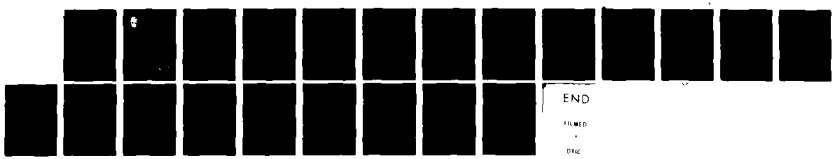
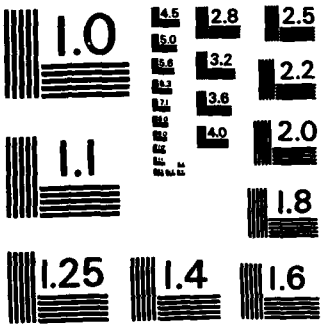
AD-A121 378 IN PRAISE OF PROCEDURES(U) ROYAL SIGNALS AND RADAR 1/1
ESTABLISHMENT MALVERN (ENGLAND) I F CURRIE JUL 82
RSRE-MEMO-3499 DRIC-BR-85316

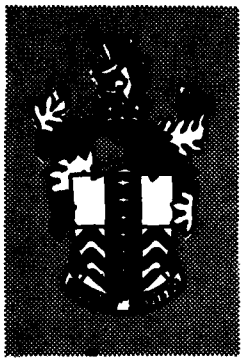
UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

UNLIMITED



RSRE
MEMORANDUM No. 3499

ROYAL SIGNALS & RADAR ESTABLISHMENT

IN PRAISE OF PROCEDURES

Author: Ian F Currie

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 3499

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3499

Title: IN PRAISE OF PROCEDURES

Author: Ian F Currie

Date: July 1982

SUMMARY

The use of procedures vary greatly from one programming language to another. This paper discusses these variations and argues for the use of procedures in a very general fashion; in particular, procedures are an obvious vehicle to provide the data abstraction and encapsulation given in a very limited form by other language constructs such as ADA packages. The implementation of these general procedural values is also discussed with reference to the Flex computer.

UNLIMITED

Title:

In praise of procedures.

Author:

Ian F Currie
Royal Signals and Radar Establishment
St Andrews Rd, Malvern, Worcestershire, England.

Keywords

procedures, abstractions, encapsulations

Abstract

The use of procedures vary greatly from one programming language to another. This paper discusses these variations and argues for the use of procedures in a very general fashion; in particular, procedures are an obvious vehicle to provide the data abstraction and encapsulation given in a very limited form by other language constructs such as ADA packages. The implementation of these general procedural values is also discussed with reference to the Flex computer.

1. Introduction

Most computer languages have some construction producing program structures something like procedures, whether one calls them subroutines, subprograms, functions or whatever. All of them purport to produce some compound action by grouping together more primitive actions; this compound action can then be used repeatedly.

Each language has its own idiosyncracies in the manner in which one can construct and use procedures. Some of the differences are purely cosmetic (e.g. differences in syntax) while others manifest themselves at a much deeper level. In this last category are included such questions as the kind of parameters and results they can have, and whether one can use procedures as data.

In order to find out more about procedures, one has to look for the common features in these diversities and try and describe their properties and the kind of environment in which they exist. Sections 2 to 4 discusses these features, while the remainder of the paper tries to illustrate the uses of those procedures and environments which allow very general parameters and answers.

2. What is a procedure?

Following Strachey and Scott [1,2], let us consider the following sets:

L = { the set of storage locations in some abstract machine },
V = { the set of values storable in these locations in L }.

We can then define a machine state as a function from storage locations to storable values, so that the set of machine states is given by S:

$$S = [\ L \rightarrow V\ ]$$

A procedure can then be defined as a member of a set P (subject to certain constraints to ensure its existence) where:

$$P = [\ X \rightarrow (\ S \rightarrow [\ Y * S\ ]\ )\ ]$$

where X is the parametric domain,
Y is the result domain.
The inclusion of S in this definition is an indication that procedures can cause side-effects in their operation.

The set, V, of storable values is one of the domains which Strachey takes to be characteristic of a given language. In most languages this is identical to the set Y above. Strachey's other characteristic domain is the set D of denotations, i.e. the set of values which can be named in the language, in the simple sense of letting an identifier stand for the value. In most languages D is closely related to the parametric domain of procedures, X.

3. Characteristic domains

A great deal of innocent amusement can be derived from working out the characteristic domains of various languages. Strachey [1] shows the rather baroque characteristics of Algol 60:

```
D =  L          {int, real and bool declarations and value
                 parameters }
   + P          { procedures }
   + ( L* )*    { arrays }
   + W          { parameters called by name }
   + W*         { switches }
   + Q          { strings }
   + J          { labels }

where  P =  [ D* -> ( S -> [ V * S ] ) ]
 and   W =  [ S -> ( [ D + V ] * S ) ]
```

The storable value domain is:

```
V  =  T     { boolean values i.e. TRUE and FALSE }
    + N     { integer values }
    + R     { real values }
```

Note that D and V for Algol 60 are disjoint!

Let us now look at the characteristic domains of Algol 68 Taking a fairly charitable interpretation of the modes and coercions, one can write down the domain of denotations of Algol 68 as:

$$
\begin{aligned}
D = \; & A && \{ \text{primitive types like ints,reals,bools and chars} \} \\
& + L_l && \{ \text{locally generated references} \} \\
& + L_g && \{ \text{globally generated references} \} \\
& + [\, D * D \,] && \{ \text{rows and structures} \} \\
& + [\, D + D \,] && \{ \text{unions} \} \\
& + [\, D \rightarrow (S \rightarrow V * S) \,] && \{ \text{procs} \}
\end{aligned}
$$

A superficial examination of the mode structure of Algol 68 might lead one to suppose that the domain of storable values is the same as D. However, the set V for Algol 68 cannot expressed simply, since although a value may be storable in one location, it may not be storable in another location of the same mode. This arises from the scope restrictions of Algol 68. These restrictions are quite difficult to express; in practice they mean that the only safe storable values are given by:

$$V = A + L_g + [\, V * V \,] + [\, V + V \,] + P'$$

where $P' < [D \rightarrow (S \rightarrow V * S)]$ and each member of P' has a routine-text whose only non-local identifiers are, in fact, global identifiers of the program.

A procedure in Algol 68 effectively has its non-local identifiers built into it and hence, for the procedure value to retain its meaning, the identifiers must also retain their meaning. The designers of Algol 68 envisiaged a stack implementation in which identifiers were associated with locations on the stack. Hence any value which "contained" an identifier could only live as long as the location on the stack corresponding to the identifier was not reused.

It will be seen that limitations of the storable value domains in both Algol 60 and Algol 68, as well as other languages, arise from the presumption of stack based implementations. The Algol 60 restriction in V implies that there is no possibility of a value which can contain elements which may disappear. The analagous restrictions in Algol 68 are only defined in terms of the scope rules; to a large extent, these rules are unenforceable and control of the values is generally done in the informal manner given above.

4. Praiseworthy procedures

If we were not to insist on a stack implementation of Algol 68 and completely relax the scope restrictions, we would get a very pleasant definition of the domains D and V:

$$D = V = A + L + [\ V * V\ ] + [\ V + V\ ] + [\ V \rightarrow (\ S \rightarrow [\ V * S\ ])] \qquad \ldots\ldots\ (1)$$

where all of the references in L are now global and the procedures in [V -> (S -> [V * S])] are implemented so that their action remains valid globally.

The praiseworthy procedures of the title of this paper are the procedures contained in this enlarged set V, i.e. our set of praiseworthy procedures is:

$$P = [\ V \rightarrow (\ S \rightarrow [\ V * S])\ ]$$

In other words, the procedures should be able to accept any values as parameters and deliver any values as result. The aim of this paper is to illustrate the power of this "completion" of the value domain in programming.

The examples given later in the text are written in "scope-free" Algol 68 whose characteristic domains are given in (1) above. The significant thing about these examples is their use of the wide domain of storable values, and not the particular language. This could be more or less any language capable of expressing procedural values.

The notation [V -> (S -> [V * S])] obscures some of the most important properties of procedures. Only a small part of the total state of the machine is involved in the evaluation and result of a procedure and this part is constant for all calls of the procedure. We can say this a little more precisely by considering a typical member, p of P in a sensible language:

$$p = \lambda\, v.\, \lambda\, s.(\ f(v,q),\ \lambda\, 1.(\ 1 \in Z(q)\ |\ e(1)\ |\ s(1)\ )\ )$$

where $q \in V$ gives the non-locals of the procedure,
$f \in V \to V$ gives the answer to the procedure,
and $e \in L \to V$ represents the side effects on the subset of locations derivable from q given by Z.

The functions f and e represent the code or commands of the procedure while q is the set of non-local values of the procedure. Given the general domain of storable values, the non-locals q could be completely inaccessible to any other part of the program, or perhaps shared between only a few procedures. This is the basis of most of the ideas of data abstraction or encapsulation which will now be examined.

5. Abstraction and packages

Consider the following trivial example in our extended Algol 68:

```
PROC make_accumulator =
            STRUCT ( PROC(REAL)VOID sample,
                     PROC STRUCT ( REAL mean,ms) answer) :
( REAL s:=0,ssq:=0; INT n:=0;

  ( ( REAL x) VOID :
    ( s+:=x; ssq+:= x*x; n+:=1 ),

    STRUCT ( REAL mean,ms):
    ( REAL m = s/n;    ( m , ssq/n - m*m ) )
  )

);
```

A call of make_accumulator creates two procedures given in the sample and answer fields of its structure result. The sample procedure accumulates statistics of the sequence of reals given as parameter in successive calls of sample, while a call of the answer procedure gives the mean and mean square of the sequence so far. Clearly one would make a call of make_accumulator for each sequence requiring analysis. Each call produces a different pair of procedures each with a non-local set consisting of the references s, ssq and n created by the call. These references are quite inaccessible elsewhere in the program; indeed, the user of make_accumulator need not know of their existence. Thus the two procedures give an abstraction of the idea of collecting the statistics of a sequence of numbers.

Many systems and languages try to implement some kind of data abstraction or encapsulation, while shying off the difficulties inherent in our praiseworthy procedures. These implementations

usually depend on either creating the abstraction before the program starts and using a conventional stack implementation thereafter, or else including it in the normal stack as a set of declarations of which only some are visible to the rest of the program outside the package. ADA [3], for example, tries to combine both of these approaches with its package construction and defines abstractions as packages which are very like procedures without parameters; they can have private variables and can construct procedures which can be used by other parts of the program. An ADA package with a similar action to make_accumulator is given in the Appendix. Syntax apart, the principal difference between a package and a procedure without parameters, is that one cannot actually call a package. Instead the placement of the text of a package acts as a group of declarations which are then accessible to the range in which the text is placed. In the case where the package is considered as a separate unit, the name of the package must be indicated at the head of the program which can then access the results of an evaluation of the package. Just exactly which evaluation is not always clear; neither is the order of their evaluation defined where more than one is involved

It is clear that packages (a la ADA) can only be accessed in a relatively static manner since they must be evaluated declaratively. For example, if we wished to analyse several sequences using the ADA package in the Appendix, we would probably require several copies of the text of the package (probably using different identifiers). This might be of little importance in this trivial example; however, it is easy to construct examples in which the dynamic aspect of procedures is the essence of the abstraction.

For example, suppose that we wished to implement semaphores in a multi-process, single-processor environment. To do this properly one requires some notion of "process" and "unitary action". Using one's imagination to provide this, a procedure for creating semaphores might be:

```
MODE SEMA = PROC(BOOL)VOID;

PROC make_sema  = ( INT initial ) SEMA:
( INT control := initial;
  PROCESSLIST waiting := empty;

  ( BOOL up ) VOID:
  ( CO unitary action CO
    IF up THEN
      IF waiting /= empty THEN
        PROCESS p = first(waiting);   waiting:=rest(waiting);
        RUN p
      ELSE control +:= 1
      FI
    ELIF control > 0 THEN control -:= 1
    ELSE waiting APPEND current_process;
         wait
    FI
  )
)
```

The semaphore produced by a call of make_sema on an integer n, say, has its control variable initialised to n; a call of the semaphore with a TRUE parameter is equivalent to the normal up operation and the down operation is produced by a FALSE parameter

It is quite clear that no static package can produce these semaphores. Having to have a separate package for each semaphore would be quite intolerable. It might be argued that semaphores (or their equivalent) are defined as primitives in ADA. In that case, let us look at another abstraction, built on semaphores, which implements queues:

```
PROC make_queue = (INT max_q_length)
                  STRUCT ( PROC (ITEM)VOID put, PROC ITEM get):
( ITEMLIST q ;
  SEMA mutex = make_sema(1),
       r = make_sema(max_q_length),
       s = make_sema(0);

  ( (ITEM i) VOID:
    ( r(FALSE);
      mutex(FALSE);  q APPEND i; mutex(TRUE);
      s(TRUE)
    ),
    ITEM:
    ( s(FALSE);
      mutex(FALSE);
      ITEM i = first(q);  q:=rest(q);
      mutex(TRUE);
      r(TRUE);
      i
    )
)
```

We cannot define all abstract machines beforehand, anymore than we can create a sufficient number of examples of a specific machine in advance.

6. Security aspects of procedures

Another important aspect of praiseworthy procedures is their use as a security barrier. As mentioned above, it is possible to create a procedure whose non-locals are not accessible outside the procedure. For example, suppose that I wished to put a password onto the evaluation of a procedure:

```
PROC bind_password = ([]CHAR p, PROC VOID f) PROC([]CHAR)VOID:
( ([]CHAR s)VOID: IF s=p THEN f FI )
```

If we wished the procedure open_cave to be obeyed only if the correct password was known, then the procedure that is made public is the result of:

```
bind_password ("sesame", open_cave)
```

This ease of expression tends to snowball with each layer of abstraction or security barrier that is required. Thus, in a practical system, the procedure open_cave is probably itself the result of such a layer of abstraction, such as the result of binding an interpreter with some find procedure which gives meanings to identifiers:

```
MODE FIND = PROC([]CHAR)VALUE;

PROC bind_find = (FIND f)PROC VOID:
( VOID:interpret(f) )
```

We could have constructed a suitable find procedure by using make_find:

```
PROC make-find = ( DICTIONARY d) FIND:
( ( [] CHAR id) VALUE:
  ( CO finds the VALUE corresponding to id in dictionary d CO
    ....
   )
)
```

Note that we could construct more complicated finds, by combining several such finds:

```
PROC combine_finds = ( [] FIND find_set) FIND:
( ( [] CHAR n ) VALUE:
  ( VALUE v := empty;
    FOR i TO UPB find_set WHILE v = empty DO
     v:= find_set[i](n)
    OD;
    v
   )
)
```

7. Implementation of procedures

Given that praiseworthy procedures are the best things since sliced bread, how does one set about implementing them and why aren't they already in general use? These questions are not unrelated, since the implementation of praiseworthy procedures requires a highly dynamic storage allocation scheme with garbage collection. Remembering our storable value domain:

$$V = A + L + [\ V * V\ ] + [\ V + V\ ] + [\ V \rightarrow (S \rightarrow [V * S])]$$

All the variables in L must be globally accessible and all procedures must have their non-local values bound closely to its code. The size of the non-locals of a procedure is independent of its mode, and thus storage for a procedure must be generated in a heap-like fashion just as surely as that required for variables.

The main reason for the non-adoption of praiseworthy procedures is that garbage collectors have hitherto been regarded as expensive and esoteric toys found only in the ivory castles of Academia. However, with properly designed architecture, the expense of running and maintaining a system which allows praiseworthy procedures is more than balanced by the saving in actually using a system with clear, well defined procedural interfaces which lend themselves naturally to data abstraction.

The Flex computer [4] provides such a suitable architecture; it is micro-programmed to understand procedures and all of its storage allocation (including garbage collection) is implemented in micro-code. Regarding the Flex commands implemented in this micro-code as a language, one can write down its storable value domain:

$$V = A + L + C + [\ V * V\ ] + [\ V + V\ ] + P$$

where C is the set of code bodies containing the commands and constants corresponding to procedure texts and P is our standard set of procedures [V -> (S -> [V * S])]. Each of the sets A, L, C and P are distinguishable by the Flex commands. All of the members of P are created by a close command:

close : [C * V] -> P

i.e. close binds the commands with non-locals to form a procedure. One can obey commands only when they have been bound into a procedure and clearly some of these commands can access the non-locals bound with the commands. The only operation that can be done on a procedure value (other than storing it) is to call it; in other words, possession of a procedural value give access to the code and non-locals of the procedure only in the manner envisaged by the writer of the code. Thus the security and integrity of the system is ensured by making the system interface be entirely procedural in ways similar to those sketched out in the examples above.

The use and construction of Flex procedures is not a restricted system facility; it is part of the repertoire of normal user instructions. The net effect of this is that users have the freedom to impose the same sort of privacy arrangements as is generally only available at the system level in more conventional computers. In effect, every user of the system becomes a system

programmer (without realising it); just as importantly, system programmers, at last, become normal programmers.

8. Conclusion

Procedures, in their full generality, are an extremely powerful weapon in the programmer's armoury. They provide a natural method of abstraction, capable of hiding the workings of a particular implementation of some idea. Once procedures become storable values, they can be divorced from the environment of their construction, so that the protection and security of their internal data-structures and actions can be ensured.

Any design of a language which allows general procedure values will be much simpler than one which pre-supposes a stack-like implementation. This is rather a paradoxical statement since its implementation is likely to be more complicated (except on a friendly architecture like Flex). Nevertheless, it is probably true because the language which lacks proper procedures will try to compensate by introducing other constructions, complicating both its syntax and semantics. The various constructions in ADA involved with packages, hiding parts of declarations and the complicated visibility rules are examples of this. In spite of all this complication, the ADA package system is still a very poor substitute for only one facet of real procedures.

The fact is that, if proper procedures were available, everybody would use them; the advantages in their use in program construction are, in practice, overwhelming. The sole barrier to their adoption lies in the difficulties of their implementation on unsuitable machine architectures; however, in these days of cheap micro-programming, it is as easy to produce a friendly architecture as it is to produce the usual run-of-the-mill architecture found in most commercial processors.

References

1. Strachey, C. Varieties of programming languages. Proc International Computer Symposium, Cuei Foundation, Venice (1972)

2. Scott, D. Outline of a mathematical theory of computation. Proc. 4th Annual Princeton Conference on Information Sciences and Systems (1970).

3. Reference Manual for the ADA programming Language. USA DOD (1980).

4. Currie, I.F. , Foster, J.M. and Edwards, P.W. Flex Firmware, Report no 81009, Royal Signals and Radar Establishment. UK MOD (1981)

Appendix

An ADA package to accumulate the mean and mean square of a sequence.

```
PACKAGE make_accumulator IS
 TYPE stats IS
  RECORD mean,ms : real END RECORD;
 PROCEDURE sample ( x : IN real);
 FUNCTION answer RETURNS stats
END;

PAGKAGE BODY make-accumulator IS
 s : real := 0.0;  ssq : real := 0.0;   n : integer :=0 ;

 PROCEDURE sample ( x : IN real) IS
  BEGIN s := s+x;  ssq := ssq+x*x;  n := n+1  END sample;

  FUNCTION answer RETURNS stats IS
  BEGIN  m : real := s/n;
         RETURNS ( m, ssq/n-m*m )
  END answer

END make-accumulator;
```

Only the names stats,sample and answer are available outside the package.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Memorandum 3499		Unclassified

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
	Royal Signals and Radar Establishment

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

IN PRAISE OF PROCEDURES

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
Currie, I F				

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

HEAD COMPUTING AND GROUND RADAR GROUP

Descriptors (or keywords)

PROCEDURES
ABSTRACTIONS
ENCAPSULATIONS

continue on separate piece of paper

Abstract

The use of procedures vary greatly from one programming language to another. This paper discusses these variations and argues for the use of procedures in a very general fashion; in particular, procedures are an obvious vehicle to provide the data abstraction and encapsulation given in a very limited form by other language constructs such as ADA packages. The implementation of these general procedural values is also discussed with reference to the Flex computer.