

# Интерпретатор командного языка shell

Shell - командный язык, могущий выполнять как команды, введенные с терминала, так и команды, хранящиеся в файле.

## 1. Основные понятия языка shell

### 1.1. Ввод-вывод

Три направления ввода-вывода являются выделенными - стандартный ввод, стандартный вывод и стандартный протокол. Как правило, команды берут исходные данные из стандартного ввода и помещают результаты в стандартный вывод.

Стандартные ввод, вывод и протокол можно переназначить. Обозначение

< <имя файла>

служит для переназначения стандартного ввода (дескриптор файла 0),

> <имя файла>

для стандартного вывода (дескриптор файла 1);

<< <строка>

ввод происходит со стандартного ввода, пока не встретится указанная <строка> или конец файла,

>> <имя файла>

для стандартного вывода; если файл существует, то выводимая информация добавляется к концу этого файла,

<& <цифра>

в качестве стандартного ввода об(г)является файл, ассоциированный с дескриптором <цифра>; аналогично для стандартного вывода

>& <цифра>  
<&- и >&-

закрывают соответственно стандартный ввод и вывод.

Если любой из этих конструкций предшествует цифра, то с указанным файлом будет ассоциирован дескриптор, равный указанной цифре, вместо 0 и 1 по умолчанию. Например,

2 > <имя файла>

для стандартного протокола используется дескриптор 2, а

```
2 >& 1
```

ассоциирует дескриптор 2 с файлом, ассоциированным с дескриптором 1.

```
... 2>protocol
```

переназначает стандартный протокол (дескриптор 2) в файл по имени protocol.

Чтобы переназначить стандартный протокол туда же, куда уже назначен стандартный вывод, следует употребить конструкцию

```
... 2>&1
```

Важен порядок переназначения: shell производит переназначение слева направо по указанному списку. Так,

```
1 > xxx 2 >& 1
```

сначала ассоциирует дескриптор 1 с файлом xxx, а затем дескриптор 2 с 1, т.е. тоже с xxx. А

```
2 >& 1 1 > xxx
```

ассоциирует дескриптор 2 с терминалом, а 1 - с файлом xxx.

Можно переназначить системный ввод на текущий файл:

```
isql - - <
```

## 1.2. Синхронное и асинхронное выполнение команд

Обычно shell ждет завершения выполнения команды. Однако имеется возможность запустить задачу в асинхронном режиме, т.е. без ожидания ее завершения. Для этого после команды (после всех ее аргументов и указаний о переназначении ввода-вывода) надо поставить знак &. При этом по умолчанию стандартный ввод команды назначается на пустой файл /dev/null.

Пример: создать файл primer можно по команде

```
echo > primer
```

Еще пример: запустить программу prog в асинхронном режиме, чтобы не надо было дожидаться его завершения, засечь время выполнения, результаты программы направить в файл prog.res, данные о времени выполнения - в файл prog.tim.

```
time prog > prog.res 2> prog.tim &
```

## 1.3. Конвейер

Конвейер - последовательность команд, разделенных знаком |. Если после конвейера стоит ; shell ждет его завершения. Если & - то не ждет. Роль ; может играть конец строки. Смысл конвейера в том, что стандартный вывод одной команды замыкается на стандартный ввод другой. Пример конвейера - подсчитать число об(р)ектных файлов в текущем каталоге.

```
ls *.o | wc -l
```

## 1.4. Метасимволы, генерация имен файлов

Метасимволы - символы, имеющие специальное значение для интерпретатора :

? \* ; & ( ) | ^ < > <пробел> <табуляция> <возврат\_каретки>

Однако каждый из этих символов может представлять самого себя, если перед ним стоит \. Все символы, заключенные между кавычками ' и ', представляют самих себя. Между двойными кавычками (") выполняются подстановки команд (см п. 2.2) и параметров (см. п. 2.3), а символы \, `," и \$ могут экранироваться предшествующим символом \.

После всех подстановок в каждом слове команды ищутся символы \*,?, и [. Если находится хотя бы один из них, то это слово рассматривается как шаблон имен файлов и заменяется именами файлов, удовлетворяющих данному шаблону (в алфавитном порядке). Если ни одно имя файла не удовлетворяет шаблону, то он остается неизменным. Значения указанных символов:

\* любая строка, включая и пустую

? один любой символ

[...] любой из указанных между ними символов. Пара символов, разделенных знаком -, означает любой символ, который находится между ними, включая и их самих. Если первым символом после "[" идет "!", то указанные символы не должны входить в имя файла

## 2. Синтаксис языка shell

### 2.1. Комментарии

Строки, начинающиеся с #, трактуются как комментарии.

### 2.2. Подстановка результатов выполнения команд

Выражения можно заключать в обратные кавычки (`). Такие выражения вычисляются в месте использования. Они могут быть, например, частью строк. Пример. Пусть параметром макрокоманды является имя файла с расширением .log. Требуется удалить одноименный файл с расширением .err.

```
name=`ena -n $1`  
rm -f ${name}.err
```

Значение, полученное в результате выполнения команды

```
ena -n $1
```

присваивается переменной name. Фигурные скобки использованы для выделения аргумента операции перехода от имени к значению. Без них .err приклеилась бы к имени.

### 2.3. Переменные и подстановка их значений

Все переменные в языке shell - текстовые. Их имена должны начинаться с буквы и состоять из латинских букв, цифр и знака подчеркивания (\_). Чтобы воспользоваться значением переменной, надо перед ней поставить символ \$. Использование значения переменной называется подстановкой.

Различается два класса переменных: позиционные и с именем. Позиционные переменные - это аргументы командных файлов, их именами служат цифры: \$0 - имя команды, \$1 - первый аргумент и т.д. Значения позиционным переменным могут быть присвоены и командой set (см. Специальные команды). Пример. После вызова программы на shelle, хранящейся в файле ficofl:

```
ficofl -d / \*.for
```

значением \$0 будет ficofl, \$1 - -d, \$2 - /, \$3 - \*.for, значения остальных позиционных переменных будут пустыми строками. Заметим, что если бы символ \* при вызове ficofl не был экранирован, в качестве аргументов передались бы имена всех фортранных файлов текущей директории.

Еще две переменные хранят командную строку за исключением имени команды: @\$ эквивалентно \$1 \$2 ..., а \$\* - "\$1 \$2 ...". Начальные значения переменным с именем могут быть установлены следующим образом:

```
<имя>=<значение> [ <имя>=<значение> ] ...
```

Не может быть одновременно функции (см. [Управляющие конструкции](#)) и переменной с одинаковыми именами. Для подстановки значений переменных возможны также следующие конструкции:

```
${<переменная>}
```

если значение <переменной> определено, то оно подставляется. Скобки применяются лишь если за <переменной> следует символ, который без скобок приклеится к имени.

```
${<переменная>:-<слово>}
```

если <переменная> определена и не является пустой строкой, то подставляется ее значение; иначе подставляется <слово>.

```
${<переменная>:=<слово>}
```

если <переменная> не определена или является пустой строкой, ей присваивается значение <слово>; после этого подставляется ее значение.

```
${<переменная>:?<слово>}
```

если <переменная> определена и не является пустой строкой, то подставляется ее значение; иначе на стандартный вывод выводится <слово> и выполнение shella завершается. Если <слово> опущено, то выдается сообщение "parameter null or not set".

```
${<переменная>:+<слово>}
```

если <переменная> определена и не является пустой строкой, то подставляется <слово>; иначе подставляется пустая строка.

Пример: если переменная d не определена или является пустой строкой, то выполняется команда pwd

```
echo ${d:-`pwd`}
```

Следующие переменные автоматически устанавливаются shell'ом:

- # количество позиционных параметров (десятичное)
- флаги, указанные при запуске shella или командой set
- ? десятичное значение, возвращенное предыдущей синхронно выполненной командой
- \$ номер текущего процесса
- ! номер последнего асинхронного процесса
- @ эквивалентно \$1 \$2 \$3 ...
- \* эквивалентно "\$1 \$2 \$3 ..."

Напомним: чтобы получить значения этих переменных, перед ними нужно поставить знак \$. Пример: выдать номер текущего процесса:

```
echo $$
```

## 2.4. Специальные переменные

Shell'ом используются следующие специальные переменные:

HOME	директория, в которую пользователь попадает при входе в систему или при выполнении команды <code>cd</code> без аргументов
PATH	список полных имен каталогов, в которых ищется файл при указании его неполного имени.
PS1	основная строка приглашения (по умолчанию \$)
PS2	дополнительная строка приглашения (по умолчанию >); в интерактивном режиме перед вводом команды shell'ом выводится основная строка приглашения. Если нажата клавиша <code>new_line</code> , но для завершения команды требуется дальнейший ввод, то выводится дополнительная строка приглашения
IFS	последовательность символов, являющихся разделителями в командной строке (по умолчанию это <пробел>, <табуляция> и <возврат_каретки>)

## 3. Управляющие конструкции

Простая команда - это последовательность слов, разделенная пробелами. Первое слово является именем команды, которая будет выполняться, а остальные будут переданы ей как аргументы. Имя команды передается ей как аргумент номер 0 (т.е. имя команды является значением \$0). Значение, возвращаемое простой командой - это ее статус завершения, если она завершилась нормально, или (восьмеричное) 200+статус, если она завершилась аварийно.

Список - это последовательность одного или нескольких конвейеров, разделенных символами `;`, `&`, `&&` или `||` и быть может заканчивающаяся символом `;` или `&`. Из четырех указанных операций `;` и `&` имеют равные приоритеты, меньшие, чем у `&&` и `||`. Приоритеты последних также равны между собой. Символ `;` означает, что конвейеры будут выполняться последовательно, а `&` - параллельно. Операция `&&` (`||`) означает, что список, следующий за ней будет выполняться лишь в том случае, если код завершения предыдущего конвейера нулевой (ненулевой).

Команда - это либо простая команда, либо одна из управляющих конструкций. Кодом завершения команды является код завершения ее последней простой команды.

### 3.1. Цикл ДЛЯ

```
for <переменная> [ in <набор> ]
do
<список>
done
```

Если часть `in <набор>` опущена, то это означает `in "$@"` (то есть `in $1 $2 ... $n`). Пример. Вывести на экран все фортранные файлы текущей библиотеки:

```
for f in *.for
do
cat $f
done
```

### 3.2. Оператор выбора

```
case $<переменная> in
```

```

        <шаблон> | <шаблон>... ) <список> ;;
        . . .
    esac

```

Оператор выбора выполняет <список>, соответствующий первому <шаблону>, которому удовлетворяет <переменная>. Форма шаблона та же, что и используемая для генерации имен файлов. Часть | шаблон... может отсутствовать.

Пример. Определить флаги и откомпилировать все указанные файлы.

```

#      инициализировать флаг
flag=
#      повторять для каждого аргумента
for a
do
    case $a in
        # об(р)единить флаги, разделив их пробелами
        -[ocSO]) flag=$flag' ' $a ;;
        -*) echo 'unknown flag $a' ;;
        # компилировать каждый исходный файл и сбросить флаги
        *.c) cc $flag $a; flag= ;;
        *.s) as $flag $a; flag= ;;
        *.f) f77 $flag $a; flag= ;;
        # неверный аргумент
        *) echo 'unexpected argument $a' ;;
    esac
done

```

### 3.3. Условный оператор.

```

    if <список1>
    then
        <список2>
    [ elif <список3>
    then
        <список4> ]
    . . .
    [ else
        <список5> ]
    fi

```

Выполняется <список1> и, если код его завершения 0, то выполняется <список2>, иначе - <список3> и, если и его код завершения 0, то выполняется <список4>. Если же это не так, то выполняется <список5>. Части elif и else могут отсутствовать.

### 3.4. Цикл ПОКА

```

while <список1>
do
    <список2>
done

```

До тех пор, пока код завершения последней команды <списка1> есть 0, выполняются команды <списка2>. При замене служебного слова while на until условие выхода из цикла меняется на противоположное.

В качестве одной из команд <списка1> может быть команда true (false). По этой команде не выполняется никаких действий, а код завершения устанавливается 0 (-1). Эти команды применяются для организации бесконечных циклов. Выход из такого цикла можно осуществить лишь по команде break (см. Специальные команды).

### 3.5. Функции

```

<имя> () {
    <список>;
}

```

Определяется функция с именем <имя>. Тело функции - <список>, заключенный между { и }.

### 3.6. Зарезервированные слова

Следующие слова являются зарезервированными:

```
if      then    else    elif    fi
case   in      esac    { }
for    while   until   do      done
```

### 3.7. Специальные команды

Как правило, для выполнения каждой команды shell порождает отдельный процесс. Специальные команды отличаются тем, что они встроены в shell и выполняются в рамках текущего процесса.

:	Пустая команда. Возвращает нулевой код завершения.
. file	Shell читает и выполняет команды из файла file, затем завершается; при поиске file используется список поиска \$PATH.
break [n]	Выход из внутреннего for или while цикла; если указано n, то выход из n внутренних циклов.
continue [n]	Перейти к следующей итерации внутреннего for или while цикла; если указано n, то переход к следующей итерации n-ого цикла.
cd [ <аргумент> ]	Сменить текущую директорию на директорию <аргумент>. По умолчанию используется значение HOME.
echo [ <arg> ... ]	Выводит свои аргументы в стандартный вывод, разделяя их пробелами.
eval [ <arg> ... ]	Аргументы читаются, как если бы они поступали из стандартного ввода и рассматриваются как команды, которые тут же и выполняются.
exec [ <arg> ... ]	Аргументы рассматриваются как команды shell'a и тут же выполняются, но при этом не создается нового процесса. В качестве аргументов могут быть указаны направления ввода-вывода и, если нет никаких других аргументов, то будет изменено лишь направление ввода-вывода текущей программы.
exit [ n ]	Завершение выполнения shell'a с кодом завершения n. Если n опущено, то кодом завершения будет код завершения последней выполненной команды (конец файла также приводит к завершению выполнения).
export [ <переменная> ... ]	Данные переменные отмечаются для автоматического экспорта в окружение (см. Окружение) выполняемых команд. Если аргументы не указаны, то выводится список всех экспортируемых переменных. Имена функций не могут экспортироваться.
hash [ -r ] [ <команда> ... ]	Для каждой из указанных команд определяется и запоминается путь поиска. Опция -r удаляет все запомненные данные. Если не указан ни один аргумент, то выводится информация о запомненных командах: hits - количество обращений shell'a к данной команде; cost - объем работы для обнаружения команды в списке поиска; command - полное имя команды. В некоторых ситуациях происходит перевычисление запомненных данных, что отмечается значком * в поле hits.
pwd	Выводит имя текущей директории.
read [ <переменная> ... ]	Читается из стандартного ввода одна строка; первое ее слово присваивается первой переменной, второе - второй и т.д., причем все оставшиеся слова присваиваются последней переменной.
readonly [ <переменная> ... ]	Запрещается изменение значений указанных переменных. Если аргумент не указан, то выводится информация обо всех переменных типа readonly.
return [ n ]	Выход из функции с кодом завершения n. Если n опущено, то кодом завершения будет код завершения последней выполненной команды.
set [ --aefkntuvx [ <arg> ... ] ]	Команда устанавливает следующие режимы:
-a	отметить переменные, которые были изменены или созданы, как переменные окружения (см. Окружение)
-e	если код завершения команды ненулевой, то немедленно завершить выполнение shell'a

-f	запретить генерацию имен файлов
-k	все переменные с именем помещаются в окружение команды, а не только те, что предшествуют имени команды (см. Окружение)
-n	читать команды, но не выполнять их
-t	завершение shell'a после ввода и выполнения одной команды
-u	при подстановке рассматривать неустановленные переменные как ошибки
-v	вывести вводимые строки сразу после их ввода
-x	вывести команды и их аргументы перед их выполнением
--	не изменяет флаги, полезен для присваивания позиционным переменным новых значений. При указании + вместо - каждый из флагов устанавливает противоположный режим. Набор текущих флагов есть значение переменной \$-. <arg> - это значения, которые будут присвоены позиционным переменным \$1, \$2 и т.д. Если все аргументы опущены, выводятся значения всех переменных.
shift [ n ]	Позиционные переменные, начиная с \$(n+1), переименовываются в \$1 и т.д. По умолчанию n=1.
test	вычисляет условные выражения (см. Дополнительные сведения. Test )
trap [ <arg> ] [ n ] ...	Команда <arg> будет выполнена, когда shell получит сигнал n (см. Сигналы). (Надо заметить, что <arg> проверяется при установке прерывания и при получении сигнала). Команды выполняются по порядку номеров сигналов. Любая попытка установить сигнал, игнорируемый данным процессом, не обрабатывается. Попытка прерывания по сигналу 11 (segmentation violation) приводит к ошибке. Если <arg> опущен, то все прерывания устанавливаются в их начальные значения. Если <arg> есть пустая строка, то этот сигнал игнорируется shell'ом и вызываемыми им программами. Если n=0, то <arg> выполняется при выходе из shell'a. Тrap без аргументов выводит список команд, связанных с каждым сигналом.
type [ <имя> ... ]	Для каждого имени показывает, как оно будет интерпретироваться при использовании в качестве имени команды: как внутренняя команда shell'a, как имя файла или же такого файла нет вообще.
ulimit [ -f ] [ n ]	Устанавливает размер файла в n блоков; -f - устанавливает размер файла, который может быть записан процессом-потомком (читать можно любые файлы). Без аргументов - выводит текущий размер.
umask [ nnn ]	Пользовательская маска создания файлов изменяется на nnn. Если nnn опущено, то выводится текущее значение маски. Пример: после команды umask 755 будут создаваться файлы, которые владелец сможет читать, писать и выполнять, а все остальные - только читать и выполнять.
unset [ <имя> ... ]	Для каждого имени удаляет соответствующую переменную или функцию. Переменные PATH, PS1, PS2 и IFS не могут быть удалены.
wait [ n ]	Ждет завершения указанного процесса и выводит код его завершения. Если n не указано, то ожидается завершения всех активных процессов-потомков и возвращается код завершения 0.

## 4. Выполнение shell-программ

### 4.1. Запуск shell'a

Программа, интерпретирующая shell-программы, находится в файле /bin/sh. При запуске ее первый аргумент является именем shell-программы, остальные передаются как позиционные параметры. Если файл, содержащий shell-программу, имеет право выполнения (x), то достаточно указания лишь его имени. Например, следующие две команды операционной системы эквивалентны (если файл ficofl обладает указанным правом и на самом деле содержит shell-программу):

```
sh ficofl -d . g\*
и
ficofl -d . g\*
```

### 4.2. Выполнение



При выполнении shell-программ выполняются все подстановки. Если имя команды совпадает с именем специальной команды, то она выполняется в рамках текущего процесса. Так же выполняются и определенные пользователем функции. Если имя команды не совпадает ни с именем специальной команды, ни с именем функции, то порождается новый процесс и осуществляется попытка выполнить указанную команду.

Переменная PATH определяет путь поиска директории, содержащей данную команду. По умолчанию это

```
::/bin:/usr/ bin:/util:/dss/rk
```

Директории поиска разделяются двоеточиями; :: означает текущую директорию. Если имя команды содержит символ /, значение \$PATH не используется: имена, начинающиеся с / ищутся от корня, остальные - от текущей директории. Положение найденной команды запоминается shell'ом и может быть опрошено командой hash.

### 4.3. Окружение

Окружение - это набор пар имя-значение, которые передаются выполняемой программе. Shell взаимодействует с окружением несколькими способами. При запуске shell создает переменную для каждой указанной пары, придавая ей соответствующее значение. Если вы измените значение какой-либо из этих переменных или создадите новую переменную, то это не окажет никакого влияния на окружение, если не будет использована команда export для связи переменной shell'a с окружением (см. также set -a). Переменная может быть удалена из окружения командой unset (см.). Таким образом, окружение каждой из выполняемых shell'ом команд формируется из всех неизменных пар имя-значение, первоначально полученных shell'ом, минус пары, удаленные командой unset, плюс все модифицированные и измененные пары, которые для этого должны быть указаны в команде export.

Окружение простых команд может быть сформировано указанием перед ней одного или нескольких присваиваний переменным. Так,

```
TERM=d460 <команда>  
и  
(export TERM; TERM=d460; <команда>)
```

эквивалентны. Переменные, участвующие в таких присваиваниях, назовем ключевыми параметрами.

Если установлен флаг -k (см. set), то все ключевые параметры помещаются в окружение команды, даже если они записаны после команды.

### 4.4. Сигналы

UNIX'ом поддерживаются следующие сигналы:

SIGHUP	- 1 -	отменить (hangup)
SIGINT	- 2 -	прерывание (interrupt)
SIGQUIT	- 3 -	нестандартный выход (quit)
SIGILL	- 4 -	неверная команда (illegal instruction)
SIGTRAP	- 5 -	ловушка (trace trap)
SIGFPE	- 8 -	исключительная ситуация при выполнении операций с плавающей запятой (floating-point exception)
SIGKILL	- 9 -	уничтожение процесса (kill)
SIGBUS	- 10 -	ошибка шины (bus error)

SIGSEGV - 11 - нарушение сегментации (segmentation violation)  
 SIGSYS - 12 - неверный системный вызов (bad argument to system call)  
 SIGPIPE - 13 - запись в канал без чтения из него (write on a pipe with no one to read it)  
 SIGALRM - 14 - будильник (alarm clock)  
 SIGTERM - 15 - программное завершение процесса (software termination signal)

Сигналы SIGINT и SIGQUIT игнорируются, если команда была запущена асинхронно. Иначе сигналы обрабатываются так же, как в процессе-предке, за исключением сигнала SIGSEGV (см. также Специальные команды. Trap).

#### 4.5. Замечания

При выполнении команд запоминается их местонахождение. Поэтому при создании команды с тем же именем, но находящейся в другой директории, все равно будет выполняться старая команда (если вызов происходит по короткому имени). Для исправления ситуации воспользуйтесь командой hash с ключом -r (см. Специальные команды).

Если вы переименовали текущую или вышележащую директорию, то команда pwd может давать неверную информацию. Для исправления ситуации воспользуйтесь командой cd с полным именем директории.

### 5. Дополнительные сведения

#### 5.1. Команда test

Команда test применяется для проверки условия. Формат вызова:

```
test <выражение>
или
[ <выражение> ]
```

Команда test вычисляет <выражение> и, если его значение - истина, возвращает код завершения 0 (true); иначе - ненулевое значение (false). Ненулевой код завершения возвращается и если опущены аргументы. <Выражение> может состоять из следующих примитивов:

-r файл	- истина, если файл существует и доступен для чтения
-w файл	- истина, если файл существует и доступен для записи
-x файл	- истина, если файл существует и является выполняемым
-f файл	- истина, если файл существует и является обычным файлом
-d файл	- истина, если файл существует и является директорией
-с файл	- истина, если файл существует и является специальным символьно-ориентированным файлом
-b файл	- истина, если файл существует и является специальным блок-ориентированным файлом
-p файл	- истина, если файл существует и является именованным каналом (pipe)
-s файл	- истина, если файл существует и имеет ненулевую длину
-t [ дескриптор файла ]	- истина, если открытый файл с указанным дескриптором (по умолчанию 1) существует и ассоциирован с терминалом
-z s1	- истина, если длина строки s1 нулевая
-n s1	- истина, если длина строки s1 ненулевая
s1 = s2	- истина, если строки s1 и s2 совпадают
s1 != s2	- истина, если строки s1 и s2 не совпадают
s1	- истина, если s1 непустая строка

n1 -eq n2            - истина, если целые n1 и n2 алгебраически совпадают . На месте -eq могут быть также -ne, -gt, -ge, -lt, -le

## 5.2. Команда expr

Команда expr применяется для вычисления выражений. Результат выводится на стандартный вывод. Операнды выражения должны быть разделены пробелами. Метасимволы должны быть экранированы. Надо заметить, что 0 возвращается в качестве числа, а не для индикации пустой строки. Строки, содержащие пробелы или другие специальные символы, должны быть заключены в кавычки. Целые рассматриваются как 32-битные числа.

Ниже приведен список операторов в порядке возрастания приоритета, операции с равным приоритетом заключены в фигурные скобки. Перед символами, которые должны быть экранированы, стоит \.

<выр> \|            если первое <выр> не пустое и не нулевое, то возвращает его, иначе возвращает второе <выр>  
<выр>  
<выр> \|&            если оба <выр> не пустые и не нулевые, то возвращает первое <выр>, иначе возвращает 0  
<выр>  
<выр> { =, >, \|>, \|<, \|<=, != } <выр>    возвращает результат целочисленного сравнения если оба <выр> - целые; иначе возвращает результат лексического сравнения  
<выр> { +, - } <выр>    сложение и вычитание целочисленных аргументов  
<выр>  
<выр> { \\*, /, \% } <выр>    умножение, деление и получение остатка от деления целочисленных аргументов  
<выр> : <выр>    оператор сопоставления : сопоставляет первый аргумент со вторым, который должен быть регулярным выражением. Обычно оператор сравнения возвращает число символов, удовлетворяющих образцу (0 при неудачном сравнении). Однако символы \( и \) могут применяться для выделения части первого аргумента.

Регулярное выражение строится следующим образом:

- .
  - \*
  - [ ]
- обозначает любой символ  
- обозначает предыдущий символ, повторенный несколько раз  
- обозначают любой один из указанных между ними символов; группа символов может обозначаться с помощью знака "-" (т.е. [0-9] эквивалентно [0123456789]); если после [ стоит ^, то это эквивалентно любому символу, кроме указанных в скобках и <возврата\_каретки>; для указания ] в качестве образца, надо поставить ее сразу за [ (т.е. [...] ); . и \* внутри квадратных скобок обозначают самих себя

Все остальные символы (и ^, если стоит не в квадратных скобках) обозначают самих себя. Для указания символов ., \*, [ и ] надо экранировать их (т.е. писать \., \\*, \[, \]).

### Примеры.

1.

```
a=`expr $a + 1`
```

- увеличение на 1 переменной a

2.

```
expr $a : '.*\/\(.*)' \| $a
```

- выделяет из имени файла короткое имя (т.е. из /usr/util/ena выделяется ena). Внимание, одиночный символ / будет воспринят как знак операции деления.

3.

```
expr $VAR : '.*'
```

- получение количества символов переменной VAR.

В качестве побочного эффекта expr возвращает следующие коды завершения:

- 0 - если выражение не нуль и не пустая строка
- 1 - если выражение нуль или пустая строка
- 2 - для некорректных выражений

Команда expr также выдает следующие сообщения об ошибках:

syntax error	- для ошибок в операторах или операндах
non-numeric argument	- для попыток применения арифметических операций к нечисловым строкам

#### **Замечание.**

Допустим, что мы хотим сравнить значение переменной а с каким-либо символом, имеющим для expr особый смысл, например, со знаком равенства. Пусть \$a на самом деле является знаком равенства. Так как аргументы предварительно обрабатываются shell'ом, то команда

```
expr $a = '='
```

будет воспринята как

```
expr = =
```

что синтаксически некорректно. В подобных случаях надо пользоваться таким приемом:

```
expr X$a = X=
```

т.е. добавлением некоторого символа к обеим строкам, что никак не влияет на результат сравнения, однако позволяет избежать синтаксической ошибки.

### **5.3. Команда ena**

Команда ena позволяет получить части полного имени файла. Первый аргумент - флаг, второй - имя файла. Команда различает следующие флаги:

- n - имя файла без расширения
- f - имя файла с расширением
- e - расширение
- d - имя директории
- p - если имя файла начинается с . или .. , то эти символы выделяются из имени

Ниже приводится текст программы ena, хранящийся в /util/ena.

```
# Get part of pathname
```

```

case $1 in
-n )
    expr $2 : '.*\/(.*\[.]*' \|| $2 : '\(.*\[.]*' \|| $2
    ;;
-f )
    expr $2 : '.*\/(.*\)' \|| $2
    ;;
-e )
    expr $2 : '.*\([.][^./]*\) ' \|| ' '
    ;;
-d )
    expr $2 : '\(.*\)\/.*' \|| $2
    ;;
-p )
    expr $2 : '\([.]\)\/.*' \|| $2 : '\([.][.]\)\/.*' \|| ' '
    ;;
* )
    echo "error: unknown part of pathname $1"
    exit 2
    ;;
esac

```