A Guide to the S-Lang Language

John E. Davis, davis@space.mit.edu

Sun Jan 25 22:38:50 1998

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, the **slrn** newsreader, and **sldxe** (unreleased), a numerical computation program. For this reason, **S-Lang** does not exist as a separate application and many of 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** extension language, 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 **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 to write using a more traditional infix syntax making it easier to use for those unfamiliar with stack based languages. 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 several people to make the library portable and easy to use. In particular, I would like to thank the following individuals:

Luchesar Ionkov <lionkov@sf.cit.bg> 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 <olesen@weber.me.queensu.ca> 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 <j.burnell@irl.cri.nz> for the OS/2 port of the video and keyboard routines. He also made value suggestions regarding the interpreter interface.

Darrel Hankerson <hankedr@mail.auburn.edu> for cleaning up and unifying some of the code and the makefiles.

Dominik Wujastyk <ucgadkw@ucl.ac.uk> who was always willing to test new releases of the library.

Michael Elkins <me@muddcs.cs.hmc.edu> for his work on the curses emulation.

Ulli Horlacher <framstag@belwue.de> and Oezguer Kesim <kesim@math.fu-berlin.de> for the **S-Lang** newsgroup and mailing list.

Hunter Goatley, Andy Harper <Andy.Harper@kcl.ac.uk>, and Martin P.J. Zinser <m.zinser@gsi.de> for their VMS support.

I am also grateful to many other people who send in bug-reports and bug-fixes, for 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

1 Introduction **1**

1.1 Language Features . . . 1

1.2 Data Types and Operators . . . 1

1.3 Statements and Functions . . . 2

1.4 Error Handling . . . 2

1.5 Run-Time Library . . . 2

1.6 Input/Output . . . 2

1.7 Obtaining **S-Lang** . . . 3

2 Overview of the Language **5**

2.1 Variables and Functions . . . 5

2.2 Strings . . . 6

2.3 Referencing and Dereferencing . . . 7

2.4 Arrays . . . 9

2.5 Structures and User-Defined Types . . . 10

3 Data Types and Literal Constants **13**

3.1 Predefined Data Types . . . 13

3.1.1 Integers . . . 13

3.1.2 Floating Point Numbers . . . 14

3.1.3 Complex Numbers . . . 14

3.1.4 Strings . . . 15

3.1.5 Null_Type . . . 15

3.1.6 Ref_Type . . . 16

3.1.7 Array_Type and Struct_Type . . . 16

3.1.8 DataType_Type Type . . . 16

3.2 Typecasting: Converting from one Type to Another . . . 17

4 Identifiers **19**

5 Variables **21**

6 Operators **23**

6.1 Unary Operators 23

6.2 Binary Operators 24

6.2.1 Arithemetic Operators 24

6.2.2 Relational Operators 24

6.2.3 Boolean Operators 24

6.2.4 Bitwise Operators 25

6.2.5 Operator Precedence 26

6.2.6 Binary Operators and Functions Returning Multiple Values 26

6.3 Mixing Integer and Floating Point Arithmetic. 27

6.4 Short Circuit Boolean Evaluation 27

7 Statements **29**

7.1 Variable Declaration Statements 29

7.2 Assignment Statements 29

7.3 Conditional and Looping Statements 31

7.3.1 Conditional Forms 31

7.3.2 Looping Forms 34

7.4 break, return, continue 36

8 Functions **39**

8.1 Declaring Functions 39

8.2 Parameter Passing Mechanism 39

8.3 Referencing Variables 41

8.4 Functions with a Variable Number of Arguments 41

8.5 Returning Values 44

8.6 Multiple Assignment Statement 44

8.7 Exit-Blocks 46

9 Arrays **49**

9.1 Creating Arrays 49

9.2 Reshaping Arrays 50

9.3 Indexing Arrays 51
9.4 Arrays and Variables 53
9.5 Using Arrays in Computations 54

10 Structures and User-Defined Types **59**
10.1 Defining a Structure 59
10.2 Accessing the Fields of a Structure 60
10.3 Linked Lists 60
10.4 Defining New Types 62

11 Error Handling **65**
11.1 Error-Blocks 65
11.2 Clearing Errors 66

12 Loading Files: evalfile and autoload **69**

13 Input/Output **71**

14 Debugging **73**

15 Future Directions **75**

A Copyright **77**
A.1 The GNU Public License 77
A.2 The Artistic License 83

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 envisoned by the programmer, thus providing the application will 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, double precision floating point, and double precision complex numbers. In addition, it supports user defined structure types and multi-dimensional array types. To facilitate the construction of sophisticated data structures such as linked lists and trees, a 'reference' type was added to the language. 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 arithemetic 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. For example, the addition operator (`+`) has been extended to work between string types to permit string concatenation.

The binary and unary operators work transparantly 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`, 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 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).

1.4 Error Handling

The **S-Lang** language defines a construct called an *error-block* that may be used for error handling and recovery. When a non-fatal run-time error is encountered, any error blocks that have been defined are executed as the run-time stack unwinds. An error block can optionally clear the error and the program will continue running after the statement that triggered the error. This mechanism is somewhat similar to try-catch in C++.

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 arithemetic, 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.

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 S-Lang

S-Lang as well as some programs that embed it are freely available via anonymous ftp in the United States from

- `ftp://space.mit.edu/pub/davis`.

It is also available outside the United States from the following mirror sites:

- `ftp://ftp.uni-stuttgart.de/pub/unix/misc/slang/`
- `ftp://ftp.fu-berlin.de/pub/unix/news/slrn/`
- `ftp://ftp.ntua.gr/pub/lang/slang/`

The library may be obtained from the World Wide Web at `http://space.mit.edu/\%7Edavis/slang.html`.

The Usenet newsgroup `alt.lang.s-lang` was created for **S-Lang** programmers to exchange information and share macros for the various programs the embed the language. The newsgroup `comp.editors` can be a useful resource for **S-Lang** macros for the **jed** editor. Similary, **slrn** users will find `news.software.readers` to be a valuable source of information.

Finally, two mailing lists dealing with the **S-Lang** library have been created:

- `slang-announce@babayaga.math.fu-berlin.de`
- `slang-workers@babayaga.math.fu-berlin.de`

The first list is for announcements of new releases of the library, while the second list is intended for those who use the library for their own code development. To subscribe to the announcement list, send an email to `slang-announce-subscribe@babayaga.math.fu-berlin.de` and include the word `subscribe` in the body of the message. To subscribe to the developers list, use the address `slang-workers-subscribe@babayaga.math.fu-berlin.de`.

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 string, 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 parameter list when declaring a function in this way.

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 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, strcat (b, c));
}
```

This function uses the built-in `strcat` function for concatenating two strings. In C, the simplist 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 section ??? 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, sum;

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

```
      }
    return sum;
 }

 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 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 = Integer_Type [10];
variable B = Integer_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 an *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 which can initialize arrays. For example,

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

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

variable A = Integer_Type [10];
init_array (A, 10);
```

creates an array of `10` integers and initializes all its elements elements to `7`. For simplicity, the number of rows was explicitly passed to the function, however, it is possible to get this information using the `array_info` function.

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

```
variable A = [7, 7, 7, 7, 7, 7, 7, 7, 7, 7];
variable A = Integer_Type [10];  A[[0:9]] = 7;
variable A = Integer_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 an implicit form. See section ??? 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.,

```
variable A = Double_Type [10];
variable B = Complex_Type [10];
variable C = String_Type [10];
variable 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 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, 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;
```

The big advantage of creating a new type is that one can go on to create arrays of the data type

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

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 section ???.

Chapter 3

Data Types and Literal Constants

The current implementation of the **S-Lang** language permits up to 256 distinct data types, including predefined data types such as integer and floating point, as well as specialized applications 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 basic **S-Lang** language supports integers only as signed values. Unsigned and long integers are not part of the basic language, though it is possible for applications to define them. 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, integers are 16 bit quantities with a range of -32767 to 32767. On a 32 bit system, integers range from -2147483648 to 2147483647.

An 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 hexidecimal (base 16) notation consisting of the characters `0` to `9` and `A` through `F`. The hexidecimal number must be preceded by the characters `0x`. For example, `0x7F` specifies an integer using hexidecimal 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.

- Using character notation containing a character enclosed in single quotes as `'a'`. The value of the integer 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 integer 48 since the `'0'` character has an ASCII value of 48.

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

3.1.2 Floating Point Numbers

Double precision floating point literals must contain either a decimal point or an exponent (or both). Here are examples of specifying the same floating 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. This suffix will be ignored unless the application provides a single precision floating point type. The current implementation only provides support for double precision floating point numbers.

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 and a more generic complex number with a non-zero real part may 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. Actually, `12i` is really an imaginary integer except that **S-Lang** automatically promotes it to a double precision 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".
```

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 go beyond this limit by string concatenation, e.g.,

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

Any character except a newline (ASCII 10) or the null character (ASCII 0) may appear explicitly in a string literal. However, these characters may be used implicitely 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 --  character expressed in HEXIDECIMAL notation
\ooo  --  character expressed in OCTAL notation
\dnnn --  character expressed in DECIMAL
```

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\""
```

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

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

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,

```
variable sin_ref = &sin;
variable 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 is also used to create an instance of a structure. For these reasons, further discussion of this important type can be found in section ??? and section ???.

3.1.7 Array_Type and Struct_Type

Variables of type `Array_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. See ???.

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:

```
Integer_Type        (integers)
Double_Type         (double precision reals)
Complex_Type        (complex numbers)
```

```
String_Type         (strings)
Struct_Type         (structures)
Ref_Type            (references)
Null_Type           (NULL)
Array_Type          (arrays)
DataType_Type       (data types)
```

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

The built-in function `typeof` returns 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`.

3.2 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,

```
variable x = [1:10];       % Array of integers
variable 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.

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.,

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

The `string` intrinsic function is similar to the typecast function except that it converts an object to a string representation. It is important to understand that a typecast from some type to `String_Type` is *not* the same as converting an object to its string operation. That is, `typecast(x,String_Type)` is not equivalent to `string(x)`. The reason for this is that when given an array, the `typecast` function acts on each element of the array to produce another array, whereas the `string` function produces a a string.

The `string` function is useful for printing 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 a letter ([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 real number 2.0.

Although the maximum length of identifiers is unspecified by the language, the length should be kept below 64 characters.

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

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

Chapter 5

Variables

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. The actual type checking 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 unitialized 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.

The following global variables are predefined by the language and 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 section ???.

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 arithemetic 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 in one of two classes: postfix-unary or prefix-unary. For example, in the expression `-x`, the minus sign is a prefix-unary operator.

Not all data types have binary or unary operations defined. For example, while `String_Type` objects support the `+` operator, they do 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 arithemetic negation operator `-` is 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 section ???. Similary, 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.

All binary and unary operators may be overloaded. For example, the arithemetic plus operator has been overloaded by the `String_Type` data type to permit concatenation between strings.

6.2.1 Arithemetic Operators

The arithemetic operators include `+`, `-`, `*`, `/`, which perform addition, subtraction, multiplication, and division, respectively. In addition to these, **S-Lang** supports the `mod` operator 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 value 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. The result of one of these comparisons is the integer `1` if the comparison is true, or `0` if the comparison is false. For example, `6 >= 5` returns `1`, but `6 == 5` produces `0`.

6.2.3 Boolean Operators

There are only two boolean binary operators: `or` and `and`. These operators are defined only for integers and produce an integer result. The `or` operator returns `1` if either of its operands are non-zero, otherwise it produces `0`. The `and` operator produces `1` if and only if both its operands are non-zero, otherwise it produces `0`.

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 abouth these constructs.

6.2.4 Bitwise Operators

The bitwise binary operators are defined only with 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 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. 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 has all of its bits set to `0`, except for the zeroth one, which is set to `1`. (At this point, it should also be apparant 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 Operator Precedence

6.2.6 Binary Operators and Functions Returning Multiple Values

Care must be exercised when using binary operators with an operand the 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, 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 occurs 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.

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 float, the other is converted to float and the result is float. For example:

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

Finally note that only integers may be used as array indices, loop control variables, and bit operations. The conversion functions, `int` and `float`, may be used convert between floats and ints where appropriate, e.g.,

```
int (1.5)        --> 1 (integer)
float(1.5)       --> 1.5 (float)
float (1)        --> 1.0 (float)
```

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 *statment* is composed of *expressions* that are grouped according to the syntax or grammar of the language to express a complete computation. Statements are analogous to sentences in a human language and expressions are like phrases. All statements in the **S-Lang** language must end in a semi-colon.

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 statments 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 chapter ???. 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 assignement can be performed. Such an object is called an *lvalue*.

The most common assignment operator is the simple assignment operator =. Simple 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 assignment operators `+=` and `-=`. Internally, **S-Lang** transforms

```
a += b;
```

to

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

Similarly, `a -= b` is transformed to `a = a - b`. It is extremely important to realize that, in general, `a+b` is not equal to `b+a`. This means that `a+=b` is not the same as `a=b+a`. As an example consider

```
a = "hello"; a += "world";
```

After executaion of these two statements, `a` will have the value `"helloworld"` and not `"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 section ???.

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

```
if (x > 0) z = log (x); else error ("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. The grammar presented above shows that the 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;
      }
  }
```

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 statemnts. 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 }
```

returns `0` since the third block will be the last to execute.

switch

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

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

The '`:`' operator is a special symbol which means to test the top item on the stack, 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 : print("Number is one.");}
  { x == 2 : print("Number is two.");}
  { x == 3 : print("Number is three.");}
  { x == 4 : print("Number is four.");}
  { x == 5 : print("Number is five.");}
  { print ("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 used the `case` keyword:

```
switch (x)
  { case 1 : print("Number is one.");}
  { case 2 : print("Number is two.");}
  { case 3 : print("Number is three.");}
  { case 4 : print("Number is four.");}
```

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

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

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

will loop forever since `i` will never be 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 ();
```

The fact is that expressions such as `i--` do not return a value in **S-Lang** as they do in C. If you must write this way, use the comma operator as in

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

do...while

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

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

The main difference between this statment 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 statment is the `for` statement; nevertheless, it is a favorite of many programmers. This statment 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* returns 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 statment 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 apparant complexity of the `for` statement, it is very easy to use. As an example, consider

```
sum = 0;
for (i = 1; i <= 10; i++) sum += 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-statment
```

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 cause the function `newline` to get called 10 times.

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--;
  }
```

7.4 break, return, continue

S-Lang also includes the non-local transfer functions `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

A function may be thought of as a group of statments that work together to perform a computation. While there are no imposed limits upon the number statements that may occur within a function, it is considered poor programming practice if a function contains many satements. This notion stems from the belief that a function should have a simple, well defined purpose.

8.1 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 chapter ???. 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 body of the function is enclosed in braces and consists of zero or more statments (*statement-list*).

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.

8.2 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 it is 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` would retain its value of zero after the function call.

S-Lang does provide a mechanism for simulating pass by reference via the reference operator. See the next section for more details.

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, we can omit one or both of the parameters by calling it 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; however, at least one of the parameters was omitted in the other examples. The interpreter will implicitly convert the last three examples 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 specifiy 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 single parameter function calls is described in the next section.

8.3 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 we write it 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 `a` will be a reference to `b`. It is only by *dereferencing* this value that `b` can be accessed and changed. So, the statement `@a=@a+10;` should be read "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.

One of the main purposes for references is that this mechanism allows reference to functions to be 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;
}
```

It can be used to differentiate the function

```
define x_squared (x)
{
   return x^2;
}
```

at the point `x = 3` via the expression `derivative(&x_squared,3)`.

8.4 Functions with a Variable Number of Arguments

S-Lang functions may be defined to take a variable number of arguments. The reason for this is that the calling routine pushes the arguments onto the stack before making a function call, and it is up to the called function to pop the values off the stack and make assignments to the variables in the parameter list. These details are, for the most part, hidden from the programmer. However, they are important when a variable number of arguments are passed.

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

A generic function of the form

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

is internally transformed by the interpreter to

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

before futher 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 sum;

   if (n == 1)
```

```
      {
         x = ();
         sum = x;
      }
    else if (n == 2)
      {
         y = ();
         x = ();
         sum = x + y;
      }
    else error ("average_n: only one or two values supported");

    return sum / 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 refecting 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 sum, x;
   sum = 0;
   loop (n)
     {
        x = ();    % get next value from stack
        sum += x;
     }
   return sum / n;
}
```

The principle 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, sum = 0;

   if (_NARGS == 0) error ("Usage: ave = average_n (x, ...);");

   loop (_NARGS)
     {
        x = ();
        sum += x;
     }
```

```
      return sum / _NARGS;
   }
```

Here, if no arguments are passed to the function, a simple message that indicates how it is to be used is printed out.

8.5 Returning Values

As stated earlier, the usual way to return values from a function is via the 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. As an example of a function that can return multiple values, consider

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

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

which is a function returning two values.

It is extremely important to note that *the calling routine must explicitly handle all values returned by a function*. Although some languages such as C do not have this restriction, **S-Lang** does and it is a direct result of a **S-Lang** function's ability to return many values and accept a variable number of parameters. Examples of properly handling the above function include

```
variable sum, diff;
(sum, diff) = sum_and_diff (5, 4);  % ignore neither
(sum, ) = sum_and_diff (5, 4);      % ignore diff
(,) = sum_and_diff (5, 4);          % ignore both sum and diff
```

See the section below on assignment statements for more information about this important point.

8.6 Multiple Assignment Statement

S-Lang functions can return more than one value, e.g.,

```
define sum_and_diff (x, y)
{
   return x + y, x - y;
}
```

returns two values. It accomplishes this by placing both values on the stack before returning. If you understand how **S-Lang** functions handle a variable number of parameters (section ???), then it should be rather obvious that one assigns such values to variables. One way is to use, e.g.,

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

However, the most convenient way to accomplish this is to use a *multiple assignment statement* such as

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

The most general form of the multiple assignment statement is

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

In fact, internally the interpreter transforms this statement into the form

```
expresssion; var_n = (); ... var_2 = (); var_1 = ();
```

for futher processing.

If you do not care about one of return values, simply omit the variable name from the list. For example,

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

assigns the sum of `9` and `4` to `s` and the difference (`9-4`) will be removed from the stack.

As another example, the **jed** editor provides a function called `down` that takes an integer argument and returns an integer. It is used to move the current editing position down the number of lines specified by the argument passed to it. It returns the number of lines it sucessfully moved the editing position. Often one does not care about the return value from this function. Although it is always possibly to handle the return value via

```
variable dummy = down (10);
```

it is more convenient to use a multiple assignment expression and omit the variable name, e.g.,

```
() = down (10);
```

Some functions return a *variable number* of values instead of a *fixed number*. 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 as in the following example:

```
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`.

Before leaving this section it is important to reiterate the fact that if a function returns a value, the caller must deal with that return value. Otherwise, the value will continue to live onto the stack and may eventually lead to a stack overflow error. Failing to handle the return value of a function is the most common mistake that inexperienced **S-Lang** programmers make. For example, the `fflush` function returns a value that many C programmer's never check. Instead of writing

```
fflush (fp);
```

as one could in C, a **S-Lang** programmer should write

```
() = fflush (fp);
```

in **S-Lang**. (Many good C programmer's write `(void)fflush(fp)` to indicate that the the return value is being ignored).

8.7 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 executed. 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 exection. 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, although nested exit-blocks are illegal.

Chapter 9

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.

9.1 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* use the syntax

```
SomeType [dim0, dim1, ..., dimN]
```

Here *dim0*, *dim1*, ... *dimN* specify the size of the individual dimensions of the array. The current implementation permits arrays consist of up to `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., `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`.

There is a more convenient syntax for creating and initializing a 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 most general form of a range array is

```
[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. The number of elements in the array is given by the expression

```
1 + (last-value - first-value)/increment
```

Another way to create an array is 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. We shall encounter it again in section ??? in the context of the `array_info` function.

9.2 Reshaping Arrays

It is sometimes possible 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 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 have 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.

9.3 Indexing Arrays

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 a range error.

A negative index may be used to index from the end of the array, with `a[-1]` referring to the last element of `a`, `a[-2]` referring 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[3] = 6;
```

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

```
y = a[3] + 7;
```

Unlike other **S-Lang** variables which inherit a type upon assignment, array elements already have a type. For example, an attempt to assign a string value to an element of an integer array will result in a type-mismatch error.

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

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

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 statment `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 understand the syntax used and, in particular, to note that `a[[0:499]]` is *not* the same as `a[0:499]`. In fact, the latter will generate a syntax error.

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`.

Now consider a multi-dimensional array. For simplicity, suppose that `a` is a `100` by `100` array of doubles. Then the expression `a[0, [:]]` specifies all elements in the zeroth row. Similary, `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`.

We conclude this section with a few examples.

Here is a function that computes the trace (sum of the diagonal elements) of a square 2 dimensional `n` by `n` array:

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

This fragment 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];
for (j = 0; j < 10; j++) a[j, j] = 5;
the_trace = array_trace(a, 10);
```

We can get rid of the `for` loop as follows:

```
j = Integer_Type [10, 2];
j[[:],0] = [0:9];
j[[:],1] = [0:9];
a[j] = 5;
```

Here, the goal was to construct a 2-d array of indices that correspond to the diagonal elements of a, and then use that array to index a. To understand how this works, consider the middle statements. They are equivalent to the following for loops:

```
variable i;
for (i = 0; i < 10; i++) j[i, 0] = i;
for (i = 0; i < 10; i++) j[i, 1] = i;
```

Thus, row n of j will have the value (n,n), which is precisely what was sought.

Another example of this technique is the function:

```
define unit_matrix (n)
{
   variable a = Integer_Type [n, n];
   variable j = Integer_Type [n, 2];
   j[[:],0] = [0:n - 1];
   j[[:],1] = [0:n - 1];

   a[j] = 1;
   return a;
}
```

This function creates an creates an n by n unit matrix, that is a 2-d n by n array whose elements are all zero except on the disgonal where they have a value of 1.

9.4 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 connsequences, some 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, n)
{
   variable i;

   for (i = 0; i < n; i++) 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, 100000);
```

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:

```
variable a, b;
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 one variable which represents an array, to another variable results in the assignment of a reference to the 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:

```
variable a, b;
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, 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.,

```
variable a, b;
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.

9.5 Using Arrays in Computations

Many functions and operations work transparantly 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 known what subset of the coefficients, a, b, c, correspond to real-valued solutions. In terms of for loops, we can write:

```
variable i, d, index_array;
index_array = Integer_Type [n];
for (i = 0; i < n; i++)
  {
     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:

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

S-Lang has a powerful built-in function called where. This function takes an array of integers and returns a 2-d array of indices that correspond to where the elements of the input array are non-zero. This simple operation is extremely useful. For example, suppose a is a 1-d array of n doubles, and it is desired to set to zero all elements of the array whose value is less than zero. One way is to use a for loop:

```
for (i = 0; i < n; i++)
  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 an 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 where which returns a 2-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.

As a final example, 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:

```
variable i, j, nn, tmp_a, tmp_b, tmp_c;

nn = 0;
for (i = 0; i < n; i++)
  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; i < n; i++)
  {
     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 alot of code, it is also clumsy and error-prone. Using the `where` function, this task is trivial:

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

All the examples up to now assumed that the dimensions of the array were known. However, the function `array_info` may be used to get information about an array, such as its data type and size. The function returns three values: the data type, the number of dimensions, and an integer array containing the size of each dimension. It may be used to determine the number of rows of an array as follows:

```
define num_rows (a)
{
   variable dims, type, num_dims;

   (dims, num_dims, type) = array_info (a);
   return dims[0];
}
```

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

```
define num_cols (a)
{
   variable dims, type, num_dims;

   (dims, num_dims, type) = array_info (a);
   if (num_dims > 1) return dims[1];
```

```
      return 1;
   }
```

Another use of `array_info` is to create an array that has the same number of dimensions as another array:

```
   define make_int_array (a)
   {
      variable dims, num_dims, type;

      (dims, num_dims, type) = array_info (a);
      return @Array_Type (Integer_Type, dims);
   }
```

Chapter 10

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.

10.1 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 stucture with N fields whose names are specified by *field-name-1*, *field-name-2*, ..., *field-name-N*. When a structure is created, all its fields are initialized to `NULL`.

For example,

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

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

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

```
variable u = t;
```

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

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

10.2 Accessing the Fields of a Structure

The dot (`.`) operator is used to specifiy the a 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

```
variable 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 statments involving the fields of `t`.

10.3 Linked Lists

One of the most important uses of structures is to create a *dynamic* data structure such as a *linked-list*. 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:

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

and suppose that we desire to create a list of such structures. 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 we can create the list via:

```
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:

```
variable Population_List = create_population_list ();
if (Population_List == NULL) error ("List is empty");
```

We can create other functions that manipulate the list. An example is a function 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.

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 and intuitive.

```
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 algorihm would fail.

10.4 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 essentually 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;
```

Chapter 11

Error Handling

Many intrinsic functions signal errors in the event of failure. User defined functions may also generate an error condition via the `error` function. Depending upon the severity of the error, it can be caught and cleared using a construct called an *error-block*.

11.1 Error-Blocks

When the interpreter encounters a recoverable run-time error, it will return to top-level by *unwinding* its function call stack. Any error-blocks that it encounters as part of this unwinding process will get executed. Errors such as syntax errors and memory allocation errors are not recoverable, and error-blocks will not get executed when such errors are encountered.

An error-block is defined using the syntax

```
ERROR_BLOCK { statement-list }
```

where *statement-list* represents a list of statements that comprise the error-block. A simple example of an error-block is

```
define simple (a)
{
   ERROR_BLOCK { message ("error-block executed"); }
   if (a) error ("Triggering Error");
   message ("hello");
}
```

Executing this function via `simple(0)` will result in the message `"hello"`. However, calling it using `simple(1)` will generate an error that will be caught, but not cleared, by the error-block and the `"error-block executed"` message will result.

Error-blocks are never executed unless triggered by an error. The only exception to this is when the user explicitly indicates that the error-block in scope should execute. This is indicated by the special keyword `EXECUTE_ERROR_BLOCK`. For example, `simple` could be recoded as

```
define simple (a)
{
   variable err_string = "error-block executed";
   ERROR_BLOCK { message (err_string); }
   if (a) error ("Triggering Error");
   err_string = "hello";
   EXECUTE_ERROR_BLOCK;
}
```

Please note that `EXECUTE_ERROR_BLOCK` does not initiate an error condition; it simply causes the error-block to be executed and control will pass onto the next statement following the `EXECUTE_ERROR_BLOCK` statement.

11.2 Clearing Errors

Once an error has been caught by an error-block, the error can be cleared by the `_clear_error` function. After the error has been cleared, execution will resume at the next statement at the level of the error block following the statement that generated the error. For example, consider:

```
define make_error ()
{
    error ("Error condition created.");
    message ("This statement is not executed.");
}

define test ()
{
    ERROR_BLOCK
      {
         _clear_error ();
      }
    make_error ();
    message ("error cleared.");
}
```

Calling `test` will trigger an error in the `make_error` function, but will get cleared in the `test` function. The call-stack will unwind from `make_error` back into `test` where the error-block will get executed. As a result, execution resumes after the statement that makes the call to `make_error` since this statement is at the same level as the error-block that cleared the error.

Here is another example that illustrates how multiple error-blocks work:

```
define example ()
{
   variable n = 0, s = "";
   variable str;

   ERROR_BLOCK {
       str = sprintf ("s=%s,n=%d", s, n);
```

```
        _clear_error ();
    }

    forever
      {
        ERROR_BLOCK {
         s += "0";
         _clear_error ();
        }

        if (n == 0) error ("");

        ERROR_BLOCK {
         s += "1";
        }

        if (n == 1) error ("");
        n++;
      }
    return str;
}
```

Here, three error-blocks have been declared. One has been declared outside the `forever` loop and the other two have been declared inside the `forever` loop. Each time through the loop, the variable `n` is incremented and a different error-block is triggered. The error-block that gets triggered is the last one encountered, since that will be the one in scope. On the first time through the loop, `n` will be zero and the first error-block in the loop will get executed. This error block clears the error and execution resumes following the `if` statement that triggered the error. The variable `n` will get incremented to `1` and, on the second cycle through the loop the second `if` statement will trigger an error causing the second error-block to execute. This time, the error is not cleared and the call-stack unwinds out of the `forever` loop, at which point the error-block outside the loop is in scope, causing it to execute. This error-block prints out the values of the variables `s` and `n`. It will clear the error and execution resumes on the statement *following* the `forever` loop. The result of this complicated series of events is that the function will return the string `"s=01,n=1"`.

Chapter 12

Loading Files: evalfile and autoload

Chapter 13

Input/Output

Chapter 14

Debugging

The current implementation provides no support for an interactive debugger, although a future version will. Nevertheless, **S-Lang** has several features that aid the programmer in tracking down problems, including function call tracebacks and the tracing of function calls. However, the biggest debugging aid stems from the fact that the language is interpreted permitting one to easily add debugging statements to the code.

Chapter 15

Future Directions

Appendix A

Copyright

The **S-Lang** library is distributed under two copyrights: the GNU Genral Public License, and the Artistic License. Any program that uses the interpreter must adhere to rules of one of these licenses.

A.1 The GNU Public License

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
```

Appendix: 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., 675 Mass Ave, Cambridge, MA 02139, 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.

A.2 The Artistic License

```
                        The "Artistic License"

                               Preamble
```

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions:

```
        "Package" refers to the collection of files distributed by the
        Copyright Holder, and derivatives of that collection of files
        created through textual modification.

        "Standard Version" refers to such a Package if it has not been
        modified, or has been modified in accordance with the wishes
        of the Copyright Holder as specified below.

        "Copyright Holder" is whoever is named in the copyright or
        copyrights for the package.

        "You" is you, if you're thinking about copying or distributing
        this Package.

        "Reasonable copying fee" is whatever you can justify on the
        basis of media cost, duplication charges, time of people involved,
        and so on.  (You will not be required to justify it to the
        Copyright Holder, but only to the computing community at large
        as a market that must bear the fee.)

        "Freely Available" means that no fee is charged for the item
        itself, though there may be fees involved in handling the item.
        It also means that recipients of the item may redistribute it
        under the same conditions they received it.
```

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.

2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.

3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:

```
    a) place your modifications in the Public Domain or otherwise make them
```

```
Freely Available, such as by posting said modifications to Usenet or
an equivalent medium, or placing the modifications on a major archive
site such as uunet.uu.net, or by allowing the Copyright Holder to include
your modifications in the Standard Version of the Package.

b) use the modified Package only within your corporation or organization.

c) rename any non-standard executables so the names do not conflict
with standard executables, which must also be provided, and provide
a separate manual page for each non-standard executable that clearly
documents how it differs from the Standard Version.

d) make other distribution arrangements with the Copyright Holder.
```

4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:

```
a) distribute a Standard Version of the executables and library files,
together with instructions (in the manual page or equivalent) on where
to get the Standard Version.

b) accompany the distribution with the machine-readable source of
the Package with your modifications.

c) give non-standard executables non-standard names, and clearly
document the differences in manual pages (or equivalent), together
with instructions on where to get the Standard Version.

d) make other distribution arrangements with the Copyright Holder.
```

5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's interpreter within an executable of yours (by linking); this shall be construed as a mere form of aggregation, provided that the complete Standard Version of the interpreter is so embedded.

6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whomever generated them, and may be sold commercially, and may be aggregated with this Package. If such scripts or library files are aggregated with this Package via the so-called "undump" or "unexec" methods of producing a binary executable image, then distribution of such an image shall neither be construed as a distribution of this Package nor shall it fall under the restrictions of Paragraphs 3 and 4, provided that you do not represent such an executable image as a Standard Version of this Package.

7. C subroutines (or comparably compiled subroutines in other languages) supplied by you and linked into this Package in order to emulate subroutines and variables of the language defined by this Package shall not be considered part of this Package, but are the equivalent of input as in

Paragraph 6, provided these subroutines do not change the language in any way that would cause it to fail the regression tests for the language.

8. Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package.

9. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.

10. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.