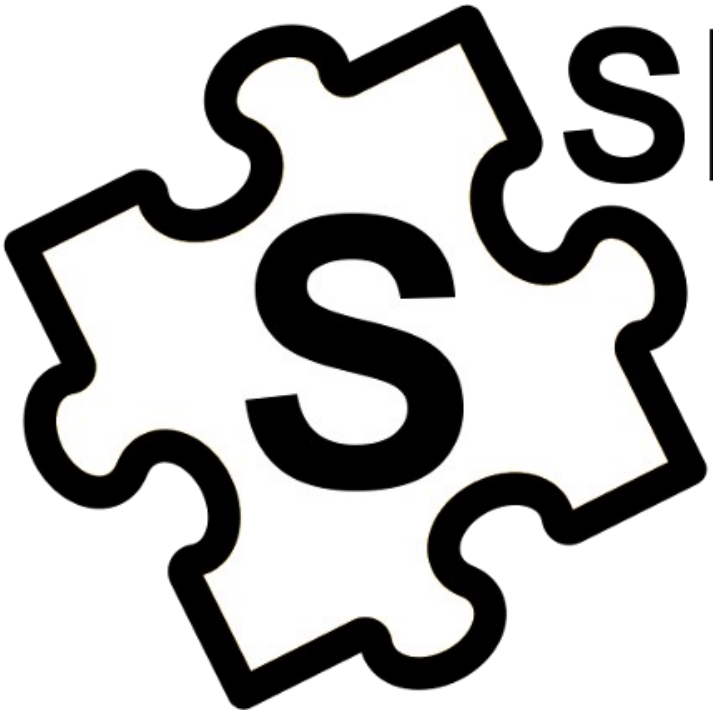


SIMPLE

Simple Interpolated Markup Language

Version 0.00154

Dan Hetrick

Last edited on March 31, 2013
Revision 187

Table of Contents

Introduction

SIMPLE (**S**imple **I**nterpolated **M**ark-u**P** **L**anguag**E**) is a procedural, Turing-complete programming language designed to be easy to teach beginners to computer programming. It uses a two-pronged paradigm:

- Rather than using C's brackets or Python's indentation to delimit code blocks, SIMPLE uses a style derived from HTML and XML: markup tags.
- Unlike other high-level programming languages, SIMPLE strives to be as close to natural English as possible. Other than the symbols required to do math, for markup tags, and for interpolation, SIMPLE uses no symbolic constructs. Statement should state, plainly, what they do in English, with a minimum of punctuation or non-numeric symbols; instead of C's "`if(i==3){`", which is difficult for non-programmers to read, SIMPLE would use "`if $i equals 3`", an easier construct to read for non-programmers.

SIMPLE was designed for the "HTML Generation"; that is, people who are familiar with HTML and markup tags, yet are not familiar with programming. It is *not* a complete application solution. SIMPLE is limited by design; it's designed to be a teaching tool. `sim.pl`, the SIMPLE compiler, takes in a SIMPLE program (a specific kind of XML document, detailed below) and compiles it into stand-alone Perl (the output program requires only a default installation of Perl). The compiler also checks programs for correct syntax, returning the line number of any errors it finds. It can't detect all errors; errors involving user or system input are displayed when the compiled program is executed, returning the line number in the original SIMPLE program where the error occurred. These two features make debugging SIMPLE programs fairly easy; runtime error messages (which include the line number in the SIMPLE program where the error occurred) are inserted into the compiled program automatically making debugging even easier.

There are three things this programming language is designed to teach:

- **Variables**[1]. Almost every programming language uses this concept.
- **Interpolation**[2]. Many common programming languages, like PHP and Perl, use some sort of interpolation system. Even C/C++ uses one (for an example, see the standard library function `printf`).
- **Variable Scoping**[3]. A feature of every major programming language and almost *none* of the beginner programming languages.

SIMPLE was heavily influenced by two different languages: Perl and XML. The interpolation system is borrowed from Perl, allowing variables to be interpolated in strings using the **$** sign (see *Variables*

---

1 *"...a variable is a storage location and an associated symbolic name (an identifier) which contains some known or unknown quantity or information, a value. The variable name is the usual way to reference the stored value; this separation of name and content allows the name to be used independently of the exact information it represents."*
From Wikipedia: http://en.wikipedia.org/wiki/Variable_(computer_science)

2 *"Variable interpolation**...**is the process of evaluating an expression or string literal containing one or more variables, yielding a result in which the variables are replaced with their corresponding values in memory."*
From Wikipedia: http://en.wikipedia.org/wiki/Variable#Variable_interpolation

3 *"...a scope is the context within a computer program in which a variable name or other identifier is valid and can be used, or within which a declaration has effect. Outside of the scope of a variable name, the variable's value may still be stored, and may even be accessible in some way, but the name does not refer to it; that is, the name is not bound to the variable's storage."*
From Wikipedia: http://en.wikipedia.org/wiki/Variable_scoping

and Interpolation, below). The overall syntax is borrowed from XML; all code blocks[4] are in the form of XML tags. I view this language as a blend of the power of string manipulation in Perl with the parse-ability of XML. Support for regular expressions (also known as a "regex") is *not* included; regular expressions are difficult to explain to experienced programmers, and as SIMPLE is a teaching language we don't need to bog down users with them (they can always learn them later). Likewise, there are only two markup tags users must learn: **`import`** and **`subroutine`**.

SIMPLE strives to be easily read by non-programmers. To that end, SIMPLE limits its syntax to help users write readable code. Limitations include:

- **Nested flow control blocks are forbidden.** An **`if`** block can't contain another **`if`** block. **`while`** blocks can't contain another **`while`** block.
- **One, and only one, command per line of code.** Code with multiple commands per line can be difficult for non-programmers to read or understand.
- **Commands are as close to natural English as possible.** A non-programmer should be able to figure out what a program does by reading its source code.

4 *"...a block is a section of code which is grouped together. Blocks consist of one or more declarations and statements."* From Wikipedia: http://en.wikipedia.org/wiki/Block_(programming)

The SIMPLE Code Compiler

The SIMPLE Code Compiler is a Perl program named "**sim.pl**". All SIMPLE programs are compiled to a stand-alone Perl program; that means that, once a SIMPLE program is compiled, it will run on any platform that Perl runs on, and does not require **sim.pl** to run.

If **sim.pl** is ran from the command-line with no arguments, basic usage information is displayed:

```
localhost:~ user$ perl sim.pl
SIMPLE Code Compiler 0.00154 (dragon)
Usage: sim.pl [OPTIONS] FILENAME
Options:
-h(elp)                 Display this text
-v(erbose)              Displays additional information while compiling
-o(utput) FILENAME      Sets the output filename; default is "out.pl"
-d(efault) SUBROUTINE   Sets the default subroutine; default is "main"
-e(xecutor)             Strips executor stub code from output
-s(tdout)               Prints output to STDOUT instead of a file
-i(nclude)              Includes the original source into the compiled source
localhost:~ user$
```

To compile a SIMPLE program, pass it as the only argument to **sim.pl**. By default, this will create a new file name "**out.pl**" which contains the output Perl code. If you want to write the compiled code to a different file, use the **-o** option.

If any syntax errors in the code are found, the errors are displayed to the user and compilation is aborted. Where the error was in the program is reported to the user; however, the line number reported is relative to what subroutine the error is in. For example, take a look at the following program:

```
1 <subroutine name="main">
2      print "This program has an error in it!"
3      this_command_doesnt_exist
4 </subroutine>
```

The command on line 3 (**this_command_doesnt_exist**) will throw an error, as the command, much like its name, doesn't exist. However, when we try to compile it:

```
localhost:~ user$ perl sim.pl buggy_program.sim
1 error found!
Error in 'main' on line 2:  Statement "this_command_doesnt_exist" not recognized
localhost:~ user$
```

Since error occurs on line 2 of the subroutine, that's the line number that is reported.

Example sim.pl Usage

Let's write a basic "hello world" program in SIMPLE. This program will print a greeting to the console. First, open a new text file, and put the following text into it:

```
<subroutine name="main">
     global greeting
     greeting equals "Hello, world!"
     print "Greeting: $greeting"
</subroutine>
```

Save this to a file named "**helloworld.sim**". Now, we will compile the code using **sim.pl**; instead of the default output filename of "**out.pl**", we'll use the more appropriate name "**helloworld.pl**". When compilation is done, we'll run our program to see if it works:

```
localhost:~ user$ perl sim.pl -o helloworld.pl helloworld.sim
localhost:~ user$ perl helloworld.pl
Greeting: Hello, world!
localhost:~ user$
```

You can find this program in the **examples/** directory, named **helloworld.sim**.

SIMPLE Markup Language

SIMPLE uses two markup tags: **`subroutine`** and **`import`**. HTML/XML style multi-line comments (that is, any text between matching tags **`<!--`** and **`-->`**) are supported.

`subroutine` tags contain SIMPLE Code (see below), and allow blocks of code to be independently executed. Each **`subroutine`** tag has one mandatory attribute (**`name`**) and one optional attribute (**`arguments`**). The **`name`** attribute defines what the subroutine's name is; each subroutine name must be unique in the program. The optional **`arguments`** attribute sets what arguments the subroutine will take. This is set as a list of names: one name for each argument, separated by commas. For example, if you wanted a subroutine to take two arguments, the first named "one" and the second named "second", you would use **`arguments="one,second"`**. When executed, a variable is created for each argument containing that argument's value; these can be used just like any other variable in interpolation. Thus, in the previous example, the argument **`one`** can be referenced with **`$one`**, and so on. These argument variables are destroyed (that is, removed from the variable table) at the conclusion of the subroutine. Every SIMPLE program *must* contain a subroutine named **`main`**; this is the "entry point" for the program, and the subroutine that is executed by default (this can be changed with the compiler option **`-d`**). A subroutine must be defined with a **`subroutine`** tag before it can be used in another subroutine; that is, like C/C++, the subroutine must be written first and before any subroutine that calls it.

`import` tags are used to "import" other SIMPLE program files into a program, much like C/C++'s **`#include`**. **`import`** tags have no attributes; they contain a single filename, complete with path. The contents of an imported file and loaded into memory and compiled; any subroutines they contain may be called by any subsequent subroutines. Multiple layers of **`import`** tags are permitted (that is, an imported file can contain **`import`** tags, which contains more **`import`** tags, etc.)

SIMPLE Code Syntax

SIMPLE Code is multi-line text with either a blank line or a single SIMPLE statement, followed by a newline, on each line. XML-style multi-line comments can also be used (any code between **`<!--`** and **`-->`** will be ignored).

There are 24 built-in SIMPLE commands. Each one is issued as a series of tokens, separated by whitespace. If a token contains whitespace in its value, it can be contained in double quotes. For example, if we were to split the string "this is a test" into tokens, it would split into four (4) tokens:

```
Token 1:  this
Token 2:  is
Token 3:  a
Token 4:  test
```

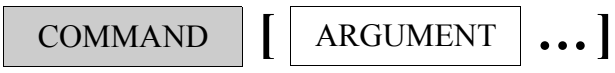
Now, let's look at a more complicated example, showing how double quotes can be used to contain complete tokens.

```
this is "a more complicated" example

This breaks down into four (4) tokens:

Token 1:  this
Token 2:  is
Token 3:  a more complicated
Token 4:  example
```

There are two types of SIMPLE statements: **commands** and **variable assignment**. Commands are formatted like so:



Each SIMPLE command has a different number of arguments that must go in the right order. This includes built-in commands (such as **`print`** and **`exit`**) as well as user-written subroutines. See *SIMPLE Commands*, below.

Variable assignment statements set a variable's value by performing some operation, and they're formatted like so:

SIMPLE features only one variable assignment operator: **`equals`**. This is used when assigning a value to a variable, or on variable creation (with one exception: when creating a variable, it *cannot* use a subroutine's return value as its initial value).

There are six (6) built-in variable assignment commands: **`lowercase`**, **`uppercase`**, **`read`**, **`binread`**, **`ascii`**, and **`character`**. These commands can't be called directly; they must be assigning a variable's value (that is, in a "VARIABLE equals" statement). These use the following format:

VARIABLE equals COMMAND ARGUMENT **[...]**

Variables assigned in the above format have their input interpolated; that is, the statement in the above space marked VARIABLE is interpolated. This allows variable names to be contained in other variables. For example:

```
global variable_name equals "target_variable"
global target_variable "this is not the target"
$variable_name equals "this is the target"

<!-- This will print "this is the target" to the console -->
print $target_value
```

Variables and Interpolation

SIMPLE has two variable types: **`local`** and **`global`**. A variable can contain either a numeric or string[5] value. SIMPLE is loosely typed, meaning that will interpret what to do with a variable depending on context. All command inputs are interpolated; this means that, like Perl, we can insert variable values into a string or statement using a special symbol ('**`$`**').

Variables are marked with a '**`$`**'. For example, assume that we have a variable named '**`myvar`**', with the value '**`John Doe`**'. If we pass the following as an argument to a command, '**`My name is $myvar`**' will be interpreted as '**`My name is John Doe`**'.

If, after all variable interpolation is complete, a statement consists solely of a mathematical expression[6] (i.e., "2+2", "(2+2)*3", or "1+((3*4)/2)+5"), then the mathematical expression is "solved" (that is, completed and turned into a value), with the solution replacing original statement. All mathematical operations can be done using interpolation in SIMPLE. For example, to determine what the average is of 5, 25, and 9, you could try:

```
global value1 equals 5
global value2 equals 25
global value3 equals 9
global average
average equals "($value1+$value2+$value3)/3"
```

The variable **`average`** now contains the correct answer (13).

Variables come in two varieties: **`global`** variables and **`local`** variables. **`global`** variables are global in scope, whereas **`local`** variables are deleted/destroyed at the completion of the current code block. What this means is that **`local`** variables are only accessible inside the subroutine that created them, while **`global`** variables are accessible in any subroutine in the program.

There are several variables that are created automatically, allowing SIMPLE programs to have access to the command line. The built-in variable **`ARGC`** contains the number of arguments the program was executed with. Every argument is placed in a variable named **`ARGx`**, where ***`x`*** is the place of the argument; for example, the first argument would be **`ARG1`**, the second **`ARG2`**, the third **`ARG3`**, and so on. The built-in variable **`ARG0`** contains the filename of the SIMPLE program (for more information, see *Handling Command-line Arguments*, below).

5 *"...a string is...a sequence of characters..."*
From Wikipedia: http://en.wikipedia.org/wiki/String_(computer_science)

6 *"...an expression is a finite combination of symbols that is well-formed according to rules that depend on the context. Symbols can designate numbers (constants), variables, operations, functions, and other mathematical symbols, as well as punctuation, symbols of grouping, and other syntactic symbols."*
From Wikipedia: http://en.wikipedia.org/wiki/Expression_(mathematics)

Subroutines

SIMPLE subroutines allow each program to expand the basic, built-in commands. Calling a subroutine is the same as calling any SIMPLE command: that is, the subroutine's name, followed by arguments to the subroutine.

SUBROUTINE **[** ARGUMENT **...]**

If we're assigning a variable a value from a subroutine, we use the same syntax as when we assign a variable from a command:

VARIABLE equals SUBROUTINE ARGUMENT **[...]**

For example, let's create a subroutine that will create a custom greeting, depending on what arguments are passed to the subroutine.

```
<subroutine name="greeting" arguments="who,where">
     global retval equals "Hello to $who in $where!"
     return $retval
</subroutine>
```

Our subroutine **`greeting`** requires two arguments: **`who`** (the person to greet) and **`where`** (where the person to greet is located). Calling our new subroutine is simple! **`greeting`** requires two arguments, so any call with less (or more) arguments will fail with an error. Let's use our new subroutine to say hello to Dan in Detroit, in the main subroutine of the program:

```
<subroutine name="greeting" arguments="who,where">
     local retval equals "Hello to $who in $where!"
     return $retval
</subroutine>

<subroutine name="main">
     global text
     text equals greeting Dan Detroit
     print $text
</subroutine>
```

If we save this to **`greeting.sim`** and use **`sim.pl`** to compile and run this program, our greeting is printed to the console!

```
localhost:~ user$ perl sim.pl -o greeting.pl greeting.sim
localhost:~ user$ perl greeting.pl
Hello to Dan in Detroit!
localhost:~ user$
```

You can find this program in the **`examples/`** directory, named **`greeting.sim`**.

Handling Command-line Arguments

As an example of command-line handling, let's modify the "average" program we wrote in *Variables and Interpolation* to accept a variable list of three numbers. First, we'll write a subroutine to handle our command-line arguments, and another to display usage information and exit:

```
<subroutine name="usage">
     print "$ARGV0 NUMBER NUMBER NUMBER"
     exit 1
</subroutine>

<subroutine name="handle_commandline">
     if ARGV1 exists
          value1 equals $ARGV1
     else
          usage
     end
     if ARGV2 exists
          value2 equals $ARGV2
     else
          usage
     end
     if ARGV3 exists
          value3 equals $ARGV3
     else
          usage
     end
</subroutine>
```

This looks for three command-line arguments, and if they aren't found, the program will display usage information and exit with an error. Now that we've got our command-line handled, let's do what we came here to do: get an average of three numbers and display it:

```
<subroutine name="main">
     global value1 equals 0
     global value2 equals 0
     variable value3 equals 0
     handle_commandline
     global average
     average equals "($value1+$value2+$value3)/3"
     print "The average of $value1, $value2, and $value3 is $average"
     exit
</subroutine>
```

You can find this program in the **`examples/`** directory, named **`arguments.sim`**.

A Fibonacci Generator in SIMPLE

As an example, let's write a Fibonacci Sequence generator in SIMPLE. The formula to calculate the sequence is, well, simple: the first two iterations of the sequence are 0 and 1. Every iteration after that is the product of the two previous iterations, or:

$$F_0 = 0, F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

First, we need to create some variables. These will keeps track of the last two values, so the current iteration can be calculated; one variable (**`counter`**) will also be used to keep track of how many iterations we've calculated. We'll start our counter at "3" because we've already calculated the first three iterations of the sequence. We'll also create a variable to keep track of how many iterations we're going to generate (**`maximum`**); by default, we'll set this to "10" so we calculate the first 10 iterations of the sequence. After that, well display the first three iterations:

```
global stack1 equals 0
global stack2 equals 1
global stack3 equals 1
global counter equals 3
global maximum equals 10

print "1) 0"
print "2) 1"
print "3) 1"
```

Now, let's write a subroutine to calculate the next iteration of the sequence, and a subroutine to display the current iteration:

```
<subroutine name="calculate">
     <!-- Shift the variables "down" -->
     stack1 equals $stack2
     stack2 equals $stack3
     <!-- Calculate the next iteration -->
     stack3 equals "$stack1+$stack2"
     <!-- Increment the counter -->
     counter equals "$counter+1"
</subroutine>

<subroutine name="display">
     print "$counter) $stack3"
</subroutine>
```

With that out of the way, let's write a while loop to run the calculate subroutine the desired number of times:

```
while $counter less than or equals $maximum
     calculate
     display
break
```

Now, we put it all together: our main code will go in the **`main`** subroutine while our subroutines will be placed before the **`main`** subroutine:

```
<subroutine name="calculate">
     stack1 equals $stack2
     stack2 equals $stack3
     stack3 equals "$stack1+$stack2"
     counter equals "$counter+1"
</subroutine>

<subroutine name="display">
     print "$counter) $stack3"
</subroutine>

<subroutine name="main">
     global stack1 equals 0
     global stack2 equals 1
     global stack3 equals 1
     global counter equals 3
     global maximum equals 10

     print "1) 0"
     print "2) 1"
     print "3) 1"

     while $counter less than or equals $maximum
          calculate
          display
     break
</subroutine>
```

Save this to a file named **`fibonacci.sim`**, compile and run it:

```
localhost:~ user$ perl sim.pl -o fibonacci.pl fibonacci.sim
localhost:~ user$ perl fibonacci.pl
1) 0
2) 1
3) 1
4) 2
5) 3
6) 5
7) 8
8) 13
9) 21
10) 34
localhost:~ user$
```

You can find this program in the **`examples/`** directory, named **`fibonacci.sim`**.

Advanced Topics

Note that it's been said previously that **`if`** and **`while`** statements can't be nested. While that is true, there are ways around this limitation. Each **`subroutine`** can have their own, un-nested **`if`** and **`while`** statements; so, you can have a **`if`** or **`while`** block that calls multiple subroutines, each with their own **`if`** and **`while`** statements.

Here's an example. Let's write a program with a while loop that runs two other loops (contained in subroutines):

```
<subroutine name="first_loop">
      local counter equals 1
      prints "First Loop: "
      while $counter less than or equals 4
            prints "$counter "
            counter equals "$counter+1"
      break
      print "!"
</subroutine>

<subroutine name="second_loop">
      local counter equals 1
      prints "Second Loop: "
      while $counter less than or equals 4
            prints "$counter "
            counter equals "$counter+1"
      break
      print "!"
</subroutine>

<subroutine name="main">
      global loops equals 1
      while $loops less than or equals 4
            first_loop
            second_loop
            loops equals "$loops+1"
      break
</subroutine>
```

Save this to a file name **`two_loops.sim`**. Then compile and run it:

```
localhost:~ user$ perl sim.pl -o two_loops.pl two_loops.sim
localhost:~ user$ perl two_loops.pl
First Loop: 1234!
Second Loop: 1234!
First Loop: 1234!
Second Loop: 1234!
First Loop: 1234!
Second Loop: 1234!
First Loop: 1234!
Second Loop: 1234!
localhost:~ user$
```

You can find this program in the **`examples/`** folder, named **`two_loops.sim`**.

Subroutines can return different values, depending on their input or code. Using our Fibonacci Sequence example, let's write a subroutine that calculates a given iteration of the sequence and returns it:

```
<subroutine name="fibonacci" arguments="iteration">

      <!-- First, let's create some local variables -->

      local s1 equals 0
      local s2 equals 1
      local s3 equals 1
      local counter equals 3
      local result equals 0

      <!-- Since we already know the first three iterations (0,1,1),
            let's return those iterations without calculating them -->

      if $iteration equals 1
            return 0
      end

      if $iteration equals 2
            return 1
      end

      if $iteration equals 3
            return 1
      end

      <!-- Now we will generate the sequence up to the desired iteration -->

      while $counter less than $iteration
            counter equals "$counter+1"
            s1 equals $s2
            s2 equals $s3
            s3 equals "$s1+$s2"
      break

      <!-- Now, let's return our result! -->

      return $s3

</subroutine>
```

Using our new subroutine is easy! Just call the sub, using the iteration you want to find as the first argument. For example, to find the 50th iteration of the sequence, you'd use:

```
<subroutine name="main">

      <!-- Calculate and display the 50th iteration of the sequence -->

      global fiftieth_iteration
      fiftieth_iteration equals fibonacci 50
      print $fiftieth_iteration

</subroutine>
```

Please note that this subroutine does *no* error checking. It will return incorrect values if the iteration requested is less than one, for example.

You can find this program in the **`examples/`** folder, named **`fibsub.sim`**.

Variable names can also be contained and referenced in variables; that is, variable assignment commands have their input interpolated. For an example, let's write a program where we change a variable's value by referencing it with a variable:

```
<subroutine name="main">

     <!-- Create a variable, and set its value to a short phrase -->
     global the_target_variable equals "this is not the target variable"

     <!-- Create a variable, and set its value to the target variable's name -->
     global target_variable_name equals "the_target_variable"

     <!-- Change the phrase in "the_target_variable" to a new value
          We use the name stored in "target_variable_name" -->
     $target_variable_name equals "this is the target variable"

     <!-- Print the phrase stored in "the_target_value"
     print $the_target_variable
</subroutine>
```

Save this to a file name **`variables.sim`**. Then compile and run it:

```
localhost:~ user$ perl sim.pl -o variables.pl variables.sim
localhost:~ user$ perl variables.pl
this is the target variable
localhost:~ user$
```

You can find this program in the **`examples/`** folder, named **`variables.sim`**.

By default, the "entry point" in a SIMPLE program (that is, the subroutine that is automatically executed when the program is ran) is **`main`**. However, using the **`-d`** option in the SIMPLE compiler can be used to define a different subroutine. Let's write a program that illustrates this:

```
<subroutine name="main">
     <!-- This will never be executed -->
     print "This is the main subroutine!"
</subroutine>

<subroutine name="other">
     print "This is the other subroutine!"
</subroutine>
```

Save this to a file name **`new_default.sim`**. Then compile and run it:

```
localhost:~ user$ perl sim.pl -d other -o new_default.pl new_default.sim
localhost:~ user$ perl new_default.pl
This is the other subroutine!
localhost:~ user$
```

You can find this program in the **`examples/`** folder, named **`new_default.sim`**.

You can use the **input** command to get user input from the command line. For example, let's write a program that gets three numbers from the user and calculates their average. Open a new text file, and type into it:

```
<subroutine name="main">
     global number1
     global number2
     global number3
     global average

     print "Program to calculate the average of three numbers"

     <!-- Get the first number -->
     prints "Input number 1: "
     input to number1

     <!-- Get the second number -->
     prints "Input number 2: "
     input to number2

     <!-- Get the third number -->
     prints "Input number 3: "
     input to number3

     <!-- Calculate the average -->
     average equals "($number1+$number2+$number3)/3"

     print "The average of $number1, $number2, and $number3 is $average"
</subroutine>
```

Save this to a file name **average.sim**. Then compile and run it:

```
localhost:~ user$ perl sim.pl -o average.pl average.sim
localhost:~ user$ perl average.pl
Program to calculate the average of three numbers
Input number 1: 2
Input number 2: 4
Input number 3: 6
The average of 2, 4, and 6 is 4
localhost:~ user$
```

You can find this program in the **examples/** folder, named **average.sim**.

SIMPLE also features file input-output features (file I/O). Let's write a program that shows off these features. First, we'll create a file and write some text to it, copy it, and rename the copy:

```
<subroutine name="main">
     <!-- Create a file, and write some text to it -->
     write "This is a text file" to "myfile.txt"

     <!-- Copy the file to a new file named "copy.txt" -->
     copy "myfile.txt" to "copy.txt"

     <!-- Move "copy.txt" to a new file named "copycopy.txt" –>
     move "copy.txt" to "copycopy.txt"

     <!-- Load the copied file into memory and print the contents -->
     global contents
     contents equals read "copycopy.txt"
     print $contents
</subroutine>
```

Save this to a file name **`fileio.sim`**. Then compile and run it:

```
localhost:~ user$ perl sim.pl -o fileio.pl fileio.sim
localhost:~ user$ perl fileio.pl
This is a text file
localhost:~ user$
```

You can find this program in the **`examples/`** folder, named **`fileio.sim`**.

For assistance in debugging, there are several commandline arguments you can use. For example, let's use the following code (this code can be found in the **`examples/`** folder, named **`helloworld.sim`**):

```
<subroutine name="main">
      global greeting
      greeting equals "Hello world!"
      print $greeting
</subroutine>
```

We can compile this to Perl and strip the executor code , using the **`--executor`** option to strip executor code and the **`--stdout`** option to print to STDOUT, like so:

```
localhost:~ user$ perl sim.pl --stdout --executor examples/helloworld.sim
create_scalar("greeting","");
if(scalar_exists(i('greeting'))){
      change_scalar_value(i('greeting'),i('Hello world!'));
} else {
      print "Error in 'main' on line 2: Variable 'greeting' doesn't exist.\n";
      exit 1;
}
print i('$greeting')."\n";
END_BLOCK_MAIN:
destroy_locals();
localhost:~ user$
```

This is what the given SIMPLE code compiles into. Using the **`--include`** option, you can inject the original source code into the compiled code, in the form of comments:

```
localhost:~ user$ perl sim.pl --stdout –executor --include examples/helloworld.sim
#  'main' line 1: variable greeting
create_scalar("greeting","");
#  'main' line 2: greeting equals "Hello world!"
if(scalar_exists(i('greeting'))){
      change_scalar_value(i('greeting'),i('Hello world!'));
} else {
      print "Error in 'main' on line 2: Variable 'greeting' doesn't exist.\n";
      exit 1;
}
#  'main' line 3: print $greeting
print i('$greeting')."\n";
END_BLOCK_MAIN:
destroy_locals();
localhost:~ user$
```

SIMPLE Commands

Miscellaneous Commands

print

Syntax	`print "Print this text"`
Arguments	1 (the text to print)
Description	Prints text to the console, followed by a newline.

---

prints

Syntax	`prints "Print this text"`
Arguments	1 (the text to print)
Description	Prints text to the console.

---

write

Syntax	`write "content" to "filename.txt"` `write $variable to "filename.txt"`
Arguments	3 (what to write to file, "to", target filename)
Description	Writes content to a filename.

---

append

Syntax	`append "content" to "filename.txt"` `append $variable to "filename.txt"`
Arguments	3 (what to append to file, "to", target filename)
Description	Writes content to the end of an existing file. If the file does not exist, it will be created.

---

delete

Syntax	`delete "filename.txt"`
Arguments	1 (the filename to delete)
Description	Deletes a file.

---

split

Syntax	`split "this is my string" with " " to mysubroutine`
Arguments	5 (the string to split, "with", delimiter, "to", subroutine name)
Description	Splits a string using a specified delimiter. For every token found, a user-supplied subroutine is called. A local variable (**`result`**) is created each time the subroutine is executed with the token's contents.

---

input

Syntax	`input to myvariable`
Arguments	2 ("to", variable)
Description	Gets user input from the command line and stores it in a variable.

---

copy

Syntax	`copy myfile to newfile`
Arguments	3 (filename, "to", filename)
Description	Copies a file.

---

move

Syntax	`move myfile to newfile`
Arguments	3 (filename, "to", filename)
Description	Moves a file from one place to another.

Variable Commands

global

Syntax	`global myvar` `global othervar equals "value"`
Arguments	1 (variable name) or 3 (variable name, "equals", variable value)
Description	Creates a global variable, with the option of setting the variable's initial value; the initial value *cannot* be the return value of a subroutine.

---

local

Syntax	`local myvar` `local othervar equals "value"`
Arguments	1 (variable name) or 3 (variable name, "equals", variable value)
Description	Creates a local variable, with the option of setting the variable's initial value; the initial value *cannot* be the return value of a subroutine. This variable will be destroyed at the completion of the current code block. Local variables can have the same name as a variable created with the variable command; this means that local variables will not overwrite existing global variables. During interpolation, the value of a local variable will be used before a global variable.

Variable Assignment Commands

lowercase

Syntax	`myvar equals lowercase "MY TEXT"`
Arguments	1 (the text to convert to lowercase)
Description	Converts the input text into lowercase, and stores it in the variable.

---

uppercase

Syntax	`myvar equals uppercase "my text"`
Arguments	1 (the text to convert to uppercase)
Description	Converts the input text into uppercase, and stores it in the variable.

---

read

Syntax	`myvar equals read "filename.txt"`
Arguments	1 (the complete filename of the file to read)
Description	Opens a file, reads it, and stores the contents in the variable.

---

binread

Syntax	`myvar equals binread "filename.dat"`
Arguments	1 (the complete filename of the file to read)
Description	Opens a file in "binmode" (that is, preserving binary data), reads it, and stores the contents in the variable.

---

ascii

Syntax	`myvar equals ascii "="`
Arguments	1 (a single character)
Description	Converts a character into an ASCII code, and stores the contents in the variable.

---

character

Syntax	`myvar equals character 61`
Arguments	1 (ASCII code)
Description	Converts an ASCII code into a character, and stores the contents in the variable.

Flow Control Commands

if

Syntax

```
if var1 exists
if $var1 greater than $var2
if $var1 greater than or equals $var2
if $var1 less than $var2
if $var1 less than or equals $var2
if $var1 equals $var2
if $var1 contains $var2
if $var1 is not $var2
if $var1 is a number
if $var2 is a string
```

Arguments

2 (variable name, "exists), 3 (data, "equals", data), 4 (data, "is" or "greater" or "less" or "a", "than" or "not", data or "number" or "string"), or 6 (data, "greater" or "less", "than", "or', "equals", data)

Description

Begins an "if" block. This value will return true (and then execute the code in the "if" block) if the appropriate condition is met. The 2 argument option requires a variable name (without the **$**! We're referencing the variable itself and not its value), and will return false if a variable with that name is not found. The 3 argument option requires a data value, the word "equals" and another data value, or a data value, the word "contains", and another data value; it will return true if the data values are identical (in the first case) or if the the first data value contains the second. The 4 argument option requires a data value, either "greater" or "less", "than", and another data value; it will return true if the first data value is greater than or less than the second data value. The 6 argument option requires a data value, either "greater" or "less", "than", "or", "equals", and a second data value; it will return true if the first value is greater than or less than or equal to the second value. Code following this command, until the following "end" command (or an "else" command, if branching), will only be executed if the statement returns true.

---

end

Syntax

```
end
```

Arguments

0

Description

Ends an "if" block. If there is no matching "if" block, the compiler will throw an error.

---

else

Syntax

```
else
```

Arguments

0

Description

Branches an "if" statement; the code following (until the **end**) will be executed if the "if" statement returns false.

while

Syntax

```
while var1 exists
while $var1 greater than $var2
while $var1 greater than or equals $var2
while $var1 less than $var2
while $var1 less than or equals $var2
while $var1 equals $var2
while $var1 contains $var2
while $var1 is not $var2
```

Arguments

2 (variable name, "exists"), 3 (data, "equals", data), 4 (data, "is" or "greater" or "less", "than" or "not", data), or 6 (data, "greater" or "less", "than", "or", "equals", data)

Description

Begins a "while" block. This value will return true (and then execute the code in the "while" block) if the appropriate condition is met. The 2 argument option requires a variable name (without the **$**! We're referencing the variable itself and not its value), and will return false if a variable with that name is not found. The 3 argument option requires a data value, the word "equals" and another data value, or a data value, the word "contains", and another data value; it will return true if the data values are identical (in the first case) or if the the first data value contains the second. The 4 argument option requires a data value, either "greater" or "less", "than", and another data value; it will return true if the first data value is greater than or less than the second data value. The 6 argument option requires a data value, either "greater" or "less", "than", "or", "equals", and a second data value; it will return true if the first value is greater than or less than or equal to the second value. Code following this command, until the following "break" command, will loop as long as the statement returns true.

---

break

Syntax

```
break
```

Arguments

0

Description

Ends a "while" block. If there is no matching "while" block, the compiler will throw an error.

---

exit

Syntax

```
exit
exit 1
```

Arguments

1 optional (the error code to exit with)

Description

Exits a program, using an optional error code.

---

return

Syntax	`return 1`
	`return "String return value"`
	`return`

Arguments	0 or 1 (the value for a subroutine to return if called in variable assignment)
Description	If the subroutine is currently being called via variable assignment, this sets the value that is returned and stored in the variable, then automatically exits the subroutine. If called without an argument, this exits the subroutine without returning a value.

License