Filters in SL5*

D. R. Hanson†

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

The filter facility of the SL5 programming language permits the programmer to attach a procedural component to a variable. This procedural component can be used to 'filter' values assigned to the variable or to filter the value of the variable whenever it is fetched. Filters are the basis for the inclusion, at the source language level, of such features as tracing, dynamic datatype checking, generators, and dynamic protection mechanisms. Most importantly, filters are used for argument binding, permitting various methods of argument transmission to be defined in SL5 itself. This paper gives an overview of SL5, describes the filter facility, and gives several examples of its use.

(Received December 1976)

1. Introduction

It is often convenient to be able to attach a procedural component to a variable without having to acknowledge explicitly the presence of the procedural component when referring to that variable. An example of this convenience is illustrated by the SNOBOL4 (Griswold *et al*., 1968) statement

OUTPUT = LINE

Assignments to OUTPUT in SNOBOL4 result in the assigned value being written to the output file. OUTPUT has a procedural component attached to it to perform the output operation during execution. This example also illustrates an advantage of such mechanisms: the extraneous, nonvarying parameters of the procedural component are not required for its use. In the case of OUTPUT, a nonvarying parameter is the file with which OUTPUT is associated. As a result, the statement given above is a precise notation for conveying what it does—writes the contents of LINE to the output file. Other features in SNOBOL4, such as tracing and keywords, also require that some variables have procedural components. In SNOBOL4, such procedural components are specialised and are handled in an *ad hoc* fashion.

This paper describes a general linguistic mechanism in the SL5 programming language (Griswold and Hanson, 1977) that permits the programmer to attach a procedural component to a variable. This procedural component is called a *filter*, and may be attached to a variable for filtering values assigned to the variable or for filtering the value of the variable whenever it is fetched.

Filters are a refinement of the variable association facility in SITBOL (Hanson, 1976b), an implementation of SNOBOL4. The SL5 concept is more general however. Filters form the basis for the inclusion, at the source language level, of features such as dynamic datatype checking, tracing, dynamic protection schemes, and SNOBOL4-style input and output. Filters also are used for argument binding, permitting the programmer to define various methods of argument binding in SL5 itself.

An overview of SL5 is given in Section 2. The SL5 procedure mechanism, upon which the filter facility is based, is described in Section 3. Section 4 describes the filter facility, and Section 5 reveals how filters are used for argument transmission. Several examples of the use of filters are given in Section 6.

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. 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; 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.

2.1 *Expressions and 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 behaviour 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.

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 {"", 0}, the result of $100 < 102$ is {"", 1}.

2.1.1 *Dereferencing*

The result of a simple variable reference is {variable, 1} 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, 1} which is written simply as $\{x, 1\}$.

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 $\{x, 1\}$ produces the result {15, 1}. 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:

*This work was supported in part by the National Science Foundation under Grant DCR75-01307 while the author was at the University of Arizona, Tucson, Arizona.

†Now at Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, USA

$\{x, 1\} + \{y, 1\}$
$\{15, 1\} + \{6, 1\}$
$\{15 + 6, 1\}$
$\{21, 1\}$

The execution of an expression without dereferencing is called *interpretation*. For example, the result of the interpretation of the simple expression x is $\{x, 1\}$. In some cases, interpretation may be followed by dereferencing, in which case a result such as $\{x, 1\}$ becomes $\{15, 1\}$. The combination of interpretation followed by dereferencing is called *evaluation*. The terms 'interpret' and 'evaluate' are used below to describe the behaviour of the SL5 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, 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.

2.1.2 *Transmission of failure*

Most built in operators and functions transmit failure if any of their arguments fail. Usually, the result {"", 0} 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 {"", 0}. Since the operator := performs the indicated assignment only if its right argument does not fail, the assignment is not performed and the result {"", 0} is returned, thereby transmitting the failure signal. The result of the entire expression is therefore {"", 0}.

2.1.3 *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.

2.2 *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.

A summary of some of the SL5 control expressions is given in **Table 1**. In Table 1 phrases enclosed in square brackets are optional. A complete list of control expressions is given in Hanson (1976a).

2.3 *Primitive datatypes*

The primitive datatypes of SL5 are integer, character, real, string, file, procedure and 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. The datatypes procedure and environment are described in subsequent sections.

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.

Table 1 SL5 control expressions

$\{e_1, e_2, \ldots, e_n\}$
Each expression in the sequence is interpreted. The result of the entire expression is the result of e_n.

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 the result of the **if** expression. If e_1 fails, the result is the result of interpreting e_3.

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

e_1 **and** e_2
The result of the **and** expression is the result of interpreting e_1, if that interpretation fails. Otherwise, the result of interpreting e_2 is returned.

while e_1 **do** e_2
e_2 is repeatedly interpreted as long as the evaluation of e_1 succeeds. The result is the result of the last interpretation of e_2. If e_2 is never interpreted, the result is {"", 1}.

repeat e
e is repeatedly interpreted until it fails. The result is the result of the last interpretation of e, which necessarily has a failure signal.

for e_1 [**from** e_2] [**to** e_3] [**by** e_4] [**while** e_5] **do** e_6
e_1 is interpreted and the V component of its result must be a variable, which is called the control variable. e_2 through e_4 are evaluated and their V components are used to control the loop in the conventional manner. The loop consists of repeatedly interpreting e_6 as long as the evaluation of e_5 succeeds, and the value of the control variable satisfies the conditions given by e_2 and e_3. The result is the result of the last interpretation of e_6, or {"", 1} if e_6 is never interpreted. If the phrase **by** e_4 is omitted, **by** 1 is assumed.

Note: Square brackets enclose optional phrases.

3. Procedures

SL5 provides a procedure mechanism that is a generalisation 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 generalise the usefulness of procedural abstractions. Indeed, filters are a direct consequence of the features of the SL5 procedure mechanism. In addition, the procedure mechanism is designed for use in goal directed programming (Hanson, 1976d; Griswold, 1976), and is the basis for the SL5 data structuring facility (Hanson, 1976c).

3.1 *Procedure construction*

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

procedure $(a_1, a_2, \ldots, a_n)$ ⟨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. Procedures can be created at any time and can be transmitted as values throughout the program.

3.2 *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 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 creation of an environment (activation record) for the procedure, binding of the actual arguments to the environment, and resumption of the execution of the procedure associated with the environment.

3.2.1 *Environment creation*

The expression

$$e := \textbf{create}\ p$$

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

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

3.2.2 *Argument binding*

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

$$e\ \textbf{with}\ (a_1, a_2, \ldots, a_n)$$

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

3.2.3 *Procedure resumption*

A procedure is activated by the **resume** expression. The resume expression is written

$$\textbf{resume}\ e$$

where e is an environment. This operation is referred to as 'resuming e'.

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

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

3.2.4 *Procedure returns*

While the **resume** expression requires an explicit indication of the environment to which control should be transferred, the **return** expression always returns control to the resumer of the current environment. The expression

$$\textbf{return}\ r$$

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

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

Since the signals success and failure are used so frequently in SL5, the expressions **succeed** r and **fail** r are provided as equivalents, respectively, to **return** r&1 and **return** r&0. Note that a value is returned even if the signal is failure. If r is omitted, the null string is assumed.

3.2.5 *Functional notation*

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

$$\textbf{resume (create}\ f\ \textbf{with}\ (e_1, e_2, \ldots, e_n))$$

Note that the functional notation results in the invocation of the procedure that is the current value of f, not a procedure named f.

3.3 *Accessing the attributes of environments*

The identifiers in an environment may be accessed using the *environment 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 environment reference. The environment reference operator associates to the left so that $a.b.c$ is equivalent to $(a.b).c$.

3.4 *The extent of identifiers*

Declarations are used to determine the *extent* to which an identifier is known throughout the program. This is sometimes referred as scope on 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 to avoid confusion.

Identifiers may be declared either **public** or **private** in declara-

tions of the form

private $id_1, \ldots, id_n$
public $id_1, \ldots, id_n$

Private identifiers are accessible only to the procedure in which they are declared. Private identifiers are used for data that is local to a particular environment. Unless otherwise declared, the formal arguments of a procedure are private identifiers.

Public identifiers are accessible to the procedure in which they are declared and to any procedure whose environment is a descendant of the environment for the procedure containing the public declaration. Identifiers that do not appear in any of the declarations for the procedure in which they are used are termed *nonlocal* identifiers. Nonlocal identifiers are bound to the appropriate public identifiers upon the creation of an environment for the procedure in which they appear. This is accomplished by examining successive creators until one is found whose procedure contains a public declaration for the nonlocal identifier. When SL5 procedures are used in the usual recursive fashion, this is equivalent to the kind of dynamic scope used in SNOBOL4 and LISP. Further details concerning the interpretation of identifiers are given in Britton *et al.* (1976) and Hanson and Griswold (1978).

3.5 *An example*

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 (p, n)
  repeat {
    succeed p || lpad(n, 4, "0");
    n := n + 1
    }
end
```

(The built in function lpad pads *n* with zeros to form a 4-character string, and || denotes string concatenation.) 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)

or by simply resetting the values of *p* and *n*:

nextlab.p := "L";
nextlab.n := 100;

4. Filters

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. Fig. 1 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.

A reference to a variable produces a result of the form {variable, 1}. 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 0, 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 Fig. 2. $\{V', S'\}$ denotes the result of modifying or 'filtering' $\{V, S\}$.

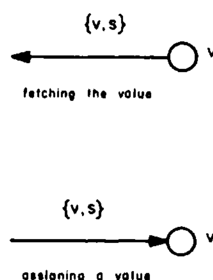
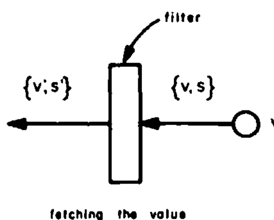


Fig. 1 Accessing a variable



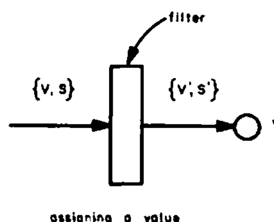


Fig. 2 Accessing a filtered variable

A variable may have more than one filter attached to it as shown in Fig. 3. 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 Fig. 3.

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 behaviour 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.

4.1 *Filter construction*

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

```
procedure (. . .)
  . . . initialisation part . . .
  succeed;
  . . . filtering part . . .
end
```

The initialisation part is executed once and performs the operations necessary to initialise the environment. The general sequence to create a filter is to first create the environment,

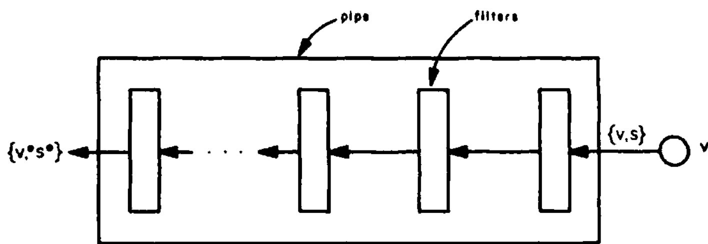


Fig. 3 A pipe

bind the actual arguments to that environment (if present), and then resume the environment once in order to initialise the filter. The result returned by the initialisation part is ignored, and it is the environment itself that constitutes the filter. This sequence is performed by the **new** expression:

x := **new** f

and is equivalent to

x := **create** f;
resume x;

If the initialisation part requires arguments, the **with** expression is used:

x := **new** f **with** $(a_1, \ldots, a_n)$

The **new** expression is also employed when environments are created for use as data structures (Hanson, 1976c).

The arguments serve two purposes. When a filter is created, the arguments are used to convey whatever information is necessary for initialisation. 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.

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 {"", 0} 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.

4.2 *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.

Filters are connected to a variable using the operators :- and :—. The :- operator is used to connect a filter for filtering results obtained by fetching the value of the variable, and :-- is used to connect a filter for filtering assignments to the variable. The right argument to both operators must be a procedure or an environment. If the argument is an environment, that environment is connected to the variable given by the left argument. 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 both operators is the result of the right argument.

As an example, the expression

str :- **new** *five*;

connects 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 *five* 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 {"", 0}.

A filter need not examine the incoming result. The label generator given in Section 3.5, 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 := n + 1
  }
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

nextl := **new** *genlabel* **with** ("X", 10)

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.

4.3 *Disconnecting a filter*

A filter can be disconnected from a variable by an expression such as v –: f where f is the filter and v is the variable. This 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 –: operator is {"", 1}, unless f is not connected to v in which case the result is {"", 0}.

4.4 *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. For example, consider the following expressions:

a := **procedure public** *x*; . . . ; *b*( ); . . . **end**;
b := **procedure** . . . ; *x* :– *f*; . . . **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 Section 3.4.

5. Argument binding using filters

As described in Section 3.2.2, 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. 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 in the source language.

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 value fetching and assignment. A filter is connected to a variable for transmission by the :--- operator, e.g. *v*:---*f* specifies that *f* 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.

5.1 *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* : *vrl*)

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.

5.2 *Programmer defined transmitters*

The built in transmitters *val* and *ref* provide the basis upon which programmer defined transmitters can be constructed. Transmitters may be constructed to perform, for example datatype and range checking, automatic type coercion, tracing of argument binding, or centralising 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, ensures 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 transmitter permits the argument to be transmitted only once. Subsequent attempts at argument transmission cause an error message to be issued and the transmission to fail.

```
onceonly := new procedure (v, s)
  succeed;
  repeat
    if s = 0 then
      return v&s
    else {
      return v&s
      repeat {
        writeline(errfile,
          "illegal argument transmission");
        fail
        }
      }
  end
```

Like the filters created from *genlabel*, the incoming result to a transmitter may be ignored. This permits, for example, certain arguments to be initialised 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).

5.3 *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

$$e \textbf{ with } (e_1, e_2, \ldots, e_n)$$

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 operator :---, 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.

6. Examples of filters

6.1 *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
```

(*datatype* is a built in function that returns a string denoting the datatype of its argument, and == denotes string comparison.) 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")
```

ensures that values assigned to the variable *lineno* are integers.

6.2 *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 := limit - 1
    };
   return v&s
   }
 end;
 v :- new tracer with "fetched, value = ";
 v :-- new tracer with "assigned";
 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.

6.3 *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 with(f);
  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.

6.4 *Data structure accessing*

In SL5, data structures are constructed using environments and their fields are accessed using the environment reference operator described in Section 3.3 (Hanson, 1976c). For example, environments for the procedure defined by

node := **procedure** (*value*, *link*) **end**

can be used as records representing nodes having a *value* and a *link* field.

Filters can be connected to the fields of a data structure for protection purposes or so that the user of the data structure is not required to know which fields have procedural values. As an example, in the implementation of a stack given below, filters are connected to each field so that simply referencing them triggers the appropriate action. A stack is created by an expression such as

p := **new** *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 := create 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
 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 initialisation. For filters that do not require any initialisation, the **create** expression may be used instead.

This example also illustrates the use of filters for protection. Two filters are connected to each field: one for value fetching and one for assignment. In some cases, one of the filters simply protects the field value 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 {" ", 0}. Likewise, attempts to assign a value to *pop* fail. *Top* is the only field 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 of keeping within the definition of a stack the information concerning its implementation and how it is accessed.

6.5 *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 {" ", 0}. 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

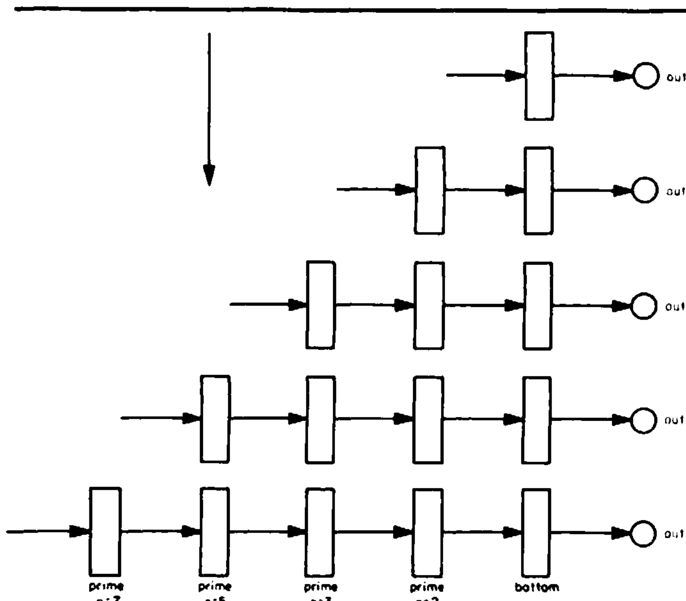


Fig. 4 A prime number sieve

of the Sieve of Eratosthenes for finding primes (for example, see Hoare, 1972).

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*

Fig. 4 depicts the arrangement of the filters as they are inserted into the pipe connected to out. The pipe acts as a sieve does by allowing only results containing a prime to pass successfully through without modification.

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 arrangement shown in Fig. 4.

6.6 *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 'desensitised' 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 as records representing 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, public signal
  private release;
  release :- create procedure (v, s)
    repeat
      return value&signal
  end;
  succeed;
end
```

The procedure *result* provides a way to manipulate desensitised 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, 1} to be assigned to x. The expression $x := (5 < 3)$ causes a data object representing the result {" ", 0} 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*.

7. Conclusions

The filter facility is a result of recent research in new linguistic mechanisms for making effective use of abstraction in the programming process (Hanson, 1976c). Filters provide a way to integrate, at the source language level, a number of seemingly disjoint features as described above. This initial success suggests that the filter facility may provide a basis for the realisation of many abstractions that are difficult to realise in other programming languages. Current research is concentrating on the use of filters in data structure processing and as a means in implementing programmer defined protection schemes.

Acknowledgements

I would like to thank Ralph E. Griswold for many stimulating discussions during the course of this research.

References

Britton, D. E. *et al.* (1976). Procedure Referencing Environments in SL5, *Third ACM Symp. on Principles of Programming Languages*, pp. 185-191.

DIJKSTRA, E. W. (1967). Recursive Programming, in *Programming Systems and Languages*, Saul Rosen (ed.), McGraw-Hill, New York.
GRISWOLD, R. E. (1976). The SL5 Programming Language and Its Use for Goal-Directed Programming, *Proc. of the Fifth Texas Conf. on Computing Systems*, pp. 1-5.
GRISWOLD, R. E. *et al.* (1968). *The SNOBOL4 Programming Language*, second edition, Prentice-Hall, Englewood Cliffs, New Jersey
GRISWOLD, R. E. and HANSON, D. R. (1977). An Overview of SL5, *SIGPLAN Notices*, Vol. 12 No. 4, pp. 40-50.
HANSON, D. R. (1976a). The Syntax and Semantics of SL5, SL5 Project Document S5LD2b, Dept. of Computer Science, The University of Arizona, Tucson, Arizona.
HANSON, D. R. (1976b). Variable Associations in SNOBOL4, *Software—Practice and Experience*, Vol. 6 No. 2, pp. 245-254.
HANSON, D. R. (1976c). *Procedure-Based Linguistic Mechanisms in Programming Languages*, Ph.D. Dissertation, Dept. of Computer Science, The University of Arizona, Tucson, Arizona.
HANSON, D. R. (1976d). A Procedure Mechanism for Backtrack Programming, *Proc. of the ACM Annual Conference*, pp. 401-405.
HANSON, D. R. and GRISWOLD, R. E. (1978). The SL5 Procedure Mechanism, *CACM*, Vol. 21, to appear.
HOARE, C. A. R. (1972). Notes on Data Structuring, *Structured Programming*, Academic Press, London, pp. 127-130.
KERNIGHAN, B. W. and PLAUGHER, P. J. (1976). *Software Tools*, Addison-Wesley, Reading, Mass.
KNUTH, D. E. (1969). *The Art of Computer Programming*, Vol. 2, Seminumerical Algorithms, Addison-Wesley, Reading, Mass., p. 9.
MCILROY, M. D. (1968). Coroutines, Technical Report, Bell Laboratories, Murray Hill, New Jersey.
RITCHIE, D. and THOMPSON, K. (1974). The UNIX Time-Sharing System, *CACM*, Vol. 17 No. 7, pp. 365-375.
VAN WIJNGAARDEN, A. *et al.* (1976). Revised Report on the Algorithmic Language ALGOL 68, *Acta Informatica*, Vol. 5 No. 1, pp. 1-236.
WULF, W. A. *et al.* (1971). Bliss: A Language for Systems Programming, *CACM*, Vol. 14 No. 12, pp. 780-790.

Book reviews

Compiler Construction (An Advanced Course), Second edition, edited by F. L. Bauer and J. Eickel, 1976; 638 pages. (*Springer-Verlag*, DM 24)

This book is a reprint of the course notes used in an *Advanced Course in Compiler Construction*, organised in Germany in March 1974 and repeated in March 1975. The course was presented by a number of persons prominent in the academic community, and covered a wide range of subjects. It is extremely difficult to review a book of this nature concisely, since it consists of many lectures of widely differing quality.

McKeeman's introduction to the course is lucid and interesting even though the lecture deals with material which must surely be present in any undergraduate course in language implementation.

A number of other lectures in this book fall into the same class as this. These are: DeRemer's lecture on formalisms and notation, Waite's lectures on the relationship of languages to machines and on assembly and linkage, McKeeman's lectures on symbol table access methods and on programming language design, Horning's lectures entitled 'What the compiler should tell the user' and 'Structuring compiler development', and Griffiths' lecture on run time storage management.

Some lectures give fairly complete and interesting coverage of their subject matter. These are: Waite's on optimisation techniques, Poole's on portable and adaptable compilers—which also contains useful case study material; and Hill's on run time organisation for ALGOL68.

Griffiths' coverage of $LL(1)$ grammars and analysers gives enough information about $LL(1)$ techniques to enable the reader to construct a parser-generator. Unfortunately, his description of the automatic elimination of left-recursion is confusing, as is his brief introduction to $LL(k)$ grammars.

Horning's description of LR grammars and analysers will also enable the reader to construct a parser-generator based on one of the various refinements of $LR(1)$ techniques.

DeRemer's lecture on lexical analysis covers elementary material. His lecture 'Transformational grammars' deals with a useful method of approaching the specification of translators, and advocates a view of language processing—as tree manipulation—which is extremely fruitful.

Koster's lecture on two level grammars is obscure. The material could easily have been presented less formally in about half the space. This reviewer would have preferred such a treatment to be augmented by an indication of the relevance to the compiler writer of two level grammars.

Waite's lecture on code generation is inadequate. Its first section is written in such an abstract style as to be almost incomprehensible to those who are not familiar with the model of code generation which he presents. Furthermore, his terminology changes from paragraph to paragraph—which adds confusion.

Koster's series of lectures on using the CDL compiler compiler demonstrates the unwieldliness of CDL, rather than the utility of compiler compilers. A newcomer to compiler compilers could, after reading these lectures, be forgiven for concluding that such systems are of little more use in the building of compilers than 'ordinary' high level languages. This is far from being the case. It is unfortunate that the latest widely known review paper on automatic compiler generation—that of Feldman and Gries—dates back to 1968. Unfortunately, Griffiths' introductory lecture on compiler compilers doesn't remedy this deficiency in the literature.

In summary, this book does not amount to 'An Advanced Course in Compiler Construction', despite the excellence of a few of the lectures in it.

B. A. SUFRIN (Colchester)

Reference

Feldman and Gries, (1968). Translator Writing Systems, *CACM*, Vol 11 No 2

Online Review, Vol. 1 No. 1, published by *Learned Information*, Oxford, $45 a year

In the context of this journal the term online describes the facility whereby data bases may be accessed remotely via terminals for a tariff.

In the UK these systems are rare; however in the United States many independent information services are in operation. A major use of these systems is by researchers interested in articles published by others on related topics. For this reason it is likely that usage will continue to be mainly by libraries, research institutes and major organisations.

Much emphasis is given to the number of items available from various data bases, frequency of update and the ease with which the required items may be accessed. One paper attempts to identify which features of a certain interrogation language are the most useful. There is some justification for hoping that as these systems spread only a handful of interrogation languages remain, allowing users easily to use a number of different information services.

A journal dedicated to online systems must at present be addressed to a limited audience. However several times contributors emphasised the need for just such a journal. It will be interesting to see how it and information services develop.

E. GILDERSLEEVES (Norwich)