SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11 853-866 (1981)

Is Block Structure Necessary?*

DAVID R. HANSON

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

SUMMARY

Block structure is traditionally considered an *a priori* requirement for algorithmic programming languages. Most new languages since Algol-60 have block structure. Reasons exist, however, to omit the general form of block structure—nested procedure definitions in which references to identifiers defined in outer procedures are permitted—from programming languages, especially those intended for systems programming applications. This paper reviews the concept of block structure and considers its advantages and disadvantages. It concludes that, in many cases, a *module* facility is superior to block structure and should be considered in lieu of block structure in future languages.

KEY WORDS Block structure Programming languages Nested procedures Modules Pascal

1. INTRODUCTION

Since Algol-60, 'block structure' has been considered an *a priori* requirement for every new programming language of the Algol variety (e.g. Simula, PL/I, Pascal, etc.). This is true in even the most recently designed languages, such as Ada[1] and the CCITT high-level language for telecommunications applications.[2]

There are, however, reasons why block structure in its most general form should *not* be included in programming languages, especially those intended for systems programming applications. This paper reviews the concept of block structure, considers its advantages and disadvantages, and identifies existing alternatives. The results is a new view of block structure in which it occupies a less exalted position among programming language features. It is concluded that, when judged on an equal basis with other language features, block structure is often simply unnecessary.

Section 2 summarizes the concepts underlying block structure, and Section 3 considers its major advantages. The disadvantages of block structure are detailed in Section 4. An existing alternative to block structure is described in Section 5, and conclusions are offered in Section 6.

2. BLOCK STRUCTURE

The term 'block structure' refers to different features in different languages, which leads to a confusion about its precise meaning. There are essentially three language features for which the term is commonly used: compound statements, blocks and nested procedure definitions.

* This work was supported in part by the National Science Foundation under Grant MCS-7802545.

0038-0644/81/080853-14$01.40 *Received 5 September 1980*

A compound statement is simply a single statement composed of a list of statements, which are usually delimited by brackets of some form. For example, the Pascal compound statement has the form.

```
begin S1; S2;...; Sn end
```

Compound statements differ from both blocks and procedures in that compound statements do not permit the introduction of new identifiers; their sole purpose is statement grouping. As expected, compound statements may be nested.

Blocks are the combination of compound statements with the declaration of identifiers. Blocks in Algol-60 are a typical example and have the form.

```
begin declarations; S1; S2;...; Sn end
```

where *declarations* define new identifiers or redefine identifiers declared in outer blocks. A block defines the scope of identifiers and restricts their visibility—identifiers introduced in *declarations* may be used only within the block. More precisely, these scope rules define the changes in the referencing environment at block entry and exit. These changes are defined to permit their effect to be determined at compile time to permit storage for identifiers declared within the block to be allocated upon entry to the block and deallocated upon exit.[3]

Procedures are blocks with formal arguments where block bodies are replaced by procedure definitions. The Pascal form is

```
procedure (formal arguments...)
  declarations;
  procedure definitions;
  begin
    S1; S2;...;Sn
  end
```

The important difference between blocks and procedures is that blocks are executed only when encountered at the site of their definition, whereas procedures are parameterized and may be invoked from any point within the scope of their names.

Putting these features in terms of actual languages, C,[4] prior to 1978, had only compound statements and unnested procedures;* Pascal has compound statements and nested procedures; and Algol-60, Algol-68, PL/1 and Ada have blocks, compound statements and nested procedures. Almost all languages with nested procedures permit so-called 'up-level addressing'—statements within nested procedures may refer to identifiers declared in outer procedures. A counterexample is the early DEC-10 version of Bliss.[5] It supported two kinds of nested procedures, one with and one without up-level addressing. Only the former kind appear in subsequent versions, however.

Clearly, compound statements are a special case of blocks, and blocks amount to parameterless procedures that are invoked only at their definition site, which substantially simplifies their implementation.[6] For the remainder of this paper, 'block structure' refers to nested procedures (with up-level addressing) since they are the most general of the mechanisms.

3. ADVANTAGES

The major advantage of block structure is the restricted visibility it provides for local identifiers.[7] The visibility, or scope, rules are used to associate an identifier with its

* Blocks were added to C circa 1978.

proper declaration. These rules are based on the static structure of the program and, in most cases, permit all associations to be determined during compilation.

In the case of blocks, identifiers may be declared in close proximity to their use. The visibility restrictions help prevent unintended interference with other uses of the same identifier.

In the case of nested procedures, the scope rules permit, for example, the definition of a procedure to include the definition of subsidiary procedures without concern for interference with other global procedures. This capability facilitates writing procedures and often simplifies their usage. A good example is a procedure for sorting an array as illustrated by the Pascal procedure in Figure 1. Within *sort*, nested procedures (*quick, partition* and *exchange*) embody the implementation details of quicksort. The scope rules restrict the use of these procedures, permitting them to be changed without impacting the use of *sort*. In languages without block structure, *sort* would probably be written as a single procedure to avoid name conflicts. In addition, the array bounds would most likely be required as additional arguments to permit their obvious use in the recursion; block structure permits this usage to be confined to a nested procedure.

Block structure permits efficient use of storage. As a consequence of the scope rules, storage may be shared among local variables of procedures declared at the level. This is the reason that an Algol program, for example, often requires less storage than the equivalent Fortran program. In Algol, storage is required for only the data associated with the *active* procedures, whereas in most implementations of Fortran, storage is required for the data associated with *all* procedures.

```
procedure sort (var a : array 1..1000 of integer);
   procedure quick (lb, ub : 1..1000);
      var k : 1..1000;
      function partition (i, j : integer) : 1..1000;
         var v : integer;
         procedure exchange (var x, y : integer);
            var t : integer;
            begin
               t := x; x := y; y := t
            end;
         begin
            j := j + 1;
            v := a[i];
            while i < j do
               begin
                  i := i + 1;
                  while a[i] < v do i := i + 1;
                  j := j - 1;
                  while a[j] > v do j := j - 1;
                  if i < j then exchange(a[i], a[j])
               end;
            exchange(a[i], a[j]);
            partition := j
         end;
      begin
         k := partition(lb, ub);
         quick(lb, k - 1);
         quick(k + 1, ub)
      end;
   begin
      quick(1, 1000)
   end;
```

Figure 1. An example of nested procedures

4. DISADVANTAGES

The advantages of block structure described in the previous section are the major justifications for including it in a language. There are, however, disadvantages that suggest that block structure should not appear in a language, especially one intended for systems programming.

4.1 Readability

Block structure can make even moderately large programs difficult to read. The difficulty is due to the physical separation of procedure headings from their bodies and the well known problems associated with using non-local variables.[8] This physical separation is a consequence of the 'definition before use' rule in most languages. PL/I does not have this rule, but its compilation requires at least two passes in order to obtain the necessary information.

The result of this physical separation is illustrated to some degree by *sort* (see Figure 1), and to a much greater degree by the program in Figure 2, which is the Banker's algorithm given by Brinch Hansen[9] modified to make it a valid Pascal program.* To read the body of procedures *safe* and *completetransactions* in detail, for example, their headings must be readily accessible. In a large program, the heading and the body may be separated by several pages. The Pascal compiler for the CDC 6000 series computers is another example of this effect; it is very hard to read because of this physical separation. There are places in the compiler, for instance, where over 10 pages separate the procedure heading from the body. Comments such as the one identifying the beginning of the procedure body for *safe* in Figure 2 are used throughout the compiler, but they are of little use in locating declarations for local variables and types. Reading even short programs is tedious if many nested procedures are present.

One common approach to organizing many procedures in a large program is to place them all at the same level in alphabetical order. The success of the software in reference 10 is evidence of the success of this approach. It is interesting to note that some programmers writing large Pascal programs organize their procedures in this way using the *forward* declaration[11] or using what amounts to an 'include' preprocessor.[12] The MAP preprocessor for Pascal[13] is an example of a large Pascal program written using this approach. For programmers like these, block structure is something to avoid in order to improve the readability of their programs.

```
program banker (f, output);
   const
      ncustomers = 200;
      ncurrencies = 5;
   type
      B = 1..ncustomers;
      D = 1..ncurrencies;
      C = array D of integer;
      S = record
            transactions : array B of
               record
                  claim : C;
                  loan : C;
                  completed : boolean
               end;
```

* It is common practice to typeset program material in italics and boldface founts. This practice makes published programs more readable. The issue here, however, is the readability of 'everyday' code that programmers must read. Hence, all program material in this paper is set in a single fount.

```
          capital : C;
          cash : C
       end;
var
   f : file of S;
function safe (currentstate : S) : boolean;
   var
      state : S;
   procedure completetransactions (var state : S);
      var
         customer : B;
         progress : boolean;
      function completionpossible (claim : C; cash : C) : boolean;
         var
            currency : D;
            count : 0..ncurrencies;
         begin
            count := 0;
            for currency := 1 to ncurrencies do
               if claim[currency] > cash[currency] then
                  count := count + 1;
            completionpossible := count = 0
         end;
      procedure returnloan (var loan, cash : C);
         var
            currency : D;
         begin
            for currency := 1 to ncurrencies do
               cash[currency] := cash[currency] + loan[currency]
         end;
      begin { completetransactions }
         with state do
            repeat
               progress := false;
               for customer := 1 to ncustomers do
                     with transactions[customer] do
                        if not completed then
                           if completionpossible(claim, cash) then
                              begin
                                 returnloan(loan, cash);
                                 completed := true;
                                 progress := true
                              end
               until not progress
         end; { completetransactions }
      function alltransactionscompleted (state : S) : boolean;
         begin
            with state do
               alltransactionscompleted := capital <> cash
         end;
      begin { safe }
         state := currentstate;
         completetransactions(state);
         safe := alltransactionscompleted(state)
      end; { safe }
   begin { program banker }
      reset(f);
      if safe(f↑) then
         writeln(' safe')
      else
         writeln(' unsafe')
   end.
```

Figure 2. The banker's algorithm

4.2 Separate Compilation

Separate compilation is essential in any large software project where the designers must work 'in-the-large' (with many modules) as opposed to 'in-the-small' (with several procedures in one module).[14] Block structure does not preclude separate compilation *per se,* but few block-structured languages provide it. Because procedures depend on the environment in which they are nested, only outermost procedures are suitable for separate compilation. Recompilation of an outermost procedure involves recompilation of the procedures nested within it. While this restriction is acceptable in theory, it is generally unworkable in practice because block structure induces the nesting of *entire* programs. For example, the CDC Pascal compiler has only a few outermost procedures.

Some block-structured languages provide separate compilation, but at the cost of severely limiting or not checking the interfaces between separately compiled procedures (e.g. agreement in the types and number of arguments). Techniques exist for *detecting* interface inconsistencies,[15] but they are incomplete and rely on the treatment of external references by the link editor. Languages that support separate compilation and complete checking usually check interfaces during link editing or execution. This approach usually necessitates language changes; an example appears in reference 16. Some languages, such as Mesa,[17, 18] have additional mechanisms for specifying separate compilation. A recent separate compilation technique for Pascal[19] has little syntactic impact on the language, but it does require the division of programs into a *main segment* and any number of *deferred segments*. The main segment contains definitions of all global entities and stubs for the procedures in the deferred segments. Deferred segments can be changed and recompiled at will; recompilation of the main segment, however, requires the recompilation of *all* deferred segments.

The efforts to provide separate compilation for Pascal demonstrate several important points. First, they indicate the importance of separate compilation in large software projects. Second, changes to the language necessary to support separate compilation encourage a usage that diminishes the importance of block structure. Finally, and most importantly, the facilities are non-trivial to implement and explain.

Language features that are especially difficult to implement or explain should be suspect.[20] In most languages, separate compilation is ultimately omitted because of implementation difficulties. Omitting block structure, however, simplifies the implementation and explanation of separate compilation. And, as described below, there are alternatives to block structure that have some of the same benefits and have little impact on the implementation of separate compilation.

4.3 Implementation

The implementation of block structure is well understood. The main problem is uplevel addressing—addressing non-local variables. This problem is simplified by the fact that the static nesting level associated with each variable can be determined during compilation. As a result, only the location of the most recent activation record for each nesting level must be maintained during execution. Some form of display[6, 21] or static chain[3] is commonly used to keep track of the appropriate information.

Both methods are logically equivalent, but the static chain method usually makes references to non-local variables less efficient than local references because it involves traversing a list. For this reason, the display method is favoured in most implementations (and texts). Logically, a display is an array d in which $d[i]$ points to the most recent activation record for a procedure at nesting level *i*. Local and non-local variables are referenced as an offset from the appropriate *d*[*i*] value.

Conceptually, the display does not greatly complicate procedure entry and exit. Most compiler texts (e.g. reference 6) suggest placing *d* in the activation record created for each procedure. This is perhaps the simplest approach, but imposes unnecessary overhead on procedure calls. In the absence of label and procedure parameters, there is another approach: Make *d* global and incrementally change it at procedure entry and exit. Assuming that actual arguments are pushed onto a stack prior to calling a procedure, the entry and exit sequences for a procedure at nesting level *i* have the form.

```
name:   push d[i] onto the stack
        set d[i] to the stack pointer
        adjust the stack pointer to accommodate locals
        .
        .
        set the stack pointer to d[i]
        pop the stack into d[i]
        return
```

The additional overhead of these sequences is acceptable—for some machine architectures. Specifically, it is the location of *d* that is of concern. On a machine that is rich in registers, *d* can be stored in registers, and the sequences outlined above can often be translated one-for-one into machine instructions. For example, on the DEC-10, the sequences are

```
name:   push    sp,di          ;push d[i] onto the stack
        move    di,sp          ; establish new d[i]
        adjsp   sp,nlocals     ;allocate space for locals
        .
        .
        move    sp,di          ;reset stack pointer
        pop     sp,di          ;restore previous d[i]
        popj    sp             ;return
```

References to variables at level *i* are made by simply indexing off *di*.

There are, however, machines that do not possess enough registers for several of them to be dedicated to *d*. The PDP-11, a machine in wide use, is a good example. It has only 6 registers for general use (the other 2 are used for the stack pointer and the program counter). An obvious solution is to place *d* in fixed memory locations. The solution makes the entry and exit sequences look superficially the same, but the additional memory references can make them twice as slow as the case in which *d* is in registers. More importantly, it makes addressing calculations inefficient, which are of paramount importance in high-level language implementation.[22]

A partial solution to the latter problem is to keep the 'top' of *d* in a register, permitting efficient access to local variables. The entry and exit sequences must be revised to update that register, however. In the following sequences for the PDP-11, register *r*5 is used as the top of *d*.

```
name:   mov     di,-(sp)       ;push d[i] onto the stack
        mov     r5,-(sp)       ;push top d onto the stack
        mov     sp,di          ; establish new d[i]
mov     sp,r5           ; establish new top of d
sub     #nlocals,sp     ; allocate space for locals
  .
  .
mov     r5,sp           ; reset stack pointer
mov     (sp)+,r5        ; restore previous top of d
mov     (sp)+,di        ; restore previous d[i]
rts     pc              ; return
```

In these sequences, *r*5 is used as a 'shadow' copy of the appropriate *d*[*i*] value. Maintenance of the 'real' *d*[*i*] is still necessary to insure the correct referencing environment for invocations of other procedures.

The implementation burden imposed by block structure becomes even greater on microprocessors, whose importance is increasing rapidly. Many microprocessors not only lack enough registers but cannot do addressing arithmetic efficiently, making the implementation of block structure awkward at best. On these machines, the code for procedure entry exit and referencing non-local variables tends to be very long, consuming scarce memory. Even extensive optimization cannot greatly change the situation since the lack of registers precludes many typical optimizations.

It is, of course, possible to avoid some overhead by handling outermost procedures differently from nested procedures. However, making the implementation of procedures depend on where in the program they are used violates the principle of transparency,[23] which states that the basic implementation of a language feature be independent of other features and that its explanation provide a clear indication of its computational cost. The danger in not following this principle, especially in languages designed for systems programming, is that programmers may formulate false ideas about the suitability of certain features and may misuse them in the name of efficiency. This tendency helps explain the reluctance of some programmers to use recursive procedures,[24] especially since the inefficiency attributed to recursion is usually due to its interaction with poor implementations of block structure.[25] If outermost procedures are more efficient than nested procedures, programmers will avoid nested procedures altogether, thereby losing what benefits block structure does have to offer.

Much of the effectiveness of a high-level language rests with the effectiveness of its procedure mechanism.[26, 27] The influence of other features—even those whose influence appears minimal—on that mechanism must be considered carefully during design. Block structure is not sufficiently useful to put it above such careful considerations.

4.4 Interference

High-level languages are meant to facilitate programming by people. It is therefore typical, and reasonable, to consider low-level implementation details such as those given above to be less important than higher-level aspects that directly affect the programmer. Block structure can, however, adversely influence language design at the higher level. Specifically, it precludes the use of some potentially useful features, or at least complicates their implementation and explanation.

Typical examples of such features are label and procedure parameters.[6] Correct implementation of procedure parameters, for instance, requires the transmission of all or part of the display in addition to the address of the procedure itself, or the placement of the display in the activation record. This is because procedure parameters can be invoked at points other than those the static scope rules would normally permit. In addition, global optimization is complicated by the undecidability of the reachability problem when procedure parameters and block structure are present.[28] Label parameters present similar problems, but they are less important, of course.

The seemingly simple extension of procedure types in a language like Pascal is another example. Assume 'procedure variables' may be declared like

var *f*: **procedure** (*T*1, *T*2);

which declares *f* to be a variable to which values representing procedures with two arguments of types *T*1 and *T*2 may be assigned, and that assignment and invocation are the only legal operations on procedure values. If *g* is declared as

```
procedure g(x: T1, y: T2)
begin
   ...
end;
```

then after the assignment

f: = *g*;

g could be invoked by *f*(...). (The syntax for invoking parameterless procedures would also have to be changed to require parentheses, i.e. *f*() instead of *f*.)

This feature is quite useful; it can be used, for example, in many table-driven programming techniques. In a block-structured language, however, the use of procedure types must be severely restricted. In the program shown in Figure 3, the assignment to *f* in procedure *c* must be prohibited. Consider the consequences if it is not: The invocation of *a* ultimately results in the invocation of *c* in which *f* is assigned *b*. The second invocation of *b* via *f* will cause a reference to a non-existent *t*. This results because the second invocation of *b* is made without first invoking *a*, which declares *t* and allocates space for it. The assignment to *f* has defeated the scope rules by making *b* available outside of *a*.

Rules restricting the values that can be assigned to *f* might avoid this problem, but make the legal use of such variables depend on where in the program they are

```
begin
   var f: procedure (T1, T2);
   procedure a(...)
      var t: integer;
      procedure b(...)
         procedure c(...)
         begin
            f := b
         end;
      begin {b}
         t := 1;
         c(...)
      end;
   begin  {a}
      b(...)
   end;
   a(...);
   f(...)
end;
```

Figure 3.

referenced. More importantly, since procedure types result in aliasing similar to that for procedure parameters, determining the proper restrictions is likely to be undecidable. Permitting only the assignment of outermost procedures to procedure variables is the only 'safe' restriction. This problem is identical to the 'dangling references' that could occur in Pascal if pointers to local variables were permitted, which is why pointers in that language may refer only to objects in the heap.

5. AN ALTERNATIVE

One of the major purposes of block structure is to hide information, but it is an inadequate mechanism for doing so. The visibility rules are implicit and based only on nesting. The need for an explicit and more general means of controlling visibility has been recognized recently,[29] especially for languages emphasizing data abstractions. The *module* concept of Modula[7] epitomizes this trend; other examples are the encapsulation mechanisms in Concurrent Pascal,[30] CLU,[31] Alphard,[32] Euclid,[33] and Ada.[1]

The module facility in the Y programming language [34] is typical of such facilities.* A module is a collection of scope, data, and procedure declarations. Explicit declarations are required to make identifiers visible outside the module ('exported' identifiers) and to use identifiers that are defined in another module('imported' identifiers). The visibility of all other identifiers is restricted to within the module; there are no implicit visibility assumptions.

Since nesting is not used to restrict visibility, the module contents can be arranged as desired. Figure 4 and 5 show the programs from Figure 1 and 2, respectively, as they would be written in Y.† The revised Banker's algorithm has been packaged as a module in which only *safe* and *S* are exported, since they are clearly the identifiers of interest; the others are only in the implementation of the algorithm.

Modules lack the disadvantages of block structure detailed in Section 4. The linear structure of the code in Figures 4 and 5 makes them easier to read than the nested versions. The *export* declarations explicitly specify, in one place, the entry points to the module. Although not shown in these examples, modules also permit static data to be exported, which cannot be expressed with block structure. Note that the modified versions of *quick* and *partition* (see Figure 4) access the array *a* as a parameter rather than as a non-local variable (see Figure 1). It is debatable, however, whether the implicit accessibility of *a* in the previous version is more readable than the explicit accessibility in the revised version.

Like block structure, modules do not impact separate compilation *per se*. Unlike block structure, however, they do not induce a structure on programs that essentially precludes separate compilation. Experience with separate compilation in Y, for example, which is described in reference 35, suggests that modules encourage programs to be decomposed into manageable, logically independent components with well-defined interfaces. An example is the Y compiler itself; it consists of 14 modules totaling nearly 2700 lines of code, but has only 106 exported identifiers.

Modules impose no overhead on the implementation of procedures. The absence of up-level addressing makes a display or static chain unnecessary. Only a pointer to the

* Y's module facility is simpler than that in most other languages because modules may not be nested. Experience with Y indicates there is little reason for nesting them.

† Y's syntax is similar to Ratfor[10] and its semantics are similar to C.[4] Pascal syntax is used in Figures 4 and 5 to provide a meaningful comparison with Figures 1 and 2.

```
module quicksort
   export sort;
   procedure sort (var a : array 1..1000 of integer);
      begin
         quick(a, 1, 1000)
      end;
   procedure quick (var a : array 1..1000 of integer; lb, ub : 1..1000);
      var k : 1..1000;
      begin
         k := partition(a, lb, ub);
         quick(a, lb, k - 1);
         quick(a, k + 1, ub)
      end;
   function partition (var a : array 1..1000 of integer; i, j : integer) : 1..1000;
      var v : integer;
      begin
         j := j + 1;
         v := a[i];
         while i < j do
            begin
               i := i + 1;
               while a[i] < v do i := i + 1;
               j := j - 1;
               while a[j] > v do j := j - 1;
               if i < j then exchange(a[i], a[j])
            end;
         exchange(a[i], a[j]);
         partition := j
      end;
   procedure exchange (var x, y : integer);
      var t : integer;
      begin
         t := x; x := y; y := t
      end
end { quicksort }
```

Figure 4. Reorganized sort procedure

```
module banker;
   export safe, S;
   const
      ncustomers = 200;
      ncurrencies = 5;
   type
      B = 1..ncustomers;
      D = 1..ncurrencies;
      C = array D of integer;
      S = record
            transactions : array B of
               record
                  claim : C;
                  loan : C;
                  completed : boolean
               end;
            capital : C;
            cash : C
         end;
   function alltransactionscompleted (state : S) : boolean;
      begin
         with state do
            alltransactionscompleted := capital <> cash
      end;
```

```
    function completionpossible (claim : C; cash : C) : boolean;
      var
        currency : D;
        count : 0..ncurrencies;
      begin
        count := 0;
        for currency := 1 to ncurrencies do
          if claim[currency] > cash[currency] then
            count := count + 1;
        completionpossible := count = 0
      end;
    procedure completetransactions (var state : S);
      var
        customer : B;
        progress : boolean;
      begin
        with state do
          repeat
            progress := false;
            for customer := 1 to ncustomers do
              with transactions[customer] do
                if not completed then
                  if completionpossible(claim, cash) then
                    begin
                      returnloan(loan, cash);
                      completed := true;
                       progress := true
                    end
            until not progress
       end;
     procedure returnloan (var loan, cash : C);
       var
         currency : D;
       begin
         for currency := 1 to ncurrencies do
           cash[currency] := cash[currency] + loan[currency]
       end;
     function safe (currentstate : S) : boolean;
       var
         state : S;
       begin
         state := currentstate;
         completetransactions(state);
         safe := alltransactionscompleted(state)
       end
   end {banker}
```

Figure 5. Reorganized banker's algorithm

current activation record, which is used to access arguments and local variables, is required. For example, the Y entry and exit sequences for the PDP-11 are

```
name:   mov     r5,-(sp)        ;save current activation record pointer
        mov     sp,r5           ;establish new activation record pointer
        sub     #nlocals,sp     ; allocate space for locals
        .
        .
        mov     r5,sp           ;reset stack pointer
        mov     (sp)+,r5        ;restore previous activation record pointer
        rts     pc              ;return
```

In these sequences, register *r*5 points to the current activation record. Even on machines with few registers, there is usually one that can be used for a similar purpose. It interesting that two popular systems implementation languages, BCPL[36] and C,[4] use a similar implementation for recursive procedures and neither have block structure as defined in this paper.

The problems with procedure types described in Section 4·4 do not arise in a module-based language like Y. All procedures are outermost procedures and hence can be assigned to procedure variables. Since modules effectively decouple scope from activation, it is even permissible to assign procedures that are not exported. Note that modules do not solve the dangling reference problem associated with pointers to local variables, however, since that problem is an activation rather than scope problem.

6. CONCLUSIONS

The disadvantages of block structure are perhaps not significant when considered individually. Taken collectively however, they at least balance, if not overshadow, the advantages. Block structure should not, therefore, be treated as an *a priori* requirement, but should be considered carefully during language design. Moreover, as described in the previous section, a simple module facility has many of the same advantages of block structure and none of its disadvantages.

Block structure has both compile-time and runtime aspects. One of the attractive aspects of a module facility is that it is strictly a compile-time feature. This not only simplifies its explanation, but insures that it cannot impact the runtime efficiency of other features. For example, it is unnecessary to maintain scope information at runtime.

Some languages that have modules also have block structure. A better design choice—or one that at least deserves consideration—is the inclusion of modules and exclusion of block structure. As this paper has attempted to point out, the result would be a language that is particularly clean, efficient to implement, easy to describe, and that has most of the advantages of block structure but not its disadvantages.

ACKNOWLEDGEMENTS

Insightful comments by Doug Comer, Chris Fraser, and Peter Downey were especially helpful in preparing this paper.

REFERENCES

1. J. D. Ichbiah *et al.*, 'Preliminary ADA reference manual', *SIGPLAN Notices,* **14**, part A (1979).
2. *Proposal for a recommendation for a CCITT high level programming language,* CCITT Study Group XI, Brown Document, 1979.
3. T. W. Pratt, *Programming Languages: Design and Implementation,* Prentice-Hall, Englewood Cliffs, NJ, 1975, Chap. 6.
4. B. W. Kernigham and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliff, NJ, 1978.
5. W. A. Wulf, D. B. Russell and A. N. Habermann, 'BLISS: A language for systems programming', *Comm. ACM,* **14**, 780–790 (1971).
6. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, MA, 1977.
7. N. Wirth, 'Modula: a language for modular multiprogramming', *Software—Practice and Experience,* 7, 3–35 (1977).

8. W. A. Wulf and M. Shaw, 'Global variables considered harmful', *SIGPLAN Notices*, **8**, 28–34 (1973).
9. P. Brinch Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, NJ, 1973, pp. 48–49.
10. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading, MA, 1976.
11. D. Comer, *private communication*, 1979.
12. W. M. Waite, *private communication*, 1978.
13. D. Comer, 'MAP: A Pascal macro preprocessor for large program development', *Software—Practice and Experience*, **9**, 203–209 (1979).
14. F. DeRemer and H. H. Kron, 'programming-in-the-large versus programming-in-the-small', *IEEE Trans. Software Eng.*, **SE-2**, 80–86 (1976).
15. R. G. Hamlet, 'High-level binding with low-level linkers', *Comm. ACM*, **19**, 642–644 (1976).
16. R. B. Keiburtz, W. Barabash and C. R. Hill, 'A type-checking linkage system for Pascal', *Proceedings 3rd International Conference on Software Engineering*, 23–28 (1978).
17. C. M. Geschke, J. H. Morris and E. H. Satterwaite, 'Early experience with Mesa', *Comm. ACM*, **20**, 540–553 (1977).
18. J. G. Mitchell, W. Maybury and R. Sweet, 'Mesa language manual', *Tech. Rep. CSL-78-1*, Xerox PARC, Palo Alto, CA, 1978.
19. R. J. LeBlanc and C. N. Fisher, 'On implementing separate compilation in block-structured languages', *Proceedings SIGPLAN Symposium on Compiler Construction*, Denver, CO, 139–143 (1979).
20. N. Wirth, 'Programming languages: what to demand and how to assess them', *Proceedings of the Symposium on Software Engineering*, Belfast, Ireland, (1976).
21. D. Gries, *Compiler Construction for Digital Computers*, Wiley, New York, 1971.
22. N. Wirth, 'The design of a PASCAL compiler', *Software—Practice and Experience*, **1**, 309–333 (1971).
23. N. Wirth, 'On the design of programming languages', *Proceedings IFIPS 74*, 386–393 (1974).
24. G. L. Steele, 'Debunking the expensive procedure call myth', *Proceedings ACM Annual Conference*, 153–162 (1977).
25. W. A. Wulf, 'Trends in the design and implementation of programming languages', *Computer*, **13**, 14–24 (1980).
26. E. W. Dijkstra, 'The humble programmer', *Comm. ACM*, **15**, 859–866 (1972).
27. D. A. Fisher, 'A survey of control structures in programming languages', *SIGPLAN Notices*, 7, 1–13 (1972).
28. H. Langmaak, 'On correct parameter transmission in higher programming languages', *Acta Informatica*, **2**, 110–142 (1973).
29. E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ, 1976, chap. 10.
30. P. Brinch Hansen, 'The programming language Concurrent Pascal', *IEEE Trans. Software Eng.*, **SE-1**, 199–207 (1975).
31. B. H. Liskov *et al.*, 'Abstraction mechanisms in CLU', *Comm. ACM*, **20**, 564–576 (1977).
32. W. A. Wulf, R. L. London and M. Shaw, 'An introduction to the construction and verification of Alphard programs', *IEEE Trans. Software Eng.*, **SE-2**, 253–265 (1976).
33. B. Lampson *et al.*, 'Report on the programming language Euclid', *SIGPLAN Notices*, **12**, 1–79 (1977).
34. D. R. Hanson, 'The Y programming language', *SIGPLAN Notices*, **16**, 59–68 (1981).
35. D. R. Hanson, 'A simple technique for controlled communication among separately compiled modules', *Software—Practice and Experience*, **9**, 921–924 (1979).
36. M. Richards, 'BCPL: A tool for compiler writing and system programming', *Proceedings AFIPS Sprint Joint Computer Conference*, **34**, 557–566 (1969).

Note added in Proof: Since the submission of this paper, two papers that are particularly relevant to the subject have appeared. The first paper [J. M. Bishop and D. W. Barron, 'Procedure Calling and Structured Architecture', *Computer Journal*, **23**, 115–122 (1980)] discusses in detail the maintenance of the environment for block structured languages in the context of structured computer architecture. The second paper [L. A. Clarke, J. C. Wiledon, and A. L. Wolf, 'Nesting in Ada Programs is for the Birds', *Proc. ACM Symposium on the Ada Programming Language, SIGPLAN Notices*, **11**, 139–145 (1980)] promotes the view that nesting is an anachronism and describes a nest-free style for Ada programs that exploits Ada's novel features.