A Comparison of SLX and C

The design of SLX was heavily influenced by our experience with C. Like C, SLX offers a high ratio of functionality to complexity; i.e., it offers a great deal of bang for the buck. For the most part, the syntax of SLX is similar to that of C. However, in areas where we felt C's terseness was a source of pitfalls (especially for the beginner), we implemented slightly lengthier, more readable forms. A brief list of the major differences between SLX and C follows.

1. In SLX, all variables are initialized to a known state. For example, arithmetic values are initialized to zero, and pointers are initialized to NULL values.

2. In C, pointers can point to just about anything. In SLX, pointers can only point to objects. Thus, you cannot have a pointer to an integer or a character in SLX. If you need a pointer to an integer value, the integer value must be encapsulated in an object.

3. In SLX, all pointer operations are validated. If x is declared as a pointer to a widget, and y is an attribute of the widget, then if an expression of the form "x -> y" is used, x must point to a widget. One of the most common errors in the use of C is to make references using either NULL or uninitialized pointers. SLX traps all such references. In addition, SLX maintains use counts for all objects. Each time the address of an object is copied (usually into a pointer variable), the object's use count is incremented. If a pointer to an object is assigned a new value, the use count of the object to which it used to point is decremented. This approach solves the "gone but not forgotten" problem, which occurs when an object is destroyed, but there are still active pointers pointing to it. In SLX, destruction of an object is deferred until the object's use count goes to zero.

   With a single exception, SLX also solves the "forgotten, but not gone" problem. This problem occurs when all pointers to a dynamically created object are reassigned. When this happens, the object still exists, but there is no longer any way to access it. In SLX, when the last pointer to an object is reassigned, the object's use count goes to zero, and the object is *automatically* destroyed. The only problem that cannot be solved in this manner is the *automatic* release of ring structures. For example if object A points to object B, B points to C, and C points to A, forming a ring, then none of the objects in the ring can be released, even though all references from outside the ring are deleted. There is no general solution to this problem.

4. In SLX, the declaration syntax for procedures is quite different from that of C. In addition, SLX, does not use function prototypes. A key concept in the design of SLX is that the SLX compiler "sees" all functions and function invocations. Thus, SLX is able to perform exhaustive error checking on function usage without requiring the use of cumbersome function prototypes. The current release of SLX is a "compile, load, and go" system; i.e., there is no provision for producing object code. However, when this feature is added, the object code produced will contain symbol table information allowing the compiler to perform error checking for uses of separately compiled functions.

5. In C, procedure arguments are passed by value. Modifications to argument values within a procedure are not reflected in the calling procedure. If you wish to modify a value passed to a C procedure, you must pass a pointer to the object. In SLX, procedure arguments are passed by address and can be declared "in" (read-only), "inout" (read/write), or "out" (write-only). By default, arguments are assumed to be "in" values. Attempts to modify an "in" argument within a procedure are flagged as errors. Passing constants or read-only values as arguments to a procedure requiring write or read-write access to the corresponding formal parameter is likewise flagged.

6. In SLX, "import" has a much deeper meaning than C's "#include." In C, "#include" is a simple mechanism for incorporating copies of C code into separately compilable source files. In SLX, imported files must contain a complete module or group of modules. The following example illustrates the subtle differences in approach. Consider two files, "file1" and "file2," each of which refers to an object of type x. In C, one must provide identical, parallel declarations of x in the two files. This is typically done by placing the declaration for x in a header file and "#including" the header file in both source files. One could, however, use separate, but identical declarations. In SLX, such parallel declarations are not allowed. There cannot be multiple copies of any "shared" definitions. This restriction is essential for performing pointer validation. If SLX code in one file calls a function in another file, passing it a pointer to an object of type x, a single definition of x must exist. If x were

declared in both files, there would be two different kinds of x objects, and pointers to one kind would not be interchangeable with the other.

7. SLX does not allow embedded assignments. For example,

```
if ((a = b) == c)
        …
```

is allowed in C but not in SLX. We disallowed this form, because in our experience, it had proved to be a rich source of pitfalls, especially for the beginner.

8. SLX does not support the comma operator. For example,

```
a = b, ++c;
```

is valid in C, but not in SLX. Is anyone ever sure how the comma operator works without looking it up?

9. In C, enumerated types ("enums") are more or less freely interchangeable with ints. In SLX, enums are a full-fledged type, *not* interchangeable with integers. This allows the SLX compiler to do a much better job of checking enum usage for errors. In addition, SLX provides two very convenient enum-based constructs: the use of enumerated types as array dimensions, and a for-loop construct for iterating through all values of an enumerated type. Consider the following example:

```
type enum(idle, running, down)      machine_state;
type enum(lathe, drill, shaper)             machine_type;

int         count[machine_type, machine_state];          // array bounds are enums
machine_type     t;
machine_state    s;
…
for (t = each machine_type)
        for(s = each machine_state)
                print (count[machine_type, machine_state]) …
```

In SLX, enums are initialized to a value of NONE. Successor and predecessor functions are provided for enums. Thus "successor(idle)" in machine_state equals "running," and "successor(running)" equals "NONE."

10. In SLX, the switch statement is more powerful than that of C. In SLX, the variable used to select a switch case can be of any mode. For example, cases can be selected by using enums or character strings.

11. Unlike C, in which character strings are treated as arrays of characters, SLX provides true string variables.. A substring operator is provided for extraction of, and modification of, substrings. (The substring operator can be used on the left side of an assignment statement.) SLX also provides a concatenation operator ("cat") and a concatenation assignment operator ("cat=").

12. SLX provides full-fledged boolean variables. In C, TRUE and FALSE are commonly defined as synonyms for integer values 1 and 0, respectively. In SLX, TRUE and FALSE are built-in boolean constants. Variables declared as boolean can assume only these values. In SLX, in constructs which require a true/false value, a boolean-valued expression *must* be used. In C, integer-valued, float-valued, and pointer-valued expressions can be used, where the convention is that non-zero (non-NULL) is true and zero (NULL) is false. This is a rich source of pitfalls. For example, in C, one could write

```
if (i = 1)          // integer-valued, embedded assignment (unintentional?)
        …
```

although the better C compilers would issue a warning. In SLX, neither the embedded assignment nor the use of an integer-valued expression in a true/false context are allowed.

13. SLX does not include the troublesome “x ? y : z” construct.

14. SLX provides macro- and statement-definition features which are much more powerful than C’s macros. The metalanguage used within SLX statement/macro definitions is, in fact, SLX itself. The significance of this fact cannot be understated. The full range of SLX language features (absent only the simulation-specific verbs such as “advance”) is available for performing logic- and data-intensive conditional compilation. For example, it is possible, within the context of a statement/macro definition, to read and write files, create and destroy objects, place objects into and remove objects from sets, and issue compile-time diagnostics for macro/statement usage errors. Features such as C’s #ifdef pale by comparison.