A Guide to the S-Lang Language

John E. Davis, davis@space.mit.edu Jun 18, 2005

Preface

S-Lang is an interpreted language that was designed from the start to be easily embedded into a program to provide it with a powerful extension language. Examples of programs that use **S-Lang** as an extension language include the **jed** text editor and the **slrn** newsreader. Although **S-Lang** does not exist as a separate application, it is distributed with a quite capable program called **slsh** ("slang-shell") that embeds the interpreter and allows one to execute **S-Lang** scripts, or simply experiment with **S-Lang** at an interactive prompt. Many of the the examples in this document are presented in the context of one of the above applications.

S-Lang is also a programmer's library that permits a programmer to develop sophisticated platform-independent software. In addition to providing the **S-Lang** interpreter, the library provides facilities for screen management, keymaps, low-level terminal I/O, etc. However, this document is concerned only with the extension language and does not address these other features of the **S-Lang** library. For information about the other components of the library, the reader is referred to **The S-Lang Library Reference**.

A Brief History of S-Lang

I first began working on **S-Lang** sometime during the fall of 1992. At that time I was writing a text editor (**jed**), which I wanted to endow with a macro language. It occured to me that an application-independent language that could be embedded into the editor would prove more useful because I could envision embedding it into other programs. As a result, **S-Lang** was born.

S-Lang was originally a stack language that supported a postscript-like syntax. For that reason, I named it **S-Lang**, where the *S* was supposed to emphasize its stack-based nature. About a year later, I began to work on a preparser that would allow one unfamiliar with stack based languages to make use of a more traditional infix syntax. Currently, the syntax of the language resembles C, nevertheless some postscript-like features still remain, e.g., the '`%`' character is still used as a comment delimiter.

Acknowledgements

Since I first released **S-Lang**, I have received a lot feedback about the library and the language from many people. This has given me the opportunity and pleasure to interact with a number of people to make the library portable and easy to use. In particular, I would like to thank the following individuals:

Luchesar Ionkov for his comments and criticisms of the syntax of the language. He was the person who made me realize that the low-level byte-code engine should be totally type-independent. He also improved the tokenizer and preparser and impressed upon me that the language needed a grammar.

Mark Olesen for his many patches to various aspects of the library and his support on AIX. He also contributed a lot to the pre-processing (`SLprep`) routines.

John Burnell for the OS/2 port of the video and keyboard routines. He also made value suggestions regarding the interpreter interface.

Darrel Hankerson for cleaning up and unifying some of the code and the makefiles.

Dominik Wujastyk who was always willing to test new releases of the library.

Michael Elkins for his work on the curses emulation.

Hunter Goatley, Andy Harper, Martin P.J. Zinser, and Jouk Jansen for their VMS support.

Dave Sims and Chin Huang for Windows 95 and Windows NT support, and Dino Sangoi for the Windows DLL support.

I am also grateful to many other people who send in bug-reports, bug-fixes, little enhancements, and suggestions, and so on. Without such community involvement, **S-Lang** would not be as well-tested and stable as it is. Finally, I would like to thank my wife for her support and understanding while I spent long weekend hours developing the library.

Contents

Chapter 1

Introduction

S-Lang is a powerful interpreted language that may be embedded into an application to make the application extensible. This enables the application to be used in ways not envisioned by the programmer, thus providing the application with much more flexibility and power. Examples of applications that take advantage of the interpreter in this way include the **jed** editor and the **slrn** newsreader.

1.1 Language Features

The language features both global and local variables, branching and looping constructs, user-defined functions, structures, datatypes, and arrays. In addition, there is limited support for pointer types. The concise array syntax rivals that of commercial array-based numerical computing environments.

1.2 Data Types and Operators

The language provides built-in support for string, integer (signed and unsigned long and short), double precision floating point, and double precision complex numbers. In addition, it supports user defined structure types, multi-dimensional array types, lists, and associative arrays. To facilitate the construction of sophisticated data structures such as linked lists and trees, the language also inclues a "reference" type. The reference type provides much of the same flexibility as pointers in other languages. Finally, applications embedding the interpreter may also provide special application specific types, such as the `Mark_Type` that the **jed** editor provides.

The language provides standard arithmetic operations such as addition, subtraction, multiplication, and division. It also provides support for modulo arithmetic as well as operations at the bit level, e.g., exclusive-or. Any binary or unary operator may be extended to work with any data type, including user-defined types. For example, the addition operator (`+`) has been extended to work between string types to permit string concatenation.

The binary and unary operators work transparently with array types. For example, if `a` and `b` are arrays, then `a + b` produces an array whose elements are the result of element by element addition of `a` and `b`. This permits one to do vector operations without explicitly looping over the array indices.

1.3 Statements and Functions

The **S-Lang** language supports several types of looping constructs and conditional statements. The looping constructs include `while`, `do...while`, `for`, `forever`, `loop`, `foreach`, and `_for`. The conditional statements include `if`, `if-then-else`, and `!if`.

User defined functions may be defined to return zero, one, or more values. Functions that return zero values are similar to "procedures" in languages such as PASCAL. The local variables of a function are always created on a stack allowing one to create recursive functions. Parameters to a function are always passed by value and never by reference. However, the language supports a *reference* data type that allows one to simulate pass by reference.

Unlike many interpreted languages, **S-Lang** allows functions to be dynamically loaded (function autoloading). It also provides constructs specifically designed for error handling and recovery as well as debugging aids (e.g., function tracebacks).

Functions and variables may be declared as private belonging to a namespace associated with the compilation unit that defines the function or variable. The ideas behind the namespace implementation stem from the C language and should be quite familiar to any one familiar with C.

1.4 Error Handling

The **S-Lang** language has a try/throw/catch/finally exception model whose semantics are similar to that of other languages. Users may also extend the exception class hierarchy with user-defined exceptions. The `ERROR_BLOCK` based exception model of **S-Lang** 1.x is still supported but deprecated.

1.5 Run-Time Library

Functions that compose the **S-Lang** run-time library are called *intrinsics*. Examples of **S-Lang** intrinsic functions available to every **S-Lang** application include string manipulation functions such as `strcat`, `strchop`, and `strcmp`. The **S-Lang** library also provides mathematical functions such as `sin`, `cos`, and `tan`; however, not all applications enable the use of these intrinsics. For example, to conserve memory, the 16 bit version of the **jed** editor does not provide support for any mathematics other than simple integer arithmetic, whereas other versions of the editor do support these functions.

Most applications embedding the languages will also provide a set of application specific intrinsic functions. For example, the **jed** editor adds over 100 application specific intrinsic functions to the language. Consult your application specific documentation to see what additional intrinsics are supported.

Operating systems that support dynamic linking allow a slang interpreter to dynamically link additional libraries of intrinsic functions and variables into the interpreter. Such loadable objects are called **modules**. A separate chapter of this manual is devoted to this important feature.

1.6 Input/Output

The language supports C-like stdio input/output functions such as `fopen`, `fgets`, `fputs`, and `fclose`. In addition it provides two functions, `message` and `error`, for writing to the standard output device and standard error. Specific applications may provide other I/O mechanisms, e.g., the **jed** editor supports I/O to files via the editor's buffers.

1.7 Obtaining more information about S-Lang

Comprehensive information about the library may be obtained via the World Wide Web from http://www.jedsoft.org/slang/ . In particular see http://www.jedsoft.org/slang/download.html for downloading the latest version of the library.

Users with generic questions about the interpreter are encouraged to post questions to the Usenet newsgroup `alt.lang.s-lang`. More specific questions relating to the use of **S-Lang** within some application may be better answered in an application-specifc forum. For example, users with questions about using **S-Lang** as embedded in the **jed** editor are more likely to be answered in the `comp.editors` newsgroup or on the **jed** mailing list. Similarly users with questions concerning **slrn** will find `news.software.readers` to be a valuable source of information.

Developers who have embedded the interpreter are encouraged to join the **S-Lang** mailing list. To subscribe to the list or just browse the archives, visit http://www.jedsoft.org/slang/mailinglists.html .

Chapter 2

Overview of the Language

This purpose of this section is to give the reader a feel for the **S-Lang** language, its syntax, and its capabilities. The information and examples presented in this section should be sufficient to provide the reader with the necessary background to understand the rest of the document.

2.1 Variables and Functions

S-Lang is different from many other interpreted languages in the sense that all variables and functions must be declared before they can be used.

Variables are declared using the `variable` keyword, e.g.,

```
variable x, y, z;
```

declares three variables, `x`, `y`, and `z`. Note the semicolon at the end of the statement. *All **S-Lang** statements must end in a semi-colon.*

Unlike compiled languages such as C, it is not necessary to specify the data type of a **S-Lang** variable. The data type of a **S-Lang** variable is determined upon assignment. For example, after execution of the statements

```
x = 3;
y = sin (5.6);
z = "I think, therefore I am.";
```

`x` will be an integer, `y` will be a double, and `z` will be a string. In fact, it is even possible to re-assign `x` to a string:

```
x = "x was an integer, but now is a string";
```

Finally, one can combine variable declarations and assignments in the same statement:

```
variable x = 3, y = sin(5.6), z = "I think, therefore I am.";
```

Most functions are declared using the `define` keyword. A simple example is

```
define compute_average (x, y)
{
   variable s = x + y;
   return s / 2.0;
}
```

which defines a function that simply computes the average of two numbers and returns the result. This example shows that a function consists of three parts: the function name, a parameter list, and the function body.

The parameter list consists of a comma separated list of variable names. It is not necessary to declare variables within a parameter list; they are implicitly declared. However, all other *local* variables used in the function must be declared. If the function takes no parameters, then the parameter list must still be present, but empty:

```
define go_left_5 ()
{
   go_left (5);
}
```

The last example is a function that takes no arguments and returns no value. Some languages such as PASCAL distinguish such objects from functions that return values by calling these objects *procedures*. However, **S-Lang**, like C, does not make such a distinction.

The language permits *recursive* functions, i.e., functions that call themselves. The way to do this in **S-Lang** is to first declare the function using the form:

```
define function-name ();
```

It is not necessary to declare a list of parameters when declaring a function in this way.

Perhaps the most famous example of a recursive function is the factorial function. Here is how to implement it using **S-Lang**:

```
define factorial ();   % declare it for recursion

define factorial (n)
{
   if (n < 2) return 1;
   return n * factorial (n - 1);
}
```

This example also shows how to mix comments with code. **S-Lang** uses the '**%**' character to start a comment and all characters from the comment character to the end of the line are ignored.

2.2 Strings

Perhaps the most appealing feature of any interpreted language is that it frees the user from the responsibility of memory management. This is particularly evident when contrasting how **S-Lang** handles string variables with a lower level language such as C. Consider a function that concatenates three strings. An example in **S-Lang** is:

```
define concat_3_strings (a, b, c)
{
   return strcat (a, b, c);
}
```

This function uses the built-in `strcat` function for concatenating two or more strings. In C, the simplest such function would look like:

```
char *concat_3_strings (char *a, char *b, char *c)
{
   unsigned int len;
   char *result;
   len = strlen (a) + strlen (b) + strlen (c);
   if (NULL == (result = (char *) malloc (len + 1)))
     exit (1);
   strcpy (result, a);
   strcat (result, b);
   strcat (result, c);
   return result;
}
```

Even this C example is misleading since none of the issues of memory management of the strings has been dealt with. The **S-Lang** language hides all these issues from the user.

Binary operators have been defined to work with the string data type. In particular the `+` operator may be used to perform string concatenation. That is, one can use the `+` operator as an alternative to `strcat`:

```
define concat_3_strings (a, b, c)
{
   return a + b + c;
}
```

See the section on 3.1.4 (Strings) for more information about string variables.

2.3 Referencing and Dereferencing

The unary prefix operator, `&`, may be used to create a *reference* to an object, which is similar to a pointer in other languages. References are commonly used as a mechanism to pass a function as an argument to another function as the following example illustrates:

```
define compute_functional_sum (funct)
{
   variable i, s;

   s = 0;
   for (i = 0; i < 10; i++)
    {
       s += (@funct)(i);
```

```
    }
   return s;
}

variable sin_sum = compute_functional_sum (&sin);
variable cos_sum = compute_functional_sum (&cos);
```

Here, the function `compute_functional_sum` applies the function specified by the parameter `funct` to the first 10 integers and returns the sum. The two statements following the function definition show how the `sin` and `cos` functions may be used.

Note the `@` operator in the definition of `compute_functional_sum`. It is known as the *dereference* operator and is the inverse of the reference operator.

Another use of the reference operator is in the context of the `fgets` function. For example,

```
define read_nth_line (file, n)
{
   variable fp, line;
   fp = fopen (file, "r");

   while (n > 0)
     {
        if (-1 == fgets (&line, fp))
          return NULL;
        n--;
     }
   return line;
}
```

uses the `fgets` function to read the nth line of a file. In particular, a reference to the local variable `line` is passed to `fgets`, and upon return `line` will be set to the character string read by `fgets`.

Finally, references may be used as an alternative to multiple return values by passing information back via the parameter list. The example involving `fgets` presented above provided an illustration of this. Another example is

```
define set_xyz (x, y, z)
{
   @x = 1;
   @y = 2;
   @z = 3;
}
variable X, Y, Z;
set_xyz (&X, &Y, &Z);
```

which, after execution, results in `X` set to 1, `Y` set to 2, and `Z` set to 3. A C programmer will note the similarity of `set_xyz` to the following C implementation:

```
void set_xyz (int *x, int *y, int *z)
{
   *x = 1;
```

```
   *y = 2;
   *z = 3;
}
```

2.4 Arrays

The **S-Lang** language supports multi-dimensional arrays of all datatypes. For example, one can define arrays of references to functions as well as arrays of arrays. Here are a few examples of creating arrays:

```
variable A = Int_Type [10];
variable B = Int_Type [10, 3];
variable C = [1, 3, 5, 7, 9];
```

The first example creates an array of 10 integers and assigns it to the variable `A`. The second example creates a 2-d array of 30 integers arranged in 10 rows and 3 columns and assigns the result to `B`. In the last example, an array of 5 integers is assigned to the variable `C`. However, in this case the elements of the array are initialized to the values specified. This is known as an *inline-array*.

S-Lang also supports something called a *range-array*. An example of such an array is

```
variable C = [1:9:2];
```

This will produce an array of 5 integers running from 1 through 9 in increments of 2.

Arrays are passed by reference to functions and never by value. This permits one to write functions that can initialize arrays. For example,

```
define init_array (a)
{
   variable i, imax;

   imax = length (a);
   for (i = 0; i < imax; i++)
     {
        a[i] = 7;
     }
}

variable A = Int_Type [10];
init_array (A);
```

creates an array of 10 integers and initializes all its elements to 7.

There are more concise ways of accomplishing the result of the previous example. These include:

```
A = [7, 7, 7, 7, 7, 7, 7, 7, 7, 7];
A = Int_Type [10];  A[[0:9]] = 7;
A = Int_Type [10];  A[*] = 7;
```

The second and third methods use an array of indices to index the array `A`. In the second, the range of indices has been explicitly specified, whereas the third example uses a wildcard form. See chapter 10 (Arrays) for more information about array indexing.

Although the examples have pertained to integer arrays, the fact is that **S-Lang** arrays can be of any type, e.g.,

```
A = Double_Type [10];
B = Complex_Type [10];
C = String_Type [10];
D = Ref_Type [10];
```

create 10 element arrays of double, complex, string, and reference types, respectively. The last example may be used to create an array of functions, e.g.,

```
D[0] = &sin;
D[1] = &cos;
```

The language also defines unary, binary, and mathematical operations on arrays. For example, if `A` and `B` are integer arrays, then `A + B` is an array whose elements are the sum of the elements of `A` and `B`. A trivial example that illustrates the power of this capability is

```
variable X, Y;
X = [0:2*PI:0.01];
Y = 20 * sin (X);
```

which is equivalent to the highly simplified C code:

```
double *X, *Y;
unsigned int i, n;

n = (2 * PI) / 0.01 + 1;
X = (double *) malloc (n * sizeof (double));
Y = (double *) malloc (n * sizeof (double));
for (i = 0; i < n; i++)
  {
    X[i] = i * 0.01;
    Y[i] = 20 * sin (X[i]);
  }
```

2.5 Lists

A **S-Lang** list is like an array except that it may contain a heterogeneous collection of data, e.g.,

```
my_list = { 3, 2.9, "foo", &sin };
```

is a list of four objects, each with a different type. Like an array, the elements of a list may be accessed via an index, e.g., `x=my_list[2]` will result in the assignment of `"foo"` to `x`. The most important difference between an array and a list is that an array's size is fixed whereas a list may grow or shrink. Algorithms that require such a data structure may execute many times faster when a list is used instead of an array.

2.6 Structures and User-Defined Types

A *structure* is similar to an array in the sense that it is a container object. However, the elements of an array must all be of the same type (or of `Any_Type`), whereas a structure is heterogeneous. As an example, consider

```
variable person = struct
{
   first_name, last_name, age
};
variable bill = @person;
bill.first_name = "Bill";
bill.last_name = "Clinton";
bill.age = 51;
```

In this example a structure consisting of the three fields has been created and assigned to the variable `person`. Then an *instance* of this structure has been created using the dereference operator and assigned to `bill`. Finally, the individual fields of `bill` were initialized. This is an example of an *anonymous* structure.

A *named* structure is really a new data type and may be created using the `typedef` keyword:

```
typedef struct
{
   first_name, last_name, age
}
Person_Type;

variable bill = @Person_Type;
bill.first_name = "Bill";
bill.last_name = "Clinton";
bill.age = 51;
```

One advantage of creating a new type is that array elements of such types are automatically initialized to instances of the type. For example,

```
People = Person_Type [100];
People[0].first_name = "Bill";
People[1].first_name = "Hillary";
```

may be used to create an array of 100 such objects and initialize the `first_name` fields of the first two elements. In contrast, the form using an anonymous would require a separate step to instantiate the array elements:

```
People = Struct_Type [100];
People[0] = @person;
People[0].first_name = "Bill";
People[1] = @person;
People[1].first_name = "Hillary";
```

Another big advantage of a user-defined type is that the binary and unary operators may be overloaded onto such types. This is explained in more detail below.

The creation and initialization of a structure may be facilitated by a function such as

```
define create_person (first, last, age)
{
    variable person = @Person_Type;
    person.first_name = first;
    person.last_name = last;
    person.age = age;
    return person;
}
variable Bill = create_person ("Bill", "Clinton", 51);
```

Other common uses of structures is the creation of linked lists, binary trees, etc. For more information about these and other features of structures, see the section on 12.3 (Linked Lists).

2.7 Namespaces

The language supports namespaces that may be used to control the scope and visibility of variables and functions. In addition to the global or public namespace, each **S-Lang** source file or compilation unit has a private or anonymous namespace associated with it. The private namespace may be used to define symbols that are local to the compilation unit and inaccessable from the outside. The language also allows the creation of named (non-anonymous or static) namespaces that permit access via the namespace operator. See the chapter on 9 (Namespaces) for more information.

Chapter 3

Data Types and Literal Constants

The current implementation of the **S-Lang** language permits up to 65535 distinct data types, including predefined data types such as integer and floating point, as well as specialized application-specific data types. It is also possible to create new data types in the language using the `typedef` mechanism.

Literal constants are objects such as the integer 3 or the string `"hello"`. The actual data type given to a literal constant depends upon the syntax of the constant. The following sections describe the syntax of literals of specific data types.

3.1 Predefined Data Types

The current version of **S-Lang** defines integer, floating point, complex, and string types. It also defines special purpose data types such as `Null_Type`, `DataType_Type`, and `Ref_Type`. These types are discussed below.

3.1.1 Integers

The **S-Lang** language supports both signed and unsigned characters, short integer, long integer, and long long integer types. On most 32 bit systems, there is no difference between an integer and a long integer; however, they may differ on 16 and 64 bit systems. Generally speaking, on a 16 bit system, plain integers are 16 bit quantities with a range of -32767 to 32767. On a 32 bit system, plain integers range from -2147483648 to 2147483647.

An plain integer *literal* can be specified in one of several ways:

- As a decimal (base 10) integer consisting of the characters 0 through 9, e.g., 127. An integer specified this way cannot begin with a leading 0. That is, 0127 is *not* the same as 127.
- Using hexadecimal (base 16) notation consisting of the characters 0 to 9 and `A` through `F`. The hexadecimal number must be preceded by the characters `0x`. For example, `0x7F` specifies an integer using hexadecimal notation and has the same value as decimal 127.
- In Octal notation using characters 0 through 7. The Octal number must begin with a leading 0. For example, 0177 and 127 represent the same integer.

Short, long, long long, and unsigned types may be specified by using the proper suffixes: `L` indicates that the integer is a long integer, `LL` indicates a long long integer, `h` indicates that the integer is a short integer, and `U` indicates that it is unsigned. For example, `1UL` specifies an unsigned long integer.

Finally, a character literal may be specified using a notation containing a character enclosed in single quotes as `'a'`. The value of the character specified this way will lie in the range 0 to 256 and will be determined by the ASCII value of the character in quotes. For example,

```
i = '0';
```

assigns to `i` the character 48 since the `'0'` character has an ASCII value of 48.

A "wide" character (unicode) may be specified using the form '\x{y...y}' where `y...y` are hexadecimal digits. For example,

```
'\x{12F}'         % Latin Small Letter I With Ogonek;
'\x{1D7BC}'       % Mathematical Sans-Serif Bold Italic Small Sigma
```

Any integer may be preceded by a minus sign to indicate that it is a negative integer.

3.1.2 Floating Point Numbers

Single and double precision floating point literals must contain either a decimal point or an exponent (or both). Here are examples of specifying the same double precision point number:

```
12.    12.0    12e0   1.2e1   120e-1   .12e2   0.12e2
```

Note that 12 is *not* a floating point number since it contains neither a decimal point nor an exponent. In fact, 12 is an integer.

One may append the `f` character to the end of the number to indicate that the number is a single precision literal. The following are all single precision values:

```
12.f    12.0f    12e0f   1.2e1f   120e-1f   .12e2f   0.12e2f
```

3.1.3 Complex Numbers

The language implements complex numbers as a pair of double precision floating point numbers. The first number in the pair forms the *real* part, while the second number forms the *imaginary* part. That is, a complex number may be regarded as the sum of a real number and an imaginary number.

Strictly speaking, the current implementation of the **S-Lang** does not support generic complex literals. However, it does support imaginary literals permitting a more generic complex number with a non-zero real part to be constructed from the imaginary literal via addition of a real number.

An imaginary literal is specified in the same way as a floating point literal except that `i` or `j` is appended. For example,

```
12i    12.0i   12e0j
```

all represent the same imaginary number.

A more generic complex number may be constructed from an imaginary literal via addition, e.g.,

```
3.0 + 4.0i
```

produces a complex number whose real part is `3.0` and whose imaginary part is `4.0`.

The intrinsic functions `Real` and `Imag` may be used to retrieve the real and imaginary parts of a complex number, respectively.

3.1.4 Strings

A string literal must be enclosed in double quotes as in:

```
"This is a string".
```

As described below, the string literal may contain a suffix that specifies how the string is to be interpreted, e.g., a string literal such as

```
"$HOME/.jedrc"$
```

with the '$' suffix will be subject to variable name expansion.

Although there is no imposed limit on the length of a string, string literals must be less than 256 characters in length. It is possible to construct strings longer than this by string concatenation, e.g.,

```
"This is the first part of a long string"
 + " and this is the second part"
```

Any character except a newline (ASCII 10) or the null character (ASCII 0) may appear explicitly in a string literal. However, these characters may embedded implicitly using the mechanism described below.

The backslash character is a special character and is used to include other special characters (such as a newline character) in the string. The special characters recognized are:

```
\"        --  double quote
\'        --  single quote
\\        --  backslash
\a        --  bell character (ASCII 7)
\t        --  tab character (ASCII 9)
\n        --  newline character (ASCII 10)
\e        --  escape character (ASCII 27)
\xhhh     --  byte expressed in HEXADECIMAL notation
\ooo      --  byte expressed in OCTAL notation
\dnnn     --  byte expressed in DECIMAL
\x{uuuu}  --  the Unicode character U+uuuu
```

For example, to include the double quote character as part of the string, it must be preceded by a backslash character, e.g.,

```
"This is a \"quote\""
```

Similarly, the next example illustrates how a newline character may be included:

```
"This is the first line\nand this is the second"
```

Suffixes

A string literal may be contain a suffix that specifies how the string is to be interpreted. The suffix may consist of one or more of the following characters:

R

Backslash substitution will not be performed on the string.

Q

Backslash substitution will be performed on the string. A string without a suffix is equivalent to one with the Q suffix.

B

If this suffix is present, the string will be interpreted as a binary string (BString_Type).

$

Variable name substitution will be performed on the string.

Not all combinations of the above controls characters are supported, nor make sense. For example, a string with the suffix `QR` will cause a parse-error because `Q` and `R` have opposing meanings.

The Q and R suffixes These suffixes turn on and off backslash expansion. Unless the `R` suffix is present, all string literals will have backslash substitution performed. Sometimes it is desirable not have such expansion. For example, pathnames on an MSDOS or Windows system use the backslash character as a path separator. The `R` prefix turns off backslash expansion, and as a result the following statements are equivalent:

```
file = "C:\\windows\\apps\\slrn.rc";
file = "C:\\windows\\apps\\slrn.rc"Q;
file = "C:\windows\apps\slrn.rc"R;
```

The only exception is that a backslash character is not permitted as the last character of a string with the `R` suffix. That is,

```
string = "This is illegal\"R;
```

is not permitted. Without this exception, a string such as

```
string = "Some characters: \"R, S, T\"";
```

would not be parsed properly.

The $ suffix If the string contains the `$` suffix, then variable name expansion will be performed upon names prefixed by a `$` character occuring within the string, e.g.,

```
"The value of X is $X and the value of Y is $Y"$.
```

with variable name substitution to be performed on the names `X` and `Y`. Such strings may be used as a convenient alternative to the `sprintf` function.

Name expansion is carried out according to the following rules: If the string literal occurs in a function, and the name corresponds to a variable local to the function, then the string representation of the value of that variable will be substituted. Otherwise, if the name corresponds to a variable that is local to the compilation unit (i.e., is declared as static or private), then its value's string representation will be used. Otherwise, if the name corresponds to a variable that exists as a global (public) then its value's string representation will be substituted. If the above searches fail and the name exists in the environment, then the value of the corresponding environment variable will be used. Otherwise, the variable will expand to the empty string.

Consider the following example:

```
private variable bar = "two";
putenv ("MYHOME=/home/baz");
define funct (foo)
{
  variable bar = 1;
  message ("file: $MYHOME/foo: garage=$MYGARAGE,bar=$bar"$);
}
```

When executed, this will produce the message:

```
file: /home/baz/foo: garage=,bar=1
```

assuming that `MYGARAGE` is not defined anywhere.

A name may be enclosed in braces. For example,

```
"${MYHOME}/foo: bar=${bar}"$
```

This is useful in cases when the name is followed immediately by other characters that may be interpreted as part of the name, e.g.,

```
variable HELLO="Hello ";
message ("${HELLO}World"$);
```

will produce the message "Hello World".

3.1.5 Null_Type

Objects of type `Null_Type` can have only one value: `NULL`. About the only thing that you can do with this data type is to assign it to variables and test for equality with other objects. Nevertheless, `Null_Type` is an important and extremely useful data type. Its main use stems from the fact that

since it can be compared for equality with any other data type, it is ideal to represent the value of an object which does not yet have a value, or has an illegal value.

As a trivial example of its use, consider

```
define add_numbers (a, b)
{
   if (a == NULL) a = 0;
   if (b == NULL) b = 0;
   return a + b;
}
variable c = add_numbers (1, 2);
variable d = add_numbers (1, NULL);
variable e = add_numbers (1,);
variable f = add_numbers (,);
```

It should be clear that after these statements have been executed, `c` will have a value of 3. It should also be clear that `d` will have a value of 1 because `NULL` has been passed as the second parameter. One feature of the language is that if a parameter has been omitted from a function call, the variable associated with that parameter will be set to `NULL`. Hence, `e` and `f` will be set to 1 and 0, respectively.

The `Null_Type` data type also plays an important role in the context of *structures*.

3.1.6 Ref_Type

Objects of `Ref_Type` are created using the unary *reference* operator `&`. Such objects may be *dereferenced* using the dereference operator `@`. For example,

```
sin_ref = &sin;
y = (@sin_ref) (1.0);
```

creates a reference to the `sin` function and assigns it to `sin_ref`. The second statement uses the dereference operator to call the function that `sin_ref` references.

The `Ref_Type` is useful for passing functions as arguments to other functions, or for returning information from a function via its parameter list. The dereference operator may also used to create an instance of a structure. For these reasons, further discussion of this important type can be found in the section on 8.4 (Referencing Variables).

3.1.7 Array_Type, List_Type, and Struct_Type

Variables of type `Array_Type`, `List_Type`, and `Struct_Type` are known as *container objects*. They are much more complicated than the simple data types discussed so far and each obeys a special syntax. For these reasons they are discussed in a separate chapters.

3.1.8 DataType_Type Type

S-Lang defines a type called `DataType_Type`. Objects of this type have values that are type names. For example, an integer is an object of type `Integer_Type`. The literals of `DataType_Type` include:

```
Char_Type           (signed character)
UChar_Type          (unsigned character)
Short_Type          (short integer)
UShort_Type         (unsigned short integer)
Integer_Type        (plain integer)
UInteger_Type       (plain unsigned integer)
Long_Type           (long integer)
ULong_Type          (unsigned long integer)
LLong_Type          (long long integer)
ULong_Type          (unsigned long long integer)
Float_Type          (single precision real)
Double_Type         (double precision real)
Complex_Type        (complex numbers)
String_Type         (strings, C strings)
BString_Type        (binary strings)
Struct_Type         (structures)
Ref_Type            (references)
Null_Type           (NULL)
Array_Type          (arrays)
List_Type           (lists)
DataType_Type       (data types)
```

as well as the names of any other types that an application defines.

The built-in function `typeof` returns the data type of its argument, i.e., a `DataType_Type`. For instance `typeof(7)` returns `Integer_Type` and `typeof(Integer_Type)` returns `DataType_Type`. One can use this function as in the following example:

```
if (Integer_Type == typeof (x)) message ("x is an integer");
```

The literals of `DataType_Type` have other uses as well. One of the most common uses of these literals is to create arrays, e.g.,

```
x = Complex_Type [100];
```

creates an array of 100 complex numbers and assigns it to `x`.

3.1.9 Boolean Type

Strictly speaking, **S-Lang** has no separate boolean type; rather it represents boolean values as `Char_Type` objects. In particular, boolean FALSE is equivalent to `Char_Type` 0, and TRUE as any non-zero `Char_Type` value. Since the exact value of TRUE is unspecfied, it is unnecessary and even pointless to define TRUE and FALSE literals in **S-Lang**.

3.2 Typecasting: Converting from one Type to Another

Occasionally, it is necessary to convert from one data type to another. For example, if you need to print an object as a string, it may be necessary to convert it to a `String_Type`. The `typecast` function may be used to perform such conversions. For example, consider

```
variable x = 10, y;
y = typecast (x, Double_Type);
```

After execution of these statements, `x` will have the integer value 10 and `y` will have the double precision floating point value `10.0`. If the object to be converted is an array, the `typecast` function will act upon all elements of the array. For example,

```
x = [1:10];       % Array of integers
y = typecast (x, Double_Type);
```

will create an array of 10 double precision values and assign it to `y`. One should also realize that it is not always possible to perform a typecast. For example, any attempt to convert an `Integer_Type` to a `Null_Type` will result in a run-time error. Typecasting works only when datatypes are similar.

Often the interpreter will perform implicit type conversions as necessary to complete calculations. For example, when multiplying an `Integer_Type` with a `Double_Type`, it will convert the `Integer_Type` to a `Double_Type` for the purpose of the calculation. Thus, the example involving the conversion of an array of integers to an array of doubles could have been performed by multiplication by `1.0`, i.e.,

```
x = [1:10];       % Array of integers
y = 1.0 * x;
```

The `string` intrinsic function should be used whenever a string representation is needed. Using the `typecast` function for this purpose will usually fail unless the object to be converted is similar to a string— most are not. Moreover, when typecasting an array to `String_Type`, the `typecast` function acts on each element of the array to produce another array, whereas the `string` function will produce a string.

One use of `string` function is to print the value of an object. This use is illustrated in the following simple example:

```
define print_object (x)
{
   message (string (x));
}
```

Here, the `message` function has been used because it writes a string to the display. If the `string` function was not used and the `message` function was passed an integer, a type-mismatch error would have resulted.

Chapter 4

Identifiers

The names given to variables, functions, and data types are called *identifiers*. There are some restrictions upon the actual characters that make up an identifier. An identifier name must start with an alphabetic character (`[A-Za-z]`), an underscore character, or a dollar sign. The rest of the characters in the name can be any combination of letters, digits, dollar signs, or underscore characters. However, all identifiers whose name begins with two underscore characters are reserved for internal use by the interpreter and declarations of objects with such names should be avoided.

Examples of valid identifiers include:

```
mary    _3    _this_is_ok
a7e1    $44   _44$_Three
```

However, the following are not legal:

```
7abc   2e0    #xx
```

In fact, `2e0` actually specifies the double precision number `2.0`.

There is no limit to the maximum length of an identifier. For practical usage it is wise to limit the length of identifiers to a reasonable value.

The following identifiers are reserved by the language for use as keywords:

```
abs           and           andelse       break         case
catch         chs           continue      define        do
do_while      else          ERROR_BLOCK   exch          EXIT_BLOCK
finally       _for          for           foreach       forever
!if           if            loop          mod           mul2
not           or            orelse        pop           private
public        return        shl           shr           sign
sqr           static        struct        switch        __tmp
throw         try           typedef       USER_BLOCK1   USER_BLOCK2
USER_BLOCK0   USER_BLOCK4   USER_BLOCK3   using         variable
while         xor
```

Chapter 5

Variables

As many of the preceeding examples have shown, a variable must be declared before it can be used, otherwise an undefined name error will be generated. A variable is declared using the `variable` keyword, e.g,

```
variable x, y, z;
```

declares three variables, `x`, `y`, and `z`. This is an example of a variable declaration statement, and like all statements, it must end in a semi-colon.

Variables declared this way are untyped and inherit a type upon assignment. As such, type-checking of function arguments, etc is performed at run-time. For example,

```
x = "This is a string";
x = 1.2;
x = 3;
x = 2i;
```

results in x being set successively to a string, a float, an integer, and to a complex number (`0+2i`). Any attempt to use a variable before it has acquired a type will result in an uninitialized variable error.

It is legal to put executable code in a variable declaration list. That is,

```
variable x = 1, y = sin (x);
```

are legal variable declarations. This also provides a convenient way of initializing a variable.

Variables are classified as either *global* or *local*. A variable declared inside a function is said to be local and has no meaning outside the function. A variable is said to be global if it was declared outside a function. Global variables are further classified as being `public`, `static`, or `private`, according to the namespace where they were defined. See the chapter on 9 (Namespaces) for more information about namespaces.

The following global variables are predefined by the language and live in the `public` namespace. They are mainly used as convenience variables:

```
$0 $1 $2 $3 $4 $5 $6 $7 $8 $9
```

An *intrinsic* variable is another type of global variable. Such variables have a definite type which cannot be altered. Variables of this type may also be defined to be read-only, or constant variables. An example of an intrinsic variable is `PI` which is a read-only double precision variable with a value of approximately `3.14159265358979323846`.

Chapter 6

Operators

S-Lang supports a variety of operators that are grouped into three classes: assignment operators, binary operators, and unary operators.

An assignment operator is used to assign a value to a variable. They will be discussed more fully in the context of the assignment statement in the section on 7.2 (Assignment Statements).

An unary operator acts only upon a single quantity while a binary operation is an operation between two quantities. The boolean operator `not` is an example of an unary operator. Examples of binary operators include the usual arithmetic operators `+`, `-`, `*`, and `/`. The operator given by `-` can be either an unary operator (negation) or a binary operator (subtraction); the actual operation is determined from the context in which it is used.

Binary operators are used in algebraic forms, e.g., `a + b`. Unary operators fall into one of two classes: postfix-unary or prefix-unary. For example, in the expression `-x`, the minus sign is a prefix-unary operator.

All binary and unary operators may be defined for any supported data type. For example, the arithmetic plus operator has been extended to the `String_Type` data type to permit concatenation between strings. But just because it is possible to define the action of an operator upon a data type, it does not mean that all data types support all the binary and unary operators. For example, while `String_Type` supports the `+` operator, it does not admit the `*` operator.

6.1 Unary Operators

The **unary** operators operate only upon a single operand. They include: `not`, `~`, `-`, `@`, `&`, as well as the increment and decrement operators `++` and `--`, respectively.

The boolean operator `not` acts only upon integers and produces `0` if its operand is non-zero, otherwise it produces 1.

The bit-level not operator `~` performs a similar function, except that it operates on the individual bits of its integer operand.

The arithmetic negation operator `-` is perhaps the most well-known unary operator. It simply reverses the sign of its operand.

The reference (`&`) and dereference (`@`) operators will be discussed in greater detail in the section on 8.4 (Referencing Variables). Similarly, the increment (`++`) and decrement (`--`) operators will be discussed in the context of the assignment operator.

6.2 Binary Operators

The binary operators may be grouped according to several classes: arithmetic operators, relational operators, boolean operators, and bitwise operators.

6.2.1 Arithmetic Operators

The arithmetic operators include `+`, `-`, `*`, and `/`, which perform addition, subtraction, multiplication, and division, respectively. In addition to these, **S-Lang** supports the `mod` operator, which divides two numbers and produces the remainder, as as well as the power operator `^`.

The data type of the result produced by the use of one of these operators depends upon the data types of the binary participants. If they are both integers, the result will be an integer. However, if the operands are not of the same type, they will be converted to a common type before the operation is performed. For example, if one is a floating point type and the other is an integer, the integer will be converted to a float. In general, the promotion from one type to another is such that no information is lost, if possible. As an example, consider the expression `8/5` which indicates division of the integer 8 by the integer 5. The result will be the integer 1 and *not* the floating point value `1.6`. However, `8/5.0` will produce `1.6` because `5.0` is a floating point number.

6.2.2 Relational Operators

The relational operators are `>`, `>=`, `<`, `<=`, `==`, and `!=`. These perform the comparisons greater than, greater than or equal, less than, less than or equal, equal, and not equal, respectively. For most data types, the result of the comparison will be a boolean value; however, for arrays the result will be an array of boolean values. The section on arrays will explain this is greater detail.

6.2.3 Boolean Operators

S-Lang supports two boolean binary operators: `or` and `and`, which for most data types, return a boolean result. In particular, the `or` operator returns a non-zero value (boolean TRUE) if either of its operands are non-zero, otherwise it produces zero (boolean FALSE). The `and` operator produces a non-zero value if and only if both its operands are non-zero, otherwise it produces zero. If either of the operands is an array then a corresponding array of boolean values will result. This is explained in more detail in the section on arrays.

Neither of these operators perform the so-called boolean short-circuit evaluation. For example, consider the expression:

```
(x != 0) and (1/x > 10)
```

Here, if `x` were to have a value of zero, a division by zero error would occur because even though `x!=0` evaluates to zero, the `and` operator is not short-circuited and the `1/x` expression would still be evaluated. Although these operators are not short-circuited, **S-Lang** does have another mechanism of performing short-circuit boolean evaluation via the `orelse` and `andelse` expressions. See below for information about these constructs.

6.2.4 Bitwise Operators

The bitwise binary operators are currently defined for integer operands and are used for bit-level operations. Operators that fall in this class include `&`, |, `shl`, `shr`, and `xor`. The `&` operator performs a boolean AND operation between the corresponding bits of the operands. Similarly, the | operator performs the boolean OR operation on the bits. The bit-shifting operators `shl` and `shr` shift the bits of the first operand by the number given by the second operand to the left or right, respectively. Finally, the `xor` performs an EXCLUSIVE-OR operation.

These operators are commonly used to manipulate variables whose individual bits have distinct meanings. In particular, `&` is usually used to test bits, | can be used to set bits, and `xor` may be used to flip a bit.

As an example of using `&` to perform tests on bits, consider the following: The **jed** text editor stores some of the information about a buffer in a bitmapped integer variable. The value of this variable may be retrieved using the **jed** intrinsic function `getbuf_info`, which actually returns four quantities: the buffer flags, the name of the buffer, directory name, and file name. For the purposes of this section, only the buffer flags are of interest and can be retrieved via a function such as

```
define get_buffer_flags ()
{
   variable flags;
   (,,,flags) = getbuf_info ();
   return flags;
}
```

The buffer flags object is a bitmapped quantity where the 0th bit indicates whether or not the buffer has been modified, the first bit indicates whether or not autosave has been enabled for the buffer, and so on. Consider for the moment the task of determining if the buffer has been modified. This can be determined by looking at the zeroth bit: if it is 0 the buffer has not been modified, otherwise it has been modified. Thus we can create the function,

```
define is_buffer_modified ()
{
   variable flags = get_buffer_flags ();
   return (flags & 1);
}
```

where the integer 1 has been used since it is represented as an object with all bits unset, except for the zeroth one, which is set. (At this point, it should also be apparent that bits are numbered from zero, thus an 8 bit integer consists of bits 0 to 7, where 0 is the least significant bit and 7 is the most significant one.) Similarly, we can create another function

```
define is_autosave_on ()
{
   variable flags = get_buffer_flags ();
   return (flags & 2);
}
```

to determine whether or not autosave has been turned on for the buffer.

The `shl` operator may be used to form the integer with only the *nth* bit set. For example, `1 shl 6` produces an integer with all bits set to zero except the sixth bit, which is set to one. The following example exploits this fact:

```
define test_nth_bit (flags, nth)
{
   return flags & (1 shl nth);
}
```

6.2.5 The Namespace Operator

The operator `->` is used to in conjunction with a namespace to access an object within the namespace. For example, if `A` is the name of a namespace containing the variable `v`, then `A->v` refers to that variable. Namespaces are discussed more fully in the chapter on 9 (Namespaces).

6.2.6 Operator Precedence

6.2.7 Binary Operators and Functions Returning Multiple Values

Care must be exercised when using binary operators with an operand that returns multiple values. In fact, the current implementation of the **S-Lang** language will produce incorrect results if both operands of a binary expression return multiple values. *At most, only one of operands of a binary expression can return multiple values, and that operand must be the first one, not the second.* For example,

```
define read_line (fp)
{
   variable line, status;

   status = fgets (&line, fp);
   if (status == -1)
     return -1;
   return (line, status);
}
```

defines a function, `read_line` that takes a single argument specifying a handle to an open file, and returns one or two values, depending upon the return value of `fgets`. Now consider

```
 while (read_line (fp) > 0)
   {
      text = ();
```

```
   % Do something with text
   .
   .
}
```

Here the relational binary operator `>` forms a comparison between one of the return values (the one at the top of the stack) and 0. In accordance with the above rule, since `read_line` returns multiple values, it must occur as the left binary operand. Putting it on the right as in

```
while (0 < read_line (fp))    % Incorrect
  {
     text = ();
     % Do something with text
     .
     .
  }
```

violates the rule and will result in the wrong answer. For this reason, one should avoid using a function that returns muliple return values as a binary operand.

6.3 Mixing Integer and Floating Point Arithmetic

If a binary operation (`+`, `-`, `*` , `/`) is performed on two integers, the result is an integer. If at least one of the operands is a floating point value, the other will be converted to a floating point value, and a floating point result be produced. For example:

```
11 / 2          --> 5   (integer)
11 / 2.0        --> 5.5 (double)
11.0 / 2        --> 5.5 (double)
11.0 / 2.0      --> 5.5 (double)
```

Sometimes to achive the desired result, it is necessary to explicitly convert from one data type to another. For example, suppose that `a` and `b` are integers, and that one wants to compute `a/b` using floating point arithmetic. In such a case, it is necessary to convert at least one of the operands to a floating point value using, e.g., the `double` function:

```
x = a/double(b);
```

6.4 Short Circuit Boolean Evaluation

The boolean operators `or` and `and` *are not short circuited* as they are in some languages. **S-Lang** uses `orelse` and `andelse` expressions for short circuit boolean evaluation. However, these are not binary operators. Expressions of the form:

expr-1 `and` *expr-2* `and ...` *expr-n*

can be replaced by the short circuited version using `andelse`:

```
andelse {expr-1} {expr-2} ...  {expr-n}
```

A similar syntax holds for the `orelse` operator. For example, consider the statement:

```
if ((x != 0) and (1/x > 10)) do_something ();
```

Here, if `x` were to have a value of zero, a division by zero error would occur because even though `x!=0` evaluates to zero, the `and` operator is not short circuited and the `1/x` expression would be evaluated causing division by zero. For this case, the `andelse` expression could be used to avoid the problem:

```
if (andelse
    {x != 0}
    {1 / x > 10})  do_something ();
```

Chapter 7

Statements

Loosely speaking, a *statement* is composed of *expressions* that are grouped according to the syntax or grammar of the language to express a complete computation. A semi-colon is used to denote the the end of a statement.

A statement that occurs within a function is executed only during execution of the function. However, statements that occur outside the context of a function are evaluated immediately.

The language supports several different types of statements such as assignment statements, conditional statements, and so forth. These are described in detail in the following sections.

7.1 Variable Declaration Statements

Variable declarations were already discussed in the chapter on 5 (Variables). For the sake of completeness, a variable declaration is a statement of the form

```
variable variable-declaration-list ;
```

where the *variable-declaration-list* is a comma separated list of one or more variable names with optional initializations, e.g.,

```
variable x, y = 2, z;
```

7.2 Assignment Statements

Perhaps the most well known form of statement is the *assignment statement*. Statements of this type consist of a left-hand side, an assignment operator, and a right-hand side. The left-hand side must be something to which an assignment can be performed. Such an object is called an *lvalue*.

The most common assignment operator is the simple assignment operator `=`. Examples of its use include

```
x = 3;
```

```
x = some_function (10);
x = 34 + 27/y + some_function (z);
x = x + 3;
```

In addition to the simple assignment operator, **S-Lang** also supports the binary assignment operators:

```
+=   -=   *=    /=   &=   |=
```

Internally, **S-Lang** transforms

```
a += b;
```

to

```
a = a + b;
```

Likewise `a-=b` is transformed to `a=a-b`, `a*=b` is transformed to `a=a*b`, and so on.

It is extremely important to realize that, in general, `a+b` is not equal to `b+a`. For example if `a` and `b` are strings, then `a+b` will be the string resulting from the concatenation of `a` and `b`, which generally is not he same as the concatenation of `b` with `a`. This means that `a+=b` may not be the same as `a=b+a`, as the following example illustrates:

```
a = "hello"; b = "world";
a += b;                      % a will become "helloworld"
c = b + a;                   % c will become "worldhello"
```

Since adding or subtracting 1 from a variable is quite common, **S-Lang** also supports the unary increment and decrement operators `++`, and `--`, respectively. That is, for numeric data types,

```
x = x + 1;
x += 1;
x++;
```

are all equivalent. Similarly,

```
x = x - 1;
x -= 1;
x--;
```

are also equivalent.

Strictly speaking, `++` and `--` are unary operators. When used as `x++`, the `++` operator is said to be a *postfix-unary* operator. However, when used as `++x` it is said to be a *prefix-unary* operator. The current implementation does not distinguish between the two forms, thus `x++` and `++x` are equivalent. The reason for this equivalence is *that assignment expressions do not return a value in the* ***S-Lang*** *language* as they do in C. Thus one should exercise care and not try to write C-like code such as

```
x = 10;
while (--x) do_something (x);     % Ok in C, but not in S-Lang
```

The closest valid **S-Lang** form involves a *comma-expression*:

```
x = 10;
while (x--, x) do_something (x);  % Ok in S-Lang and in C
```

S-Lang also supports a *multiple-assignment* statement. It is discussed in detail in the section on 8.7 (Multiple Assignment Statement).

7.3 Conditional and Looping Statements

S-Lang supports a wide variety of conditional and looping statements. These constructs operate on statements grouped together in *blocks*. A block is a sequence of **S-Lang** statements enclosed in braces and may contain other blocks. However, a block cannot include function declarations. In the following, *statement-or-block* refers to either a single **S-Lang** statement or to a block of statements, and *integer-expression* is an integer-valued or boolean expression. *next-statement* represents the statement following the form under discussion.

7.3.1 Conditional Forms

if

The simplest condition statement is the `if` statement. It follows the syntax

```
if (integer-expression) statement-or-block next-statement
```

If *integer-expression* evaluates to a non-zero (boolean TRUE) result, then the statement or group of statements implied *statement-or-block* will get executed. Otherwise, control will proceed to *next-statement*.

An example of the use of this type of conditional statement is

```
if (x != 0)
  {
     y = 1.0 / x;
     if (x > 0) z = log (x);
  }
```

This example illustrates two `if` statements where the second `if` statement is part of the block of statements that belong to the first.

if-else

Another form of `if` statement is the *if-else* statement. It follows the syntax:

```
if (integer-expression) statement-or-block-1 else statement-or-block-2
next-statement
```

Here, if *expression* evaluates to a non-zero integer, *statement-or-block-1* will get executed and control will pass on to *next-statement*. However, if *expression* evaluates to zero, *statement-or-block-2* will get executed before continuing on to *next-statement*. A simple example of this form is

```
if (x > 0)
  z = log (x);
else
  throw DomainError, "x must be positive";
```

Consider the more complex example:

```
if (city == "Boston")
  if (street == "Beacon") found = 1;
else if (city == "Madrid")
  if (street == "Calle Mayor") found = 1;
else found = 0;
```

This example illustrates a problem that beginners have with *if-else* statements. Syntactically, this example is equivalent to

```
if (city == "Boston")
  {
    if (street == "Beacon") found = 1;
    else if (city == "Madrid")
      {
        if (street == "Calle Mayor") found = 1;
        else found = 0;
      }
  }
```

although the indentation indicates otherwise. It is important to understand the grammar and not be seduced by the indentation!

!if

One often encounters `if` statements similar to

`if (`*integer-expression* `== 0)` *statement-or-block*

or equivalently,

`if (not(`*integer-expression*`))` *statement-or-block*

The `!if` statement was added to the language to simplify the handling of such statements. It obeys the syntax

`!if (`*integer-expression*`)` *statement-or-block*

and is functionally equivalent to

`if (not (`*expression*`))` *statement-or-block*

orelse, andelse

These constructs were discussed earlier. The syntax for the `orelse` statement is:

`orelse` {*integer-expression-1*} ... {*integer-expression-n*}

This causes each of the blocks to be executed in turn until one of them returns a non-zero integer value. The result of this statement is the integer value returned by the last block executed. For example,

```
orelse { 0 } { 6 } { 2 } { 3 }
```

returns 6 since the second block is the first to return a non-zero result. The last two block will not get executed.

The syntax for the `andelse` statement is:

`andelse` {*integer-expression-1*} ... {*integer-expression-n*}

Each of the blocks will be executed in turn until one of them returns a zero value. The result of this statement is the integer value returned by the last block executed. For example,

```
andelse { 6 } { 2 } { 0 } { 4 }
```

evaluates to 0 since the third block will be the last to execute.

switch

The switch statement deviates from its C counterpart. The syntax is:

```
switch (x)
  { ...  :  ...}
    .
    .
  { ...  :  ...}
```

The ':' operator is a special symbol that in the context of the switch statement, causes the the top item on the stack to be tested, and if it is non-zero, the rest of the block will get executed and control will pass out of the switch statement. Otherwise, the execution of the block will be terminated and the process will be repeated for the next block. If a block contains no : operator, the entire block is executed and control will pass onto the next statement following the `switch` statement. Such a block is known as the *default* case.

As a simple example, consider the following:

```
switch (x)
  { x == 1 : message("Number is one.");}
  { x == 2 : message("Number is two.");}
  { x == 3 : message("Number is three.");}
  { x == 4 : message("Number is four.");}
  { x == 5 : message("Number is five.");}
  { message ("Number is greater than five.");}
```

Suppose `x` has an integer value of 3. The first two blocks will terminate at the ':' character because each of the comparisons with `x` will produce zero. However, the third block will execute to completion. Similarly, if `x` is 7, only the last block will execute in full.

A more familiar way to write the previous example is to make use of the `case` keyword:

```
switch (x)
  { case 1 : message("Number is one.");}
  { case 2 : message("Number is two.");}
  { case 3 : message("Number is three.");}
  { case 4 : message("Number is four.");}
  { case 5 : message("Number is five.");}
  { message ("Number is greater than five.");}
```

The `case` keyword is a more useful comparison operator because it can perform a comparison between different data types while using `==` may result in a type-mismatch error. For example,

```
switch (x)
  { (x == 1) or (x == "one") : message("Number is one.");}
  { (x == 2) or (x == "two") : message("Number is two.");}
  { (x == 3) or (x == "three") : message("Number is three.");}
  { (x == 4) or (x == "four") : message("Number is four.");}
  { (x == 5) or (x == "five") : message("Number is five.");}
  { message ("Number is greater than five.");}
```

will fail because the `==` operation is not defined between strings and integers. The correct way to write this is to use the `case` keyword:

```
switch (x)
  { case 1 or case "one" : message("Number is one.");}
  { case 2 or case "two" : message("Number is two.");}
  { case 3 or case "three" : message("Number is three.");}
  { case 4 or case "four" : message("Number is four.");}
  { case 5 or case "five" : message("Number is five.");}
  { message ("Number is greater than five.");}
```

7.3.2 Looping Forms

while

The `while` statement follows the syntax

`while (`*integer-expression*`)` *statement-or-block next-statement*

It simply causes *statement-or-block* to get executed as long as *integer-expression* evaluates to a non-zero result. For example,

```
i = 10;
while (i)
  {
```

```
    i--;
    newline ();
  }
```

will cause the `newline` function to get called 10 times. However,

```
i = -10;
while (i)
  {
    i--;
    newline ();
  }
```

would loop forever (or until `i` wraps from the most negative integer value to the most positive and then decrements to zero).

If you are a C programmer, do not let the syntax of the language seduce you into writing this example as you would in C:

```
i = 10;
while (i--) newline ();
```

Keep in mind that expressions such as `i--` do not return a value in **S-Lang** as they do in C. The same effect can be achieved to use a comma to separate the expressions as as in

```
i = 10;
while (i, i--) newline ();
```

do...while

The `do...while` statement follows the syntax

```
do statement-or-block while (integer-expression);
```

The main difference between this statement and the `while` statement is that the `do...while` form performs the test involving *integer-expression* after each execution of *statement-or-block* rather than before. This guarantees that *statement-or-block* will get executed at least once.

A simple example from the **jed** editor follows:

```
bob ();      % Move to beginning of buffer
do
  {
    indent_line ();
  }
while (down (1));
```

This will cause all lines in the buffer to get indented via the **jed** intrinsic function `indent_line`.

for

Perhaps the most complex looping statement is the `for` statement; nevertheless, it is a favorite of many C programmers. This statement obeys the syntax

```
for (init-expression; integer-expression; end-expression) statement-or-block
next-statement
```

In addition to *statement-or-block*, its specification requires three other expressions. When executed, the `for` statement evaluates *init-expression*, then it tests *integer-expression*. If *integer-expression* evaluates to zero, control passes to *next-statement*. Otherwise, it executes *statement-or-block* as long as *integer-expression* evaluates to a non-zero result. After every execution of *statement-or-block*, *end-expression* will get evaluated.

This statement is *almost* equivalent to

```
init-expression; while (integer-expression) { statement-or-block
end-expression; }
```

The reason that they are not fully equivalent involves what happens when *statement-or-block* contains a `continue` statement.

Despite the apparent complexity of the `for` statement, it is very easy to use. As an example, consider

```
s = 0;
for (i = 1; i <= 10; i++) s += i;
```

which computes the sum of the first 10 integers.

loop

The `loop` statement simply executes a block of code a fixed number of times. It follows the syntax

```
loop (integer-expression) statement-or-block next-statement
```

If the *integer-expression* evaluates to a positive integer, *statement-or-block* will get executed that many times. Otherwise, control will pass to *next-statement*.

For example,

```
loop (10) newline ();
```

will execute the `newline` function 10 times.

_for

Like `loop`, the `_for` statement simply executes a block of code a fixed number times. Unlike the `loop` statement, the `_for` loop is useful in situations where the loop index is needed. It obeys the syntax

```
_for loop-variable (first-value , last-value , increment) block next-statement
```

Each time through the loop, the loop-variable will take on the successive values dictated by the other parameters. The first time through, the loop-variable will have the value of *first-value*. The second time its value will be *first-value* + *increment*, and so on. The loop will terminate when the value of the loop index exceeds *last-value*. The current implementation requires the control parameters *first-value*, *last-value*, and *increment* to be integer-valued expressions.

For example, the `_for` statement may be used to compute the sum of the first ten integers:

```
s = 0;
_for i (1, 10, 1)
  s += i;
```

The execution speed of the `_for` loop is more than twice as fast as the more powerful `for` loop making it a better choice for many situations.

forever

The `forever` statement is similar to the `loop` statement except that it loops forever, or until a `break` or a `return` statement is executed. It obeys the syntax

```
forever statement-or-block
```

A trivial example of this statement is

```
n = 10;
forever
  {
     if (n == 0) break;
     newline ();
     n--;
  }
```

foreach

The `foreach` statement is used to loop over one or more statements for every element of an object. Most often the object will be a container object such as an array, structure, or associative arrays, but it need not be.

The simple type of `foreach` statement obeys the syntax

```
foreach var (object) statement-or-block
```

Here *object* can be an expression that evaluates to a value. Each time through the loop the variable *var* will take on a value that depends upon the data type of the object being processed. For container objects, *var* will take on values of successive members of the object.

A simple example is

```
foreach fruit (["apple", "peach", "pear"])
  process_fruit (fruit);
```

This example shows that if the container object is an array, then successive elements of the array are assigned to `fruit` prior to each execution cycle. If the container object is a string, then successive characters of the string are assigned to the variable.

What actually gets assigned to the variable may be controlled via the `using` form of the `foreach` statement. This more complex type of `foreach` statement follows the syntax

`foreach` *var* `(` *container-object* `) using (` *control-list* `)` *statement-or-block*

The allowed values of *control-list* will depend upon the type of container object. For associative arrays (`Assoc_Type`), *control-list* specifies whether *keys*, *values*, or both are used. For example,

```
foreach k (a) using ("keys")
  {
      .
      .
  }
```

results in the keys of the associative array `a` being successively assigned to `k`. Similarly,

```
foreach v (a) using ("values")
  {
      .
      .
  }
```

will cause the values to be used. The form

```
foreach k,v (a) using ("keys", "values")
  {
      .
      .
  }
```

may be used when both keys and values are desired.

Similarly, for linked-lists of structures, one may walk the list via code like

```
foreach s (linked_list) using ("next")
  {
       .
       .
  }
```

This `foreach` statement is equivalent

```
s = linked_list;
while (s != NULL)
```

```
{
   .
   .
  s = s.next;
}
```

Consult the type-specific documentation for a discussion of the `using` control words, if any, appropriate for a given type.

7.4 break, return, and continue

S-Lang also includes the non-local transfer statements `return`, `break`, and `continue`. The `return` statement causes control to return to the calling function while the `break` and `continue` statements are used in the context of loop structures. Consider:

```
define fun ()
{
   forever
     {
        s1;
        s2;
        ..
        if (condition_1) break;
        if (condition_2) return;
        if (condition_3) continue;
        ..
        s3;
     }
   s4;
   ..
}
```

Here, a function `fun` has been defined that contains a `forever` loop consisting of statements `s1`, `s2`,...,`s3`, and three `if` statements. As long as the expressions `condition_1`, `condition_2`, and `condition_3` evaluate to zero, the statements `s1`, `s2`,...,`s3` will be repeatedly executed. However, if `condition_1` returns a non-zero value, the `break` statement will get executed, and control will pass out of the `forever` loop to the statement immediately following the loop, which in this case is `s4`. Similarly, if `condition_2` returns a non-zero number, the `return` statement will cause control to pass back to the caller of `fun`. Finally, the `continue` statement will cause control to pass back to the start of the loop, skipping the statement `s3` altogether.

Chapter 8

Functions

There are essentially two classes of functions that may be called from the interpreter: intrinsic functions and slang functions.

An intrinsic function is one that is implemented in C or some other compiled language and is callable from the interpreter. Nearly all of the built-in functions are of this variety. At the moment the basic interpreter provides nearly 300 intrinsic functions. Examples include the trigometric functions `sin` and `cos`, string functions such as `strcat`, etc. Dynamically loaded modules such as the `png` and `pcre` modules add additional intrinsic functions.

The other type of function is written in **S-Lang** and is known simply as a "**S-Lang** function". Such a function may be thought of as a group of statements that work together to perform a computation. The specification of such functions is the main subject of this chapter.

8.1 Calling Functions

The most important rule to remember in calling a function is that *if the function returns a value, do something with it*. While this might sound like a trivial statement it is the number one issue that trips-up novice users of the language.

To elaborate on this point further, consider the *fputs* function, which writes a a string to a file descriptor. This function can fail when, e.g., a disk is full, or the file is located on a network share and the network goes down, etc.

S-Lang supports two mechanisms that a function may use to report a failure: raising an exception, returning a status code. The latter mechanism is used by the **S-Lang** fputs function. i.e., it returns a value to indicate whether or not is was successful. Many users familiar with this function either seem to forget this fact, or assume that the function will succeed and not bother handling the return value. While some languages silently remove such values from the stack, **S-Lang** regards the stack as a dynamic data structure that programs can utilize. As a result, the value will be left on the **S-Lang** stack and can cause problems later on.

There are a number of correct ways of "doing something" with the return value from a function. Of course the recommended procedure is to use the return value as it was meant to be used. In the case of `fputs`, the proper thing to do is to check the return value, e.g.,

```
if (-1 == fputs ("good luck", fp))
  {
     % Handle the error
  }
```

Other acceptable ways to "do something" with the return value include assigning it to a dummy variable,

```
dummy = fputs ("good luck", fp);
```

or simply "popping" it from the stack:

```
fputs ("good luck", fp);  pop();
```

The latter mechanism can also be written as

```
() = fputs ("good luck", fp);
```

The last form is a special case of the *multiple assignment statement*, which is discussed in more detail below. Since this form is simpler than assigning the value to a dummy variable or explicitly calling the `pop` function, it is recommended over the other two mechanisms. Finally, this form has the redeeming feature that it presents a visual reminder that the function is returning a value that is not being used.

8.2 Declaring Functions

Like variables, functions must be declared before they can be used. The `define` keyword is used for this purpose. For example,

```
define factorial ();
```

is sufficient to declare a function named `factorial`. Unlike the `variable` keyword used for declaring variables, the `define` keyword does not accept a list of names.

Usually, the above form is used only for recursive functions. In most cases, the function name is almost always followed by a parameter list and the body of the function:

`define` *function-name* `(`*parameter-list*`)` { *statement-list* }

The *function-name* is an identifier and must conform to the naming scheme for identifiers discussed in the chapter on 4 (Identifiers). The *parameter-list* is a comma-separated list of variable names that represent parameters passed to the function, and may be empty if no parameters are to be passed. The variables in the *parameter-list* are implicitly declared, thus, there is no need to declare them via a variable declaration statement. In fact any attempt to do so will result in a syntax error.

The body of the function is enclosed in braces and consists of zero or more statements (*statement-list*). While there are no imposed limits upon the number statements that may occur within a **S-Lang** function, it is considered poor programming practice if a function contains many statements. This notion stems from the belief that a function should have a simple, well defined purpose.

8.3 Parameter Passing Mechanism

Parameters to a function are always passed by value and never by reference. To see what this means, consider

```
define add_10 (a)
{
   a = a + 10;
}
variable b = 0;
add_10 (b);
```

Here a function `add_10` has been defined, which when executed, adds 10 to its parameter. A variable `b` has also been declared and initialized to zero before being passed to `add_10`. What will be the value of `b` after the call to `add_10`? If **S-Lang** were a language that passed parameters by reference, the value of `b` would be changed to 10. However, **S-Lang** always passes by value, which means that `b` will retain its value during and after after the function call.

S-Lang does provide a mechanism for simulating pass by reference via the reference operator. This is described in greater detail in the next section.

If a function is called with a parameter in the parameter list omitted, the corresponding variable in the function will be set to `NULL`. To make this clear, consider the function

```
define add_two_numbers (a, b)
{
   if (a == NULL) a = 0;
   if (b == NULL) b = 0;
   return a + b;
}
```

This function must be called with two parameters. However, either of them may omitted by calling the function in one of the following ways:

```
variable s = add_two_numbers (2,3);
variable s = add_two_numbers (2,);
variable s = add_two_numbers (,3);
variable s = add_two_numbers (,);
```

The first example calls the function using both parameters, but at least one of the parameters was omitted in the other examples. If the parser recognizes that a parameter has been omitted by finding a comma or right-parenthesis where a value is expected, it will substitute `NULL` for missing value. This means that the parser will convert the latter three statements in the above example to:

```
variable s = add_two_numbers (2, NULL);
variable s = add_two_numbers (NULL, 3);
variable s = add_two_numbers (NULL, NULL);
```

It is important to note that this mechanism is available only for function calls that specify more than one parameter. That is,

```
variable s = add_10 ();
```

is *not* equivalent to `add_10(NULL)`. The reason for this is simple: the parser can only tell whether or not `NULL` should be substituted by looking at the position of the comma character in the parameter list, and only function calls that indicate more than one parameter will use a comma. A mechanism for handling single parameter function calls is described later in this chapter.

8.4 Referencing Variables

One can achieve the effect of passing by reference by using the reference (`&`) and dereference (`@`) operators. Consider again the `add_10` function presented in the previous section. This time it is written as:

```
define add_10 (a)
{
   @a = @a + 10;
}
variable b = 0;
add_10 (&b);
```

The expression `&b` creates a *reference* to the variable `b` and it is the reference that gets passed to `add_10`. When the function `add_10` is called, the value of the local variable `a` will be a reference to the variable `b`. It is only by *dereferencing* this value that `b` can be accessed and changed. So, the statement `@a=@a+10` should be read as "add 10 to the value of the object that `a` references and assign the result to the object that `a` references".

The reader familiar with C will note the similarity between *references* in **S-Lang** and *pointers* in C.

References are not limited to variables. A reference to a function may also be created and passed to other functions. As a simple example from elementary calculus, consider the following function which returns an approximation to the derivative of another function at a specified point:

```
define derivative (f, x)
{
   variable h = 1e-6;
   return ((@f)(x+h) - (@f)(x)) / h;
}
define x_squared (x)
{
   return x^2;
}
dydx = derivative (&x_squared, 3);
```

When the `derivative` function is called, the local variable `f` will be a reference to the `x_squared` function. The `x_squared` function is called is called with the specified parameters by dereferencing `f` with the dereference operator.

8.5 Functions with a Variable Number of Arguments

When a **S-Lang** function is called with parameters, those parameters are placed on the run-time stack. The function accesses those parameters by removing them from the stack and assigning them to the variables in its parameter list. This details of this operation are for the most part hidden from the programmer. But what happens when the number of parameters in the parameter list is not equal to the number of parameters passed to the function? If the number passed to the function is less than what the function expects, a `StackUnderflow` error could result as the function tries to remove items from the stack. If the number passed is greater than the number in the parameter list, then the extras will remain on the stack. The latter feature makes it possible to write functions that take a variable number of arguments.

Consider the `add_10` example presented earlier. This time it is written

```
define add_10 ()
{
   variable x;
   x = ();
   return x + 10;
}
variable s = add_10 (12);  % ==> s = 22;
```

For the uninitiated, this example looks as if it is destined for disaster. The `add_10` function appears to accept zero arguments, yet it was called with a single argument. On top of that, the assignment to `x` looks strange. The truth is, the code presented in this example makes perfect sense, once you realize what is happening.

First, consider what happens when `add_10` is called with the parameter 12. Internally, 12 is pushed onto the stack and then the function called. Now, consider the function `add_10` itself. In it, `x` is a local variable. The strange looking assignment '`x=()`' causes whatever is on the top of the stack to be assigned to `x`. In other words, after this statement, the value of `x` will be 12, since 12 is at the top of the stack.

A generic function of the form

```
define function_name (x, y, ..., z)
{
   .
   .
}
```

is transformed internally by the parser to

```
define function_name ()
{
   variable x, y, ..., z;
   z = ();
   .
   .
   y = ();
   x = ();
```

```
   .
   .
}
```

before further parsing. (The `add_10` function, as defined above, is already in this form.) With this knowledge in hand, one can write a function that accepts a variable number of arguments. Consider the function:

```
define average_n (n)
{
   variable x, y;
   variable s;

   if (n == 1)
     {
        x = ();
        s = x;
     }
   else if (n == 2)
     {
        y = ();
        x = ();
        s = x + y;
     }
   else throw NotImplementedError;

   return s / n;
}
variable ave1 = average_n (3.0, 1);        % ==> 3.0
variable ave2 = average_n (3.0, 5.0, 2);   % ==> 4.0
```

Here, the last argument passed to `average_n` is an integer reflecting the number of quantities to be averaged. Although this example works fine, its principal limitation is obvious: it only supports one or two values. Extending it to three or more values by adding more `else if` constructs is rather straightforward but hardly worth the effort. There must be a better way, and there is:

```
define average_n (n)
{
   variable s, x;
   s = 0;
   loop (n)
     {
        x = ();    % get next value from stack
        s += x;
     }
   return s / n;
}
```

The principal limitation of this approach is that one must still pass an integer that specifies how many values are to be averaged. Fortunately, a special variable exists that is local to every function

and contains the number of values that were passed to the function. That variable has the name _NARGS and may be used as follows:

```
define average_n ()
{
   variable x, s = 0;

   if (_NARGS == 0)
     usage ("ave = average_n (x, ...);");

   loop (_NARGS)
     {
        x = ();
        s += x;
     }
   return s / _NARGS;
}
```

Here, if no arguments are passed to the function, the `usage` function will generate a `UsageError` exception along with a simple message indicating how to use the function.

8.6 Returning Values

As stated earlier, the usual way to return values from a function is via the `return` statement. This statement has the simple syntax

```
return expression-list ;
```

where *expression-list* is a comma separated list of expressions. If the function does not return any values, the expression list will be empty. A simple example of a function that can return multiple values (two in this case) is:

```
define sum_and_diff (x, y)
{
    variable sum, diff;

    sum = x + y;  diff = x - y;
    return sum, diff;
}
```

8.7 Multiple Assignment Statement

In the previous section an example of a function returning two values was given. That function can also be written somewhat simpler as:

```
define sum_and_diff (x, y)
{
```

```
   return x + y, x - y;
}
```

This function may be called using

```
(s, d) = sum_and_diff (12, 5);
```

After the above line is executed, `s` will have a value of 17 and the value of `d` will be 7.

The most general form of the multiple assignment statement is

```
( var_1, var_2, ..., var_n ) = expression;
```

Here `expression` is an arbitrary expression that leaves `n` items on the stack, and `var_k` represents an l-value object (permits assignment). The assignment statement removes those values and assigns them to the specified variables. Usually, `expression` is a call to a function that returns multiple values, but it need not be. For example,

```
(s,d) = (x+y, x-y);
```

produces results that are equivalent to the call to the `sum_and_diff` function. Another common use of the multiple assignment statement is to swap values:

```
(x,y) = (y,x);
(a[i], a[j], a[k]) = (a[j], a[k], a[i]);
```

If an l-value is omitted from the list, then the corresponding value will be removed fro the stack. For example,

```
(s, ) = sum_and_diff (9, 4);
```

assigns the sum of 9 and 4 to `s` and the difference (`9-4`) is removed from the stack. Similarly,

```
() = fputs ("good luck", fp);
```

causes the return value of the `fputs` function to be discarded.

It is possible to create functions that return a *variable number* of values instead of a *fixed number*. Although such functions are discouraged, it is easy to cope with them. Usually, the value at the top of the stack will indicate the actual number of return values. For such functions, the multiple assignment statement cannot directly be used. To see how such functions can be dealt with, consider the following function:

```
define read_line (fp)
{
   variable line;
   if (-1 == fgets (&line, fp))
     return -1;
   return (line, 0);
}
```

This function returns either one or two values, depending upon the return value of `fgets`. Such a function may be handled using:

```
status = read_line (fp);
if (status != -1)
  {
     s = ();
     .
     .
  }
```

In this example, the *last* value returned by `read_line` is assigned to `status` and then tested. If it is non-zero, the second return value is assigned to `s`. In particular note the empty set of parenthesis in the assignment to `s`. This simply indicates that whatever is on the top of the stack when the statement is executed will be assigned to `s`.

8.8 Exit-Blocks

An *exit-block* is a set of statements that get executed when a functions returns. They are very useful for cleaning up when a function returns via an explicit call to `return` from deep within a function.

An exit-block is created by using the `EXIT_BLOCK` keyword according to the syntax

`EXIT_BLOCK { `*statement-list*` }`

where *statement-list* represents the list of statements that comprise the exit-block. The following example illustrates the use of an exit-block:

```
define simple_demo ()
{
   variable n = 0;

   EXIT_BLOCK { message ("Exit block called."); }

   forever
    {
      if (n == 10) return;
      n++;
    }
}
```

Here, the function contains an exit-block and a `forever` loop. The loop will terminate via the `return` statement when `n` is 10. Before it returns, the exit-block will get executed.

A function can contain multiple exit-blocks, but only the last one encountered during execution will actually get used. For example,

```
define simple_demo (n)
{
```

```
    EXIT_BLOCK { return 1; }

    if (n != 1)
      {
         EXIT_BLOCK { return 2; }
      }
    return;
}
```

If 1 is passed to this function, the first exit-block will get executed because the second one would not have been encountered during the execution. However, if some other value is passed, the second exit-block would get executed. This example also illustrates that it is possible to explicitly return from an exit-block, but nested exit-blocks are illegal.

Chapter 9

Namespaces

By default, all global variables and functions are defined in the global or public namespace. In addition to the global namespace, every compilation unit (e.g., a file containing **S-Lang** code) has a private, or anonymous namespace. The private namespace is used when one wants to restrict the usage of one or more functions or variables to the compilation unit that defines them without worrying about objects with the same names defined elsewhere.

Objects are declared as belonging to the private namespace using the `private` declaration keyword. Similarly if a variable is declared using the `public` qualifier, it will be placed in the public namespace. For example,

```
private variable i;
public variable j;
```

defines a variable called `i` in the private namespace and one called `j` in the public namespace.

The `implements` function may be used to create a new namespace of a specified name and have it associated with the compilation unit. Objects may be placed into this namespace space using the `static` keyword, e.g.,

```
static variable X;
static define foo () {...}
```

For this reason, such a namespace will be called the *static namespace* associated with the compilation unit. Such objects may be accessed from outside the local compilation unit using the namespace operator `->` in conjunction with the name of the namespace.

Since it is possible for three namespaces (private, static, public) to be associated with a compilation unit, it is important to understand how names are resolved by the parser. During the compilation stage, symbols are looked up according to the current scope. If in a function, the local variables of the function are searched first. Then the search proceeds with symbols in the private namespace, followed by those in the `static` namespace associated with the compilation unit (if any), and finally with the public namespace. If after searching the public namespace the symbol has not been resolved, an `UndefinedNameError` exception will result.

In addition to using the `implements` function, there are other ways to associate a namespace with

a compilation unit. One is via the optional namespace argument of the `evalfile` function. For example,

```
() = evalfile ("foo.sl", "bar");
```

will cause `foo.sl` to be loaded and associated with a namespace called `bar`. Then any static symbols of `foo.sl` may accessed using the `bar->` prefix.

It is important to note that if a static namespace has been associated with the compilation unit, then any symbols in that unit declared without an namespace qualifier will be placed in the static namespace. Otherwise such symbols will be placed in the public namespace, and any symbols declared as `static` will be placed in the private namespace.

To illustrate these concepts, consider the following example:

```
% foo.sl
variable X = 1;
static variable Y;
private variable Z;
public define set_Y (y) { Y = y; }
static define set_z (z) { Z = z; }
```

If `foo.sl` is loaded via

```
() = evalfile ("foo.sl");
```

then no static namespace will be associated with it. As a result, `X` will be placed in the public namespace since it was declared with no namespace qualifier. Also `Y` and `set_z` will be placed in the private namespace since no static namespace has been associated with the file. In this scenario there will be no way to get at the `Z` variable from outside of `foo.sl` since both it and the function that accesses it (`set_z`) are placed in the private namespace.

On the other hand, suppose that the file is loaded using a namespace argument:

```
() = evalfile ("foo.sl", "foo");
```

In this case `X`, `Y`, and `get_z` will be placed in the `foo` namespace. These objects may be accessed from outside `foo.sl` using the `foo->` prefix, e.g.,

```
foo->set_z (3.0);
if (foo->X == 2) foo->Y = 1;
```

Because a file may be loaded with or without a namespace attached to it, it is a good idea to avoid using the `static` qualifier. To see this, consider again the above example but this time without the use of the `static` qualifier:

```
% foo.sl
variable X = 1;
variable Y;
private variable Z;
public define set_Y (y) { Y = y; }
define set_z (z) { Z = z; }
```

When loaded without a namespace argument, the variable `Z` will remain in the private namespace, but the `set_z` function will be put in the public namespace. Previously `set_z` was put in the private namespace making both it and `Z` inaccessible.

Chapter 10

Arrays

An array is a container object that can contain many values of one data type. Arrays are very useful objects and are indispensable for certain types of programming. The purpose of this chapter is to describe how arrays are defined and used in the **S-Lang** language.

10.1 Creating Arrays

The **S-Lang** language supports multi-dimensional arrays of all data types. Since the `Array_Type` is a data type, one can even have arrays of arrays. To create a multi-dimensional array of *SomeType* and assign to some variable, use:

```
a = SomeType [dim0, dim1, ..., dimN];
```

Here *dim0*, *dim1*, ... *dimN* specify the size of the individual dimensions of the array. The current implementation permits arrays to contain as many as 7 dimensions. When a numeric array is created, all its elements are initialized to zero. The initialization of other array types depend upon the data type, e.g., the elements in `String_Type` and `Struct_Type` arrays are initialized to `NULL`.

As a concrete example, consider

```
a = Integer_Type [10];
```

which creates a one-dimensional array of 10 integers and assigns it to `a`. Similarly,

```
b = Double_Type [10, 3];
```

creates a `30` element array of double precision numbers arranged in `10` rows and `3` columns, and assigns it to `b`.

10.1.1 Range Arrays

There is a more convenient syntax for creating and initializing 1-d arrays. For example, to create an array of ten integers whose elements run from 1 through 10, one may simply use:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

Similarly,

```
b = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0];
```

specifies an array of ten doubles.

An even more compact way of specifying a numeric array is to use a *range-array*. For example,

```
a = [0:9];
```

specifies an array of 10 integers whose elements range from 0 through 9. The syntax for the most general form of range array is given by

```
[first-value : last-value : increment]
```

where the *increment* is optional and defaults to 1. This creates an array whose first element is *first-value* and whose successive values differ by *increment*. *last-value* sets an upper limit upon the last value of the array as described below.

If the range array `[a:b:c]` is integer valued, then the interval specified by `a` and `b` is closed. That is, the kth element of the array x_k is given by x_k=a+kc and satisfies a¡=x_k¡=b. Hence, the number of elements in an integer range array is given by the expression 1 + (b-a)/c.

The situation is somewhat more complicated for floating point range arrays. The interval specified by a floating point range array `[a:b:c]` is semi-open such that `b` is not contained in the interval. In particular, the kth element of `[a:b:c]` is given by x_k=a+kc such that a¡=x_k¡b when c¿=0, and b¡x_k¡=a otherwise. The number of elements in the array is one greater than the largest k that satisfies the open interval constraint.

Here are a few examples that illustrate the above comments:

```
[1:5:1]         ==> [1,2,3,4,5]
[1.0:5.0:1.0]   ==> [1.0, 2.0, 3.0, 4.0]
[5:1:-1]        ==> [5,4,3,2,1]
[5.0:1.0:-1.0]  ==> [5.0, 4.0, 3.0, 2.0];
[1:1]           ==> [1]
[1.0:1.0]       ==> []
[1.0:1.0001]    ==> [1.0]
[1:-3]          ==> []
```

10.1.2 Creating arrays via the dereference operator

Another way to create an array is to apply the dereference operator `@` to the `DataType_Type` literal `Array_Type`. The actual syntax for this operation resembles a function call

```
variable a = @Array_Type (data-type, integer-array);
```

where *data-type* is of type `DataType_Type` and *integer-array* is a 1-d array of integers that specify the size of each dimension. For example,

```
variable a = @Array_Type (Double_Type, [10, 20]);
```

will create a 10 by 20 array of doubles and assign it to `a`. This method of creating arrays derives its power from the fact that it is more flexible than the methods discussed in this section. It is particularly useful for creating arrays during run-time in situations where the data-type can vary.

10.2 Reshaping Arrays

It is sometimes useful to change the 'shape' of an array using the `reshape` function. For example, a 1-d 10 element array may be reshaped into a 2-d array consisting of 5 rows and 2 columns. The only restriction on the operation is that the arrays must be commensurate. The `reshape` function follows the syntax

```
reshape (array-name, integer-array);
```

where *array-name* specifies the array to be reshaped to the dimensions given by `integer-array`, a 1-dimensional array of integers. It is important to note that this does *not* create a new array, it simply reshapes the existing array. Thus,

```
variable a = Double_Type [100];
reshape (a, [10, 10]);
```

turns `a` into a 10 by 10 array, as well as any other variables attached to the array.

The `_reshape` function works like `reshape` except that it creates a new array instead of changing the shape of an existing array:

```
new_a = _reshape (a, [10,10]);
```

10.3 Simple Array Indexing

An individual element of an array may be referred to by its *index*. For example, `a[0]` specifies the zeroth element of the one dimensional array `a`, and `b[3,2]` specifies the element in the third row and second column of the two dimensional array `b`. As in C, array indices are numbered from 0. Thus if `a` is a one-dimensional array of ten integers, the last element of the array is given by `a[9]`. Using `a[10]` would result in an `IndexError` exception.

A negative index may be used to index from the end of the array, with `a[-1]` referring to the last element of `a`. Similarly, `a[-2]` refers to the next to the last element, and so on.

One may use the indexed value like any other variable. For example, to set the third element of an integer array to 6, use

```
a[2] = 6;
```

Similarly, that element may be used in an expression, such as

```
y = a[2] + 7;
```

Unlike other **S-Lang** variables which inherit a type upon assignment, array elements already have a type and any attempt to assign a value with an incompatible type will result in a `TypeMismatchError` exception. For example, it is illegal to assign a string value to an integer array.

One may use any integer expression to index an array. A simple example that computes the sum of the elements of a 10 element 1-d array is

```
variable i, s;
s = 0;
for (i = 0; i < 10; i++) s += a[i];
```

(In practice, do not carry out sums this way— use the `sum` function instead, which is much simpler and faster, i.e., `s=sum(a)`).

10.4 Indexing Multiple Elements with Ranges

Unlike many other languages, **S-Lang** permits arrays to be indexed by other integer arrays. Suppose that `a` is a 1-d array of 10 doubles. Now consider:

```
i = [6:8];
b = a[i];
```

Here, `i` is a 1-dimensional range array of three integers with `i[0]` equal to 6, `i[1]` equal to 7, and `i[2]` equal to 8. The statement `b = a[i];` will create a 1-d array of three doubles and assign it to `b`. The zeroth element of `b`, `b[0]` will be set to the sixth element of `a`, or `a[6]`, and so on. In fact, these two simple statements are equivalent to

```
b = Double_Type [3];
b[0] = a[6];
b[1] = a[7];
b[2] = a[8];
```

except that using an array of indices is not only much more convenient, but executes much faster.

More generally, one may use an index array to specify which elements are to participate in a calculation. For example, consider

```
a = Double_Type [1000];
i = [0:499];
j = [500:999];
a[i] = -1.0;
a[j] = 1.0;
```

This creates an array of 1000 doubles and sets the first 500 elements to `-1.0` and the last 500 to `1.0`. Actually, one may do away with the `i` and `j` variables altogether and use

```
a = Double_Type [1000];
a[[0:499]] = -1.0;
a[[500:999]] = 1.0;
```

It is important to note that the syntax requires the use of the double square brackets, and in particular that `a[[0:499]]` is *not* the same as `a[0:499]`. In fact, the latter will generate a syntax error.

Index-arrays are not contrained to be one-dimensional arrays. Suppose that `I` represents a multidimensional index array, and that `A` is the array to be indexed. Then what does `A[I]` represent? Its value will be an array of the same type as `A`, but with the dimensionality of I. For example,

```
a = 1.0*[1:10];
i = _reshape ([4,5,6,7,8,9], [2,3]);
```

defines `a` to be a 10 element array of doubles, and `i` to be `2x3` array of integers. Then `a[i]` will be a `2x3` array of doubles with elements:

```
a[4]   a[5]   a[6]
a[7]   a[8]   a[9]
```

Often, it is convenient to use a "rubber" range to specify indices. For example, `a[[500:]]` specifies all elements of `a` whose index is greater than or equal to `500`. Similarly, `a[[:499]]` specifies the first 500 elements of `a`. Finally, `a[[:]]` specifies all the elements of `a`. The latter form may also be written as `a[*]`.

One should be careful when using index arrays with negative elements. As pointed out above, a negative index is used to index from the end of the array. That is, `a[-1]` refers to the last element of `a`. How should `a[[[0:-1]]` be interpreted?

In version 1 of the interpreter, when used in an array indexing context, a construct such as `[0:-1]` was taken to mean from the first element through the last. While this might seem like a convenient shorthand, in retrospect it was a bad idea. For this reason, the meaning of a ranges over negative valued indices was changed in version 2 of the interpreter as follows: First the index-range gets expanded to an array of indices according to the rules for range arrays described above. Then if any of the resulting indices are negative, they are interpreted as indices from the end of the array. For example, if `a` is an array of 10 elements, then `a[[-2:3]]` is first expanded to `a[[-2,-1,0,1,2,3]]`, and then to the 6 element array

```
[ a[8], a[9], a[0], a[1], a[2], a[3] ]
```

So, what does `a[[0:-1]]` represent in the new interpretation? Since `[0:-1]` expands to an empty array, `a[[0:-1]]` will also produce an empty array.

Indexing of multidimensional arrays using ranges works similarly. Suppose `a` is a 100 by 100 array of doubles. Then the expression `a[0, *]` specifies all elements in the zeroth row. Similarly, `a[*, 7]` specifies all elements in the seventh column. Finally, `a[[3:5],[6:12]]` specifies the 3 by 7 region consisting of rows 3, 4, and 5, and columns 6 through 12 of `a`.

Before leaving this section, a few examples are presented to illustrate some of these points.

The "trace" of a matrix is an important concept that occurs frequently in linear algebra. The trace of a 2d matrix is given by the sum of its diagonal elements. Consider the creation of a function that computes the trace of such a matrix.

The most straightforward implementation of such a function uses an explicit loop:

```
define array_trace (a, n)
{
   variable s = 0, i;
   for (i = 0; i < n; i++) s += a[i, i];
   return s;
}
```

Better yet is to recognize that the diagonal elements of an `n` by `n` array are given by an index array `I` with elements 0, `n+1`, `2*n+2`, ..., `n*n-1`, or more precisely as

```
[0:n*n-1:n+1]
```

Hence the above may be written more simply as

```
define array_trace (a, n)
{
   return sum (a[[0:n*n-1:n+1]]);
}
```

The following example creates a 10 by 10 integer array, sets its diagonal elements to 5, and then computes the trace of the array:

```
a = Integer_Type [10, 10];
a[[0:99:11]] = 5;
the_trace = array_trace(a, 10);
```

In the previous examples, the size of the array was passed as an additional argument. This is unnecessary because the size may be obtained from array itself by using the `array_shape` function. For example, the following function may be used to obtain the indices of the diagonal element of an array:

```
define diag_indices (a)
{
   variable dims = array_shape (a);
   if (length (dims) != 2)
     throw InvalidParmError, "Expecting a 2d array";
   if (dims[0] != dims[1])
     throw InvalidParmError, "Expecting a square array";
   variable n = dims[0];
   return [0:n*(n-1):n+1];
}
```

Using this function, the trace function may be written more simply as

```
define array_trace (a)
{
   return sum (a[diag_indices(a)]);
}
```

Another example of this technique is a function that creates an `n` by `n` unit matrix:

```
define unit_matrix (n)
{
   variable a = Int_Type[n, n];
   a[diag_indices(a)] = 1;
   return a;
}
```

10.5 Arrays and Variables

When an array is created and assigned to a variable, the interpreter allocates the proper amount of space for the array, initializes it, and then assigns to the variable a *reference* to the array. So, a variable that represents an array has a value that is really a reference to the array. This has several consequences, most good and some bad. It is believed that the advantages of this representation outweigh the disadvantages. First, we shall look at the positive aspects.

When a variable is passed to a function, it is always the value of the variable that gets passed. Since the value of a variable representing an array is a reference, a reference to the array gets passed. One major advantage of this is rather obvious: it is a fast and efficient way to pass the array. This also has another consequence that is illustrated by the function

```
define init_array (a)
{
   variable i;
   variable n = length(a);
   _for i (0, n-1, 1)
     a[i] = some_function (i);
}
```

where `some_function` is a function that generates a scalar value to initialize the *ith* element. This function can be used in the following way:

```
variable X = Double_Type [100000];
init_array (X);
```

Since the array is passed to the function by reference, there is no need to make a separate copy of the `100000` element array. As pointed out above, this saves both execution time and memory. The other salient feature to note is that any changes made to the elements of the array within the function will be manifested in the array outside the function. Of course, in this case this is a desirable side-effect.

To see the downside of this representation, consider:

```
a = Double_Type [10];
b = a;
a[0] = 7;
```

What will be the value of `b[0]`? Since the value of `a` is really a reference to the array of ten doubles, and that reference was assigned to `b`, `b` also refers to the same array. Thus any changes made to the elements of `a`, will also be made implicitly to `b`.

This begs the question: If the assignment of a variable attached to an an array to another variable results in the assignment of the same array, then how does one make separate copies of the array? There are several answers including using an index array, e.g., `b = a[*]`; however, the most natural method is to use the dereference operator:

```
a = Double_Type [10];
b = @a;
a[0] = 7;
```

In this example, a separate copy of `a` will be created and assigned to `b`. It is very important to note that **S-Lang** never implicitly dereferences an object. So, one must explicitly use the dereference operator. This means that the elements of a dereferenced array are not themselves dereferenced. For example, consider dereferencing an array of arrays, e.g.,

```
a = Array_Type [2];
a[0] = Double_Type [10];
a[1] = Double_Type [10];
b = @a;
```

In this example, `b[0]` will be a reference to the array that `a[0]` references because `a[0]` was not explicitly dereferenced.

10.6 Using Arrays in Computations

Many functions and operations work transparently with arrays. For example, if `a` and `b` are arrays, then the sum `a + b` is an array whose elements are formed from the sum of the corresponding elements of `a` and `b`. A similar statement holds for all other binary and unary operations.

Let's consider a simple example. Suppose, that we wish to solve a set of `n` quadratic equations whose coefficients are given by the 1-d arrays `a`, `b`, and `c`. In general, the solution of a quadratic equation will be two complex numbers. For simplicity, suppose that all we really want is to know what subset of the coefficients, `a`, `b`, `c`, correspond to real-valued solutions. In terms of `for` loops, we can write:

```
index_array = Char_Type [n];
_for i (0, n-1, 1)
  {
     d = b[i]^2 - 4 * a[i] * c[i];
     index_array [i] = (d >= 0.0);
  }
```

In this example, the array `index_array` will contain a non-zero value if the corresponding set of coefficients has a real-valued solution. This code may be written much more compactly and with more clarity as follows:

```
index_array = ((b^2 - 4 * a * c) >= 0.0);
```

Moreover, it executes about 20 times faster than the version using an explicit loop.

S-Lang has a powerful built-in function called `where`. This function takes an array of boolean values and returns an array of indices that correspond to where the elements of the input array are

non-zero. The utility of this simple operation cannot be overstated. For example, suppose `a` is a 1-d array of `n` doubles, and it is desired to set all elements of the array whose value is less than zero to zero. One way is to use a `for` loop:

```
_for i (0, n-1, 1)
  if (a[i] < 0.0) a[i] = 0.0;
```

If `n` is a large number, this statement can take some time to execute. The optimal way to achieve the same result is to use the `where` function:

```
a[where (a < 0.0)] = 0;
```

Here, the expression `(a < 0.0)` returns a boolean array whose dimensions are the same size as `a` but whose elements are either 1 or 0, according to whether or not the corresponding element of `a` is less than zero. This array of zeros and ones is then passed to the `where` function, which returns a 1-d integer array of indices that indicate where the elements of `a` are less than zero. Finally, those elements of `a` are set to zero.

Consider once more the example involving the set of `n` quadratic equations presented above. Suppose that we wish to get rid of the coefficients of the previous example that generated non-real solutions. Using an explicit `for` loop requires code such as:

```
nn = 0;
_for i (0, n-1, 1)
  if (index_array [i]) nn++;

tmp_a = Double_Type [nn];
tmp_b = Double_Type [nn];
tmp_c = Double_Type [nn];

j = 0;
_for i (0, n-1, 1)
  {
     if (index_array [i])
       {
          tmp_a [j] = a[i];
          tmp_b [j] = b[i];
          tmp_c [j] = c[i];
          j++;
       }
  }
a = tmp_a;
b = tmp_b;
c = tmp_c;
```

Not only is this a lot of code, making it hard to digest, but it is also clumsy and error-prone. Using the `where` function, this task is trivial and executes in a fraction of the time:

```
i = where (index_array != 0);
a = a[i];
b = b[i];
c = c[i];
```

Most of the examples up till now assumed that the dimensions of the array were known. Although the intrinsic function `length` may be used to get the total number of elements of an array, it cannot be used to get the individual dimensions of a multi-dimensional array. The `array_shape` function may be used to determine the dimensionality of an array. It may be used to determine the number of rows of an array as follows:

```
define num_rows (a)
{
   return array_shape (a)[0];
}
```

The number of columns may be obtained in a similar manner:

```
define num_cols (a)
{
   variable dims = array_shape (a);
   if (length(dims) > 1) return dims[1];
   return 1;
}
```

The `array_shape` function may also be used to create an array that has the same number of dimensions as another array:

```
define make_int_array (a)
{
   return @Array_Type (Int_Type, array_shape (a));
}
```

Finally, the `array_info` function may be used to get additional information about an array, such as its data type and size.

Chapter 11

Associative Arrays

An associative array differs from an ordinary array in the sense that its size is not fixed and that it is indexed by a string, called the *key*. For example, consider:

```
A = Assoc_Type [Int_Type];
A["alpha"] = 1;
A["beta"] = 2;
A["gamma"] = 3;
```

Here, `A` has been assigned to an associative array of integers (`Int_Type`) and then three keys were been added to the array.

As the example suggests, an associative array may be created using one of the following forms:

```
Assoc_Type [type] Assoc_Type [type, default-value] Assoc_Type []
```

The last form returns an *un-typed* associative array capable of storing values of any type.

The form involving a *default-value* is useful for associating a default value with non-existent array members. This feature is explained in more detail below.

There are several functions that are specially designed to work with associative arrays. These include:

- `assoc_get_keys`, which returns an ordinary array of strings containing the keys of the array.
- `assoc_get_values`, which returns an ordinary array of the values of the associative array. If the associative array is un-typed, then an array of `Any_Type` objects will be returned.
- `assoc_key_exists`, which can be used to determine whether or not a key exists in the array.
- `assoc_delete_key`, which may be used to remove a key (and its value) from the array.

To illustrate the use of an associative array, consider the problem of counting the number of repeated occurrences of words in a list. Let the word list be represented as an array of strings given by `word_list`. The number of occurrences of each word may be stored in an associative array as follows:

```
a = Assoc_Type [Int_Type];
foreach word (word_list)
  {
     if (0 == assoc_key_exists (a, word))
       a[word] = 0;
     a[word]++;  % same as a[word] = a[word] + 1;
  }
```

Note that `assoc_key_exists` was necessary to determine whether or not a word was already added to the array in order to properly initialize it. However, by creating the associative array with a default value of 0, the above code may be simplified to

```
variable a, word;
a = Assoc_Type [Int_Type, 0];
foreach word (word_list)
  a[word]++;
```

Associative arrays are extremely useful and have may other applications. Whenever there is a one to one mapping between a string and some object, one should always consider using an associative array to represent the mapping. To illustrate this point, consider the following code fragment:

```
define call_function (name, arg)
{
   if (name == "foo") return foo (arg);
   if (name == "bar") return bar (arg);
     .
     .
   if (name == "baz") return baz (arg);
   throw InvalidParmError;
}
```

This represents a mapping between names and functions. Such a mapping may be written in terms of an associative array as follows:

```
private define invalid_fun (arg) { throw InvalidParmError; }
Fun_Map = Assoc_Type[Ref_Type, &invalid_fun];
define add_function (name, fun)
{
   Fun_Map[name] = fun;
}
add_function ("foo", &foo);
add_function ("bar", &bar);
   .
   .
add_function ("baz", &baz);
define call_function (name, arg)
{
   return (@Fun_Map[name])(arg);
}
```

The most redeeming feature of the version involving the series of `if` statements is that it is easy to understand. However, the version involving the associative array has two significant advantages over the former. Namely, the function lookup will be much faster with a time that is independent of the item being searched, and it is extensible in the sense that additional functions may be added at run-time, e.g.,

```
add_function ("bing", &bing);
```

Chapter 12

Structures and User-Defined Types

A *structure* is a heterogeneous container object, i.e., it is an object with elements whose values do not have to be of the same data type. The elements or fields of a structure are named, and one accesses a particular field of the structure via the field name. This should be contrasted with an array whose values are of the same type, and whose elements are accessed via array indices.

A *user-defined* data type is a structure with a fixed set of fields defined by the user.

12.1 Defining a Structure

The `struct` keyword is used to define a structure. The syntax for this operation is:

```
struct {field-name-1, field-name-2, ...  field-name-N};
```

This creates and returns a structure with N fields whose names are specified by *field-name-1*, *field-name-2*, ..., *field-name-N*. When a structure is created, the values of its fields are initialized to `NULL`.

For example,

```
variable t = struct { city_name, population, next };
```

creates a structure with three fields and assigns it to the variable `t`.

Alternatively, a structure may be created by dereferencing `Struct_Type`. Using this technique, the above structure may be created using one of the two forms:

```
t = @Struct_Type ("city_name", "population", "next");
t = @Struct_Type (["city_name", "population", "next"]);
```

This approach is useful when creating structures dynamically where one does not know the name of the fields until run-time.

Like arrays, structures are passed around by reference. Thus, in the above example, the value of `t` is a reference to the structure. This means that after execution of

```
u = t;
```

both `t` and `u` refer to the *same* underlying structure, since only the reference was copied by the assignment. To actually create a new copy of the structure, use the *dereference* operator, e.g.,

```
variable u = @t;
```

It create new structure whose field names are identical to the old and copies the field values to the new structure. If any of the values are objects that are passed by reference, then only the references will be copied. In other words,

```
t = struct{a};
t.a = [1:10];
u = @t;
```

will produce a structure `u` that references the same array as `t`.

12.2 Accessing the Fields of a Structure

The dot (`.`) operator is used to specify the particular field of structure. If `s` is a structure and `field_name` is a field of the structure, then `s.field_name` specifies that field of `s`. This specification can be used in expressions just like ordinary variables. Again, consider

```
t = struct { city_name, population, next };
```

described in the last section. Then,

```
t.city_name = "New York";
t.population = 13000000;
if (t.population > 200) t = t.next;
```

are all valid statements involving the fields of `t`.

12.3 Linked Lists

One of the most important uses of structures is the creation of *dynamic* data structures such as *linked-lists*. A linked-list is simply a chain of structures that are linked together such that one structure in the chain is the value of a field of the previous structure in the chain. To be concrete, consider the structure discussed earlier:

```
t = struct { city_name, population, next };
```

and suppose that it is desired to create a linked-list of such objects to store population data. The purpose of the `next` field is to provide the link to the next structure in the chain. Suppose that there exists a function, `read_next_city`, that reads city names and populations from a file. Then the list may be created using:

```
define create_population_list ()
{
   variable city_name, population, list_root, list_tail;
   variable next;

   list_root = NULL;
   while (read_next_city (&city_name, &population))
     {
        next = struct {city_name, population, next };

        next.city_name = city_name;
        next.population = population;
        next.next = NULL;

        if (list_root == NULL)
          list_root = next;
        else
          list_tail.next = next;

        list_tail = next;
     }
   return list_root;
}
```

In this function, the variables `list_root` and `list_tail` represent the beginning and end of the list, respectively. As long as `read_next_city` returns a non-zero value, a new structure is created, initialized, and then appended to the list via the `next` field of the `list_tail` structure. On the first time through the loop, the list is created via the assignment to the `list_root` variable.

This function may be used as follows:

```
Population_List = create_population_list ();
if (Population_List == NULL)
  throw RunTimeError, "List is empty";
```

Other functions may be created that manipulate the list. Here is one that finds the city with the largest population:

```
define get_largest_city (list)
{
   variable largest;

   largest = list;
   while (list != NULL)
     {
        if (list.population > largest.population)
          largest = list;
        list = list.next;
     }
   return largest.city_name;
}
```

```
vmessage ("%s is the largest city in the list",
          get_largest_city (Population_List)));
```

The `get_largest_city` is a typical example of how one traverses a linear linked-list by starting at the head of the list and successively moves to the next element of the list via the `next` field.

In the previous example, a `while` loop was used to traverse the linked list. It is also possible to use a `foreach` loop for this:

```
define get_largest_city (list)
{
   variable largest, elem;

   largest = list;
   foreach item (list)
     {
        if (item.population > largest.population)
          largest = item;
     }
   return largest.city_name;
}
```

Here a `foreach` loop has been used to walk the list via its `next` field. If the field name linking the elements was not called `next`, then it would have been necessary to use the `using` form of the `foreach` statement. For example, if the field name implementing the linked list was `next_item`, then

```
 foreach item (list) using ("next_item")
   {
      .
      .
   }
```

would have been used. In other words, unless otherwise indicated via the `using` clause, `foreach` walks the list using a field named `next`.

Now consider a function that sorts the list according to population. To illustrate the technique, a *bubble-sort* will be used, not because it is efficient (it is not), but because it is simple, intuitive, and provides another example of structure manipulation:

```
define sort_population_list (list)
{
   variable changed;
   variable node, next_node, last_node;
   do
     {
        changed = 0;
        node = list;
        next_node = node.next;
        last_node = NULL;
        while (next_node != NULL)
```

```
            {
               if (node.population < next_node.population)
                 {
                    % swap node and next_node
                    node.next = next_node.next;
                    next_node.next = node;
                    if (last_node != NULL)
                      last_node.next = next_node;

                    if (list == node) list = next_node;
                    node = next_node;
                    next_node = node.next;
                    changed++;
                 }
               last_node = node;
               node = next_node;
               next_node = next_node.next;
            }
         }
      while (changed);

      return list;
   }
```

Note the test for equality between `list` and `node`, i.e.,

```
                    if (list == node) list = next_node;
```

It is important to appreciate the fact that the values of these variables are references to structures, and that the comparison only compares the references and *not* the actual structures they reference. If it were not for this, the algorithm would fail.

12.4 Defining New Types

A user-defined data type may be defined using the `typedef` keyword. In the current implementation, a user-defined data type is essentially a structure with a user-defined set of fields. For example, in the previous section a structure was used to represent a city/population pair. We can define a data type called `Population_Type` to represent the same information:

```
typedef struct
{
   city_name,
   population
} Population_Type;
```

This data type can be used like all other data types. For example, an array of Population_Type types can be created,

```
variable a = Population_Type[10];
```

and 'populated' via expressions such as

```
a[0].city_name = "Boston";
a[0].population = 2500000;
```

The new type `Population_Type` may also be used with the `typeof` function:

```
if (Population_Type == typeof (a))
  city = a.city_name;
```

The dereference `@` may be used to create an instance of the new type:

```
a = @Population_Type;
a.city_name = "Calcutta";
a.population = 13000000;
```

Another feature that user-defined types possess is that the action of the binary and unary operations may be defined for them. This idea is discussed in more detail below.

12.5 Operator Overloading

The binary and unary operators may be extended to user-defined types. To illustrate how this works, consider a data type that represents a vector in 3-space:

```
typedef struct { x, y, z } Vector_Type;
```

and a function that instantiates such an object:

```
define vector_new (x, y, z)
{
   variable v = @Vector_Type;
   v.x = double(x); v.y = double(y); v.z = double(z);
   return v;
}
```

This function may be used to define a function that adds two vectors together:

```
define vector_add (v1, v2)
{
   return vector_new (v1.x+v2.x, v1.y+v2.y, v1.z+v2.z);
}
```

Using these functions, three vectors representing the points `(2,3,4)`, `(6,2,1)`, and `(-3,1,-6)` may be created using

```
V1 = vector_new (2,3,4);
V2 = vector_new (6,2,1);
V3 = vector_new (-3,1,-6);
```

and then added together via

```
V4 = vector_add (V1, vector_add (V2, V3));
```

The problem with the last statement is that it is not a very natural way to express the addition of three vectors. It would be far better to extend the action of the binary `+` operator to the `Vector_Type` objects and then write the above sum more simply as

```
V4 = V1 + V2 + V3;
```

The `__add_binary` function defines the result of a binary operation between two data types:

```
__add_binary (op, result-type, funct, typeA,typeB);
```

Here, *op* is a string representing any one of the binary operators (`"+"`, `"-"`, `"*"`, `"/"`, `"=="`,...), and *funct* is reference to a function that carries out the binary operation between objects of types *typeA* and *typeB* to produce an object of type *result-type*.

This function may be used to extend the `+` operator to *Vector_Type* objects:

```
__add_binary ("+", Vector_Type, &vector_add, Vector_Type, Vector_Type);
```

Similarly the subtraction and equality operators may be extended to `Vector_Type` via

```
define vector_minus (v1, v2)
{
   return vector_new (v1.x-v2.x, v1.y-v2.y, v1.z-v2.z);
}
__add_binary ("-", Vector_Type, &vector_minus, Vector_Type, Vector_Type);

define vector_eqs (v1, v2)
{
   return (v1.x==v2.x) and (v1.y==v2.y) and (v1.z==v2.z);
}
__add_binary ("==", Char_Type, &vector_eqs, Vector_Type, Vector_Type);
```

permitting a statement such as

```
if (V2 != V1) V3 = V2 - V1;
```

The `-` operator is also an unary operator that is customarily used to change the sign of an object. Unary operations may be extended to `Vector_Type` objects using the `__add_unary` function:

```
define vector_chs (v)
{
   return vector_new (-v.x, -v.y, -v.z);
}
__add_unary ("-", Vector_Type, &vector_chs, Vector_Type);
```

A trivial example of the use of the unary minus is `V4 = -V2`.

It is interesting to consider the extension of the multiplication operator `*` to `Vector_Type`. A vector may be multiplied by a scalar to produce another vector. This can happen in two ways as reflected by the following functions:

```
define vector_scalar_mul (v, a)
{
   return vector_new (a*v.x, a*v.y, a*v.z);
}
define scalar_vector_mul (a, v)
{
   return vector_new (a*v.x, a*v.y, a*v.z);
}
```

Here `a` represents the scalar, which can be any object that may be multiplied by a `Double_Type`, e.g., `Int_Type`, `Float_Type`, etc. Instead of using multiple statements involving `__add_binary` to define the action of `Int_Type+Vector_Type`, `Float_Type+Vector_Type`, etc, a single statement using `Any_Type` to represent a "wildcard" type may be used:

```
__add_binary ("*", Vector_Type, &vector_scalar_mul, Vector_Type, Any_Type);
__add_binary ("*", Vector_Type, &scalar_vector_mul, Any_Type, Vector_Type);
```

There are a couple of natural possibilities for `Vector_Type*Vector_Type`: The cross-product defined by

```
define crossprod (v1, v2)
{
   return vector_new (v1.y*v2.z-v1.z*v2.y,
                      v1.z*v2.x-v1.x*v2.z,
                      v1.x*v2.y-v1.y*v2.x);
}
```

and the dot-product:

```
define dotprod (v1, v2)
{
   return v1.x*v2.x + v1.y*v2.y + v1.z*v2.z;
}
```

The binary `*` operator between two vector types may be defined to be just one of these functions—it cannot be extended to both. If the dot-product is chosen then one would use

```
__add_binary ("*", Double_Type, &dotprod, Vector_Type_Type, Vector_Type);
```

Just because it is possible to define the action of a binary or unary operator on an user-defined type, it is not always wise to do so. A useful rule of thumb is to ask whether defining a particular operation leads to more readable and maintanable code. For example, simply looking at

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

in isolation one can easily overlook the fact that a function such as `vector_add` may be getting executed. Moreover, in cases where the action is ambiguous such as `Vector_Type*Vector_Type` it may not be clear what

```
c = a*b;
```

means unless one knows exactly what choice was made when extending the `*` operator to the types. For this reason it may be wise to leave `Vector_Type*Vector_Type` undefined and use "old-fashioned" function calls such as

```
c = dotprod (a, b);
d = crossprod (a, b);
```

to avoid the ambiguity altogether.

Finally, the `__add_string` function may be used to define the string representation of an object. Examples involving the string representation include:

```
message ("The value is " + string (V));
vmessage ("The result of %S+%S is %S", V);
str = The value of V is $V"$;
```

For the `Vector_Type` one might want to use the string represention generated by

```
define vector_string (v)
{
   return sprintf ("(%S,%S,%S)", v.x, v.y, v.z);
}
__add_string (Vector_Type, &vector_string);
```

Chapter 13

Lists

Sometimes it is desirable to utilize an object that has many of the properties of an array, but can also easily grow or shrink upon demand. The `List_Type` object has such properties.

An empty list may be created either by the `list_new` function or more simply using curly braces, e.g.,

```
list = {};
```

More generally a list of objects may be created by simply enclosing them in braces. For example,

```
list = { "hello", 7, 3.14, {&sin, &cos}}
```

specifies a list of 4 elements, where the last element is also a list. The number of items in a list may be obtained using the `length` function. For the above list, `length(list)` will return 4.

One may examine the contents of the list using an array index notation. For the above example, `list[0]` refers to the zeroth element of the list (`"hello"` in this case). Similarly,

```
list[1] = [1,2,3];
```

changes the first element of the list (7) to the array `[1,2,3]`. Also as the case for arrays one may index from the end of the list using negative indices, e.g., `list[-1]` refers to the last element of the list.

The functions `list_insert` and `list_append` may be used to add items to a list. In particular, `list_insert(list,obj,nth)` will insert the object `obj` into the list at the `nth` position. Similarly, `list_append(list,obj,nth)` will insert the object `obj` into the list right after `nth` position. If

```
list = { "hello", 7, 3.14, {&sin, &cos}}
```

then

```
list_insert (list, 0, "hi");
list_append (list, 0, "there");
list_insert (list, -1, "before");
list_append (list, -1, "after");
```

will result in the list

```
{"hi", "there", "hello", 7, 3.14, "before", {&sin,&cos}, "after"}
```

One might be tempted to use

```
list = {"hi", list};
```

to insert `"hi"` at the head of the list. However, this simply creates a new list of two items: `hi` and the original list.

Items may be removed from a list via the `list_delete` function, which deletes the item from the specified position and shrinks the list. In the context of the above example,

```
list_delete (list, 2);
```

will shrink the list to

```
{"hi", "there", 7, 3.14, "before", {&sin,&cos}, "after"}
```

Another way of removing items from the list is to use the `list_pop` function. The main difference between it and `list_delete` is that `list_pop` returns the deleted item. For example,

```
item = list_pop (list, -2);
```

would reduce the list to

```
{"hi", "there", 7, 3.14, "before", "after"}
```

and assign {`&sin,&cos`} to `item`. If the position parameter to `list_pop` is left unspecified, then the position will default to the zeroth, i.e., `list_pop(list)` is equaivalent to `list_pop(list,0)`.

To copy a list, use the dereference operator `@`:

```
new_list = @list;
```

Keep in mind that this does not perform a so-called deep copy. If any of the elements of the list are objects that are assigned by reference, only the references will be copied.

The `list_reverse` function may be used to reverse the elements of a list. Note that this does not create a new list. To create new list that is the reverse of another, copy the original using the dereference operator (@) and reverse that, i.e.,

```
new_list = list_reverse (@list);
```

Chapter 14

Error Handling

All non-trivial programs or scripts must be deal with the possibility of run-time errors. In fact, one sign of a seasoned programmer is that such a person pays particular attention to error handling. This chapter presents some techniques for handling errors using **S-Lang**. First the traditional method of using return values to indicate errors will be discussed. Then attention will turn to **S-Lang**'s more powerful exception handling mechanisms.

14.1 Traditional Error Handling

The simplist and perhaps most common mechanism for signalling a failure or error in a function is for the function to return an error code, e.g.,

```
define write_to_file (file, str)
{
   variable fp = fopen (file, "w");
   if (fp == NULL)
     return -1;
   if (-1 == fputs (str, fp))
     return -1;
   if (-1 == fclose (fp))
     return -1;
   return 0;
}
```

Here, the `write_to_file` function returns 0 if successful, or -1 upon failure. It is up to the calling routine to check the return value of `write_to_file` and act accordingly. For instance:

```
 if (-1 == write_to_file ("/tmp/foo", "bar"))
   {
      () = fprintf (stderr, "Write failed\n");
      exit (1);
   }
```

The main advantage of this technique is in its simplicity. The weakness in this approach is that the return value must be checked for every function that returns information in this way. A more subtle

problem is that even minor changes to large programs can become unwieldy. To illustrate the latter aspect, consider the following function which is supposed to be so simple that it cannot fail:

```
define simple_function ()
{
    do_something_simple ();
    more_simple_stuff ();
}
```

Since the functions called by `simple_function` are not supposed to fail, `simple_function` itself cannot fail and there is no return value for its callers to check:

```
define simple ()
{
    simple_function ();
    another_simple_function ();
}
```

Now suppose that the function `do_something_simple` is changed in some way that could cause it to fail from time to time. Such a change could be the result of a bug-fix or some feature enhancement. In the traditional error handling approach, the function would need to be modified to return an error code. That error code would have to be checked by the calling routine `simple_function` and as a result, it can now fail and must return an error code. The obvious effect is that a tiny change in one function can be felt up the entire call chain. While making the appropriate changes for a small program can be a trivial task, for a large program this could be a major undertaking opening the possibility of introducing additional errors along the way. In a nutshell, this is a code maintainence issue. For this reason, a veteran programmer using this approach to error handling will consider such possibilities from the outset and allow for error codes the first time regardless of whether the functions can fail or not, e.g.,

```
define simple_function ()
{
    if (-1 == do_something_simple ())
      return -1;
    if (-1 == more_simple_stuff ())
      return -1;
    return 0;
}
define simple ()
{
    if (-1 == simple_function ())
      return -1;
    if (-1 == another_simple_function ())
      return -1;
    return 0;
}
```

Although latter code containing explicit checks for failure is more robust and more easily maintainable than the former, it is also less readable. Moreover, since return values are now checked the code will execute somewhat slower than the code that lacks such checks. There is also no clean

separation of the error handling code from the other code. This can make it difficult to maintain if the error handling semantics of a function change. The next section discusses another approach to error handling that tries to address these issues.

14.2 Error Handling through Exceptions

This section describes **S-Lang**'s exception model. The idea is that when a function encounters an error, instead of returning an error code, it simply gives up and *throws* an exception. This idea will be fleshed out in what follows.

14.2.1 Introduction to Exceptions

Consider the `write_to_file` function used in the previous section but adapted to throw an exception:

```
define write_to_file (file, str)
{
   variable fp = fopen (file, "w");
   if (fp == NULL)
     throw OpenError;
   if (-1 == fputs (str, fp))
     throw WriteError;
   if (-1 == fclose (fp))
     throw WriteError;
}
```

Here the `throw` statement has been used to generate the appropriate exception, which in this case is either an `OpenError` exception or a `WriteError` exception. Since the function now returns nothing (no error code), it may be called as

```
write_to_file ("/tmp/foo", "bar");
next_statement;
```

As long as the `write_to_file` function encounters no errors, control passes from `write_to_file` to `next_statement`.

Now consider what happens when the function encounters an error. For concreteness assume that the `fopen` function failed causing `write_to_file` to throw the `OpenError` exception. The `write_to_file` function will stop execution after executing the `throw` statement and return to its caller. Since no provision has been made to handle the exception, `next_statement` will not execute and control will pass to the previous caller on the call stack. This process will continue until the exception is either handled or until control reaches the top-level at which point the interpreter will terminate. This process is known as *unwinding* of the call stack.

An simple exception handler may be created through the use of a *try-catch* statement, such as

```
try
 {
   write_to_file ("/tmp/foo", "bar");
```

```
 }
catch OpenError:
 {
    message ("*** Warning: failed to open /tmp/foo.");
 }
next_statement;
```

The above code works as follows: First the statement (or statements) inside the try-block are executed. As long as no exception occurs, once they have executed, control will pass on to `next_statement`, skipping the catch statement(s).

If an exception occurs while executing the statements in the try-block, any remaining statements in the block will be skipped and control will pass to the "catch" portion of the exception handler. This may consist of one or more `catch` statements and an optional *finally* statement. Each `catch` statement specifies a list of exceptions it will handle as well as the code that is to be excecuted when a matching exception is caught. If a matching `catch` statement is found for the exception, the exception will be cleared and the code associated with the catch statement will get executed. Control will then pass to `next_statement` (or first to the code in an optional `finally` block).

Catch-statements are tested against the exception in the order that they appear. Once a matching `catch` statement is found, the search will terminate. If no matching `catch`-statement is found, an optional `finally` block will be processed, and the call-stack will continue to unwind until either a matching exception handler is found or the interpreter terminates.

In the above example, an exception handler was established for the `OpenError` exception. The error handling code for this exception will cause a warning message to be displayed. Execution will resume at `next_statement`.

Now suppose that `write_to_file` successfully opened the file, but that for some reason, e.g., a full disk, the actual write operation failed. In such a case, `write_to_file` will throw a `WriteError` exception passing control to the caller. The file will remain on the disk but not fully written. An exception handler can be added for `WriteError` that removes the file:

```
try
 {
   write_to_file ("/tmp/foo", "bar");
 }
catch OpenError:
 {
    message ("*** Warning: failed to open /tmp/foo.");
 }
catch WriteError:
 {
    () = remove ("/tmp/foo");
    message ("*** Warning: failed to write to /tmp/foo");
 }
next_statement;
```

Here the exception handler for `WriteError` uses the `remove` intrinsic function to delete the file and then issues a warning message. Note that the `remove` intrinsic uses the traditional error handling mechanism— in the above example its return status has been discarded.

Above it was assumed that failure to write to the file was not critical allowing a warning message to suffice upon failure. Now suppose that it is important for the file to be written but that it is still desirable for the file to be removed upon failure. In this scenario, `next_statement` should not get executed upon failure. This can be achieved as follows:

```
try
 {
   write_to_file ("/tmp/foo", "bar");
 }
catch WriteError:
 {
    () = remove ("/tmp/foo");
    throw WriteError;
 }
next_statement;
```

Here the exception handler for `WriteError` removes the file and then re-throws the exception.

14.2.2 Obtaining information about the exception

When an exception is generated, an exception object is thrown. The object is a structure containing the following fields:

error

The exception error code (`Int_Type`).

descr

A brief description of the error (`String_Type`).

file

The filename containing the code that generated the exception (`String_Type`).

line

The line number where the exception was thrown (`Int_Type`).

function

The name of the currently executing function, or `NULL` if at top-level (`String_Type`).

message

A text message that may provide more information about the exception (`String_Type`).

object

A user-defined object.

If it is desired to have information about the exception, then an alternative form of the `try` statement must be used:

```
try (e)
{
   % try-block code
}
catch SomeException: { code ... }
```

If an exception occurs while executing the code in the try-block, then the variable `e` will be assigned the value of the exception object. As a simple example, suppose that the file `foo.sl` consists of:

```
define invert_x (x)
{
   if (x == 0)
     throw DivideByZeroError;
   return 1/x;
}
```

and that the code is called using

```
try (e)
{
   y = invert_x (0);
}
catch DivideByZeroError:
{
   vmessage ("Caught %s, generated by %s:%d\n",
             e.descr, e.file, e.line);
   vmessage ("message: %s\nobject: %S\n",
             e.message, e.object);
   y = 0;
}
```

When this code is executed, it will generate the message:

```
Caught Divide by Zero, generated by foo.sl:5
message: Divide by Zero
object: NULL
```

In this case, the value of the `message` field was assigned a default value. The reason that the `object` field is `NULL` is that no object was specified when the exception was generated. In order to throw an object, a more complex form of `throw` statement must be used:

`throw` *exception-name* [, *message* [, *object*]]

where the square brackets indicate optional parameters

To illustrate this form, suppose that `invert_x` is modified to accept an array object:

```
private define invert_x(x)
{
   variable i = where (x == 0);
   if (length (i))
```

```
            throw DivideByZeroError,
                  "Array contains elements that are zero", i;
         return 1/x;
      }
```

In this case, the message field of the exception object will contain the string `"Array contains elements that are zero"` and the object field will be set to the indices of the zero elements.

14.2.3 The finally block

The full form of the try-catch statement obeys the following syntax:

```
try [(opt-e)] { try-block-statements } catch Exception-List-1:
{ catch-block-1-statements } .  .  catch Exception-List-N: {
catch-block-N-statements } [ finally { finally-block-statements } ]
```

Here an exception-list is simply a list of exceptions such as:

```
catch OSError, RunTimeError:
```

The last clause of a try-statement is the *finally-block*, which is optional and is introduced using the `finally` keyword. If the try-statement contains no catch-clauses, then it must specify a finally-clause, otherwise a syntax error will result.

If the finally-clause is present, then its corresponding statements will be executed **regardless of whether an exception occurs**. If an exception occurs while executing the statements in the try-block, then the finally-block will execute after the code in any of the catch-blocks. The finally-clause is useful for freeing any resources (file handles, etc) allocated by the try-block regardless of whether an exception has occurred.

14.2.4 Creating new exceptions: the Exception Hierarchy

The following table gives the class hierarchy for the built-in exceptions.

```
AnyError
   OSError
      MallocError
      ImportError
   ParseError
      SyntaxError
      DuplicateDefinitionError
      UndefinedNameError
   RunTimeError
      InvalidParmError
      TypeMismatchError
      UserBreakError
      StackError
         StackOverflowError
         StackUnderflowError
```

```
ReadOnlyError
VariableUnitializedError
NumArgsError
IndexError
UsageError
ApplicationError
InternalError
NotImplementedError
LimitExceededError
MathError
   DivideByZeroError
   ArithOverflowError
   ArithUnderflowError
   DomainError
IOError
   WriteError
   ReadError
   OpenError
DataError
UnicodeError
InvalidUTF8Error
UnknownError
```

The above table shows that the root class of all exceptions is `AnyError`. This means that a catch block for `AnyError` will catch any exception. The `OSError`, `ParseError`, and `RunTimeError` exceptions are subclasses of the `AnyError` class. Subclasses of `OSError` include `MallocError`, and `ImportError`. Hence a handler for the `OSError` exception will catch `MallocError` but not `ParseError` since the latter is not a subclass of `OSError`.

The user may extend this tree with new exceptions using the `new_exception` function. This function takes three arguments:

```
new_exception (exception-name, baseclass, description);
```

The *exception-name* is the name of the exception, *baseclass* represents the node in the exception hierarchy where it is to be placed, and *description* is a string that provides a brief description of the exception.

For example, suppose that you are writing some code that processes numbers stored in a binary format. In particular, assume that the format specifies that data be stored in a specific byte-order, e.g., in big-endian form. Then it might be useful to extend the `DataError` exception with `EndianError`. This is easily accomplished via

```
new_exception ("EndianError", DataError, "Invalid byte-ordering");
```

This will create a new exception object called `EndianError` subclassed on `DataError`, and code that catches the `DataError` exception will additionally catch the `EndianError` exception.

Chapter 15

Loading Files: evalfile, autoload, and require

Chapter 16

Modules

16.1 Introduction

A module is a shared object that may be dynamically linked into the interpreter at run-time to provide the interpreter with additional intrinsic functions and variables. Several modules are distributed with the stock version of the **S-Lang** library, including a `pcre` module that allows the interpreter to make use of the *Perl Compatible Regular Expression library*, and a `png` module that allows the interpreter to easily read and write PNG files. There are also a number of modules for the interpreter that are not distributed with the library. See http://www.jedsoft.org/slang/modules/ for links to some of those.

16.2 Using Modules

In order to make use of a module, it must first be "imported" into the interpreter. There are two ways to go about this. One is to use the `import` function to dynamically link-in the specfied module, e.g.,

```
import ("pcre");
```

will dynamically link to the `pcre` module and make its symbols available to the interpreter using the active namespace. However, this is not the preferred method for loading a module.

Module writers are encouraged to distribute a module with a file of **S-Lang** code that performs the actual import of the module. Rather than a user making direct use of the `import` function, the preferred method of loading the modules is to load that file instead. For example the `pcre` module is distributed with a file called `pcre.sl` that contains little more than the `import("pcre")` statement. To use the `pcre` module, load `pcre.sl`, e.g.,

```
require ("pcre");
```

The main advantage of this approach to loading a module is that the functionality provided by the module may be split between intrinsic functions in the module proper, and interpreted functions contained in the `.sl` file. In such a case, loading the module via `import` would only provide partial

functionality. The `png` module provides a simple example of this concept. The current version of the `png` module consists of a couple intrinsic functions (`png_read` and `png_write`) contained in the shared object (`png-module.so`), and a number of other interpreted **S-Lang** functions defined in `png.sl`. Using the `import` statement to load the module would miss the latter set of functions.

In some cases, the symbols in a module may conflict with symbols that are currently defined by the interpreter. In order to avoid the conflict, it may be necessary to load the module into its own namespace and access its symbols via the namespace prefix. For example, the GNU Scientific Library Special Function module, `gslsf`, defines a couple hundred functions, some with common names such as `zeta`. In order to avoid any conflict, it is recommended that the symbols from such a module be imported into a separate namespace. This may be accomplished by specifying the namespace as a second argument to the `require` function, e.g.,

```
require ("gslsf", "gsl");
   .
   .
y = gsl->zeta(x);
```

This form requires that the module's symbols be accessed via the namespace qualifier `"gsl->"`.

Chapter 17

File Input/Output

S-Lang provides built-in support for two different I/O facilities. The simplest interface is modeled upon the C language *stdio* interface and consists of functions such as `fopen`, `fgets`, etc. The other interface is modeled on a lower level POSIX interface consisting of functions such as `open`, `read`, etc. In addition to permitting more control, the lower level interface permits one to access network objects as well as disk files.

17.1 Input/Output via stdio

17.1.1 Stdio Overview

The *stdio* interface consists of the following functions:

- `fopen`: opens a file for reading or writing.
- `fclose`: closes a file opened by `fopen`.
- `fgets`: reads a line from a file.
- `fputs`: writes text to a file.
- `fprintf`: writes formatted text to a file.
- `fwrite`: writes one of more objects to a file.
- `fread`: reads a specified number of objects from a file.
- `fread_bytes`: reads a specified number of bytes from a file and returns them as a string.
- `feof`: tests if a file pointer is at the end of the file.
- `ferror`: tests whether or not the stream associated with a file has an error.
- `clearerr`: clears the end-of-file and error indicators for a stream.
- `fflush`, forces all buffered data associated with a stream to be written out.

- `ftell`: queries the file position indicator a the stream.
- `fseek`: sets the position of a file position indicator of the stream.
- `fgetslines`: reads all the lines from a text file and returns them as an array of strings.

In addition, the interface supports the `popen` and `pclose` functions on systems where the corresponding C functions are available.

Before reading or writing to a file, it must first be opened using the `fopen` function. The only exceptions to this rule involve use of the pre-opened streams: `stdin`, `stdout`, and `stderr`. `fopen` accepts two arguments: a file name and a string argument that indicates how the file is to be opened, e.g., for reading, writing, update, etc. It returns a `File_Type` stream object that is used as an argument to all other functions of the *stdio* interface. Upon failure, it returns `NULL`. See the reference manual for more information about `fopen`.

17.1.2 Stdio Examples

In this section, some simple examples of the use of the *stdio* interface is presented. It is important to realize that all the functions of the interface return something, and that return value must be handled in some way by the caller.

The first example involves writing a function to count the number of lines in a text file. To do this, we shall read in the lines, one by one, and count them:

```
define count_lines_in_file (file)
{
   variable fp, line, count;

   fp = fopen (file, "r");    % Open the file for reading
   if (fp == NULL)
     throw OpenError, "$file failed to open"$;

   count = 0;
   while (-1 != fgets (&line, fp))
     count++;

   () = fclose (fp);
   return count;
}
```

Note that `&line` was passed to the `fgets` function. When `fgets` returns, `line` will contain the line of text read in from the file. Also note how the return value from `fclose` was handled (discarded in this case).

Although the preceding example closed the file via `fclose`, there is no need to explicitly close a file because the interpreter will automatically close a file when it is no longer referenced. Since the only variable to reference the file is `fp`, it would have automatically been closed when the function returned.

Suppose that it is desired to count the number of characters in the file instead of the number of lines. To do this, the `while` loop could be modified to count the characters as follows:

```
while (-1 != fgets (&line, fp))
  count += strlen (line);
```

The main difficulty with this approach is that it will not work for binary files, i.e., files that contain null characters. For such files, the file should be opened in *binary* mode via

```
fp = fopen (file, "rb");
```

and then the data read using the `fread` function:

```
while (-1 != fread (&line, Char_Type, 1024, fp))
     count += length (line);
```

The `fread` function requires two additional arguments: the type of object to read (`Char_Type` in the case), and the number of such objects to be read. The function returns the number of objects actually read in the form of an array of the specified type, or -1 upon failure.

Sometimes it is more convenient to obtain the data from a file in the form of a character string instead of an array of characters. The `fread_bytes` function may be used in such situations. Using this function, the equivalent of the above loop is

```
while (-1 != fread_bytes (&line, 1024, fp))
     count += bstrlen (line);
```

The `foreach` construct also works with `File_Type` objects. For example, the number of characters in a file may be counted via

```
foreach ch (fp) using ("char")
  count++;
```

Similarly, one can count the number of lines using:

```
foreach line (fp) using ("line")
 {
    num_lines++;
    count += strlen (line);
 }
```

Often one is not interested in trailing whitespace in the lines of a file. To have trailing whitespace automatically stripped from the lines as they are read in, use the `"wsline"` form, e.g.,

```
foreach line (fp) using ("wsline")
 {
     .
     .
 }
```

Finally, it should be mentioned that none of these examples should be used to count the number of bytes in a file when that information is more readily accessible by another means. For example, it is preferable to get this information via the `stat_file` function:

```
define count_chars_in_file (file)
{
   variable st;

   st = stat_file (file);
   if (st == NULL)
     throw IOError, "stat_file failed";
   return st.st_size;
}
```

17.2 POSIX I/O

17.3 Advanced I/O techniques

The previous examples illustrate how to read and write objects of a single data-type from a file, e.g.,

```
num = fread (&a, Double_Type, 20, fp);
```

would result in a `Double_Type[num]` array being assigned to `a` if successful. However, suppose that the binary data file consists of numbers in a specified byte-order. How can one read such objects with the proper byte swapping? The answer is to use the `fread_bytes` function to read the objects as a (binary) character string and then *unpack* the resulting string into the specified data type, or types. This process is facilitated using the `pack` and `unpack` functions.

The `pack` function follows the syntax

```
BString_Type pack (format-string, item-list);
```

and combines the objects in the *item-list* according to *format-string* into a binary string and returns the result. Likewise, the `unpack` function may be used to convert a binary string into separate data objects:

```
(variable-list) = unpack (format-string, binary-string);
```

The format string consists of one or more data-type specification characters, and each may be followed by an optional decimal length specifier. Specifically, the data-types are specified according to the following table:

```
c     char
C     unsigned char
h     short
H     unsigned short
i     int
I     unsigned int
l     long
L     unsigned long
j     16 bit int
J     16 unsigned int
```

```
k     32 bit int
K     32 bit unsigned int
f     float
d     double
F     32 bit float
D     64 bit float
s     character string, null padded
S     character string, space padded
z     character string, null padded
x     a null pad character
```

A decimal length specifier may follow the data-type specifier. With the exception of the `s` and `S` specifiers, the length specifier indicates how many objects of that data type are to be packed or unpacked from the string. When used with the `s` or `S` specifiers, it indicates the field width to be used. If the length specifier is not present, the length defaults to one.

With the exception of `c`, `C`, `s`, `S`, `z`, and `x`, each of these may be prefixed by a character that indicates the byte-order of the object:

```
>    big-endian order (network order)
<    little-endian order
=    native byte-order
```

The default is to use the native byte order.

Here are a few examples that should make this more clear:

```
a = pack ("cc", 'A', 'B');         % ==> a = "AB";
a = pack ("c2", 'A', 'B');         % ==> a = "AB";
a = pack ("xxcxxc", 'A', 'B');     % ==> a = "\0\0A\0\0B";
a = pack ("h2", 'A', 'B');         % ==> a = "\0A\0B" or "\0B\0A"
a = pack (">h2", 'A', 'B');        % ==> a = "\0\xA\0\xB"
a = pack ("<h2", 'A', 'B');        % ==> a = "\0B\0A"
a = pack ("s4", "AB", "CD");       % ==> a = "AB\0\0"
a = pack ("s4s2", "AB", "CD");     % ==> a = "AB\0\0CD"
a = pack ("S4", "AB", "CD");       % ==> a = "AB  "
a = pack ("S4S2", "AB", "CD");     % ==> a = "AB  CD"
```

When unpacking, if the length specifier is greater than one, then an array of that length will be returned. In addition, trailing whitespace and null characters are stripped when unpacking an object given by the `S` specifier. Here are a few examples:

```
(x,y) = unpack ("cc", "AB");         % ==> x = 'A', y = 'B'
x = unpack ("c2", "AB");             % ==> x = ['A', 'B']
x = unpack ("x<H", "\0\xAB\xCD");    % ==> x = 0xCDABuh
x = unpack ("xxs4", "a b c\0d e f");  % ==> x = "b c\0"
x = unpack ("xxS4", "a b c\0d e f");  % ==> x = "b c"
```

17.3.1 Example: Reading /var/log/wtmp

Consider the task of reading the Unix system file `/var/log/utmp`, which contains login records about who logged onto the system. This file format is documented in section 5 of the online Unix man

pages, and consists of a sequence of entries formatted according to the C structure `utmp` defined in the `utmp.h` C header file. The actual details of the structure may vary from one version of Unix to the other. For the purposes of this example, consider its definition under the Linux operating system running on an Intel 32 bit processor:

```
struct utmp {
   short ut_type;              /* type of login */
   pid_t ut_pid;               /* pid of process */
   char ut_line[12];           /* device name of tty - "/dev/" */
   char ut_id[2];              /* init id or abbrev. ttyname */
   time_t ut_time;             /* login time */
   char ut_user[8];            /* user name */
   char ut_host[16];           /* host name for remote login */
   long ut_addr;               /* IP addr of remote host */
};
```

On this system, `pid_t` is defined to be an `int` and `time_t` is a `long`. Hence, a format specifier for the `pack` and `unpack` functions is easily constructed to be:

```
 "h i S12 S2 l S8 S16 l"
```

However, this particular definition is naive because it does not allow for structure padding performed by the C compiler in order to align the data types on suitable word boundaries. Fortunately, the intrinsic function `pad_pack_format` may be used to modify a format by adding the correct amount of padding in the right places. In fact, `pad_pack_format` applied to the above format on an Intel-based Linux system produces the result:

```
 "h x2 i S12 S2 x2 l S8 S16 l"
```

Here we see that 4 bytes of padding were added.

The other missing piece of information is the size of the structure. This is useful because we would like to read in one structure at a time using the `fread` function. Knowing the size of the various data types makes this easy; however it is even easier to use the `sizeof_pack` intrinsic function, which returns the size (in bytes) of the structure described by the pack format.

So, with all the pieces in place, it is rather straightforward to write the code:

```
variable format, size, fp, buf;

typedef struct
{
   ut_type, ut_pid, ut_line, ut_id,
   ut_time, ut_user, ut_host, ut_addr
} UTMP_Type;

format = pad_pack_format ("h i S12 S2 l S8 S16 l");
size = sizeof_pack (format);

define print_utmp (u)
{
```

```
   () = fprintf (stdout, "%-16s %-12s %-16s %s\n",
                 u.ut_user, u.ut_line, u.ut_host, ctime (u.ut_time));
 }


fp = fopen ("/var/log/utmp", "rb");
if (fp == NULL)
  throw OpenError, "Unable to open utmp file";

() = fprintf (stdout, "%-16s %-12s %-16s %s\n",
                       "USER", "TTY", "FROM", "LOGIN@");

variable U = @UTMP_Type;

while (-1 != fread (&buf, Char_Type, size, fp))
  {
    set_struct_fields (U, unpack (format, buf));
    print_utmp (U);
  }

() = fclose (fp);
```

A few comments about this example are in order. First of all, note that a new data type called `UTMP_Type` was created, although this was not really necessary. The file was opened in binary mode, but this too was optional because under a Unix system where there is no distinction between binary and text modes. The `print_utmp` function does not print all of the structure fields. Finally, last but not least, the return values from `fprintf` and `fclose` were handled by discarding them.

Chapter 18

Debugging

There are several ways to debug a **S-Lang** script. When the interpreter encounters an uncaught exception, it can generate a traceback report showing where the error occurred and the values of local variables in the function call stack frames at the time of the error. Often just knowing where the error occurs is all that is required to correct the problem. More subtle bugs may require a deeper analysis to diagnose the problem. While one can insert the appropriate print statements in the code to get some idea about what is going on, it may be simpler to use the interactive debugger.

18.1 Tracebacks

When the value of the `_traceback` variable is non-zero, the interpreter will generate a traceback report when it encounters an error. This variable may be set by putting the line

```
_traceback = 1;
```

at the top of the suspect file. If the script is running in **slsh**, then invoking **slsh** using the `-g` option will enable tracebacks:

```
slsh -g myscript.sl
```

If `_traceback` is set to a positive value, the values of local variables will be printed in the traceback report. If set to a negative integer, the values of the local variables will be absent.

Here is an example of a traceback report:

```
Traceback: error
***string***:1:verror:Run-Time Error
/grandpa/d1/src/jed/lib/search.sl:78:search_generic_search:Run-Time Error
Local Variables:
      String_Type prompt = "Search forward:"
      Integer_Type dir = 1
      Ref_Type line_ok_fun = &_function_return_1
      String_Type str = "ascascascasc"
      Char_Type not_found = 1
```

```
      Integer_Type cs = 0
/grandpa/d1/src/jed/lib/search.sl:85:search_forward:Run-Time Error
```

There are several ways to read this report; perhaps the simplest is to read it from the bottom. This report says that on line 85 in `search.sl` the `search_forward` function called the `search_generic_search` function. On line 78 it called the `verror` function, which in turn called `error`. The `search_generic_search` function contains 6 local variables whose values at the time of the error are given by the traceback output. The above example shows that a local variable called `"not_found"` had a `Char_Type` value of 1 at the time of the error.

18.2 Using the sldb debugger

The interpreter contains a number of hooks that support a debugger. **sldb** consists of a set of functions that use these hooks to implement a simple debugger. Although written for **slsh**, the debugger may be used by other **S-Lang** interpreters that permit the loading of **slsh** library files. The examples presented here are given in the context of **slsh**.

In order to use the debugger, the code to to be debugged must be loaded with debugging info enabled. This can be in done several ways, depending upon the application embedding the interpreter.

For applications that support a command line, the simplest way to access the debugger is to use the `sldb` function with the name of the file to be debugged:

```
require ("sldb");
sldb ("foo.sl");
```

When called without an argument, `sldb` will prompt for input. This can be useful for setting or removing breakpoints.

Another mechanism to access the debugger is to put

```
require ("sldb");
sldb_enable ();
```

at the top of the suspect file. Any files loaded by the file will also be compiled with debugging support, making it unnecessary to add this to all files.

If the file contains any top-level executable statements, the debugger will display the line to be executed and prompt for input. If the file does not contain any executable statements, the debugger will not be activated until one of the functions in the file is executed.

As a concrete example, consider the following contrived **slsh** script called `buggy.sl`:

```
define divide (a, b, i)
{
   return a[i] / b;
}
define slsh_main ()
{
   variable x = [1:5];
```

```
   variable y = x*x;
   variable i;
   _for i (0, length(x), 1)
     {
        variable z = divide (x, y, i);
        () = fprintf (stdout, "%g/%g = %g", x[i], y[i], z);
     }
}
```

Running this via

```
slsh buggy.sl
```

yields

```
Expecting Double_Type, found Array_Type
./buggy.sl:13:slsh_main:Type Mismatch
```

More information may be obtained by using **slsh**'s `-g` option to cause a traceback report to be printed:

```
slsh -g buggy.sl
Expecting Double_Type, found Array_Type
Traceback: fprintf
./buggy.sl:13:slsh_main:Type Mismatch
Local variables for slsh_main:
     Array_Type x = Integer_Type[5]
     Array_Type y = Integer_Type[5]
     Integer_Type i = 0
     Array_Type z = Integer_Type[5]
Error encountered while executing slsh_main
```

From this one can see that the problem is that `z` is an array and not a scalar as expected.

To run the program under debugger control, startup **slsh** and load the file using the `sldb` function:

```
slsh> sldb ("./buggy.sl");
```

Note the use of `"./"` in the filename. This may be necessary if the file is not in the **slsh** search path.

The above command causes execution to stop with the following displayed:

```
slsh_main at ./buggy.sl:9
9    variable x = [1:5];
(sldb)
```

This shows that the debugger has stopped the script at line 9 of buggy.sl and is waiting for input. The `print` function may be used to print the value of an expression or variable. Using it to display the value of `x` yields

```
(sldb) print x
Caught exception:Variable Uninitialized Error
(sldb)
```

This is because `x` has not yet been assigned a value and will not be until line 9 has been executed. The `next` command may be used to execute the current line and stop at the next one:

```
(sldb) next
10    variable y = x*x;
(sldb)
```

The `step` command functions almost the same as `next`, except when a function call is involved. In such a case, the `next` command will step over the function call but `step` will cause the debugger to enter the function and stop there.

Now the value of `x` may be displayed using the `print` command:

```
(sldb) print x
Integer_Type[5]
(sldb) print x[0]
1
(sldb) print x[-1]
5
(sldb)
```

The `list` command may be used to get a list of the source code around the current line:

```
(sldb) list
5     return a[i] / b;
6  }
7  define slsh_main ()
8  {
9     variable x = [1:5];
10    variable y = x*x;
11    variable i;
12    _for i (0, length(x), 1)
13      {
14      variable z = divide (x, y, i);
15      () = fprintf (stdout, "%g/%g = %g", x[i], y[i], z);
```

The `break` function may be used to set a breakpoint. For example,

```
(sldb) break 15
breakpoint #1 set at ./buggy.sl:15
```

will set a break point at the line 15 of the current file.

The `cont` command may be used to continue execution until the next break point:

```
(sldb) cont
Breakpoint 1, slsh_main
    at ./buggy.sl:15
15      () = fprintf (stdout, "%g/%g = %g", x[i], y[i], z);
(sldb)
```

Using the `next` command produces:

```
Received Type Mismatch error.  Entering the debugger
15      () = fprintf (stdout, "%g/%g = %g", x[i], y[i], z);
```

This shows that during the execution of line 15, a `TypeMismatchError` was generated. Let's see what caused it:

```
(sldb) print x[i]
1
(sldb) print y[i]
1
(sldb) print z
Integer_Type[5]
```

This shows that the problem was caused by `z` being an array and not a scalar— something that was already known from the traceback report. Now let's see why it is not a scalar. Start the program again and set a breakpoint in the `divide` function:

```
slsh_main at ./buggy.sl:9
9    variable x = [1:5];
(sldb) break divide
breakpoint #1 set at divide
(sldb) cont
Breakpoint 1, divide
at ./buggy.sl:5
5    return a[i] / b;
(sldb)
```

The values of `a[i]/b` and `b` may be printed:

```
(sldb) print a[i]/b
Integer_Type[5]
(sldb) print b
Integer_Type[5]
```

From this it is easy to see that `z` is an array because `b` is an array. The fix for this is to change line 5 to

```
z = a[i]/b[i];
```

The debugger supports several other commands. For example, the `up` and `down` commands may be used to move up and down the stack-frames, and `where` command may be used to display the stack-frames. These commands are useful for examining the variables in the other frames:

```
(sldb) where
#0 ./buggy.sl:5:divide
#1 ./buggy.sl:14:slsh_main
(sldb) up
#1 ./buggy.sl:14:slsh_main
14      variable z = divide (x, y, i);
(sldb) print x
```

```
Integer_Type[5]
(sldb) down
#0 ./buggy.sl:5:divide
5    return a[i] / b;
(sldb) print z
Integer_Type[5]
```

On some operating systems, the debugger's `watchfpu` command may be used to help isolate floating point exceptions. Consider the following example:

```
define solve_quadratic (a, b, c)
{
   variable d = b^2 - 4.0*a*c;
   variable x = -b + sqrt (d);
   return x / (2.0*a);
}
define print_root (a, b, c)
{
   vmessage ("%f %f %f %f\n", a, b, c, solve_quadratic (a,b,c));
}
print_root (1,2,3);
```

Running it via **slsh** produces:

```
1.000000 2.000000 3.000000 nan
```

Now run it in the debugger:

```
<top-level> at ./example.sl:12
11 print_root (1,2,3);
(sldb) watchfpu FE_INVALID
(sldb) cont
*** FPU exception bits set: FE_INVALID
Entering the debugger.
solve_quadratic at ./t.sl:4
4    variable x = -b + sqrt (d);
```

This shows the the `NaN` was produced on line 4.

The `watchfpu` command may be used to watch for the occurrence of any combination of the following exceptions

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
```

by the bitwise-or operation of the desired combination. For instance, to track both `FE_INVALID` and `FE_OVERFLOW`, use:

```
(sldb) watchfpu FE_INVALID | FE_OVERFLOW
```

Chapter 19

Regular Expressions

The S-Lang library includes a regular expression (RE) package that may be used by an application embedding the library. The RE syntax should be familiar to anyone acquainted with regular expressions. In this section the syntax of the **S-Lang** regular expressions is discussed.

NOTE: At the moment, the **S-Lang** regular expressions do not support UTF-8 encoded strings. The **S-Lang** library will most likely migrate to the use of the PCRE regular expression library, deprecating the use of the **S-Lang** REs in the process. For these reasons, the user is encouraged to make use of the `pcre` module if possible.

19.1 S-Lang RE Syntax

A regular expression specifies a pattern to be matched against a string, and has the property that the contcatenation of two REs is also a RE.

The **S-Lang** library supports the following standard regular expressions:

```
.                 match any character except newline
*                 matches zero or more occurences of previous RE
+                 matches one or more occurences of previous RE
?                 matches zero or one occurence of previous RE
^                 matches beginning of a line
$                 matches end of line
[ ... ]           matches any single character between brackets.
                  For example, [-02468] matches '-' or any even digit.
                  and [-0-9a-z] matches '-' and any digit between 0 and 9
                  as well as letters a through z.
\<                Match the beginning of a word.
\>                Match the end of a word.
\( ... \)
\1, \2, ..., \9   Matches the match specified by nth \( ... \)
                  expression.
```

In addition the following extensions are also supported:

```
\c              turn on case-sensitivity (default)
\C              turn off case-sensitivity
\d              match any digit
\e              match ESC char
```

Here are some simple examples:

`"^int "` matches the `"int "` at the beginning of a line.

`"\<money\>"` matches `"money"` but only if it appears as a separate word.

`"^$"` matches an empty line.

A more complex pattern is

```
"\(\<[a-zA-Z]+\>\)[ ]+\1\>"
```

which matches any word repeated consecutively. Note how the grouping operators `\(` and `\)` are used to define the text matched by the enclosed regular expression, and then subsequently referred to `\1`.

Finally, remember that when used in string literals either in the **S-Lang** language or in the C language, care must be taken to "double-up" the `'\'` character since both languages treat it as an escape character.

19.2 Differences between S-Lang and egrep REs

There are several differences between **S-Lang** regular expressions and, e.g., **egrep** regular expressions.

The most notable difference is that the **S-Lang** regular expressions do not support the **OR** operator | in expressions. This means that `"a|b"` or `"a\|b"` do not have the meaning that they have in regular expression packages that support egrep-style expressions.

The other main difference is that while **S-Lang** regular expressions support the grouping operators `\(` and `\)`, they are only used as a means of specifying the text that is matched. That is, the expression

```
"@\([a-z]*\)@.*@\1@"
```

matches `"xxx@abc@silly@abc@yyy"`, where the pattern `\1` matches the text enclosed by the `\(` and `\)` expressions. However, in the current implementation, the grouping operators are not used to group regular expressions to form a single regular expression. Thus expression such as `"\(hello\)*"` is *not* a pattern to match zero or more occurrences of `"hello"` as it is in e.g., **egrep**.

One question that comes up from time to time is why doesn't **S-Lang** simply employ some posix-compatible regular expression library. The simple answer is that, at the time of this writing, none exists that is available across all the platforms that the **S-Lang** library supports (Unix, VMS, OS/2, win32, win16, BEOS, MSDOS, and QNX) and can be distributed under both the GNU licenses. It is particularly important that the library and the interpreter support a common set of regular expressions in a platform independent manner.

Appendix A

S-Lang 2 Interpreter NEWS

A.1 What's new for S-Lang 2

Here is a brief list of some of the new features and improvements in **S-Lang** 2.0.

- **slsh**, the generic **S-Lang** interpreter now supports and interactive command-line mode with readline support.
- Native support for Unicode via UTF-8 throughout the library.
- A `List_Type` object has been added to the language, e.g.,

  ```
  x = {1, 2.7, "foo", [1:10]};
  ```

  will create a (heterogeneous) list of 4 elements.
- A much improved exception handling model.
- Variable expansion within string literals:

  ```
  file = "$HOME/src/slang-$VERSION/"$;
  ```

- Operator overloading for user-defined types. For example it is possible to define a meaning to `X+Y` where `X` and `Y` are defined as

  ```
  typedef struct { x, y, z } Vector;
  define vector (x,y,z) { variable v = @Vector; v.x=x; v.y=y; v.z=z;}
  X = vector (1,2,3);
  Y = vector (4,5,6);
  ```

- Syntactic sugar for objected-oriented style method calls. **S-Lang** 1 code such as

  ```
  (@s.method)(s, args);
  ```

  may be written much more simply as

  ```
  s.method(args);
  ```

This should make "object-oriented" code somewhat more readable. See also the next section if your code uses constructs such as

```
@s.method(args);
```

because it is not supported by **S-Lang** 2.

- More intrinsic functions including math functions such as `hypot`, `atan2`, `floor`, `ceil`, `round`, `isnan`, `isinf`, and many more.
- Support for `long long` integers.

```
X = 18446744073709551615ULL;
```

- Large file support
- Performance improvements. The **S-Lang** 2 interpreter is about 20 percent faster for many operations than the previous version.
- Better debugging support including an interactive debugger. See the section on 18.2 (Using the sldb debugger) for more information.

See the relevent chapters in in the manual for more information.

A.2 Upgrading to S-Lang 2

For the most part **S-Lang** 2 is backwards-compatible with **S-Lang** 1. However there are a few important differences that need to be understood before upgrading to version 2.

++ and – operators in function calls

Previously the `++` and {–} operators were permitted in a function argument list, e.g.,

```
some_function (x++, x);
```

Such uses are flagged as syntax errors and need to be changed to

```
x++; some_function (x);
```

Array indexing of strings

Array indexing of strings uses byte-semantics and not character-semantics. This distinction is important only if UTF-8 mode is in effect. If you use array indexing with functions that use character semantics, then your code may not work properly in UTF-8 mode. For example, one might have used

```
i = is_substr (a, b);
if (i) c = a[[0:i-2]];
```

to extract that portion of `a` that preceeds the occurrence of `b` in `a`. This may nolonger work in UTF-8 mode where bytes and characters are not generally the same. The correct way to write the above is to use the `substr` function since it uses character semantics:

```
i = is_substr (a, b);
if (i) c = substr (a, 1, i-1);
```

Array indexing with negative integer ranges

Previously the interpretation of a range array was context sensitive. In an indexing situation `[0:-1]` was used to index from the first through the last element of an array, but outside this context, `[0:-1]` was an empty array. For **S-Lang 2**, the meaning of such arrays is always the same regardless of the context. Since by itself `[0:-1]` represents an empty array, indexing with such an array will also produce an empty array. The behavior of scalar indices has not changed: `A[-1]` still refers to the last element of the array.

Range arrays with an implied endpoint make sense only in indexing situations. Hence the value of the endpoint can be inferred from the context. Such arrays include `[*]`, `[:-1]`, etc.

Code that use index-ranges with negative valued indices such as

```
B = A[[0:-2]];    % Get all but the last element of A
```

will have to be changed to use an array with an implied endpoint:

```
B = A[[:-2]];     % Get all but the last element of A
```

Similarly, code such as

```
B = A[[-3:-1]];   % Get the last 3 elements of A
```

must be changed to

```
B = A[[-3:]];
```

Dereferencing function members of a structure

Support for the non-parenthesized form of function member dereferencing has been dropped. Code such as

```
@s.foo(args);
```

will need to be changed to use the parenthesized form:

```
(@s.foo)(args);
```

The latter form will work in both **S-Lang** 1 and **S-Lang** 2.

If your code passes the structure as the first argument of the method call, e.g.,

```
(@s.foo)(s, moreargs);
```

then it may be changed to

```
s.foo (moreargs);
```

However, this *objected-oriented* form of method calling is not supported by **S-Lang** 1.

ERROR_BLOCKS

Exception handling via `ERROR_BLOCKS` is still supported but deprecated. If your code uses `ERROR_BLOCKS` it should be changed to use the new exception handling model. For example, code that looks like:

```
ERROR_BLOCK { cleanup_after_error (); }
do_something ();
   .
   .
```

should be changed to:

```
variable e;
try (e)
  {
     do_something ();
       .
       .
  }
catch RunTimeError:
  {
     cleanup_after_error ();
     throw e.error, e.message;
  }
```

Code that makes use of `EXECUTE_ERROR_BLOCK`

```
ERROR_BLOCK { cleanup_after_error (); }
do_something ();
   .
   .
EXECUTE_ERROR_BLOCK;
```

should be changed to make use of a `finally` clause:

```
variable e;
try (e)
  {
     do_something ();
       .
       .
  }
finally
  {
     cleanup_after_error ();
  }
```

It is not possible to emulate the complete semantics of the `_clear_error` function. However, those semantics are flawed and fixing the problems associated with the use of `_clear_error` was one of the primary reasons for the new exception handling model. The main problem with the `_clear_error` method is that it causes execution to resume at the byte-code following the code that triggered the error. As such, `_clear_error` defines no absolute resumption point. In contrast, the try-catch exception model has well-defined points of execution. With the above caveats, code such as

```
ERROR_BLOCK { cleanup_after_error (); _clear_error ();}
do_something ();
   .
   .
```

should be changed to:

```
variable e;
try (e)
```

```
  {
     do_something ();
       .
       .
  }
catch RunTimeError:
  {
     cleanup_after_error ();
  }
```

And code using `_clear_error` in conjunction with `EXECUTE_ERROR_BLOCK`:

```
ERROR_BLOCK { cleanup_after_error (); _clear_error ();}
do_something ();
   .
   .
EXECUTE_ERROR_BLOCK;
```

should be changed to:

```
variable e;
try (e)
  {
     do_something ();
       .
       .
  }
catch RunTimeError:
  {
     cleanup_after_error ();
  }
finally:
  {
     cleanup_after_error ();
  }
```

fread

When reading `Char_Type` and `UChar_Type` objects the **S-Lang** 1 version of `fread` returned a binary string (`BString_Type` if the number of characters read was greater than one, or a `U/Char_Type` if the number read was one. In other words, the resulting type depended upon how many bytes were read with no way to predict the resulting type in advance. In contrast, when reading, e.g, `Int_Type` objects, `fread` returned an `Int_Type` when it read one integer, or an array of `Int_Type` if more than one was read. For **S-Lang** 2, the behavior of `fread` with respect to `UChar_Type` and `Char_Type` types was changed to have the same semantics as the other data types.

The upshot is that code that used

```
nread = fread (&str, Char_Type, num_wanted, fp)
```

will no longer result in `str` being a `BString_Type` if `nread > 1`. Instead, `str` will now become a `Char_Type[nread]` object. In order to read a specified number of bytes from a file in the form of a string, use the `fread_bytes` function:

```
#if (_slang_version >= 20000)
nread = fread_bytes (&str, num_wanted, fp);
#else
nread = fread (&str, Char_Type, num_wanted, fp)
#endif
```

The above will work with both versions of the interpreter.

strtrans

The `strtrans` function has been changed to support Unicode. One ramification of this is that when mapping from one range of characters to another, the length of the ranges must now be equal.

str_delete_chars

This function was changed to support unicode character classes. Code such as

```
y = str_delete_chars (x, "\\a");
```

is now implies the deletion of all alphabetic characters from `x`. Previously it meant to delete the backslashes and `a`s from from `x`. Use

```
y = str_delete_chars (x, "\\\\a");
```

to achieve the latter.

substr, is_substr, strsub

These functions use character-semantics and not byte-semantics. The distinction is important in UTF-8 mode. If you use array indexing in conjunction with these functions, then read on.

Appendix B

Copyright

The **S-Lang** library is distributed under the terms of the GNU General Public License.

B.1 The GNU Public License

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
                       59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble
```

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices
stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in
whole or in part contains or is derived from the Program or any
part thereof, to be licensed as a whole at no charge to all third
parties under the terms of this License.

```
c) If the modified program normally reads commands interactively
when run, you must cause it, when started running for such
interactive use in the most ordinary way, to print or display an
announcement including an appropriate copyright notice and a
notice that there is no warranty (or else, saying that you provide
a warranty) and that users may redistribute the program under
these conditions, and telling the user how to view a copy of this
License.  (Exception: if the Program itself is interactive but
does not normally print such an announcement, your work based on
the Program is not required to print an announcement.)
```

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

```
a) Accompany it with the complete corresponding machine-readable
source code, which must be distributed under the terms of Sections
1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three
years, to give any third party, for a charge no more than your
cost of physically performing source distribution, a complete
machine-readable copy of the corresponding source code, to be
distributed under the terms of Sections 1 and 2 above on a medium
customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer
to distribute corresponding source code.  (This alternative is
allowed only for noncommercial distribution and only if you
received the program in object code or executable form with such
an offer, in accord with Subsection b above.)
```

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus

any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.