

Introduction to the SAM User Language

National Renewable Energy Laboratory

September 22, 2009

Abstract

The SAM User Language (SamUL) is a built-in scripting or programming language that allows a user to automate tasks and perform more complex analyses. This guide assumes some rudimentary facility with programming concepts and familiarity with the SAM interface, capabilities, and general work flow.

The Solar Advisor Model (SAM) provides a consistent framework for analyzing and comparing power system performance and costs across the range of solar technologies and markets, from residential photovoltaic systems to utility scale concentrating solar power plants.

Solar Advisor is based on an hourly simulation engine that interacts with performance, cost, and finance models to calculate energy output, costs, and cash flows, including the effect of incentives.

Contents

1	Introduction	4
1.1	Why use SamUL?	4
1.2	Entering a SamUL Script	4
1.3	Hello world!	5
1.4	Why SamUL instead of VBA?	5
2	Data Variables	5
2.1	General Syntax	5
2.2	Variables	6
2.3	Arithmetic	7
2.4	Simple Input and Output	7
2.5	Data Types and Conversion	8
2.6	Special Characters	9
3	Flow Control	9
3.1	Comparison Operators	9
3.2	Branching	10
3.2.1	if Statements	10
3.2.2	else Construct	11
3.2.3	Multiple if Tests	11
3.2.4	Single line ifs	11
3.3	Looping	12
3.3.1	while Loops	12
3.3.2	Counter-driven Loops	12
3.3.3	for Loops	13
3.3.4	Loop Control Statements	13
3.4	Quitting	14
4	Arrays of Data	14
4.1	Initializing and Indexing	14
4.2	Array Length	15
4.3	Processing Arrays	15
4.4	Multidimensional Arrays	16
4.5	Managing Array Storage	16
5	Functions Calls	17
5.1	User Functions	17
5.1.1	Definition	17
5.1.2	Returning a Value	18
5.1.3	Parameters	18
5.1.4	Variable Scope	19
5.2	Built-in SamUL Functions	20
6	Input, Output, and System Access	20
6.1	Working with Text Files	20

6.2	File System Functions	21
6.3	Standard Dialogs	22
6.4	Calling Other Programs	22
7	Interfacing With SAM Analyses	23
7.1	Changing a Base Case Input	23
7.2	Simulating and Saving Output	23
7.3	Batching Weather Files	24

1 Introduction

1.1 Why use SamUL?

Suppose you are an energy analyst, and your client asks for a custom map of the United States showing the levelized cost of energy (LCOE) at several hundred locations in the country for a specific photovoltaic system. Since you are familiar with the Solar Advisor Model, and have access to weather data for all the requested locations, the task falls to you to generate the LCOE values for the custom map. With the PV system specifications, you succeed in setting up a "base case" simulation in SAM for one location.

You could run a SAM analysis for each weather file individually, recording the results in a spreadsheet. You could even set up many locations in a parametric simulation. Either way would be tedious and error prone at best for several hundred iterations. Even worse, when your client decides to change even just one specification of the system, you would have to start all over.

As others have done in the past, you could extract the generated PV simulator setup files, and write a program in C or Excel/VBAScript to call the TRNSYS simulation engine that sits behind SAM to loop over the various weather files. This too proves troublesome, because if any specifications change, you would have to edit the complicated TRNSYS input files by hand. Furthermore, processing the large amounts of data in the system performance output files would at best be an unpleasant chore. The worst part is that this "solution" does not even give you the LCOE at the end, because that is calculated inside the SAM financial model not TRNSYS!

The SAM User Language was designed to solve these problems. It lets you perform this otherwise painful LCOE task in no more than 10 lines of code. When a system specification changes, or new weather data is available, you only need to re-run the script after modifying the appropriate inputs in SAM. Even better, there is no need for any additional software or expertise. This guide will help you learn the basics of SamUL programming in the hope that it will save you significant time and effort in the future.

1.2 Entering a SamUL Script

SamUL is included by default in every installation of the Solar Advisor Model, and is accessed from the 'Developer' menu in SAM.

Since SamUL files are saved as part of a SAM project, you must start a new project or open one of the included sample files to begin. Once your project is open, you can create a new script from "Developer:New SamUL Script". A tab will open showing an empty text editor and a toolbar with a few buttons.

The first button will interpret and run the SamUL script that is in the text editor. If there are any syntax errors, SAM will display an error message box and force you to correct any errors before it runs the script. If you have typed something incorrectly, the first error location displayed is nearest the point in the source code where the typo occurred.

If your code is syntactically correct, SAM will run it and display a console dialog to capture any output that your script may create.

1.3 Hello world!

As with any new language, the traditional first program is to print the words "Hello, world!" on the screen, and the SamUL version is listed below.

```
out( "Hello, world!\n" )
```

Notable features:

1. The `out` command generates output from the script
2. The text to output is enclosed in double-quotes
3. To move to a new line, the symbol `\n` is used

To run Hello world, type it into a new SamUL script, and press the 'Run' button on the toolbar. Now we will learn some more about variables and program control in SamUL.

1.4 Why SamUL instead of VBA?

Since SamUL is so similar in functionality and syntax to Visual Basic, you might wonder why we didn't simply put a VBA engine behind SAM. Visual Basic for Applications (VBA) is a closed source Microsoft product that is very tightly integrated into the Office suite of applications, and while there are some freely available interpreters for subsets of VB-like languages, we decided that they would not suit our purposes well enough.

SamUL is close enough to VBA in general syntax and program structure to make it easily understandable by people familiar with VBA, but with some differences and additional functionality to alleviate some of the more annoying aspects of VBA programming. By developing our own script engine, we have been able to integrate it very tightly into the SAM environment for maximum ease of use.

Some notable points of departure from VBA include:

1. 'for' loop syntax follows the C / Perl convention
2. Array arithmetic is automatically performed
3. 'elseif' statement formatting is similar to PHP
4. No distinction between functions and procedures

2 Data Variables

2.1 General Syntax

SamUL is a programming language that is similar in style to Visual Basic. Each program statement is generally placed on a line by itself, and the end-of-line marks the end of the statement. However, unlike Visual Basic, there is no facility to split a long statement across multiple lines.

Blank lines may be inserted between statements. While they have no meaning, they can help make a script easier to read. Spaces can also be added or removed nearly anywhere, except for in the middle of a word. The following statements all have the same meaning.

```
out("Hello 1\n")
    out ("Hello 2\n")
out ( "Hello 3\n" )
```

Comments are lines in the program code that are ignored by SamUL. They serve as a form of documentation, and can help other people (and you!) more easily understand what the script does. Comments begin with the single-quote ' character, and continue to the end of the line.

```
' this program creates a greeting
out( "Hello, world!\n" ) ' display the greeting to the user
```

2.2 Variables

Variables store information while your script is running. SamUL variables share many characteristics with other computer languages.

1. Each variable stores a single value
2. Variables do not need to be "declared" in advance of being used
3. There is no distinction between variables that store text and variables that store numbers

Variable names may contain letters, digit, and the underscore symbol. A limitation is that variables cannot start with a digit. Unlike some languages like C and Perl, SamUL does not distinguish between upper and lower case letters in a variable (or subroutine) name. As a result, the name `myData` is the same as `MYdata`.

Values are assigned to variables using the equal sign =. Some examples are below.

```
Num_Modules = 10
ArrayPowerWatts = 4k
Tilt = 18.2
system_name = "Super PV System"
Cost = "unknown"
COST = 1e6
cost = 1M
```

A second assignment to a variable erases its previous value. As shown above, decimal numbers can be written using scientific notation or engineering suffixes. The last two assignments to `Cost` are the same value. Recognized suffixes are listed in the table below. Note that suffixes are case-sensitive.

Name	Suffix	Multiplier
Tera	T	1e12
Giga	G	1e9
Mega	M	1e6
Kilo	k	1e3
Milli	m	1e-3
Micro	u	1e-6
Nano	n	1e-9
Pico	p	1e-12
Femto	f	1e-15
Atto	a	1e-18

Table 1: Recognized Engineering Suffixes

2.3 Arithmetic

SamUL supports the four basic operations $+$, $-$, $*$, and $/$. The usual algebraic precedence rules are followed, so that multiplications are performed before additions. Parentheses are also understood and can be used to change the default order of operations.

More complicated operations like raising to a power and performing modulus arithmetic are possible using built-in function calls in the standard SamUL library.

Examples of arithmetic operations:

```
battery_cost = cost_per_kwh * battery_capacity

' multiplication takes precedence
degraded_output = degraded_output - degraded_output * 0.1

' use parentheses to subtract before multiplication
cash_amount = total_cost * ( 1 - debt_fraction/100.0 )
```

2.4 Simple Input and Output

You can use the built-in `out` and `outln` functions to write data to the console window. The difference is that `outln` automatically appends a newline character to the output. To output multiple text strings or variables, use the $+$ operator, or separate them with a comma.

```
array_power = 4.3k
array_eff = 0.11
outln("Array power is " + array_power + " Watts.")
outln("It is " + (array_eff*100) + " percent efficient.")
outln("It is ", array_eff*100, " percent efficient.") ' same as above
```

The console output generated is:

```
Array power is 4300 Watts.  
It is 11 percent efficient.
```

Use the `in` function to read input from the user. You can optionally pass a message to `in` to display to the user when the input popup appears. The user can enter either numbers or text, and SamUL will perform any type conversions if needed (and if possible).

```
cost_per_watt = in("Enter cost per watt:") ' Show a message. in() also is fine.  
notice( "Total cost is: " + cost_per_watt * array_power + " dollars")
```

The `notice` function works like `out`, except that it displays a popup message box on the computer screen.

2.5 Data Types and Conversion

SamUL supports four basic types of data, although most conversions between types happen automatically.

Type	Conversion Function	Valid Values
Integer Number	<code>integer()</code>	+/- approx. 2 billion
Double-precision Decimal Number	<code>double()</code>	1e-308 to 1e308, with infinity
Boolean	<code>boolean()</code>	true or false (1 or 0)
Text Strings	<code>string()</code>	Any length text string

Table 2: Intrinsic SamUL Data Types

Sometimes you have two numbers in text strings that you would like to multiply. This can happen if you read data in from a text file on the computer, for example. Since it does not make sense to try to multiply text strings, you need to first convert the strings to numbers. To convert a variable to a double-precision decimal number, use the `double` function, as below.

```
a = "3.5"  
b = "-2"  
c1 = a*b ' this will cause an error when you click 'Run'  
c2 = Double(a) * Double(b) ' this will assign c2 the number value of -7
```

You can also use `integer` to convert a string to an integer or truncate a decimal number, or the `string` function to explicitly convert a number to a string variable.

If you need to find out what type a variable currently has, use the `typeof` function to get a description.

```
a = 3.5  
b = -2  
c1 = a+b ' this will set c1 to -1.5  
c2 = String( Integer(a) ) + String( b ) ' c2 set to text "3-2"
```

```
outln( typeof(a) ) ' will display "double"  
outln( typeof(c2) ) ' will display "string"
```


2.6 Special Characters

Text data can contain special characters to denote tabs, line endings, and other useful elements that are not part of the normal alphabet. These are inserted into quoted text strings with *escape sequences*, which begin with the `\` character.

Escape Sequence	Meaning
<code>\n</code>	New line
<code>\t</code>	Tab character
<code>\r</code>	Carriage return
<code>\"</code>	Double quote
<code>\\</code>	Backslash character

Table 3: Text String Escape Sequences

So, to print the text "Hi, tabbed world!", or assign `c:\Windows\notepad.exe`, you would have to write:

```
outln("\nHi,\t\ttabbed world!\n")
program = "c:\\Windows\\notepad.exe"
```

3 Flow Control

3.1 Comparison Operators

SamUL supports many ways of comparing data. These types of tests can control the program flow with branching and looping constructs that we will discuss later.

There are six standard comparison operators that can be used on most types of data. For text strings, "less than" and "greater than" are with respect to alphabetical order.

Comparison	Operator
Equal	<code>==</code>
Not Equal	<code>!=</code>
Less Than	<code><</code>
Less Than or Equal	<code><=</code>
Greater Than	<code>></code>
Greater Than or Equal	<code>>=</code>

Table 4: Comparison Operators

Examples of comparisons:

```
divisor != 0
state == "oregon"
error <= -0.003
"pv" > "csp"
```

Single comparisons can be combined by *boolean* operators into more complicated tests.

1. The **not** operator yields true when the test is false. It is placed before the test whose result is to be notted.
Example: `not (divisor == 0)`
2. The **and** operator yields true only if both tests are true.
Example: `divisor != 0 and dividend > 1`
3. The **or** operator yields true if either test is true.
Example: `state == "oregon" or state == "colorado"`

The boolean operators can be combined to make even more complex tests. The operators are listed above in order of highest precedence to lowest. If you are unsure of which test will be evaluated first, use parentheses to group tests. Note that the following statements have very different meanings.

```
state_count > 0 and state_abbrev == "CA" or state_abbrev == "OR"
state_count > 0 and (state_abbrev == "CA" or state_abbrev == "OR")
```

3.2 Branching

Using the comparison and boolean operators to define tests, you can control whether a section of code in your script will be executed or not. Therefore, the script can make decisions depending on different circumstances and user inputs.

3.2.1 if Statements

The simplest branching construct is the **if** statement. For example:

```
if ( tilt < 0.0 )
    outln("Error: tilt angle must be 0 or greater")
end
```

Note the following characteristics of the **if** statement:

1. The test condition is placed in parentheses after the **if** keyword.
2. The following program lines include the statements to execute when the **if** test succeeds.
3. For the sake of program readability, the statements inside the **if** are indented. The program will still be correct if they are not indented, but then the script becomes much harder to understand and debug.
4. The construct concludes with the **end** keyword.
5. When the **if** test fails, the program statements inside the **if-end** block are skipped.

3.2.2 else Construct

When you also have commands you wish to execute when the `if` test fails, use the `else` clause. For example:

```
if ( power > 0 )
    energy = power * time
    operating_cost = energy * energy_cost
else
    outln("Error, no power was generated.")
    energy = -1
    operating_cost = -1
end
```

3.2.3 Multiple if Tests

Sometimes you wish to test many conditions in a sequence, and take appropriate action depending on which test is successful. In this situation, use the `elseif` clause. Be careful to spell it as a single word, as both `else if` and `elseif` can be syntactically correct, but have different meanings.

```
if ( angle >= 0 and angle < 90 )
    text = "first quadrant"
elseif ( angle >= 90 and angle < 180 )
    text = "second quadrant"
elseif ( angle >= 180 and angle < 270 )
    text = "third quadrant"
else
    text = "fourth quadrant"
end
```

You do not need to end a sequence of `elseif` statements with the `else` clause, although in most cases it is appropriate so that every situation can be handled. You can also nest `if` constructs if needed. Again, we recommend indenting each "level" of nesting to improve your script's readability. For example:

```
if ( angle >= 0 and angle < 90 )
    if ( print_value == true )
        outln( "first quadrant: " + angle )
    else
        outln( "first quadrant" )
    end
end
```

3.2.4 Single line ifs

Sometimes you only want to take a single action when an `if` statement succeeds. To reduce the amount of code you must type, SamUL accepts single line `if` statements, as shown below.

```
if ( azimuth < 0 ) outln( "Warning: azimuth < 0, continuing..." )
```

```
if ( tilt > 90 ) tilt = 90    ' set maximum tilt value
```

You can also use an **else** statement on single line **if**. Like the **if**, it only accepts one program statement, and must be typed on the same program line. Example:

```
if ( value > average ) outln("Above average") else outln("Not above average")
```

3.3 Looping

A loop is a way of repeating the same commands over and over. You may need to process each line of a file in the same way, or sort a list of names. To achieve such tasks, SamUL provides two types of loop constructs, the **while** and **for** loops.

Like **if** statements, loops contain a "body" of program statements followed by the **end** keyword to denote where the loop construct ends.

3.3.1 while Loops

The **while** loop is the simplest loop. It repeats one or more program statements as long as a logical test holds true. When the test fails, the loop ends, and the program continues execution of the statements following the loop construct. For example:

```
while ( done == false )  
    ' process some data  
    ' check if we are finished and update the 'done' variable  
end
```

The test in a **while** loop is checked before the body of the loop is entered for the first time. In the example above, we must set the variable **done** to **false** before the loop, because otherwise no data processing would occur. After each iteration ends, the test is checked again to determine whether to continue the loop or not.

3.3.2 Counter-driven Loops

Counter-driven loops are useful when you want to run a sequence of commands for a certain number of times. As an example, you may wish to display only the first 10 lines in a text file.

There are four basic parts of implementing a counter-driven loop:

1. Initialize a counter variable before the loop begins.
2. Test to see if the counter variable has reached a set maximum value.
3. Execute the program statements in the loop, if the counter has not reached the maximum value.
4. Increment the counter by some value.

For example, we can implement a counter-driven loop using the **while** construct:

```
i = 0      ' use i as counter variable
while (i < 10)
    outln( "value of i is " + i )
    i = i + 1
end
```

3.3.3 for Loops

The **for** loop provides a streamlined way to write a counter-driven loop. It combines the counter initialization, test, and increment statements into a single line. The script below produces exactly the same effect as the **while** loop example above.

```
for ( i = 0; i < 10; i = i+1 )
    outln( "value of i is " + i )
end
```

The three loop control statements are separated by semicolons in the **for** loop statement. The initialization statement (first) is run only once before the loop starts. The test statement (second) is run before entering an iteration of the loop body. Finally, the increment statement is run after each completed iteration, and before the test is rechecked. Note that you can use any assignment or calculation in the increment statement.

Just like the **if** statement, SamUL allows **for** loops that contain only one program statement in the body to be written on one line. For example:

```
for ( val=57; val > 1; val = val / 2 ) outln("Value is " + val )
```

3.3.4 Loop Control Statements

In some cases you may want to end a loop prematurely. Suppose under normal conditions, you would iterate 10 times, but because of some rare circumstance, you must break the loop's normal path of execution after the third iteration. To do this, use the **break** statement.

```
value = double( in("Enter a starting value") )
for ( i=0; i<10; i=i+1 )
    outln("Value is " + value )
    if (value < 0)
        break
    end
    value = value / 3.0
end
```

In another situation, you may not want to altogether break the loop, but skip the rest of program statements left in the current iteration. For example, you may be processing a list of files, but each one is only processed if it starts with a specific line. The **continue** keyword provides this functionality.

```

for ( i=0; i<file_count; i=i+1 )
    file_header_ok = false

    ' check if whether current file has the correct header

    if (file_header_ok == false)
        continue
    end

    ' process this file
end

```

The **break** and **continue** statements can be used with both **for** and **while** loops. If you have nested loops, the statements will act in relation to the nearest loop structure. In other words, a **break** statement in the body of the inner-most loop will only break the execution of the inner-most loop.

3.4 Quitting

SamUL script execution normally ends when there are no more program statements to run at the end of the program. However, sometimes you may need to halt early, if the user chooses not to continue an operation, for example.

The **exit** statement will end the SamUL script immediately. For example:

```

if ( yesno("Do you want to quit?") == true )
    outln("Aborted.")
    exit
end

```

The **yesno** function call displays a message box on the user's screen with yes and no buttons, showing the given message. It returns **true** if the user clicked yes, or **false** otherwise.

4 Arrays of Data

Often you need to store a list of related values. For example, you may need to refer to the price of energy in different years. Or you might have a table of state names and capital cities. In SamUL, you can use arrays to store these types of collections of data.

4.1 Initializing and Indexing

An *array* is simply a list of variables that are indexed by numbers. Each variable in the array is called an *element* of the array, and the position of the element within the array is called the element's *index*. The index of the first element in an array is always 0.

To access array elements, enclose the index number in square brackets immediately following the variable name. Unlike most computer languages, SamUL does not require you to declare or allocate space for the array data in advance.

```
names[0] = "Sean"
names[1] = "Walter"
names[2] = "Pam"
names[3] = "Claire"
names[4] = "Patrick"
```

```
outln( names[3] ) ' output is "Patrick"
my_index = 2
outln( names[my_index] ) ' output is "Pam"
```

You can also initialize a fixed array using the `array` command provided in SamUL. Simply separate each element with a comma. There is no limit to the number of elements you can pass to `array`.

```
names = array("Sean", "Walter", "Pam", "Claire", "Patrick")
outln( "First: " + names[0] )
outln( "All: " + names )
```

Note that calling the `typeof` function on an array variable will return "array" as the type description, not the type of the elements. This is because SamUL is not strict about the types of variables stored in an array, and does not require all elements to be of the same type.

4.2 Array Length

Sometimes you do not know in advance how many elements are in an array. This can happen if you are reading a list of numbers from a text file, storing each as an element in an array. After the all the data has been read, you can use the `length` function to determine how many elements the array contains.

```
count = length( names )
```

4.3 Processing Arrays

Arrays and loops naturally go together, since frequently you may want to perform the same operation on each element of an array. For example, you may want to find the total sum of an array of numbers.

```
numbers = array( 1, -3, 2.4, 9, 7, 22, -2.1, 5.8 )

count = length( numbers )
sum = 0
for (i=0; i<count; i=i+1)
    sum = sum + numbers[i]
```

end

The important feature of this code is that it will work regardless of how many elements are in the array.

4.4 Multidimensional Arrays

As previously noted, SamUL is not strict with the types of elements stored in an array. Therefore, a single array element can even be another array. This allows you to define matrices with both row and column indexes, and even three dimensional arrays.

To create a multi-dimensional array, simply separate the indices with commas between the square brackets. For example:

```
data[0,0] = 3
data[0,1] = -2
data[1,0] = 5
data[2,0] = 1
```

```
nrows = length(data) ' result is 4
ncols = length(data[0]) ' result is 2
```

```
row1 = data[0] ' extract the first row
```

```
x = row1[0] ' value is 3
y = row1[1] ' value is -2
```

4.5 Managing Array Storage

When you define an array, SamUL automatically allocates sufficient computer memory to store the elements. If you know in advance that your array will contain 100 elements, for example, it can be much faster to allocate the computer memory before filling the array with data. Use the `allocate` command to make space for 1 or 2 dimensional arrays.

```
data = allocate(3,2) ' a matrix with 3 rows and 2 columns
data[2,1] = 3
```

```
prices = allocate( 5 ) ' a simple 5 element array
```

As before, you can extend the array simply by using higher indexes. However, if you know in advance how many more elements you will be adding, it can be faster to use the `resize` command to reallocate computer memory to store the array. `resize` preserves any data in the array, or truncates data if the new size is smaller than the old size.

```
data = allocate(5)
outln( length(data) )
resize(data, 10)
```



```

outln( length(data) )

resize(data, 2, 4)
outln( length(data) )
outln( length( data[0] ) )

```

5 Functions Calls

It is usually good programming practice to split a larger program up into smaller sections, often called procedures, functions, or subroutines. A program may be easier to read and debug if it is not all thrown together, and you may have common blocks of code that appear several times in the program.

5.1 User Functions

A function is simply a named chunk of code that may be called from other parts of the script. It usually performs a well-defined operation on a set of variables, and it may return a computed value to the caller.

Functions can be written anywhere in your SAM script, including after they are called. If a function is never called by the program, it has no effect.

5.1.1 Definition

Consider the very simple procedure listed below.

```

function show_welcome()
    outln("Thank you for choosing SamUL.")
    outln("This text will only be displayed at the start of the script.")
end

```

Notable features:

1. Use the **function** keyword to define a new function.
2. The name immediately follows. Valid function names can have letters, digits, and underscores, but cannot start with a digit.
3. The empty parentheses after the name indicate that this function takes no parameters.
4. The **end** keyword closes the function definition.

To call the function from elsewhere in the code, simply write the function's name, followed by the parentheses.

```

' show a message to the user
show_welcome()

```

5.1.2 Returning a Value

A function is generally more useful if it can return information back to the program that called it. In this example, the function will not return unless the user enters "yes" or "no" into the input dialog.

```
function require_yes_or_no()
    while( true )
        answer = in("Destroy everything? Enter yes or no:")
        if (answer == "yes") return true
        if (answer == "no") return false
        outln("That was not an acceptable response.")
    end
end

' call the input function
result = require_yes_or_no() ' returns true or false
if ( not result )
    outln("user said no, phew!")
    exit
else
    outln("destroying everything...")
end
```

The important lesson here is that the main script does not worry about the details of how the user is questioned, and only knows that it will receive a **true** or **false** response. Also, the function can be reused in different parts of the program, and each time the user will be treated in a familiar way.

5.1.3 Parameters

In most cases, a function will accept arguments when it is called. That way, the function can change its behavior, or take different inputs in calculating a result. Analogous to mathematical functions, SamUL functions can take arguments to compute a result that can be returned. Arguments to a function are given names and are listed between the parentheses on the function definition line.

For example, consider a function to determine the minimum of two numbers:

```
function minimum(a, b)
    if (a < b) return a else return b
end

' call the function
count = 129
outln("Minimum: " + minimum( count, 77))
```

In SamUL, changing the value of a function's named arguments will modify the variable in the calling program. Instead of passing the actual value of a parameter `a`, SamUL always passes a *reference* to the variable in the original program. The reference is hidden from the user, so the variable acts just like any other variable inside the function.

Because arguments are passed by reference (as in Fortran, for example), a function can "return" more than one value. For example:

```
function sumdiffmult(s, d, a, b)
    s = a+b
    d = a-b
    return a*b
end
```

```
sum = -1
diff = -1
mult = sumdiffmult(sum, diff, 20, 7)
```

`outln("Sum: " + sum + " Diff: " + diff + " Mult: " + mult) ' will output 27, 13, and 140`

5.1.4 Variable Scope

Generally, variables used inside a function are considered "local", and cannot be accessed from the caller program. For example:

```
function triple(x)
    y = 3*x
end
```

```
triple( 4 )
outln( y ) ' this will fail because y is local to the triple function
```

As we have seen, we can write useful functions using arguments and return values to pass data into and out of functions. However, sometimes there are some many inputs to a function that it becomes very cumbersome to list them all as arguments. Alternatively, you might have some variables that are used throughout your program, or are considered reference values or constants. For these situations, you can define variables to be **global** in SamUL, and then they can be used inside functions and in the main program. For example:

```
global pi = 3.1415926
```

```
function circumference( r )
    return 2*pi*r
end
```

```
function deg2rad( x )
    return pi/180*x
end
```

```

outln( "PI: " + pi )
outln( "CIRC: " + circumference( 3 ) )
outln( "D2R: " + deg2rad( 180 ) )

```

Common programming advice is to minimize the number of global variables used in a program. Sometimes they are certainly necessary, but too many can lead to mistakes that are harder to debug and correct, and can reduce the readability and maintainability of your script.

5.2 Built-in SamUL Functions

Throughout this guide, we have made use of built-in functions like `in`, `outln`, and others. These functions are included with SamUL automatically, and called in exactly the same way as user functions. Like user functions, they can return values, and sometimes they modify the arguments sent to them. Refer to the "Standard Library Reference" at the end of this guide for documentation on each function's capabilities, parameters, and return values.

6 Input, Output, and System Access

SamUL provides a variety of standard library functions to work with files, directories, and interact with other programs. So far, we have used the `in`, `out`, and `outln` functions to accept user input and display program output in the runtime console window. Now we will learn about accessing files and other programs.

6.1 Working with Text Files

To write data to a text file, use the `writetextfile` function. `writetextfile` accepts any type of variable, but most frequently you will write text stored in a string variable. For example:

```

data = ""
for (i=0;i<10;i=i+1) data = data + "Text Data Line " + string(i) + "\n"
ok = writetextfile( "C:/test.txt", data )
if (not ok) outln("Error writing text file.")

```

Reading a text file is just as simple with the `readtextfile` function.

```

mytext = ""
if (not readtextfile( "C:/test.txt", mytext ))
    outln("could not read text file.")
else
    outln("text data:")
    out(mytext)
end

```

While these functions offer an easy way to read an entire text file, often it is useful to be able to access it line by line. SamUL provides the `readline` for this purpose. You need to provide a position variable that is set to the location in the text data to begin reading.

```
mytext = ""
readtextfile( "C:/test.txt", mytext )

line = ""
pos = 0      ' start at position 0
while ( readline( mytext, pos, line ) )
    outln( "My Text Line='" + line + "'" )
end
```

In the example above, `pos` is updated by `readline`, and the `line` variable is filled in with the current line of text. The `readline` function will return `true` as long as there are more lines to be read from the input text in `mytext`.

Another way to access individual lines of a text file uses the `split` function to return an array of text lines. For example:

```
mytext = ""
readtextfile( "C:/test.txt", mytext )
lines = split( mytext, "\n" )
outln("There are " + length(lines) + " lines of text in the file.")
if (length(lines) > 5) outln("Line 5: '", lines[5], "'")
```

6.2 File System Functions

Suppose you want to run SAM with many different weather files, and consequently need a list of all the files in a folder that have the `.tm2` extension. SamUL provides the `directorylist` function to help out in this situation. If you want to filter for multiple file extensions, separate them with commas.

```
file_names = directorylist( "C:/Windows", "dll" ) ' could also use "txt,dll"
outln("Found " + length(file_names) + " files that match.")
outln(unsplit(file_names, "\n"))
```

To list all the files in the given folder, leave the extension string empty or pass in `"*"`.

Sometimes you need to be able to quickly extract the file name from the full path, or vice versa. The functions `filenameonly` and `dirnameonly` extract the respective sections of the file name, returning the result.

To test whether a file or directory exist, use the `direxists` or `fileexists` functions. Examples:

```
path = "C:/SAM/2009.8.13/samsim.dll"
dir = dirnameonly( path )
name = filenameonly( path )
outln( "Path: " + path )
```

```
outln( "Name: " + name + " Exists? " + fileexists(path) )
outln( "Dir: " + dir + " Exists? " + direxists(dir))
```

6.3 Standard Dialogs

To facilitate writing more interactive scripts, SamUL includes various dialog functions. We have already used the `notice` and `yesno` functions in previous examples.

The `choosefile` function pops up a file selection dialog to the user, prompting them to select a file. `choosefile` will accept three optional parameters: the path of the initial directory to show in the dialog, a wildcard filter like `"*.txt"` to limit the types of files shown in the list, and a dialog caption to display on the window. Example:

```
file = choosefile("c:/SAM", "*.dll", "Choose a DLL file")
if (file == "")
    notice("You did not choose a file, quitting.")
    exit
else
    if ( not yesno("Do you want to load:\n\n" + file)) exit

    ' proceed to load .dll file
    outln("Loading " + file)
end
```

6.4 Calling Other Programs

Suppose you have a program on your computer that reads an input file, makes some complicated calculations, and writes an output file. For example, a program could read in some system specifications and calculate its heat loss coefficients that could be used in a SAM analysis.

There are two very similar ways to call external programs: the `system` and `shell` functions. They are identical except that `shell` pops up an interactive system command window and runs the program in it. Both functions will wait until the called program finishes before returning to SamUL, so that the program runs *synchronously*. Examples:

```
system("notepad.exe") ' run notepad and wait
shell("ipconfig /all > c:/test.txt")
output = ""
readtextfile("c:/test.txt", output)
outln(output)
```

Each program runs in a folder that the program refers to as the *working directory*. Sometimes you may need to switch the working directory to conveniently access other files, or to allow an external program to run correctly.

```
working_dir = cwd() ' get the current working directory
chdir( "C:/windows" ) ' change the working directory
```

```

outln("cwd=" + cwd() )
chdir(working_dir)      ' change it back to the original one
outln("cwd=" + cwd() )

```

7 Interfacing With SAM Analyses

All of the SamUL capabilities we've discussed so far would be of little interest if they did not allow for interaction and automation of SAM analyses. To this end, there is a set of included function calls that can set input variables, invoke a simulation, and retrieve output variables.

All of the SamUL function calls involve only "base case" analysis. That is, the built-in parametrics, sensitivity, optimization, and statistical simulation types that are controlled from the user interface are not accessible from SamUL. This is probably less of a hindrance than it sounds, as SamUL exists primarily to allow for specialized simulations that do not fall into one of those categories.

7.1 Changing a Base Case Input

You must first select a case in your SAM project to work on using the `setactivecase` function call. Once an "active" case has been chosen, you can change base case input values using the `setinput` function. All other inputs are unchanged from the values shown on the case page. If a variable affects other calculated inputs, they are automatically recalculated, and the updated values are shown on the case input page. For example, in a project with a case titled "Residential PV System":

```

setactivecase( "Residential PV System" )
setinput( "system.degradation", 12.5 )      ' set degradation to 12.5 percent

```

SamUL requires that you provide the internal names of variables to access them. These names can be accessed from the SamUL toolbar by way of their grouping and labels shown on the input page. Simply select the input or output variable from the hierarchical menu, and the internal name will be pasted into the SamUL script at the cursor position.

7.2 Simulating and Saving Output

To start a base case simulation, use the `simulate` function. It takes a boolean parameter to specify whether to save the hourly (8760) outputs from the simulation. After the simulation has finished, you can access the outputs using the `getoutput` function.

```

setactivecase( "Residential PV System" )
setinput( "system.degradation", 12.5 )      ' set degradation to 12.5 percent
simulate( false )                          ' do not save the hourly outputs this time
lcoe = getoutput("sv.lcoe_real")
notice( "LCOE = " + lcoe )

```

As with the input variables, the internal variable names of the available outputs are also accessible from the SamUL toolbar.

You can also save several outputs to a comma-separated value (CSV) file to work with in Excel or another program using the `writeresults` function. The outputs variables are passed to the function separated by commas in a single string, and each variable is dumped as a separate column in the CSV file.

```
setactivecase( "Residential PV System" )
setinput( "system.degradation", 12.5 )    ' set degradation to 12.5 percent
simulate( true )                          ' save the hourly outputs this time
writeresults( "c:/test.csv", "system.hourly.e_net,system.monthly.e_net,sv.lcoe_nom")
```

7.3 Batching Weather Files

Let's return to the original hypothetical example discussed in the introduction. You have directory of weather files, and for each one you are asked to calculate the hourly generation and LCOE for the system.

The code addresses this need, and makes use of many SamUL language capabilities and built-in function calls.

```
' Set the active case from the current ones
setactivecase("Simple PV System")

' Specify a directory to use for weather batching
dir = "c:/Documents and Settings/David Smith/Desktop/Weather Files"

' List all the files in a directory that have the extension "tm2"
file_list = DirectoryList(dir, "tm2")

' loop through all the files
count = length(file_list)
for (i=0;i<count;i=i+1)

    out("Weather (",(i+1)," of ",count,")="+FileNameOnly(file_list[i])+"\n")
    ' set the climate variable to the file name
    setinput("climate.location", file_list[i])

    ' run the base case
    simulate( true )

    ' make the output file name
    output_file = dir + "/output_" + FileNameOnly(file_list[i]) + ".csv"

    out("Writing Output File: "+FileNameOnly(output_file)+"\n\n")
```



```
    ' dump the needed results into a CSV file
    writeresults(output_file, "system.hourly.e_net,sv.lcoe_nom")
end
```

This example and several others are included in the standard SAM sample files. Please browse them at your convenience for additional information, or contact Solar Advisor Support at

`solar.advisor.support@nrel.gov`.