SJF является оптимальным с точки зрения минимизации среднего времени ожидания среди класса всех невытесняющих алгоритмов.

Для рассмотрения примера вытесняющего SJF планирования мы возьмем ряд процессов p0, p1, p2 и p3с различными временами CPU burst и различными моментами их появления в очереди процессов готовых к исполнению (см. таблицу 6.).



Таблица 6.

В начальный момент времени в состоянии ***готовность*** находятся только два процесса p0 и p4. Меньшее время очередного CPU burst оказывается у процесса p3, поэтому он и выбирается для исполнения (см. таблицу 7.). По прошествии 2-х единиц времени в систему поступает процесс p1. Время его CPU burst меньше, чем остаток CPU burst у процесса p3, который вытесняется из состояния ***исполнение*** и переводится в состояние ***готовность***. По прошествии еще 2-х единиц времени процесс p1 завершается, и для исполнения вновь выбирается процесс p3. В момент времени t = 6 в очереди процессов готовых к исполнению появляется процесс p2, но поскольку ему для работы нужно 7 единиц времени, а процессу p3 осталось трудиться всего 2 единицы времени, то процесс p3 остается в состоянии ***исполнение***. После его завершения в момент времени t = 7 в очереди находятся процессы p0 и p2, из которых выбирается процесс p0. Наконец, последним получит возможность выполняться процесс p2.

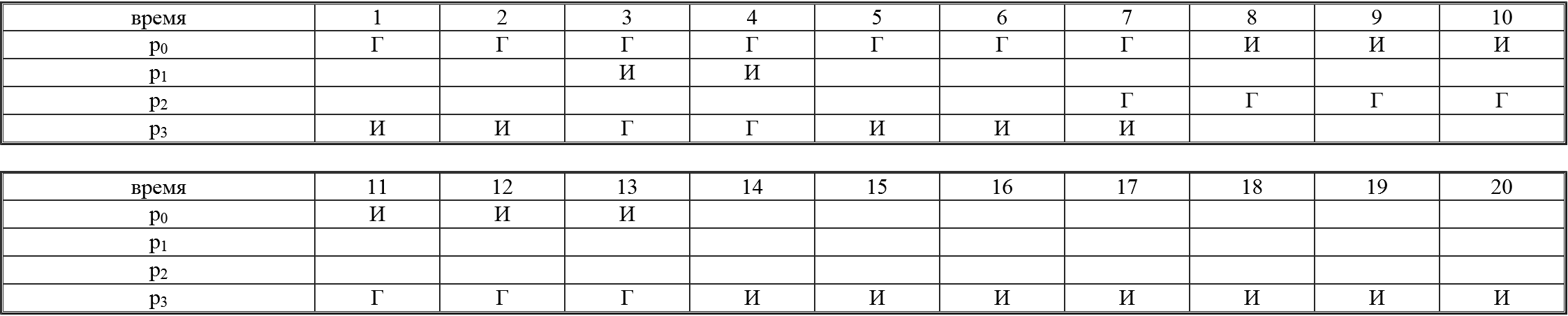


Таблица 7.

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания времени очередного CPU burst для исполняющихся процессов. В пакетных системах количество процессорного времени, требующееся заданию для выполнения, указывает пользователь при формировании задания. Можно взять эту величину для осуществления долгосрочного SJF планирования. Если пользователь укажет больше времени, чем ему нужно, он будет ждать получения результата дольше, чем мог бы, так как задание будет загружено в систему позже. Если же он укажет меньшее количество времени, задача может не досчитаться до конца. Таким образом, в пакетных системах решение задачи оценки времени использования процессора перекладывается на плечи пользователя. При краткосрочном планировании мы можем делать только прогноз длительности следующего CPU burst, исходя из предыстории работы процесса. Пусть t(*n*) – величина *n*-го CPU burst, T(*n* + 1)– предсказываемое значение для *n* + 1-го CPU burst, a – некоторая величина в диапазоне от 0 до 1.

Определим рекуррентное соотношение

http://cs.mipt.ru/docs/courses/osstud/03/images/sr3-f1.gif

T(0) положим произвольной константой. Первое слагаемое учитывает последнее поведение процесса, тогда как второе слагаемое учитывает его предысторию. При a  = 0 мы перестаем следить за последним поведением процесса, фактически полагая

http://cs.mipt.ru/docs/courses/osstud/03/images/sr3-f2.gif

т.е. оценивая все CPU burst одинаково, исходя из некоторого начального предположения.

Положив a  = 1, мы забываем о предыстории процесса. В этом случае мы полагаем, что время очередного CPU burst будет совпадать со временем последнего CPU burst:

http://cs.mipt.ru/docs/courses/osstud/03/images/sr3-f3.gif

Обычно выбирают

http://cs.mipt.ru/docs/courses/osstud/03/images/sr3-f4.gif

для равноценного учета последнего поведения и предыстории. Надо отметить, что такой выбор a удобен и для быстрой организации вычисления оценки T(*n* + 1). Для подсчета новой оценки нужно взять старую оценку, сложить с измеренным временем CPU burst и полученную сумму разделить на 2, например, с помощью ее сдвига на 1 бит вправо. Полученные оценки T(*n* + 1) применяются как продолжительности очередных промежутков времени непрерывного использования процессора для краткосрочного SJF планирования.

**2.5.4. Гарантированное планирование**

При интерактивной работе N пользователей в вычислительной системе можно применить алгоритм планирования, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении ~ 1/N часть процессорного времени. Пронумеруем всех пользователей от 1 до N. Для каждого пользователя с номером i введем две величины: Ti - время нахождения пользователя в системе, или, другими словами длительность сеанса его общения с машиной, и ti - суммарное процессорное время, уже выделенное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение Ti/N процессорного времени. Если

http://cs.mipt.ru/docs/courses/osstud/03/images/sr3-f5.gif

то *i* - й пользователь несправедливо обделен процессорным временем. Если же

http://cs.mipt.ru/docs/courses/osstud/03/images/sr3-f6.gif

то система явно благоволит к пользователю с номером *i*. Вычислим для каждого пользовательского процесса значение коэффициента справедливости

http://cs.mipt.ru/docs/courses/osstud/03/images/sr3-f7.gif

и будем предоставлять очередной квант времени процессу с наименьшей величиной этого отношения. Предложенный алгоритм называют алгоритмом гарантированного планирования. К недостаткам этого алгоритма можно отнести невозможность предугадать поведение пользователей. Если некоторый пользователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно много процессорного времени.

**2.5.5. Приоритетное планирование**

Алгоритмы SJF и гарантированного планирования представляют собой частные случаи приоритетного планирования. При приоритетном планировании каждому процессу присваивается определенное числовое значение — приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс. Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости. Чем он меньше, тем больше приоритет у процесса.

Принципы назначения приоритетов могут опираться как на внутренние критерии вычислительной системы, так и на внешние по отношению к ней. Внутренние используют различные количественные и качественные характеристики процесса для вычисления его приоритета. Это могут быть, например, определенные ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей I/O burst к CPU burst и т. д. Внешние критерии исходят из таких параметров, как важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и других политических факторов. Высокий внешний приоритет может быть присвоен задаче лектора или того, кто заплатил $100 за работу в течение одного часа.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов приоритетного планирования.

Пусть в очередь процессов, находящихся в состоянии ***готовность***, поступают те же процессы, что и в примере из раздела 2.5.3. для вытесняющего алгоритма SJF, только им дополнительно еще присвоены приоритеты (см. таблицу 8.). В вычислительных системах не существует определенного соглашения, какое значение приоритета - 1 или 4 считать более приоритетным. Во избежание путаницы, во всех наших примерах мы будем предполагать, что большее значение соответствует меньшему приоритету, т.е. наиболее приоритетным в нашем примере является процесс p3, а наименее приоритетным — процесс p0.

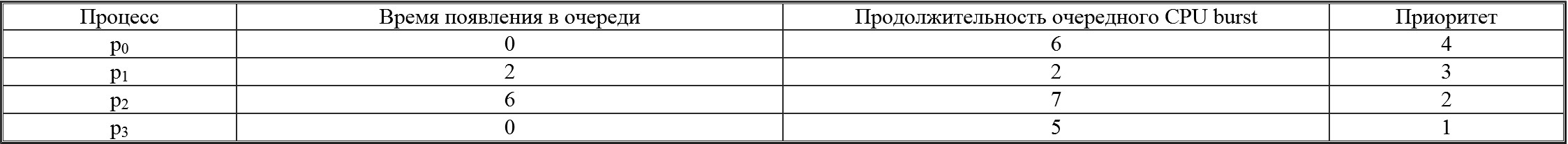


Таблица 8.

Как будут вести себя процессы при использовании невытесняющего приоритетного планирования? Первым для выполнения в момент времени t = 0 выбирается процесс p3, как обладающий наивысшим приоритетом. После его завершения в момент времени t = 5 в очереди процессов готовых к исполнению окажутся два процесса p0 и p1. Больший приоритет из них у процесса p1 он и начнет выполняться (см. таблицу 9.). Затем в момент времени t = 8 для исполнения будет избран процесс p2 и лишь потом процесс p0.

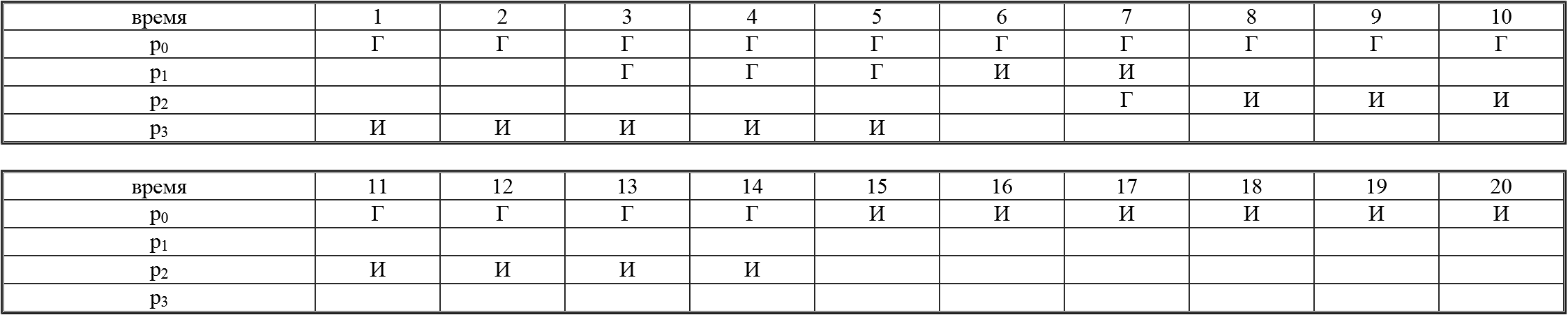


Таблица 9.

Иным будет предоставление процессора процессам в случае вытесняющего приоритетного планирования (см. таблицу 10.). Первым, как и в предыдущем случае, исполняться начнет процесс p3, а по его окончании процесс p1. Однако в момент времени t = 6 он будет вытеснен процессом p2 и продолжит свое выполнение только в момент времени t = 13. Последним, как и раньше будет исполнен процесс p0.

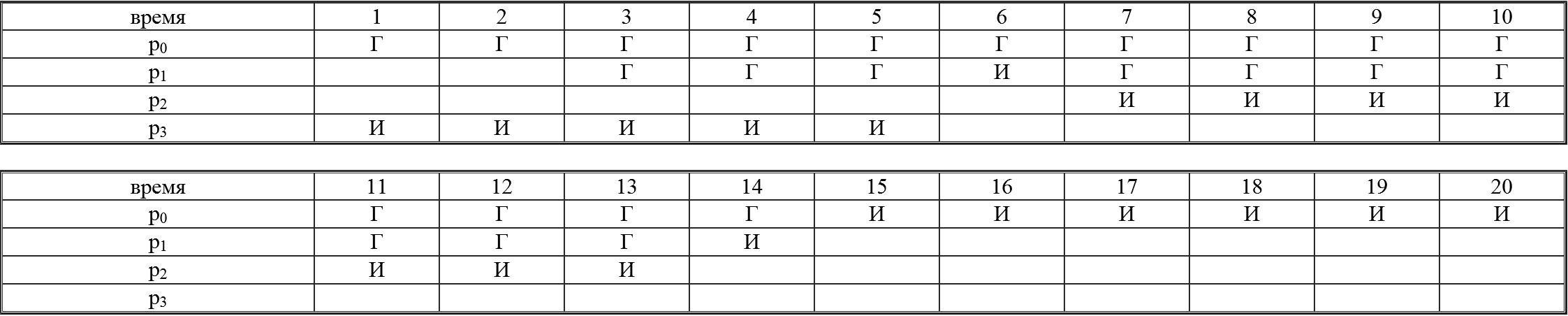


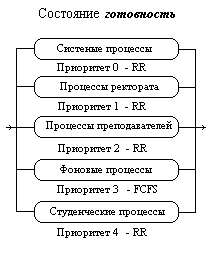
Таблица 10.

В рассмотренном выше примере приоритеты процессов не изменялись с течением временем. Такие приоритеты принято называть статическими. Механизмы статической приоритетности легко реализовать, и они сопряжены с относительно небольшими издержками на выбор наиболее приоритетного процесса. Однако статические приоритеты не реагируют на изменения ситуации в вычислительной системе, которые могут сделать желательной корректировку порядка исполнения процессов. Более гибкими являются динамические приоритеты процессов, изменяющие свои значения по ходу исполнения процессов. Начальное значение динамического приоритета, присвоенное процессу, действует в течение лишь короткого периода времени, после чего ему назначается новое, более подходящее значение. Изменение динамического приоритета процесса является единственной операцией над процессами, которую мы до сих пор не рассмотрели. Как правило, изменение приоритета процессов проводится согласованно с совершением каких-либо других операций: при рождении нового процесса, при разблокировке или блокировании процесса, по истечении определенного кванта времени или по завершении процесса. Примерами алгоритмов с динамическими приоритетами являются алгоритм SJF и алгоритм гарантированного планирования. Схемы с динамической приоритетностью гораздо сложнее в реализации и связанны с большими издержками по сравнению со статическими схемами. Однако их использование предполагает, что эти издержки оправдываются улучшением поведения системы.

Главная проблема приоритетного планирования заключается в том, что при ненадлежащем выборе механизма назначения и изменения приоритетов низкоприоритетные процессы могут быть не запущены неопределенно долгое время. Обычно случается одно из двух. Или они все же дожидаются своей очереди на исполнение (в девять часов утра в воскресенье, когда все приличные программисты ложатся спать). Или вычислительную систему приходится выключать, и они теряются (при остановке IBM 7094 в Массачусетском технологическом институте в 1973 году были найдены процессы, запущенные в 1967 году и ни разу с тех пор не исполнявшиеся). Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения приоритета процесса, находящегося в состоянии ***готовность***. Пусть изначально процессам присваиваются приоритеты от 128 до 255. Каждый раз, по истечении определенного промежутка времени, значения приоритетов готовых процессов уменьшаются на 1. Процессу, побывавшему в состоянии ***исполнение***, восстанавливается первоначальное значение приоритета. Даже такая грубая схема гарантирует, что любому процессу в разумные сроки будет предоставлено право на исполнение.

**2.5.6. Многоуровневые очереди (Multilevel Queue)**

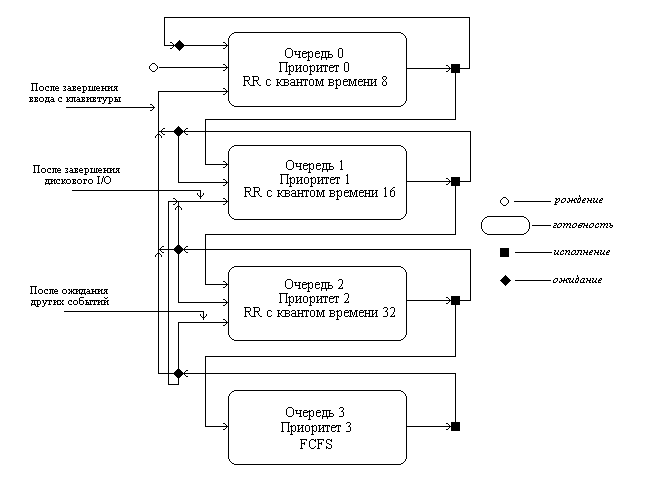
Для систем, в которых процессы могут быть легко рассортированы на разные группы, был разработан другой класс алгоритмов планирования. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии ***готовность*** (см. рисунок 5). Этим очередям приписываются фиксированные приоритеты. Например, приоритет очереди системных процессов устанавливается больше, чем приоритет очередей пользовательских процессов. А приоритет очереди процессов, запущенных студентами, — ниже, чем для очереди процессов, запущенных преподавателями. Это значит, что ни один пользовательский процесс не будет выбран для исполнения, пока есть хоть один готовый системный процесс, и ни один студенческий процесс не получит в свое распоряжение процессор, если есть процессы преподавателей, готовые к исполнению. Внутри этих очередей для планирования могут применяться самые разные алгоритмы. Так, например, для больших счетных процессов, не требующих взаимодействия с пользователем (*фоновых* процессов), может использоваться алгоритм FCFS, а для интерактивных процессов – алгоритм RR. Подобный подход, получивший название многоуровневых очередей, повышает гибкость планирования: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм.

  
Рис. 5. Несколько очередей планирования.

**2.5.7. Многоуровневые очереди с обратной связью (Multilevel Feedback Queue)**

Дальнейшим развитием алгоритма многоуровневых очередей является добавление к нему механизма обратной связи. Здесь процесс не постоянно приписан к определенной очереди, а может мигрировать из очереди в очередь, в зависимости от своего поведения.

Для простоты рассмотрим ситуацию, когда процессы в состоянии ***готовность*** организованы в 4 очереди, как на рисунке 6. Планирование процессов между очередями осуществляется на основе вытесняющего приоритетного механизма. Чем выше на рисунке располагается очередь, тем выше ее приоритет. Процессы в очереди 1 не могут исполняться, если в очереди 0 есть хотя бы один процесс. Процессы в очереди 2 не будут выбраны для выполнения, пока есть хоть один процесс в очередях 0 и 1. И, наконец, процесс в очереди 3 может получить процессор в свое распоряжение только тогда, когда очереди 0, 1 и 2 пусты. Если при работе процесса появляется другой процесс в какой-либо более приоритетной очереди, исполняющийся процесс вытесняется появившимся. Планирование процессов внутри очередей 0—2 осуществляется с использованием алгоритма RR, планирование процессов в очереди 3 основывается на алгоритме FCFS.

  
Рис. 3.6.Схема миграции процессов в многоуровневых очередях планирования с обратной связью. Вытеснение процессов более приоритетными процессами и завершение процессов на схеме не показано.

Родившийся процесс поступает в очередь 0. При выборе на исполнение он получает в свое распоряжение квант времени размером 8 единиц. Если продолжительность его CPU burst меньше этого кванта времени, процесс остается в очереди 0. В противном случае, он переходит в очередь 1. Для процессов из очереди 1 квант времени имеет величину 16. Если процесс не укладывается в это время, он переходит в очередь 2. Если укладывается — остается в очереди 1. В очереди 2 величина кванта времени составляет 32 единицы. Если и этого мало для непрерывной работы процесса, процесс поступает в очередь 3, для которой квантование времени не применяется, и, при отсутствии готовых процессов в других очередях, он может исполняться до окончания своего CPU burst. Чем больше значение продолжительности CPU burst, тем в менее приоритетную очередь попадает процесс, но тем на большее процессорное время он может рассчитывать для своего выполнения. Таким образом, через некоторое время все процессы, требующие малого времени работы процессора окажутся размещенными в высокоприоритетных очередях, а все процессы, требующие большого счета и с низкими запросами к времени отклика, — в низкоприоритетных.

Миграция процессов в обратном направлении может осуществляться по различным принципам. Например, после завершения ожидания ввода с клавиатуры процессы из очередей 1, 2 и 3 могут помещаться в очередь 0, после завершения дисковых операций ввода-вывода процессы из очередей 2 и 3 могут помещаться в очередь 1, а после завершения ожидания всех других событий из очереди 3 в очередь 2. Перемещение процессов из очередей с низкими приоритетами в очереди с большими приоритетами позволяет более полно учитывать изменение поведения процессов с течением времени.

Многоуровневые очереди с обратной связью представляют собой наиболее общий подход к планированию процессов из числа подходов, рассмотренных нами. Они наиболее трудоемки в реализации, но в то же время они обладают наибольшей гибкостью. Понятно, что существует много других разновидностей такого способа планирования помимо варианта, приведенного выше. Для полного описания их конкретного воплощения необходимо указать:

* Количество очередей для процессов, находящихся в состоянии ***готовность***.
* Алгоритм планирования, действующий между очередями.
* Алгоритмы планирования, действующие внутри очередей.
* Правила помещения родившегося процесса в одну из очередей.
* Правила перевода процессов из одной очереди в другую.

Изменяя какой-либо из перечисленных пунктов, можно существенно менять поведение вычислительной системы.

**2.6. Выводы**

Одним из наиболее ограниченных ресурсов вычислительной системы является процессорное время. Для его распределения между многочисленными процессами в системе приходится применять процедуру планирования процессов. По степени длительности влияния планирования на поведение вычислительной системы различают краткосрочное, среднесрочное и долгосрочное планирование процессов. Конкретные алгоритмы планирования процессов зависят от поставленных целей, класса решаемых задач и опираются на статические и динамические параметры процессов и компьютерных систем. Различают вытесняющий и невытесняющий режимы планирования. При невытесняющем планировании исполняющийся процесс уступает процессор другому процессу только по своему собственному желанию, при вытесняющем планировании исполняющийся процесс может быть вытеснен из состояния исполнения помимо собственной воли.

Простейшим алгоритмом планирования является невытесняющий алгоритм FCFS, который, однако, может существенно задерживать короткие процессы, не вовремя перешедшие в состояние ***готовность***. В системах разделения времени широкое распространение получила вытесняющего версия этого алгоритма — RR.

Среди всех невытесняющих алгоритмов оптимальным с точки зрения среднего времени ожидания процессов является алгоритм SJF. Существует и вытесняющий вариант этого алгоритма. В интерактивных системах часто используется алгоритм гарантированного планирования, обеспечивающий пользователям равные части процессорного времени.

Алгоритм SJF и алгоритм гарантированного планирования являются частными случаями планирования с использованием приоритетов. В более общих методах приоритетного планирования применяются многоуровневые очереди процессов готовых к исполнению и многоуровневые очереди с обратной связью. Будучи наиболее сложными в реализации, эти способы планирования обеспечивают гибкое поведение вычислительных систем и их адаптивность к решению задач разных классов.

**3. Задания для выполнения**

Даны 10 процессов. Составить таблицу готовности исполнения процессов, подсчитать среднее время ожидания и среднее время выполнения, исходя из индивидуального варианта (столбец X) и при условии, что большее значение в столбце «Приоритет» соответствует меньшему приоритету. Рассмотреть вариант задания как для вытесняющего, так и для невытесняющего алгоритмов. Составить программы по каждому варианту.

|  |  |  |  |
| --- | --- | --- | --- |
| Процесс | Время появления в очереди | Приоритет | **X** Продолжительность очередного CPU burst |
| p0 | **0** | **10** | **X** |
| p1 | **2** | **9** | **X** |
| p2 | **4** | **8** | **X** |
| p3 | **6** | **7** | **X** |
| p4 | **8** | **6** | **X** |
| p5 | **8** | **5** | **X** |
| p6 | **6** | **4** | **X** |
| p7 | **4** | **3** | **X** |
| p8 | **2** | **2** | **X** |
| p9 | **0** | **1** | **X** |

**4. Варианты индивидуальных заданий.**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Номер в журнале** | | | | | | | | | |
| Процесс/ **X** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| p0 | **10** | **9** | **8** | **7** | **10** | **5** | **10** | **3** | **7** | **1** |
| p1 | **8** | **1** | **6** | **3** | **9** | **4** | **9** | **7** | **1** | **10** |
| p2 | **4** | **6** | **5** | **4** | **8** | **3** | **7** | **5** | **3** | **9** |
| p3 | **5** | **2** | **4** | **5** | **7** | **2** | **8** | **1** | **5** | **8** |
| p4 | **6** | **8** | **3** | **2** | **6** | **1** | **5** | **2** | **9** | **6** |
| p5 | **9** | **7** | **7** | **1** | **5** | **6** | **6** | **4** | **10** | **5** |
| p6 | **1** | **3** | **9** | **9** | **4** | **7** | **3** | **6** | **8** | **2** |
| p7 | **2** | **9** | **10** | **8** | **3** | **8** | **4** | **8** | **6** | **4** |
| p8 | **3** | **4** | **2** | **6** | **2** | **9** | **1** | **10** | **4** | **7** |
| p9 | **7** | **10** | **1** | **10** | **1** | **10** | **2** | **9** | **2** | **3** |
| Алгоритм | SJF | FCFS | SJF | FCFS | SJF | FCFS | SJF | FCFS | SJF | FCFS |
| Планирование | НП | НП | НП | НП | НП | ВП | ВП | ВП | ВП | ВП |

***Условные обозначения:***

Вытесняющий алгоритм **SJF**

First-Come, First-Served **FCFS**

Невытесняющее планирование **НП**

Вытесняющее планирование **ВП**