An Overview of SL5*

Ralph E. Griswold and David R. Hanson+

Department of Computer Science, The University of Arizona,
Tucson, Arizona 85721

1. Introduction

SL5 is a programming language designed primarily as a tool for programming-language research, mainly in the areas of string and list processing. SL5 derives many of its characteristics from SNOBOL4 [1] and recent programming-language research [2-7]. Although SL5 has clear roots in SNOBOL4, its primary role is in research; it is not intended as a replacement for SNOBOL4 or as a step in the evolution of the SNOBOL languages. In fact, many of the features of SL5 are esoteric, and the generality of the implementation necessary to support research applications makes SL5 inappropriate for general-purpose use.

Structurally, SL5 is very different from SNOBOL4 -- SL5 is an expression-oriented language with many of the conventional control mechanisms (although some have novel interpretations). Among the notable characteristics of SL5 is its procedure mechanism, which permits the decomposition of procedure invocation into more elementary components. This mechanism provides a natural method for coroutine programming. Runtime flexibility is an important aspect of SL5 and is motivated by the research orientation of its intended applications. SL5 has no type declarations and supports a variety of data types with runtime coercion and checking for appropriateness of type in context. Scoping of identifiers is dynamic and establishes bindings at runtime that permit flexible use of coroutine methodologies.

SL5 is designed to support programming language research at two levels. At the higher level, the SL5 language provides a tool for exploring a variety of programming techniques that are awkward or impractical in other languages. At the lower, implementation level, SL5 is organized into modules that are bound at load time. There are a number of features that constitute the SL5 language nucleus. Other modules can be developed independently and appended to the nucleus to provide language variants and extensions. Thus, individual researchers can work on independent (and possibly incompatible) features without interfering

*This work was supported by the National Science Foundation under Grant DCR75-01307.

+Present address: Department of Computer Science, Yale University, New Haven, Connecticut 06250.

with each other. This linguistic modularity has been used to develop modules for string analysis and synthesis [8,9], data-structure processing [10], programmer-defined process associations [11], and facilities for accessing the abstract machine that underlies the implementation of SL5 [12].

The remainder of this document describes the main features of SL5. It is assumed that the reader has some familiarity with SNOBOL4. A more complete description of SL5 is given in References 13-16.

2. Syntax

Unlike SNOBOL4, the syntax of SL5 is expression-oriented, similar to that of BLISS or Algol 68. An SL5 program is a single expression; a sequence of expressions can be grouped together using the begin ... end or { ... } constructs. SL5 possesses most of the "modern" control structures described below, each of which is an expression and returns a result. Reserved words are used to specify control expressions.

SL5 supports unary and binary operators. There are two types of unary operators: prefix and suffix. For example, the expression *j*+ causes the value of the variable *j* to be incremented by 1 with the result returned as the value of the expression. All operators in SL5 return a value although in many cases, such as with the binary comparison operators, that value is the null string.

Built-in operators include the usual numeric binary and prefix operators, binary comparators, and string operators such as concatenation (||) and comparison (==, ~==). Assignment (:=) is a binary operator, which returns its right argument as its result. There are a variety of built-in procedures. An example is *section*(*s*,*n*,*k*), which returns the section of *s* that is *k* characters long and begins at character *n*. *section* is similar to the PL/I function SUBSTR, but differs in that the offset is zero-based. Examples of other built-in procedures are given in Section 5.

3. Control Expressions and Signaling

An expression returns a signal (typically "success" or "failure") as well as a value. The combination of a value and a signal is called the result of the expression. Control structures are driven by signals rather than by boolean values. For an example, consider the control expression

if *e1* then *e2* else *e3*

The expression *e1* is evaluated first. If the resulting signal is success, *e2* is evaluated. Otherwise, *e3* is evaluated.

For example, the expression

if *x* > *y* then *x* := *x* - *y* else *y* := *y* - *x*

first evaluates the expression *x* > *y*. If that expression succeeds, *x* := *x* - *y* is evaluated; otherwise *y* := *y* - *x* is evaluated.

The if-then-else construction is itself an expression and its result (value and signal) is the result of *e2* or *e3* (whichever is evaluated).

Note that although the signaling mechanism is similar to that of SNOBOL4, a value is returned even if the signal is failure. Signals are not limited to success and failure; see References 13 and 15 for details.

Other typical control expressions are:

```
while e1 do e2
until e1 do e2
repeat e
for v from e1 to e2 by e3 while e4 do e5
e1 or e2
e1 and e2
```

The while expression repeatedly evaluates *e2* as long as *e1* succeeds, and returns the last result of *e2* when *e1* fails. The until expression behaves in a similar manner except *e2* is repeatedly evaluated as long as *e1* fails. The repeat expression evaluates *e* repeatedly until *e* fails. The result of the repeat expression is the last (failing) result of *e*. The for expression behaves in the conventional manner. The phrases from *e1*, to *e2*, by *e3*, while *e4* are optional. The result of the for expression is the last result returned by *e5*. The or expression evaluates *e1* and returns its result if it succeeds. Otherwise *e2* is evaluated and its result is returned. The and expression evaluates *e1* and returns its result if it fails. Otherwise *e2* is evaluated and its result is returned.

There are other, complementary control expressions provided for convenience. See Reference 13 for a complete list.

4. Procedures

One of the novel features of SL5 is that procedures and their environments ("activation records") are source-language data objects. Thus a procedure is "constructed" by assigning it to a variable, e.g., the expression

```
gcd := procedure (x, y)
  while x ~= y do
    if x > y then x := x - y else y := y - x;
  succeed x
end
```

assigns to *gcd* a procedure that computes the greatest common divisor of its arguments.

SL5 procedures can be used as recursive functions or as coroutines. The invocation of a procedure in the standard recursive fashion is accomplished by using the usual functional notation -- *f(e1,e2,...,en)*. This expression results in the invocation of the procedure that is the current value of the variable *f*, not a procedure named *f*. The actual arguments are given by the expressions *e1* through *en*.

Procedure activation may be decomposed into several distinct source-language operations that allow SL5 procedures to be used as coroutines. These operations are the creation of an environment for the execution of the given procedure, the binding of the actual arguments to that environment, and the resumption of the execution of the procedure.

The first component, environment creation, is accomplished by the create operator:

```
e := create f
```

The create operator takes a single argument of datatype procedure and creates an environment for its execution. This environment is a source-language data object, and in the above expression is assigned to *e*.

The with operator is used to bind the actual arguments to an environment. The expression

```
e with (e1, e2, ...,en)
```

binds the actual arguments, *e1* through *en*, to the environment that is the value of *e*.

The execution of a procedure is effected by "resuming" it using the resume operator. The expression

```
resume e
```

causes the execution of the current procedure to be suspended and the execution of the procedure for which the value of *e* is an environment to begin or continue.

A procedure usually "returns" a result to the "calling", or the "resuming", procedure. This is accomplished by the expressions

succeed *e*
fail *e*

which return *e* as the value of the procedure and signal either success or failure as indicated. If *e* is omitted, the null string is assumed.

If the procedure is activated by a resume expression, the result given in the succeed or fail expression is transmitted and becomes the result of the resume expression. The execution of succeed or fail causes the suspension of that particular instance of the procedure. If the environment representing that execution is again resumed, execution proceeds from where it left off. It is possible to resume a procedure with a result (value and signal), which becomes the result of the succeed or fail operator.

4.1 Argument Transmitters

A transmitter is associated with each formal argument for a procedure. Transmitters are specified in the procedure heading, e.g.

gcd := procedure (*x*:*val*, *y*:*val*)

specifies that the transmitter associated with the arguments *x* and *y* is the value of the identifier *val*, which corresponds to transmission by value. If the transmitter is omitted, transmission by value is assumed.

A transmitter may be a built-in procedure, a programmer-defined procedure, or an environment. The built-in transmitters are *val*, which transmits arguments by value, *ref*, which transmits arguments by reference, and *exp*, which transmits arguments by expression (similar to transmission by name in Algol 60).

When argument binding occurs, either by explicit use of with or by procedure invocation using functional notation, each actual argument expression is passed to the transmitter associated with the corresponding formal argument. The value returned by the transmitter is used as the value of the formal argument. If any of the transmitters signal failure, the binding operation fails.

Programmer-defined transmitters can be used for datatype checking, tracing, or common preprocessing of arguments. For

example, the expression

```
positiveint := procedure (n)
  if (n := integer(n)) > 0
    then succeed n
    else fail
end
```

assigns to *positiveint* a procedure that succeeds and returns *n* if *n* is a positive integer, and fails otherwise. The procedure *integer(n)* is a built-in procedure that converts *n* to an integer and returns it as its result, or fails if *n* is not convertible to an integer. The procedure *positiveint* can be used as a transmitter for *x* and *y* in *gcd*, for example, by including it in the procedure heading:

```
gcd := procedure (x:positiveint, y:positiveint)
```

This mechanism can also be used to replace similar operations at the beginning of several procedures that put the arguments in a canonical form by a single transmitter.

4.2 Declarations

Unlike SNOBOL4, SL5 has declarations for identifiers. These declarations, however, are used to determine only the interpretation and scope of identifiers that appear in a procedure; not their type. The declarations are motivated by the decomposition of procedure activation described above, and are designed to provide the inter-procedure communication that is suitable and convenient for coroutine programming.

Identifiers may be declared either public or private. These declarations have the form

```
public id1, id2, ..., idn
private id1, id2, ..., idn
```

The value of a private identifier is available only to the procedure in which it is declared; it cannot be examined or modified by any other procedure. Private identifiers are used for data that is local to a particular instance of a procedure. For example, when a procedure is used in a coroutine fashion, private identifiers can be used to "remember" information necessary to reverse effects during backtracking.

Public identifiers are used as the principal means of dynamic inter-procedure communication. The value of a public identifier is accessible to the procedure in which it is declared and to any other procedure whose environment is a descendant of the environment for the procedure containing the public declaration. This is equivalent to the dynamic scope of identifiers in SNOBOL4 when procedures are invoked in the usual recursive fashion.

If an identifier does not appear in any of the declarations for a procedure, it is said to be free. The interpretation of a free identifier is dynamic and occurs when an environment for the procedure containing the free identifier is created (i.e., by the create operation). The interpretation is obtained by searching for an environment for a procedure that contains a public declaration for the identifier. This search is guided by creation history. Each environment contains information that indicates its creator, which is the environment that caused its creation via the create operator. The search for the interpretation of a free identifier is performed by examining each creator until one is found whose procedure contains a public declaration for the identifier. That environment is called the custodian of the identifier. After the custodian is located, the free identifier, in the environment that is being created, henceforth refers to the public identifier located in the custodian.

If the search fails to find an interpretation, an "implicit" public interpretation is provided for the identifier. The environment whose creation resulted in the unsuccessful search becomes a custodian of the identifier just as if the identifier had been declared public.

5. Programming Examples

The following procedure illustrates a simple example of coroutine usage.

```
nextc := procedure(s) private n, c;
  repeat {
    for n from 0 while c := section(s,n,1) do
      succeed c;
    fail;
    }
end
```

An environment for *nextc* returns the next character from its argument *s* each time it is resumed. If the environment is resumed after the last character of *s* has been returned, failure is signalled to indicate the end of the string. Subsequent resumption, however, begins the process anew, returning each character in the argument until the string is once again exhausted.

For example, the expression

```
nch := create nextc with "SNOBOL4"
```

assigns to *nch* an environment for *nextc* with the argument "SNOBOL4". The value of the expression

```
resume nch
```

is the next character in "SNOBOL4". The eighth resumption of *nch* signals failure, the ninth resumption returns "S", and so on.

The following procedure, *find*, returns the next line in the file *f* that contains the string given in *str*. It fails at end of file.

```
find := procedure (str:string, f:file) private line, match;
  while line := readline(f) do {
    match := 0;
    for i from 0 to length(line) - length(str)
      while match = 0 do
        if substring(str,line,i) then
          match+;
    if match > 0 then
      succeed line;
    };
  fail
end
```

This procedure illustrates the use of the built-in procedures *string* and *file* as argument transmitters that perform datatype coercion. Both procedures attempt to convert their argument to the indicated type and return it as their result. They fail if the conversion is not possible.

The procedure *readline*(*f*) returns the next line from file *f*, failing at end of file. The procedure *length*(*s*) returns the number of characters in its argument, and *substring*(*str*,*line*,*i*) succeeds if *str* is a substring of *line* beginning at character position *i* and fails otherwise.

For example, the sequence

```
getnext := create find with ("programmer", "manual");
repeat writeline(outfile, resume getnext);
```

prints all lines in the file "manual" that contain the word "programmer" on the file given by the value of *outfile*. The built-in procedure *writeline*(*f*,*s*) writes string *s* to the file *f*.

The next example computes all the permutations of the integers 1 to *n* using Knuth's Method 1 [17]. The procedure assigned to the variable *permutations* returns the next permutation of the integers from 1 to *n* each time it is resumed. This is done by creating an environment for itself to compute the permutations for the integers 1 to *n*-1, and inserting *n* in all possible positions to form *n* permutations of the integers 1 to *n*.

```
public wid, permutations;

permutations := procedure (n) private s, p, i;
  if n = 1 then
    succeed lpad("1", wid)
  else {
    p := create permutations with (n - 1);
```

```
        while s := resume p do
          for i from 0 to wid*(n-1) by wid do
            succeed section(s,0,i) ||| lpad(n,wid) ||
              section(s,i,length(s)-i);
        };
      fail
    end;

    wid := length(n) + 1;
    p := create permutations with n;
    repeat writeline(outfile, resume p);
```

The public identifier *wid* is used to format the integers in a permutation into columns whose width is determined by the length of *n*. *lpad(s,n)* is a built-in procedure that returns a string consisting of *s* padded on the left with blanks to form an n-character string. For *n* = 3, the output of this program is

```
3 2 1
2 3 1
2 1 3
3 1 2
1 3 2
1 2 3
```

6. Conclusions

SL5 is written in Sil/2 [18], a language designed for the implementation of transportable interpreters. SL5 is currently running on the DECsystem-10.

Preliminary results in the use of SL5 have been quite encouraging; it is presently being used in several projects related to high-level data structure processing, string analysis and synthesis, and goal-oriented programming [19]. As a result of knowledge gained in these applications, SL5 has undergone continual revision. The language design and its implementation are expected to reach a state of relative stability in the near future.

Acknowledgements

SL5 is the result of the work of several persons and synthesizes results obtained from related research. In addition to the work sited in the following references, significant contributions have been made by Dianne E. Britton and Frederick C. Druseikis.

References

1. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. The SNOBOL4 Programming Language, 2nd ed. Prentice-Hall, Englewood Cliffs, N.J. 1971.

2. Hanson, David R. "Variable Associations in SNOBOL4", Software -- Practice and Experience, Vol. 6, No. 2, 245-254 (1976).

3. Griswold, Ralph E. "A Portable Diagnostic Facility for SNOBOL4", Software -- Practice and Experience, Vol. 5, No. 1, 93-104 (1975).

4. Druseikis, Frederick C. and John N. Doyle. "A Procedural Approach to Pattern Matching in SNOBOL4", Proceedings of the ACM Annual Conference, November, 1974, pp. 311-317.

5. Griswold, Ralph E. "Extensible Pattern Matching in SNOBOL4", Proceedings of the ACM Annual Conference, October, 1975, pp. 248-252.

6. Doyle, John N. A Generalized Facility for the Analysis and Synthesis of Strings and a Procedure-Based Model of an Implementation. SNOBOL4 Project Document S4D48, The University of Arizona, Tucson, Arizona. February 11, 1975.

7. Hallyburton, John C. Jr. Advanced Data Structure Manipulation Facilities for the SNOBOL4 Programming Language. SNOBOL4 Project Document S4D42, The University of Arizona, Tucson, Arizona. May 24, 1974.

8. Griswold, Ralph E. String Scanning in SL5. SL5 Project Document S5LD5, The University of Arizona, Tucson, Arizona. December 3, 1975.

9. Griswold, Ralph E. "String Analysis and Synthesis in SL5", Proceedings of the ACM Annual Conference, October, 1976, to appear.

10. Hanson, David R. A Data Structure Facility for SL5. SL5 Project Document S5LD6, The University of Arizona, Tucson, Arizona. April 20, 1976.

11. Hanson, David R. Associated Processes in SL5. SL5 Project Document S5LD7, The University of Arizona, Tucson, Arizona. June 3, 1976.

12. Griswold, Ralph E. The Window to Hell in SL5. SL5 Project Document S5LD8a, The University of Arizona, Tucson, Arizona. June 1, 1976.

13. Hanson, David R. The Syntax and Semantics of SL5. SL5 Project Document S5LD2a, The University of Arizona, Tucson, Arizona. April 28, 1976.

14. Griswold, Ralph E. A Catalog of Built-In SL5 Operators and Functions. SL5 Project Document S5LD3a, The University of Arizona, Tucson, Arizona. May 11, 1976.

15. Hanson, David R. and Ralph E. Griswold. The SL5 Procedure Mechanism. SL5 Project Document S5LD4, The University of Arizona, Tucson, Arizona. February 19, 1976.

16. Britton, Dianne E., et al. "Procedure Referencing Environments in SL5", Third ACM Symposium on Principles of Programming Languages, January, 1976, pp. 185-191.

17. Knuth, Donald E. The Art of Computer Programming, Volume 1, Fundamental Algorithms, 2nd ed. Addison-Wesley, Reading, Mass., 1973, pp. 44-45.

18. Druseikis, Frederick C. The Design of Transportable Interpreters. SNOBOL4 Project Document S4D49, The University of Arizona, Tucson, Arizona. February 27, 1975.

19. Hanson, David R. "A Procedure Mechanism for Backtrack Programming", Proceedings of the ACM Annual Conference, October, 1976, to appear.