Procedure-Based Linguistic Mechanisms in Programming Languages

David Roy Hanson

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

August 26, 1976

This report reproduces a dissertation, titled as above, submitted to The University of Arizona in partial fulfillment of the requirements for the degree of Doctor of Philosopy.
This work was supported by the National Science Foundation under Grant DCR75-01307.

ABSTRACT

High-level programming languages are now widely accepted as the preferred tool for program construction because they provide facilities that permit the programmer to make effective use of the mental process of abstraction. A good programming language provides facilities that assist the programmer in transforming an abstract solution of a problem into a program. Most languages provide facilities for the realization of procedural and data abstractions. This dissertation describes a procedural approach to the design and implementation of advanced programming language facilities for realizing these kinds of abstractions. The proposed facilities are included as a part of the SL5 programming language.

SL5 provides an integrated procedure mechanism that permits procedures to be used as recursive functions or coroutines. This integration is accomplished by treating procedures and their environments as data objects, which can be manipulated by the programmer at the source-language level.

Environments for SL5 procedures are also used for data structures. Thus, SL5 provides a single unified linguistic mechanism for the realization of both procedural and data abstractions. The result is that the programmer can define the attributes and access mechanisms of a data structure to suit the application.

SL5 also permits the programmer to attach a procedural component to a variable that can be used to "filter" values assigned to the variable or to filter the value of the variable whenever it is fetched. The filter facility is the basis for the inclusion of such features as tracing, dynamic datatype checking, and dynamic protection schemes at the source-language level. Most importantly, filters are used for argument binding, permitting the programmer to define various methods of argument transmission in SL5 itself.

ACKNOWLEDGMENTS

The work described in this dissertation was supported by the National Science Foundation under Grant DCR75-01307.

The programming language SL5, described in this dissertation, was designed by a group consisting of Dianne E. Britton, Frederick C. Druseikis, and the author under the direction of Ralph E. Griswold. I would like to thank the members of the SL5 group for many stimulating discussions, which have had a substantial impact on the ideas embodied in this dissertation.

I would also like to thank the members of my committee, Douglas K. Brotz and G. David Ripley, for numerous suggestions and comments that contributed significantly to the finished product. The efforts of my advisor, Ralph E. Griswold, deserve special mention. His continual interest in my work, and his deep understanding of computer science were not only the driving forces behind this research, but will be sources of inspiration for the rest of my career.

Finally, and most importantly, I would like to thank my family, Marianne and Chris, for putting up with it all.

TABLE OF CONTENTS

1. INTRODUCTION

High-level programming languages are now widely accepted as an appropriate tool for program construction. The principal reason for this acceptance is that high-level programming languages usually provide facilities that permit the effective use of abstraction.

Abstraction

Abstraction is a mental process that attempts to embody a group of situations, objects, or processes in a single concept by concentrating on their similarities and suppressing their differences (Hoare 1972). The novel is an example of a common use of abstraction. The term "novel" is a symbolic way of referring to a class of books that have certain similarities. Novels exhibit obvious differences, for example, different plots, writing style, and even length. But for the abstract concept of a novel, these differences are unimportant; the abstraction is formed in recognition of the similarities of this class of literature.

An important aspect of abstraction is the use of a representation to symbolize the concept. In the example above, the word "novel" is the representation of the abstract concept. A representation provides a way to manipulate and use the abstraction, but a representation cannot affect the situations, objects, or processes embodied by the abstraction. For instance, the word "novel" can be manipulated in many ways, but no manipulation can change the concept denoted by the word.

Abstraction appears in programming languages in many forms. Some programming language abstractions are very similar to those used by mathematicians; for example, nearly every language permits the programmer to manipulate integers without having to know how they are handled by the computer itself. At a higher level of abstraction, some languages permit the programmer to manipulate objects of varying length, such as strings, as if the machine were composed of "elastic" words that changed size depending on their contents. Procedures are another example of an abstraction mechanism in programming languages. Procedures permit a frequently occurring or logically separate computational sequence to be replaced by an abbreviated notation. Indeed, the procedure has been called "one of the greatest software inventions" because it allows the realization of many useful abstractions (Dijkstra 1972).

The activity of programming can be viewed as the process of transforming an abstract solution of a problem into a concrete solution. A programming language is the tool with which the programmer performs this transformation. Thus the design of a programming language can be viewed as the design of a cohesive set of features that assist the programmer in formulating concrete realizations for abstractions.

Programming Language Design

The characterization of programming language design given above is rather broad; most languages are designed with more specific goals. For example, the well-known programming languages Fortran and Algol 60 were designed primarily for numerical applications. LISP, on the other hand, was motivated primarily by its intended application to symbolic data processing. These are examples of languages that were designed for specific problem areas in which the use of various abstractions was already established. As such, they contain facilities for realizing a limited number of abstractions. Provided that the abstractions used in solving a particular problem are within the domain of applicability of the language, programming the solution to the problem is often a straightforward transformation.

There are more instances, however, in which the solution to a problem requires the use of abstractions beyond the domain of the chosen language. In this case, the programmer is forced to distort or overspecify the abstract solution in the process of transforming it into a concrete program.

In response to this problem, a substantial amount of research and development has been conducted in the search for better programming languages. Most of this work, which has been performed since the introduction of Fortran and Algol 60, either introduced new languages or proposed generalizations or extensions to existing ones. The number of new languages that have been introduced is too large to permit an effective presentation here; a summary of many of the new languages up to 1969 is given in Sammet (1969).

Among the extensions to established languages, the most notable are proposals for various extensions to Algol. For example, Wirth and Hoare (1966) sought to enlarge the range of applications for which Algol was suitable by proposing better data-structuring facilities. One of the motivations for their extensions was an opinion that "programs expressed in the language should correspond closely with the dynamic processes they describe." In other words, programs should correspond closely with the abstractions used to formulate them.

Galler and Perlis (1967) proposed a definition facility for Algol that would permit the definition of new datatypes and operators in the language. Their work is often cited as one of the early contributions to so-called extensible languages (Wegbreit 1970).

Another approach to extending a programming language is the use of macros (McIlroy 1960; Cheatham 1966; Leavenworth 1966). This approach relies on the definition and subsequent substitution of textual material in order to extend a given language. An attractive feature of this approach is that the compiler for the language need not be modified to include the extensions, since the substitution text for a macro is written in the language that is being extended. This permits the use of a general-purpose macro processor, such as ML/I (Brown 1967), which is not connected in any way to the compiler for the language being extended. A disadvantage of macros is that the extension is static. Dynamic extension defined during the course of program execution is not possible.

The inadequacy of existing programming languages for some special application areas has resulted in the design and implementation of special-purpose languages. A good example of this class of languages are those designed for research in artificial intelligence (for example, Bobrow and Raphael 1974). Another example is illustrated by the development of "systems programming" languages such as Bliss (Wulf, Russell and Habermann 1971) and BCPL (Richards 1969).

More recent work has focused on extensible languages as a way to permit the programmer to make effective use of abstraction (Cheatham 1969; Galler 1974). An extensible language is one in which the programmer is able to modify the language itself to best suit the particular application. Thus, rather than supplying numerous built-in facilities for realizing abstractions, the programmer is able to define the facilities required for the particular application. Despite great expectations, extensible languages have not met with great success (Standish 1975).

Current research in programming language design seems to be following the trend set by the design and implementation of Pascal (Wirth 1971) and Simula 67 (Dahl and Nygaard 1966; Dahl, Myhrhaug and Nygaard 1968; Dahl and Hoare 1972). This trend is toward block-structured languages that include strong typing, static scope rules, and stress efficiency. These characteristics are held to promote proper programming practices, and, in view of the overall increase in software costs, result in more reliable programs (Wirth 1974; Hoare 1973).

Examples of various facilities in the more recent languages are given throughout the remaining chapters of this dissertation.

A different approach to language design is exemplified by languages such as SNOBOL4 (Griswold, Poage and Polonsky 1971). SNOBOL4 is a general-purpose language oriented toward string and list processing applications. SNOBOL4 contains several features that are not commonly found in most programming languages, such as a pattern-matching facility, dynamic array creation, an associative store facility, and a mechanism for defining additional datatypes.

In many respects, the development of SNOBOL4 has proceeded orthogonally to the main trends in programming language design. For instance, SNOBOL4 has no declarations; it is a "typeless" language in which a variable may have a value of any datatype at any time during execution. There is no block structure in SNOBOL4; dynamic scope (Dijkstra 1967) is used to determine the meaning of identifiers.

In spite of the "unstructured" appearance of some SNOBOL4 programs, SNOBOL4 is a powerful language that enjoys substantial and continued popularity. SNOBOL4 has been implemented on nearly all major large-scale machines. The most important reason for its popularity is that programs written in SNOBOL4 tend to be short and, above all, tend to work. Indeed, SNOBOL4 programs for many string and list processing applications are often several times shorter than programs written in other languages.

The design of SNOBOL4 was partly motivated by a desire for flexibility (Griswold 1972). As a result, much of the detail required for programs written in other languages is not required for equivalent SNOBOL4 programs. Viewed another way, SNOBOL4 provides some unusually convenient high-level facilities for abstraction. The programmer can construct a program that mirrors the abstract formulation of a problem, without having to be concerned about tedious details. The most obvious examples of such facilities are the pattern-matching and data structure facilities.

Overview of This Research

The research described in this dissertation concerns the design and implementation of advanced programming language features to facilitate the effective use of abstraction. The proposed facilities are included as a part of an experimental programming language called SL5.

The general philosophy behind the research reported here is very similar to the philosophy behind SNOBOL4, which is substantially different from current philosophical trends in programming language design. SL5 and its facilities are designed with the philosophy that new concepts in programming languages should be first developed and used without the inhibition of efficiency considerations. This is unlike the philosophy of language design used in other recent languages such as Pascal. In Pascal, for example, the predominant philosophy stresses the concept of transparency (Wirth 1974, 1975). This concept leads to the design of programming language features that can be explained in terms of their implementation. The stated advantage of this approach is that the programmer can estimate the computational effort required for the use of the features. Other aspects, such as security, verification, and concern for efficiency, lead to languages with features like strong typing, static scope rules, and a limited number of built-in datatypes.

Although a philosophy similar to that used in Pascal may result in a language in which reliable programs may be written (Wirth 1975), it does not seem appropriate for the investigation of new programming language facilities. SNOBOL4, for example, owes much of its success to its flexibility. In keeping with this spirit, the experimental programming language facilities of SL5 are designed to permit the programmer to work more in terms of abstractions than in terms of implementation.

The facilities of SL5, which were designed in this frame of reference, tend to be higher-level than those found in other languages. One way to measure the “height” of a language is in terms of the “distance” between the realization of various abstractions and their implementation on a particular machine (Wirth 1974). Using this criterion, SL5 is a language of substantially higher level than other recent languages. As a simple example, SL5 includes varying-length strings as a built-in datatype, whereas in Pascal strings must be implemented by the programmer in terms of arrays of characters.

Finally, another motivation for insisting on flexibility is that there exists a complex symbiotic relationship between a programming language and the problems for which it is used. The facilities of a programming language may help to formulate a solution to a problem by providing semantics that are useful for its conceptualization. In recognition of this relationship and in hopes of making its affect beneficial, the SL5 facilities have been designed with a concerted effort to minimize the restraining effect of implementation, and to maximize their usefulness for abstraction.

The concept of flexibility does not, however, imply simply implementing a myriad of features that might possibly prove to be useful. Instead, the goal of this work was to design a basic linguistic mechanism of sufficient flexibility so that the programmer can use it to realize many different abstractions. As such, the facilities described here are designed using a new procedure mechanism as their basis. This approach is motivated by the flexibility of procedural mechanisms and by their success as abstraction tools.

The approach of this research is experimental; that is, all of the SL5 facilities described here have been implemented and used. The viability of the features is measured in terms of their usefulness for practical application, not by a theoretical criterion or implementation considerations. In short, the needs of the programmer were the most important factor in designing the SL5 facilities.

It should be emphasized that the goal of this research is not to devise better algorithms for solving problems, but instead to provide better linquistic mechanisms for expressing algorithms. It is hoped that better linguistic mechanisms will have a beneficial impact on the problem-solving process.

Organization of This Dissertation

The basic features of the SL5 language are described in Chapter 2. The SL5 procedure mechanism is explained in Chapter 3. Chapters 4 and 5 describe facilities that are a consequence of, and are based on, the procedure mechanism. Specifically, a data structure facility is described in Chapter 4, and a facility whereby the programmer may attach a procedural component to a variable is described in Chapter 5. Chapter 6 contains a brief overview of the implementation of SL5. An evaluation of the SL5 facilities and suggestions for further research are given in Chapter 7. A list of built-in operators and functions in SL5 is given in the Appendix.

2. THE SL5 PROGRAMMING LANGUAGE

SL5 is a programming language designed for programming-language research, mainly in the areas of advanced string and structure processing. To some extent, SL5 is a product of programming-language research using SNOBOL4 (Griswold et al. 1971). SNOBOL4 has been used as the vehicle for recent research in advanced programming language facilities (Druseikis and Doyle 1974; Doyle 1975; Griswold 1975a and 1975b; Hallyburton 1974; Hanson 1976c). Results indicated that in order to continue these lines of research, a better experimental vehicle was required. This is, in part, the motivation for the development of SL5.

In addition, it has long been felt that SNOBOL4 could benefit from some "modernization" (Griswold 1974). Consequently, SL5 provides not only a tool for programming-language research, but also provides a more modern language for solving complex string and list processing problems.

Structurally, SL5 is similar to Algol 68 (van Wijngaarden et al. 1976) and Bliss (Wulf et al. 1971). The syntax of SL5 is expression-oriented, and most of the "modern" iterative control structures are included. There are no type declarations in SL5; like SNOBOL4, a variable can have a value of any type at any time during program execution. As evidenced by this feature, the semantics of the language stress execution-time flexibility, which is motivated in part by its intended use as a research tool.

The remainder of this chapter describes the basic features of SL5 necessary for understanding this dissertation. An overview of SL5 is given in Griswold and Hanson (1976). A detailed explanation of the syntax and semantics are given in Hanson (1976b), and Griswold (1976a) contains a summary of the built-in operators and procedures.

Syntax

The syntax of SL5 is expression-oriented. An SL5 program is a single expression; a sequence of expressions can be grouped together using the ***begin*** … ***end*** or { … } constructs. Reserved words are used for control expressions, and may not be used for any other purpose.

There are three kinds of operators in SL5: infix, prefix, and suffix. Infix operators include the arithmetic operations such as addition, subtraction, and the usual arithmetic comparisons. Examples of prefix operators are negation (-) and affirmation (+). Suffix operators follow their operands, for example k+ increments the value of k by 1. A list of some of the operators is given in the Appendix; a complete list may be found in Griswold (1976a).

Infix operators may be composed of one or more operator symbols; prefix and suffix operators consist of only one symbol. The disambiguation of infix and prefix or suffix operators is governed by the conventions described in Hanson (1976b). Ambiguity may always be resolved using blanks, and this method is used throughout this dissertation. For example, in the expression a+b, the operator + is the infix +. In the expression a+ - b, the suffix operator + is applied to a, and the infix - is applied to that result and b.

Results

The execution of an SL5 expression returns a *result*. In contrast to the conventional meaning of the result of an expression, a result in SL5 is composed of two parts—a *value* and a *signal*. The notation {value,signal} is used to denote a result. To facilitate the description of the behavior of SL5 constructs, 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*) means "the value component of the result *r*."

Signals are nonnegative integers. The programmer may associate certain meanings with various signal values, but the SL5 control expressions and most built-in operators and functions are sensitive to only two kinds of signals—success and failure—where zero values indicate failure and positive values indicate success. Built-in operators and functions use the value 1 for success. The letters *s* and *f* are used in conjunction with the result notation for the signals 1 and 0, respectively.

For example, the expression 25 + 6 has the result {31,1}. *V*({31,1}) is 31 and *S*({31,1}) is 1. All expressions return a result, even though in some cases the *V* component is not used. In these cases, the *V* component of the result of the result is the null string, which is denoted "". The comparison operators are an example of this convention: the result of 100 = 102 is {"",*f*}, the result of 100 < 102 is {"",*s*}.

Dereferencing

The result of a simple variable reference is {variable,*s*} where a "variable" is the place where a "value" resides. The result of an expression consisting of only an identifier is of this form. For example, the result of the expression `x` is {location of the value of `x`,*s*}, which is written simply as {`x`,*s*}.

A result of this form is *dereferenced* when the value of the variable is fetched. Thus, if the value of `x` is 15, dereferencing the result {`x`,*s*} produces the result {15,*s*}. If the dereferencing operation is applied to a result in which *V* is already a value, the result is returned unchanged. As an example of this process, assume the value of `x` is 15 and the value of `y` is 6. The evaluation of the expression `x + y` can be illustrated by the following sequence of expressions containing results:

{`x`,*s*} + {`y`,*s*}
{15,*s*} + {6,*s*}
{15+6,*s*}
{21,*s*}

The execution of an expression without dereferencing is called *interpretation*. For example, the result of the interpretation of the simple expression `x` is {`x`,*s*}. In some cases, interpretation may be followed by dereferencing, in which case a result such as {`x`,*s*} becomes {15,*s*}. The combination of interpretation followed by derefencing is called *evaluation*. The terms "interpret" and "evaluate" are used below to describe the behavior of the SL5 control expressions, and indicate in which cases dereferencing is performed.

In general, interpretation is performed in situations where a variable may be desired. For example, the assignment operator, `:=`, interprets its left argument and evaluates its right argument. If both of these operations succeed, i.e., neither of the *S* components is *f*, and if the *V* component of the left argument is a variable, then the *V* component of the right argument is stored in the variable. The result of the assignment operation is the result of the right argument.

Transmission of Failure

Most built-in operators and functions transmit failure if any of their arguments fail. Usually, the result {`""`,*f*} is transmitted to indicate this condition. For example, consider the expression `x := ` e_1 `+` e_2. If the evaluation of e_1 or e_2 fails, the operator `+` transmits this failure by returning the result {`""`,*f*}. Since the operator `:=` performs the indicated assignment only if its right argument does not fail, the assignment is not performed and the result {`""`,*f*} is returned thereby transmitting the failure signal. The result of the entire expression is therefore {`""`,*f*}.

Composing a Result

The built-in operator `&` is used to compose a result. The expression e_v `&` e_s returns the result {$V(e_v)$,$V(e_s)$}. The expression e_v is interpreted and e_s is evaluated. The signal portions of their results are ignored.

Control Expressions

In SL5, the program structures that control the flow of execution are expressions, each of which returns a result. They are driven by signals, not by Boolean values as in most programming languages, although in many cases the use of a signal is equivalent to the use of a Boolean value. All of the control expressions treat any nonzero signal as success.

The following list describes some of the more commonly used control expressions; a complete list is given in Hanson (1976b).

Expression Sequencing

The ***begin*** … ***end*** and `{` … `}` constructs are equivalent and are enclose a sequence of expressions. The syntax is

begin e_1`;` e_2`;` …`;` e_n ***end***
`{` e_1`;` e_2`;` …`;` e_n `}`

Each expression in the sequence is interpreted, and the result of the entire expression is the result of e_n. For instance, the result of `{x := 5; y := -2; x + y}` is {3,*s*}. An SL5 program is a list of expressions, in which case the reserved words ***begin*** and ***end*** are not required.

Conditional and Selection Expressions

In the expression ***if*** e_1 ***then*** e_2 ***else*** e_3, e_1 is evaluated first. If it succeeds, e_2 is interpreted and its result is returned as the result of the ***if*** expression. If the evaluation of e_1 fails, the result of interpreting e_3 is returned. The ***else*** clause may be omitted. In nested ***if*** expressions an ***else*** is associated with the nearest preceding ***then***.

The expression e_1 ***or*** e_2 returns the result of interpreting e_1, if that interpretation succeeds. Otherwise the result of e_2 is returned.

The expression e_1 ***and*** e_2 is the complement of the ***or*** expression. If the interpretation of e_1 fails, i.e.,the *S* component of its result is 0, that result is returned as the result of the ***and*** expression. If the interpretation of e_1 succeeds, the result of the ***and*** expression is the result of interpreting e_2.

The ***and*** expression succeeds only if the interpretation of both of its arguments succeeds, and the ***or*** expression succeeds if the interpretation of either of its arguments succeeds. In both expressions, if the interpretation of e_1 satisfies the given criterion, e_2 is not interpreted.

The ***and*** construct can be used to interpret a sequence of expression until one fails. The result of the expression

e_1 ***and*** e_2 ***and*** e_3 … ***and*** e_n

is the result of the first expression whose interpretation fails, or the result of e_n if none of e_1 through e_{n-1} fail.

The syntax of the ***case*** expression is

```
case e of
      l_1 : e_1;
      l_2 : e_2;
      …
      l_n : e_n;
      default: e_d
end
```

where l_i denotes a list of one or more literals separated by commas, and e_i is an arbitrary expression. The elements of the literal lists may be integers, reals, or strings. The expression *e* is evaluated and the *V* component of its result is compared with each literal l_i in order to select the appropriate expression e_i. Failure of *e* is treated as a programming error. If a matching literal is found, the associated expression is interpreted and its result is returned. If no match is found, the result of interpreting e_d is returned. The ***default*** label may be omitted, in which case the result {`""`,*f* } is returned if a matching literal is not found. An example of a ***case*** expression is

```
age := (case name of
            "sam": 23;
            "mary": 42;
            "chris": 7;
            "edward": "deceased";
            default: "unknown";
        end)
```

Iterative Expressions

The expression ***while*** e_1 ***do*** e_2 repeatedly interprets e_2 as long as the evaluation of e_1 does not fail. The result is the result of the last interpretation of e_2. If e_2 is never interpreted, the result is {`""`,*s*}.

The expression ***repeat*** *e* repeatedly interprets *e* until *e* fails. The result of the ***repeat*** expression is the result of the last interpretation of *e*, which necessarily has an *S* component containing *f*.

The syntax of the ***for*** expression is

for e_1 ***from*** e_2 ***to*** e_3 ***by*** e_4 ***while*** e_5 ***do*** e_6

The phrases ***from*** e_2, ***to*** e_3, ***by*** e_4, and ***while*** e_5 are optional. If the ***by*** phrase is omitted, ***by*** 1 is assumed. The expression e_1 is interpreted, and the *V* component of its result must be a variable, which is referred to as the control variable. The expressions e_2, e_3, and e_4 are evaluated, and the *V* components of their results are used to control the loop in the conventional manner. Execution of the loop consists of repeatedly interpreting e_6 as long as the evaluation of e_5 does not fail, and as long as the value of the control variable satisfies the conditions given by e_2 and e_3. The result of the ***for*** expression is the result of the last interpretation of e_6, or {`""`,*s*} if conditions are such that e_6 is never interpreted.

As an example, the expression

```
for i from 3 to 100 by 2 while abs(term) > 0.001 do {
    term := -term*x*x/(i*(i - 1));
    sin := sin + term;
}
```

is used in a procedure that computes sin(x) using the Maclaurin series expansion.

For convenience, there are several other complementary control expressions. See Hanson (1976b) for a complete list.

Primitive Datatypes

The primitive datatypes of SL5 are

integer
character
real
string
file
list
procedure
environment

Integers and reals are the usual scalar types found in most programming languages. A character differs from a string of length 1 in that its numeric code can be obtained. A string in SL5, as in SNOBOL4, can be used as a scalar type. That is, an entire string can be manipulated as a single entity; it is not an array of characters. There are, however, built-in functions that permit the programmer to access the individual characters of a string.

An object of datatype file is used for input and output. For the most part, files are used as arguments to built-in functions for reading and writing strings or characters.

Lists permit the programmer to manipulate an aggregate of values as a single entity. A list is specified by enclosing its elements within square brackets, for example,

```
x := ["string", a + b, 25]
```

assigns to x a list of three elements. The elements of a list may be of any datatype, including list.

The elements of a list are accessed using the built-in infix operator `!`. For example, the expression `x!2` refers to the second element in the list assigned to `x`. The indices that may successfully be applied to a list must be between 1 and *n*, the number of elements in the list. If the index is outside this range, the result of the `!` operator is {`""`,*f* }. For example, if `x` has been assigned the list given above, the expression `x!k :=` 30 changes the `k`th element to 30 provided `k` is between 1 and 3, inclusive.

An alternative method of building a list is the built-in function `list(n,x)`, which returns a list of size `n` whose elements are initialized to the value of `x`.

The number of elements in a list `x` is returned by the built-in function `size(x)`.

The datatypes procedure and environment are described in subsequent chapters, and play a central role in the programming language features described in this dissertation.

Conversions

Implicit conversions. Most built-in operators and functions attempt to convert their arguments to the expected type. Consequently, the programmer usually does not have to be concerned about type conversion. For example, in the expression `x + y`, the values of `x` and `y` are converted to a numeric type—integer or real—prior to the addition. The failure of an implicit type conversion is treated as an error by most built-in functions.

Explicit conversions. Built-in functions are provided so that the programmer can explicitly convert a value from one datatype to another. Each of these functions takes a single argument and attempts to convert it to the indicated type. If the conversion is successful, the converted value is returned, otherwise the result {`""`,*f* } is returned. The names of these functions are the same as the datatype to which the argument is converted, for example `integer(x)` converts `x` to an integer.

Built-In Functions

SL5 has a substantial repertoire of built-in operators and functions, a list of which is contained in Griswold (1976a). A list of some of the built-in functions, sufficient for understanding the programming examples given in this dissertation, is given in the Appendix.

Programming Examples

The following examples illustrate the use of some of the SL5 features described in the preceding sections. The examples also show the programming style and conventions used throughout this work. In all programs that perform input or output, the value of the variable `infile` is assumed to be the “standard” input file, for example a terminal or card reader. Likewise, the value of `outfile` is assumed to be the “standard” output file, which might be a printer or a terminal. The “standard” error file, which is usually the same as the output file, is assumed to be the value of `errfile`.

The following program counts the number of lines in the standard input file.

```
nlines := 0;
while line := readline(infile) do
    nlines := nlines + 1;
writeline(outfile, nlines || " lines");
```

(The operator `||` denotes string concatenation.) This program can be simplified by using the suffix operator `+`, and by omitting the assignment to `line`, which amounts to using only the S component of the result returned by `readline`:

```
nlines := 0;
while readline(infile) do nlines+;
writeline(outfile, nlines || " lines");
```

The next program also counts lines in a file, but only those containing the string given in the variable `str`. Each line containing the specified string is also printed.

```
nlines := 0;
strlen := length(str);
while line := readline(infile) do {
    match := 0;
    for i from 0 to length(line) - strlen do
        while match = 0 do
            if str == section(line, i, strlen) then {
                writeline(outfile, line);
                nlines+; match+;
            };
};
writeline(outfile,
    (if nlines #> 0 then nlines else "no") ||
    " lines matched");
```

This program illustrates the use of the ***for*** expresssion and the built-in function `section` (see the Appendix).

3. THE SL5 PROCEDURE MECHANISM

Almost all programming languages provide some sort of procedure facility as an integral part of the language. The procedure is a very effective abstraction tool because it permits the programmer to define operations for a specific program that can be used as if they were a part of the language. More importantly, the programmer can use a procedure knowing only “what” is performed by the procedure; knowledge of “how” its particular operation is done is not required for its use.

The procedure mechanism in most languages is based on the concept of the recursive function. This model is well-suited to many abstractions that programmers often make, and accounts for its success as a tool for abstraction. There is another procedural mechanism, the coroutine (for example, Dahl and Hoare 1972; Knuth 1973, Section 1.4.2; Pratt 1975, Section 5-4), that is more useful or convenient in many programming situations. For instance, there are situations in which the relationship between two functions is not that of a master and a subordinate, which is imposed by a procedure mechanism based on the recursive function. Coroutines permit the programmer to write a set of “cooperating” procedures that operate essentially on the same level. Multiple-pass algorithms, such as those used in compilers, are frequently cited examples in which the use of coroutines results in a program that is easier to write and to understand (Knuth 1973). Coroutines have also been used for goal-oriented processes such as string pattern matching (Druseikis and Doyle 1974).

A coroutine differs from the usual notion of a function in that the execution of a function is always initiated at its beginning, while the execution of a coroutine is initiated at the location immediately following where it last terminated. When a coroutine is first invoked, execution starts at its beginning. While the local variables of a function are not saved from one activation to the next (except in the case of ***own*** variables in Algol), the local variables of a coroutine retain their values. Because of this property, the activation record for a coroutine exists after its termination. This feature permits a coroutine to “remember” information. The activation record for a function, on the other hand, is usually inaccessible after its termination.

Although a coroutine mechanism can be simulated to some degree by recursive functions, to do so the programmer must think and program in terms of recursive functions, and often must use overly elaborate or unstructured schemes to achieve the desired effects. A more general procedure mechanism is required for applications that are outside the domain of procedures based on recursive functions. For example, the implementation of many input and output problems and discrete simulation problems can often be better understood in terms of coroutines.

SL5 provides a procedure mechanism that is a generalization of functions in which ordinary recursive function use is a special case. SL5 procedures can be used as recursive functions or as coroutines. Unlike procedures in most languages, SL5 procedures are data objects, and are capable of being manipulated and transmitted throughout the program like other data objects. Procedure invocation may be decomposed into separate operations, each of which is available at the source-language level.

This procedure mechanism is motivated by the desire to generalize the usefulness of procedural abstractions. Indeed, the programming language facilities described in the following chapters are a direct consequence of the features of the SL5 procedure mechanism, and provide evidence of its viability.

This chapter describes the main features of procedures in SL5 and how to use them. Included are explanations of procedure construction, procedure invocation, and the extent of identifiers that appear in procedures. The last section compares the SL5 procedure mechanism with that of other languages.

Procedure Construction

A procedure is constructed and returned by the ***procedure*** expression. The syntax is

procedure (a_1, a_2, …, a_n) *<declarations>*
    <body>
end

where a_1 through a_n are the formal argument identifiers, *<declarations>* are of the form described below, and *<body>* is a sequence of expressions separated by semicolons. For example, the expression

```
sine := procedure (x)
    sin := term := x;
    for i from 3 to 100 by 2
```

```
            while abs(term) > 0.001 do {
                term := -term*x*x/(i*(i - 1));
                sin := sin + term;
            };
        succeed sin
    end
```

assigns to `sine` a procedure that computes *sin*(*x*) using the Maclaurin series expansion. The procedure `abs`, that is, the procedure that is the value of the variable `abs`, can be constructed by the expression

```
    abs := procedure (x)
        succeed (if x >= 0 then x else -x)
    end
```

The obvious advantage of procedures as data objects is that they can be manipulated at execution time, and can appear in arbitrary expressions. For example, the following expression assigns to `f` a different procedure depending on the values of `x` and `y`.

```
    f := if x > y then
        procedure (a, b) … end
    else
        procedure (a, b) … end
```

In addition, several variables can have the same procedure as value, for example, if

```
    f := procedure () … end
```

then the expression `g := h := f` assigns to `g` and `h` the same procedure as `f`.

Procedure Invocation

Procedure invocation, which is an atomic operation in most programming languages, is decomposed into three distinct components in SL5. These components are available to the programmer at the source-language level, and are what permit SL5 procedures to be used as coroutines as well as recursive functions.

The three components of procedure invocation are

create—creation of an environment (activation record) for the procedure;

with—binding of the actual arguments to the environment;

resume—resumption of the execution of the procedure associated with the environment.

Environment creation. An environment for the execution of a procedure is created by an expression such as *e* `:=` ***create*** *f*, which creates an environment for the procedure that is the value of *f* and assigns this environment to *e*. An environment is essentially an activation record for the procedure. An environment contains space for the identifiers appearing in the procedure, parameters, and several system items that are used during the activation and deactivation of the environment. The treatment of environments as data objects is the important difference between SL5 environments and activation records in other languages (for example, Pratt 1975, Section 5-4). The specific contents of an environment are described in the next section.

Argument binding. The ***with*** expression binds the actual arguments to the formal arguments in an environment for a procedure. An expression of the form

e ***with*** $(e_1, e_2, \ldots, e_n)$

evaluates each of the argument expressions, from e_1 to e_n, and assigns the *V* component of the results to the corresponding formal arguments of the procedure for which *e* is an environment. The arguments are transmitted “by value”. If the number of actual arguments is less than the number of formal arguments, null strings are supplied for the omitted arguments. Excess arguments are ignored. The result of the ***with*** expression is $\{e,s\}$, provided none of the actual argument expressions fails during evaluation. If the evaluation of e_i fails, the ***with*** expression returns the result {`""`,*f* }. In this event, the subsequent arguments are not evaluated or transmitted.

Resumption of execution. The actual execution of a procedure is activated by the expression ***resume*** *e*, which causes the execution of the current procedure to be suspended, and the execution of the procedure for which *e* is an environment to resume. The execution of the current environment is suspended within the ***resume*** expression. When the environment is subsequently reactivated, the ***resume*** expression returns its result.

The general form of the ***resume*** expression is ***resume*** (*e*, e_r), which resumes the execution of *e*, and transmits the result of interpreting e_r. This result becomes the result of the expression in *e* that caused its prior suspension, for example a ***resume*** expression. For example, the sequence

```
sinpi := create sine with 3.1416;
x := resume sinpi;
```

assigns to `x` the sine of pi.

The ***resume*** expression requires an explicit indication of the environment to be resumed. In contrast, the expression ***return*** e_r suspends execution of the current environment and returns the result of interpreting e_r to the environment that last *resumed* the current environment. Like ***resume***, ***return*** causes the execution of the current environment to be suspended inside the ***return*** expression so that upon subsequent resumption, the result transmitted becomes the result of the ***return*** expression.

In both the ***resume*** and the ***return*** expressions, the result {`""`,*s*} is assumed if e_r is omitted.

Since the success and failure signals are used so often in SL5, the expressions ***succeed*** e_r and ***fail*** e_r are provided and are equivalent, respectively, to the expressions ***return*** *e*`&1` and ***return*** *e*`&0`.

Functional Notation

The decomposition of procedure invocation and the explicit appearance of environments at the source-language level is necessary and convenient for many coroutine programming situations. There are, of course, situations in which the use of an SL5 procedure as a recursive function is adequate or more suitable. The functional notation

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

may be used in these cases, and is equivalent to the expression

resume (***create*** *f* ***with*** (e_1, e_2, …, e_n))

The Contents of Environments

In most programming languages, the programmer cannot explicitly access the activation record for a procedure. In SL5, however, activation records are data objects that are accessible at the source-language level as environments. Some knowledge of their contents is required to fully understand the operation of the ***create***, ***with***, ***resume***, and ***return*** expressions.

An environment for a procedure contains the storage for all the identifiers that appear in the procedure. An identifier name appearing in a procedure simply denotes a location in an environment where its value is stored. Hence the identifiers are more appropriately thought of as "belonging to" the environment rather than to the procedure.

An environment also contains the information necessary for the execution of its associated procedure. There are three components of an environment, in addition to the storage for the identifiers that appear in the procedure, that are implicitly accessed during the execution of the ***create***, ***with***, ***resume***, and ***return*** expressions.

Creator. An environment whose execution caused the creation of an environment is called the *creator* for that environment. Each environment contains information designating its creator, which is established during the execution of the ***create*** expression. Since an environment is only created once, the creator remains constant throughout program execution.

Resumer. An environment's *resumer* is the environment that caused its most recent resumption via the ***resume*** expression. Notice that, in contrast to the creator, the resumer of an environment may change during the course of program execution. Prior to the initial resumption of an environment, the resumer is the same as the creator.

Continuation point. Whenever the execution of an environment is suspended, by either a ***resume*** or a ***return*** expression, the point of execution in the procedure body is stored as the continuation point for the environment. The continuation point may be thought of as the "location counter" for the procedure. This information is used when the suspended environment is subsequently reactivated in order to restore the state of the execution of that particular environment for the procedure.

Details concerning the implementation of environments are described in Chapter 6.

The Difference Between return and resume

The definitions given in the previous section provide the terminology necessary to describe the important difference between ***resume*** and ***return***: ***return*** does not establish a new resumer for the environment to which control is being returned. Only a ***resume*** expression, which explicitly indicates the environment to be resumed, establishes the resumer.

Accessing the Attributes of Environments

The identifiers in the environment constitute a set of programmer-defined *attributes* of the environment. They may be accessed using the *attribute reference* operator, which is indicated by a dot. The expression *e*`.`*x* refers to the identifier *x* in the environment *e*. The left argument of the operator may be an arbitrary expression including an attribute reference. The attribute reference operator associates to the left so that `a.b.c` is equivalent to `(a.b).c`.

The Life Span of Environments

The life span of an SL5 environment is completely determined by its accessibility at the source-language level. This is a consequence of defining an environment as an SL5 data object. There is no correspondence between the activation of a procedure and the life span of its environment; an environment is not destroyed upon the execution of a ***return*** expression. This is in marked contrast to most other programming languages in which the procedure mechanism is based on the recursive function and an activation record cannot outlive the expression that caused its creation.

When SL5 procedures are invoked using functional notation, the environment is inaccessible to the programmer. In this special case, its life span is equal to the life span of the invoking expression.

The Interchangeability of Procedures and Environments

Most of the procedure activation expressions described in the preceding sections accept a procedure as an argument wherever an environment is required and vice versa. If a procedure is used where an environment is expected, an environment for the given procedure is created. If, on the other hand, an environment is given in place of a procedure, that environment is used directly or the procedure associated with the environment is used. More specifically, under these conditions the procedure activation expressions behave as follows.

create *f*: If *f* is an environment rather than a procedure, a new environment for the associated procedure is created and returned.

e ***with*** $(e_1, e_2, \ldots, e_n)$: If *e* is a procedure rather than an environment, an environment for the procedure is created, and the arguments e_1 through e_n are bound to that environment.

resume *e*: If *e* is a procedure rather than an environment, an environment for the procedure is created and resumed. Notice that in this case there are no arguments bound to the environment prior to its initial resumption. The values of the formal arguments are initialized to null strings in this case.

Functional notation: If, in the expression

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

f is an environment as opposed to a procedure, the effect is to omit the environment creation part of the three-expression sequence given above. Thus, in this case, functional notation is equivalent to

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

Although this last convention is somewhat different from what might be expected given the conventions chosen for ***create*** and ***with***, it has proven more useful in its present form. Indeed, the behavior of all the activation expressions when presented with a procedure rather than an environment, or vice versa, is based on practical considerations motivated by experimental use.

The Extent of Identifiers

Declarations in SL5 procedures are used to determine the *extent* to which an identifier is known throughout the program. This is sometimes referred as the scope of an identifier in other programming languages. The term scope, however, is most often associated with the concept of static scope rules. As described below, the conventions in SL5 are more like the dynamic scope (Dijkstra 1967) used in SNOBOL4 and LISP. Since there are subtle differences between this kind of dynamic scope and the conventions used in SL5, the term extent is used in SL5 to avoid confusion.

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

> **`public`** id_1, id_2, …, id_n
> **`private`** id_1, id_2, …, id_n

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

Private Identifiers

The extent of private identifiers is restricted to the procedure in which they are 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.

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 extent 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 notation $e_p \rightarrow e_q$ denotes that the environment e_p is the creator of e_q. The environment e_q is called the *direct descendant* of e_p, and e_p is called the *direct ancestor* of e_q. Note that for any environment e_q there is one and only one environment e_p such that $e_p \rightarrow e_q$. This relationship is established at the time of the creation of e_q. On the other hand, there may be many environments e_q such that $e_p \rightarrow e_q$, i.e., an environment can be the creator for an arbitrary number of environments.

If, for e_p and e_q, there exist environments $e_1, e_2, \ldots, e_i$ for $i \geq 0$ such that

$$e_p \rightarrow e_1 \rightarrow e_2 \rightarrow \ \ldots \rightarrow e_i \rightarrow e_q$$

then e_q is a *distant descendant* of e_p, and e_p is a *distant ancestor* of e_q. This relationship is indicated by the notation $e_p \rightarrow^+ e_q$. For any environment e_q, there may be many environments e_p such that $e_p \rightarrow^+ e_q$.

These relations impose a tree structure on the set of all environments in an SL5 program. Since the relation $e_p \rightarrow e_q$ is established at environment creation time, this tree is referred to as the *creation history tree*. Figure 1 illustrates an example of a creation history tree. The arrows in Figure 1 denote that $e_p \rightarrow e_q$; for example, $e_1 \rightarrow e_7$.

If $e_p \rightarrow^+ e_q$ holds, then e_p and e_q are called *serial* environments. For example, in Figure 1, e_2 and e_4 are serial environments as are e_2 and e_6, e_7 and e_8, and so on. Note that e_4 and e_5, for example, are not serial environments because neither $e_4 \rightarrow^+ e_5$ or $e_5 \rightarrow^+ e_4$ hold. Environments that exhibit this property are called *parallel* environments: Two environments e_p and e_q are parallel environments if neither $e_p \rightarrow^+ e_q$ or $e_q \rightarrow^+ e_p$ hold.

An environment for a procedure in which public identifiers are declared is said to be the *custodian* of those identifiers. The value of a public identifier is available to its custodian, e_c, and to all e such that $e_c \rightarrow^+ e$. The extent of nonlocal identifiers that appear in a procedure is determined when an environment for the procedure is created, and is obtained by examining the history of serial environments. For each nonlocal identifier in a newly created environment e, a search is performed along ancestral lines for the first environment e_c such that $e_c \rightarrow^+ e$ and e_c is a custodian for the nonlocal identifier. In other words, the search is performed by examining each successive creator until the custodian of the nonlocal identifier is found. If the search is successful, the nonlocal identifier in e henceforth refers to the public identifier located in its custodian, e_c.

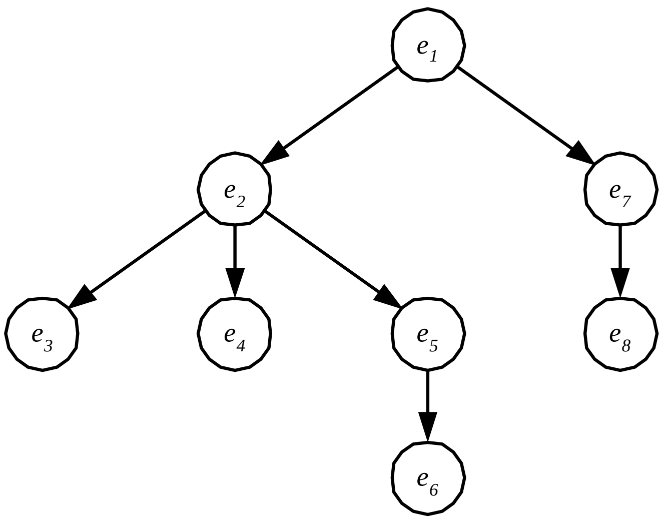

Figure 1. A Creation History Tree.

This search of the creation history tree may fail to find a custodian for a nonlocal identifier, *id*. In this case, an "implicit" public declaration is assumed for the identifier. The environment *e*, whose creation resulted in the unsuccessful search, becomes a custodian for *id* as if **`public`** *id* had appeared in the procedure.

An environment for a "main procedure" is the root of the creation history tree in which, conceptually, all built-in identifiers are declared public, and are initialized to their predefined values before program execution begins. The root environment contains, for example, identifiers that have built-in functions as their initial values. This is necessary so that, unless explicitly declared otherwise, when these identifiers are used as nonlocal identifiers in programmer-defined procedures, they will refer to the built-in identifier in the root environment.

The convention for determining the extent of nonlocal identifiers provides a means for inter-procedure communication between serial and parallel environments. Two or more environments can communicate by means of a nonlocal identifier whose custodian is their common ancestor. For example, referring to Figure 1, communication between serial environments e_5 and e_6 can be accomplished by using a nonlocal identifier `x` in the procedures associated with e_5 and e_6 that is declared public in e_2. Since $e_2 \rightarrow^+ e_5$ and $e_2 \rightarrow^+ e_6$, references to `x` in e_5 and e_6 refer to the `x` in e_2. Communication between e_5 and e_6 can also be accomplished by using a nonlocal identifier `y` in e_6 that is declared public in e_5. Since $e_5 \rightarrow^+ e_6$, and since e_5 is a custodian for `y`, references to `y` in e_6 refer to the `y` in e_5.

Communication between parallel environments, such as e_2 and e_7, can be accomplished by references to a nonlocal identifier declared public in a common ancestor such as e_1. Note that unlike serial environments, parallel environments cannot communicate via a public identifier in one of them, since for two parallel environments, neither is the ancestor of the other. For example, neither $e_2 \rightarrow^+ e_7$ or $e_7 \rightarrow^+ e_2$ hold.

The Resumption Graph

The relationships among environments described in the preceding section are realized by the creator fields of all of the environments in an actual SL5 program. The resumer fields of all of the environments provide another relationship among environments.

Analogous to the notation used above, the notation $e_p \Rightarrow e_q$ indicates that e_p is the resumer of e_q. The environment e_q is called the *direct successor* of e_p, and e_p is called the *direct predecessor* of e_q. Unlike the descendant relation, the successor relation between two environments changes during the course of program execution, and it is possible for both $e_p \Rightarrow e_q$ and $e_q \Rightarrow e_p$ to hold simultaneously.

Likewise, if for e_p and e_q, there exist environments $e_1, e_2, \ldots, e_i$ for $i \geq 0$ such that

$$e_p \Rightarrow e_1 \Rightarrow e_2 \Rightarrow \ldots \Rightarrow e_i \Rightarrow e_q$$

then e_q is said to be a *distant successor* of e_p, and e_p is a *distant predecessor* of e_q. This relationship is indicated by the notation $e_p \Rightarrow^+ e_q$. Note that both $e_p \Rightarrow^+ e_q$ and $e_q \Rightarrow^+ e_p$ can hold simultaneously.

The successor relationships impose a graph structure on the environments in an SL5 program that is called the *resumption graph*. Unlike the creation history tree, this structure may be a general graph since both $e_p \Rightarrow e_q$ and $e_q \Rightarrow e_p$ may hold simultaneously.

The descendant and successor relationships provide a concise way to describe the behavior of the ***return*** and ***resume*** expressions. Essentially, the ***resume*** expression modifies the resumption graph by changing the successor relation, whereas the ***return*** expression simply uses the resumption graph to determine the environment to which control should be returned. The ***resume*** expression modifies the resumption graph.

Examples of Procedures

The procedures sine and abs given at the beginning of this chapter illustrate SL5 procedures that are written to be used as recursive functions. The examples given in this section illustrate how SL5 procedures may be used in a coroutine fashion.

A Label Generator

A common use for a coroutine is to generate the next element of a sequence each time it is resumed. For example, the following procedure can be used to create label generators.

```
   genlabel := procedure (l, n)
      repeat {
          succeed l || lpad(n, 4, "0");
          n+;
      }
  end
```

(The built-in function lpad is described in the Appendix.) An environment for `genlabel` is created by an expression such as

```
   nextlab := create genlabel with ("X", 10)
```

which assigns to `nextlab` an environment for genlabel that generates the sequence of labels `X0010`, `X0011`, etc. The next label is returned each time `nextlab` is resumed, i.e., in expressions such as `x :=` ***resume*** `nextlab`. The sequence can be restarted or changed by retransmitting the arguments:

```
   nextlab := nextlab with ("L", 100)
```

Note that it is the environment for genlabel that is the generator, not the procedure itself. Thus the procedure may be thought of as a template from which to create label generators.

Finding the Next Line

A portion of the program that counts the number of lines containing a specified string, given at the end of Chapter 2, can be reorganized as a procedure that returns the next line in a file `f` containing the string given in `str`:

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

```
        };
        fail
    end
```

An environment for find returns the next line in `f` that contains `str` each time it is resumed, and fails when the end of file has been reached. Using this procedure, the program given in Chapter 2 can be rewritten as follows.

```
    nextline := create find with (str, infile)
    nlines := 0;
    while writeline(outfile, resume nextline) do nlines+;
    writeline(outfile,
        (if nlines #> 0 then nlines else "no") ||
        " lines matched");
```

Reading Words

As illustrated in the preceding example, the use of a procedure as a coroutine permits the procedure to “remember” its current “state”. The following procedure uses this feature to separate lines of text into words. Each resumption of an environment for the procedure returns the next word from the indicated file.

```
    getword := procedure (f) private line, let, dig, both;
        let := "abcdefghijklmnopqrstuvwxyz" ||
               "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        dig := "0123456789";
        both := let || dig;
        while line := readline(f) do
            while line := past(line, let) do {
                succeed thru(line, both);
                line := after(line, both);
            };
        repeat
            fail;
    end
```

Even after the end of file has been reached, subsequent resumptions of an environment for getword continue to succeed until all the words have been returned. After the last word is returned, subsequent resumptions fail. The built-in functions `after`, `past`, and `thru` are described in the Appendix.

Comparison with Other Procedure Mechanisms

The SL5 procedure mechanism is a substantial departure from the usual notion of a procedure. There are several features in the SL5 mechanism that do not have counterparts in other programming languages. The following sections describe some of the differences and similarities between the SL5 mechanism and the procedure mechanisms of other languages.

Procedures and Environments as Data Objects

Although the idea of treating procedures and environments as data objects is basically simple, this feature is one of the most obvious and significant differences between SL5 and most languages. In most languages, procedures are viewed as something entirely different from data. Notable exceptions are LISP and EL1 (Wegbreit 1971). To a limited degree, a procedure is treated as a data object by some “systems implementation languages”, such as BCPL (Richards 1969), in which the address of a procedure is associated with the value of an identifier.

The Decomposition of Procedure Invocation

SL5 appears to be the first language in which procedure invocation can be decomposed at the source-language level. Simula 67 (Dahl and Nygaard 1966; Dahl et al. 1968), discussed below, does provide an operation similar to the

resume expression for use with the class facility, but does not permit a general decomposition of recursive function notation.

In most languages, values must be transmitted to a procedure as arguments or as global quantities. The ability to transmit a result to an instance of a procedure via the ***resume*** expression provides a sort of symmetry between the operations of resuming a procedure and returning from a procedure that are usually considered to be unrelated.

The Extent of Identifiers

When SL5 procedures are used only as recursive functions invoked via functional notation, the extent of public identifiers is equivalent to the dynamic scope of identifiers in SNOBOL4 and LISP. In this case, the creation history tree and the resumption graph are equivalent. In languages whose procedure mechanisms are based on the recursive function, only active serial environments exist, and their life span is the same as that of the expression that caused their invocation. It is this characteristic that permits the activation records (environments) to be allocated and deallocated using a stack. Parallel environments require a more general storage scheme due to their unpredicatable life spans. This is especially the case in SL5 where environments are data objects and their accessibility is under the control of the programmer. (A brief description of the scheme used in SL5 is given in Chapter 6.)

The concept of private identifiers is different from the concept of local identifiers used in most languages. Local identifiers declared in procedures in languages with static scope, such as Algol 60, achieve the same effect only if the procedure does not contain other procedure declarations, i.e., it is a “leaf” procedure.

Simula 67

The class mechanism of Simula 67 (Dahl and Nygaard 1966; Dahl et al. 1968) is, in some respects, similar to the SL5 procedure mechanism. In Simula, an instance of a block may outlive its calling statement. Block instances of this type are called classes.

Classes are used mainly for data structures, but can also be used as coroutines. The basic similarity between SL5 environments and Simula classes is that they are both data objects and may both be used as coroutines. The principal difference lies in the operations that govern the transfer of control among environments in SL5 and classes in Simula. In addition, there is no way in Simula to transmit values to a class when it is resumed, except through global variables. Finally, static scope rules are used in Simula to determine the meaning of nonlocal identifiers.

4. DATA STRUCTURES

The procedure mechanism of a programming language is one tool with which the programmer can realize various abstractions for a particular problem. For any nontrivial program, the programmer must also make use of various *data abstractions*. The programmer must take the abstract data that is associated with the specific problem, which might be matrices, sets, complex numbers, and so on, and produce concrete representations of those abstractions in the program.

To assist the programmer in this task, most high-level languages provide various forms of data structures, such as arrays, that can be used for this purpose. If the data abstractions used to formulate the problem closely parallel the data structures provided in the given language, the resulting program is usually easy to write and mirrors the abstractions in terms of the facilities of the language.

Unfortunately, the solutions to many problems require the use of data abstractions for which there are no corresponding facilities in the given language. For example, a frequently used abstraction is a symbol table of some kind. Few languages provide a built-in data structure “symbol table”, so the programmer must construct one using the available facilities. It is indeed unreasonable to expect a language to contain all of the various structures that might conceivably be used by a programmer. A few basic structures usually are provided upon which the programmer can build.

It is quite easy, however, to push the built-in facilities of most languages beyond their range of applicability. When this happens, the programmer becomes bogged down in the unnecessary and tedious detail of the representation of a data abstraction rather than concentrating on its use. For example, trying to implement a symbol table in Fortran can easily lead the inexperienced programmer into a situation in which the representation details are evident throughout the program. A good Fortran programmer, on the other hand, will recognize that the use of the symbol table is what is important, and will suppress its representation details by using subroutines or functions to access the table.

A programming language should provide a mechanism that permits the programmer to work in terms of the logical structure of a data abstraction rather than a physical structure imposed by an implementation. This is the type of mechanism that is missing in languages such as Fortran and Algol 60. Its omission in these langauges forces the programmer to cast all data abstractions in terms of arrays. Consequently, the realization of a complex data abstraction in Fortran usually bears little or no resemblance to its abstract structure.

Current Approaches to Data Abstractions

It has been known for some time that linguistic mechanisms for specifying the realization of a data abstraction are as important as the mechanisms for specifying procedural abstractions. Early papers on this subject (Balzer 1967; Galler and Perlis 1967; Mealy 1967) stress the need for separating the logical structure, or semantics, of a data abstraction from its implementation. It has been pointed out that these types of facilities must be an essential part of a programming language in order for the programmer to “express the meaning of some computation without worrying too much about its implementation” (Earley 1971). Knowledge of the implementation of a data abstraction can greatly interfere with the effectiveness of that abstraction. Thus, the goal is to provide facilities that permit the programmer to specify and use a data abstraction independent of its implementation.

The programming language Pascal (Wirth 1971) was the first widely available language that provided and emphasized facilities for defining additional datatypes. Although Simula (Dahl and Nygaard 1966; Dahl et al. 1968) appeared earlier, the importance of its class facility as a basis for realizing data abstractions was not immediately recognized. These two languages have been used as the starting point for much of the more recent work in this area.

Specification Techniques

One area of interest is the study of formal techniques for specifying data abstractions (Earley 1971; Guttag 1976; Liskov and Zilles 1974; Zilles 1975). Specification techniques attempt to specify the important properties of a data abstraction completely in a representation-independent fashion, without reference to any specific programming language. The goal is to discover specification techniques, at the right level of abstraction, that can be used not only as a definition of a data abstraction for the programmer, but also as an aid in constructing reliable and correct software.

There are two general approaches to the specification techniques—operational and definitional—that reflect to some extent the way in which data abstractions can be viewed. The operational approach describes an abstract datatype in terms of the operations that may be performed on it, while the definitional approach describes its properties. Neither approach, however, suggests features that might be included in a programming language, since that is not their goal.

Data-Structuring Facilities

It is the goal of other research to design and implement programming language facilities that aid the programmer in making effective use of data abstractions. Most of this work is performed by designing new languages that include the desired facilities in some form. Examples are CLU (Liskov and Zilles 1974), EL1 (Wegbreit 1974), Mesa (Geschke and Mitchell 1975), VERS (Earley 1971), PASQUAL (Tennent 1975), Alphard (Wulf 1974; Wulf, London and Shaw 1976), Algol 68 (van Wijngaarden et al. 1976), and Simula. Although not all of these languages have been implemented, they are all motivated in part by the desire to discover better data-structuring facilities through experimentation. Some of these languages have additional goals; for example, EL1 is a result of studies in extensible languages, and Alphard is concerned with the “symbiosis between program verification and programming methodology”.

Strong typing and a compiler-based implementation are common characteristics of most of these languages. This seems to be a result of the success of Pascal and Simula. There as been almost no research of this kind using so-called “typeless” languages whose implementations are frequently interpreter-based, such as LISP, APL, SNOBOL4, and SL5.

Another important characteristic of current approaches to data-structuring facilities is that they are based on declarative mechanisms. That is, data structures are *defined* by declarations at compile time rather than by operations at execution time. The tendency is to view a data abstraction as something completely different from a procedural abstraction, and to insist on the complete definition of its behavior during compilation.

A third characteristic of current approaches is an insistence on making the actual implementation of a data structure invisible. This is done by isolating the implementation of a data structure so that only the operations defined on it have access to its “representation variables”. This isolation is provided by the cluster in CLU, by the form in Alphard, and by the class in Simula.

Efficiency appears to be one of the prime motivations for the approach taken in much of the work cited. Indeed, efficiency (both time and space) is sometimes given as the reason for a declarative approach that includes strong typing (for example, Liskov and Zilles 1974). Too much concern over efficiency, however, would seem to have the same inhibiting interference with the effective use of a data abstraction that knowledge of its implementation has. The SL5 approach is motivated in part by the philosophy that the important value of such facilities is that their use may suggest different and better ways to solve problems, rather than limit solutions to a specific regimen.

The SL5 Approach

The approach used in SL5 for the realization of a data abstraction is motivated by, and is a consequence of, the procedure mechanism. The basic idea is that environments, as source-language data objects, provide the basis upon which to construct the realization of a data abstraction. Thus, in SL5, data abstractions and procedural abstractions are built using a common foundation, since in many situations, a portion of a data abstraction is procedural in nature, and vice versa. As described below, data structures in SL5 *are* environments and may be used as such when the need arises. The converse is also true: environments *are* data objects and may be manipulated as such.

As previously mentioned, other approaches to data structure facilities tend to “hide” the implementation of an abstraction from its user, lest conscious or subconscious knowledge of the implementation inhibit its effective use. The facilities in SL5 are motivated in part by the attitude that the design and use of a data abstraction is inhibited, not by its implementation, per se, but by the lack of control over the implementation. The procedural approach gives the programmer control over both the implementation and use of a data abstraction, which should lessen any inhibitive effects.

To support these basic concepts, syntactic structures are provided so that environments can be used to realize both procedural and data abstractions. In the case of procedural abstractions, the syntactic structures are those given in Chapter 3; ***`create`***, ***`with`***, ***`resume`***, ***`return`***, and functional notation manipulate environments as procedural abstractions. The remainder of this chapter describes the various syntactic structures that support the use of environments as data abstractions. To some degree, the use of environments as data abstractions in SL5 is accomplished by observing a few programming conventions. The conventions described below have been designed to facilitate the implementation of a wide variety of data abstractions that programmers often use. The programmer can, however, use whatever conventions best suit the specific application. This flexibility is a consequence of treating environments as data objects.

Using Environments as Records

The ability to access the identifiers in an environment, using the attribute reference operator described in Chapter 3, permits environments to be used as simple record structures. When used in this manner, the identifiers constitute the fields of the record. For example,

```
   employee := procedure (name, age, salary) end
```

assigns to employee a procedure whose environments might be used as records describing some attributes of a company's employees. An environment for employee is an instance of the record:

```
  newhire := create employee
```

The fields of the record could be filled in using the expressions

```
  newhire.name := "Harry";
  newhire.age := 32;
  newhire.salary := 12000.00;
```

or, since the attributes are also the arguments, the record can be created and initialized by the single expression

```
  newhire := create employee with ("Harry", 32, 12000.00)
```

The attributes are not restricted to the arguments. For example, employee could be defined as

```
  employee := procedure (name, age, salary)
      private daysworked;
  end
```

in which the fields `name`, `age`, and `salary` are initialized when the record is created, and the field `daysworked` is updated on a daily basis by an application program.

The new Expression

The creation of a record may also require some initialization in addition to setting the arguments to the given values. Continuing with the example given above, assume that each new employee is credited with 5 working days. In addition, assume that the record must also contain the number of years until the employee is 65 and the proper percentage for the monthly tax deduction. If the record is defined as

```
  employee := procedure (name, age, salary)
      private daysworked, yearstogo, taxrate;
  end
```

then to enter a new employee the following expressions are executed:

```
  newhire := create employee with ("Harry",32,12000.00);
  newhire.daysworked := 5;
  newhire.yearstogo := 65 - newhire.age;
  newhire.taxrate := findrate(newhire.salary);
```

The point is that the creation of a record, or any data structure, often requires some computation in addition to its creation. This "procedural activity" is easily accommodated in the SL5 model since a record is an environment. The record employee can now be redefined as

```
  employee := procedure (name, age, salary)
      private daysworked, yearstogo, taxrate;
      daysworked := 5;
      yearstogo := 65 - age;
      taxrate := findrate(salary);
```

```
        succeed;
    end
```

To create an instance of the record, the environment is created and is resumed once in order to initialize itself:

```
    newhire := create employee with ("Harry",32,12000.00);
    resumenewhire;
```

Notice that the result returned by the initial resumption is ignored; it is the environment itself that is of interest.

The general form for a procedure whose environments are intended to be used as records is

```
        procedure (…)
            … initialization part …
            succeed;
        end
```

The “initialization part” is referred to as the *initializer* for the record inasmuch as it performs whatever is necessary to initialize an instance of the record in addition to its creation.

The general sequence to create an instance of a record is first to create the environment, bind the actual arguments (if present) to that environment, and then resume the environment to permit the initializer to initialize the record. The result returned by the initializer is ignored, and it is the environment itself that constitutes the instance of the record. This sequence is performed by the ***new*** expression. The expression

```
        x := new e
```

is equivalent to

```
        x := (case datatype(e) of
                "procedure": { t := create e; resume t; t };
                "environment": { resume e; e };
            end)
```

where *t* is a temporary variable and is not available to the programmer.

Argument binding for records is accomplished using ***with***. Since ***with*** creates an environment if its left argument is a procedure, an explicit ***create*** is not required. For a record defined by

```
        record := procedure (a1, a2, …, an)
            … initializer …
        end
```

an instance of the record is created and returned by the expression

```
        new record with (e1, e2, …, en)
```

where e_1 through e_n denote the actual argument expressions. The ***with*** expression binds more tightly than the ***new*** expression.

For example, the sequence given above to create an instance of the record `employee` can be replaced by the single expression

```
     newhire := new employee with ("Harry", 32, 12000.00)
```

Using Environments as Data Structures

Records provide a convenient way to represent data objects with a fixed number of fields whose names are known. They are not suitable, however, for use when the “field names” are computed or some sort of subscript is used to access the elements, as in arrays and matrices.

Although lists, described in Chapter 2, provide a simple and efficient mechanism for handling an aggregate of values, they are not meant to be used as *the* primary realization of a data structure. Rather they are intended to be used in conjunction with the mechanisms described this section. Lists themselves provide no more that simple Fortran-like arrays of one dimension and, as such, suffer the same limitations when used in other than that manner.

These types of data structures are also represented by environments, and the ***new*** expression may be used to create instances of them. The general form of the procedure from which to create an instance of a data structure is similar to that given above, with an additional portion called the *accessor*. The general form is

```
 x := procedure (a_1, a_2, …, a_n)
        … initializer …
        succeed;
        … accessor …
end
```

After its creation and initialization, an environment for `x` representing the data structure is resumed with the appropriate arguments in order to access an element of the structure. For example, if `y` is assigned an environment representing a two-dimensional array, the expression `y(5,2)` causes the accessor to reference an element of the array stored within the environment.

This schema is an example of a “segmented” procedure. Notice that the arguments in a segmented procedure of this type are used for *two* purposes. When the data structure is created using the ***new*** expression, the arguments are used to pass information that is required by the initializer. When the environment is subsequently resumed to access an element in the data structure, the arguments are used by the accessor to obtain the values of the subscripts. The initializer is executed only once whereas the accessor may be executed many times. Consequently, the accessor is typically a loop of some kind. A ***repeat*** loop is convenient in cases where the loop body is an expression sequence with a single entry and a single exit.

An Example—Vectors

As a simple example, consider the implementation of vectors with arbitrary bounds. Although a list may be used as a vector, its lower bound is always 1. In order for a list to be used as a vector with arbitrary bounds, addition or subtraction of an offset would be required at each reference. A better approach is to embed this operation inside an implementation of vectors since it is not relevant to their use.

The following procedure illustrates one way to implement vectors. Notice the use of the arguments for two purposes.

```
vector := procedure (a, b) private lb, ub, v;
    v := list(a-b+1);
    lb := a;
    ub := b;
    succeed;
    repeat
        return (v!(a-lb+1) or ""&0);
end
```

An instance of a vector is returned by an expression of the form

```
 x := new vector with (-2, 10)
```

which specifies a vector with a lower bound of -2, an upper bound of 10. Since the formal arguments are changed when a vector is accessed, the lower and upper bounds must be copied to other variables.

The accessor consists of the ***repeat*** expression. If the subscript with which the accessor is resumed is within the bounds of the vector, the expression `a-lb+1` evaluates to a valid index for `v`, the list containing the elements of the vector, and the appropriate element is returned. If the subscript is outside the bounds of the vector, the list indexing operation fails and the accessor returns the result {`""`,*f* }. An important property of this example is that the programmer can define what action is to be taken if the subscript is out of range.

The ***succeed*** expression, described in Chapter 3, interprets its argument, but does not dereference it. Since the built-in operator `!` returns a result whose *V* component is a variable that refers to the appropriate element in the list, the result returned by the accessor in this example contains a variable in its *V* component. This permits expressions that access elements of a vector to be used in contexts requiring a variable, for example, in an assignment expression.

This implementation of vectors is only one of the many possible ways to achieve the desired result. Other implementations may also be used. For example, the elements could be stored in a linked list if it was anticipated that relatively few of the elements would ever be accessed. Conversely, very large vectors can be accommodated by using secondary storage for portions of the vector. The point is that the use of the data abstraction is independent of its implementation and is accomplished by using the procedure vector in the ***new*** expression and using the environment that is returned to reference the elements by resuming it. Thus the programmer can change the implementation of vectors without having to modify the programs that use them.

Attributes of Data Structures

There is no distinction between environments that are used primarily as records and those that are used primarily as data structures. On the contrary, many data abstractions require the use of both forms of access.

In the example of vectors given above, an instance of a vector contains two attributes, `lb` and `ub`, specifying the bounds of the vector. If `x` has been assigned an instance of a vector as illustrated above, the expressions `x.lb` and `x.ub` can be used to refer to the lower and upper bound of that particular instance of a vector, respectively.

It is often more useful to reference the fields of a record by position rather than by name. As an example, consider records created from the procedure `employee`. Suppose it is required that the subscripts 1 through 4 be permitted as an alternate method of accessing the fields `name`, `daysworked`, `taxrate`, and `salary`, respectively. The procedure can be rewritten to provide this feature as follows, without affecting its prior usage.

```
employee := procedure (x, age, salary)
    private name, daysworked, yearstogo, taxrate;
    name := x;
    daysworked := 5;
    yearstogo := 65 - age;
    taxrate := findrate(salary);
    succeed;
    repeat
        (case x of
            1: succeed name;
            2: succeed daysworked;
            3: succeed taxrate;
            4: succeed salary;
        end) or {
            writeline(errfile,
                "illegal subscript to employee");
            fail;
        }
end
```

Note that the first argument had to be renamed because of its dual usage. The representation used for these records may be changed without affecting their use; only the accessor must be modified if necessary in order to maintain the correspondence between the subscripts 1 through 4 and the field names.

Examples of Data Structures

The following examples further illustrate the use of environments as data structures and some programming techniques to manipulate them.

Tables

The table is one of the more useful built-in datatypes in SNOBOL4. A table is an associative store facility that is useful in many applications such as text processing, compilers, and assemblers. The elements in a table `t` are referenced by expressions such as `t("angle")`, which returns the value of the entry associated with the string `"angle"`. Unlike array or vector subscripts, the elements of a table are conceptually unordered. There is no successor function for table subscripts, i.e., given a table subscript, there is no way to compute the “next” subscript.

In the implementation given below, tables behave similarly to SNOBOL4 tables. If a reference is made to a nonexistent entry, one is created and given the default value specified when the table is created as its initial value. This action can be defined by the programmer in SL5, whereas in SNOBOL4 it is fixed by the implementation. The entries in the table are stored as a singly linked list of lists. A new entry is made by inserting a new list at the head of the list.

```
table := procedure (x) private init, tlist, search;
    init := x;

    search := procedure (l, x)
        until compare(l) do
            if compare(l!1, x) then
                succeed l!2
            else l := l!3;
        fail;
    end;

    succeed;

    repeat
        succeed (search(tlist, x) or {
            tlist := [x, init, tlist];
            tlist!2;
        })

    end
```

Of course, this implementation of tables is not very efficient if the number of entries becomes large. This problem concerns only the implementation, however. It does not affect the abstract notion of a table, since the details of the implementation are hidden within the accessor.

Since the implementor of tables does not have to divulge the details of the implementation, different techniques may be used without affecting the use of tables in various programs. For example, the implementor might write the accessor to collect statistics on the use of tables. Such data could then be used to suggest better implementation schemes.

A more interesting feature is that the concrete representation of a data structure does not have to remain constant throughout program execution. Only the abstraction as seen by the user must remain invariant. The accessor can be written to modify the representation depending on the use of that particular instance of the data structure. For example, a table can be implemented to start out with the entries stored in a singly linked list as above. If subsequent accessing of the table indicates that the table is getting large, the representation can be reorganized to provide faster access. The following implementation of tables operates in this fashion: When the table becomes too large, the representation is changed from a list to a hash table.

```
table := procedure (x)
    private init, tlist, n, buckets, t, h;
    private search, hashno;

    init := x;
    n := 0;

    search := procedure (l, x)
        until compare(l) do
        if compare(l!1, x) then
            succeed l!2
        else l := l!3;
        fail;
    end;

    hashno := procedure (x)
```

```
        succeed length(datatype(x))
    end;

    succeed;

    while n < 25 do
        succeed (search(tlist, x) or {
            tlist := [x, init, tlist];
            n+;
            tlist!2;
        });

    buckets := list(37);
    until compare(tlist) do {
        h := remdr(hashno(tlist!1), 37) + 1;
        t := tlist!3;
        tlist!3 := buckets!h;
        buckets!h := tlist;
        tlist := t;
    };

    repeat {
        h := remdr(hashno(x), 37) + 1;
        succeed (search(buckets!h, x) or {
            buckets!h := [x, init, buckets!h];
            n+;
            buckets!h!2;
        });
    }

    end
```

The hash table is organized using linked lists to resolve collisions. When a particular instance of a table contains 25 or more entries, the next resumption causes that table to be reorganized from the single list into the hash table format.

Sequencing through a table. Tables are convenient because of their associative nature. There is no means, however, to sequence through all the entries without knowing all the subscripts. Such an operation is needed, for example, to print the entries. This restriction is taken care of in SNOBOL4 by providing a built-in function that converts a table to a two-dimensional array. The same solution can be used for the tables defined above by writing a procedure that "knows" about the internal structure of tables and produces an array containing all of the entries.

One of the motivations for converting a table to an array is that an array has a well-defined method of sequencing through every element. If this is all that is required, it is not necessary to create a separate structure. It is simpler to write a procedure whose environment takes a table as its argument and returns the next entry in the table each time it is resumed:

```
tseq := procedure (t) private p, i;
    if compare(t.buckets) then {
        p := t.tlist;
        until compare(p) do {
            succeed p!1;
            succeed p!2;
            p := p!3;
        };
    else
        for i from 1 to size(t.buckets) do {
            p := t.buckets!i;
            until compare(p) do {
                succeed p!1;
                succeed p!2;
                p := p!3;
            };
        };
    fail
end
```

A concordance program. To illustrate the use of tables and `tseq`, the following program produces a concordance of the words contained in `infile`. An environment for the procedure `getword(f)`, given in Chapter 3, returns the next word in the file `f` each time it is resumed, and fails at the end-of-file.

```
nextw := getword with infile;
count := new table with 0;
while w := resume nextw do
    count(w)+;
nextw := tseq with count;
while w := resume nextw do
    writeline(outfile, rpad(w, 20) || (resume nextw));
```

Stacks

A common example of a data abstraction is a stack (for example, Liskov and Zilles 1974). A stack `p` is an object with the following operations defined as its attributes.

> `p.push(x)`—pushes `x` onto `p` and returns the result {`x`,*s*}.
>
> `p.pop()`—if `p` is empty, the result {`""`,*f* } is returned; otherwise `p` is popped and the top element is returned.
>
> `p.top()`—if `p` is empty, the result {`""`,*f* } is returned; otherwise the result {`v`,*s*} is returned where `v` is a variable that refers to the top element of the stack.

This example illustrates the definition and use of procedure-valued attributes. The following implementation of stacks is similar to the cluster approach of CLU (Liskov and Zilles 1974). The storage for the stack is represented using a singly linked list of records called `node`s that have two fields, `value` and `link`. In this implementation, the accessor is written to issue an error message and signal failure if an attempt is made to access the stack using some type of subscript.

```
stack := procedure ()
    private push, pop, top;
    public stk;

    push := create procedure (x)
        repeat {
            stk := new node with (x, stk);
            succeed x;
        }
    end;

    pop := create procedure () private t;
        repeat
            if compare(stk) then
                fail
        else {
            t := stk.value;
```

```
                stk := stk.link;
                succeed t;
            }
        end;

        top := create procedure ()
            repeat
                if compare(stk) then
                    fail
                else succeed stk.value
        end;

        repeat {
            writeline(errfile, "illegal access of a stack");
            fail;
        }

    end

    node := procedure (a, b) private value, link;
        value := a;
        link := b;

        succeed;
        stop("illegal access of a node")

    end
```

The public identifier `stk` is the head of the list of nodes representing the stack. The attributes `push`, `pop`, and `top` are assigned environments for procedures in which `stk` is a nonlocal identifier. As a result, the nonlocal identifier `stk` appearing in each procedure refers to the public identifier `stk` in `stack`. This is an example of four parallel environments that communicate through a single public identifier declared in their common ancestor.

The attributes are written using the ***repeat*** expression so their environments may be resumed using functional notation. For example, if `p` is assigned a stack by the expression

```
 p := new stack
```

then `p.push("cat")` pushes the string `"cat"` onto the top of the stack `p`. The expression `p.top() := "dog"` changes it to `"dog"`, and `writeline(outfile, p.pop())` writes `"dog"` to the standard output file.

Infix to Polish Suffix Translation

Writing a procedure that translates arithmetic expressions from infix syntax to Polish suffix syntax provides more substantial examples of the use of data abstractions. The solution to this problem parallels that of Liskov and Zilles (1974), which was used to demonstrate the facilities of CLU, and hence can be used to compare SL5 and CLU.

The procedure polish translates infix expressions such as

$$a + b * (c + d)$$

into their corresponding parenthesis-free Polish suffix form, in this case

$$a\ b\ c\ d + * +$$

The procedure makes use of three data abstractions and one procedural abstraction. It is invoked with two arguments: `scan`, which is an environment that returns the next “token” from the infix expression every time it is resumed, and `grm`, which is a data structure representing the input grammar. The procedure returns the translation as a string. The procedure polish is as follows.

```
polish := procedure (scan, grm)
    private mustscan, t, s, polish;

    s := new stack;
    s.push(grm.eoe);
    mustscan := "yes";
    while s.top() do {
        if mustscan == "yes" then
            t := resume scan or fail
        else mustscan := "yes";
        if t.isop() then
            case grm.precrel(s.top(), t) of
                "<": s.push(t);
                "=": s.pop();
                ">": {polish := polish || s.pop().symbol;
                      mustscan := "no";
                    };
                default: stop("bad grammar")
            end
        else polish := polish || t.symbol;
    };
    succeed polish;

end;
```

This procedure manipulates three different data abstractions. A stack, as defined in the previous example, is used in the conventional fashion for translating infix expressions. The object returned at each resumption of `scan` is an instance of a token having two attributes: `symbol`, which is a string containing the print representation of the token, for example `"a"`, and `isop`, which is a procedure that succeeds if the particular instance of token represents an operator such as +.

The precedence of the various operators are contained in an instance of the grammar, `grm`. Grammars have two attributes: `eoe`, which is a token that indicates the end of an expression, and the procedure `precrel`, which is used to determine the precedence relation between two operator tokens.

The procedure `polish` operates as follows. Upon entry, a stack is created and assigned to `s`, and the end-of-expression token for the grammer is pushed onto the stack. This is done in order to match the actual end-of-expression when it is encountered during translation. The variable `mustscan` is used to indicate if another token is needed from `scan`, and it is initialized to `"yes"`. The main loop is controlled by the state of the stack `s`. As long as it is not empty, which is indicated by `s.top()` succeeding, it holds part of the translation. If `mustscan` indicates that another token is required, `scan` is resumed and the token it returns is assigned to `t`. If `scan` fails, indicating that there are no more expressions to translate, polish returns {`""`,*f*}. If `t` is an operator, the grammar is consulted to determine the precedence relation between the top of the stack and the operator token. If the precedence of `t` is greater than that of the token at the top of the stack, `grm.precrel` returns `"<"`, and t is pushed onto the stack (e.g. `"+"` < `"*"`). If the precedence relation is `"="`, as is the case between left and right parentheses, the stack is popped and both `t` and the token returned by `s.pop()` are ignored. If the precedence relation is `">"`, the operator at the top of the stack is popped and its print representation is appended to the string containing the translation. The variable `mustscan` is set to `"no"` since a new operator has been exposed at the top of the stack with which `t` must be compared. The end-of-expression token has the lowest precedence, and its appearance causes the stack to be emptied as each operator is appended to the translation string. This causes the ***`while`*** loop to terminate, and the translation to be returned.

The important feature of the procedure `polish` is the way in which the various abstractions are used without making a commitment to an implementation. The data abstractions are accessed via their attributes, and `scan` is simply resumed to get the next token. The actual source of infix expressions is not known to the procedure `polish`.

Assume that the procedure `scanner(f, g)` is written to read infix expressions, one on a line, from file `f` returning `g.eoe` as the last token of each one, and to fail at an end-of-file. The following program writes the Polish suffix translation of the expressions in `infile` to `outfile`.

```
g := new grammar;
scan := scanner with (infile, g);
repeat writeline(outfile, polish(scan, g));
```

The implementations of `scanner`, `stack`, `token`, and `grammar` are required to complete this example. Stack is given in the preceding section. A simplified version of the procedure `scanner` can be written as

```
scanner := procedure (f, g) private line, t, lets;
    lets := "abcdefghijklmnopqrstuvwxyz";
    while line := readline(f) do {
        line := after(line," ");
        until line == "" do {
            t := (thru(line,lets) or section(line,0,1));
            line := after(line, t || " ");
            succeed new token with t;
        };
        succeed g.eoe;
    };
    fail;
end
```

Blanks are ignored by scanner, one or more letters are treated as an operand, and one-character operators are assumed. Note that at the end of an expression, the end-of-expression token, which is grammar-dependent, is returned.

A token is created with its print representation as the argument. The scanner creates instances of tokens without knowledge of their other properties, such as `isop`. Tokens are defined by the following procedure.

```
token := procedure (s) private isop, symbol;

    symbol := s;
    isop := (case s of
        "/","+","-","*",
        ")","(","#": procedure () succeed end;
        default: procedure () fail end;
    end);
    succeed;
    stop("illegal reference to a token")

end
```

The sharp symbol, `"#"`, is used by token and grammar as the print representation of the end-of-expression token.

The procedure grammar creates and initializes an operator precedence grammar:

```
grammar := procedure ()
    private eoe, precrel; public prec;

    prec := new table;
    prec("#") := 0;
    prec("(") := prec(")") := 1;
    prec("+") := prec("-") := 2;
    prec("*") := prec("/") := 3;
    eoe := new token with "#";
    precrel := create procedure (t1, t2)
        repeat
            if t2.symbol == "(" or
                prec(t1.symbol) < prec(t2.symbol) then
                    succeed "<"
            else if prec(t1.symbol) > prec(t2.symbol) then
                    succeed ">"
            else succeed "=";
    end;

    succeed;
    stop("illegal reference to a grammar")

end
```

Tables, given above, are used to store the precedences of the various operators. With the exception of `"#"` being used as the print representation of the end-of-expression token, grammar does not need to know about the implementation of tokens.

Protection

One the frequent criticisms of facilities such as the Simula class facility is that all of the attributes of a class are accessible to the user of the class (Liskov and Zilles 1974). Thus the representation of a data object is potentially accessible to the programmer.

A solution to this problem has recently been incorporated into Simula (Palme 1976). Essentially, the solution consists of additional declarations that permit the attributes of a class to be declared *hidden* or *protected*. Attributes declared hidden cannot be accessed outside the class body. The values of attributes declared protected may be inspected outside the class body, but may be changed only within the class body.

Currently, SL5 possesses no protection feature to prohibit the programmer from accessing the representation-dependent attributes of a data structure. It is possible to add declarations, similar to the hidden and protected declarations in Simula, that would have the following effects.

The *hidden* declaration would have no effect on the extent of identifiers described in Chapter 3. It would be a syntactic structure designed to support the use of environments as data abstractions, and would not interfere with their use as procedural abstractions. Hence, unlike Simula, in which identifiers declared hidden must also appear in a type declaration, identifiers in SL5 could be declared hidden without having to also be declared public or private.

Accessible attributes in SL5 environments could be declared *protected* analogous to the ***`protected`*** declaration in Simula. An attribute reference such as `e.x` returns a result of the form {*variable*,*s*}. References to attributes declared protected would cause automatic dereferening of this result as soon as it is formed. Consequently, protected attributes could not be modified by other than the environment in which they are declared. Like the hidden declaration, the protected declaration would have no effect on the extent of identifiers.

The Uniform Reference Problem

The uniform reference problem consists of creating a convenient data-structuring facility that decouples as much as possible the specification of the logical operations on a data structure from its concrete representation (Ross 1969, Geschke and Mitchell 1975); If this goal is attainable, then programs using an object *t* can refer to the attributes of *t* and access its elements without knowledge of the actual implementation of *t*. Thus programs using t do not have to be modified when and if the implementation of *t* is changed.

One way to assess a data-structuring facility is to see to what extent it solves the uniform reference problem. The facilities of SL5 described in the preceding sections provide a partial solution to the uniform reference problem. As illustrated in the examples, the elements of a data structure can be accessed without requiring any knowledge of their implementation. Unlike data structures in other languages, SL5 data structures are what might be termed “intelligent” by virtue of their procedural component. This procedural component, which is programmer-defined, links the syntactic structure to the act of accessing a data structure. It is this procedural component that provides the decoupling between the logical operations on a data structure and its concrete implementation. As a result, uniform references to data structures are possible and are under the programmer's control.

There are two features, however, that prevent the SL5 facility from being a complete solution to the problem. The first is illustrated in the stack example: The programmer must know which attributes in a data structure are procedural in order make correct use of them. This reflects, to some degree, the way in which the data structure is implemented. In a stack, for example, the correct use of the attribute `top` requires that the programmer know that it is a procedural attribute. The actual implementation of a stack may be changed, but the stated use of `top` must remain as initially defined, even if a better implementation scheme is employed.

The second shortcoming of the facility is that there is no simple way to extend the built-in operators and functions to accept programmer-defined data structures as arguments. For example, the programmer might wish to extend the built-in function size so that if it is called with a vector as its argument, it returns the number of elements in the vector. This can be done in a sense by replacing the built-in function by a programmer-defined version. This approach is similar to the OPSYN facility of SNOBOL4 and suffers from the same deficiency. Namely, if upward compatibility is desired, the programmer-defined version must be written to handle all types of arguments instead of just the programmer-defined type. As such, the programmer is formally required to replace rather than extend a built-in operator or function to achieve the desired effect. However, since procedures are data objects, this does not require duplication of the built-in operation. For example, the built-in function `size` can be extended as described above by assigning to `lsize` the original value of `size` and assigning to `size` a procedure that checks if its argument is a vector. If the argument is not a vector, this new version of `size` returns the result of calling `lsize`.

These problems are largely a consequence of the nature of SL5 rather than a deficiency of the data-structuring facilities. Features such as dynamic binding and the lack of type declarations are incompatible with the notion of strong typing that is necessary to resolve these problems. Most languages that claim to have solved the uniform reference problem have compiler-based implementations. In these languages, declarations permit the compiler to determine, for example, which attributes of a data structure are procedures. The compiler can also deduce the types of the arguments to operators such as + and generate code to call a programmer-defined version if one exists. For example, suppose an expression such as `a + b` is equivalent to the function call `plus(a,b)`. If there is a programmer-defined version of `plus` in which the types of the arguments match those of `a` and `b`, the appropriate code can be generated to call that version, which is required to handle only arguments of a prespecified type. Since the types of variables in SL5 are not bound at compile time, transformations such as these cannot be performed by the compiler, and type checking must be done at execution time.

There is no reason why the concepts that underly the SL5 data-structuring facilities cannot be applied to a language in which the types of all the variables are known at compile time and the meaning of operators such as + cannot be changed. In this case, both the shortcomings present in SL5 can be overcome using technique used in Mesa (Geschke and Mitchell 1975), which is similar to the scheme described above.

5. FILTERS

It is often convenient to be able to attach a procedural component to a variable without having to explicitly acknowledge the presence of the procedural component when referring to that variable. For example, as noted in the previous chapter, references to some of the attributes of a data structure should not require explicit knowledge that the attribute is of a procedural nature. The SNOBOL4 statement

```
OUTPUT = LINE
```

is another example. Assignments to OUTPUT (or any other output-associated variable) in SNOBOL4 result in the value assigned being written to the output file. This requires that the variable OUTPUT have a procedural component attached to it to perform the output operation during execution. This statement also illustrates an advantage of such mechanisms, namely, that the extraneous, nonvarying parameters of the procedural component are not required for its use. In the case of OUTPUT, the nonvarying parameters are those that indicate with which file OUTPUT is associated. As a result, the SNOBOL4 statement given above is a precise notation for conveying what it does—writes the contents of LINE to the output file.

In SL5, a procedural component of a variable is called a *filter.* A filter is an environment that is attached to a variable using the mechanism described below. Filters are a generalization of the variable association facility included in SITBOL (Hanson 1976c). Filters form a bridge between the use of environments as procedural abstractions and their use as data abstractions; they possess characteristics of both these uses.

As procedural abstractions, filters provide access, at the source-language level, to the process of argument binding. Thus, unlike most programming languages, in SL5 various methods of argument binding can be defined and controlled by the programmer to best suit the specific application.

As data abstractions, filters provide a method for attaching an implicit procedural component to an attribute of a data structure. Perhaps more importantly, they permit the programmer to define and implement various dynamic protection schemes in conjunction with specific data abstractions, rather than having to rely on a predefined or preconceived notion of protection.

In addition, filters are the basis for the inclusion, at the source-language level, of such seemingly disjoint features as tracing, result modification, and dynamic datatype checking.

Environments form the foundation for the filter facility much in the same way as for the data-structuring facility. Indeed, the usefulness of filters is a consequence of the fact that filters *are* environments, and include all the useful attributes of environments.

The rest of this chapter describes filters and illustrates their use in the ways mentioned above.

Accessing a Variable

A variable denotes a place in an environment where a value resides. The two operations that may be performed on a variable are assigning it a new value and fetching its value. These operations are referred to as value fetching and value assignment, respectively. Figure 2 gives a pictorial representation of these two operations. The circle represents the place denoted by the variable. An arrow directed away from the variable represents fetching its value, and an arrow directed toward the variable represents assignment.

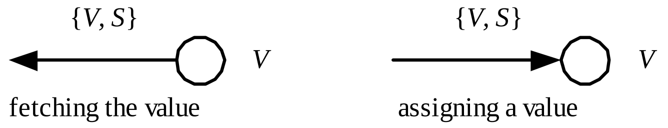

Figure 2. Accessing a Variable.

A reference to a variable produces a result of the form {*variable*,*s*}. The value of the variable is fetched when this result is dereferenced and the *V* component is replaced by the value. Assignment of the result {*V*,*S*} to a variable causes *V* to replace the current value of the variable, unless *S* is the signal *f*, in which case the assignment is not performed. The result of the assignment operation is $\{V,S\}$.

A filter may be thought of as a screen that is placed in the path of these operations. The result of value fetching or value assignment must pass through this screen. A filter is an environment that is capable of modifying the result as it passes through. Pictorially, a filter is placed “in front” of the variable as shown in Figure 3. $\{V',S'\}$ denotes the result of modifying or “filtering” $\{V,S\}$.

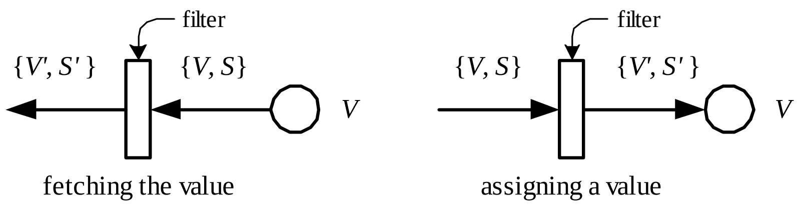

Figure 3. Accessing a Filtered Variable.

A variable may have more than one filter attached to it as shown in Figure 4. An arrangement of several filters in this fashion is called a *pipe*; the result entering one end of the pipe is passed through each filter in turn, finally emerging from the other end, which is indicated by $\{V^*,S^*\}$ in Figure 4.

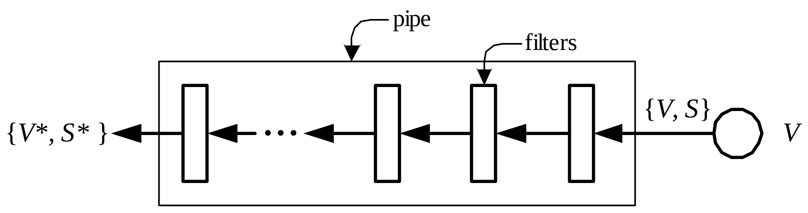

Figure 4. A Pipe.

The terms filter and pipe were suggested by the use of similar terms in the Unix operating system (Ritchie and Thompson 1974) for describing the behavior of a class of programs. For example, a program that converts the characters in a file to upper case by writing a new file is a filter. A program of this type performs a transformation on the contents of an input file to produce a new file as output. Likewise, a pipe is composed of several programs that perform successive transformations. Filters are also described in Kernighan and Plauger (1976) as classes of software tools.

Filter Construction

A filter is an environment for a procedure having the general schema

```
procedure (…)
    … initializer …
    succeed;
    … filtering part …
end
```

This schema is much like that for a procedure whose environments are to be used as data structures. As for data structures, the initializer performs whatever initialization is required, but it is the environment that is of interest. The ***new*** expression is used to create a filter.

The arguments serve two purposes. When a filter is created, the arguments are used to convey whatever information is necessary for initialization. Once a filter is connected to a variable, the filter is resumed whenever the value of the variable is fetched or whenever an assignment is made to the variable, depending on the type of connection. In either case, the value and signal portions of the result to be filtered are passed to the filter as its arguments. Notice that the signal portion of the result is passed as a value in the second argument. Thus the result is decomposed into its two components as it is passed to the filter. After performing the desired modification to each component, the filter recomposes the filtered result and returns it to its resumer.

An Example

As a simple example, a filter created from the following procedure might be called a “cutoff” filter. It permits a result in which the *V* component is a string of five or fewer characters to pass, but changes the result to {`""`,*f*} for longer strings.

```
five := procedure (v, s)
    succeed;

    repeat
        if s = 0 then
            return v&s
        else if length(v) <= 5 then
            return v&s
        else fail

end
```

This filter “propagates” failure in the event that it is resumed with a result having a failure signal. This is a common property of many filters; most modify only the *V* component but must be written to handle results with any signal.

Connecting a Filter to a Variable

A filter may be connected to a variable for one of two operations—value fetching or value assignment. Filters connected to a variable for the purpose of assignment do not interfere with those connected for filtering the value, and vice versa. Any number of filters, for either operation, may be connected to any variable.

The connect Expression

Filters are connected to a variable using the ***connect*** expression:

connect (f_1, f_2) ***to*** *v*

where f_1 and f_2 are the filters (i.e., environments) and *v* is the variable to which they are to be connected. The filter f_1 is connected for filtering results obtained by fetching the value of *v*, and f_2 is connected for filtering assignments to *v*. Both f_1 and f_2 are optional; no corresponding connection is made if an argument is omitted. If only f_1 is specified, the parentheses are not required. In most cases, f_1 and f_2 are environments that have been created and returned by the ***new*** expression. If a procedure is used instead, an implicit ***new*** operation is performed, using that procedure as its argument, and the resulting environment is connected to the variable. The result of the ***connect*** expression is {`""`,*s*}.

As an example, the expressions

```
fiver := new five;
connect fiver to str;
```

connect a filter created from the procedure `five`, given above, to the variable `str`. Subsequent attempts to fetch the value of `str` result in the filter `fiver` being resumed with the current value of `str` and a success signal as arguments. This filter does not interfere with assignments, however. For instance, the expression

```
 str := "too long"
```

assigns the string `"too long"` to `str`. However, since the length of this string is greater than five, subsequent references to the value of `str` cause the filter to return the result {`""`,*f* }.

A filter need not examine the incoming result. The label generator given in Chapter 3, when rewritten as a filter, illustrates this usage:

```
genlabel := procedure (v, s) private prefix, n;

    prefix := v;
    n := s;
    succeed;

    repeat {
        succeed prefix || lpad(n, 4, "0");
        n+;
    }

end
```

A filter is created from this procedure with the arguments indicating the label prefix and the initial label number, respectively. For example, the expression

```
connect new genlabel with ("X", 10) to nextl
```

causes subsequent successive references to the variable `nextl` to return the sequence of labels `X0010`, `X0011`, etc. The value of the variable, given by the argument `v`, is ignored by the filter. The filtered result is completely determined by the procedural component attached to the variable, i.e., the filter.

When several filters are connected to a variable to form a pipe, the order in which they are invoked is determined by the order in which they are connected. The first filter to be connected is “closest” to the variable, and so on.

Disconnecting a Filter

A filter can be disconnected from a variable by an expression such as

`disconnect` *f* ***`from`*** *v*

where *f* is the filter and *v* is the variable. The ***`disconnect`*** expression causes any filters that are the same as *f* to be disconnected from *v*. Two filters are equivalent only if they refer to the same environment. The result of the ***`disconnect`*** expression is {`""`,*s*}, unless *f* is not connected to *v* in which case the result is {`""`,*f* }.

The Operator Form of connect

The ***`connect`*** expression is somewhat cumbersome if many filters are used. The operators `:-` and `:--` are provided as notational shorthand for ***`connect`***. Specifically, the expressions

v `:-` *f*
v `:--` *f*

are equivalent, respectively, to the ***`connect`*** expressions

`connect` `(`*f*`, )` ***`to`*** *v*
`connect` `( ,`*f*`)` ***`to`*** *v*

The Extent of a Filter

When a filter is connected to a variable, it affects all other variables that refer to that variable. This condition arises when a filter is connected to a nonlocal identifier that refers to a location in its custodian. For example, consider the following expressions:

```
a := procedure () public x;
     …
     b();
     …
end

b := procedure ()
     …
     connect f to x;
     …
end
```

where x is a nonlocal identifier in the procedure assigned to b. When b is called from within a, the environment for a becomes the custodian for x. References to x in a after b has been called are affected by the filter connected to x during the execution of b.

Thus the effect of connecting a filter to a variable is the same as assigning it a new value, and is determined by the rules governing the extent of identifiers given in Chapter 3.

Argument Binding Using Filters

As described in Chapter 3, arguments are bound to an environment using the ***with*** expression. By default, actual arguments are transmitted by value. This mode of transmission, however, is only one of many that might be desired by the programmer. Argument transmission is a special form of assignment in which the formal argument identifiers are assigned the *V* components of the results of the corresponding actual argument expressions. As such, the filter facility provides a mechanism for the definition of various modes of argument transmission by the programmer at the source-language level.

The special form of assignment that occurs with argument binding is another primitive operation on a variable. It is called *transmission*. A filter may also be connected to a variable for the operation of transmission. This type of connection does not interfere with the operations of assignment and value fetching. It can be accomplished with an additional argument to the ***connect*** expression:

connect (f_1, f_2, f_3) ***to*** *v*

specifies that f_3 is to be connected to *v* for transmission.

A more convenient way of making a transmission connection is in the procedure heading. The general form of a procedure is

```
procedure (<formal>, <formal>, … ) <declarations>
     <body>
end
```

where a <*formal*> specification has the form

{<*decl*>} *id* {: *e*}

Here, <*decl*> is a ***private*** or ***public*** specification. If <*decl*> is omitted, ***private*** is assumed. The value of *e* is a filter that is called the transmitter for the corresponding argument. This form of specification causes the filter to be connected to the corresponding argument for the operation of transmission when an environment for the procedure is created. The transmitter specification, *e,* may be an arbitrary expression and is evaluated during the execution of the ***procedure*** expression. Its value must be a procedure or an environment in order to be connected to the corresponding argument.

This method of connecting a transmitter is necessary since the arguments are usually transmitted before the execution of the procedure, and connecting the filter from within the procedure would affect only subsequent argument transmission. The transmitter may, however, be changed or augmented during execution by connecting another filter to the desired argument.

Built-In Transmitters

If the transmitter specification is omitted, the current value of the variable `val` is used as the transmitter. The initial value of `val` is a built-in filter that transmits arguments by value. Thus a procedure heading such as

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

is equivalent to

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

The initial value of the variable `ref` is a built-in filter that transmits arguments by reference. This transmitter causes the actual argument expression to be interpreted, rather than evaluated, and the *V* component of the result to be stored in the corresponding formal argument. This is similar to the way in which arguments are passed in Fortran.

Programmer-Defined Transmitters

The built-in transmitters `val` and `ref` provide the basis upon which programmer-defined transmitters can be constructed. The programmer may construct transmitters to perform, for example, datatype and range checking, automatic type coercion, tracing of argument binding, or centralizing common preprocessing of arguments.

For example, the expression

```
positiveint := new procedure (v, s)

    succeed;

    repeat
        if s = 0 then
            return v&s
        else if (v := integer(v)) > 0 then
            return v&s
        else fail

end
```

assigns to `positiveint` a filter that, when used as a transmitter, insures that the corresponding actual argument is a positive integer. To use `positiveint` in this manner, it is included in a procedure heading such as

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

The filter `positiveint` can also be connected to any other variable for the assignment operation in order to insure the variable is assigned only positive integers. The filter itself does not know for what operation it is connected; it simply filters results.

The ***with*** expression fails if the binding of any of the arguments fails. In terms of filters, this occurs when the result emerging from a transmitter contains a failure signal. The following filter modifies the incoming result so that the signal is always success. If the incoming signal is failure, the signal is changed to success.

```
neverfail := new procedure (v, s)

    succeed;

    repeat
        return v&(if s = 0 then 1 else s)

end
```

Note that if the signal does not indicate failure, it is passed along even though it may be something other than the success indicator 1.

A transmitter may behave differently at each resumption. For example, the following filter could be used as the transmitter for the second argument to `vector`, given in Chapter 4, in order to prevent an instance of a `vector` from being resumed with more than one subscript.

```
onceonly := new procedure (v, s)

    succeed;

    repeat
        if s = 0 then
            return v&s
        else if (v := integer(v)) > 0 then {
            return v&s;
            repeat {
                writeline(errfile,
                    "illegal argument transmission");
                fail;
            }
        }
        else fail

end
```

Like the filters created from `genlabel`, the incoming result to a transmitter may be ignored. This permits, for example, the programmer to initialize certain arguments with an initial value that is computed independent of the actual argument expression. If this is a frequent requirement, it can be placed in a single transmitter rather than in every procedure that needs it. The following filter transmits a random number each time it is resumed:

```
random := create procedure (v, s)
    private seed, p, c, m, n;

    seed := 0;
    p := 12621;
    c := 21131;
    m := 100000;
    n := 100;
    succeed;

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

end
```

A random number between 1 and 100 is generated every time this filter is resumed using the linear congruence method (Knuth 1969, p. 9).

Transmitting a Single Argument

The ***with*** expression requires that all of the arguments be bound to the environment at once. Arguments can be bound individually by using the operator `<-`. If `x` is an argument identifier of the procedure for which `e` is an environment, the expression

```
e.x <- exp
```

transmits the expression *exp* to the argument `x`, passing the result through the appropriate filters. The **`<-`** operator can be used to describe the semantics of the ***`with`*** expression: If a_1 through a_n are the formal arguments of the procedure for which *e* is an environment, the expression

```
e with (e1,e2, …, en)
```

is equivalent to the sequence

```
{    e.a1 <- e1 and
     e.a2 <- e2 and
       …    and
     e.an <- en;
     e }
```

The operator <- is not restricted to use with arguments, but may be used with any variable. In the absence of a filter connected for transmission, the operator `<-` is equivalent to `:=`. Using the ***`connect`*** expression, however, a filter may be connected for transmission to any variable. If an assignment is made using the `<-` operator, the result is passed through the filter. Alternatively, the `<-` operator can be used to assign a value to a variable without activating any of the filters connected for assignment.

Examples of Filters

The following examples illustrate various uses of filters. In some cases, filters are used as procedural abstractions while in others they are used as data abstractions. Since filters are concerned with the basic operations of fetching and storing values, this duality is to be expected.

Dynamic Type Declarations

A convenient way to connect a filter to a variable is to make the connection within another procedure. As an example, the procedure `declare(v, dt, tag)` connects a filter to `v` that permits only values of datatype `dt` to be assigned to `v`. The third argument, `tag`, is used to identify `v` in an error message should an attempt be made to assign a value of the wrong type.

```
declare := procedure (v:ref, dt, tag) private typechkr;

    typechkr := procedure (v, s) private dt, tag;
        dt := v;
        tag := s;
        succeed;
        repeat
            if s = 0 then
                return v&s
            else if datatype(v) == dt then
                return v&s
            else {
                writeline(errfile,
                    "attempt to assign illegal value to " || tag);
                fail;
            }
    end;

    v :-- new typechkr with (dt, tag);
    succeed;

end
```

The first argument to declare is passed by reference so that filter is properly connected to the variable used as the actual argument. For example, the expression

```
declare(lineno, "integer", "line number counter")
```

insures that values assigned to the variable `lineno` are integers.

Tracing

A useful feature in SNOBOL4 is the ability to trace assignments to a variable (Griswold et al. 1971, Chapter 8). The filter facility can be used to trace references to a variable for both assignment and fetching its value. The following procedure connects two filters, each created from one procedure, to the variable for which access tracing is desired. Messages are issued to the specified file only if the incoming result indicates success.

```
trace := procedure (v:ref, public tag, public limit, public f)
    private tracer;

    tracer := procedure (v, s) private msg;
        msg := v;
        succeed;
        repeat {
            if s ~= 0 and limit ~= 0 then {
                writeline(f, tag || msg || v);
                limit-;
            };
        return v&s;
        }
    end;

    connect (new tracer with " fetched, value = ",
              new tracer with " assigned ") to v;
    succeed;

    end
```

The argument `limit` is used to limit the amount of trace information issued on a per-variable basis. It is decremented by one for each message that is issued. By declaring the arguments to trace public, it is unnecessary to pass them to tracer. Since they are nonlocal in `tracer`, they will refer to the identifiers with the same name in the environment for `trace`.

As an example of the use of trace, the expression

```
trace(incr, "incr", 100, errfile)
```

causes a message to be issued to the error file for each of the next 100 accesses to `incr`. If `incr` has the initial value 22, then the expression

```
incr := incr + 1
```

results in the messages

```
incr fetched, value = 22
incr assigned 23
```

The important point of this example is that by using filters the programmer can implement the form of tracing that best suits the specific program. Moreover, since any number of filters may be connected to a variable, a filter that is connected by `trace` does not interfere with other filters that might be connected.

Input and Output

Input and output in SNOBOL4 are very simple, and are often cited by programmers as one of the virtues of using the language. The same effect can be *defined* by the programmer in SL5 using a filter. The procedure

```
input := procedure (v:ref, f)

    v :- new procedure (v, s) private f;
        f := v;
        succeed;
        repeat
            return readline(f)
    end;

    succeed;

end
```

performs essentially the same function as the SNOBOL4 function INPUT. It connects a filter to the given variable which returns the next line from the indicated file whenever its value is fetched, ignoring the incoming result. For example, the expression

```
input(in, infile)
```

causes references to `in` to return the next line from the standard input file. The reference fails on end-of-file. As in SNOBOL4, the failure of the reference on end-of-file can be used to control loops:

```
while line := in do
    … process a line …
```

A similar procedure can be written to handle output as in SNOBOL4.

A filter that returns the next line in a file can be rewritten to return whatever is needed for the particular application. For example, a document preparation program usually operates at the level of “text segments”, which consist of the next word including its preceding spaces and immediately adjoining punctuation marks. The following expression connects a filter to `nextsegment`, which returns the next text segment from `infile`.

```
nextsegment :- new procedure (v, s)
    private line, sp, nonsp;

    succeed;

    while line := in do
        while line ~== "" do {
            sp := thru(line, " ") or "";
            line := after(line, " ");
            nonsp := upto(line, " ") or line;
            line := after(line, nonsp);
            succeed sp || nonsp;
        };
    repeat
        fail
end
```

After reaching end of file, subsequent references to `nextsegment` return the result {`""`,*f* }.

Stacks

Filters may be used as procedural components of a data structure. For example, in the implementation of stacks given in Chapter 4, it was noted that the procedural nature of some of the attributes should be kept within the implementation. Specifically, it should not be necessary to know that the attribute `top` is procedural in order to use it. In the implementation of a stack given below, *none* of the attributes require knowledge of their properties in order to be used correctly. This is accomplished by connecting filters to each attribute so that simply referencing them triggers the appropriate action. If `p` is an instance of a stack, assignments to `p.push` cause the value assigned to be pushed onto `p` and references to `p.top` refer to the top stack element or fail if `p` is empty. Fetching the value of `p.pop` returns the top element and pops the stack or fails if `p` is empty.

```
stack := procedure ()
    private push, pop, top, empty;
    public stk;

    push :- create procedure (v, s)
        repeat
            fail
    end;

    push :-- create procedure (v, s)
        repeat {
            if s ~= 0 then
                stk := new node with (v, stk);
                return v&s;
        }
    end;

    pop :- create procedure (v, s) private t;
        repeat
            if compare(stk) then
                fail
            else {
                t := stk.value;
                stk := stk.link;
                succeed t;
            }
    end;

    pop :-- create procedure (v, s)
        repeat
            fail
    end;

    top :- create procedure (v, s)
        repeat
            if compare(stk) then
                fail
            else succeed stk.value
    end;

    top :-- create procedure (v, s)
        repeat
            if s = 0 then
                return v&s
            else if compare(stk) then
                fail
            else {
                stk.value := v;
                return v&s;
            }
    end;

    succeed;
    stop("illegal access to a stack")

end
```

Notice that the filters are created without the use of the ***new*** expression. The ***new*** expression is most useful when the filter requires some initialization. For filters that do not require any initialization, the ***create*** expression may be used instead.

This example also illustrates the use of filters for protection. Two filters are connected to each attribute; one for value fetching and one for assignment. In some cases, one of the filters simply protects the attribute by always failing when it is resumed. For instance, `push` is meant to be used in assignments. In this implementation of a stack, fetching the value of `push` is meaningless so the filter connected to `push` for that purpose converts any incoming result to {`""`,*f*}. Likewise, attempts to assign a value to `pop` fail. The attribute `top` is the only one in which both filters access the stack in a meaningful way.

The actual values of `push`, `pop`, and `top` are not used. The filters contain all the necessary information, and the variables simply provide places to connect the filters. Most importantly, the filters provide a means for keeping within the definition of a stack the information concerning its implementation and how it is accessed.

A Prime Number Sieve

In the examples given thus far, filters are connected to variables only once. No use has been made of the dynamic nature of a filter connection or of pipes. A prime number sieve can be written using a pipe in such a way that each filter in the pipe filters out multiples of a particular prime number by transmitting the result {`""`,*f*}. If the result emerging from the end of the pipe contains a success signal, the *V* component contains a prime number. In this case, another filter, which filters multiples of the new prime, is connected to the variable. This example was inspired by an example given by McIlroy (1968) for demonstrating the use of coroutines, and represents an implementation of the Sieve of Eratosthenes for finding primes (for example, Hoare 1972, pp. 127-130).

This example involves the use of two procedures from which filters are created: `prime`, which filters out numbers that are multiples of a given prime, and `bottom`, which is positioned at the end of the pipe to print out new primes as they emerge and to connect another filter created from `prime` to the variable. These two procedures are written as follows. The procedure `bottom` assumes that `out` is the variable to which the pipe is connected.

```
prime := procedure (v, s) private n;

    n := v;
    succeed;

    repeat
        if s = 0 then
            return v&s
        else if remdr(v, n) = 0 then
            fail
        else return v&s

end;

bottom := procedure (v, s)

    succeed;
    repeat
        if s = 0 then
            fail
        else {
            writeline(outfile, v);
            out :-- new prime with v;
            return v&s;
        }
end;
```

Initially, a filter created from `bottom` must be connected to `out`:

```
out :-- new bottom
```

The primes from 2 to 100 can then be printed by executing the expression

```
for i from 2 to 100 do out := i
```

Figure 5 depicts the arrangement of the filters as they are inserted into the pipe connected to `out`. The pipe acts like a sieve does by allowing only results containing a prime to pass successfully through without modification.

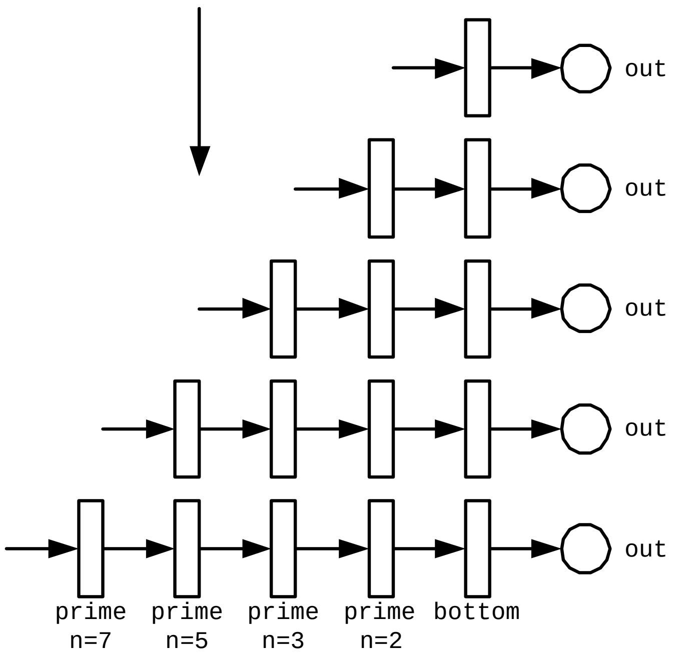

Figure 5. A Prime Number Sieve.

As mentioned above, when several filters are connected to a variable to form a pipe, the order in which they are activated is determined by the order in which they are connected. This convention leads to the arrangment shown in Figure 5.

Capturing a Result

Normally, the signal portion of a result is volatile; its existence can be detected only by its effect on the flow of program execution or by its effect on assignments. A filter, however, receives a “desensitized” result that has been decomposed into its components. This feature can be used to "capture" a result and store it as a data object for subsequent inspection or modification. In addition, once a result has been captured, it can be "released" at a later time.

Environments for the following procedure can be used to represent results with two fields, `value` and `signal`. References to the attribute `release` cause the components to be recombined to form a true result.

```
result := procedure ()
    public value, signal;
    private release;

    release :- create procedure (v, s)
        repeat
            return value&signal
    end;

    succeed;
    stop("illegal access of a result");

end
```

The procedure `result` provides a way to manipulate desensitized results and to release them when desired. It does not, however, capture them. This is accomplished by connecting a filter created from the following procedure:

```
capture := procedure (v, s)

    succeed;

    repeat
        succeed new result with (v, s)

end
```

For example, the expression

```
x :-- capture
```

connects a filter to `x` that captures results assigned to `x` and transforms them into the data representation described above. For instance, an expression such as `x := 25` causes a data object representing the result {25,*s*} to be assigned to `x`. The expression `x := (5 < 3)` causes a data object representing the result {`""`,*f* } to be assigned to `x`. The two attributes of a `result` assigned to `x` may be inspected or modified by using expressions involving `x.value` and `x.signal`. The captured result may be released, as many times as desired, by referencing `x.release`.

6. IMPLEMENTATION

The programming language features described in the preceding chapters have been treated in terms of their use for realizing abstractions. This was also the approach taken for the implementation of the facilities. This chapter gives a brief overview of the implementation of these programming language features.

For the most part, implementation considerations were not permitted to influence the design of the SL5 features. As mentioned in Chapter 1, implementation considerations can inhibit the design of a new programming language. This is especially true for the design of novel features, where lack of experience may lead to the belief that their implementation is difficult when in fact there may exist good techniques that can be discovered. Much of SL5 was designed with the belief that good programming language features lend themselves to good, clean implementations (Wirth 1974). Consequently, this chapter also provides some basis upon which to judge the features. The implementation techniques described in this chapter are essentially equivalent to those used in SL5. The actual implementation of SL5 is somewhat different as a result of additional design goals, for example, portability (Griswold and Hanson 1976).

Much of SL5 is implemented using techniques, or simple extensions of techniques, that are common to many programming language implementations. For example, strings are implemented using a scheme similar to that used in SITBOL (Gimpel 1973). The handling of results in SL5 is a simple extension of the convention used in most languages whereby the value of a function is left in a prespecified location, which is usually a register: SL5 operations leave a value and signal in prespecified registers designated for that purpose.

There are other features of SL5, most notably the procedure mechanism and filters, that are not found in most programming languages, however. The discussion in the remainder of this chapter concentrates on the implementation of these features.

Procedures and Environments

All of the programming language features described in the previous chapters are based in some way on procedures and environments. As such, the implementation of procedures and environments is the most important area to consider.

The implementation of procedures is conceptually similar to the implementation of “routines” in Sil/2, the language in which SL5 is implemented (Druseikis 1975). A procedure is a data object composed of several fields as shown in Figure 6. The environment size field indicates the amount of storage that is required by an environment for the procedure. The symbol table field points to a table containing entries for all the identifiers that appear in the procedure. Each entry indicates the various properties of an identifier, for example, whether it is public, private, or nonlocal. The entry point field indicates the location in the procedure code at which execution should begin. This is established when an environment for the procedure is created.

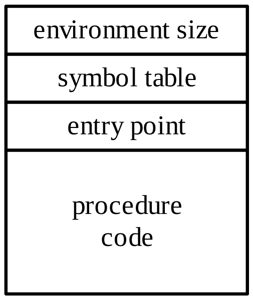

Figure 6. A Procedure.

The code for the procedure body contains the actual instructions for the procedure. Since the storage for all the identifiers in a procedure is contained in an environment for the procedure, references to those identifiers in the code are offsets relative to the contents of a register that points to an environment during execution. These offsets are assigned during compilation.

An environment is a data object that contains all of the information necessary for the execution of the procedure. Figure 7 illustrates the contents of an environment. The creator and resumer fields are the same as those described in Chapter 4. The creator field for an environment points to the environment whose execution caused its creation, and the resumer field points to the environment that last resumed it. When an environment is created, the resumer field is initialized to point to the same environment as the creator field. The procedure field points to the procedure for which the environment is created.

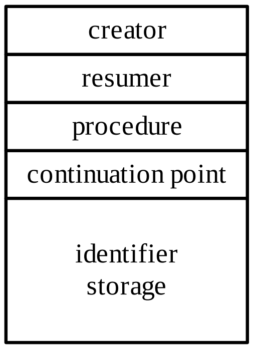

Figure 7. The Contents of an Environment.

The continuation point indicates the location in the procedure body at which execution was most recently suspended. The continuation point is changed only when control is transferred to another environment by the execution of a ***return*** or ***resume*** expression. Although the continuation point can be thought of as the "location counter" of an environment for a procedure, the contents of the continuation point are meaningful only for inactive environments. The continuation point is updated only upon the execution of a ***resume*** or ***return*** expression. The remainder of an environment is used for storing the values of the identifiers that appear in the procedure.

Operations on Procedures and Environments

The implementation of the ***create***, ***with***, ***resume***, and ***return*** expressions can be described in terms of the components of procedures and environments. To facilitate the description of these operations, assume that *ce* indicates the currently active environment, that the fields of an environment or procedure may be accessed using attribute references, and that the function *allocate*(*n*) allocates a block of *n* locations from a free storage region. In terms of an actual machine, *ce* is typically kept in a register assigned for that purpose, the fields of a environment are accessed as fixed offsets from a base register, and allocate is a part of a storage management subsystem.

An environment is created by allocating the appropriate amount of storage and initializing its fields. For a procedure *p*, the required steps are:

```
e := allocate(p.environmentsize)
e.creator := ce
e.resumer := ce
e.procedure := p
e.continuationpoint := p.entrypoint
```

The ***with*** expression is performed by assigning the actual argument expressions to the corresponding formal arguments. This is described in more detail below, in connection with filters.

The ***resume*** and the ***return*** expressions both involve the transfer of control to another environment. This requires that the continuation point of the currently active environment be updated. Then the environment to which control is to be transferred is established as the currently active environment. This two-step operation is referred to as the ***switch to*** operation. The operation ***switch to*** *e*, where *e* is an environment, is implemented by the following sequence.

```
        ce.continuationpoint := z
        ce := e
        go to ce.continuationpoint
    z:  …
```

Here *z* is the location at which execution continues when the suspended environment is subsequently resumed.

The difference between ***return*** and ***resume*** is that the latter requires an explicit indication of the environment to resume and also changes the resumer of that environment. The ***return*** expression simply performs a ***switch to*** the resumer. Specifically, ***return*** is implemented by

```
    switch to ce.resumer
```

and ***resume*** *e* is implemented by

```
    e.resumer := ce
    switch to e
```

Public, Private, and Nonlocal Identifiers

The values of public and private identifiers are stored in an environment for the procedure in which they are declared. Access to an identifier in the procedure simply references the appropriate offset within an environment.

A nonlocal identifier, which does not appear in any of the declarations for the procedure in which it is used, is also assigned an offset in an environment for the procedure. The environment does not contain its value, however. It contains a pointer to the appropriate entry in a custodian for the identifier. This pointer is inserted into the environment when the environment is created. At creation time, for each entry in the symbol table for the procedure that specifies a nonlocal identifier, a search is performed by examining the symbol tables of successive creators until a custodian for the identifier is found. A pointer to the appropriate location in the custodian is stored in the environment under construction.

References to nonlocal identifiers "follow" the pointer in order to locate the value of the identifier. Typically, this requires an extra instruction to accomplish the single level of indirection. For example, Figure 8 shows two parallel environments that communicate through a public identifier `x` declared in a common ancestor. Both locations for `x` in the two parallel environments contain pointers to the value of `x` in the custodian. Each location in an environment includes an indication of whether it is a value or a pointer to a value.

Attribute References

Another way to access a location within an environment is by using an attribute reference. A reference such as `e.x` causes the symbol table of the procedure for which `e` is an environment to be searched for the identifier `x`. If the search succeeds, a pointer the appropriate location is returned.

Since the value of `e` may be an environment for any procedure, the pointer must be computed at execution time by searching the symbol table. If the values of `e` were restricted to be environments for a specific procedure, the offset could be determined at compile time. This is the major difference between the attribute reference in SL5 and similar features in other languages.

Storage Allocation

The storage for procedures and environments is allocated from a heap in the same way as storage is allocated for other objects in the language. When a request for storage cannot be satisfied, a compactifying garbage collection is performed to reclaim storage that is occupied by inaccessible objects.

Although other schemes for the allocation of coroutine environments have been proposed (Prenner, Spitzen and Wegbreit 1972; Bobrow and Wegbreit 1973; Hanson 1976a), they are more oriented towards systems in which environments are protected from direct manipulation by the programmer. In a system such as SL5, in which environments are data objects and are used by the programmer for constructing data objects, there is no guarantee of any relation-

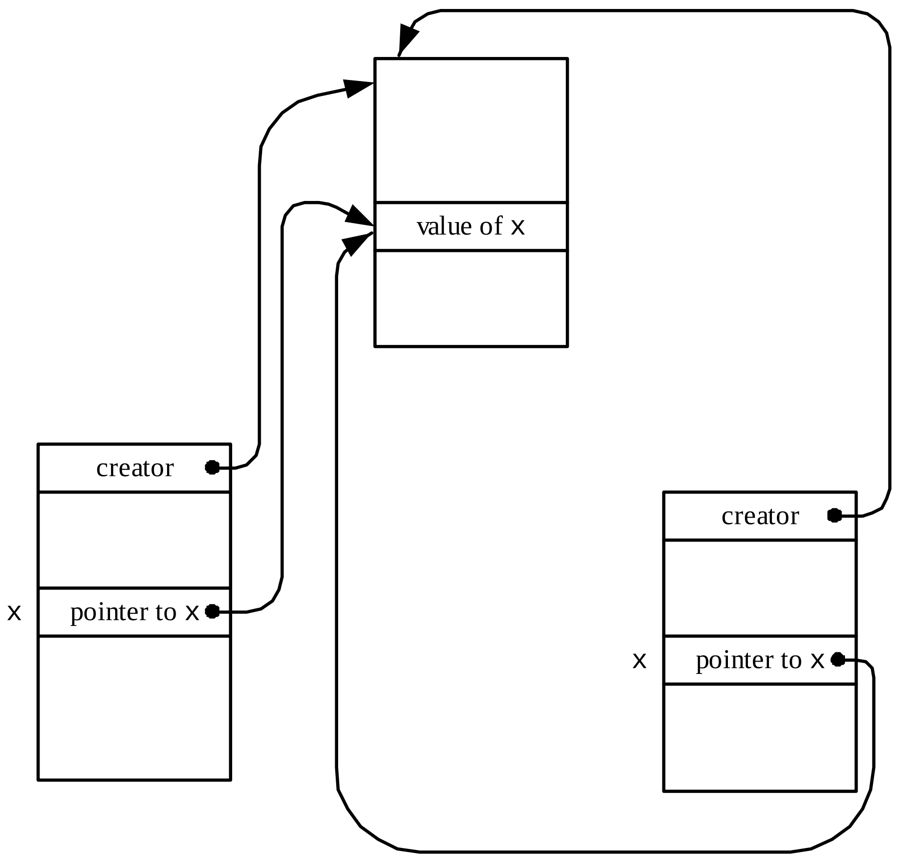

Figure 8. Two Parallel Environments.

ship between the use of an environment and its retention period. Experience so far indicates that it is best to treat environments no differently from other data objects.

Filters

The implementation of filters is similar to the implementation of so-called “trapped variables” in SITBOL (Gimpel 1973; Hanson 1976c). When a filter is connected to a variable, the value of the variable is replaced by a pointer, referred to as a trapped variable, that points to a data structure containing the filter, a code indicating for which operation the filter is connected, and the original value. The procedure code for assignment, argument transmission, and value fetching is written to detect the occurrence of a trapped variable and to invoke the filter.

This approach places the burden of handling the filter with the variable itself rather than having to search a table of some kind whenever an assignment is made or the value is fetched, as is done in the macro implementation of SNOBOL4 (Griswold 1972). The occurrence of a trapped variable indicates that a filter is attached. As such, the overhead for accessing a variable without a filter amounts to testing for a trapped variable.

Multiple filters are implemented using the same technique. In this case, the variable to which the filter is to be connected already contains a trapped variable. Thus the “value” replaced is a trapped variable, which is stored in the data structure containing the filter as the original value of the variable. This leads to a list of trapped variables, which is terminated by the actual value of the variable.

The resumption of a filter is performed using the equivalent of the ***`resume`*** expression, which is described above.

Argument Binding

In addition to the components shown in Figure 6, a procedure also contains the filters for each formal argument. At the time of creation of an environment for a procedure, these filters are connected to the formal arguments. This is accomplished in the same way that other filters are connected: the value of a formal argument in the newly created environment is replaced by a pointer to a data structure containing the filter and the original value, which in this case is the null string.

The procedures for the ***`with`*** expression and the **`<-`** operator simply perform assignments that cause the appropriate filter to be resumed.

7. CONCLUSIONS

This dissertation describes the design of a general procedure mechanism and illustrates its use as the basis for other programming language facilities, most notably a data structuring facility. The motivation for this work was to provide better linguistic mechanisms for the effective use of abstraction. A procedure-based approach was taken in recognition of the effectiveness of procedural abstractions.

SL5 provides a coherent procedure mechanism designed to extend the domain of applicability of procedural abstractions beyond that of procedures based on recursive functions. The major differences between the SL5 mechanism and other procedure mechanisms are that procedures and their environments are data objects, procedure invocation is decomposed into separate linguistic components, and argument binding and transmission is under the control of the programmer. These differences are what permit SL5 environments to be used as coroutines, data strucutures, and filters.

These features are integrated into the procedure mechanism of SL5. For example, the difference between a recursive function and a coroutine is defined by the use of the procedure rather than by any syntactic difference in the specification of the procedure. There are other languages that support some form of coroutines, for example, Bliss (Wulf et al. 1971), SAIL (VanLehn 1973), and Simula. In these languages, however, the linguistic mechanisms for using coroutines are not well-integrated with their respective procedure mechanisms. The result is that the facilities appear to be more of an “add-on” than a planned part of the language, which tends to discourage the use of such facilities. The integrated approach taken for the SL5 procedure mechanism permits the programmer to adapt the general mechanism to a specific application. For example, in addition to the uses described in this paper, the SL5 procedure mechanism has also been used for goal-oriented processes involving backtracking (Griswold 1976c; Hanson 1976d).

The string scanning facilities of SL5 (Griswold 1976b, 1976c) illustrate another way in which environments may be used as data objects. In SL5, environments for certain procedures are used in a manner similar to patterns in SNOBOL4, and may be applied to strings for analysis and synthesis purposes. One of the important differences between the facilities in SL5 and SNOBOL4 is that, in SL5, a scanning procedure and its arguments are embodied in a single environment, which is manipulated by the programmer in the source language. This is a prime example of an application in which the facilities of SL5 permit the realization of an abstraction that has both procedural and data components.

The most significant contribution of this work is the unification of linquistic mechanisms for realizing both procedural and data abstractions. The use of procedural and data abstractions often overlaps. The approach taken in SL5 provides the flexibility that is required to make effective use of this overlap while retaining the ability to enforce a separation of the two types of use if necessary. The net result is that the programmer can define data structures with attributes and access mechanisms that best suit the particular application instead of having to rely on a limited number of built-in facilities. This flexibility permits the programmer to derive the implementation of a data structure from the abstraction instead of deriving the abstraction from knowledge of the implementation of the built-in facilities. Moreover, the language designer is not required to anticipate which of many possible data structures might be useful. This approach is analogous to the approach taken in extensible pattern matching in SNOBOL4 (Griswold 1975b) and string scanning in SL5, both of which include a mechanism for defining scanning procedures so that the vocabulary of built-in pattern primitives can be substantially reduced.

The filter facility, described in Chapter 5, is the product of successive refinements of an implementation technique first used in SITBOL (Gimpel 1973). Its evolution through SNOBOL4 (Hanson 1976c) into SL5 has been accomplished by various changes that enhance its usefulness as a high-level programming language facility. This is evidenced by the use of filters for including programmer-defined methods of argument transmission in the source language. The success of filters for including other features, such as tracing, dynamic protection, and input and output facilities, at the source-language level, suggests that the full potential of filters has yet to be realized.

Suggestions for Further Research

The programming language facilities described in the previous chapters suggest several areas for further research.

Applications

Although the SL5 facilities were designed with some idea of the possible areas of application, there are undoubtedly unexplored areas that might benefit from the facilities of SL5. Additional experience with SL5 is needed to better identify the problem areas for which SL5 is a “good” programming language, and to help determine the suitability of the SL5 features. A more interesting possibility is to determine what affect the features of SL5 might have on the formulation of solutions to problems. It has been said that, as a tool, a programming language has a substantial affect on the way the user of the language approaches a problem (Dijkstra 1972). A systematic study of the use of SL5 is needed to more fully evaluate the usefulness and affect of its facilities.

Numerical programming has received little recent attention from programming language designers. There are, however, aspects of numerical computations that might benefit from a more flexible procedure mechanism and the ability for data structures to have procedural components. Some numerical processes, such as numerical integration, are essentially “stream-oriented” or “producer-consumer” processes. A procedure mechanism that included coroutines might be a more suitable programming tool for these kinds of problems.

Another application area that deserves more attention from programming language designers is data base management systems. The unified view of procedures and data structures in SL5 may be useful in the implementation of such systems.

Data Structure Processing

SL5 provides a unified view of procedures and data structures. This unification, however, has not yet been put to extensive use for data structure manipulation. The facilities described here, and their use in string scanning as mentioned above, suggest that they may provide the necessary mechanism for advanced data structure manipulation of the kind proposed by Hallyburton (1974) for SNOBOL4. Further research in this area would undoubtedly lead to a better understanding of data structures and their role in programming languages.

Theory

The procedure mechanisms of most programming languages are based on the theoretical foundation of recursive function theory. There is no corresponding theoretical foundation upon which to base a procedure mechanism such as that in SL5. Some theoretical work along these lines has been done by Wang and Dahl (1971), but that work was concerned mainly with Simula. More general theoretical models are required in order to deal with the kind of dynamic scope and the decomposition of procedure activation used in SL5.

Program Optimization

Recent research in program optimization has focused on the optimization of binding time (for example, Jones and Muchnick 1976). The basic idea is to permit the programmer to write a program in a language such as SL5 and have the language processor optimize that particular program as much as possible to gain efficiency. The effectiveness of this kind of optimization would vary greatly depending on the features used in a particular program. The advantage of this approach to optimization is that the programmer can write programs in a very flexible language but will incur inefficiencies only when using the more inefficient features.

The evaluation of SL5 in terms of its suitability for this form of optimization might provide additional information concerning the applicability of the facilities described here. Most importantly, if optimization of this type can be performed on SL5 programs, then efficient programs can be realized without requiring a reduction in the flexibility of the language.

Concurrent Processing

The term “coroutine” often connotes the notion of concurrent processes as used in operating systems (Hansen 1973, Chapter 3). Few programming languages provide for concurrent processing, primarily because of the difficulties of specifying a standard interface to an operating system. Simula, PL/I, and SAIL contain facilities for some form of concurrent processing, usually in the form of so-called tasks, but the mechanisms do not appear well-integrated into the rest of the language. A variant of Pascal, called Concurrent Pascal (Hansen 1975), provides these kinds of facilities in connection with its intended use for implementing operating systems.

It has recently been pointed out that the determinacy imposed by modern programming languages is sometimes unnecessary, and that the nondeterminacy resulting from concurrent processing is more appropriate (Dijkstra 1975). The procedure mechanism of SL5 seems ideally suited as a basis for including concurrent processing facilities in programming languages. More intriguing are the possibilities for concurrent data structure processing. Since the SL5 data structure facilities are based on its procedure mechanism, the language may provide a suitable vehicle for investigations in concurrent programming techniques. Research in this area might also result in linguistic mechanisms for unifying some of the concepts used in operating systems and those used in SL5, such as filters and pipes.

APPENDIX: BUILT-IN SL5 OPERATORS AND FUNCTIONS

SL5 contains numerous built-in functions and operators. This appendix contains a list of those functions that are necessary for the programming examples given in this dissertation. A complete list may be found in Griswold (1976a).

Built-In Operators

e_1 `+` e_2 returns the arithmetic sum of e_1 and e_2.
e_1 `-` e_2 returns the arithmetic difference of e_1 and e_2.
e_1 `*` e_2 returns the arithmetic product of e_1 and e_2.
e_1 `/` e_2 returns the arithmetic quotient of e_1 and e_2.
e_1 `<` e_2 succeeds if the arithmetic value of e_1 is less than that of e_2, and fails otherwise.
e_1 `<=` e_2 succeeds if the arithmetic value of e_1 is less than or equal to that of e_2, and fails otherwise.
e_1 `=` e_2 succeeds if the arithmetic value of e_1 is equal to that of e_2, and fails otherwise.
e_1 `~=` e_2 succeeds if the arithmetic value of e_1 is not equal to that of e_2, and fails otherwise.
e_1 `>=` e_2 succeeds if the arithmetic value of e_1 is greater than or equal to that of e_2, and fails otherwise.
e_1 `>` e_2 succeeds if the arithmetic value of e_1 is greater than that of e_2, and fails otherwise.
e_1 `==` e_2 succeeds if the lexical value of e_1 is equal to that of e_2, and fails otherwise.
e_1 `~==` e_2 succeeds if the lexical value of e_1 is not equal to that of e_2, and fails otherwise.
e_1 `||` e_2 returns the result of concatenating the lexical value of e_2 onto the end of the lexical value of e_1.
e_1 `:=` e_2 assigns the value of e_2 to the variable e_1 and returns e_2.
e_1 `&` e_2 returns the result $\{V(e_1),V(e_2)\}$.
`-`e_1 returns the arithmetic value of e_1 with the opposite sign.
`+`e_1 returns the arithmetic value of e1.
e_1`+` increments the value of the variable e_1 by one and returns that value.
e_1`-` decrements the value of the variable e_1 by one and returns that value.

Built-In Functions

`after(s1,s2)` returns the trailing portion of `s1` that follows an initial span of any of the characters from `s2`. Fails if `s1` does not contain any characters from `s2`.

`character(x)` converts `x` to character. Conversion succeeds if `x` is an integer or if `x` is a string of length 1, and fails otherwise.

`compare(x,y)` compares `x` and `y` and succeeds returning the null string if they are equivalent and fails otherwise.

`datatype(x)` returns a string indicating the datatype of `x`, for example, `datatype(5)` returns `"integer"`.

`dupl(s,i)` returns a string consisting of the concatentation of `i` duplications of `s`.

`file(x)` converts `x` to a file. Fails if `x` cannot be converted.

`integer(x)` converts `x` to an integer. Fails if `x` cannot be converted.

`length(s)` returns the number of characters in `s`.

`list(n,x)` returns a list of `n` elements, each initialized to `x`.

`lpad(s,i,c)` returns the result of padding the string `s` on the left with character `c` to produce a string of length `i`. If `c` is omitted, a blank is assumed.

`past(s1,s2)` returns the trailing portion of `s1` that follows and includes the first occurrence of a character from `s2`. Fails if `s1` does not contain any characters from `s2`.

`readline(f)` returns the next line from the file `f`. Fails if `f` is at the end-of-file.

`real(x)` converts `x` to real. Fails if `x` cannot be converted.

`replace(s1,s2,s3)` returns the result of replacing, in `s1`, characters of `s2` by corresponding characters of `s3`. Fails if `s2` and `s3` are of unequal lengths.

`rpad(s,i,c)` returns the result of padding the string `s` on the right with character `c` to produce a string of length `i`. If `c` is omitted, a blank is assumed.

`section(s,i,l)` returns the section of `s` starting at position `i` of length `l`. Character positions in `s` are numbered from the left beginning at 0. Fails if the specified bounds do not fall within `s`. Note that `l` may be negative, in which case the section returned consists of the string between positions `i`-|`l`| and `i`.

`size(l)` returns the number of elements in the list `l`.

`stop(s)` issues a message on outfile consisting of `s` and terminates execution.

`string(x)` converts `x` to a string. Fails if `x` cannot be converted.

`substring(s1,s2,i)` succeeds if `s1` is a substring of `s2` beginning at character position `i` and fails otherwise.

`thru(s1,s2)` returns the initial portion of `s1` consisting of any of the characters in `s2`. Fails if `s1` does not contain any characters from `s2`.

`trim(s1,s2)` returns the result of removing all characters of `s2` from the right end of `s1`.

`upto(s1,s2)` returns the initial portion of `s1` up to the first occurrence of a character from `s2`. Fails if `s1` does not contain any characters from `s2`.

`writeline(f,s)` writes the string `s` to the file `f` followed by the newline character. Returns the null string.

`writestring(f,s)` writes the string `s` to the file `f`. Returns the null string.

REFERENCES

Balzer, Robert M. Dataless Programming, *Proceedings of the Fall Joint Computer Conference*, 1967, 535–544.

Bobrow, Daniel G. and Bertram Raphael. New Programming Languages for Artificial Intelligence, *Computing Surveys*, 6, September 1974, 155–174.

Bobrow, Daniel G. and Ben Wegbreit. A Model and Stack Implementation of Multiple Environments, *Communications of the ACM*, 6, October 1973, 591–603.

Brown, Peter J. The ML/I Macro Processor, *Communications of the ACM*, 10, October 1967, 618–623.

Cheatham, Thomas E. The Introduction of Definitional Facilities into Higher Level Programming Languages, *Proceedings of the Fall Joint Computer Conference*, 1966, 623–637.

Cheatham, Thomas E. Motivation for Extensible Languages, *SIGPLAN Notices*, 4, August 1969, 45–49.

Dahl, Ole-Jahn and C. A. R. Hoare. Hierarchical Program Structures, *Structured Programming*, Academic Press, London, 1972, 175–220.

Dahl, Ole-Jahn, Bjorn Myhrhaug and Kristen Nygaard. The Simula 67 Common Base Language, Norwegian Computing Centre, Oslo, Norway, 1968.

Dahl, Ole-Jahn and Kristen Nygaard. Simula—an Algol-Based Simulation Language, *Communications of the ACM*, 9, September 1966, 671–678.

Dijkstra, Edsger W. Recursive Programming, *Programming Systems and Languages*, Saul Rosen (ed.), McGraw-Hill, New York, 1967.

Dijkstra, Edsger W. The Humble Programmer, *Communications of the ACM*, 15, October 1972, 859–866.

Dijkstra, Edsger W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs, *Communications of the ACM*, 18, August 1975, 453–457.

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

Druseikis, Frederick C. The Design of Transportable Interpreters, SNOBOL4 Project Document S4D49, Department of Computer Science, The University of Arizona, Tucson, February 1975.

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

Earley, Jay. Toward an Understanding of Data Structures, *Communications of the ACM*, 14, October 1971, 617–627.

Galler, Bernard A. Extensible Languages, *Information Processing 74*, North Holland, Amsterdam, 1974, 313–316.

Galler, Bernard A. and Alan J. Perlis. A Proposal for Definitions in Algol, *Communications of the ACM*, 10, April 1967, 204–219.

Geschke, Charles M. and James G. Mitchell. On the Problem of Uniform References to Data Structures, *IEEE Transactions on Software Engineering*, SE-1, June 1975, 207–219.

Gimpel, James F. A Design for SNOBOL4 for the PDP-10, SNOBOL4 Project Document S4D30b, Bell Laboratories, Holmdel, New Jersey, May 1973.

Griswold, Ralph E. *The Macro Implementation of SNOBOL4, A Case Study of Machine-Independent Software*, W. H. Freeman, San Francisco, 1972.

Griswold, Ralph E. Suggested Revisions and Additions to the Syntax and Control Mechanisms of SNOBOL4, *SIGPLAN Notices*, 9, February 1974, 7–23.

Griswold, Ralph E. A Portable Diagnostic Facility for SNOBOL4, *Software—Practice and Experience*, 5, January-March 1975a, 93–104.

Griswold, Ralph E. Extensible Pattern Matching in SNOBOL4, *Proceedings of the ACM Annual Conference*, October 1975b, 248–252.

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

Griswold, Ralph E. String Scanning in SL5, SL5 Project Document S5LD5a, Department of Computer Science, The University of Arizona, Tucson, June 1976b.

Griswold, Ralph E. String Analysis and Synthesis in SL5, *Proceedings of the ACM Annual Conference*, October 1976c, 410–414.

Griswold, Ralph E. and David R. Hanson. An Overview of the SL5 Programming Language, SL5 Project Document S5LD1a, Department of Computer Science, The University of Arizona, Tucson, February 1976.

Griswold, Ralph E., James F. Poage and Ivan P. Polonsky. *The SNOBOL4 Programming Language*, second edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

Guttag, John V. Abstract Data Types and the Development of Data Structures, *Proceedings of the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition and Structure*, March 1976.

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

Hansen, Per Brinch. *Operating System Principles*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

Hansen, Per Brinch. The Programming Language Concurrent Pascal, *IEEE Transactions on Software Engineering*, SE-1, June 1975, 199–207.

Hanson, David R. A Simple Variant of the Boundary-Tag Algorithm for the Allocation of Coroutine Environments, *Information Processing Letters*, 4, January 1976a, 109–112.

Hanson, David R. The Syntax and Semantics of SL5, SL5 Project Document S5LD2a, Department of Computer Science, The University of Arizona, Tucson, April 1976b.

Hanson, David R. Variable Associations in SNOBOL4, *Software—Practice and Experience*, 6, April-June 1976c, 245–254.

Hanson, David R. A Procedure Mechanism for Backtrack Programming, *Proceedings of the ACM Annual Conference,* October 1976d, 401–405.

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

Hoare, C. A. R. Notes on Data Structuring, *Structured Programming*, Academic Press, London, 1972, 83–174.

Hoare, C. A. R. Hints on Programming Language Design, Technical Report, Computer Science Department, Stanford University, Stanford, California, October 1973.

Jones, Neil D. and Steven S. Muchnick. Binding Time Optimization in Programming Languages: Some Thoughts Toward the Design of an Ideal Language, *Third ACM Symposium on Principles of Programming Languages*, January 1976, 77–94.

Kernighan, Brian W. and P. J. Plauger. *Software Tools*, Addison-Wesley, Reading, Massachusetts, 1976.

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading, Massachusetts, 1969.

Knuth, Donald E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, second edition, Addison-Wesley, Reading, Massachusetts, 1973.

Leavenworth, Burt M. Syntax Macros and Extended Translation, *Communications of the ACM*, 9, November 1966, 790–793.

Liskov, Barbara H. and Stephen N. Zilles. Programming with Abstract Data Types, *SIGPLAN Notices*, 9, April 1974, 50–59.

Liskov, Barbara H. and Stephen N. Zilles. Specification Techniques for Data Abstractions, *IEEE Transactions on Software Engineering*, SE-1, March 1975, 7–19.

McIlroy, M. Douglas. Macro Instruction Extensions of Compiler Languages, *Communications of the ACM*, 4, April 1960, 214–220.

McIlroy, M. Douglas. Coroutines, Technical Report, Bell Laboratories, Murray Hill, New Jersey, May 1968.

Mealy, George M. Another Look at Data, *Proceedings of the Fall Joint Computer Conference*, 1967, 525–534.

Palme, Jacob. A New Feature for Module Protection in Simula, *SIGPLAN Notices*, 11, May 1976, 59–62.

Pratt, Terrence W. *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

Prenner, Charles, Jay Spitzen, and Ben Wegbreit. An Implementation of Backtracking for Programming Languages, *Proceedings of the ACM Annual Conference*, August 1972, 763–771.

Richards, Martin. BCPL: A Tool for Compiler Writing and Systems Programming, *Proceedings of the Spring Joint Computer Conference*, 34, 1969, 557–566.

Ritchie, Dennis M. and Ken Thompson. The Unix Time-Sharing System, *Communications of the ACM*, 17, July 1974, 365–375.

Ross, Dougles T. Uniform Referents: An Essential Property for a Software Engineering Language, in *Software Engineering*, vol. 1, Julius T. Tou, ed., Academic Press, New York, 1969, 91–101.

Sammet, Jean E. *Programming Languages: History and Fundamentals*, Prentice-Hall, Englewood Cliffs, New Jersey, 1969.

Standish, Thomas A. Extensibility in Programming Language Design, *SIGPLAN Notices*, 10, July 1975, 18–21.

Tennent, Robert D. PASQUAL: A Proposed Generalization of Pascal, Technical Report, Department of Computing and Information Science, Queens University, Kingston, Canada, February 1975.

VanLehn, Kurt A. SAIL User Manual, Technical Report STAN-CS-73-373, Department of Computer Science, Stanford University, Stanford, California, July 1973.

van Wijngaarden, Adriaan, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens and R. G. Fisher, Revised Report on the Algorithmic Language Algol 68, *Acta Informatica*, 5, January 1976, 1–236.

Wang, Arne and Ole-Jahn Dahl. Coroutine Sequencing in a Block Structured Environment, *BIT*, 11, April 1971, 425–449.

Wegbreit, Ben. *Studies in Extensible Programming Languages*, Ph.D. Dissertation, Center for Research in Computing Technology, Harvard University, Cambridge, 1970.

Wegbreit, Ben. The ECL Programming System, *Proceedings of the Fall Joint Computer Conference*, 1971, 253–262.

Wegbreit, Ben. The Treatment of Data Types in EL1, *Communications of the ACM*, 17, May 1974, 251–264.

Wirth, Niklaus. The Programming Language Pascal, *Acta Informatica*, 1, January 1971, 35–63.

Wirth, Niklaus. On the Design of Programming Languages, *Information Processing 74*, North Holland, Amsterdam, 1974, 386–393.

Wirth, Niklaus. An Assessment of the Programming Language Pascal, *IEEE Transactions on Software Engineering*, SE-1, June 1975, 192–198.

Wirth, Niklaus and C. A. R. Hoare. A Contribution to the Development of Algol, *Communications of the ACM*, 9, June 1966, 413–432.

Wulf, William A. Alphard: Toward a Language to Support Structured Programming, Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, April 1974.

Wulf, William A., Ralph L. London and Mary Shaw. Abstraction and Verification in Alphard, Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, February 1976.

Wulf, William A., D. B. Russell and A. N. Habermann, Bliss: A Language for Systems Programming, *Communications of the ACM*, 14, December 1971, 780–790.

Zilles, Stephen N. Algebraic Specification of Data Types, Technical Report, IBM Research Laboratory, San Jose, California, January 1975.