iRules 101 - #02 - If and Expressions

Joe Pruitt, 2007-08-11

The if command is used to execute scripts dependent on a certain condition. This article discusses the rules for the if command as well as details on the format and use of TCL expressions. Other articles in the series:

- iRules 101 – #01 – Introduction to iRules
- iRules 101 – #02 – If and Expressions
- iRules 101 – #03 – Variables
- iRules 101 – #04 – Switch
- iRules 101 – #05 – Selecting Pools, Pool Members, and Nodes
- iRules 101 – #06 – When
- iRules 101 – #07 – Catch
- iRules 101 – #08 – Classes
- iRules 101 – #09 – Debugging
- iRules 101 – #10 – Regular Expressions
- iRules 101 – #11 – Events
- iRules 101 – #12 – The Session Command
- iRules 101 – #13a – Nested Conditionals
- iRules 101 – #13b – TCL String Commands Part 1
- iRules 101 – #14 – TCL String Commands Part 2
- iRules 101 – #15 – TCL List Handling Commands
- iRules 101 – #16 – Parsing String with the TCL Scan Command
- iRules 101 – #17 – Mapping Protocol Fields with the TCL Binary Scan Command

Command Usage:

if expr1 ?**then**? body1 **elseif** expr2 ?**then**? body2 **elseif** ... ?**else**? ?bodyN?

- The if command evaluates expr1 as an expression (in the same way that **expr** evaluates its argument).
- The value of the expression must be a boolean (a numeric value, where 0 is false and anything is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then body1 is executed by passing it to the Tcl interpreter. Otherwise expr2 is evaluated as an expression and if it is true then **body2** is executed, and so on.
- If none of the expressions evaluates to true then bodyN is executed.
- The **then** and **else** arguments are optional "noise words" to make the command easier to read.
- There may be any number of **elseif** clauses, including zero.
- BodyN may also be omitted as long as **else** is omitted too.
- The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no bodyN.

Examples

Here's a simple conditional

```
if { [HTTP::header Content-Length] > 0 } then {
  log local0. "Something's coming..."
}
```

If you want to add in an additional conditional into, you would do so like this:

```
if { [HTTP::header Content-Length] > 0 } then {
  log local0. "Something's coming..."
} elseif { [HTTP::uri] contains "foobar" } then {
  log local0. "The user wants some foobar!"
}
```

And to add a default case where control will pass when none of the if/elseif conditionals are passed you would add on an else clause:

```
if { [HTTP::header Content-Length] > 0 } then {
  log local0. "Something's coming..."
} elseif { [HTTP::uri] contains "foobar" } then {
  log local0. "The user wants some foobar!"
} else {
  log local0. "I don't know what this user wants!"
}
```

Also, remember that "**then**" and "**else**" are optional. In the examples above, "**then**" and "**else**" can be omitted for equivalent functionality:

```
if { [HTTP::header Content-Length] > 0 } {
  log local0. "Something's coming..."
} elseif { [HTTP::uri] contains "foobar" } {
  log local0. "The user wants some foobar!"
} {
  log local0. "I don't know what this user wants!"
}
```

For those VB Programmers out there, you may want to leave in the "**then**" argument B-).

Also, remember that expressions can be on multiple lines. That may be a good case for using the "**then**" argument.

```
if {
  ([HTTP::uri] contains "foo") ||
  ([HTTP::uri] contains "bar")
} then {
  log local0. "found foo!"
}
```

Expressions

At the heart of the "if" command, are the expressions that you use for the conditional test. These expressions are evaluated in the same way as the TCL "expr" command evaluates it's arguments. Here are the rules for expressions:

- A Tcl expression consists of a combination of operands, operators, and parentheses.
- White space may be used between the operands and operators and parentheses; it is ignored by the expression's instructions.
- Where possible, operands are interpreted as integer values.
- Integer values may be specified in decimal (the normal case), in octal (if the first character of the operand is **0**), or in hexadecimal (if the first two characters of the operand are **0x**).
- If an operand does not have one of the integer formats given above, then it is treated as a floating-point number if

- If an operand does not have one of the integer formats given above, then it is treated as a floating-point number if that is possible.
- Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler (except that the **f**, **F**, **l**, and **L** suffixes will not be permitted in most installations). For example, all of the following are valid floating-point numbers: 2.1, 3., 6e4, 7.91e+16.
- If no numeric interpretation is possible (note that all literal operands that are not numeric or boolean must be quoted with either braces or with double quotes), then an operand is left as a string (and only a limited set of operators may be applied to it).

Operands

Operands may be specified in any of the following ways:

1. As a numeric value, either integer or floating-point.
2. As a boolean value, using any form understood by **string is boolean**.
3. As a Tcl variable, using standard **$** notation. The variable's value will be used as the operand.
4. As a string enclosed in double-quotes. The expression parser will perform backslash, variable, and command substitutions on the information between the quotes, and use the resulting value as the operand
5. As a string enclosed in braces. The characters between the open brace and matching close brace will be used as the operand without any substitutions.
6. As a Tcl command enclosed in brackets. The command will be executed and its result will be used as the operand.
7. As a mathematical function whose arguments have any of the above forms for operands, such as **abs($x)**. See below for a list of defined functions.

Operators

The valid operators are listed below, grouped in decreasing order of precedence:

- + ~ !
Unary minus, unary plus, bit-wise NOT, logical NOT. None of these operators may be applied to string operands, and bit-wise NOT may be applied only to integers.

*** / %**
Multiply, divide, remainder. None of these operators may be applied to string operands, and remainder may be applied only to integers. The remainder will always have the same sign as the divisor and an absolute value smaller than the divisor.

+ -
Add and subtract. Valid for any numeric operands.

<< >>
Left and right shift. Valid for integer operands only. A right shift always propagates the sign bit.

< > <= >=
Boolean less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise. These operators may be applied to strings as well as numeric operands, in which case string comparison is used.

== !=
Boolean equal and not equal. Each operator produces a zero/one result. Valid for all operand types.

eq ne
Boolean string equal and string not equal. Each operator produces a zero/one result. The operand types are interpreted only as strings.

&
Bit-wise AND. Valid for integer operands only.

^
Bit-wise exclusive OR. Valid for integer operands only.

|
Bit-wise OR. Valid for integer operands only.

&&
Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise. Valid for boolean and numeric (integers or floating-point) operands only.
||
Logical OR. Produces a 0 result if both operands are zero, 1 otherwise. Valid for boolean and numeric (integers or floating-point) operands only.
x**?**y**:**z
If-then-else, as in C. If x evaluates to non-zero, then the result is the value of y. Otherwise the result is the value of z. The x operand must have a boolean or numeric value.

In addition to the operators in the core TCL language, the following operators have been added within iRules:

contains
Tests if one string contains another string.
ends_with
Tests if one string ends with another string.
equals
Tests if one string equals another string.
matches
Tests if one string contains a match of a second string.
matches_regex
Tests if one string matches a given regular expression.
starts_with
Tests if one string starts with another string.
and
User-friendly equivalent for Logical AND (&&).
not
User-friendly equivalent for Logical NOT (!).
or
User-friendly equivalent for Logical OR (||).

<u>Math Functions</u>

The following TCL based math functions have been included in iRules

abs(arg)
Returns the absolute value of arg. Arg may be either integer or floating-point, and the result is returned in the same form.
double(arg)
If arg is a floating-point value, returns arg, otherwise converts arg to floating-point and returns the converted value.
int(arg)
f arg is an integer value of the same width as the machine word, returns arg, otherwise converts arg to an integer by truncation and returns the converted value.
rand()
Returns a pseudo-random floating-point value in the range (0,1). The generator algorithm is a simple linear congruential generator that is not cryptographically secure. Each result from **rand** completely determines all future results from subsequent calls to **rand**, so **rand** should not be used to generate a sequence of secrets, such as one-time passwords. The seed of the generator is initialized from the internal clock of the machine or may be set with the **srand** function.
round(arg)
If arg is an integer value, returns arg, otherwise converts arg to integer by rounding and returns the converted value.
srand(arg)
The arg, which must be an integer, is used to reset the seed for the random number generator of **rand**. Returns the first random number (see **rand()**) from that seed. Each interpreter has its own seed.

wide(arg)
Converts arg to an integer value at least 64-bits wide (by sign-extension if arg is a 32-bit number) if it is not one already.

Types, Overflow, and Precision

All internal computations involving integers are done with the C type long, and all internal computations involving floating-point are done with the C type double. When converting a string to floating-point, exponent overflow is detected and results in a Tcl error. For conversion to integer from string, detection of overflow depends on the behavior of some routines in the local C library, so it should be regarded as unreliable. In any case, integer overflow and underflow are generally not detected reliably for intermediate results. Floating-point overflow and underflow are detected to the degree supported by the hardware, which is generally pretty reliable.

Conversion among internal representations for integer, floating-point, and string operands is done automatically as needed. For arithmetic computations, integers are used until some floating-point number is introduced, after which floating-point is used. For example,

```
expr 5 / 4
```

returns 1, while

```
expr 5 / 4.0
expr 5 / ( [string length "abcd"] + 0.0 )
```

both return 1.25. Floating-point values are always returned with a ``**.**'' or an **e** so that they will not look like integer values. For example,

```
expr 20.0/5.0
```

returns **4.0**, not **4**.

String Operations

String values may be used as operands of the comparison operators, although the expression evaluator tries to do comparisons as integer or floating-point when it can, except in the case of the **eq** and **ne** operators. If one of the operands of a comparison is a string and the other has a numeric value, the numeric operand is converted back to a string using the C sprintf format specifier **%d** for integers and **%g** for floating-point values. For example, the commands

```
expr {"0x03" > "2"}
expr {"0y" < "0x12"}
```

both return 1. The first comparison is done using integer comparison, and the second is done using string comparison after the second operand is converted to the string **18**. Because of Tcl's tendency to treat values as numbers whenever possible, it isn't generally a good idea to use operators like **==** when you really want string comparison and the values of the operands could be arbitrary; it's better in these cases to use the **eq** or **ne** operators, or the **string** command instead.

Performance Considerations

Enclose expressions in braces for the best speed and the smallest storage requirements. This allows the Tcl bytecode compiler to generate the best code.

As mentioned above, expressions are substituted twice: once by the Tcl parser and once by the **expr** command. For example, the commands

```
set a 3
set b {$a + 2}
expr $b*4
```

return 11, not a multiple of 4. This is because the Tcl parser will first substitute **$a + 2** for the variable **b**, then the **expr** command will evaluate the expression **$a + 2*4**.

Most expressions do not require a second round of substitutions. Either they are enclosed in braces or, if not, their variable and command substitutions yield numbers or strings that don't themselves require substitutions. However, because a few unbraced expressions need two rounds of substitutions, the bytecode compiler must emit additional instructions to handle this situation. The most expensive code is required for unbraced expressions that contain command substitutions. These expressions must be implemented by generating new code each time the expression is executed.

Refer to The iRules Optimization Tech Tip on Expressions and Braces for a deeper dive on how to optimize your expressions

Conclusion

The "if" statement is the foundation of any iRule you write. While it's not important for you to know all of these details, this document can serve as a solid reference that you can look back to.

Links
iRules Optimizations #2 - Expressions and Braces - http://devcentral.f5.com/Default.aspx?tabid=63&articleType=ArticleView&articleId=110
TCL eval Documentation - http://aspn.activestate.com/ASPN/docs/ActiveTcl/8.4/tcl/TclCmd/expr.htm

Credit where credit is due: Portions of this Tech Tip are taken from the TCL expr documentation at http://aspn.activestate.com/ASPN/docs/ActiveTcl/8.4/tcl/TclCmd/expr.htm and http://tmml.sourceforge.net/doc/tcl/index.html

Get the Flash Player to see this player.