Computer Languages, Vol. 3, pp. 181-192
© Pergamon Press Ltd 1978. Printed in Great Britain

0096-0551/78/0801-0181$02.00/0

DATA STRUCTURES IN SL5*

DAVID R. HANSON

Department of Computer Science, The University of Arizona, Tucson, AZ 85721, U.S.A.

(*Received 28 October 1977; in revised form 7 February 1978; received for publication 6 March 1978*)

Abstract—The procedural approach to data structures used in the SL5 programming language is described. The SL5 procedure mechanism forms the basis for this approach to data structures by treating procedures and their activation records (environments) as data objects and by decomposing the traditionally atomic operation of procedure invocation into more elementary components. The basic idea is that environments, since they are data objects, can also be used as data structures. The result is a single unified linguistic mechanism for constructing both procedures and data structures. Several examples are given.

Programming languages Data structures SNOBOL4 SL5 Coroutines

INTRODUCTION

Two OF the major components of any programming language are the procedure mechanism and the datatype mechanism. Most high-level languages provide a set of built-in procedures and a facility for including programmer-defined procedures. Most languages also provide a set of built-in datatypes, but facilities for defining additional datatypes are found in only the more recent languages.

Following early work in datatype definition mechanisms [1–4], Pascal [5] was the first widely available language that provided and emphasized facilities for defining additional datatypes. Although Simula [6] appeared earlier, the importance of its class facility for datatype definition was not immediately recognized. SNOBOL4 [7] also preceeded Pascal, but the emphasis on string pattern matching and the "typeless" nature of the language obscured the presence of its datatype definition facilities.

More recent research has been concerned with facilities that permit the definition of datatypes and data structures. Some of this work is performed by designing new languages. CLU [8] and Alphard [9] are examples of this approach. Other work (cf. [10, 11]) has concentrated on language-independent mechanisms. The basic approach is to view an datatype as a set of values and a collection of procedures that operate on these values.

Another way to view a data structure is as a form of procedure. This view leads to the notion of "functional", or "procedural", data structures. The distinction between procedures and data structures is blurred in this approach, resulting in a single concept upon which to base language facilities. An attractive property of this approach is that the resulting facilities are usually quite simple yet very flexible. Procedural data structures are frequently mentioned in the description of applicative programming languages [12], in some AI languages [13], and in languages based on the lambda calculus [14].

This paper describes the procedural approach to data structures used in SL5 [15]. The SL5 procedure mechanism [16] encourages this approach by treating procedures and their activation records as data objects and by decomposing the traditionally atomic operation of procedure invocation into more elementary operations. As a result, procedures may be used as recursive functions or as coroutines. The basic idea is that SL5 procedure activations can also be used as data structures.

Most other approaches to procedural data structures are based on recursive functions. Recursive functions are often unsuitable for use as data structures because pure recursive functions do not have "memory", which is frequently required to implement complex

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

data structures. In the absence of an alternative, global variables are usually used for this purpose. The use of SL5 procedures as coroutines leads to the use of coroutines as data structures. This is the distinguishing characteristic of the SL5 approach. Because coroutines "remember" their state from one activation to the next, it is unnecessary to resort to global variables or other static mechanisms to obtain the effect of "memory".

The remainder of this paper describes SL5, its procedure mechanism, and the use of coroutines as data structures. The description of SL5 given in this paper reflects the current state of an implementation for the DECsystem-10.

THE SL 5 PROGRAMMING LANGUAGE

SL5 is a language for investigating control mechanisms and advanced string and structure processing. SL5 derives many of its characteristics from SNOBOL4, but there are several important differences. For example, SL5 is an expression-oriented language and includes most of the "modern" control structures. SL5 has no type declarations. A variety of datatypes are provided along with the appropriate runtime coercions. A brief overview of SL5 follows; additional details are given in [15–17].

Expressions return *results*, which are composed of two parts: a *value* and a *signal*. The value is used in the traditional fashion. The signal, which is a small non-negative integer, is used to control the flow of execution. For the most part, two signal values are used: 1 meaning "success", and 0 meaning "failure". Signals are used to drive control structures much like Boolean values. Some examples of control structures are (brackets enclose optional phrases)

if e_1 **then** e_2 [**else** e_3]
while e_1 **do** e_2
for v [**from** e_1] [**to** e_2] [**by** e_3] [**while** e_4] **do** e_5
case e **of** l_1: e_1; ... ; l_n : e_n; [**default**: e_{n+1}] **end**

These expressions behave in the conventional manner except that execution is controlled by signals rather than by values. There are several control expressions that are not typical:

e_1 **or** e_2
e_1 **and** e_2
repeat e

The result of the **or** expression is e_1 if e_1 succeeds, otherwise the result is e_2. The **and** expression is complementary: its result is e_1 if e_1 fails, otherwise it is e_2. The **repeat** expression repeatedly evaluates e until e fails.

SL5 includes the usual scalar types, such as integers and real numbers. In addition, strings are scalar types. That is, a string can be manipulated as a single entity; it is not an array of characters. There are built-in functions for accessing the individual characters of a string, and a comprehensive string scanning facility [18].

Lists permit an aggregate of values of any type to be manipulated as a single entity. A list is specified by enclosing its elements within square brackets, e.g.

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

The elements of a list are accessed using the infix operator ! For example, x!2 refers to the second element in the list assigned to x. The indices of a list are 1 to the number of elements in the list. If the index is outside this range, the reference fails. An alternative method of building a list is *list* (n, x), which returns a list of n elements each initialized to x.

PROCEDURES

SL5 procedures are data objects that are created at runtime by expressions of the form

procedure (*formal arguments*)

```
    declarations
    body
end
```

The expression **return** *e* returns *e* as the result of the procedure. Explicit signalling is provided by **succeed** *e* and **fail** *e*. If *e* is omitted, the null string is assumed.

Procedure invocation may be decomposed, at the source-language level, into three components: environment creation, argument binding, and procedure resumption.

The **create** expression takes a procedure as its argument and creates an environment for the execution of the procedure. An environment is an activation record for the procedure, and is a data object capable of being manipulated in the source language. For example, the expression

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

assigns to y an environment for the procedure p.

The **with** expression binds the actual arguments to the formal arguments in an environment. For example, the expression

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

binds the values of a_1–a_n to the formal arguments of p. Unless otherwise specified, argument transmission is "by value". Other modes of argument transmission are provided [16]. Null strings are supplied for omitted arguments, and excess arguments are ignored.

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. When the environment is subsequently reactivated, **resume** returns its result. The functional notation $f(e_1, \ldots, e_n)$ may be used and is equivalent to

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

An environment for a procedure contains the storage for all of the identifiers appearing 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 two 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 **create**, **with**, **resume**, and **return**. An environment whose execution caused the creation of an environment is called the *creator* for that environment. An environment's *resumer* is the environment that caused its most recent resumption via **resume**.

These definitions 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 **resume**, which explicitly indicates the environment to be resumed, establishes the resumer: **return** always returns control to the resumer of the environment in which it is executed.

The life span of an environment is completely determined by its accessibility at the source-language level. This is a consequence of defining an environment as a data object. There is 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 **return**. This is in marked contrast to most other languages having procedure mechanisms based on recursive functions where an activation record cannot outlive the expression that caused its creation.

Most of the procedure activation expressions described above 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.

Declarations are used to determine the scope of identifiers, not their type. Identifiers may be declared either **public** or **private**. Private identifiers are accessible only to the procedure in which they are declared and 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 within the dynamic scope 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 by reference 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 procedures are used in the usual recursive fashion, this is equivalent to the kind of dynamic scope used in SNOBOL4 and LISP.

DATA STRUCTURES

The SL5 approach to data structures is motivated by, and is a consequence of, its procedure mechanism. The basic idea is that environments, as source-language data objects, can be used to fabricate data structures. Thus data structures and procedures are defined using a single mechanism, since in many situations, a portion of data structure is procedural in nature, and vice versa. As described below, data structures *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.

To some degree, the use of environments as data structures is accomplished by observing a few programming conventions. The conventions described below have been used to implement of a wide variety of frequently used data structures. It is possible, however, to use whatever conventions best suit the specific application. This flexibility is a consequence of treating environments as data objects.

Using environments as records

The identifiers in an environment can be viewed as attributes of that environment. Another way to access them is by the environment reference operator, which is indicated by an infix dot. The expression *e.x* refers to the identifier *x* in the environment *e*. This operator provides a means of accessing the values of identifiers in an environment independent of scoping conventions, and permits accessing the identifiers in an inactive environment. This operator also permits environments to be used as records. 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 are used as records. 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: = "Smith";
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** ("Smith", 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
```

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

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

a new employee can be entered as follows:

```
newhire: = create employee with ("Smith", 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 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);
  return
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 ("Smith", 32, 12000.00);
resume newhire;
```

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

It would be simpler if a record could be created by executing an expression such as

```
newhire: = employee ("Smith", 32, 12000.00)
```

If this is done, however, *newhire* is not assigned the desired environment, but rather the null string that is returned. This problem is remedied by having the procedure return its own environment instead of the null string. This is accomplished by using the built-in function *self* (), which returns the environment in which it is evaluated. Thus, *employee* may be rewritten as

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

```
    return self()
  end
```

Thus, the general form for a procedure whose environments are intended to be used as records is

```
  procedure (...)
    ...initialization part...
    return self()
  end
```

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 above, 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 techniques described in this section. Lists themselves do not provide much more than 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. 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*:

```
  x: = procedure (a_1,..., a_n)
    ... initialization part...
    return self();
    ... accessor...
  end
```

After its creation and initialization, an environment for x representing the data structure is resumed with 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. The decomposition of procedure invocation, described above, makes this scheme possible. Notice that the use of y does not require a knowledge of how a two-dimensional array is implemented or even that y is an 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, the arguments are used to pass information that is required for initialization. When the environment is subsequently resumed to access an element, the arguments are used to obtain the values of the subscripts. The initialization part is executed only once whereas the accessor may be executed many times.

As a simple example, consider the implementation of vectors with arbitrary bounds.

```
  vector: = procedure (a, b) private lb, ub, v;
    v: = list (b - a + 1);
    lb: = a;
    ub: = b;
    return self();
    repeat
      return v! (a - lb + 1)
  end
```

An instance of a vector is returned by an expression such as

$$x := vector\ (-2, 10)$$

which specifies a vector with a lower bound of −2 and 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. In an expression such as $x(k)$, x is resumed with the argument k. If k is within the bounds of the vector, the expression $a - lb + 1$ evaluates to a valid index for v, and the appropriate element is returned. If k is outside the bounds of the vector, the list indexing operation fails causing the accessor to fail. Note that the action to be taken if the subscript is out of range can be defined in SL5.

Neither the **return** expression nor the ! operator dereference their argument. This permits expressions that access elements of a vector to be used in contexts requiring a variable, for example, in an assignment expression.

Other versions of *vector* are possible. For example, to avoid the dual use of the arguments, or the use of two arguments for initialization and one for accessing, the following version can be used.

```
vector: = procedure (public lb, public ub)
  return (create procedure (i) private v;
    v: = list (ub − lb + 1);
    repeat
      return v! (i − lb + 1)
    end)
end
```

In this implementation, the accessor is isolated in a separate unnamed procedure and an environment for that procedure is returned as the value of vector. The use of **public** identifiers in *vector* insures that the nonlocal identifiers by the same name in the accessor will refer to the correct environment for *vector*. In addition, the list that houses the vector elements is not allocated until the first time the vector is accessed.

These implementations are only two of the many possible ways to achieve the desired result. Other implementations may also be used, e.g. for sparse vectors the elements could be stored in a linked list. Conversely, very large vectors can be accommodated by using secondary storage for portions of the vector. The use of vectors is independent of their implementation and is accomplished by resuming the environment returned by *vector*.

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 complex data structures require the use of both forms of access. In the implementations of *vector* given above, the attributes *lb* and *ub* specify the bounds of the vector. If x has been assigned a vector, the expressions *x.lb* and *x.ub* can be used to refer to the lower and upper bounds of x, respectively.

EXAMPLES

Stacks

A common example of a datatype and its associated operations is a stack. This kind of methodology can be achieved in SL5 by defining the operations during initialization. A stack p is an environment with the following operations: *p.push*(x) pushes x onto p and returns x; *p.pop*() returns the top element of p and pops p, or fails if p is empty; and *p.top*() returns the top element of p, or fails if p is empty.

The storage for the stack is represented using a singly linked list of records called nodes that have two fields, *value* and *link*.

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

```
        repeat {
          stk: = node (x. stk):
          return x
          }
      end:
      pop: = create procedure private t:
        repeat
          if ident (stk) then
            fail
          else {
            t: = stk.value:
            stk: = stk.link:
            return t
            }
      end:
      top: = create procedure
        repeat
          if ident (stk) then
            fail
          else return stk.value
      end:
      return self():
    end
```

(*ident* (*x*. *y*) succeeds if *x* is identical to *y* and fails otherwise. Null strings are supplied for omitted arguments.)

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 so that the nonlocal identifier *stk* appearing in each procedure refers to the public identifier *stk* in *stack*.

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

Tables

The table. which is an associative store facility. is one of the more useful built-in datatypes in SNOBOL4. 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.

In the implementation given below. tables behave like SNOBOL4 tables. If a reference is made to a nonexistent element. one is created and given the default value specified when the table is created. This action can be defined in SL5. whereas in SNOBOL4 it is fixed by the implementation. The elements in the table are stored as a singly linked list of records. A new element is inserted at the head of the list. The elements of the linked list are records defined by

```
    entry: = procedure (index. value. next)
      return self():
    end
```

Tables are defined as follows.

```
    table: = procedure (x) private init. tlist. t:
        init: = x:
        tlist: = "":
```

```
        return self():
        repeat {
          t: = tlist:
          while differ(t) and differ(t.index, x) do
            t: = t.next:
          if ident(t) then {
            t: = entry (x, init, tlist): #  add a new element
            tlist: = t
            }:
          return t.value
          }
    end
```

(*differ* is the complement of *ident*.)

Of course, this implementation of tables is not very efficient if the number of entries becomes large. But the implementation may be changed without affecting the behavior of tables. And 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 accessor might be written 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 representation of a data structure does not have to remain constant throughout program execution. Only the behavior 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 elements 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, hashno;
        init: = x:
        tlist: = "":
        n: = 0:
        buckets: = "":
        hashno: = procedure (x)
          return length(datatype(x))
        end:
        return self():
        while n < 25 do { # access as singly linked list
          t: = tlist:
          while differ(t) and differ(t.index, x) do
            t: = t.next;
          if ident(t) then {
            t: = entry(x, init, tlist):
            tlist: = t:
            n: = n + 1
            }:
          return t.value
          }:
        buckets: = vector (1, 37): # re-organize into a hash table
        while differ (tlist) do {
            h: = remdr (hashno (tlist.index), 37) + 1;
            t: = tlist.next
        tlist.next: = buckets(h);
        buckets(h): = tlist;
        tlist: = t
        };
      repeat { # access as a hash table
        h: = remdr(hashno(x), 37) + 1;
        t: = buckets(h);
        while differ(t) and differ(t.index, x) do
          t: = t.next;
        if ident(t) then {
          t: = entry(x, init, buckets(h));
          buckets(h): = t;
          n: = n + 1
          };
        return t.value
        }
   end
```

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

Sequencing through the elements of a data structure

Tables are convenient because of their associative nature. There is no means, however, to sequence through all the elements without knowing all the subscripts. Such an operation is needed, for example, to print the elements. This problem is resolved in SNOBOL4 by providing a built-in function that converts a table to a two-dimensional array. A table is converted to an array because an array has a well-defined method of sequencing through every element. 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 elements.

This problem arises whenever it is necessary to sequence through all the elements of a data structure in which the elements are conceptually unordered [11]. Sequencing through the elements in a set is another example. The usual solution to this problem is to map the data structure into an array, as in SNOBOL4, or to reveal to the user enough implementation details of the data structure to permit sequencing through all the elements by explicitly manipulating the underlying representation. Because SL5 procedures can be used as coroutines, it is possible to include a procedure to perform this sequencing operation as a part of the data structure. For example, the following expression, when included in the initialization for tables, assigns to *seq* an environment that returns the value of the next element in the table each time the environment is resumed, failing after all of the elements have been returned. In addition, the identifiers *tlist* and *buckets* in *table* must be declared *public*. (This version of *seq* does not handle the case where elements are being added to the table while sequencing through all of the elements.)

```
   seq: = create procedure private p, i;
     if ident (buckets) then {
       p: = tlist; # table is a linked list
       while differ(p) do {
         return p.value;
         p: = p.next
         }
       }
     else # table is a hash table
```

```
        for i from buckets.lb to buckets.ub do {
          p: = buckets(i);
          while differ(p) do {
            return p.value;
            p: = p.next
            }
          };
      fail
    end
```

The point is that the sequencing operation can be provided as an attribute of the data structure. To illustrate the use of *seq*, assume that *count* is a table of integers. The sum of all of the elements is computed by

```
sum: = 0;
next: = create count.seq;
while x: = resume next do
  sum: = sum + x;
```

This technique suggests that *all* data structures should contain a *seq* attribute, provided that such an operation is meaningful. If this requirement is adopted, syntactic structures can be provided to make use of the *seq* attribute. For instance, the above sequence might be written as

```
sum: = 0;
foreach x in count do
  sum: = sum + x;
```

This scheme is similar to that used for the Alphard iteration constructs [9], which allow the definition of a function to control sequencing. SAIL [19] provides a similar construct for its associative storage facility. But the SAIL construct is defined for only a built-in data structure whereas SL5 permits the definition of both the data structure and its sequencing operation.

CONCLUSIONS

This paper has described a general procedure mechanism and illustrated its use for implementing procedural data structures. The procedure mechanism was designed to extend the domain of applicability of procedural mechanisms beyond that of recursive functions. The major differences between the SL5 mechanism and other procedure mechanisms are that procedures and their environments are data objects and procedure invocation is decomposed into separate operations. These differences are what permit environments to be used as data structures.

The string scanning facilities of SL5 [18] illustrate another use of environments as data structures. 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 at the source-language level. This is a prime example of an application in which SL5 permits the realization of a data structure having procedural components.

The uses of procedures and data structures often overlap. The procedural approach provides the flexibility necessary 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 data structures can be defined 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 encourages the implementation to be derived from the abstract characteristics of the data structure instead of the abstract characteristics being derived from knowledge of the implementation of specific 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 that taken in extensible pattern matching in SNOBOL4 [20] 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.

Although the procedural approach to data structures has been described within the framework of SL5, the approach is not limited to that language. The essential characteristics are the treatment of environments as data objects and the decomposition of procedure invocation into more elementary operations. Provided that these characteristics are maintained, this scheme can be integrated into languages with strong typing and static scope rules, thereby gaining the protection mechanisms and execution efficiencies attainable in such languages. Current work is focusing on ways of accomplishing this integration without sacrificing the flexibility that makes the SL5 approach so attractive.

Acknowledgements—I would like to thank Dianne E. Britton, Frederick C. Druseikis, John T. Korb, and Ralph E. Griswold, for the many discussions that greatly contributed to this work. The perceptive comments of the referees were also very helpful.

REFERENCES

1. B. Galler and A. J. Perlis, A proposal for definitions in ALGOL, *Comm. ACM* 10, 204–219 (1967).
2. C. A. R. Hoare, Record handling, in *Programming Languages* (edited by F. Genuys) pp. 291–347. Academic Press, New York (1968).
3. C. A. R. Hoare, Notes on data structuring, in *Structured Programming*, Academic Press, New York, pp. 83–174 (1972).
4. N. Wirth and H. Weber, EULER—a generalization of ALGOL and its formal definition, *Comm. ACM* 9, 13–23; 89–99 (1966).
5. N. Wirth, The programming language Pascal, *Acta Informatica* 1, 35–63 (1971).
6. G. M. Birtwistle, *SIMULA Begin*, Student Litteratur, Petrocelli/Charter, New York (1973).
7. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*. Prentice-Hall, Englewood Cliffs, N.J. (1971).
8. B. H. Liskov, Abstraction mechanisms in CLU, *Comm. ACM* 20, 564–576 (1977).
9. M. Shaw, W. A. Wulf and R. L. London, Abstraction and verification in Alphard: Defining and specifying iteration and generators, *Comm. ACM* 20, 553–564 (1977).
10. J. G. Mitchell and B. Wegbreit, Schemes: a high level data structuring concept, in *Current Trends in Programming Methodology* (edited by R. T. Yeh). Prentice-Hall, Englewood Cliffs, N.J. (to appear).
11. D. Gries and N. Gehani, Some ideas on data types in high level languages, *Comm. ACM* 20, 414–420 (1977).
12. J. C. Reynolds, Gendanken—a simple typeless language based of the principle of completeness and the reference concept, *Comm. ACM* 13, 308–319 (1970).
13. C. E. Hewitt and B. Smith, Toward a programming apprentice, *IEEE Trans. Software Engng* 1, 26–45 (1975).
14. W. H. Burge, *Recursive Programming Techniques*. Addison-Wesley, Reading, MA (1975).
15. R. E. Griswold and D. R. Hanson, An overview of SL5, *SIGPLAN Notices* 12, 40–50 (1977).
16. D. R. Hanson and R. E. Griswold, The SL5 procedure mechanism, *Comm. ACM* 21, 392–400 (1978).
17. D. R. Hanson, Procedure-based linguistic mechanisms in programming languages, Ph.D. dissertation, University of Arizona, Tucson (1976).
18. R. E. Griswold, String analysis and synthesis in SL5, *Proc. ACM Ann. Conf.* pp. 410–414 (1976).
19. K. A. VanLehn, SAIL User Manual, Technical Report STAN-CS-73-373, Stanford University, Stanford (1973).
20. R. E. Griswold, Extensible pattern matching in SNOBOL4, *Proc. ACM Ann. Conf.* pp. 248–252 (1975).

About the Author—DAVID R. HANSON was born in Oakland, California in 1948. He received the B.S. degree in physics in 1970 from Oregon State University. He received the M.S. degree in Optical Sciences in 1972 and the Ph.D. degree in Computer Science in 1976, both from the University of Arizona.

From 1970 to 1973, he was a Member of the Research Staff at Western Electric Engineering Research Center in Princeton, New Jersey where he did applied research initially in laser physics and then in computer science. From 1976 to 1977, he was an Assistant Professor of Computer Science at Yale University, New Haven, Connecticut. He is presently an Assistant Professor of Computer Science at the University of Arizona. He is also a consultant for ITT Telecommunications Technology Center in Stamford, Connecticut.

His areas of interest include the design and implementation of programming languages, programming methodology, operating systems, and software engineering. Dr. Hanson is a member of IEEE, the Association for Computing Machinery, and the American Physical Society.