PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

THE DESIGN OF A SUCCESSOR TO PASCAL

Ken Robinson

Department of Computer Science
University of New South Wales

ABSTRACT

A new programming language intended as a contribution to the development of Pascal is discussed.

* The language supports all of the type constructs of Pascal except the variant record.

* A class construct has been provided to allow the implementation of abstract data types and the hiding of the actual implementation.

* A type union replaces the variant record.

* Parameterized types allow procedures to operate on wider classes of conformable data types than Pascal.

* Predefined data types of "complex" and "string" have been added.

* A new concept called a "selector" has been added.

* The var parameter has been discarded. Parameter modes are now const, value, result and value result.

* Functions are generalized to any type and additionally may return more than one value.

* The assignment statement is replaced by the concurrent assignment statement.

* The if-, while- and repeat-statements are replaced by Dijkstra's if- and do-statements.

* A pipeline facility is provided for communication and synchronization of sequential processes.

---

* This work was initiated while the author was visiting the Computer Studies Group, Department of Mathematics, Southampton University.

1. INTRODUCTION

The design of a new programming language frequently is approached with a great deal of trepidation and announced with apologies for the introduction of "yet another programming language". The trepidation is understandable - there are too many existing examples of bad language design or bad language implementation - yet the apologies are not appropriate - apologies cannot cover up a bad design and only other programmers and time can pass judgement on a language. Perhaps there is a need to apologize in advance for the fact that should the language gain substantial acceptance then it is destined to outstay its original welcome. Despite the existence of so many programming languages this author feels that there are not enough good languages with good compilers widely available on a large number of machines. Pascal is one such language and having now established itself in many areas of computing it can justifiably claim to be the most influential language of the last decade. Pascal is now in the process of being standardized and this inevitably gives rise to much argument. Pascal has probably achieved all that its designer Wirth (1971) ever hoped for it and yet there are many programmers who want and require more. Thus there is a constant demand for extensions to the language, but there are limits to every design and few significant extensions to Pascal are possible without changing the existing language. Indeed Pascal is one language where the "defects" are in many cases the result of deliberate compromises embodied in the language design objectives. It is the opinion of this author that it will prove better to complete the definition of Pascal and accept a standard language which is not significantly extended over the language defined in the Pascal Revised Report, Jensen & Wirth(1975).

This paper discusses the design of a new language intended as a development of Pascal. The language developed is not a straightforward extension of Pascal; it is about as different from Pascal as Pascal was from Algol 60. The paper is intended to be informal and is not a reference manual for the language; that will be released with the implementation of a compiler.

2. DESIGN AIMS

The language is intended to be suitable for use in a wide area of scientific and system programming. The emphasis here is on the suitability to a particular class of programming rather than the simple feasibility of use. For example, Pascal is not an attractive programming language for a large class of scientific problems simply because of the omission of the basic data type "complex".

In general the aims and objectives of Pascal apply with the following being either new or reiterated for emphasis:

* to allow more abstract algorithms and data structures to be realized directly;

* to provide the simpler, yet more powerful, control structures devised by Dijkstra (1975,1976);

* to provide an increased degree of data hiding;

* to provide a facility for parallel programming by implementing a form of communication between processes based on Hoare (1978);

* to design a language which is capable of reasonably efficient execution;

* to provide for rigorous checking of the consistency of the operations on the data, preferably during compilation;

* to provide no implicit default actions;

* to provide for the separate compilation of procedures;

* to design a language in which programmers, particularly students, can concentrate on the design of reliable algorithms and data structures;

* to keep the language "small", that is the language specification should be compact and the number of language constructs few. It would be regarded as an advantage if the language were even more compact than Pascal;

* to reduce the degree to which sequential computation intrudes into the realization of an algorithm, thus reducing the number of transient variables required.

3. LANGUAGE ELEMENTS

Since we are discussing a development of Pascal it follows that many features of Pascal remain and it is assumed that the reader is familar with Pascal. The following is not a comprehensive definition of a complete language but rather a description of those developments which are different from Pascal. Syntax definitions, where given, are in extended BNF (Wirth 1977).

3.1 Types and Type Definitions

Class. A class type, in addition to scalar and structured types, which is similar to the class type of Simula (Dahl et al 1967), Concurrent Pascal (Brinch Hansen 1976) and Hoare(1972), is introduced. The class concept permits a collection of data structures and associated procedures, functions and selectors to realize an abstract data structure. The initialization of a class occurs implicitly at the time of entry to the block in which a class variable is declared, rather than via an explicit request as in Simula and Concurrent Pascal. The class construct permits a programmer to implement a high degree of data hiding. The implementation of a class also determines the degree of access to the data structure possible from outside the class, thus varying degrees of data security can be achieved. Within a class a procedure gives access to an action, a function access to a value and a selector access to a variable.

Figure 1 shows a class implementation of a list-structure. It should be noted that the realization of "list" using pointers cannot be "seen" from outside the class declaration. If a variable is declared to be of type "list"

```
L: list;
```

then only the procedures and functions of the class can be referenced as for example L.prepend(info), L.head, etc.

```
list = class
          listptr = ↑listelement;
          listelement = record
                           content: T;
                           link: listptr
                        end;
          var
             listhead: listptr;
          procedure prepend(x: T);
          begin
             listhead := listptr(x,listhead)
          end;
          function head: T;
                ...
          ...
          begin head := nil end;
```

Figure 1

In the example shown in Figure 2 the definition of "table" uses the type "person" as an abstract type. The actual definition could be a structured type (record) or it could be a class. The only requirement is that "person" possesses a selector named "key". The table could have been implemented in many ways other than an array (e.g. a linear list, a tree) without any difference discernible outside the class. The only requirement is that type "table" provide sufficient procedures, functions and selectors to implement the desired table functions.

```
person = ...;
table = class
           var
              A: array (...) of person;
           procedure insert(...);
           function find(k: keytype): person;
           begin
              "determine i such that A(i).key = k"
              return(A(i))
           end;
                ...
           begin ... end;
```

Figure 2

Notice that the basic structure record itself could be realized by a class. The following types are indistinguishable:

```
record                          class
  field1: t1;                   var
  field2: t2;                     f1: t1;
    ...                           f2: t2;
end                                 ...
                                selector field1 = f1;
                                selector field2 = f2;
                                  ...
                                begin end
```

Parameterized Types

Size Parameters Parameterized types were suggested by Wirth(1975) as a possible method of overcoming the problem, in Pascal, of passing arrays, of the same dimension but of different actual size, as arguments to the same procedure or function. The facility described here is a general form of Wirth's suggestion.

Parameterized type definition:-

type-identifier param-list "=" type

examples

```
realvector(n: integer) = array (1..n) of real
matrix(m,n: integer) = array (1..m,1..n) of real
```

Parameterized type use:-

var-identifier ":" type-identifier parameters

examples

```
avector: realvector(100)
amatrix: matrix(10,20)
procedure sort(value result v: realvector(n))
function product(value a: matrix(m1,n1);
                 value b: matrix(m2,n2)): matrix(m1,n2)
```

Notice that the parameters "n", "m1", n1", "m2" and "n2" declared within the procedure formal parameter section become constants (i.e., read only) whose values are the sizes associated with the actual parameters.

Type Parameters The precedent for this is Pascal itself where file-, set- and array-type declarations have the form

identifier [parameters] "of" type

establishing a type-class.

It is proposed that the following general forms be allowed:

Definition:-

type-identifier [formal-params] "of" formal-type-identifier "=" type

reference:-

type-identifier [actual-params] "of" actual-type-identifier

Examples

```
table of T = class
                var
                   A: array (...) of T;
                ...
```

The Pascal type "file" can be regarded as a parameterized class:

```
file of T = class
                var
                   f: "primitive file" of T;
                procedure get;
                   ...
                procedure put;
                   ...
                ...
```

Predefined Types

Scalar Types

Integer and Boolean As for Pascal.

Character Character sets remain a problem in programming languages as we are still unable, after three decades of computing, to agree on a standard set! The only axioms which are valid for all character encodings are extremely weak:

1. the characters in the subrange '0'..'9' are ordered and contiguous;

2. characters in the subrange 'A'..'Z' (not 'a'..'z' which may not exist!) are ordered.

The intimidation of programming languages, and through them programmers, by the various whims of computer manufacturers is intolerable - character processing in Pascal is crippled severely as a consequence of accepting the above axioms. There is a strong temptation to adopt the most rational of the contending character codes for the definition of the collating sequence of characters for a particular language. If this were done then there is really only one candidate, ASCII (ISO), since none of the others could be described as rational, least of all that unbelievable mess known as EBCDIC culpably loosed on an undeserving world by IBM, and regrettably followed by some other manufacturers.

Implementation notes

1. Character data should be represented internally by at least 8 bits. On machines with 24 bit, 36 bit and 60 bit words characters should be represented by 8 bits, 9 bits and 10 bits respectively. For such machines it will not be possible to utilize any 6 bit "character" handling instructions which might exist.

2. Reading and writing of text-files may require character translation.

3. A distinction should be made between "character" and "byte" data. A byte is an integer subrange (6-bit or 8-bit) and it is expected that there will be a predefined type "byte" which will provide for the processing of special character encodings.

Against the above possible disadvantages must be weighed the advantage of program portability.

Real and Complex To the type "real" (as for Pascal) is added the predefined class "complex" possessing a real component (.r) and an imaginary component (.i)

Sets "Set" is a predefined parameterized type as for Pascal except that "set of char" must be implemented.

Strings A predefined parameterized class

"string" "of" T

is introduced, where T is a scalar type-identifier.

Pascal does not possess a true string type. An array can be used to implement the data-structure known as a string, but a string is not an array (packed or otherwise). It must be possible to reference a string as a whole or any substring of that string. In addition it should be possible to have varying length strings.

Files and Text-files Files as for Pascal except that "file" is a predefined parameterized class. Both sequential and random access file processing procedures will be implemented.

A text-file ("text") is a predefined parameterized class which is a structured file consisting of "pages" which in turn consist of "lines" which in turn consist of "characters". It should be possible to write any data, with the possible exception of pointers, to a text-file. In general reciprocity should exist between input and output statements, meaning that if some data has been written on a text-file by "f.write(x)" then it will be possible to recover that same data by "f.read(x)".

Pipes

"pipe" "of" type-identifier

"Pipe" is a predefined parameterized class for communication between sequential processes. A pipe has two associated procedures "send" and "receive" by which a single item of data may be sent or received through the pipe.

Unions A predefined parameterized class "union" is introduced replacing Pascal's variant record. Pascal provided a means of representing a type-union as a variant record. The greatest weakness of this facility is that it becomes the programmer's responsibility to ensure that the tag setting correctly reflects the current status of the variant.

Example

```
R: record
        case tag: tagtype of
             fool: (x: t1; ... );
             idiot: (y: t2; ... );
             ...
   end
 ...
R.tag := fool;      (1)
R.x := ...;         (2)
R.tag := idiot;     (3)
if R.y = ...        (4)
```

The reference at (4) is obviously invalid yet it may be very difficult to check since it is necessary to determine whether the field "y" has been reset since the last resetting of the tag-field. Notice that the statements at (2), (3) and (4) could be widely separated and still be analogous to the above.

The example illustrates the point that the value of the tag-field alone is not a reliable indicator of the actual variant which has been established. Thus the value of a tag-field cannot be used to determine the validity of a variant reference and this may explain why tag-field checking is omitted from many Pascal compilers.

The problem stems from the division of the action of establishing a variant into two separate actions, the setting of the tag-field (the intended variant) and the setting of the components of that variant.

There is also the disadvantage that unions are forced to be explicitly structured data and thus, for example, a union of scalar types becomes cumbersome.

Examples

```
x: union of (integer,real,char);
x := 1;  (* x.type becomes "integer" *)
if x.type = char ->
 ...
case x.type of
integer ->
real->
char->
 ...
```

Data Constructors It is desirable to be able to initialize variables and to generate instances of a data structure within a program. It is convenient to use the type-identifier to construct an instance of data of that type.

Examples

```
v: integer(1);  (* v is initialized to 1 *)
x := complex(y,z)  (* x.r = y and x.i = z *)
```

In the case of scalar type-identifiers it is convenient to use them also as type transfer functions.

Definition

```
If T1 and T2 are scalar type-identifiers
and x is an element of T1
then T2(x) is an element of T2
such that ord(T2(x)) = ord(x)
```

Examples

```
integer('a') = ord('a')
colour((ord(red)+ord(blue)) div 2)
char(i) is equivalent to Pascal's chr(i)
```

Constant Definitions

constant-definition = constant-identifier "=" constant-expression.

Constant definitions will be allowed to contain constant expressions, that is expressions whose values can be determined at their point of occurrence.

3.2 Procedures, Functions and Selectors

Procedures Similar to Pascal procedures except for parameter modes (see below).

Functions Functions in Pascal are severely restricted in that they may return only scalar or pointer values. This is unfortunate as frequently it is with structured types that functions are most required. For example, in Pascal we could define the types "point" and

"line" as follows:

```
point = record x, y: real end;
line = record p1, p2: point end
```

in order to represent points and lines in plane geometry and then we could establish an algebra of points and lines. However the algebra would be to little avail since we could not define the required functions in Pascal. If the reader thinks that the above example is too esoteric then it will be a revealing exercise to work out in detail the implementation of complex arithmetic in Pascal.

It is necessary, in order that computations on data-structures may proceed at a high level, to allow functions to return values of any type.

Figure 3 shows a complex product function which could be used if the language did not possess "complex" as a basic type.

```
type cmplxunion = union of (integer,real,complex);
function cmplxprod(value a, b: cmplxunion): complex;
begin
      do
           a.type isnt complex -> a := complex(a.0)
      []
           b.type isnt complex -> b := complex(b,0)
      od;
      return(complex(a.r*b.r-a.i*b.i, a.r*b.i+a.i*b.r))
end
```

Figure 3

Traditionally in programming languages there is a lack of uniformity in the treatment of functions which return one value and functions which return more than one value. It is recognized that functions may compute more than one value and this fact may then be used to motivate the concept of the var parameter.

It has been suggested that the semantics of procedures and functions should be describable in terms of a concurrent assignment statement (Hoare 1971, Hoare & Wirth 1973). That is, the effect of a procedure or function (as far as the var parameters and global variables are concerned) is that of a concurrent assignment of expressions to the var parameters and global variables.

Since the proposed language possesses a concurrent assignment statement it seems opportune to realize the semantics rather literally. Functions may return more than one value and may have only const or value parameters. The multiple values may be assigned to variables in a concurrent assignment statement. Figure 4 shows a simple function returning two values.

```
function minmax(const g: realvector(m)): (real,real);
var
      min,max: real;
      i: integer;
begin
      assert(m >= 1);
      min,max := g[1],g[1];
      i := 2;
      do i < m ->
           if g[i] < g[i+1] ->
                do g[i] < min -> min := g[i]
                [] g[i+1] > max -> max := g[i+1]
                od
           [] g[i] >= g[i+1] ->
                do g[i] > max -> max := g[i]
                [] g[i+1] < min -> min := g[i+1]
                od
           fi;
           i :+ 2
      od;
      if i = m ->
           do g[i] < min -> min := g[i]
           [] g[i] > max -> max := g[i]
           od
      [] i <> m -> skip
      fi;
      return(min,max)
end
```

Figure 4. Function returning two values

Selectors A selector returns a variable rather than a value. Such a facility is necessary with classes if access is to be given to actual components of a structure rather than simply a copy of the component. For example, suppose it is required to search "mytable" of class "table", defined in Figure 2, for an entry whose key is 'Fred' and to increment the value in the associated field "age". The function "find" defined in Figure 2 is of no use since it returns the value of the record not the record itself hence

```
mytable.find('Fred').age :+ 1
```

is not correct, in fact it is illegal. However changing "find" to a selector rather than a function makes the above assignment both legal and correct.

Parameters Parameters to procedures may be passed by const, value, result or value result only.

Parameters to functions and selectors may be passed by const or value only.

3.3 Statements

Concurrent Assignment

```
assignment = variable-list ":=" expression-list .
```

For procedural programming languages, like Pascal, variables and assignment are, for better or worse (Backus 1978), their most fundamental concepts. The state of a program is represented by the state of the variables and the assignment statement is the principal mechanism for changing the value of a variable. It is one of the objectives of a high-level programming language to reduce the number of variables required to implement an algorithm, yet languages which implement a simple assignment statement are imposing a very simple sequential mechanism for changing the state of a program: at most the value of one variable may be changed. In a situation where a change of state requires the simultaneous changing of the values of a number a variables this simple sequential mechanism may require extra variables to effect the change. As a consequence of the increase in variables the realization becomes longer and more complex than the algorithm.

Just two simple examples will be given to illustrate the above point. The first is the common case of swapping the values of two variables:

```
a, b := b, a
=> x := a;  a := b;  b := x
```

the second comes from an algorithm for reversing the order of the elements in a linear linked list:

```
r, p, p↑.next := p, p↑.next, r
=> x := r;  y := p;  r := p;  p := p↑.next;  y↑.next := x
```

The concurrent assignment appears to have been first used in CPL (Barron et al 1963) and largely ignored by programming languages since then. Its importance seems not to have been appreciated: as indicated above it is not simply (or even primarily) a notational convenience such as, for example, the multiple assignment of Algol 60.

Other Assignment Operators In addition to the normal assignment operator ":=", other forms are introduced for incrementing ":+", decrementing ":-", etc., the value of a variable. More generally:

```
v :? u is equivalent to v := v ? u for "?" ≠ "="
```

If-statement

```
if-statement = "if" guarded-command-list "fi" .
guarded-command-list = guarded-command {"□" guarded-command} .
guarded-command = guard "->" statement {";" statement} .
guard = Boolean-expression .
```

The statement is due to Dijkstra (1975,1976). The interpretation of the statement is as follows. All the guards are evaluated and of those

guarded commands whose guards are true one command is chosen arbitrarily and executed. At least one guard must be true otherwise the statement aborts.

The if-then-else-statement has attracted much comment on both its asymmetry and the complexity of the implied pre-condition for a statement nested deeply within a nested if-then-else-statement. In many cases the if-then-else-statement is another example of sequential ordering being imposed where none is required. If an ordering is required the above if-statement forces the logical complexity to be explicit since the guards will have to be mutually exclusive.

Case-statement The Pascal case-statement is retained even though it is made redundant by the if-statement, since selection on the basis of a set of constants is likely to be more efficient using a case-statement.

Do-statement

do-statement = "do" guarded-command-list "od" .

This statement is due to Dijkstra (1975,1976). The do-statement is an iterative statement which terminates when all the guards are false. While any guard is true one command whose guard is true is arbitrarily selected and executed. The comments made above on the if-then-else-statement apply similarly to the while-statement of Pascal and no clearer demonstration of the beauty of the do-statement compared with the relative ugliness of the while-statement is likely to be found than the one shown in Figure 5. The expression of Euclid's algorithm seems to have played an important role during Dijkstra's own formulation of his concept of guarded commands.

```
function gcd(a,b: posint): posint;
begin
      do a > b -> a :- b
      [] a < b -> b :- a
      od;
      return(a)
end
```

Figure 5a. Euclid's algorithm expressed with do-statement

```
function gcd(a,b: posint): posint;
begin
      while a <> b
      do begin
           while a > b do a := a-b;
           while a < b do b := b-a
         end;
      gcd := a
end
```

Figure 5b. Euclid's algorithm expressed in Pascal

Programming note It may appear at first that the statement:

```
if B then S
```

should be translated to:

```
if B -> S
[]  not B -> skip
fi
```

but in many cases the following translation is possible:

```
do B -> S od
```

This second construct is stronger than the former in so far as it makes clear that the consequent of the statement implies "not B" (i.e., {B} S {not B}).

Implementation note It is important that the selection of guarded commands for both the if-statement and the do-statement is arbitrary. Requiring all guards to be evaluated and then selecting arbitrarily from those guarded commands whose guards are true will prove inefficient on a sequential machine (but not perhaps on a parallel machine). Alternatively the compiler could order the guarded commands arbitrarily (such that the order is likely to be different for different compilations) and then the guards could be evaluated sequentially with selection of the first command whose guard is true.

For-statement

```
for-statement
 = "for" variable ":=" expression "to" expression "->" statement
 | "for" variable ":=" expression "downto" expression "->" statement
 | "forall" variable "in" set-expression "->" statement .
```

The Pascal for-statement will be retained. In addition a for-statement which executes a statement for all values contained in a set, the order of selection being arbitrary, is included.

With-statement

```
with-statement = "with" variable-list "->" statement .
```

The with-statement becomes a necessity rather than a convenience, as it is in Pascal, since in general it is required in order to hold the variable returned by a selector.

3.4 Parallel Execution

```
par-sequence = "parbegin" proc-call {"||" proc-call} "parend"
```

Processes can communicate through "pipes". The implementation attempted here is different to, but based on, that suggested by Hoare(1978). In Hoare's paper processes communicate directly with a named process rather than through a named pipeline. In a network sense

Hoare's approach is to name the nodes whereas the approach here is to name the edges. There are notational advantages in the latter but a number of implementational disadvantages, one of which is the fact that the processes associated with a pipe may vary with time. In particular the termination of a process cannot be used to signal the end of a pipe. A number of constraints may be required.

Figure 6 is a realization of the procedures "disassemble", "assemble" and process "reformat" discussed in Hoare (1978).

4. SEPARATE COMPILATION

Separate compilation of procedures is a requirement for large programs and for programs which wish to use a library of procedures.

It is intended that the compiler will accept an object known as a "module" where a module consists of a definition and declaration section followed by a sequence of procedure declarations and/or a program declaration.

module = [definitions-and-declarations] {procedure} [program] .

In this way procedures at level 0 (i.e. external to the program) may be compiled separately.

4.1 External Procedures

As a consequence of separate compilation it must be possible to reference procedures which are external to the current module. In addition the facility should exist to reference and use external procedures compiled by another language compiler (in particular FORTRAN and assembler).

5. IMPLEMENTATION AND EXTENSIBILITY

No programming language is better than its implementation: the quality of the compiler and run-time support system has a profound effect on the observed quality of any programming language. This is particularly true of the language being discussed here since it is not an experimental "paper" language; one of the objectives is to produce a Pascal-like language which will be attractive to a large community of programmers. For this reason a full description of the language will not be generally available until a satisfactory implementation exists.

One of the objectives of the implementation is to produce a compiler which is portable enough to become the basis of all implementations. To achieve this the code generation will be made very "visible" and will be documented to aid transportation to various machines. The initial compiler will of course be written in Pascal and it is planned always to have a Pascal version of a compiler available for initial bootstrapping onto any machine for which a Pascal compiler exists.

One of the notable achievements of Pascal is the degree to which a family of Pascal compilers now exists, each compiler clearly based on

the compiler implemented by Ammann (1970,1974). The objective here is to carry the process further and attempt to produce a "standard" compiler. This would seem to have important implications for the standardization of a language: it should be easier to revise a language if at the same time as a revised language standard is produced a revised standard compiler is also produced.

```
type charpipe = pipe of char;
procedure disassemble(result X: charpipe);
var
      cardimage: string(80) of char;
begin
      do not input.eof ->
            input.readln(cardimage);
            for i := 1 to 80 ->
                  if i <= cardimage.length ->
                        X.send(cardimage[i])
                  [] i > cardimage.length ->
                        X.send(' ')
                  fi;
            X.send(' ')
      od;
      X.close
end
```

Figure 6a. Procedure disassemble

```
procedure assemble(value X: charpipe);
var
      c: char;
      lineimage: string(125) of char;
begin
      lineimage := '';
      do not X.end ->
            X.receive(c);
            lineimage.append(c);
            do lineimage.length = 125 ->
                  output.writeln(lineimage);
                  lineimage := ''
            od
      od;
      if lineimage.length = 0 -> skip
      [] lineimage.length <> 0 ->
            output.writeln(lineimage)
      fi
end
```

Figure 6b. Procedure assemble

```
var X: charpipe;
 ...
parbegin disassemble(X) || assemble(X) parend
```

Figure 6c. Process reformat

6. ACKNOWLEDGEMENTS

The design presented here is obviously derived from many sources. Influences are numerous and in many cases unremembered. I apologize to any whose influence has gone unacknowledged.

For stimulating discussions which preceded this design I would like to thank Ian Hayes, David Carrington, Jeffrey Tobias, Greg Rose, Carroll Morgan and Tony Gerber. I wish to thank all my colleagues at Southampton especially John Goodson, Mike Rees and Ralph Elliott for valuable discussions during my visit, even though they remain justifiably sceptical about the language described in this paper. I am grateful to David Barron for the opportunity of visiting the Computer Studies Group and the chance to commence this project. John Goodson must be thanked again for proof-reading this paper, although I am solely responsible for any remaining split infinitives.

7. REFERENCES

Ammann, U. (1970) "Pascal-6000 compiler", ETH Zurich.

Ammann, U. (1974) "The method of structured programming applied to the development of a compiler", International Computing Symp. 1973, (Günther, et al, Eds), 93-99.

Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, F., Strachey, G.S. (1963) "The main features of CPL", Computer Journal 6, 134-143.

Backus, J. (1978) "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", Comm ACM 21,8(August), 613-641.

Brinch Hansen, P. (1976) "The programming language Concurrent Pascal", I.E.E.E. Trans. Software Eng. 1,2 (June), 199-207.

Dahl,O-J., et al. (1967) "SIMULA 67, common base language", Norwegian Computing Centre, Forskningveien, Oslo.

Dijkstra, E.W.D. (1975) "Guarded commands, nondeterminacy and formal derivation of programs", Comm ACM 18,8 (August), 453-457.

Dijkstra, E.W.D. (1976) "A discipline of programming", Prentice-Hall.

Hoare, C.A.R. (1971) "Procedures and parameters; an axiomatic approach", Symposium on Semantics of Algorithmic Languages (E.Engeler, ed), Lecture Notes in Mathematics 188, Springer-Verlag.

Hoare, C.A.R. (1972) "Proof of correctness of data representations", Acta Informatica 1, 271-281.

Hoare, C.A.R. (1978) "Communicating sequential processes", Comm ACM 21,8 (August), 666-677.

Hoare, C.A.R. & Wirth, N. (1973) "An axiomatic definition of Pascal", Acta Informatica2, 335-355.

Jensen,K. & Wirth,N. (1975) "Pascal User Manual and Report", Springer-Verlag.

Wirth, N. (1971) "The programming language Pascal", Acta Informatica 1,1, 35-63.

Wirth, N. (1975) "An assessment of the programming language Pascal", SIGPLAN Notices 10,6 (June), 23-30.

Wirth, N. (1977) "What can we do about the unnecessary diversity of notations for syntactic definitions?", Comm ACM " 20,11(November), 822-823.