Учебная практика BASH для чайников

Методическое пособие создано для потока 1-го курса ИУ7 2023 года для помощи в освоении предмета ПТП.

Оглавление

Глава I. Да кто этот ваш `.bash`?

1.1. Для чего нужен bash?

Программисты — люди ленивые, привыкшие все автоматизировать. Поэтому неудивительно, что появилась такая вещь, как bash-скрипты (далее автор будет называть bash-скрипты просто **bash**). В bash можно записывать целые сценарии для выполнения рутинной работы. Приведу пример:

```
#!/bin/bash
# test.sh

for file in "./tests/in/*"; do
    cat $file | python3 my_lab.py > ${file/in/out}
done
```

Скрипт автоматически вводит данные в Вашу программу (например, лабораторную по программированию) и записывает результаты в файл. Таким образом Вы можете протестировать свою программу сразу на нескольких массивах входных данных *одной командой!*

```
bash test.sh
```

Неплохо, не правда ли? А ведь это только начало! Возможности BASH-скриптов ограничены лишь Вашим воображением.

И да, `bash` есть на `Linux`-системах. Предмет ПТП подразумевает, что у Вас уже есть данная ОС в любом виде.

Глава II. Переменные как смысл жизни

> В bash-е нет типов переменных!

Просто запомните это. Каждый раз, когда Вы используете переменную, помните эти слова.

Рассмотрим на примере:

```
# Python3
a = 5
b = "abc"
```

```
# Bash
a=5
b=abc
```

Может показаться, что никакой разницы нет. Однако, в случае с `Python3` переменные имеют свой тип:

```
>>> type(a)
<class 'int'>
>>> type(b)
<class 'str'>
```

В то время как в `bash` в обоихз случаях мы задаём **строку**. Т.е. это эквивалетно скрипту на языке `Python3`:

```
a = "5"
b = "abc"
```

И как с этим жить? Об это и пойдёт речь.

2.1. Создание переменных

2.1.1. Обычное создание переменной

Объявить переменную несложно:

```
var=value
```

> Рядом со знаком `=` не должно быть пробелов!

```
var=42     # CORRECT
var = 42   # WRONG
var= foo   # WRONG
```

Если же по каким-то причинам пробел всё-таки нужен, используйте кавычки:

```
greet="Hello world!"
```

> Кавычки при объявлении переменных необязательны: они требуются только для тех случаев, когда нужно явно задать границы значения, например, когда есть пробелы.

2.2. Чтение переменных или почему программистам так много платят

Просто потому, что онииспользуют символ `$` каждый раз, когда читают переменную! 😃

Подстановка символа `$` перед названием переменной *заменяет* вызов чтения этой переменной на её значение. Т.е.:

```
a="echo foo"
$a      # echo foo
```

В примере выше интерпретатор подставляет на место `$a` её значение `echo foo`. После он рассматривает **всю** строку как **команду**, и если получается, выполняет её. Этот алгоритм работает во всех случаях использования `$`.

Иногда нужно строго задать границы названия вызываемой переменной:

```
a=foo
abar="i wanted sth else..."
echo $abar    # i wanted sth else
echo ${a}bar # foobar
```

Чтобы использовать знак доллара как знак доллара (а не как вызов переменной), необходимо его *экранировать*:

```
a="300\$"
echo $a     # 300$
```

2.3. Вопросы существования с точки зрения `bash`

2.3.1. Несуществующие переменные

А что, если мы попытаемся обратиться к несуществующей переменной? Попробуем:

```
a=5
echo $a
echo $b
```

И мы получим...

```
5

```

... просто пустую строку на месте переменной `b`! Никаких вам `NameError: name 'b' is not exists`!

> В bash несуществующая переменная является пустой строкой.

Поэтому код ниже абсолютно рабочий:

```
echo $a             # Empty output
b=$a                # Empty var 'b'
c="my answer: $a"   # my answer:
```

2.3.2. Фокусы с исчезноваением

Удалить переменную можно с помощью команды `unset`:

```
a=foo
echo $a     # foo
unset a
echo $a     # nothing
```

2.4. Особые переменные

Такие переменные уникальны тем, что им нельзя вручную задать значение — они доступны только для чтения.

2.4.1. По порядку ра-а-а-с-с-читайсь!

Позиционные переменные — переменные вида: `$0 $1 $2 ...`

`$0` хранит в себе *абсолютный* путь до скрипта:

```
linux@vasya:~/bmstu/bash$ cat test.sh
echo $0

linux@vasya:~/bmstu/bash$ bash test.sh
/home/vasya/bmstu/bash/test.sh

linux@vasya:~/bmstu/bash$
```

Остальные переменные хранят *аргументы*:

```
$ cat test.sh
echo $1
echo $2
echo $3

$ bash test.sh foo bar "hello, bash!"
foo
bar
hello, bash!

linux@vasya:~/bmstu/bash$
```

Кол-во заданных позиционных параметров (кроме `$0`) можно узнать из переменной `$#`:

```
$ cat test.sh
echo $#

$ bash test.sh foo bar "hello world!"
3
```

Мы получим ответ "3", т.к.:

- Нулевой аргумент присутствует всегда — это путь до исполняемого файла, но он не считается;
- Далее 2 очевидных параметра;
- В конце идёт *один* параметр, представляющий из себя два слова с пробелом между.

Все аргументы (т.е. позиционные переменные, начиная с `$1`) вместе хранятся в ещё двух уникальных переменных: `$@` и `$*`:

```
$ cat test.sh
echo $@
echo $*

$ bash test.sh foo bar "hello world!"
foo bar hello world!
foo bar hello world!
```

```
```

Может показаться, что они абсолютно эквивалентны. Однако на самом деле есть один нюанс, который будет рассмотрен в главе VII.

2.4.2. Код возврата

`$?` хранит код возврата *последней выполненной команды*.

Небольшое отступление про коды возврата.

Каждая программа в linux-е возвращает некоторое число, показывающее состояние программы на момент завершения. Если программа завершилась успешно, это число `0`. Иначе она может вернуть любое другое число, каждое из которых имеет свой смысл, однако нам важно понять, что *всё что не нуль — это некорректное завершение*.

Чтобы пазл окончательно сложился, увидим на примере:

```
echo Hello!
echo $?
```

Получим:

```
Hello!
0
```

Или такой пример:

```
true
echo $?     # 0
false
echo $?     # 1
```

Да-да, в bash-е есть *команды* `true` и `false`! Важно не перепутать: `true` возвращает **нуль**, т.к. это код успешного выполнения программы, а `false` возвращает **единицу**, т.к. это ненулевой код возврата некорректного завершения программы. Однако, не каждое некорректное завершение программы вернёт `1` (может быть любое другое ненулевое значение).

Вернёмся к кодам возврата в будущем.

2.5. Манипуляции со строками

2.5.1. Длины строк

Любая переменная в `bash`-е является строкой, а строка имеет длину. Длину строки можно узнать:

```
a=foo
echo ${#a}  # 3
echo ${#b}  # 0
```

2.5.2. Жонглируем регистрами

У переменной можно изменить регистр (т.е. заглавные-строчные буквы):

```
a=foo
echo ${a,,}     # foo
echo ${a^^}     # FOO
echo ${a^}      # Foo
```

2.5.3. Конкатенация — не сложение!

Для сложения строк и чисел во многих ЯП используется символ `+`:

```
# Python3
a = "foo"
b = "bar"
c = a + b
print(c)  # foobar
```

Но не в bash. Правильная запись "сложения" (конкатекации) строк:

```
a=foo
b=bar
echo ${foo}${bar}    # foobar
```

Глава III. Взаимодействие с внешним окружением

3.1. Ввод переменных

Когда нужно получить ввод пользователя, используется команда `read`:

```
read a
```

Нет необходимости объявлять переменную заранее. Она будет создана автоматически.

```
# WRONG
a=""
read a

# CORRECT
read a
```

Что делать, если вводимых значений несколько? Если ввод построчный, то:

```
read a
read b
```

Если ввод идёт через пробел:

```
read a b
```

Если ввод разделён *не* пробелом:

```
IFS=", "
read a b
```

Пример выше позволяет вводить переменные через запятую и пробел: `5, abc`

`IFS` — переменная, которая хранит строку, разделяющая вводимые переменные. Хорошей практикой будет сохранять его предыдущее значение:

```
OLD_IFS=$IFS
IFS=", "

read a b
```

```
...

IFS=$OLD_IFS
```

3.2. Коды выхода

Через команду `exit` можно завершить выполнение скрипта. А передав в неё числовой параметр, мы возвращаем его как код возврата:

```
$ cat test.sh
echo hello!
exit 1

$ bash test.sh
hello!

$ echo $?
1
```

Небольшая справка про зарезервированные коды возврата — https://tldp.org/LDP/abs/html/exitcodes.html

Мы можем использовать:

- `0` — программа завершилась успешно.
- `1` — код для ошибки (буквально для любых ошибок).
- `2` — обозначает ошибку, возникшую по вине пользователя.
- `3-126` — свободные коды ошибок, используем как хотим.

3.3. «Трое из ларца» или std-потоки

Существует три основных потока:

- `stdin` (0) — стандартный поток ввода в программу.
- `stdout` (1) — стандартный поток вывода из программы.
- `stderr` (2) — стандартный поток ошибок из программы.

По умолчанию команда `echo` выводит текст в `stdout`. Чтобы задать другой поток, используется синтаксис:

```
echo ERROR >&2
```

В примере выше мы выводим текст ошибки в поток `stderr`.

Глава IV. Ветвления в bash-е или «А что если...»

4.1. Это база!

Условие в bash записывается просто:

```
if condition1
then
    ...
elif condition2
    ...
else
    ...
fi
```

Для понимания всей творящейся магии, ещё раз повторим:

> В bash-е эквивалент True — это 0, а False — != 0

И оператор if смотрит *на код возврата выражения, стоящего после него*:

```
if some_program
then
    echo Program finished successfully
else
    echo Program finished with non-zero code $?
fi
```

В примере выше мы запускаем программу, получаем код возврата и:

- если код возврата равен нулю (программа завершилась корректно), то срабатывает верхняя ветвь условия, и мы видим "Program finished successfully".
- иначе мы выводим "Program finished with non-zero code N", где N — код возврата программы, который мы получили через переменную $? (см. 2.4.2).

Но вся прелесть условий раскрывается при использовании скобок.

4.2. "[" и его друзья

4.2.1. Немного душной теории

Как Вы думаете, что такое [ ]? Возможно, мир для Вас уже не станет прежним, но... [— это команда. Да, команда с очень странным названием, но это самая что ни на есть команда. А] — это её последний аргумент. Её знают ещё под одним именем — test. Т.е. каждый раз, когда Вы пишете:

```
```

```
if [ $a ]; then
    echo $a
fi
```

Вы на самом деле пишите что-то похожее на это:

```
if test $a; then
    echo $a
fi
```

Часто можно увидеть такую запись условия:

```
if [[ ... ]]; then
    ...
fi
```

И в этом случае `[[` — команда, а `]]` — её последний аргумент.

В чём разница? Если кратко, `[` — это стандарт, принятый во всех `POSIX`-системах. Это значит, что такая запись сработает почти на любой ОС, поддерживающей `POSIX`. А `[[` — это уже локальные особенности bash-а (и не только его), и не факт, что, оказавшись на незнакомой машине, такое условие сработает.

Мы изучаем именно `bash`, поэтому смело:

> Правило «Чем больше — тем лучше» — между `[[` и `[` в учебной практике смело выбирайте `[[`!

Просто потому, что некоторые особенности `[[` не работают на `[`, зато все возможности `[` поддерживаются `[[` (обратная совместимость). И лучше лишний раз записать больше скобок, чем скрипт неожиданно сломается.

Автор далее будет использовать обе формы записи, используя `[` везде, где возможно.

Также, к "друзьям" `[` можно отнести `((`, но к этой парочке вернёмся в пятой главе.

4.2.2. Социальное дистанционирование

Именно потому, что `[` и `[[` — это на самом деле команды, они так требовательны к пробелам:

```
if [$a]                 # WRONG
if [[$a]]               # WRONG
if [! $a ]              # WRONG
if ![[ $a ]]            # WRONG
if !([ $a ] && [ $b ])  # WRONG

if [ $a ]               # CORRECT
```

```
if [[ $a ]]             # CORRECT
if [ ! $a ]             # CORRECT
if ! [[ $a ]]           # CORRECT
if ! ([[ $a ]])         # CORRECT
```

4.3. Возможности "["

4.3.1. Экзистенциальные вопросы

Для проверки, существует ли переменная, просто записывайте переменную в `[ ]`:

```
if [ $a ]; then
    ...
fi
```

Есть нюансы с использованием `-n` и `-z` параметров... Их поведение на самом деле не так очевидно, поэтому автор рекомендует избегать их использования.

4.3.2. Эквивалентность

Редкий случай, когда для сравнения используется именно `=`, а не `==`:

```
a=foo
b=foo
if [ $a = $b ]; then
    echo "\$a = \$b"
else
    echo "\$a != \$b"
fi

# Output: $a = $b
```

4.3.3. Л-Логика

В порядке приоритета: `NOT` = `! AND` = `-a OR` = `-o`

Чтобы изменить приоритет, используются `( )`-скобки, которые нужно обязтаельно экранировать (так надо) — `\( \)`. Эти скобки на самом деле обозначают создание нового subshell-а, который возвращает результат выражения внутри... Но нам главное, что эти скобки похожи на человеческие скобки, и делают они то же самое, что мы от них и ждём:

```
if [ ! \($a -a $b\) ]; then
    echo ERROR >&2
    exit 1
```

```
fi
...
```

Скрипт выше выводит `ERROR`, если одна из переменных не задана.

4.3.4. Алгебраические сравнения

К нюансам работы с числами вернёмся в главе V.

Если переменные представлены строками, содержащими **целые** числа, нам доступны такие операции:

- `-eq` (**eq**ual) = `==`
- `-ne` (**n**ot **e**qual) = `!=`
- `-gt` (**g**reater **t**hen) = `>`
- `-lt` (**l**ess **t**hen) = `<`
- `-ge` (**g**reater or **e**qual) = `>=`
- `-le` (**l**ess or **e**qual) = `<=`

Например:

```
if [ $# -lt 2 ]; then
    echo ERROR >&2
    exit 1
fi
...
```

Скрипт вернёт ошибку, если кол-во параметров будет меньше двух.

Если хотя бы одна переменная будет *не* числом, то команда `[` вернёт код ошибки 2 и текст ошибки в `stderr`. Т.к. любой код, отличный от нуля, преобразовывается в логическое `false`, то выполнится ветвь код для отрицания. Поэтому работает такой фокус:

```
a=abc
if [ $a -eq $a ]; then
    echo var 'a' is a number
else
    echo var 'a' is not a number
fi

b=123
if [ $b -eq $b ]; then
    echo var 'b' is a number
else
    echo var 'b' is not  a number
fi
```

Если смотреть поток `stdout`, то мы увидим:

```
var 'a' is not a number
var 'b' is a number
```

Но на самом деле пользователь увидит что-то такое:

```
bash: [: abc: integer expression expected
var 'a' is not a number
var 'b' is a number
```

Чтобы проверить, является ли условие ложным или в нём произошла такая ошибка, можно проверить код возврата:

```
a=abc
# a=123
c=321
if [ $c -gt $a ]; then
    echo 'c' greaten than 'a'
elif [ $? -eq 1 ]; then
    echo 'c' less than 'a'
else
    echo ERROR >&2
    exit 1
fi
```

> Лучше не допускать попадание некорректных данных в условие, а валидировать числа заранее. Подробнее об этом будет описано в V главе.

4.4. Возможности "[["

`[[` имеет все те же возможности, что и `[`, но с некоторыми дополнениями.

4.4.1. Л-Логика ч.2

В `[[` можно смело использовать более популярный синтаксис логических операций (как в C).

`==`, `!=` — работает как для чисел, так и для строк

`&&`, `||` — работает для чисел

Перепишем пример с `[`:

```
if ! [[ $a && $b  ]]; then
    echo ERROR >&2
```

```
    exit 1
fi
```

В отличие от `[`, здесь экранировать `(  )` не требуется.

4.4.2. Используем регулярные выражения

К регулярным выражениям вернёмся в главе IV.

Чтобы проверить текст на соответствие регулярному выражению, используется оператор `=~`.

```
a=123
if [[ $a =~ \d+ ]]; then
    echo var 'a' is a number
else
    echo var 'a' isn\'n a number
fi
```

> В записи регулярного выражения внутри регулярного выражения не должно быть кавычек!

Потому что такие кавычки будут восприняты как *символ кавычек*, а не как граница выражения.

4.5. Фокусы!

4.5.1. Красиво, но...

`bash` очень гибок. Код ниже абсолютно валиден:

```
[[ $condition ]] && echo yes || echo no
```

Если условие верно, то выведится `yes`, иначе `no`.

Как это работает? Рассмотрим для начала последовательность выполнения команд при истинном условии.

1. Условие заменяется на 0, что эквивалентно `true`

```
[[ $condition ]] --> test $condition --> 0 --> true
```

2. `bash` смотрит на это выражение как на логическое и пытается вычислить его значение (`true` или `false`):

```
true && echo yes || echo no
```

По законам алгебры логики:

```
1 && a || b
```

преобразуется в

```
a || b
```

Причём если верно `a`, то независимо от значения `b` итог будет `true`. Интерпретатор — как студент — очень ленивый. Если `a` верно, то зачем считать `b`? Поэтому сначала он выполнит только команду слева от `||`, т.е. `echo yes`. Она выполнится успешно (а почему нет?) и всё, что после `||` выполнено не будет. Получаем вывод `yes`.

Теперь случай для ложного условия.

1. Условие заменяется на 1 (или другой код), который эквивалентен `false`.

```
[[ $condition ]] --> test $condition --> 1 --> false
```

2. `bash` смотрит на это выражение как на логическое и пытается вычислить его значение:

```
false && echo yes || echo no
```

По законам алгебры логики:

```
0 && a || b
```

преобразуется в

```
b
```

Т.е независимо от значения `a` итоге будет равен значению `b`. Интерпретатору не имеет смысла выполнять команда до `||` и сразу выполняет команду после. Получаем вывод `no`.

4.5.2. ... лучше не использовать

Красиво, но такую запись ветвления не рекомандуется использовать в коде. Почему?

1. Ухудшение читабельности. В короткой записи довольно-таки сложно разобраться.
2. Тяжелее работать с кодом. Если Вы записали условие, например:

```
[ $a ] && echo $a || echo ERROR
```

Вы захотели добавить `exit 1`, то Вам придётся перезаписывать условие в полную форму:

```
if [ $a ]; then
    echo $a
else
    echo ERROR
    exit 1
fi
```

3. Непредсказуемое поведение. В некоторых случаях может выскочить не то, что Вы хотели:

```
[ $condition ] && command1 || command2
```

Если команда `command1` завершится с ненулевым кодом возврата, то будет выполнена вторая команда (почему? попробуйте разобрать последовательность вызовов), несмотря на то, что условие истинно.

Тогда для чего это было показано? Есть две причины:

1. В интернете нередко можно заметить такие фокусы, которые в первый раз могут вызвать затруднения с пониманием, как это работает и что оно делает.
2. В командной строке, когда нужно быстро проверить какое-то условие в командной строке.

Глава V. Маски и регулярные выражения — с чем их есть и зачем?

Сделаем небольшое отступление от синтаксиса `bash`-а, т.к. это нам пригодится в будущем. В этой главе будут разобраны *маски файлов* и *регулярные выражения* — очень похожие, но такие разные инструменты работы со строками.

5.1. Маски(ровка)

Маски файлов преимущественно нужны для работы с каталогами и с файлами. Они очень просты: `*` - обозначает любую последовательность символов; `?` - обозначает ровно один любой символ; Все остальные символы обозначают сами себя.

Пример для команды `ls`:

```
$ ls
4.txt 40.txt 42.txt 45.py

$ ls *.txt
4.txt 40.txt 42.txt

$ ls 4*
4.txt 40.txt 42.txt 45.txt

$ ls 4?.txt
40.txt 42.txt
```

5.2. Регулярные выражения

> https://regex101.com — регулярные выражения лучше всего писать и тестировать именно здесь. И все примеры лучше проверять здесь

Регулярное выражение, простыми словами — некоторый шаблон, по которому можно определить соответствие строк. Это не строгое сравнение, как в случае с `==`, а "умное". По регулярным выражениям есть целый предмет — Теория Формальных Языков (ТФЯ), кошмар для ИУ9.

Регулярные выражения — очень полезный инструмент, с базовыми основами которого попробуем разобраться в этой главе.

Единственное, что не могут регулярные выражения — так это распарсить HTML

5.2.1. Это база!

Регулярные выражения могут не использовать специальных символов:

```
text
```

Для строки:

```
some text
```

данному регулярному выражению будет соответствовать подстрока:

```
text
```

(проверьте на сайте выше, как это выглядит)

Пример:

```
if [[ "some text" =~ text ]]; then
    echo yes
else
    echo no
fi
```

`"some text"` — это строка, которую проверяем на соответствие регулярному выражению (на этом месте может быть переменная), и `text` — как ни странно, **само регулярное выражение**, просто записанное без спец. символов.

Т.к. в строке `some text` есть подстрока `text`, результатом работы скрипта будет ответ `yes`.

5.2.2. Снова `[ ]`

`[ ]` — обозначают любой символ внутри. Например:

```
[ab]c
```

Будут соответствовать строки:

```
ac
bc
```

Чтобы не писать весь алфавит, можно использовать диапазоны:

- `[A-Z]` — заглавные латинские буквы
- `[a-z]` — строчные латинские буквы
- `[A-z]` — латинские буквы (именно в таком порядке!)
- `[0-9]` — цифры

- `[0-f]` — так тоже можно, это будут шестнадцатеричные цифры в нижнем регистре

Например:

```
[0-2][0-9]
```

Будут соответствовать строки:

```
00
18
29
```

и т.п.

Чтобы не перебирать все возможные символы, "любой символ" можно задать `.`:

```
..
```

Будут соответствовать строки:

```
ab
5$
```

5.2.3. Кванторы

Если простым языком, кванторы определяют количество предыдущей строки/символа в подстроке.

- `?` — от 0 до 1

```
[0]?[1-9]
```

Будут соответствовать строки:

```
2
01
```

- `*` — больше или равно 0

```
[0-9]*
```

Будут соответствовать строки:

```
0
35434646
```

- `+` — больше 0

```
[0-9]+
```

Будут соответствовать строки:

```
1
1235
```

5.2.4. Экранирование

Чтобы использовать специальный символ как сам символ, используется **экранирование** при помощи `\`. Например, если нам нужно регулярное выражение на проверку расширения файла:

```
.*\.txt
```

Первая точка будет обозначать **любой символ**, а вторая, экранированная, обозначает саму себя, т.е. точку.

5.2.5. Повторения

Если нужно конкретное количество повторений подстроки, используются такие скобки — `{}`:

```
[1-9][0-9]{3}
```

Будут соответствовать строки:

```
1830
2023
```

Или можно записать так:

```
[1-9][0-9]{1-3}
```

Будут соответствовать строки:

```
10
128
1830
```

5.2.6. Группы

Если кванторы нужно применить не к одному символу, можно оборачивать нужную подстроку в *группу* `(  )`:

```
(ab)*
```

Будут соответствовать строки:

```
ab
ababab
```

5.2.7. Якоря

Якоря обозначают начало и конец строки:

- `^` — начало строки
- `$` — конец строки

При помощи якорей можно "отбросить" соответствие на подстроки. Например:

```
[[ "file.txt.a" =~ .*\.txt ]]    # True
[[ "file.txt.a" =~ .*\.txt$ ]]   # False
[[ "file.txt" =~ .*\.txt$ ]]     # True
```

5.2.8. Числа!

Итак, наконец мы получили достаточно знаний, чтобы написать регулярное выражение для целого числа:

```
^[-+]?[0-9]+$
```

```
```

Разберем каждый символ:

- `^` — начало строки, т.е. мы отбрасываем "ab566" значения
- `$` — конец строки, т.е. мы отбрасываем "1223ab" значения
- `[-+]` — любой символ, + или -
- `[-+]?` — обозначет, что "+" или "-" необязательны, но они могут быть в строке, обозначающей число
- `[0-9]` — любая цифра
- `[0-9]+` — цифра должна быть хотя бы одна

Можно попробовать сделать регулярное выражение для дробного числа:

```
^[+-]?[0-9]+(\.[0-9]+)?$
```

Подробнее:

- `[+-]?[0-9]+` — проверка на подстроку-число из предыдущего примера. Обратите внимание, что якоря "переехали", т.к. начало и конец строки теперь в других местах.
- `\.` — точка как символ точки
- `\.[0-9]+` — дробная часть числа
- `(\.[0-9]+)?` — дробная часть числа необязательная

Автор настоятельно рекомендует самостоятельно попробовать написать регулярные выражения для даты, чтобы закрепить материал.

Глава VI. Магия математики

Наконец можно приступить к математическим операциям!

Числа желательно (обязательно) проверять регулярным выражением, т.к. всё в bash-е является строкой и нет отдельного типа данных для чисел.

```
read a b
re_num=^[-+]?[0-9]+$
if ! ([[ $a =~ re_num ]] && [[ $b =~ re_num ]]); then
    echo ERROR >&2
    exit 1
else
    ...
fi
```

6.1. Школьная арифметика

> Bash не поддерживает арифметические операции с дробными числами.

Совсем. И незачем. Bash-у не нужно что-то считать, с этой задачей справится любой классический полноценный язык программирования. Поэтому берём только целые числа.

Для произведения **любых математических** операций, используются двойные круглые скобки: `(( ))`:

```
a=2
(( b = a * a ))
echo $b   # 4

(( b++ ))
echo $b   # 5

(( b *= -1 ))
echo $b   # -5
```

Как Вы можете заметить, **внутри** скобок $ не обязателен. Однако его нужно использовать для особых переменных, например, для позиционных:

```
# test.sh 5 2
(( res = $1 + $2 ))
echo $res   # 7
```

Какие операции поддерживаются?

- `+`, `-`, `*`, `/` (целочисленное), `%`, `**`
- `+=`, `-=`, `*=` и т.п.
- `++` и `--`

6.2. Возврат результата вычисления

Запустив скрипт

```
b=2
echo (( b * 2 ))   # bash: syntax error
```

Вы получите ошибку, т.к. `(( ))` ничего не возвращают.

Чтобы вычислить результат и сразу вернуть его, используется, как можно догадаться, знак доллара перед (так же, как и чтение переменной):

```
b=2
echo $(( b * 2 ))   # 4
```

Злоупотреблять этим символом тоже не стоит:

```
i=0
$(( i += 1 ))      # bash: 1: command not found...
```

Как в случае с чтением переменных, интерпретатор подставляет на место доллара результат вычисления:

```
  $(( i += 1 ))
#      |
#    i -> 0
#      |
  $(( 0 + 1 ))
#      |
#  0 + 1 = 1
#      |
   $(( 1 ))
#      |
# $(( 1 )) -> 1
#      |
#      V

1   # bash: 1: command not found...
```

6.4. Математические условия

При помощи `(( ))` можно записывать условия, использующие математические операции:

```
if (( a % 2 == 0)); then
    echo $a — odd
else
    echo $a — even
fi
```

Глава VII. Вот они, слева направо: `while`, `until`, `for`

7.1. `while` и `until`

`while` — самый обычный, но не всегда самый удобный способ задать цикл:

```
while [ $condition ]; do
    ...
done
```

В качестве `condition` может выступать любое выражение — как в ветвлениях:

```
i=0
while [[ $i < 5 ]]; do
    echo -n "$i "
    (( i++ ))
done
# 0 1 2 3 4
```

Редко, но используется другая форма записи — через `until`:

```
i=0
until [[ $i >= 5 ]]; do
    echo -n "$i "
    (( i++ ))
done
# 0 1 2 3 4
```

В чём отличие? Цикл `while` работает, пока значение *истинно*, в то время как `until` работает, пока значение *ложно*.

Конечно, можно не морочить голову и везде писать `while`... Однако если есть такая возможность в языке, то почему бы не использовать? План такой:

- Записывате условие для `while`
- Если в условии есть отрицание всего выражение, например:

```
! ([[ $a ]] && [[ $b ]])
```

в таком случае проверяете, возможно ли избавиться от символа, уменьшив количество операторов. Если же нет, то записываете цикл через `until`. Напиример:

```
```

```
while !([[ $a ]] && [[ $b ]]); do
    ...
done
```

можно трансформировать в

```
until [[ $a ]] && [[ $b ]]; do
    ...
done
```

И такая запись будет короче и приятнее глазу.

7.2. Всемогущий `for`

7.2.1. Для массивов

Самый типичный пример использования `for` — перебор значений в массиве:

```
for it in array; do
    echo $it
done
```

Например:

```
OLD_IFS=$IFS
IFS=" "

line="i love bash scripts"

for word in $line; do
    echo word
done

IFS=$OLD_IFS

# i
# love
# bash
# scripts
```

К массивам вернёмся в 7 главе.

Но у `for` есть ещё несколько удобных конструкций, которые и делают его основным способом задания цикла.

7.2.2. Для простых последовательностей

Очень просто записывается `for` для последовательностей:

```
for i in $(seq 1 3); do
    echo -n "$i "
done
# 1 2 3
```

> Обратите внимание: границы включены в последовательность!

7.2.3. Си-стайл

Если же Вам нужно записать сложный цикл, как в С-подобных языках программирования, то Вам на помощь придут математические скобки:

```
for (( i=4; i>=0; i-- )); do
    echo -n "$i "
done
# 4 3 2 1 0
```

Но не стоит злоупотреблять такой возможностью. Если будет что-то такое:

```
for (( i=0; i<5; i++ )); do
    ...
done
```

можно заменить на более простую запись через `seq`:

```
for i in $(seq 0 4); do
    ...
done
```

Ещё раз обратим внимание: **в `seq` обе границы последовательности включительно!**

7.3. Обработка позиционных параметров

Ранее мы разобрали, что такое позиционные параметры и команду `shift`. Пора применить их вместе:

```
# test.sh foo bar "hello world"
```

```
while [ $1 ]; do
    echo $1
    shift
done

# foo
# bar
# hello world
```

Скрипт выводит построчно аргументы при запуске скрипта (за исключением нулевого). Вот так это работает:

```


# /pathtoyourscript/test.sh foo bar hello world
#                ^            ^   ^       ^
#                $0           $1  $2      $3

echo $1     # foo

shift       # ----->
# /pathtoyourscript/test.sh foo bar hello world
#                ^                ^       ^
#                $0               $1      $2

echo $1     # bar

shift       # ----->
# /pathtoyourscript/test.sh foo bar hello world
#                ^                        ^
#                $0                       $1

echo $1     # hello world

shift       # ----->
# /pathtoyourscript/test.sh foo bar hello world
#                ^
#                $0
```

Та самая последняя страница

Автор — *Жихарев Кирилл, студент МГТУ ИУ7*

- t.me/zhikhkirill
- vk.com//zhikh.localhost
- github.com/zhikh23

Главный редактор — *Дарья Соколова, студентка факультета журналистики МГУ*

- vk.com/curious_meerkat