Language Facilities for Programmable Backtracking*

Ralph E. Griswold

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

David R. Hanson†

Department of Computer Science, Yale University,
New Haven, Connecticut 06520

Most languages intended for artificial intelligence applications include a search and backtracking facility. While built-in backtracking facilities are convenient, they are often too inflexible for use beyond a limited range of applicability. Other mechanisms, such as those based on explicit manipulations of bindings or stack frames, tend to be very unstructured, and give the programmer little control over the backtracking process. This paper describes the SL5 programming language and its use for "programmable backtracking." SL5 includes a general procedure mechanism that permits procedures to be used as recursive functions or as coroutines. Using this mechanism, the programmer can construct control hierarchies that are tailored to the specific application. A string pattern-matching facility is given as an example of the use of the SL5 facilities.

1. Introduction

Search and backtracking facilities are a traditional component of many programming languages designed for artificial intelligence applications [1]. Such facilities are used, for example, in pattern-directed search of data bases in theorem proving systems.

Some languages, such as PLANNER [2,3], include a built-in mechanism for *automatic* backtracking. While automatic backtracking facilities are adequate for some problems, it has been noted that they just as often "get in the way" [4]. CONNIVER [4] was derived from PLANNER and contains no automatic backtracking mechanisms, but provides a means for the programmer to construct a backtracking control regime. This is accomplished using global variables and *gotos*, and by permitting explicit manipulations of bindings. One problem with CONNIVER is that backtracking is achieved in a rather unstructured manner [5]. In short, while CONNIVER does not "get in the way" in cases where PLANNER does, CONNIVER provides insufficient control of the backtracking process and its associated data. There appears to be a need for linguistic mechanisms that facilitate "programmable backtracking," enabling the programmer to adapt a general mechanism to a specific application in a structured manner.

This paper describes the use of the SL5 programming language [6,7] for programmable backtracking. SL5 was not designed for AI *per se*, but its procedure mechanism seems particularly well-suited for such application. The SL5 procedure mechanism is quite general, permitting procedures to be used as recursive functions or as coroutines. This generality is accomplished by treating activation records for procedures as data objects and by decomposing procedure invocation into more elementary operations. The resulting mechanism facilitates a structured approach to programmable backtracking.

2. The SL5 Programming Language

SL5 is a high-level expression-oriented language designed as a research tool for studies in programming language control mechanisms and in advanced string and structure processing. SL5 derives many of its characteristics from SNOBOL4 but departs from that language in several essential areas. For example, SL5 includes most of the "modern" control structures. Like SNOBOL4, SL5 has no type declarations but it supports a variety of datatypes with runtime coercion where appropriate.

An SL5 expression returns a *result*, which is

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

†Present address: Department of Computer Science, The University of Arizona, Tucson, Arizona 85721.

composed of two parts: a value and a signal. The value component is used in the traditional fashion. The signal, which is a small nonnegative integer, is used to derive control expressions. By convention, the signal 1 means "success" and the signal 0 means "failure." An arbitrary result may be composed using the & operator; *e1* & *e2* derives its value component from the value of *e1* and its signal component from the value of *e2* (the signals of *e1* and *e2* are ignored). Most built-in functions and operators transmit the result ""&0 if the evaluation of any of their arguments produces a failure signal.

Signals are used to derive control expressions. For example, in the expression if *e1* then *e2* else *e3*, *e1* is evaluated first. If the resulting signal is 1, the result of the if expression is the result of *e2*, otherwise the result is the result of *e3*. Other typical control expressions include

```
while e1 do e2
until e1 do e2
unless e1 do e2
repeat e
{e1; e2; ...; en}
```

The while. until, and unless expressions behave in the conventional manner. The repeat expression evaluates *e* repeatedly until *e* signals failure. The braces enclose a sequence of expressions.

Procedures and their environments (activation records) are data objects that are created at runtime. For example, the expression

```
genlab := procedure (p, n, i)
   repeat {
      return p || n;     # || is concatenation
      n := n + i
      }
end
```

assigns to *genlab* a procedure whose environments can be used as label generators as described below.

The expression

```
return exp
```

returns the result of evaluating *exp* as the result of the procedure. The expressions

```
succeed v
fail v
```

are provided and are equivalent, respectively, to

```
return v&1
return v&0
```

If *v* is omitted, the null string is assumed.

In addition to the usual function notation, namely *f*(*x*), procedure invocation may be decomposed into three components: environment creation, argument binding, and procedure resumption. The create expression takes a procedure as argument and creates an environment for the execution of the procedure. For example, the expression

```
x := create genlab
```

assigns to *x* an environment for *genlab*. The with expression is used to bind the actual arguments to an environment. The expresiion

```
y := x with ("L", 10, 1)
```

binds the actual arguments to the formal arguments in the environment for *genlab* and assigns that environment to *y*. Procedure resumption is accomplished by the resume expression. For example,

```
next := resume y
```

which assigns the label "L10" to *next*. Subsequent resumptions of *y* proceed from where *y* left off (the return expression). Thus, because of the repeat loop, a resumption of *y* produces the next label.

The form

```
resume (e, r)
```

may be used to transmit the result of *r* to *e* upon resumption. The result *r* becomes the result of the expression that caused the suspension of *e*. The expression

```
fresume e
```

is equivalent to

```
resume (e, ""&0)
```

In other words, *e* is resumed with a failure signal.

The decomposition of procedure invocation and the treatment of environments as data onjects permit SL5 procedures to be used as recursive functions or as coroutines.

The scope of identifiers is determined by declarations: identifiers may be declared either public or private. Identifiers that do not appear in any procedure declarations are termed nonlocal.

Private identifiers are accessible only to the proce re in which they are declared. Public identifiers are accessible to the procedure in which they are declared and to other procedures containing nonlocal identifiers by the same names whose environments are within the dynamic scope of the environment for the procedure containing the public declaration.

Further details concerning procedure referencing environments are given in Reference 7.

3. String Pattern-Matching

A string pattern-matching facility, similar to that in SNOBOL4, illustrates the use of the SL5 mechanisms for programmable backtracking. A string pattern-matching facility, based on the coroutine model of pattern-matching [8], is implemented by writing a set of procedures that work in concert to achieve the necessary backtracking control regime. The required "cooperation" among the procedures is accomplished by observing programming conventions and communication protocols.

Pattern-matching is performed by *match*(*s*, *e*), where *s* is the subject string to be scanned and *e* is environment that controls the scanning. The separate notions of "pattern" and "matcher" in SNOBOL4 are embodied in the scanning environment *e*. Roughly speaking, scanning environments constitute both the data component (pattern) and control component (matcher) found in most pattern matching systems. Various "patterns" are constructed by forming a network of scanning environments. Some scanning environments direct other scanning environments in various control relationships, such as alternation ala SNOBOL4. Other scanning environments analyze the subject string.

Communication among scanning environments is provided by the global variables *subject*, which contains the subject string, and *cursor*, which

indicates the current position of attention in the subject.

Cooperation among scanning environments is accomplished by establishing programming conventions and a communication protocol that is suitable to this particular application.

A scanning environment *e* is resumed (resume *e*) when a match for that environment is needed.

If *e* is able to perform the match, it returns with a success signal (succeed) to indicate the successful match. On the other hand, if *e* is unable to perform the match, it returns with a failure signal (fail).

If a scanning environment successfully matches, it may be resumed again, which indicates a request for an alternate match. Some environments may be able to match more than one way, while others may have no alternative match. As on initial resumption, the success or failure signal is used on return as appropriate.

By convention, once an environment fails, it is not resumed again; to do so would be a programming error. This convention embodies a particular aspect of the kind of pattern matching implemented here, not a general property of all search strategies.

Scanning environments have the responsibility for all the "global" effects they may cause. Thus an environment that moves the cursor is responsible for this action. This responsibility requires an environment to reverse any effect (for example, restoring the cursor to its original position) that is a result of a previous match. For example, if an alternative is requested, but none exists, the environment to which the request is made must restore any global variables to the values they had before the environment was initially resumed.

In some circumstances, backtracking to an earlier situation is necessary. In this case, an alternative match by a current environment may not be desired, but this environment may nonetheless have caused effects that need to be reversed before backtracking. To assure reversal of effects, an environment is resumed with a failure signal (fresume *e*) as an indication that effects are to be reversed, as opposed to resumption with a success signal, which indicates a request for an alternative match.

Several scanning procedures are given below to illustrate programming techniques and the use of the conventions and communication protocol described above.

The actual code for *match*(*s*, *e*) is simple: The communication variables are initialized and *e* is resumed.

```
match := procedure (s, e)
   subject := s;
   cursor := 0;
   if resume copy(e) then
      succeed
   else
      fail
end
```

The built-in function *copy*(*e*) copies an environment. The copy is "reset" so that execution will begin at the beginning of the procedure. The environment is copied in order to avoid possible interference resulting from the resumption of the same environment in different places.

Scanning environments are created from scanning procedures. For example, a procedure to match literal strings is

```
slit := procedure (s) private c;
   if section(subject,cursor,length)) == s
      then {
      c := cursor
      cursor := cursor + length(s);
      succeed;
      cursor := c; # reverse the effect of slit
      fail
      }
   else
      fail
end
```

(The built-in function *section*(*s*, *i*, *j*) returns the substring of *s* starting at character *i* and of length *j*. The operator *s1* == *s2* succeeds if *s1* and *s2* are the same string.) The procedure *slit* increments the cursor by the length of *s* if the subject contains *s* at the current cursor position. If, following a successful return, the environment is again resumed, the original value of *cursor* is restored (reversing the effects of *slit*) and *slit* fails, since there are no alternatives.

An example of the use of *slit* is

```
match(s, create slit with "language")
```

which causes a search of *s* for the string "language."

Several improvements can be made in the notation. The awkwardness of constructing environments for the second argument of *match* can be avoided by coercing the arguments of *match* to be the desired datatypes. This is accomplished by transmitters, attached to the formal parameters, which process the actual arguments during binding [7]. A transmitter is specified in the procedure parameter list by appending a colon and the transmitter to the parameter. For *match*, the procedure heading is

```
match := procedure (s:string, e:environment)
```

where *string* in a built-in conversion function and *environment* is a procedure that converts strings to environments for *slit*, as follows:

```
environment := procedure (x)
   if datatype(x) == "environment" then
      succeed x
   else if datatype(x) == "string" then
      succeed create slit with x
   else
      fail
end
```

An additional cosmetic improvement can be obtained by using an operator instead of the function name *match*. SL5 permits assigning procedures to operators as well as identifiers. In the case of operators, the print name is used. An operator print name is a string that consists of the operator symbol with parentheses indicating argument placement to distinguish prefix, infix, and suffix uses of the same symbol. Thus the print name of the infix operator ? is ")?(". Values of strings are accessed by the built-in function *value*(*s*). Thus the matching procedure above can be associated with the infix ? operator by

```
value(")?(") :=
   procedure (s:string, e:environment)
      :
```

Using these two improvements, the example above then becomes

```
s ? "language"
```

Various matching procedures can be developed. A simple one is *smove*, which moves the cursor by a fixed amount.

```
smove := procedure (n) private c;
   c := cursor;
   cursor := cursor + n;
   if cursor >= 0 and
      cursor <= length(subject) then
         succeed;
   cursor := c;
   fail
end
```

If *smove* is resumed a second time, it has no alternative. Before it fails, it restores the cursor to its former value, thus reversing the effect that it caused. A procedure to construct scanning environments for *smove* is

```
move := procedure (n:integer)
   return create smove with n
end
```

Alternation (*e1* "or" *e2*) is an example of a scanning procedure that performs a control function. The procedure for alternation, *salt*, has two arguments, *e1* and *e2*. It first applies *e1*. If *e1* matches, *salt* succeeds indicating a successful match. If *e1* does not match, *e2* is applied. Note the use of fresume to reverse effects by the arguments of *salt*. Although *salt* causes no effects of its own, the environment that it resumes may cause effects.

```
salt := procedure (e1, e2)
   e1 := copy(e1);
   while resume e1 do
      unless succeed do {
         fresume e1;
         fail
         };
   e2 := copy(e2);
   while resume e2 do
      unless succeed do {
         fresume e2;
         fail
         };
   fail
end
```

A procedure for applying two environments in sequence (*e1* "then" *e2*) is

```
sseq := procedure (e1, e2)
   e1 := copy(e1);
   while resume e1 do {
      e2 := copy(e2);
      while resume e2 do
         unless succeed do {
            fresume e2;
            fresume e1;
            fail
            }
      };
   fail
end
```

Procedures to construct environments for alternation and the successive application of other scanning environments are similar to *move*. Using the infix operators | and -- for the two, respectively, the procedures are:

```
value(")|(") :=
   procedure (e1:environment, e2:environment)
   return create salt with (e1, e2)
end
value(")--(") :=
   procedure (e1:environment, e2:environment)
   return create sseq with (e1, e2)
end
```

For example, the expression

```
s ? ("DEC10" | "HP3000") -- " computer"
```

matches either "DEC10 computer" or "HP3000 computer."

"Recusive" pattern-matching specifications present an interesting problem. The expression

```
x := "a" | ("b" -- x)
```

appears to specify a recursive reference to *x*, but actually refers to the previous value of *x*, which is used in the construction of environments for the right side. The effect of recursion can be achieved by deferring evaluation of a variable until it is encountered during pattern matching. A procedure to accomplish this is

```
sdefer := procedure (v:ref) private e;
   unless e := copy(environment(v)) do fail;
   while resume e do
      unless succeed do {
         fresume e;
         fail
         }
end
```

(*ref* is a built-in transmitter that transmits by reference in the style of FORTRAN. Note the use of *environment*(*v*) to coerce the value of *v* to be an environment.) If a procedure to construct scanning environments for *sdefer* is assigned to the prefix operator *, a recursive pattern specification corresponding to the example above is

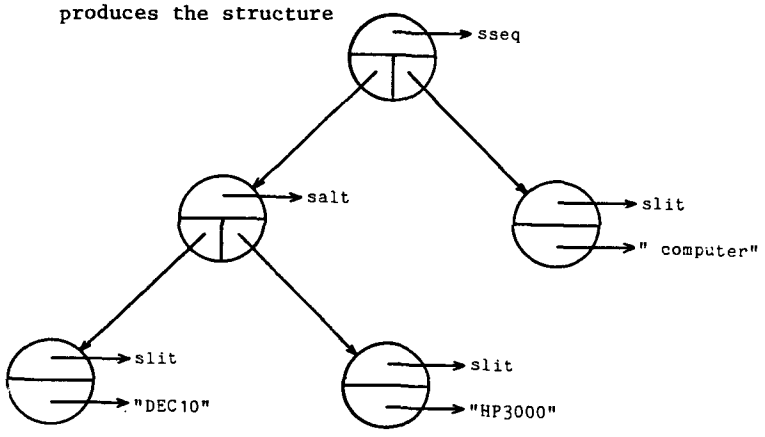
```
x := "a" | ("b" -- *x)
```

This pattern matches strings of the form a, ba, bba, and so forth.

Scanning environments can be thought of as data objects with specified procedures and arguments that are either literals or other scanning environments. The application of the alternation and succession operators produces tree structures of environments. For example, the expression

```
("DEC10" | "HP3000") -- "computers"
```

produces the structure



The ability to manipulate environments explicitly and to establish customized conventions such as those given above makes possible, and even encourages, novel programming styles. For example, consider the scanning procedure *sarbno*(*e*) that matches zero or more consecutive occurrences of whatever *e* matches. Its first alternative matches the null string. Its second matches *e*, its third matches *e* followed by *e*, and so on (similar to ARBNO in SNOBOL4).

While it is possible to write *sarbno* directly, a more interesting approach is to fabricate an appropriate environment network from existing scanning procedures. The behavior of *sarbno*(*e*) is equivalent to

```
"" | e | (e -- e) | (e -- e -- e) | ...
```

which is equivalent to the recursive specification

```
x := "" | (e -- *x)
```

This behavior can be accomplished by building a "structural loop" in which the **x* in *e* -- **x* refers to the structure

```
"" | e -- *x
```

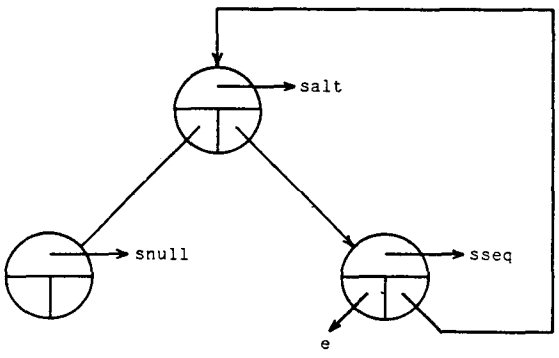
The procedure *arbno*(*e*) constructs such a network (note the use of *salt*):

```
arbno := procedure (e:environment) private x;
   x := create salt;
   return (x with (snull, copy(e) -- x))
end
```

snull is a scanning environment that matches the null string, i.e.,

```
snull := create procedure
   succeed;
   fail
end
```

Viewing this as a structure as illustrated above, the result of *arbno*(*x*) is



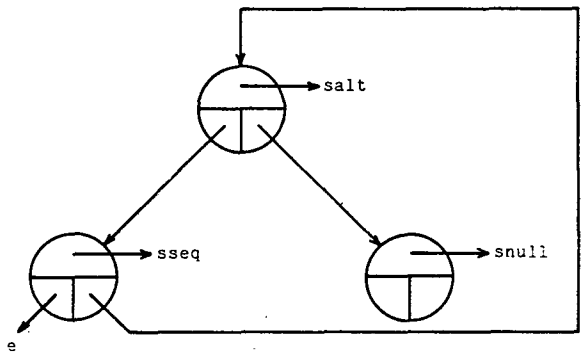
Here, the control regime is imbedded in the structural loop. Other control regimes are easy to construct. For example, *arbno*(*e*) is "reluctant," attempting to match as few instances of *e* as context will allow. A more "enthusiastic" procedure is

```
rpt := procedure (e:environment) private x;
   x := create salt;
   return (x with (copy(e) -- x, snull))
end
```

which attempts to match as many instances of *e* as context will allow. The procedure *rpt* builds a structure that is the equivalent of the recursive pattern

```
x := e -- *x | ""
```

An environment returned by *rpt* matches one less instance of *e* each time it is resumed. Environments built by *rpt* have the form



Many other pattern matching procedures and structures are possible. Unlike a language such as SNOBOL4 in which the pattern-matching primitives are fixed and built into the language, SL5 permits construction of pattern-matching operations that can be tailored to the needs of specific situations. Further details of string pattern-matching in SL5 are given in Refernces 9 and 10; another example of backtracking in SL5 is given in Reference 11.

4. Conclusions

Although SL5 was not designed specifically for AI applications, it seems to represent an alternate approach to the problems of control in programming languages that has received much attention recently (see Reference 12).

As demonstrated by the string pattern-matching example, the SL5 facilities permit the programmer to construct whatever control hierarchies and relationships best suit the particular application. The SL5 procedure mechanism provides a way to achieve this flexibility in a reasonably structured fashion.

In the pattern-matching example, it is important to note that the model used, as well as the specific procedures, are not restricted to string pattern matching, but can be extended to the more

general data structures that are typically found in AI applications. This is particularly true of those scanning procedures such as *salt* and *sseq*, and the construction procedures *arbno*(*e*) and *rpt*(*e*) that serve only to establish control relationships. The scanning environments that they control could equally well synthesize strings or parse trees. Similarly, the subject could be an arbitrary structure. The concept of a cursor is also easily generalized. The procedure *smove* can be easily generalized to index through an array, advanced through a linked list, or sequence through a data base.

Acknowledgments

SL5 is the result of the work of several persons. Significant contributions have been made by Dianne E. Britton, Frederick C. Druseikis, and John T. Korb. Discussions with Drew V. McDermott concerning AI and AI programming languages were especially helpful.

References

1. D. G. Bobrow and B. Rapheal. New Programming Languages for Artificial Intelligence Research, Comp. Surveys, 6, Sept. 1974, 153-174.

2. C. E. Hewitt. PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot, Proc. IJCAI-69, May 1969, 295-301.

3. I. Grief and C. E. Hewitt. Actor Semantics for PLANNER-73, Conf. Rec. ACM Symp. on Princ. of Prog. Lang., Jan. 1975, 67-77.

4. D. V. McDermott and G. J. Sussman. From PLANNER to CONNIVER - A Genetic Approach, Proc. FJCC, 1972, 1171-1179.

5. G. L. Steele. LAMBDA, The Ultimate Declarative, AI Memo 379, Mass. Inst. of Tech., Nov. 1976.

6. R. E. Griswold and D. R. Hanson. An Overview of SL5, SIGPLAN Notices, 12, April 1977, 40-50.

7. D. R. Hanson and R. E. Griswold. The SL5 Procedure Mechanism, Comm. ACM, to appear.

8. F. C. Druseikis and J. N. Doyle. A Procedural Approach to Pattern Matching in SNOBOL4, Proc. ACM Annual Conf., Nov. 1974, 311-317.

9. R. E. Griswold. String Analysis and Synthesis in SL5, Proc. ACM Annual Conf., Oct. 1976, 410-414

10. R. E. Griswold. The SL5 Programming Language and Its Use for Goal-Directed Programming, Proc. Fifth Texas Conf. on Computing Systems, Oct. 1976, 1-5.

11. D. R. Hanson. A Procedure Mechanism for Backtrack Programming, Proc. ACM Annual Conf., Oct. 1976, 401-405.

12. C. E. Hewitt and B. Smith. Towards a Programming Apprentice, IEEE Trans. on Software Eng., SE-1, March 1976, 26-45.