Procedure Referencing Environments in SL5*

Dianne E. Britton, Frederick C. Druseikis, Ralph E. Griswold
David R. Hanson, and Richard A. Holmes

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

SL5 is a programming language developed for experimental work in generalized pattern matching and high-level data structuring and access mechanisms. This paper describes the procedure mechanism and the conventions for the interpretation of identifiers in SL5. Procedure invocation in SL5 is decomposed into the separate source-language operations of context creation, argument binding and procedure activation, and allows SL5 procedures to be used as recursive functions or coroutines. This decomposition has led to rules for scoping and for the interpretation of identifiers that are different from those found in other programming languages. Several examples of SL5 procedures are given, including a scanner based on the coroutine model of pattern matching.

1. Introduction

The SNOBOL4 programming language has been used as a basic experimental tool for recent research in generalized string pattern matching [2,4] and advanced data structure manipulation [6]. Until recently, the approach has been to extend or modify existing SNOBOL4 implementations in order to conduct investigations in these areas. The formulation of the coroutine model for pattern matching [2,3], and the introduction of programmer-defined scanning procedures based on that model [4], have made this approach impracticable. The procedure mechanism required by the coroutine model necessitates a more versitile research vehicle than is provided by SNOBOL4 and other existing programming languages. Motivated primarily by the requirements of the coroutine model, the SL5 programming language [5] has been developed and implemented as a research tool for experimental work in the areas mentioned above.

The purpose of this paper is to describe the procedure mechanism of SL5 and the conventions that have been chosen for the interpretation of identifiers in SL5. SL5 provides a coroutine mechanism that is based on a generalization of recursive procedures in which ordinary recursive function use is a special case. The conventions for the dynamic interpretation of identifiers have been designed to facilitate the insertion of programmer-defined procedural abstractions into prior, or builtin, procedural abstractions. For example, the conventions allow the extension of the pattern-matching scanner by the inclusion of programmer-defined scanning procedures. The dynamic interpretation of identifiers used in SL5 is conceptually simpler than that used for programming languages such as Algol 60, whose scoping rules are too restrictive for general coroutine programming. For example, procedures as arguments require special handling in Algol 60. In addition, the conventions provide for the necessary inter-procedure communication that is convenient and suitable for applications requiring backtracking such as programmer-defined scanning procedures [4].

2. The SL5 Programming Language

Experience with SNOBOL4 has motivated some of the features of SL5, e.g., SL5 is a "typeless" language in the same sense that SNOBOL4 is -- a variable can have a value of any datatype at any time during program execution. A signaling mechanism is also included in SL5. Some parts of the language, however, are derived from the desire to include facilities that are not commonly found in programming languages and for which SNOBOL4 is inadequate. For example, the syntax of SL5 is expression-oriented, and the control structures are similar to those in Algol 68. An expression returns a value and signals either "success" or "failure" (as in SNOBOL4). Control structures are

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

driven by these signals rather than by boolean values. 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.

Loop constructs are controlled in the same manner:

```
while e1 do e2
```

repeatedly evaluates e_2 as long as e_1 succeeds. A sequence of expressions can be grouped together as a single expression using the begin ... end construct. Other typical constructs are included, and have interpretations consistent with the signaling mechanism.

Procedures in SL5 are data objects and an identifier can be assigned a procedure as its value. For example, the expression

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

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

2.1 The Decomposition of Procedure Activation

In most programming languages, the invocation of a procedure is considered an atomic operation. In SL5, procedure invocation is decomposed into several distinct components available to the programmer at the source-language level. The invocation and execution of a procedure requires the *creation* of a context, the *binding* of the actual arguments to that context, and the *resumption* of the procedure represented by the context. In SL5, the conventional function notation

$$f(e_1,e_2,\ldots,e_n)$$

for the invocation of a procedure is decomposed into the steps

```
t1 := create f;
t2 := t1 with (e1,e2,...,en);
resume t2;
```

The operator create takes a single argument of datatype procedure and returns an object of datatype context. This source-language data object is a context for the execution of the given procedure. Argument binding is accomplished using the with operator that binds the actual arguments, e_1 through e_n, to the indicated context.

The actual activation of a procedure is accomplished by the resume operator, which takes a single argument of datatype context. resume causes the execution of the current procedure to be suspended and the execution of the procedure represented by the given context to commence or continue.

The operations

```
succeed e
fail e
```

result in the suspension of the execution of the current procedure and resumption of the instance of the "resuming" procedure. These operations are equivalent to return in the recursive case. The indicated signal (success or failure) and value are transmitted and become the signal and value of the resume operator.

This decomposition provides the mechanism for SL5 procedures to be used as recursive functions or coroutines. While this decomposition is useful and necessary in some coroutine programming situations, the abbreviated notation $f(e_1,e_2,\ldots,e_n)$ may be used for the usual recursive invocation.

3. The Interpretation of Identifiers

The interpretation and scope of identifiers that appear in a procedure are determined by declarations. In many cases, a procedure invoked in the standard recursive fashion needs some mechanism for inter-procedure communication. In contrast, a coroutine often requires that some data be inaccessible from any other coroutine. The declarations in SL5 are motivated by the need for dynamic communication between instances of procedures and the need for identifiers whose values cannot be modified by any other procedure.

3.1 Public and Private Identifiers

Declarations have the form

```
public v1, v2, ..., vn
private v1, v2, ..., vn
```

Public identifiers allow for dynamic communication between instances of procedures. A public identifier is accessible to the procedure in which it is declared and in any other procedure whose context is within the dynamic scope [8] of the context for the procedure containing the public declaration. When SL5 procedures are used in the standard recursive fashion, the interpretation of public

identifiers is equivalent to the dynamic scope of identifiers in SNOBOL4.

Private identifiers are available only to the procedure in which they are declared. The value of a private identifier cannot be examined or modified by any other procedure. This type of identifier is used, for example, in situations where a procedure must "remember" information in order to be able to reverse effects during backtracking. Unless otherwise declared, the formal parameters of a procedure are considered to be *private* identifiers.

3.2 Free Identifiers

Free identifiers are those that do not appear in any of the declarations in the procedure in which they are used. The interpretation of this type of identifier is *dynamic* and occurs at the time of the creation of a context for the procedure that contains the free identifiers. The interpretation is obtained by examining the current state of the *creation history tree*.

The creation history tree provides information concerning the history of the creation of contexts during the course of program execution. The tree grows whenever a create operation is performed and is pruned upon the destruction of a context. The actual tree consists of interconnections among contexts that result from the create operation.

An example of a creation history tree is shown in Figure 1. The nodes of the tree in Figure 1 represent instances of contexts for particular procedures. An arrow indicates the ancestral linkage between a context and the context that caused its creation. The subscripts on B denote specific occurrences of a context for the same procedure. Contexts with a common ancestor but that do not share a common ancestral linkage are referred to as *parallel* contexts. *Serial* contexts are those in which creation history follows ancestral lines. B_1 and B_2 are serial contexts; B_1 and B_3 are parallel contexts.

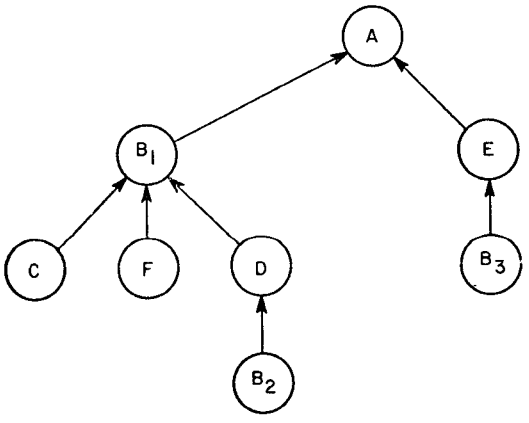

Figure 1. A Creation History Tree

The context for the procedure in which public identifiers are declared is said to be the *custodian* of those identifiers. As described above, public identifiers are available to their custodian and to any procedure whose context is a descendant of their custodian in the creation history tree.

The dynamic interpretation of free identifiers is determined by the history of serial contexts. A search is performed along the ancestral linkage for the first context that contains a public declaration for the free identifier, i.e., for the closest custodian of that free identifier. If the search is successful, the free identifier (in the particular instance of the context) henceforth refers to the public identifier located in the custodian.

In Figure 1, assume that the procedure A (that is, the procedure represented by A) contains the declaration

public *x*

and that procedures B and E contain *x* as a free identifier. At the time of creation of the contexts B_1 and E, the interpretation given the free identifier *x* is the common public identifier *x* in the context A. In this case, the context A is the custodian of the public identifier *x*. During the execution of B_1 and E, *x* can be used for inter-procedure communication between A, E, and B_1.

As another example, consider an identifier *y* declared public in both D and E but that is free in B. In this case, the interpretation given to *y* in B_2 refers to *y* declared public in D, whereas in B_3 the free identifier *y* refers to the public identifier declared in E.

Communication between the serial contexts B_1 and B_2, which are invoked in a recursive fashion, can be effected by reference to a free identifier (such as *x*) in B that is declared public in A.

This interpretation of free identifiers allows procedures with instances of either serial or parallel contexts to communicate via public identifiers declared in procedures that are represented by their common ancestors in the creation history tree. Since private identifiers are not considered during the interpretation of free identifiers, their values are inaccessible from any other procedure activation.

The root of the creation history tree is a context for a "main procedure" in which, conceptually, all builtin identifiers are declared public, and are initialized to their predefined values before program execution begins. The root context contains, for example, identifiers that have builtin procedures as their initial value. Unless explicitly declared otherwise, when these identifiers are used as free identifiers in programmer-defined procedures, the interpretation made is the value of the builtin identifier in the root context.

The search of the creation history tree may fail to provide an interpretation for a free identifier. There are several possible solutions to this problem. Perhaps the most reasonable solution is to consider this situation a programming error. An alternative is to provide an "implicit" public or private interpretation for the identifier. If an implicit public interpretation is chosen, either the active context or the root of the creation history tree could be chosen to be the custodian of the identifier.

4. Examples

4.1 Redefinition of a Procedure

Since a procedure is a source-language data object, public identifiers provide a convenient means for the dynamic redefinition of a programmer-defined or builtin procedure within a subtree of the creation history tree. For example, assume the procedure represented by B_1, B_2, and B_3 in Figure 1 contains the expression

```
j := length(s)
```

and *length* is a free identifier in B. At the time of the creation of the contexts B_1 and B_3 an interpretation for *length* is sought. Assuming that the procedure represented by E does not contain a public declaration for *length*, the interpretation made is the builtin identifier *length* (whose value is a procedure that returns the number of characters in its argument). If the identifier is declared public in the procedure represented by D, and assigned a procedure as value, the effect is to cause the redefinition of *length* in the instance of procedure B represented by B_2. The programmer can use this facility to monitor the use of certain procedures or extend the domain of a procedure or operator to accomodate different types of arguments.

4.2 A Generator of Random Number Generators

The values of private identifiers in a context for a procedure partly characterize the state of that instance of the procedure. As such, they provide a mechanism for the parameterization of a given instance of a procedure. For example, consider the procedure *rangen* defined as follows.

```
rangen := procedure(private s,p,c,m,n)
 repeat begin
  s := remdr((s * p) + c,m);
  succeed ((s * n) / m) + 1;
 end
end;
```

The procedure *rangen* computes a random number using the linear congruence method [7]. A context for *rangen* computes the next random number within the range 1 to n in the sequence defined by the parameters s, p, c, and m. A context for *rangen* is parameterized by the values of the arguments to *rangen* and generates the next random number from an independent sequence every time it is resumed. Thus any number of generators may be created using a common procedure, *rangen*. It is the context for each instance of *rangen* that is the generator. For example, the expressions

```
g1 := create rangen
      with (0,12621,21131,100000,100);

g2 := create rangen
      with (0,12641,11241,10000,10);

g3 := create rangen
      with (111,12321,12231,10000,50);
```

assign to $g1$, $g2$, and $g3$ three separate contexts for *rangen*, each of which generates a distinct sequence of random numbers. To obtain the next random number in a sequence, the execution of the desired context is resumed, e.g.,

```
x := resume g2;
```

The superfluous data structures that are usually needed to effect the type of parameterization desired for *rangen* are unnecessary in SL5. Private identifiers allow the parameterization data to be implicitly stored as a part of the context for the procedure. Moreover, since a context is a source-language data object, a procedure activation and its associated parameterization data can be manipulated as a single object.

4.3 The Pattern-Matching Scanner

Part of the motivation for the procedure facility in SL5 comes from the coroutine model of pattern matching [2,3]. The conventions adopted for the interpretation of identifiers facilitate the interaction between the builtin scanner and scanning procedures and programmed-defined scanning procedures. In addition, the procedure mechanism is sufficiently

general to allow the scanner to be written in SL5 for experimental purposes without sacrificing any of its capabilities. An SL5 implementation of the scanner, given below, illustrates the need for the conventions as defined.

The pattern-matching scanner has two arguments -- a subject and pattern. The pattern contains the information necessary to direct the analysis of the subject string. In the coroutine model, the analysis is performed by scanning procedures, which are invoked as coroutines. It is the use of public and private identifiers in the scanning procedures that is of interest in this example. Further details of the coroutine model are given in References 2, 3, and 4.

Patterns in SL5, as in SNOBOL4, are data objects. A pattern can be visualized as a tree of nodes in which each node has three fields: *sproc*, *arg1*, and *arg2*. The *sproc* field contains the scanning procedure, which is an object of datatype procedure. The *arg1* and *arg2* fields contain arguments supplied for the scanning procedure when the pattern is constructed. A pattern is constructed by a pattern-construction procedure and the actual matching is performed by the associated scanning procedure. For example, *len(n)* constructs a single-node pattern shown in Figure 2 whose *sproc* field contains the scanning procedure *slen*, which attempts to advance the cursor position by *n* characters during matching. A scanning procedure is called with a single argument: the node in the pattern that caused its invocation.

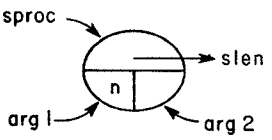

Figure 2. The Pattern Constructed by *len(n)*.

Unlike pattern matching in SNOBOL4, operations such as alternation and concatenation are themselves scanning procedures that perform only control operations. The pattern-constructing procedures for these operations in SL5 are denoted by the operators "\" and "--" respectively; *salt* and *scat* are the associated scanning procedures. For example, Figure 3 illustrates the pattern constructed by the expression

```
p := (len(1) -- rpos(0)) \ len(3)
```

Pattern matching is initiated by the creation and resumption of the scanning procedure given in the root node of the pattern passed to the scanner. Matching proceeds as scanning procedures in the pattern are activated. A successful match is signaled by the resumption of the scanner with a success signal. Thus the scanner can be written in SL5 as follows.

```
scan := procedure(public subject,
                  private pattern)
 public cursor;
 private e;
 cursor := 0;
 e := create sproc(pattern) with pattern;
 if resume e then succeed else fail;
end;
```

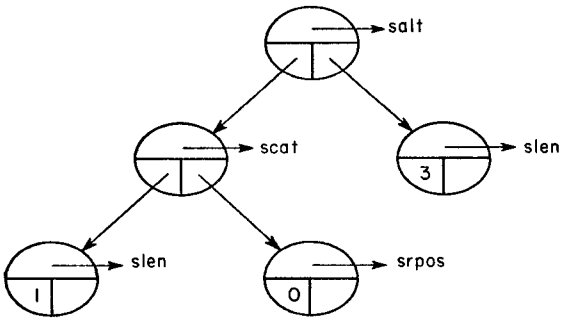

Figure 3. A Pattern.

Communication between scanning procedures and the scanner is provided by the identifiers *subject* and *cursor*, which are declared public in *scan*. The value of *subject* is the string that is to be scanned and the value of *cursor* indicates the position in the string that is to be examined by a scanning procedure.

The interpretations of the free identifiers *subject* and *cursor* that appear in scanning procedures are made when a context for a scanning procedure is created and refers to the public identifiers in the nearest custodian in the creation history tree. The use of *subject* and *cursor* in scanning procedures is illustrated by *slen*.

```
slen := procedure(private p)
 private c;
 c := cursor;
 cursor := cursor + arg1(p);
 if cursor >= 0 & cursor <= length(subject)
  then succeed;
 cursor := c;
 fail
end;
```

Note that *slen* is more general than the corresponding SNOBOL4 primitive since its argument may specify a negative integer thereby causing the cursor to be decremented.

By convention [2-4], a scanning procedure is resumed with a success signal as a request to search for alternatives. It fails if it cannot find an alternative.

In addition, before a scanning procedure can fail it must reverse any effects, such as cursor movement, that it caused during matching. Private identifiers are used within scanning procedures to retain values that are necessary in order to reverse effects during backtracking. In the procedure for *slen* given above, the private identifier *e* is used to save the previous value of *cursor*. *slen* does not possess alternatives, so if it is resumed after signaling success, it restores *cursor* to its previous value and fails. (Subsequent resumption of a scanning procedure that has signaled failure is a programming error.)

Scanning procedures that perform control operations, such as *scat* and *salt*, do not examine or modify *subject* or *cursor*. They initiate other scanning procedures in order to implement their specific control relationships. For example, the scanning procedure for a primitive version of *salt* can be written as follows.

```
salt := procedure(private p)
 private e;
 e := create sproc(arg1(p)) with arg1(p);
 while resume e do succeed;
 e := create sproc(arg2(p)) with arg2(p);
 while resume e do succeed;
 fail
end;
```

(This primitive implementation of *salt* does not provide for reversal of effects caused by arguments that contain alternatives. A complete version of *salt* is given in Reference 2.)

The procedure *salt* creates a context for its first argument and resumes its execution. As long as it succeeds, *salt* succeeds. If the first argument fails, indicating that no further alternatives exist, *salt* creates a context for its second argument and resumes its execution, succeeding as long as it does. Finally, *salt* signals failure only after all of the alternatives for both arguments have been exhausted.

A scanning procedure may invoke the scanner. Since *subject* and *cursor* are declared in *scan*, each instance of *scan* becomes a custodian for new occurrences of these identifiers. The free identifiers *subject* and *cursor* in scanning procedures that are invoked in the course of pattern matching controlled by the nested instance of *scan* refer to those identifiers in that custodian. This is illustrated in Figure 4, which shows a creation history tree containing two instances of *scan* (indicated by S_1 and S_2) and six instances of scanning procedures (C1 through C6). C1 causes a nested invocation of *scan*. Thus S_1 is the custodian of the *subject* and *cursor* referenced in C1, C2, and C6 while S_2 is the custodian of those identifiers in C3, C4, and C5.

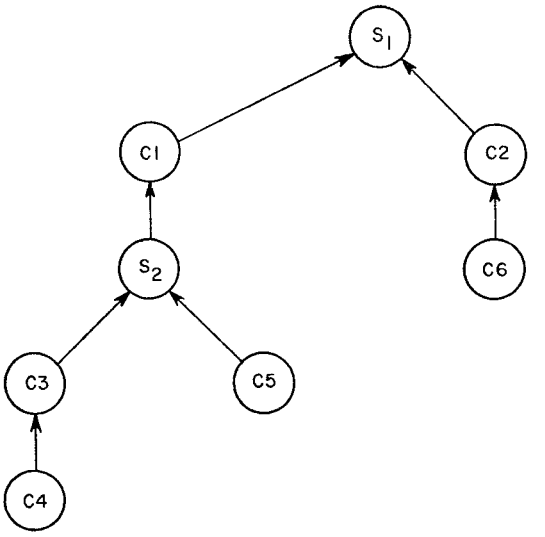

Figure 4. A Creation History Tree for the Scanner.

The builtin scanner is implemented in the same fashion as *scan*, viz., *subject* and *cursor* are treated as public identifiers in each instance of the builtin scanner. Thus the concept of public identifiers and the interpretation chosen for free identifiers are what permit programmer-defined scanning procedures to "adapt" themselves to the builtin mechanism. They also make possible the replacement of the builtin scanner by a programmer-defined procedure, such as *scan*, which is able to interact with either builtin or programmer-defined scanning procedures.

5. Conclusions

The determination of the scope of identifiers is a persistent problem in programming language design. The static scoping conventions of most high-level languages, such as Algol 60, are due in part to the atomic nature of traditional procedure activation. It is clear that in order to support the decomposition of procedure activation described in this paper, the static mechanisms are inappropriate, and additional language features are required in order to perform functions such as backtracking [1,8].

The decomposition of procedure activation into separate components gives the programmer the control, at the linguistic level, that is necessary for applications that involve sophisticated processes. This decomposition and the implications of program manipulation of procedures and

contexts as data objects have motivated the scoping conventions and dynamic interpretation of free identifiers used in SL5. The effect is a means for inter-procedure communication as well as a method of avoiding inter-procedure interference.

Initial experience indicates that the SL5 procedure mechanism does provide a research tool of sufficient generality for experimentation in the areas mentioned at the beginning of this paper. The facilities allow significant language extensions to be made without changing the base implementation of SL5. For example, the scanner can be written completely in SL5 without loss of generality. As illustrated by *scan*, the ability to create and manipulate procedure contexts independently of argument binding or invocation allows the programmer to adapt the general mechanism to a specific experimental application.

References

1. Bobrow, D. G. and Wegbreit, B. "A Model and Stack Implementation of Multiple Environments", *Communications of the ACM*, Vol. 6, No. 10, 591-603 (October 1973).

2. Doyle, J. 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, February 1975.

3. Druseikis, F. C. and Doyle, J. N. "A Procedural Approach to Pattern Matching in SNOBOL4", *Proceedings of the ACM Annual Conference*, 311-317 (November 1974).

4. Griswold, R. E. "Extensible Pattern Matching in SNOBOL4", *Proceedings of the ACM Annual Conference*, 248-252 (October 1975).

5. Griswold, R. E. and Hanson, D. R. *An Overview of the SL5 Programming Language*, SL5 Project Document S5LD1, The University of Arizona, Tucson, 1975.

6. Hallyburton, J. C., Jr. *Advanced Data Structure Manipulation Facilities for the SNOBOL4 Programming Language*, Ph.D. Dissertation, The University of Arizona, May 1974.

7. Knuth, D. E. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 1969, p. 9.

8. Prenner, C., Spitzen, J. and Wegbreit, B. "An Implementation of Backtracking for Programming Languages", *Proceedings of the ACM Annual Conference*, 763-771 (August 1972).