Chapter Five

SLX'S EXTENSIBILITY MECHANISMS

5.1. Introduction

This chapter describes mechanisms that allow you to extend the SLX language to meet your own special needs. The benefits of developing extensions to SLX include the following:

- Extensions allow you to develop your own dialect of SLX, which facilitates describing components of the kinds of systems you simulate and specifying actions to be performed on these components.
- Extensions can provide shorthand notations that allow you to avoid writing the same things over and over.
- Extensions facilitate *central modification*. If you need to make a change to an SLX program, changing the definition of an extension automatically redefines all uses of the extension in a program. Such centralized modifications are easier and far less error-prone than changing multiple occurrences of similar functionality expressed without extensions.
- Extensions can improve execution efficiency. For example, consider an application that does a lot of conversion of angles from degrees to radians. One could write an SLX function to accomplish such conversions, or one could write a macro that expands into in-line code. In SLX, the overhead of procedure calls is higher than that of other languages. A macro for converting degrees to radians eliminates that overhead. In general, the use of very short procedures is inefficient.

SLX provides three forms of extensions:

- *Macros* are similar to macros used in other languages. They are most commonly used in contexts in which expressions must be supplied.
- *Statements* are "super-macros" that are used only in contexts in which a complete statement must be supplied. Apart from the contexts in which they are used, macros and statements are very similar.
- *Precursor Modules* are SLX modules that contain data, code, and macro/statement definitions that are typically used during program compilation. For example, a precursor module could include the definition of a dictionary, code to read files to populate the dictionary, and macro/statement definitions that utilize data stored in the dictionary.

All three forms of extensions operate as follows:

- When the SLX compiler encounters the start of a definition of an extension, it suspends what it is currently doing and initiates compilation of the extension.
- The extension in its entirety is compiled into machine instructions.
- Upon completion of compilation of the extension, the SLX compiler updates its symbol table to include all public information defined in the extension.
- The SLX compiler then resumes what it was originally doing.

The advantages of the above approach are as follows:

- Extensions can make used of previously defined extensions. Thus, the extensibility of SLX is unbounded, because extensions can be extended.
- Because extensions are translated into machine instructions; i.e., they are *not* interpretively processed, extensive reliance on extensions does not significantly increase program compilation times.
- Because extensions execute at compile-time, by writing extensions, you can become as involved as you wish in the translation of your programs from source code into executable code. If your application is unusual, you can devise notations that are application-specific and tell SLX how to translate these notations into "normal" SLX.

SLX macros and statements are similar to traditional macros in the following respects:

- They are most commonly used to generate source code that becomes part of the program being compiled. This process is called *macro expansion*.
- They usually include arguments that allow expressions to be "plugged into" their invocations.
- They are defined using a *metalanguage*. (A metalanguage is a language for describing a language.)
- They can include logic for *conditional expansion*; i.e., expansions can be made to depend on arguments and/or other specifications.

SLX macros and statements differ from traditional macros in the following respects:

- Traditional macro definitions have relatively high ratios of expansion text to logic. In SLX, the opposite if often true. In SLX, the logic for generating expansion text can be arbitrarily complex, consistent with its design goal of maximizing extensibility.
- Traditional macros employ a metalanguage that is distinct from, and far weaker than, the host language. For example, in C/C++, "preprocessor" directives such as #ifdef provide a limited form of conditional expansion. In SLX, the metalanguage used in statement/macro definitions is SLX itself, plus a handful of additions, and minus statements that are meaningful only during the execution of a simulation. (For example, all forms of time delays are disallowed.) This allows you to use the full power of SLX to describe complex logic. You can write loops, store data in sets, read and write files, etc., within macro definitions.

5.2. Defining and Using SLX Macros

5.2.1. The Format of a Macro Definition

An SLX macro definition takes the following form:

```
macro name prototype
  definition
    {
    expand ( … )  format;
    …
    }
```

The prototype portion of a macro describes the allowable syntax of a macro invocation. The following notations are available:

Notation	Description	Details
#*name*	Macro argument	Any valid SLX expression can be supplied in invocations of the macro. Supplied expressions are stored in an automatically-defined string variable named #*name*.
name	Keyword	User-defined keywords
@*name*	Keyword with normal meaning suppressed	SLX contains many reserved words. For example, "into" is used in "place" statements; e.g., "place x into y". If you wish to use "into" as a keyword in a macro prototype, you must prefix it with "@".
<#*name1* name2>	Keyword with assignment	"name2" is a keyword. If "name2" is supplied in a macro invocation, the "name2" keyword is stored into the variable named #name1. This notation is useful for detecting the presence of optional keywords in macro invocations. Note that more than one keyword can share the same #name1; e.g., it's possible to also specify <#*name1* name3>.
( )	Punctuation	Indicates required use of parentheses, which must be used in balanced pairs.
=	Punctuation	The most common use is "keyword = #variable".
,	Punctuation	Indicates the required presence of a comma.
{ }	Metalanguage grouping	All items inside a { } group are required. For example, "{ keyword1 = #k1 keyword2 = #k2 }" indicates that all six components are required and must be in the order shown.
[ ]	Metalanguage grouping	The items inside a [ ] group are optional, but if the first item inside the [ ] group is supplied in a macro invocation, all items within the group must be supplied. For example, "keyword [=] #k1" indicates that the "=" is optional.
|	Metalanguage separator used between two items or groups	Example: "{ keyword1 = #k1 } | { keyword2 = #k2 }" indicates that either "keyword1 = #k1" or "keyword2 = #k2" must be specified.

*	Metalanguage "repeat" operator	"*" indicates that the preceding item or group can be specified zero or more times. If the preceding item or group contains #name variables, the variables are treated as subscripted, where "#name[i]" specifies the ith instance of #name.
...	Metalanguage "repeat" operator	"..." indicates that the preceding item or group can be specified one or more times. If the preceding item or group contains #name variables, the variables are treated as subscripted, where "#name[i]" specifies the ith instance of #name.
,...	Comma-separated list	",..." indicates that the preceding item or group can be repeated. If it is repeated, successive instances must be separated by commas. If the preceding item or group contains #name variables, the variables are treated as subscripted, where "#name[i]" specifies the ith instance of #name.

5.2.2. Using the Expand Statement

The expand statement is used to format text that is generated by statements / macros and pass it to the SLX compiler. The expand statement is very similar to SLX's print statement. The major difference is that the print statement uses "_" or "*" characters to denote fields into which values are to be edited, while the expand statement uses "#" characters for this purpose. Thus, a typical print statement has the following appearance:

```
print (x, sqrt(x))    "The square root of _.___ is _.___\n";
```

A typical expand statement has the following appearance:

```
expand (#arg)              "sqrt(#)"
```

If the picture describing the format of expanded text contains quotation marks, or if the picture spans multiple lines, we strongly suggest using |" … "| strings. Consider the following example:

```
expand(#condition, #variable)

|" {
   if (#)
   print (#) "The answer is \"__._\"\n";
   }
"|;
```

Strings delimited by |" and "| include *all* characters between the delimiters, including invisible tab characters and linefeeds, and "normally" quoted strings. The inclusion of linefeed characters implies that a string using the new delimiters can span multiple lines without requiring any special actions. If you supply a |" start-of-string delimiter, but forget the "| end-of-string terminator, SLX will gobble up the remainder of the source file before complaining about an unterminated string. This is an error that anyone using this form of string will make sooner or later; however, such errors are clearly pointed out by SLX and are easily corrected. The notational convenience outweighs the risks.

Within a |"..."| string, escape sequences, e.g. "\n", are interpreted only if they fall outside any normally quoted string inside the |"..."| string. Escape sequences that fall inside normally quoted strings are passed through verbatim, with the understanding that they will be processed when the text produced by the expand statement is processed later.

5.2.3. Macro Examples

Suppose that you wanted to write a "min" macro such that "min(x, y)" returned the minimum of x and y. The following example illustrates how to do so:

```
macro min(#x, #y)
 definition
    {
    expand(#x, #y, #x, #y)                "(#) <= (#) ? (#) : (#)";
    }
```

A sample invocation is

```
    z = min(a+b, c-d);
```

This invocation would expand into

```
    (a+b) <= (c-d) ? (a+b) : (c-d);            // note the defensive use of parentheses
```

Suppose that you wanted to write a "dmin" macro that returned the minimum of two *or more* double expressions. As we saw in the preceding example, if the minimum of two values is to be found, SLX's "a ? b : c" operator can be used. When the minimum of more than two values is to be determined, the macro definition must (1) define a temporary variable to contain the result, and (2) iterate through the list of values. The use of the "inline" operator is also very helpful, if not essential. The following is one possible implementation:

```
    macro dmin (#expression,...)          // comma-delimited list of expressions
     definition
        {
        static int    expansion_count;
        string(20)  temp_name;

        int            i;

        write string=temp_name (++expansion_count)    "tempmin\_\__";

        expand(temp_name, temp_name, #expression[1])

|"   inline double
    {
      double  #;
      # = #;
"|;
            for (i = 2; #expression[i] != ""; i++)          // iterate until list exhausted

                  expand(#expression[i], temp_name, temp_name, #expression[i])

|"    if (# < #)
            # = #;
"|;

            expand(temp_name)

|"    return #;
      }
"|;
            }
```

In the above example, a static integer counter, expansion_count, is incremented for each expansion of the dmin macro. The count is used to construct a temporary variable name.

The following is a sample expansion of dmin(a, b, c):

```
•+        inline double
•+        {
•+        double tempmin__2;
•+        tempmin__2 = a;
•+        if (b < tempmin__2)
•+                    tempmin__2 = b;
•+        if (c < tempmin__2)
•+                    tempmin__2 = c;
•+        return tempmin__2;
•+        }
```

5.3. Defining and Using SLX Statements

A statement definition takes the following form:

```
statement name prototype ;
  definition
    {
    expand ( … )  format;
    …
    }
```

Note that a statement prototype must contain a terminating “;”. In all other respects, statement definitions and macro definitions follow identical rules. Typically, although not necessarily, statement definitions use a broader range of syntax options than do macro definitions.

5.3.1. Using the diagnose Statement

The diagnose statement is used to issue compile-time and run-time warnings and error messages. Although one might think that such a statement would be "hard-wired" into SLX, the diagnose statement is in fact a defined statement that expands into lower-level SLX statements. The format of the diagnose statement is as follows:

```
statement diagnose {    <#caller caller>
                       | <#invocation invocation >
                       | { #faulty_item [, #faulty_item2 ] } }

                       [ <#when compile_time> | <#when run_time> ]
                       [ <#severity error> | <#severity warning> ]
                       [( #output_item,... )]
                       #picture ;
```

Let's work through the above definition. The first component of the diagnose statement is a mandatory ({ }-enclosed) group of three possible alternatives, the last of which has two possible forms. The first component can be

	caller
or	invocation
or	#faulty_item
or	#faulty_item, #faulty_item2

If "caller" is specified, the string variable named #caller is set to "caller". This form should be used only within a procedure, because it specifies that the diagnostic message will be placed at the point at which the procedure containing the diagnose statement was called. If "invocation" is specified, the string variable named #invocation is set to "invocation", and the diagnostic message is placed at the point of the diagnose statement itself. ("invocation" refers to the invocation of the diagnose statement.) If neither "caller" nor "invocation" is specified, either one or two faulty items must be supplied. For diagnose statements used inside statement / macro definitions, faulty items must be #names that appear in the prototype of the statement / macro. When faulty items are specified, they are highlighted in red in the program listing.

The next component is either "compile_time" or "run_time". These keywords indicate when a diagnostic is meant to be issued. If either is specified, the string variable #when is set to the name of the keyword.

The next component is either "error" or "warning". These keywords indicate the severity of a diagnostic message. If either is specified, the string variable #severity is set to the name of the keyword.

The next component is optional. If specified, it is a parenthesized list of expressions to be edited into the picture (a character string) that specifies the format of the message.

The last component is the picture. If the "faulty items" form of the diagnose statement is used, within the picture, a "^" character must be supplied for each faulty item. The faulty items replace the "^" characters when the diagnose statement is executed. If a list of expressions to be edited into the diagnostic message is specified, “_” fields must be provided in the picture to specify how the expressions are to be edited. An example of the diagnose statement is given in the next section.

5.3.2. Sample Statement Definition

The following comprehensive example is the “read” statement, predefined in SLX itself:

```
statement read { [ <#newline newline > ] | [ <#record record> ] }*

{  file = #file
 | @string = #string
 | prompt = #prompt
 | end = #eof
 | err = #err }*

 [ data = ] ( #input_item,... ) ;

   definition
     {
     int i, ioflags;
     string(50) eof_exit, err_exit;
     if (#eof == "")
          eof_exit = "NULL";
     else
          {
          eof_exit = #eof;
          ioflags |= TRAP_EOF;
          }

     if (#err == "")
          err_exit = "NULL";
     else
          {
          err_exit = #err;
          ioflags |= TRAP_ERROR;
          }

     if (#record != "")
           ioflags |= READ_RECORD;

     if (#newline != "")
           ioflags |= READ_NEWLINE;

     if (#file != "")
          if (#prompt == "")
               expand (#file, ioflags)

|"        {
          slx_start_input(#,#, "");
"|;
          else
               expand (#file, ioflags, #prompt)
```

```
|"        {
          slx_start_input(#,#,#);
"|;
     else
          if (#string != "")
               expand (#string, ioflags)

|"        {
          slx_start_string_input(#,#);
"|;
          else
               if (#prompt == "")
                    expand (ioflags)
|"        {
          slx_start_input(stdin,#, "");
"|;
               else
                    expand (ioflags, #prompt)
|"        {
          slx_start_input(stdin,#,#);
"|;

     for (i = 1; #input_item[i] != ""; i++)
          expand(#input_item[i], eof_exit, err_exit)

|"        slx_read(#, #, #);
"|;

     expand

|"        }
"|;
     if (#record != "" && i != 2)
          diagnose #record compile_time error (i-1)

               "When the ^ option is used, the entire record is read at once.  "
               "Therefore, a single string variable should be used in the input list."
               "You have specified # variables";
     }
```

Let’s work our way through the above definition. The first component of a read statement is a group that may be specified zero or more times, as indicated by { }*. Within the group, there are two options, the “newline” keyword and the “record” keyword. If an invocation of the read statement uses either keyword, a #name variable is set to contain the keyword. Because the group can be repeated, both “newline” and “record” can be specified. In fact, the metalanguage notation allows multiple instances of both keywords.

The next component of the read statement is another group that can be specified zero or more times. Within this group, there are five options, each of which takes the form *keyword = #name*. Note that for the last option, “data =” is optional. Because all five specifications are optional, and the group containing them can be repeated, all five can be specified in any order. As above, the metalanguage notation allows more than one specification of each option. If an option is specified more than once in an invocation of the read statement, the option's #name will end up being set to the rightmost specification.

The next several sections of the statement definition test for the presence of various options and take actions that reflect the options used. In many cases, #name variables are copied into local string variables local to the read statement's definition, e.g., eof_exit = #eof;

The next-to-last section of the read statement's definition iterates through the list of #input_items, generating an slx_read() call for each.

Finally, a test is made to see if the "record" option has been used, and more than one #input_item has been specified. "read record" requires that exactly one #input_item be specified. If more than one #input_item is specified, the read statement will force a compile-time error, using the diagnose statement to do so. The "record" keyword will be edited into the diagnostic message, and it will be highlighted in red in the program listing.

5.4. Using a Precursor Module

The following example shows a precursor module, OperatorDefinitionIO, that contains definitions of a file and a statement. By virtue of its placement in a precursor module, the file becomes available for use when SLX completes compilation of the precursor module. Furthermore, the file remains open for use across successive invocations of the "ctread" statement. Thus, each invocation and expansion of "ctread" reads another line in the file.

```
precursor module OperatorDefinitionIO
    {
    filedef infile name="opdefio.dat";

    statement ctread;
       definition
         {
         string(80)    name;
         int        number;

         read file=infile (name, number);

         expand(name, number)

|"   print "we read in name = # and number = #\n";
"|;
         }
    }
```

```
module test
    {
    procedure main()
        {
        ctread;
        ctread;
        }
    }
```

This is a very simple example. Since precursor modules can contain anything a normal SLX module can contain, in the general case, precursor modules can include definitions of classes, functions, statements, macros, etc. The primary advantage of precursor modules is that once a precursor module has been compiled, everything within them (subject to rules of public/private visibility) becomes available for subsequent use not only at run-time, but also at *compile*-time.

In traditional languages, the distinction between operations that can occur at compile-time and those that can occur at run-time is very sharp. In SLX, this distinction is deliberately blurred. The ability to execute user-supplied code at compile-time is one of the unique characteristics of SLX.

5.5. Details and Pitfalls

Thus far, we have not discussed *how* SLX scans statement / macro arguments to assign them into their corresponding #names. The first rule is that supplied arguments must be *complete*, valid SLX expressions. For example "(a + b) * c" is a valid argument, while "a +" is not.

The second rule is that the syntax specified in a statement / macro prototype must be such that as SLX scans statement / macro arguments from left-to-right, it must be able to distinguish argument syntax based only on the current symbol. For example consider the following statement prototype:

```
statement MyStatement { keyword1A #arg1 keyword1B } | { keyword2A #arg2 keyword2 } ;
```

SLX can easily distinguish between the two alternatives by looking at the first keyword specified. If it's "keyword1A", the argument must be of the first format, and if it's "keyword2A", it must be of the second format. In contrast, consider the following statement prototype:

```
statement MyStatement { #arg1 keyword1 } | { #arg2 keyword2 } ;
```

In the above prototype, SLX cannot distinguish which of the two alternative formats is to be used based on the first symbol, because both alternatives can begin with expressions of arbitrary complexity; thus SLX would not be able to distinguish between "(x+y) keyword1" and "(x+y) keyword2".

The third rule is that SLX must be able to parse arguments based only on their syntax. This is primarily of importance when consecutive #names are used in a prototype. Consider the following example:

```
statement MyStatement #arg1 #arg2 ;
```

For the above statement, an invocation of the form “MyStatement a b;” is easily parsed, because “a” and “b” are each obviously complete expressions; however, an invocation of the form “MyStatement a (b+c);” is problematical. As far as SLX is concerned, “a (b+c)” has the syntax of a procedure call, and the second argument of the statement is missing. Because parsing of statement / macro arguments is done on a purely syntactic basis, SLX will not check to see whether “a” is in fact a procedure. Actually, it *cannot* do so, because forward references to procedures defined later in a program are perfectly valid, while statements and macros are expanded as they are encountered. One possible work-around in the current case would be to write “MyStatement (a) (a+b);”. Because of the potential confusion, if at all possible, do not use consecutive #names in statement / macro prototypes.