13. ARITHMETIC EXPRESSIONS

Elvis can evaluate expressions involving numbers, strings, and boolean values, using a C-like syntax. These are used in several EX commands, one VI command, and a couple of other situations.

There are two syntaxes. The normal syntax is extremely similar to C, and is used in circumstances where you probably would never use a literal value, such as for the :if command. The simpler syntax makes literal values easier to enter, while still making the full power of the expression evaluator available if you need it.

13.1 Normal (C-like) Syntax

The :calculate command uses the normal syntax and displays the results. We'll use it for most of the examples in this section.

The normal syntax is intended to resemble the syntax of the C programming language very closely. You can't define your own functions or use flow-control constructs though; you can only use expressions. In traditional C documentation, these would be called "rvalues." Basically that means you can use literal values, option names, operators, parentheses, and some built-in functions.

13.1.1 Primary expressions

Literals can be given in any of the following formats:

"*text*"
: Any text in double-quotes is taken literally. The usual C escapes are supported: \b, \E (uppercase, representing the **Esc** character), \f, \n, \r, and \t. Also, you can use \\ for a literal backslash character, or \" for a literal double-quote character within a string.

\$
\(
\)
****
: You can use a backslash to quote a single dollar sign, parenthesis, or backslash as though it was a string of length 1. This was done mostly for the benefit of the simpler syntax, where these four character are normally the only ones which have any special interpretation.

digits
: Any word which contains only digits will be taken as a literal value. Generally this value will be interpreted as a number, but

internally the expression evaluator always stores values as strings. Some operators look at their arguments and act differently depending on whether those strings happen to look like numbers or Boolean values.

0*octaldigits*
0x*hexdigits*
'*character*'
Octal, hex, and character constants can be used in expressions. These are converted to decimal when they are parsed, before they are passed to any operator or function. Passing an octal, hex, or character constant therefore is exactly like passing the equivalent decimal number. Elvis supports escapes as character constants: '\0', '\b', '\E', '\f', '\n', '\r', and '\t'.

true
false
These can be used as Boolean literals. Technically, they are implemented via options (as described below) named true and false. All of the boolean operators accept "false", "0", "", or the value of the false option as Boolean **false** values, and anything else as a Boolean **true** value.

The following examples produce *exactly* identical results.

```
:calc "8"
8
:calc 8
8
:calc 010
8
:calc 0x8
8
:calc '\b'
8
```

You can also use option names in elvis the same way you would use variable names in C.

```
:calc list
false
:calc scroll
12
:calc display
normal
```

Additionally, a dollar sign followed by the name of an environment variable is replaced by the value of that environment variable. If there is no such environment variable, then elvis will act as though it exists and has a null value.

In some circumstances, you can use a dollar sign followed by a digit to access special arguments. This is used in error messages and also in the values of a few options, as described in section 13.6. These special arguments can only be supplied by elvis' internal code, and it only supplies them in a few special circumstances so you can't use them in :calculate, for example.

13.1.2 Operators

The following operators are available. When passed integer values, these operators act like their C counterparts. When passed string values, most of them concatenate their arguments with the operator name in between, but some of them do something that is useful for strings, as described below. Items at the top of this list have a higher precedence than those lower down.

(no operator)
Any two expressions placed side-by-side with no operator between them will be concatenated as strings. C does this for literal strings, but elvis does it for anything.

~
Perform a bitwise NOT operation on the argument, if it is a number.

!
Return **true** if the argument is **false** and vice versa.

*** / %**
The usual arithmetic operators. (% is the modulo operator.)

Also, the / operator can be used to combine a directory name and a file name, to form an absolute pathname. Here are some examples showing how this works in DOS:

```
:set dir home
directory=C:\temp  home=C:\
:calc dir/"tempfile"
C:\temp\tempfile
:calc home/"elvis.rc"
C:\elvis.rc
```

+ -
The usual arithmetic operators. Note that there is no special unary - sign; the minus sign serves double-duty. Because C normally gives the unary - sign a higher precedence than other operators and elvis doesn't, you may occasionally need to enclose negated values in parentheses to achieve the same effect.

<< >>
For integers these operators perform bitwise shifting, exactly like C. However, if the left argument is a string and the right argument is a number then elvis will pad or truncate the string to make its length match the number argument. << pads/truncates on the right, and >> pads/truncates on the left.

```
:calc \[("port" << 6)\]
[port  ]
:calc \[("starboard" >> 6)\]
```

[rboard]

< <= > >= == !=
Compare the arguments and return **true** if the comparison holds, and **false** otherwise. If both arguments look like numbers, then they will be compared as numbers; otherwise they will be compared as strings.

&
Bitwise AND of the arguments, if they're numbers.

^
Bitwise XOR of the arguments, if they're numbers.

|
Bitwise OR of the arguments, if they're numbers.

&&
Returns **false** if either argument is one of the four false string values, and **true** otherwise. Both arguments are always evaluated; this is different from C, where the right argument is only evaluated if the left argument is **true**.

||
Returns **false** if both arguments are one of the four false string values, and **true** otherwise. Both arguments are always evaluated; this is different from C, where the right argument is only evaluated if the left argument is **false**.

?:
This one is tricky because internally elvis always uses binary (two operand) operators. In C this is a ternary operator but in elvis it is implemented as two binary operators which cooperate in a subtle way so they seem like a single ternary operator. You probably don't need to know the details, but the upshot of it all is that 1) It associates left-to-right (instead of right-to-left as in C), and 2) The : and third argument are optional; if omitted, then elvis mentally sticks :"" on the end of the expression.

,
(That's a comma, not an apostrophe.) Concatenates two strings, with a comma inserted between them. This can be handy when you're passing arguments to the quote() and unquote() functions.

;
Concatenates two strings without inserting any extra characters. The result is exactly like (no operator), except that (no operator) has an extremely high precedence, and ; has an extremely low precedence.

```
:calc 1+2 3*4
93
:calc 1+2;3*4
312
```

13.2 Simpler Syntax

In comparison to the normal expression syntax, the simpler syntax makes it easier to enter literal strings because outside of parentheses the only special characters are the backslash, dollar sign, and parentheses. (These may be escaped by preceding them with a backslash.) Inside parentheses, the normal syntax is used.

The :eval command uses the simpler syntax, and the :echo command displays its arguments. These commands can be used together to experiment with the simpler syntax, the same way we used :calculate to experiment with the normal syntax.

```
:eval echo TERM=$TERM
TERM=xterm
:eval echo home=(home)
home=/home/steve
:eval echo 2+2=(2+2)
2+2=4
```

13.3 Functions

There are several built-in functions. When you call one of these functions, there must **not** be any whitespace between the function name and the following parenthesis. The built-in functions are:

FUNCTION(ARG)	RETURN VALUE
strlen(string)	number of characters in the string
toupper(string)	uppercase version of string
tolower(string)	lowercase version of string
isnumber(string)	"true" iff string is a decimal number
htmlsafe(string)	convert characters from ASCII to HTML
hex(number)	string of hex digits representing number
octal(number)	string of octal digits representing number
char(number)	convert number to 1 ASCII char, as a string
quote(list,str)	insert backslashes before chars in list
unquote(list,str)	remove backslashes before chars in list
exists(file)	"true" iff file exists
dirperm(file)	string indicating file attributes
dirfile(file)	filename.ext part of a path
dirname(file)	directory part of a pathname
dirdir(file)	directory, like dirname(file)
dirext(file)	extension (including the .)
basename(file)	filename without extension
fileeol(file)	newline style of the file
absolute(file)	return a full path-name for a given file
getcwd()	return the current working directory name
elvispath(file)	locate a file in elvis' configuration path

shell(program)	run program, and return its output
knownsyntax(file)	language of a file if in elvis.syn, else ""
buffer(bufname)	"true" iff buffer exists
alias(name)	"true" iff an alias exists with that name
current(item)	value indicating an aspect of elvis' state
line(bufname,num)	return the contents of a given line
feature(name)	"true" iff a given feature is supported

(Note: "iff" is short for "if and only if")

Some of these deserve further comment.

The **isnumber()** function uses the same test that the operators use when deciding whether to use the string version or the number version of their behavior. You can use **isnumber()** to predict how operators will behave.

The **hex()** and **octal()** functions return strings which look like C-style hex or octal constants, respectively. The **isnumber()** function will return **false** when passed one of these strings; they are no longer considered to be numbers. In fact, the only reason you can use hex and octal literals is because they are converted into decimal strings by the parser, before evaluation even begins. The following example demonstrates that hex literals are converted to decimal, and that the value returned by **hex()** is something else.

```
:calc strlen(0xff)
3
:calc strlen(hex(255))
4
```

The **char()** function returns a one-character string; that character's decimal value will be the argument number. For example, "char(65)" returns "A". Note that the returned value does *not* look quite like a character constant.

The **quote()** and **unquote()** functions add and remove backslashes before special characters. The backslash character itself is always considered to be "special," so backslashes are converted to double-backslashes and vice versa. In the argument, any characters which precede the first comma are used as a list of other special characters, and the remainder of the argument is the string to be quoted/unquoted. For example...

```
:set t="/* regexp */"
:set r="*^$/.["
:eval /(quote(r, t))/
```

... will search for the next instance of of the literal string "/* regexp */". The '/' and '*' characters won't be treated as metacharacters in the regular expression, because the quote() function inserts backslashes before them. Also, notice that the comma operator concatenates two strings and inserts a comma between them. That's handy!

The **dirperm()** function returns one of the following strings to indicate the file's type and permissions:

"invalid"
The argument is malformed; it could not possibly be a valid file name.

"badpath"
The argument is a pathname, and one or more of the directories named in that pathname either doesn't exist or is something other than a directory.

"notfile"
The argument is the name of something other than a file; for example, it may be a directory.

"new"
There is no file, directory, or anything else with the given name.

"unreadable"
The file exists but you don't have permission to read it.

"readonly"
The file exists and you can read it, but you don't have permission to write to it.

"readwrite"
The file exists and you can read or write it.

The **fileeol()** function opens the file in binary mode, reads the first hundred bytes, and inspects those bytes to make a guess about the file's newline format. It is commonly used for setting the readeol option. fileeol() returns one of the following strings:

"unix"
It appears to be a text file which uses Line Feed characters for newlines.

"dos"
It appears to be a text file which uses Carriage Return/Line Feed pairs for newlines.

"mac"
It appears to be a text file which uses Carriage Return characters for newlines.

"binary"
It appears to be a binary file. Note that all HTTP and FTP URLs are assumed to be binary.

"text"
Anything else; e.g., a non-existent file or empty file.

The **elvispath()** function searches through the directories listed in the elvispath option's value, looking for the argument file name. If it is found, then the full pathname of the file is returned; otherwise it returns a null string.

The **absolute()** function attempts to construct a full pathname for a given file name. If the given file name is actually a URL, or if it is already a full pathname, then this function returns it unchanged. Otherwise it combines the getcwd() value with the given name.

The **knownsyntax()** function determines whether the given file can be displayed in the syntax display mode. It does this by looking for the file name extension in the elvis.syn configuration file. If the file's extension is listed there, then this function returns the name of the language. Otherwise, it just returns an empty string.

The **current()** function examines elvis' internal variables, and returns a string indicating the value of one of them. The argument determines which variable is examined, as follows:

current("line")
Current line number.

current("column")
Current column number.

current("word")
The word at the cursor location. If the cursor isn't on a word, then this returns an empty string. (Note: To get the contents of the current line, use the **line()** function.)

current("tag")
If the showtag option is true, then this returns the name of the tag that is defined at the cursor location, or the nearest one before it. If the showtag option is false, or the cursor is located above the first tag defined in this file, then current("tag") will return an empty string.

current("mode")
Current key parsing mode. This returns the same string that the showmode option displays, except that this function converts it to all lowercase, and strips out whitespace. The usual return values are "command", "input", and "replace". If the window isn't editing the its main buffer (i.e., if you're entering an ex command line, regular expression, or filter command) then this function will return an empty string.

current("selection")
Visible selection type. This returns one of "character", "rectangle", or "line" to indicate the type of visible selection which is currently marked in the window, or an empty string if no visible selection is marked.

current("next")
Next file. This returns the name of the file that the :next command would load, or an empty string if you're at the end of the args list.

current("previous")
Previous file. This returns the name of the file that the

:previous command would load, or an empty string if you're at the start of the args list.

current("tagstack")
If the window's tag stack is empty, this returns "". Otherwise it returns the name of the buffer to which :pop would move the cursor.

The **line()** function returns the contents of a single line from an edit buffer. If two arguments are given, then the first argument is taken to be the name of the edit buffer, and the second argument is used as the line number. If only one argument is given, then it is assumed to be a line number within the current buffer. If no arguments are given, then it assumes it should use the current line of the current buffer. If the line is too long to fit in the result variable, then it is truncated.

The **feature()** function is intended to allow you to write EX scripts which work with different configurations of elvis. For example, you can compile elvis without support for the hex display mode; if you do that, then feature("hex") will return **false**. Currently **feature()** returns **true** for all supported display modes, network protocols, and maybe "showtag" and "lpr"; it returns **false** for anything else. As new features are added to future versions of elvis, I expect to add them to **feature()**'s list.

13.4 EX Commands Which Use Expressions

The :calculate command evaluates its argument using the normal syntax, and displays the result.

The :if command evaluates its argument using the normal syntax. If the resulting value is any Boolean **true** value then a flag is set; otherwise the flag is reset. After that, you can use :then and :else commands to conditionally execute some commands, depending on the state of that flag.

The :eval command evaluates its arguments using the simpler syntax. The resulting string value is then interpreted as an EX command line. This gives you a way to use the expression evaluator with commands which otherwise wouldn't evaluate expressions.

The :let command allows you to change the values of options. Its syntax is ":let *option*=*expression*", where *expression* is any expression using the normal syntax. You can use this to change the value of any unlocked option, similarly to :set.

```
:set i=14
:calc i
14
:let i=i+1
:set i?
i=15
:eval set i=(i*2)
:calc i
```

```
30

:let elvispath="."
:let list="false"
:let sidescroll=0x10
```

13.5 VI Commands Which Use Expressions

There is only one way to use expressions in a visual command: Move the cursor to the start of some expression in your edit buffer, hit the lowercase **v** key, move to the other end, and then hit the **=** key. Elvis will then evaluate the highlighted expression, and replace the original expression with the result.

Note that the = operator only works this way when used with the v command for marking characters. If you visibly mark lines, or use the traditional **=***movement* syntax, then elvis will send the selected lines though the external filter program named in the equalprg option.

The # command doesn't use expressions, but it does perform some simple math.

13.6 Other Uses of Expressions

13.6.1 Messages

All of elvis' warning and error messages are actually expressions, using the simpler syntax. When outputting a message, elvis may supply other parameters which are accessible as **$1** through **$9**. See the Messages chapter for a longer description of how elvis handles messages.

13.6.2 Options

The ccprg and makeprg options' values are expressions, using the simpler syntax. When evaluating these expressions, **$1** is replaced by whatever arguments are supplied on the ex command line, and **$2** is replaced by the the name of the file being edited.

13.6.3 File Names

File names are evaluated as expressions (using the simpler syntax), primarily as a means for expanding environment variable names. This is done prior to wildcard expansion.

The full power of the expression evaluator is available; you can use it to do more than just expand environment variable names. For example, you could store the name of a file in one of the user options, and then later use that option name in parentheses wherever a filename was expected.

```
:set f=myfile.txt
:w (f)
```

```
wrote myfile.txt, ...
```

If you use this trick, remember that it only works when elvis is expecting a file name. It won't work when invoking external programs, because elvis doesn't know which program arguments are supposed to be file names. Elvis always passes program arguments literally.

Recall that when a backslash character is followed by an alphanumeric character, both the backslash and the alphanumeric character become part of the resulting value. This was done mostly for the benefit of file names. If the backslash was always dropped then MS-DOS users would have a heck of a time entering pathnames of files! By making the backslash a little smarter, we avoid that problem.

```
:eval echo c:\tmp \(notice the backslashes\)
c:\tmp (notice the backslashes)
```

To simplify the task of writing portable ex scripts, the behavior of the **/** operator has been extended. When one or both of its arguments are strings, it concatenates them as a directory name and a file name, yielding a full pathname.