SML

Language Reference Manual

1. Introduction

SML has been proposed as a means of simplifying SPICE coding. By developing a wrapper language that generates SPICE code, the developers of SML have given the typical engineer the power to easily manipulate cumbersome circuit configurations, while harnessing the full power of the latest SPICE engine technologies. SML can be done without the hassle of being distracted by the nuances of any one particular SPICE implementation.

2. Lexical Conventions

2.1 Comments

Multiple line comments begin with "${" and end with "$}". Single line comments begin with a single "$" and are terminated by an end of line character.

2.2 Identifiers

An identifier is a letter followed by any combination of letters (a...z or A...Z), numbers and underscores ("_"). Identifiers are case-sensitive, and cannot be identical to any keyword. Identifiers are the valid names that can be given to variables.

2.3 Numbers

A number is a sequence of digits followed by an optional decimal point which may itself be followed by more digits.

2.4 Strings

A string is a sequence of ASCII characters surrounded in double quotes (" "). Just as in C, the character '\' acts as an escape character. '\t' represents a tab, '\n' a line feed, '\r' a carriage return and '\\' is used to represent a single '\'.

2.5 Objects

An object is any variable of type int, float, list, string etc. All variables are implemented as generic objects that internally store information about their type and where their data is stored.

2.6 Numerical Objects

Numerical values are stored in int and float objects. In practice, int values store the equivalent data of c++ long values and floats store the equivalent data of c++ double values.

2.7 Tokens

Let **n** denote a positive integer.
Let expr**n** be any expression that evaluates to an object.
Let list**n** be an expression that evaluates to a list object.
Let num**n** be an expression that evaluates to an int or float object.
Let string**n** be an expression that evaluates to a string object.
Let var**n** be a valid identifier (variable name).
Let vartype**n** be a valid data type, such as "int", "float", "list" or "string".

( ) (expr1)

Guarantees that expr1 will be evaluated before it is combined with surrounding expressions.

{ } {expr1, expr2, expr3}

Constructs a list containing the objects returned by expr1, expr2 and expr3. Any number of expressions can be enclosed in this manner to construct a list on the fly.

[ ] list1[num1]

As long as num1 evaluates to a positive integer N, the Nth element of list1 is returned.

[ ] list1[num1, num2]

As long as num1 and num2 evaluate to positive integers, this expression returns a list containing elements num1 to num2 of list1. Modifying the elements of the returned list will, in fact, affect the elements of list1.

[ ] string1[num1]
As long as num1 evaluates to a positive integer N, this returns a string object containing the Nth character of string1. Modifying this new string will affect the corresponding character in string1.

[ ] string1[num1, num2]
As long as num1 and num2 evaluate to positive integers, this returns a string object containing characters num1 to num2 of string1. Modifying this new string will affect those characters of string1.

[ ] datatype1[num1]

As long as num1 evaluates to a positive integer N, this returns a new list containing N objects of type datatype1.

,

The comma token separates items in lists that are constructed on the fly and separates input parameters to functions

+	num1 + num2
-	num1 - num2
*	num1 * num2
/	num1 / num2
%	num1 % num2

These return the sum, difference, product, quotient and remainder (respectively) of num1 and num2 (as a numerical object). Neither num1 nor num2 is effected by these operations.

=	expr1 = expr2

This sets the value of the object returned by expr1 to the value of the object returned by expr2. If expr1 and expr2 evaluate to lists then the list object returned by expr1 is left containing elements that have data identically matching those in the result of expr2. Note that after using the '=' operator, modifying the value (or elements) of what is returned by expr1 will have no effect on the value (or elements) of what is returned by expr2. In particular, if expr1 is a list and expr2 is a list containing SPICE objects then any connections between these objects will be preserved, so that in the left hand list the nth SPICE object will connect to the kth SPICE object in exactly the manner that they were connected in the right hand list for all positive integers n and k.

&=	var1 &= expr1

The name var1 can now be used to refer to the object that is returned by expr1. If the object returned by expr1 has been explicitly named before then the old name for the object becomes invalid.

==	expr1 == expr2

Returns an integer of value 1 if expr1 and expr2 contain data that is identical. Otherwise returns an integer of value 0. If expr1 and expr2 evaluate to lists then each element of the lists must contain identical data for this expression to return 1.

&==	var1 &== expr1

Returns an integer of value1 if var1 is a name for the object returned by expr1. Otherwise returns a zero valued integer.

&== expr1 &== expr2

Returns an integer of value 1 if the object returned by expr1 is referenced by the same named as the object returned by expr2. Otherwise returns a zero valued integer.

> num1 > num2
< num1 < num2
>= num1 >= num2
<= num1 <= num2

Returns an integer of value one if num1 has (respectively) a greater, smaller, greater or equal, or smaller or equal value than num2. Otherwise, returns a zero valued integer.

@ string1 @ string2

Returns a string object that is the result of appending string2 to the end of string1. Neither string1 nor string2 is affected by this operation.

@ list1 @ list2

Returns a list object that is the result of appending elements storing identical data to those in list2 to the end of list1. Neither list1 nor list2 are affected, and modifying the elements of the resulting list will not affect either of these lists.

#list1

Returns the number of elements in list1 as an integer value.

#string1

Returns the number of characters in string1 as an integer value.

OR num1 OR num2

Returns an integer with value 1 if either num1 or num2 evaluates to a non-zero integer. Otherwise returns an integer with value zero.

AND num1 AND num2

Returns an integer with value 1 if num1 and num2 both evaluate to non-zero integers. Otherwise returns an integer with value zero.

. expr1.property

Here the dot (.) operator returns an object representing some property of the object returned by expr1. Modifying this property object modifies the corresponding property of expr1. Depending on the type that expr1 evaluates to, property could be of any valid data type.

-> expr1 -> expr2

If expr1 and expr2 evaluate to node objects, this creates a SPICE connection or circuit wire between them. In particular, this can be used to bind one terminal of a device to a terminal of a different device. For example, resistor1.pos -> capacitor1.neg creates a wire (connection) between the positive terminal of resistor1 and the negative terminal of capacitor1 in our generated SPICE circuit.

2.8 Keywords

Below is a list of the keywords in SML:

if	list	true
while	string	false
else	res	AND
break	cap	OR
continue	ind	return
int	cs	
float	vs	

3. Data Types

3.1 Basic Object Types

float
: Stores a c++ double floating point value.

int
: Stores a c++ long integer value.

list
: Stores a list of objects.

string
: Stores a sequence of characters.

3.2 SPICE Object types

node

A SPICE node. These objects are stored within resistors, capacitors, etc. to represent the various terminals of a SPICE object.

For all of the following object types, obj.pos returns a node object representing the positive terminal of the object obj, and obj.neg returns a node representing its negative terminal.

res
A SPICE resistor object.
obj.resistance returns a float objecting representing the resistance of a resistor object obj.

cap
A SPICE capacitor object.
obj.capacitance returns a float objecting representing the capacitance of a capacitor object obj.

ind
A SPICE inductor object.
obj.inductance returns a float objecting representing the inductance of an inductor object obj.

cs
A SPICE current source object.
obj.current returns a float objecting representing the current of a current source object obj.

vs
A SPICE voltage source object.
obj.voltage returns a float objecting representing the voltage of a voltage source object obj.

4. Expressions and Statements

An expression is a syntactically permissible sequence of tokens, keywords and identifiers. Some expressions evaluate to objects and others do not. A statement is an expression terminated by one or more end of line character.

4.1 Primary Expressions

4.1.1 Identifiers
An identifier is a left or right value expression.

4.1.2 Constants
A constant is either a number or a string that is a right-value expression.

4.2 Arithmetic Expressions

Use of operators such as +,-,*, / to modify primary expressions. Multiplication and division take precedence over addition and subtraction.

4.3 Relational Expressions

Comparison using the < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), == (equal), != (not equal) operators on two primary expressions.

5. Statements

5.1 Assignment

An assignment assigns a constant or expression to a specified identifier.
< *left_expression* > = < *right_expression* >

5.2 Conditional Statement

Let exprn denote an expression where n is a positive integer

if	if expr1 { expr2}

Evaluates expr2 if and only if expr1 evaluates to a non-zero integer. An error is thrown if expr1 does not evaluate to an integer.

else	if expr1 {expr2} else if expr3 { expr 4} else {expr5}

Evaluates expression2 if expression1 evaluates to a non-zero integer. Otherwise, evaluates expr4 if expr3 evaluates to a non-zero integer. If neither expr2 nor expr4 was evaluated, then expr5 is evaluated. An error is thrown if expr1 or expr3 does not evaluate to an integer.

5.3 Iterative Statement

Let exprn denote an expression where n is a positive integer

while	while expr1 { expr2 }

Repeatedly evaluates expr2 until expr1 evaluates to an integer that has the value zero or the break keyword is reached in expr2. An error is thrown if expr1 does not evaluate to an integer.

break	while expr1 {

```
                expr2
                if( expr3 ) break
                expr4
        }
```

Evaluates the interior of the while loop until expr3 evaluates to a non-zero integer or expr1 evaluates to a non zero integer.

continue

```
while expr1
{
        expr2
        if(expr3) continue
        expr4
}
```

During each iteration of the while loop, expr4 is evaluated only if expr3 evaluates to an integer that has value zero.

return

```
type1 func()
{
        expr2
        return expr3
}
```

Returns the value that expr3 evaluates to as long it is of data type type1.

6. Operator precedence

SML operators follow the following precedence ordering (from weakest to strongest):

,
= &= ->
AND
OR
== &==
< > <= >=
@
+ -
* /
#
()
{ }
. []

7. Functions

Functions are defined as follows:

```
output_type func(input_type1 name1, input_type2 name2)
{
        function_body
}
```

Where output_type, input_type1 and input_type2 are valid data types. Functions can have any number of input arguments and can only be called below where they have been defined.

8. SPICE insertion

Lines of SPICE can be included directly in SML programs. These lines will be inserted into the final generated SPICE file in the order that they are executed in the SML program and will appear higher in the file than all SML related SPICE circuit elements and nodes that are generated. The syntax is as follows:

```
$[ SPICE_commands ]$
```

In particular, we can refer to SML variables in our SPICE insertions. To do this we essentially inject an SML expression into our injected lines of SPICE. This SML expression will be evaluated and as long as it evaluates to a float, int, node or valid circuit element such as res or cap the expression will be replaced with corresponding SPICE code that references that object or takes on that value. These SML injections within SPICE injections must be preceded by the # token, and they must be followed by a blank space. This final blank space will be removed when the SPICE code is generated. Thus, if in our SML we have a resistor object called R we might inject the following SPICE:

```
$[ SPICE_commands#R more_SPICE_commands#R.resistance even_more_SPICE_commands ]$
```

The interpreter will then replace #R by the appropriate SPICE referring to the circuit element R and #R.resistance will be replaced by the resistance value of resistor R. This power to inject allows SML to harness the power of any SPICE commands that are not directly implemented, such as those used to measure currents and voltages.

9. Sample SPICE insertions

```
$[.op ]$
```

This statement instructs SPICE to compute the DC steady state operating point for the circuit: voltage at the nodes, current in each voltage source, and the operating point for each element.

$[.dc #obj.param start stop incr]$

The .dc analysis allows you to sweep a variable var and perform a DC steady state solution of each value of that variable. The variable can be the name of an independent source or any element parameter. Start and stop are the starting and final values of obj.param. And, incr is the incremental value of the parameter being swept. Additionally, the sweep is done independently of the value of obj.param at simulation time.

$[.tran tincr stop START=tini UIC]$

This statement allows for a transient analysis to be conducted in the time domain for basic circuit elements where tincr is the time increment for calculating the variables, tstop and tini are the times at which the transient analysis stops and starts, respectively. The statement UIC (Use Initial conditions) causes SPICE to ttake the specified the initial values of dynamical components such as capacitors and inductors into account. Omitting the START statement is equivalent to setting START=0.

$[.ac TYPE np fstart fstop]$

This statement allows for time-invariant dynamical circuit analysis to be done in the frequency domain where fstart denotes the starting frequency, fstop denotes the final frequency, and TYPE defines how the np points on the frequency axis are taken (DEC for decade variation) and (LOG for logarithmic variation). For AC analysis to be conducted, a voltage or current source must be identified as an AC source with a particular phase and magnitude.

$[.print <analysis type> v(#obj1), i(#obj2), v(#obj2.pos, #gnd)]

This statement represents the simplest way of reporting simulation results in the output log where v(#obj) denotes the voltage across the positive and negative terminals of the object and v(#obj.pos, #gnd) denotes the voltage of the positive terminal with respect to ground. I(obj) denotes the current through the object.