Programming Languages

J. J. Horning
Editor

# The SL5 Procedure Mechanism

David R. Hanson
Yale University

Ralph E. Griswold
The University of Arizona

**This paper describes an integrated procedure mechanism that permits procedures to be used as recursive functions or as coroutines. This integration is accomplished by treating procedures and their activation records (called environments) as data objects and by decomposing procedure invocation into three separate components at the source-language level. In addition, argument binding is under the control of the programmer, permitting the definition of various methods of argument transmission in the source language itself. The resulting procedure mechanism, which is part of the SL5 programming language, is well suited to goal-oriented problems and to other problems that are more readily programmed by using coroutines. Several examples are given.**

**Key Words and Phrases: procedures, coroutines, programming languages, interpreters, SNOBOL4, backtracking**

**CR Categories: 4.2, 4.20, 4.22, 4.13**

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

Authors' addresses: D. R. Hanson, Department of Computer Science, Yale University, New Haven, CT 06520; R. E. Griswold, Department of Computer Science, The University of Arizona, Tucson, AZ 85721.

## 1. Introduction

The procedure has always been an important component of programming languages. Recognition of the procedure as one of the most powerful tools for abstraction [6] has focused substantial attention on procedure mechanisms [10]. Recursive functions have become accepted in high-level languages, but coroutines, in spite of their long history [19], have been confined mainly to operating system applications.

Recent increased interest in search and backtrack algorithms has focused more attention on coroutines. The primary areas of application have been AI languages [2, 20] and string pattern matching [8]. Other areas of interest include simulation [17], applicative languages [4, 23], and extensible languages [27].

While several languages have coroutine facilities [1, 2, 25, 26, 28], these facilities are generally designed around specific applications and areas of interest, rather than being integrated components of the procedure mechanism of the language.

The SL5 programming language was designed as a research tool for investigation into high-level programming language facilities and programming methodologies in string and structure processing. Mechanisms for supporting search and backtrack algorithms were an important consideration, and motivated a procedure mechanism from which coroutine programming follows naturally. This procedure mechanism is characterized by the following features:

1. Procedures and activation records for procedures (called environments) are source-language data objects.
2. Procedure invocation is decomposed into three separate components available to the programmer at the source-language level.
3. The interpretation of identifiers (scoping) is dynamic and is designed to provide for a method of interprocedure communication that is useful for coroutine programming.
4. Argument binding and transmission are under the control of the programmer.

This paper describes the SL5 procedure mechanism and illustrates its use.

## 2. The SL5 Programming Language

SL5 is similar in many respects to SNOBOL4 [13] and has most of the facilities of SNOBOL4. SL5 departs in many respects from SNOBOL4, most notably as described above and in its syntax and control structures. It has an expression syntax with reserved words. Expressions produce both values and signals, with signals being used primarily for control purposes.

Runtime flexibility, motivated by the research orientation of its expected applications, is emphasized. SL5 supports many data types, but has no type declarations. Type checking and coercion, where appropriate, are performed dynamically. The remainder of this section gives a brief introduction to SL5 to provide a framework for the material in the rest of this paper. A summary of the language is given in [12].

## 2.1 The Result of SL5 Expressions

An SL5 expression returns a *result*, which is composed of a *value* and a *signal*. The value component of a result corresponds to the conventional concept of a value. The signal component of a result is a nonnegative integer and is used for control purposes. For example, built-in control expressions in SL5 are driven by two signals—success and failure—which are indicated by 1 and 0, respectively. A result is indicated by {value, signal}. The selectors $V$ and $S$ are used to refer to the value and signal components of a result, respectively. For example, if $r$ is a result, $V(r)$ stands for the value component of the result $r$.

The procedure mechanism allows the programmer to transmit nonnegative integers as signals and to attach whatever meaning to those signals that is needed for the specific application. In this paper, the names "success" and "failure" are used to mean 1 and 0, respectively.

The $V$ component of a result can be a variable, that is, a place where a value resides, such as a reference to an identifier. The term *dereferencing* refers to obtaining the value of a variable. This operation returns a result containing the value of the variable.

The dereferencing operation distinguishes *interpretation* from *evaluation*. Interpretation is the execution of the basic operation given in an expression. Evaluation is the combination of interpretation followed by dereferencing. The basic operations of interpretation and evaluation are an integral part of the procedure mechanism.

A result may be composed by using the & operator. The expression $e_1$ & $e_2$ returns the result $\{V(e_1), V(e_2)\}$. The signal components of $e_1$ and $e_2$ are ignored.

## 2.2 Control Expressions

Programming languages typically provide control structures in the form of statements, such as the familiar if-then-else construct. In SL5 control operations are handled by expressions, which are referred to as "control expressions" since their primary purpose is to control program execution. Although SL5 control expressions generally resemble control structures in other languages, they are driven by signals, rather than by boolean values. An example is

**if** $e_1$ **then** $e_2$ **else** $e_3$

The expression $e_1$ is evaluated first. If the resulting signal is 1, $e_2$ is evaluated. Otherwise, $e_3$ is evaluated. For example, in

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

if $x$ is greater than $y$, the difference of $x$ and $y$ is assigned to $x$. Otherwise, the difference of $y$ and $x$ is assigned to $y$. The if-then-else construct is itself an expression and returns a result, which is the result (value and signal) of $e_2$ or $e_3$ depending on the signal of $e_1$.

SL5 has a large repertoire of control expressions in order to facilitate programming. The following control expressions are representative:

**while** $e_1$ **do** $e_2$
**unless** $e_1$ **do** $e_2$
**case** $e$ **of** $l_1$:$e_1$; ... $l_n$:$e_n$; **default**: $e_{n+1}$ **end**
**repeat** $e$
$e_1$ **and** $e_2$
$e_1$ **or** $e_2$
$\{e_1; \dots ; e_n\}$

The **while** and **unless** expressions have conventional interpretations consistent with the signaling mechanism. In the **case** expression, $l_1, \dots , l_n$ can be any literals (strings, integers, and so on). The **repeat** expression evaluates $e$ repeatedly until $e$ signals failure. In the expression $e_1$ **or** $e_2$, $e_1$ is evaluated first. If $e_1$ succeeds, its result becomes the result of the control expression. Otherwise, $e_2$ is evaluated and its result becomes the result of the control expression. The **and** expression is complementary; if the evaluation of $e_1$ succeeds, $e_2$ is evaluated and its result is the result of the control expression. If evaluation of $e_1$ fails, the result of the **and** expression is the result of $e_1$. The braces enclose a sequence of expressions. The result of this control expression is the result of the last expression in the sequence.

## 3. Procedure Creation

In SL5, procedures are data objects. A procedure is created by the expression

**procedure** ( $\langle id \rangle_1$, $\langle id \rangle_2$, ... , $\langle id \rangle_n$) ⟨declarations⟩ ⟨body⟩ **end**

where $\langle id \rangle_1$ through $\langle id \rangle_n$ are the formal arguments, ⟨*declarations*⟩ are of the form given below, and ⟨*body*⟩ is a sequence of expressions separated by semicolons. For example,

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

assigns to *gcd* a procedure that computes the greatest common divisor of its arguments. Procedures can be created at any time during program execution and can be manipulated as values throughout the program.

## 4. Decomposition of Procedure Activation

In most programming languages, the invocation of a procedure is considered to be an atomic operation. In SL5, the invocation of a procedure is decomposed into three distinct components that are available to the programmer at the source-language level. This decomposition provides the linguistic mechanism for SL5 procedures to be used as recursive functions or as coroutines. The steps in this decomposition are: **create**—creation of an environment for the procedure; **with**—binding of the actual arguments to an environment; **resume**—resumption of the execution of the procedure for an environment.

## 4.1 Environment Creation

The expression

$e$ := **create** $p$

creates an environment for the procedure $p$ and assigns this environment to $e$. An environment for a procedure contains the storage for variables corresponding to the identifiers appearing in the procedure. An environment also contains the procedure continuation point, which indicates where execution is to begin when the environment is activated. When an environment for a procedure is created, the continuation point is at the beginning of the procedure. When a procedure returns, the continuation point is set to the position following the point of return. An environment is a source-language data object that can be manipulated as such.

The environment in which the **create** $p$ is executed is called the *creator* for that environment. Each environment contains information identifying its creator.

## 4.2 Argument Binding

The binding of the actual arguments to an environment is accomplished by the **with** expression. The expression

$e$ **with** $(a_1, a_2, \dots, a_n)$

transmits the actual arguments, $a_1$ through $a_n$, to the environment $e$ and returns that environment as its value. The methods by which the actual arguments are transmitted to the environment are controlled by the argument transmitters associated with the procedure for which $e$ is an environment. This mechanism is described in Section 6. In the absence of any explicit specification of transmitters, arguments are transmitted by value, that is, the expressions $a_1$ through $a_n$ are *evaluated* and the $V$ components of their results are assigned to the formal arguments for that environment. If the evaluation of any of the argument expressions fails, the **with** expression is aborted and fails.

## 4.3 Procedure Resumption

A procedure is activated by the **resume** expression. In its simplest form, the **resume** expression is written

**resume** $e$

where $e$ is an environment. In the discussion that follows, this operation is referred to as "resuming $e$."

The **resume** expression causes the execution of the current procedure to be suspended and $e$ to be resumed. The current procedure is suspended within the **resume** expression itself. When the environment is subsequently reactivated, the **resume** expression produces the result that is provided by the environment causing the subsequent reactivation.

The general form of the **resume** expression is

**resume** $(e, r)$

This expression suspends the current environment, resumes $e$, and transmits the result of interpreting $r$ to $e$.

The **resume** expression also establishes the *resumer* for an environment—the environment that caused its most recent resumption via the **resume** expression. Note that an environment's resumer changes during the course of program execution, whereas the creator, defined above, remains constant, since an environment is created only once.

## 4.4 Procedure Returns

The **resume** expression provides one method for communicating results among procedures. The other method is by the **return** expression. While the **resume** expression requires an explicit indication of the environment to which control should be transferred, the **return** expression returns control to the *resumer* of the current environment. The expression

**return** $r$

returns the result of interpreting $r$. If $r$ is omitted in **return**, the null string is assumed. Like **resume**, the **return** expression causes the current procedure to be suspended. When the procedure is subsequently reactivated, the result transmitted becomes the result of the **return** expression.

The important difference between **return** and **resume** is that **return** does not establish a new resumer for the environment to which control is returned. Only an explicit **resume** establishes a resumer.

Since the signals success and failure are used so frequently in SL5, the expressions

**succeed** $r$ and **fail** $r$

are provided as equivalents, respectively, to

**return** $r$ & 1 and **return** $r$ & 0

Note that a value is returned even if the signal is failure. If $r$ is omitted, the null string is assumed.

## 4.5 Functional Notation

The abbreviated notation $f(e_1, e_2, \dots, e_n)$ may be used for the usual recursive function invocation. This form of procedure invocation is equivalent to

**resume** (**create** $f$ **with** $(e_1, e_2, \dots, e_n)$)

Note that the functional notation results in the invocation of the procedure that is the current value of $f$, not a procedure named $f$. This feature permits functional composition since procedures are data objects. For example, $f(x)(a, b)$ first invokes the procedure given by the value of $f$ with the argument $x$. Assuming that $f$ returns a procedure as its result, that procedure is then invoked with the arguments $a$ and $b$.

## 4.6 Two Simple Examples

Assuming that *gcd* has been assigned the procedure given in Section 3, the conventional function invocation of *gcd*

```
z := gcd(45, 27);
```

results in $z$ being assigned 9 and is equivalent to the decomposition

```
c := create gcd;
c := c with (45, 27);
z := resume c;
```

Note that the expression **return** $x$ in *gcd* transmits the result {9, success}, which in turn becomes the result of the expression **resume** *c*. The $V$ component of that result is then assigned to $z$. The procedure *gcd* is not written to be used as a coroutine, and resumption of $c$ after it has returned would be a programming error.

To illustrate a simple application of coroutine usage, consider a label generator for generating a series of labels:

```
genlab := procedure (prefix, n, limit)
  repeat
    if n ≤ limit then {succeed prefix || n; n := n + 1}
    else fail
end;
```

(The operator || denotes string concatenation as in PL/I.) The arguments *prefix*, *n*, and *limit* indicate the prefix for the labels, the initial label number, and the maximum label number, respectively. For example,

```
gen := create genlab with ("L", 10,100);
```

assigns to *gen* an environment that generates the label sequence $L10$, $L11$, ... , $L100$. To obtain the next label in this sequence, *gen* is resumed:

```
x := resume gen;
```

After the label $L100$ has been generated, subsequent resumptions of *gen* result in failure. The sequence may be reinitialized by retransmitting the arguments. For example,

```
gen with ("L", 10,1000);
```

resets *gen* with the same prefix, the original value of $n$, and a limit of 1000.

## 4.7 Generalized Transfer of Control

As illustrated by the examples above, coroutine programming follows naturally from the treatment of environments as data objects and the decomposition of procedure invocation. While experience indicates that the implicit updating of the resumer that is performed by the **resume** operation is usually desirable, there are situations in which it is necessary to resume an environment without changing the resumer.

This kind of resumption is accomplished by a more general form of the **return** expression. In most circumstances, the **return** expression is used to return *from* an environment to its resumer. The **return** expression also may be used to return *to* an arbitrary environment. The **return** expression is used in this fashion to resume an environment without changing the resumer.

The general form of the **return** expression is

```
return r to e
```

This expression suspends the current environment, resumes the execution of $e$, and transmits the result of interpreting $r$ to $e$. If $r$ is omitted, the null string is assumed.

To facilitate the construction of general control relationships, the built-in functions *creator*($e$) and *resumer*($e$) are provided to specify the creator and resumer of $e$, respectively. If $e$ is omitted, the current environment is assumed. Thus, the return expression

```
return r
```

is equivalent to

```
return r to resumer ( )
```

## 5. The Interpretation of Identifiers

In search and backtrack algorithms, the relationships between procedures often are most conveniently derived from data. This motivates a procedure mechanism in which the binding of identifiers is dynamic (during program execution), rather than static (during program compilation).

Declarations in SL5 procedures are used to determine the scope of identifiers. The scoping conventions in SL5 are similar to the dynamic scoping conventions [7] used in SNOBOL4 and LISP.

An identifier may be declared either **public** or **private** in declarations having the form

**public** $\langle id\rangle_1, \langle id\rangle_2, \ldots, \langle id\rangle_n$
**private** $\langle id\rangle_1, \langle id\rangle_2, \ldots, \langle id\rangle_n$

An identifier that does not appear in any of the declarations for the procedure in which it is used is termed *nonlocal.*

## 5.1 Private Identifiers

The scope of a private identifier is restricted to the procedure, and only the procedure, in which it is declared. Private identifiers are used for data that is local to a particular environment for a procedure. For example, when a procedure is used as a coroutine, private identifiers can be used to "remember" information from one resumption to the next. This type of facility is useful when procedures are used for goal-oriented programming involving backtracking. Unless otherwise declared, the formal argument identifiers for a procedure are considered to be private identifiers.

### 5.2 Public and Nonlocal Identifiers

Public identifiers provide the principal means of inter-procedure communication. Public declarations provide the information that is necessary to determine the scope of nonlocal identifiers. This determination is guided by the way in which environments are connected to each other, namely, as descendants. An environment is the descendant of its creator. Descendants of an environment possess the same transitive closure property that descendants shown in lineal charts do, that is, an environment is a descendant of its creator, its creator's creator, and so on.

The scope of nonlocal identifiers is determined when an environment for the procedure is created, and is obtained by examining the creation history. For each nonlocal identifier in a newly created environment, a search is performed by examining each successive creator until an environment containing a public identifier by the same name is found. If the search is successful, the nonlocal identifier henceforth refers to that public identifier. If this search fails, the nonlocal identifier is considered to be erroneous.

These conventions give SL5 aspects of both dynamic and static scope. The initial interpretation is dynamic and is made at the time of creation. Once an environment is created, the interpretation of nonlocal identifiers remains static throughout the lifetime of the environment. As a result, environments are similar to lambda closures (FUNARGS) in LISP [21]. One of the important differences between lambda closures and environments is that environments include an implicit indication of their execution state, which is used to continue execution upon resumption. Implementation of a similar mechanism using lambda closures requires that each execution state be represented explicitly by a separate lambda closure [24].

### 5.3 Accessing Identifiers in Environments

The identifiers in an environment are attributes of that environment. They may be accessed from outside the environment by using the environment reference operator, which is indicated by an infix dot. The expression *e.i* refers to the identifier *i* in the environment *e*. The environment reference operator provides an external means of fetching and setting the values of identifiers in inactive environments. For example, the sequence number in the label generator described in Section 4 can be reset as follows:

*gen.n* := *m*;

## 6. Argument Binding and Transmission

The **with** operator is used to bind the actual arguments to an environment. The method by which this is done is determined by the argument transmitters associated with the arguments. Transmitters may be built-in or programmer-defined procedures or environments.

Transmitters are specified when a procedure is created. For example,

*gcd* := **procedure** (*x*:*val*, *y*:*val*) ··· **end**;

creates a procedure with two arguments, *x* and *y*, each of which has the transmitter that is the value of the identifier *val*. In general, a formal argument of a procedure has the form

[⟨*scope*⟩] ⟨*id*⟩ [: ⟨*exp*⟩]

where ⟨*scope*⟩ is either **public** or **private**, ⟨*id*⟩ is the formal argument, and the value of ⟨*exp*⟩ is the transmitter for that argument. The transmitter expression ⟨*exp*⟩ is evaluated when the procedure is created.

For example, the procedure heading

**procedure** (*i*, **public** *x*:*val*, **public** *y*:*val*, **private** *z*:*ref*)

specifies a procedure with four arguments. The first argument, *i*, has the default scope and transmitter—**private** and *val*. The second and third arguments, *x* and *y*, have the same transmitter and are declared public. The fourth argument, *z*, is private and its transmitter is *ref*, which transmits arguments by reference.

### 6.1 Transmission

When the **with** expression is invoked, either implicitly or explicitly, each actual argument expression is passed to the transmitter associated with the corresponding formal argument for the procedure. The value returned by the transmitter is established as the value of the formal argument identifier. If any of the transmitters fails, the **with** expression fails, and any remaining arguments are not transmitted.

If the number of actual arguments exceeds the number of formal arguments, the excess argument expressions are ignored and are not evaluated. If there are fewer actual arguments than formal arguments, null strings are provided for the omitted arguments. In this case, the transmitter for an omitted argument is invoked with the null string as argument.

### 6.2 Built-in Transmitters

The value of *val* is a built-in transmitter that transmits arguments by value. In this case, the actual argument expression is *evaluated* and, provided that evaluation is successful, the value component of the result is established as the value of the associated formal argument. The signal component of the result is discarded. If the evaluation fails, the argument transmission fails.

The value of *ref* is a built-in transmitter that transmits arguments by reference. The actual argument expression is *interpreted*. If the actual argument is a variable, assignments and references to the formal argument refer to the actual argument that is passed. For example, if the *gcd* procedure given in Section 3 is defined by using the procedure heading

*gcd* := **procedure** (*x*:*ref*, *y*:*ref*) ··· **end**;

the expressions

```
a := 45;
b := 27;
c := gcd(a, b);
```

result in the values of *a*, *b*, and *c* being changed to 9.

Since argument binding is a separate operation and is not a part of procedure resumption, transmission by reference can be used to pass arguments before their intended values are assigned. For example, the sequence

```
e := create gcd with (a, b);
a := 45;
b := 27;
c := resume e;
```

has the same effect as above; the formal arguments *x* and *y* refer to *a* and *b* in the calling procedure, not just to their values.

The value of *exp* is a built-in transmitter that transmits arguments by expression, which is similar to transmission by name in Algol 60. The actual argument expression is interpreted at each reference to the corresponding formal argument in the procedure. This interpretation is performed in the environment in which the argument transmission occurred.

## 6.3 Programmer-Defined Argument Transmitters

A transmitter may be a built-in procedure, a programmer-defined procedure, or an environment. Programmer-defined transmitters can be used to perform datatype checking, tracing, or common preprocessing of arguments. For example,

```
integerval := procedure (x)
  if ident(datatype(x), "integer") then
    succeed x;
  out := "integer argument expected";
  fail
end;
```

assigns to *integerval* a procedure that checks the datatype of its argument. If the argument is not an integer, an error message will then be issued and transmission fails. If one wishes to use *integerval* as the transmitter for the arguments to *gcd*, the following procedure heading may be employed:

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

Since procedures are data objects, it is possible to provide transmitters such as *integerval* for documentation and debugging purposes and then to change their values in order to dispense with the time-consuming type checking after the program is working. For example, the assignment

```
integerval := val;
```

changes integerval to the default value transmission transmitter.

Like other identifiers in the language, the values of the identifiers *val* and *ref* may be changed. For example, by changing the value of *val*, it is possible to change the default argument transmitter for subsequently created procedures. Further details concerning argument transmitters are given in [15].

# 7. Examples

## 7.1 A Generator of Random Number Generators

The values of private identifiers in an environment 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 environment for a procedure. For example, consider the procedure *rangen* defined as follows.

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

The procedure *rangen* computes a sequence of pseudorandom numbers using the linear congruence method [18]. An environment for *rangen* computes the next number within the range 1 to *n* in the sequence defined by the parameters *s*, *p*, *c*, and *m*. An environment for *rangen* is parameterized by the values of the arguments to *rangen* and generates the next number from an independent sequence every time it is resumed. For example,

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

assigns to *rg* an environment for *rangen*. To obtain the next number in the sequence, *rg* is resumed:

```
x := resume rg;
```

The sequence of pseudorandom numbers may be restarted by retransmitting the arguments, as was done for the label generator, or by simply resetting the seed *s* to its original value:

```
rg.s := 0
```

Since it is the environments for *rangen* that constitute the generators, any number of generators can be created by using the single common procedure *rangen*. For instance, the expressions

```
rg2 := create rangen with (0, 12641, 11241, 10000, 10);
rg3 := create rangen with (111, 12321, 12231, 100000, 50);
```

assign two separate environments for *rangen* to *rg*2 and *rg*3. Each generator, *rg*, *rg*2, and *rg*3, generates an independent sequence of pseudorandom numbers.

This approach to independent generators differs from the usual methods that require extraneous data structures or the creation of separate procedures for each generator. Here, there is only a single procedure, with separate environments for each generator. With this approach, it becomes natural to conceptualize program organization in terms of environments, rather than procedures. Thus the creation of appropriate environments is of more concern than the procedures that these environments use.

## 7.2 Sentence Generation

A less familiar application of coroutine programming is illustrated by the problem of systematically generating the sentences of the language described by a context-free grammar. The approach is to construct an environment for each symbol of the grammar. Each time such an environment is resumed, it generates a sentence of the language it describes.

The form of environment for a terminal symbol is very simple and has the procedure

```
terminal := procedure (s)
  return s;
  fail
end;
```

For example,

```
A := create terminal with "A";
```

creates an environment for the terminal symbol *A*.

When the environment for a terminal symbol is resumed, it returns that symbol. On the next resumption it fails, since there are no other sentences for this symbol. By convention, a generating environment is not resumed after it has signaled failure.

In a BNF-like grammar, nonterminal symbols are defined in terms of the alternation and concatenation of other symbols. These relations correspond to environments that control the resumption of their arguments. The alternative of two symbols is represented by an environment for

```
alt := procedure (e1, e2) private s;
  e1 := copy(e1);
  while s := resume e1 do succeed s;
  e2 := copy(e2);
  while s := resume e2 do succeed s;
  fail
end;
```

The *copy* operations in this procedure make fresh copies of $e_1$ and $e_2$ to avoid possible interference from other resumptions of these environments. The first environment is repeatedly resumed until it fails, thus yielding all the sentences of $e_1$. Next all the sentences of $e_2$ are generated.

The concatenation of two symbols is represented by an environment for

```
cat := procedure (e1, e2) private s1, s2;
  e1 := copy(e1);
  while s1 := resume e1 do {
    e2 := copy(e2);
    while s2 := resume e2 do succeed s1 || s2
    };
  fail
end;
```

In this case, every sentence of $e_2$ is concatenated onto each sentence of $e_1$.

Environments for the alternation of symbols are created by

```
altsym := procedure (e1:genenv, e2:genenv)
  return create alt with (e1, e2)
end;
```

A procedure, *catsym*, for creating environments for concatenation is similar. The transmitter, *genenv*, is used to coerce terminal symbols to corresponding generating environments and is given by

```
genenv := procedure (x)
  case datatype(x) of
    "environment": return x;
    "string": return create terminal with x;
    default: fail
  end
end;
```

SL5 permits the association of procedures with operator symbols (see [11] for the method). This allows a syntax for the construction of generators that approximates the notation of BNF. For example, if the infix operators | and -- are chosen for *altsym* and *catsym*, respectively, the generators for the BNF grammar

```
<D>: := d|.d.
<GOAL>: := a|b<D>
```

are constructed by the SL5 expressions

```
D := "d"|".d.";
GOAL := "a"|("b" -- D);
```

The sentences for *GOAL* are generated by

```
repeat out := resume GOAL;
```

which produces

```
a
bd
b.d.
```

Other generators can be added to extend the basic set given above. For example, recursive references to nonterminal symbols can be handled by an environment that defers the evaluation of the nonterminal symbol until it is needed during generation. It should be noted that unless the first argument to *alt* ($e_1$) and the second argument to *cat* ($e_2$) generate finite sublanguages, there are portions of the entire language that are never reached. This problem can be avoided by more sophisticated versions of *alt* and *cat* that use a scheme similar to the method for enumerating the union of two denumerable sets.

Enhancements to the somewhat barren descriptive power of BNF are easily provided. Of course, there is no barrier to extensions beyond the domain of context-free languages since any type of computation can be incorporated in a generator.

It should be noted that the conventions used above for generation of successive sentences on successive resumptions and the use of the failure signal to signal the end of a language comprise a sort of communication protocol. The synthetic process of generation is, in fact, closely related to the analytic process of pattern matching. Pattern matching requires a somewhat more complicated communication protocol because of the backtracking required. See [11] for a description of the process.

## 8. Comparison with Other Mechanisms

Coroutines have been proposed or have appeared in other programming languages in various forms. One of the earliest proposals was made by McIlroy [19], who suggested linguistic mechanisms for defining coroutines and for connecting coroutines together in various control relationships. That proposal, like several others, requires that the identity of coroutines connected together be known to each other. The equivalent of the SL5 **return** expression was not included. Such control relationships can be implemented in SL5 by assigning the environments of interest to public variables or by using the environment reference operator, described in Section 5.3, to make connections between environments.

The asymmetry of the SL5 operations of **resume** and **return** permits the realization of many control relationships without having to know the identity of all of the environments involved. This is a result of using a decomposition of procedure invocation as the basis for the semantics of the **resume** and **return** expressions. The **return to** expression can be used for those situations where the **resume** and **return** expressions do not match the desired relationship well.

Another approach, used in connection with LISP systems, is to use lambda closures as environments, passing them to other functions as explicit continuation parameters [4, 23]. Instead of returning, the functions "resume" their continuations. Coroutines can therefore be implemented by constructing appropriate continuations.

While this approach, based on the lambda calculus, is mathematically elegant, it is somewhat clumsy in practice. The problem is that the programmer must be too explicit. In effect, these languages do not include coroutines as an integral component of the language, but rather provide facilities that permit the programmer to implement coroutines. For example, the programmer must explicitly save the continuation point of a coroutine in some form, depending on the particular implementation scheme that is used. This is accomplished by the construction of a continuation, for instance. In SL5, these kinds of operations are implicit, permitting the programmer to work in terms of control relationships instead of their implementation.

Similar comments apply to schemes for including backtracking mechanisms in programming languages by permitting the explicit manipulation of stack frames [3, 22]. This approach has been used, for example, in CONNIVER [25]. Using this approach, the programmer must think in terms of "reactivating" stack frames, and continuation points for a coroutine must be indicated explicitly. In addition, the coroutines that correspond to the SL5 notions of creator and resumer must be explicitly indicated when transferring control. The explicit manipulation of stack frames is similar to the use of so-called "label variables" [9], which are used to "capture" the bindings associated with a particular program block. By transferring to a label of this kind, the saved bindings are "reinstated" and execution continues. This technique is used in GEDANKEN [23] to implement coroutines.

The class mechanism of Simula 67 [1, 5] is, in some respects, similar to the SL5 procedure mechanism. In Simula 67, an instance of a block may outlive its calling statement. Block instances of this kind are called classes.

Classes are used mainly for data structures, but they also can be used as coroutines. The basic similarities between SL5 environments and Simula 67 classes are that they are both data objects and both may be used as coroutines. One difference is that there is no way in Simula 67 to transmit values to a class when it is resumed, except through global variables. The principal differences between SL5 and Simula 67, however, lie in the operations that govern the transfer of control among environments in SL5 and classes in Simula 67.

SL5 permits procedure decomposition at the source-language level. Simula 67 does provide an operation similar to the **resume** expression for use with classes, but does not permit a general decomposition of recursive function invocation. As a result, recursive functions and coroutines are treated as linguistically different concepts in Simula 67, whereas they are treated as variations of the same linguistic concept in SL5.

The difference between procedures and classes in Simula 67 can be understood by examining the subtle subordinate relationship that classes have to procedures. In Simula 67, classes are said to be "attached" to a procedure instance. This attachment is made when the class is created and can be changed by the primitive **call**( $x$ ), which causes class $x$ to be attached to the current procedure instance and the execution of $x$ to be resumed. This operation is similar to the SL5 expression **resume** when the resumer is viewed as the means of attachment. A class can "detach" itself and return control to the procedure to which it is attached by executing the primitive **detach**. This is similar to the **return** expression in SL5. In addition, a class can pass control to another class using the primitive **resume**( $y$ ), which causes the execution of class $y$ to continue. This primitive also causes $y$ to become attached to whatever procedure instance $x$ is attached. Hence the execution of a **detach** in $y$ causes a return to that procedure, not a return to $x$ at the point of the **resume**( $y$ ) primitive.

A subordinate relationship is imposed by the restriction that classes can be attached only to procedure instances; a class cannot be attached to another class. In addition, the **detach** and **resume** primitives can be executed only from within a class. The SL5 mechanism was designed specifically to avoid these kinds of subordinate relationships. While it is possible to devise schemes that favor each mechanism, initial experience with SL5 suggests that the treatment of functions and coroutines as alternative ways of using a general procedure mechanism is more flexible.

## 9. Conclusions

Experience in using SL5 has shown it to be a good vehicle for coroutine programming. In addition to the classical problems [16], it has been used to implement, at the source-language level, the built-in pattern-matching mechanism of SNOBOL4 and the more extensive string analysis and synthesis facility that is built into SL5 [11]. SL5 appears to be well suited for the kinds of goal-oriented programming problems that arise so frequently in artificial intelligence applications and around which AI languages proliferate [2]. In addition, the procedure mechanism is the basis for the SL5 data structuring facility [14] and for a facility that enables an environment to be connected to a variable for filtering assignment and value fetching [15].

While a number of programming languages have coroutine mechanisms, the distinguishing characteristic of SL5 is the incorporation of these facilities as a natural part of the procedure mechanism of the language. In practice, the utility of these features seems to derive largely from the decomposition of procedure activation into more primitive components and the consequent characterization of coroutines as an integral part of the procedure mechanism, instead of their being a separate ad hoc mechanism. An additional advantage is gained by the ability to manipulate procedures and environments as source-language data objects, which allows the programmer to control processes that are traditionally hidden in the implementation.

*Acknowledgments.* SL5 is the result of the work of several persons and synthesizes results from related research. In particular, we would like to express our gratitude to Dianne E. Britton, Frederick C. Druseikis, and John T. Korb for their contributions to SL5 and its implementation. We would also like to thank Drew V. McDermott for his helpful suggestions concerning the preparation of this paper.

Received October 1976; revised April 1977

## References

**1.** Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K. *SIMULA Begin.* Student Literature, Auerbach, Philadelphia, Pa., 1973.

**2.** Bobrow, D.G., and Raphael, B. New programming languages for artificial intelligence. *Computing Surveys 6*, 3 (Sept. 1974), 155–174.

**3.** Bobrow, D.G., and Wegbreit, B. A model and stack implementation of multiple environments. *Comm. ACM 16*, 10 (Oct. 1973), 591–603.

**4.** Burge, W.H. *Recursive Programming Techniques.* Addison-Wesley, Reading, Mass., 1975.

**5.** Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. *Structured Programming.* Academic Press, London, 1972, Sect. III.

**6.** Dijkstra, E.W. The humble programmer. *Comm. ACM 15*, 10 (Oct. 1972), 859–866.

**7.** Dijkstra, E.W. Recursive programming. In *Programming Systems and Languages*, S. Rosen, Ed., McGraw-Hill, New York, 1967.

**8.** Druseikis, F.C., and Doyle, J.N. A procedural approach to pattern matching in SNOBOL4. Proc. ACM Annual Conf., Nov. 1974, pp. 311–317.

**9.** Fenichel, R. On the implementation of label variables. *Comm. ACM 14*, 5 (May, 1971), 349–350.

**10.** Fisher, D.A. A survey of control structures in programming languages. SIGPLAN Notices (ACM) *7*, 11 (Nov. 1972), 1–13.

**11.** Griswold, R.E. The SL5 programming language and its use for goal-directed programming. Proc. Fifth Texas Conf. on Comptng. Syst., Oct. 1976, pp. 1–5.

**12.** Griswold, R.E., and Hanson, D.R. An overview of SL5. SIGPLAN Notices (ACM) *12*, 5 (April 1977), 40–50.

**13.** Griswold, R.E., Poage, J.F., and Polonsky, I.P. *The SNOBOL4 Programming Language.* Prentice-Hall, Englewood Cliffs, N. J., second ed., 1971.

**14.** Hanson, D.R. Data structures in SL5. To appear in *J. Computer Languages.*

**15.** Hanson, D.R. Filters in SL5. To appear in *Computer J.*

**16.** Hanson, D.R. A procedure mechanism for backtrack programming. Proc. ACM Annual Conf., Oct. 1976, pp. 401–405.

**17.** Kiviat, P.J., Villanueva, R., and Markowitz, H.M. *The SIMSCRIPT II Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1968.

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

**19.** McIlroy, M.D. Coroutines. Tech. Rep., Bell Labs., Murray Hill, N.J., May 1968.

**20.** Melli, L.F. The 2.Pak language primitives for AI applications. M.S. Th., Dept. Comptr. Sci., U. of Toronto, Toronto, Canada, Dec. 1974.

**21.** Moses, J. The function of FUNCTION in LISP. SIGSAM Bulletin (ACM) *4* (July 1970), 13–27.

**22.** Prenner, C.J., Spitzen, J.M., and Wegbreit, B. An implementation of backtracking for programming languages. Proc. ACM Annual Conf., Aug. 1972, 763–771.

**23.** Reynolds, J.C. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. *Comm. ACM 13*, 5 (May 1970), 308–319.

**24.** Reynolds, J.C. Definitional interpreters for higher-order programming languages. Proc. ACM Annual Conf., Aug. 1972, pp. 717–739.

**25.** Sussman, G.J., and McDermott, D.V. From PLANNER to CONNIVER—a genetic approach. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 1171–1179.

**26.** VanLehn, K.A. SAIL user manual. Tech. Rep. STAN-CS-73-373, Dept. Comptr. Sci., Stanford U., Stanford, Calif., July 1973.

**27.** Wegbreit, B., et al. ECL programmer's manual. Ctr. Res. Comptg. Tech., Harvard U., Cambridge, Mass., 1970.

**28.** Wulf, W.A., Russell, D.B., and Habermann, A.N. BLISS: a language for systems programming. *Comm. ACM 14*, 12 (Dec. 1971), 780–790.