The SMOG Language Reference

Contents

The SMOG language is a scripting language that's intended to make it easy to write insecure programs. Technically it runs line-by-line, keeping track of the state the program's been put in by previous lines in the global state, but some structures appear to be multi-line and can be safely imagined as such by a user.

1 Comments

Single-line comments (the only kind supported) are indicated with a double slash:

```
// this is a comment
print("this line runs") // but this is another comment
```

Normally this is where you'd be encouraged to write clear and plentiful comments in your code, but since we're trying to encourage insecure programming you can let your imagination run wild.

2 Variables

2.1 Variable Definition

Variables are defined by the *let* form:

```
let <name> = <expression>
```

<name> may be any string of alphabetic characters and underscores. *<expression>* may be any expression, see the **expressions** section. For example:

```
let x = 3
let hello_world = "hello" ++ " " ++ "world"
let _TOP = first(!some_list!) + 2
```

2.2 Variable Use

The values of variables can be accessed in two different ways, one of which is safe and one of which is versatile. The versatile way is to wrap the name of the variable in $ signs:

```
let var = 3
print($var$)
```

This kind of variable invocation is replaced with its value before the line is parsed. This means that you can put strings containing SMOG code in them and have that code execute, but it also means you need to be more careful with them. For instance, in:

```
let x = "hello world"
let y = $x$
```

the second line will parse as:

```
let y = hello world
```

which is a syntax error. Instead you'd have to wrap the variable in quotes:

```
let y = "$x$"
```

but watch out for string variables that contain quotes themselves.

The safe way is to wrap the name of the variable in exclamation marks:

```
let x = "hello world"
let y = !x!
```

This approach delays variable expansion until after parsing, so whether a line is a syntax error cannot depend on the value of the variable. Note that wrapping such a variable name in quotes will no longer work, it'll just be read as a literal string.

3 Flow Control

3.1 If Statements

The *if/elseif/else* form allows chunks of code to run or not depending on the value of some expression. In particular, a branch that tests an expression runs if that expression has any value except 0.

```
if <expression>:
  ...
elseif <expression>:
  ...
else:
  ...
endif
```

The *else* and *elseif* branches are optional, and there can be any number of either in any order as long as *if* is the first one. However, no more than one branch of any if statement will run, so 1) if an *elseif* is reached whose expression is not equal to 0 but an earlier branch already ran, that branch won't run, and 2) any branch after an *else* certainly won't run. For example:

```
if first($a_list$):
  print("first value is not zero")
endif

if 0:
  print("won't happen")
elseif 1:
  print("will happen")
elseif "true":
  print("won't happen")
endif

if !var!:
  print("var is not zero")
else:
  print("var is zero")
endif
```

3.2 Case Statements

The *case* form is less general than the *if* form but shorter to write. It takes the value of an expression and runs the branch, if any, whose label is equal to the value of that expression.

```
case <expression>:
<literal 1> ... <literal n>:
  ...
<literal 1> ... <literal n>:
  ...
else:
  ...
endcase
```

There can be any number of sections. A literal can be a literal string, number or list, and each label can have any (nonzero) number of them separated by spaces, any of which will match. Like with the *if* statement, *else* is a label that matches anything and once a branch has been run no further branches will run even if they match.

For example:

```
case $n$:
1 3 5 7 9:
  print("odd number")
0 2 4 6 8 10:
  print("even number")
```

```
else:
  print("not an integer 0-10")
endcase

case "$name$":
"Alice" "alice":
  let a = 1
"Alice":
  let a = "won't happen"
endcase
```

3.3 Goto Statements

The lines of a SMOG program are implicitly numbered from the start of the program, beginning at 0. The *goto* statement makes it possible to jump between them.

```
goto <expression>
```

The result of the expression must be an integer, but the program will halt cleanly if it isn't between zero and the highest line number in the program inclusive.

For example:

```
let n = 5        //       0
goto 3           // --.   1
let n = -1       //   |   2
goto !n!         // <-'-. 3
print("hello")   // <-. | 4
goto 4           // <-'-' 5
```

Will print "hello" repeatedly forever

4 Expressions

As well as being part of other forms, expressions make valid lines on their own. Although their values are discarded, some expressions (in particular function calls) are useful like this for their side-effects.

4.1 Literals

A literal expression is one of a string, a number or a list, although strictly speaking this category probably shouldn't be called 'literals' because lists can contain non-literal expressions. Example of literals:

```
"this is a string literal"
123.5
["here's a list", -23, some_function(1)]
```

4.2 Function Calls

Functions take a list of arguments and return some value that may depend on those arguments. They may also have side-effects. Unlike statements, all functions have a return value and can therefore be used in expressions. There is no support for user-defined functions, a list of the builtin ones is provided in the section **builtins**. Function calls use the common paren-argument-list syntax:

```
<function name>(<expression 1>, ... , <expression n>)
<function name>() // if zero arguments
```

An error will be raised if the wrong number of arguments is provided or the arguments are of the wrong type (out of the three SMOG types: *string*, *number*, *list*, plus the meta-type *any* for functions that don't care about the types of some of their arguments). Individual functions may raise errors for other reasons, otherwise the function evaluates to a value that depends on which function it is.

For example:

```
print("hello world") // evaluates to 1
list_add_front($some_list$, 3) // evaluates to the list in some_list with
                               // an extra 3 added on the front
ascii_code_to_char(!n!) // evaluates to a string containing the character
                        // whose ascii code is equal to n, if one exists
```

4.3 Operators

Operators are a special kind of function that can be called with a different syntax. All operators take exactly two arguments.

```
<expression> <operator> <expression>
```

Otherwise, operators work exactly like functions. The parser doesn't have any order of precedence for operators, instead the leftmost operator takes precedence.

For example:

```
1 + 2 * 3 // parses as (1 + 2) * 3
1 * 2 + 3 // parses as (1 * 2) + 3
"hello" ++ " world" // evaluates to "hello world"
```

5 Builtins

Although the forms above are technically turing-complete on their own, they aren't very convenient and they don't have any capacity for user interaction. Therefore, there are the following builtin functions and operators available. Make good use of them, because you can't define any more!

5.1 Builtin Functions

The following functions are available. Each is specified in the form

```
<function name>(<type of argument 1>, ..., <type of argument n>):
<description>
```

The possible types are *string*, *number*, *list* and *any* (for arguments whose type doesn't matter).

`print(any):`
Prints the value of the supplied argument to output. Always returns 1.

`first(list):`
Returns the first member of the supplied list, which must not be empty.

`rest(list):`
Returns the supplied list (which must not be empty) with the first member omitted.

`contains(list, any):`
Returns 1 if the second argument is a member of the list, otherwise 0.

`list_add_front(list, any):`
Returns the supplied list with the second argument appended onto the front.

`list_add_back(list, any):`
Returns the supplied list with the second argument appended onto the end.

`list_remove(list, any):`
Returns the supplied list with the first instance of the second argument removed from it, if any.

`implode(list, string):`
Returns a string made by inserting the second argument between each member of the first. The list must be a list of strings only, but it may be empty in which case the result is the empty string.

`string_to_char_list(string):`
Returns a list whose members are strings, each one containing a single character of the original string in order.

`char_to_ascii_code(string):`
The argument must be a string of exactly one character. Returns a number which is the ascii code of that character.

`ascii_code_to_char(number):`
The reverse of *char_to_ascii_code*. The number must be in 0-256.

`split_on(string, string):`

Returns a list which is made by removing every instance of the second string from the first and splitting it at the points those removals happened.

`read():`

Waits for the user to type some input and press enter, then returns that input as a string.

5.2 Builtin Operators

The following operators are available. All the available operators take the same type on both sides, athough they may not necessarily be symmetrical (if that type is *any* then it need not be the same specific type on each side every time). Each is specified in the form

```
<operator> (<type>):
<description>
```

`+ (numbers):`

Returns the sum of its operands.

`- (numbers):`

Returns its right hand side subtracted from its left hand side.

`* (numbers):`

Returns the product of its operands.

`/ (numbers):`

Returns its left hand side divided by its right hand side.

`% (numbers):`

Returns its left hand side modulo its right hand side.

`< (numbers):`

Returns 1 if its left hand side is strictly less than its right hand side, otherwise 0.

`> (numbers):`

Returns 1 if its left hand side is strictly greater than its right hand side, otherwise 0.

`++ (strings):`

Returns its operands appended together.

`| (any):`

Returns 1 if either of its operands is nonzero, 0 otherwise.

`& (any):`

Returns 1 if both of its operands are nonzero, 0 otherwise.

`== (any):`

Returns 1 if its operands are equal to one another, 0 otherwise.

`!= (any):`

Returns 1 if its operands are not equal to one another, 0 if they are.