Arithmetic expressions

Arithmetic expressions are used in several situations:

- arithmetic evaluation command
- arithmetic expansion
- substring parameter expansion
- the ''let'' builtin command
- C-style for loop
- array indexing
- conditional expressions
- Assignment statements, and arguments to declaration commands of variables with the integer attribute.

These expressions are evaluated following some rules described below. The operators and rules of arithmetic expressions are mainly derived from the C programming language.

This article describes the theory of the used syntax and the behaviour. To get practical examples without big explanations, see this page on Greg's wiki.

Constants

Mathematical constants are simply fixed values you write: `1`, `3567`, or `4326`. Bash interprets some notations specially:

- `0…` (leading zero) is interpreted as an **octal** value
- `0x…` is interpreted as a **hex** value
- `0X…` also interpreted as a **hex**
- <BASE>#… is interpreted as a number according to the **specified base** <BASE>, e.g., `2#00111011` (see below)

If you have a constant set in a variable, like,

```
x=03254
```

this is interpreted as an octal value. If you want it to be interpreted as a decimal value, you need to expand the parameter and specify base 10:

```
# this is interpreted as a decimal:
echo $(( 10#$x ))

# this is interpreted as an octal:
echo $(( x ))

# this is an invalid digit for base 10 (the "x")...:
echo $(( 10#x ))
```

Different bases

For a constant, the base can be specified using the form

```
<BASE>#<DIGITS...>
```

Regardless of the specified base, the arithmetic expressions will, if ever displayed, be **displayed in decimal**!

When no base is specified, the base 10 (decimal) is assumed, except when the prefixes as mentioned above (octals, hexadecimals) are present. The specified base can range from 2 to 64. To represent digits in a specified base greater than 10, characters other than 0 to 9 are needed (in this order, low ⇒ high):

- `0 … 9`
- `a … z`
- `A … Z`
- `@`
- `_`

Let's quickly invent a new number system with base 43 to show what I mean:

```
$ echo $((43#1))
1

$ echo $((43#a))
10

$echo $((43#A))
36

$ echo $((43#G))
42

$ echo $((43#H))
bash: 43#H: value too great for base (error token is "43#H")
```

If you have no clue what a base is and why there might be other bases, and what numbers are and how they are built, then you don't need different bases.

If you want to convert between the usual bases (octal, decimal, hex), use the printf command and its format strings.

Shell variables

Shell variables can of course be used as operands, even when the integer attribute is not turned on (by declare -i <NAME>). If the variable is empty (null) or unset, its reference evaluates to 0. If the variable doesn't hold a value that looks like a valid expression (numbers or operations), the expression is re-used to reference, for example, the named parameters, e.g.:

```
test=string
string=3

echo $((test))
# will output "3"!
```

Of course, in the end, when it finally evaluates to something that is **not** a valid arithmetic expression (newlines, ordinary text, …) then you'll get an error.

When variables are referenced, the notation `1 + $X` is equivalent to the notation `1 + X`, both are allowed.

When variables are referenced like `$X`, the rules of parameter expansion apply and are performed **before** the expression is evaluated. Thus, a construct like `${MYSTRING:4:3}` is valid inside an arithmetic expression.

Truth

Unlike command exit and return codes, arithmetic expressions evaluate to logical "true" when they are not 0. When they are 0, they evaluate to "false". The arithmetic evaluation compound command reverses the "truth" of an arithmetic expression to match the "truth" of command exit codes:

- if the arithmetic expression brings up a value not 0 (arithmetic true), it returns 0 (shell true)
- if the arithmetic expression evaluates to 0 (arithmetic false), it returns 1 (shell false)

That means, the following `if`-clause will execute the `else`-thread:

```
if ((0)); then
  echo "true"
else
  echo "false"
fi
```

Operators

Assignment

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td><ID> = <EXPR></td><td>normal assignment</td></tr>
<tr><td><ID> *= <EXPR></td><td>equivalent to <ID> = <ID> * <EXPR>, see calculation operators</td></tr>
<tr><td><ID> /= <EXPR></td><td>equivalent to <ID> = <ID> / <EXPR>, see calculation operators</td></tr>
<tr><td><ID> %= <EXPR></td><td>equivalent to <ID> = <ID> % <EXPR>, see calculation operators</td></tr>
<tr><td><ID> += <EXPR></td><td>equivalent to <ID> = <ID> + <EXPR>, see calculation operators</td></tr>
<tr><td><ID> -= <EXPR></td><td>equivalent to <ID> = <ID> - <EXPR>, see calculation operators</td></tr>
<tr><td><ID> <<= <NUMBER></td><td>equivalent to <ID> = <ID> << <NUMBER>, see bit operations</td></tr>
<tr><td><ID> >>= <NUMBER></td><td>equivalent to <ID> = <ID> >> <NUMBER>, see bit operations</td></tr>
<tr><td><ID> &= <EXPR></td><td>equivalent to <ID> = <ID> & <EXPR>, see bit operations</td></tr>
<tr><td><ID> ^= <EXPR></td><td>equivalent to <ID> = <ID> ^ <EXPR>, see bit operations</td></tr>
<tr><td><ID> |= <EXPR></td><td>equivalent to <ID> = <ID> | <EXPR>, see bit operations</td></tr>
</table>

Calculations

Operator	Description
`*`	multiplication
`/`	division
`%`	remainder (modulo)
`+`	addition
`-`	subtraction
`**`	exponentiation

Comparisons

Operator	Description
`<`	comparison: less than
`>`	comparison: greater than
`<=`	comparison: less than or equal
`>=`	comparison: greater than or equal
`==`	equality
`!=`	inequality

Bit operations

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>~</td><td>bitwise negation</td></tr>
<tr><td><<</td><td>bitwise shifting (left)</td></tr>
<tr><td>>></td><td>bitwise shifting (right)</td></tr>
<tr><td>&</td><td>bitwise AND</td></tr>
<tr><td>^</td><td>bitwise exclusive OR (XOR)</td></tr>
<tr><td>|</td><td>bitwise OR</td></tr>
</table>

Logical

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>!</td><td>logical negation</td></tr>
<tr><td>&&</td><td>logical AND</td></tr>
<tr><td>||</td><td>logical OR</td></tr>
</table>

Misc

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>id++</td><td>post-increment of the variable id (not required by POSIX®)</td></tr>
<tr><td>id--</td><td>post-decrement of the variable id (not required by POSIX®)</td></tr>
<tr><td>++id</td><td>pre-increment of the variable id (not required by POSIX®)</td></tr>
<tr><td>--id</td><td>pre-decrement of the variable id (not required by POSIX®)</td></tr>
<tr><td>+</td><td>unary plus</td></tr>
<tr><td>-</td><td>unary minus</td></tr>
<tr><td><EXPR> ? <EXPR> : <EXPR></td><td>conditional (ternary) operator
<condition> ? <result-if-true> : <result-if-false></td></tr>
<tr><td><EXPR> , <EXPR></td><td>expression list</td></tr>
<tr><td>(<EXPR>)</td><td>subexpression (to force precedence)</td></tr>
</table>

Precedence

The operator precedence is as follows (highest → lowest):

- Postfix (`id++`, `id--`)
- Prefix (`++id`, `--id`)
- Unary minus and plus (`-`, `+`)
- Logical and bitwise negation (`!`, `~`)
- Exponentiation (`**`)
- Multiplication, division, remainder (`*`, `/`, `%`)
- Addition, subtraction (`+`, `-`)
- Bitwise shifts (`<<`, `>>`)
- Comparison (`<`, `>`, `<=`, `>=`)
- (In-)equality (`==`, `!=`)
- Bitwise AND (`&`)
- Bitwise XOR (`^`)
- Bitwise OR (`|`)
- Logical AND (`&&`)
- Logical OR (`||`)
- Ternary operator (<EXPR> ? <EXPR> : <EXPR>)
- Assignments (`=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`)
- Expression list operator (<EXPR> , <EXPR>)

The precedence can be adjusted using subexpressions of the form (<EXPR>) at any time. These subexpressions are always evaluated first.

Arithmetic expressions and return codes

Bash's overall language construct is based on exit codes or return codes of commands or functions to be executed. `if` statements, `while` loops, etc., they all take the return codes of commands as conditions.

Now the problem is: The return codes (0 means "TRUE" or "SUCCESS", not 0 means "FALSE" or "FAILURE") don't correspond to the meaning of the result of an arithmetic expression (0 means "FALSE", not 0 means "TRUE").

That's why all commands and keywords that do arithmetic operations attempt to **translate** the arithmetical meaning into an equivalent return code. This simply means:

- if the arithmetic operation evaluates to 0 ("FALSE"), the return code is not 0 ("FAILURE")
- if the arithmetic operation evaluates to 1 ("TRUE"), the return code is 0 ("SUCCESS")

This way, you can easily use arithmetic expressions (along with the commands or keywords that operate them) as conditions for `if`, `while` and all the others, including `set -e` for autoexit on error:

```
MY_TEST_FLAG=0

if ((MY_TEST_FLAG)); then
  echo "MY_TEST_FLAG is ON"
else
  echo "MY_TEST_FLAG is OFF"
fi
```

Beware that `set -e` can change the runtime behavior of scripts. For example,

This non-equivalence of code behavior deserves some attention. Consider what happens if v happens to be zero in the expression below:

```
((v += 0))
echo $?
```

1

("FAILURE")

```
v=$((v + 0))
echo $?
```

0

("SUCCESS")

The return code behavior is not equivalent to the arithmetic behavior, as has been noted.

A workaround is to use a list operation that returns True, or use the second assignment style.

```
((v += 0)) || :
echo $?
```

0

("SUCCESS")

This change in code behavior was discovered once the script was run under set -e.

Arithmetic expressions in Bash

- The C-style for-loop
- Arithmetic expansion
- Arithmetic evaluation compound command
- The "let" builtin command

Discussion

sbin_bash, 2011/11/27 10:34

Now the problem is: The return codes (0 means "TRUE" or "SUCCESS", not 0 means "FALSE" or "FAILURE") don't correspond to the meaning of the result of an arithmetic expression (0 means "TRUE", not 0 means "FALSE").

==⇒arithmetic expression (0 means "FALSE", not 0 means "TRUE")

Techlive Zheng, 2012/11/02 18:01

@sbin_bash, fixed.

Joan, 2013/04/19 15:47

The link a the begin should direct to: http://mywiki.wooledge.org/BashGuide/CompoundCommands#Arithmetic_Evaluation

Jan Schampera, 2013/04/19 18:50

Done, thanks!

You could leave a comment if you were logged in.