PASCALPLUS

USERS MANUAL

Version 1b
August 1978.

A USER MANUAL FOR PASCALPLUS

D. W. Bustard.

Department of Computer Science

The Queen's University of Belfast

(Version 1)

This manual describes several major extensions to the programming language PASCAL. The resulting language, referred to as PASCALPLUS, retains PASCAL as a pure sub-set and PASCAL programs are compiled and executed with the same efficiency.

The language extensions are:-

(i) The envelope structure, which is an aid to program modularisation and data abstraction.

(ii) The process, monitor and condition structures which provide a means for representing multiprocessing programs.

(iii) A simulation monitor, which provides pseudo-time control facilities for multiprocessing programs.

The manual assumes that the reader is completely familiar with the base language PASCAL.

Contents

1. Envelopes

1.1 Introduction

The task of designing a computer program for a given problem, involves successively breaking down the problem into its constituent components and then translating the resulting system into a computer language. In PASCAL, procedures and functions are used for representing the system components but these structures are not always adequate.

Consider, for example, a program which has as one of its components, a procedure whose action is to throw to the top of a new lineprinter page and print the current page number. The program might take the form shown in fig. 1.1

```
PROGRAM PASCALFORM ;
        ...
  VAR
        ...
     THISPAGE : INTEGER ;
        ...
  PROCEDURE NEWPAGE ;
  BEGIN
     PAGE ;
     WRITELN ('PAGE',THISPAGE) ;
     THISPAGE := THISPAGE + 1
  END  (* NEW PAGE *) ;
        ...
BEGIN
        ...
  THISPAGE := 0 ;
  print data
END  (* PASCAL FORM *).
```

fig. 1.1 : page control in PASCAL

This program structure has three significant shortcomings:-

(i) The variable THISPAGE is used only by procedure NEWPAGE but as it has to outlive successive calls on this procedure it must be declared in the block which textually encloses NEWPAGE, i.e. in this case the main program. In this position, THISPAGE can be inspected and modified by all the code of the program rather than just NEWPAGE.

(ii) The declaration of THISPAGE, its initialisation in the body of the program and its use in NEWPAGE may all be physically remote from each other, making the parts difficult to find and the program as a whole more difficult to understand.

(iii) The programmer must take care to ensure that THISPAGE is initialised before the first call on NEWPAGE is made.

The envelope facility of PASCALPLUS can be used to overcome structuring difficulties of this kind.

1.2 The envelope module

The program in fig. 1.2 is a re-written version of the program in fig. 1.1 using an envelope module.

```
PROGRAM  PASCALPLUSFORM ;
            ...
   ENVELOPE MODULE PAGECONTROL ;
      VAR THISPAGE : INTEGER ;
      PROCEDURE * NEWPAGE ;
      BEGIN
         PAGE ;
         WRITELN ('PAGE',THISPAGE) ;
         THISPAGE := THISPAGE + 1
      END  (* NEW PAGE *) ;
   BEGIN
      THISPAGE := 0 ;
      ***
   END  (* PAGE CONTROL *) ;
            ...
BEGIN
            ...
   print data
END  (* PASCALPLUS FORM *) .
```

fig 1.2 : page control in PASCALPLUS

An envelope has the same form as a PASCAL procedure but in application more closely resembles a PASCAL record; it can be thought of as an extension of the record structure where not only data is defined but also the allowed operations on it.

The PAGECONTROL envelope in fig. 1.2 may be likened to a record declaration of the form

```
VAR
   PAGECONTROL : RECORD
                    ...
                 END ;
```

Unlike a record, the declarations within an envelope are not normally accessible outside it. Those declarations which are to be made available to other blocks are indicated by preceding their identifiers with a star.

In the PAGECONTROL envelope only procedure NEWPAGE is marked as externally accessible, with variable THISPAGE being completely invisible and therefore secure.

Starred declarations are referenced via the envelope module name using the dot notation, in the same way as the fields of a PASCAL record are accessed

e.g. PAGECONTROL.NEWPAGE

Again like a record, the repeated use of the access description may be avoided by using a WITH statement.

e.g. WITH PAGECONTROL DO NEWPAGE

The three stars in the body of the PAGECONTROL envelope denote an inner statement. In most applications an inner statement serves to separate the "initialisation" and "finalisation" code of an envelope. The initialisation code is executed prior to the execution of the body of the block in which the envelope instance is declared - the parent block. The finalisation code is executed when the execution of the parent block is complete.

The order of execution of the program in fig. 1.2 is therefore

(i) initialisation code of PAGECONTROL : THISPAGE := 0

(ii) body of the main program : print data etc.

(iii) finalisation code of PAGECONTROL : (in this case nothing)

As an example of the use of finalisation code, the specification of the PAGECONTROL envelope could be extended to include a throw of the final page of output. The body of the envelope would then have the form

```
BEGIN
   THISPAGE := 0 ;
   *** ;
   PAGE
END  (* PAGE CONTROL *)
```

and the body of the program would effectively be

```
BEGIN
   THISPAGE := 0 ;
   ...
   print data etc. ;
   PAGE
END  (* PASCALPLUS FORM *)
```

The inner statement is explained in more detail in section 1.8.

1.3 The envelope type

The PAGECONTROL envelope illustrates the form of an envelope module, or more precisely an envelope which has its definition and declaration combined as only one instance of it is required. Normally an envelope is regarded as a type definition from which several variables, or instances, may be declared.

The syntax definition of an envelope is exactly the same as a PASCAL procedure (fig. 1.3).

```
ENVELOPE name (formal parameter list) ;
   local declarations ;
BEGIN
   body
END ;
```

fig. 1.3 an envelope definition

Envelope instances are declared in a section preceded by the word INSTANCE. The actual parameter list, if any, for an instance is specified after the envelope type name

e.g. I : E (actual parameter list) ;

An envelope must be fully defined before an instance of it can be declared.

Fig. 1.4 is an example of an envelope type. This envelope might be used to batch output to a lineprinter.

```
ENVELOPE OUTPUTCONTROL ;
   TYPE *LINE : PACKED ARRAY [1..120] OF CHAR ;
   VAR   THISPAGE : INTEGER ;
         THISLINE : 0..80 ;
         LP : TEXT ;
   PROCEDURE NEWPAGE ;
   BEGIN
      PAGE (LP) ;
      WRITELN (LP,'PAGE',THISPAGE) ;
      THISPAGE := THISPAGE + 1
   END  (* NEW PAGE *) ;

   PROCEDURE *PRINTLINE (L : LINE) ;
   BEGIN
      IF THISLINE = 80 THEN
      BEGIN NEWPAGE ; THISLINE := 0 END ;
      WRITELN (LP,L) ;
      THISLINE := THISLINE + 1
   END  (* PRINT LINE *)

BEGIN
   REWRITE (LP) ;
   THISPAGE := 0 ; THISLINE := 0 ;
   NEWPAGE ;
   *** ;
   PAGE (LP) ;
   copy file LP to the lineprinter
END  (* OUTPUT CONTROL *) ;
```

fig. 1.4 : illustration of an envelope type

The effect of declaring an instance of the OUTPUTCONTROL envelope in any block is to make available to that block a virtual lineprinter and the single operation PRINTLINE on it. The definition of LINE is also provided by the envelope, so that the block may declare variables of this type and use them when calling PRINTLINE.

e.g.
```
INSTANCE
   REPORT : OUTPUTCONTROL ;
VAR
   L : REPORT.LINE ;
      ...
REPORT.PRINTLINE (L) ;
```

Several instances of the same envelope may be declared in a block

e.g.
```
INSTANCE
   REPORT : OUTPUTCONTROL ;
   TRACEDATA : OUTPUTCONTROL ;
```

These may also be declared in one statement using the same shorthand notation available for PASCAL variables

e.g.
```
REPORT,TRACEDATA : OUTPUTCONTROL ;
```

The parameter lists for envelope instances declared in this way are enumerated after the envelope type name, in the same order as the instance names on the left hand side of the declaration.

e.g.
```
I1,I2 : E (list for I1)(list for I2) ;
```

Instances may also be declared as vectors

e.g.
```
LPS : ARRAY [1..N] OF OUTPUTCONTROL ;
```

The starred identifiers of the envelope are referenced using the normal array access description.

e.g.
```
LPS [M].PRINTLINE (L) ;
```

The parameter list for a vector is made up of the enumerated parameter lists for each element of the vector.

e.g.
```
I : ARRAY [0..1] OF E ((list for I[0])(list for I[1])) ;
```

1.4 Envelope comparison with PASCAL files

The concept of an envelope is not entirely new to PASCAL, as its principle is already embodied in the PASCAL file mechanism.

The effect of declaring a file in any block is to make available to that block both data and an associated set of operations on that data, such as GET, PUT, and EOF. In addition, the block in which the file is declared is implicitly enveloped by the code which opens the file prior to the execution of the block and closes it at the end. A PASCAL file can, in fact, be represented by an envelope of the form shown in fig. 1.5.

```
ENVELOPE FILE ;
   TYPE
     *R : RECORDDESCRIPTION ;
   PROCEDURE *GET (VAR R : RECORDDESCRIPTION) ;
   PROCEDURE *PUT (R : RECORDDESCRIPTION) ;
   FUNCTION *EOF : BOOLEAN ;
            ...
BEGIN
   open file ;
   *** ;
   close file
END  (* FILE *) ;
```

fig. 1.5 : a PASCAL file represented by an envelope

It should be noted that files with differing record descriptions would require separate envelope definitions.

1.5 Envelope scope rules

The interface provided by the OUTPUTCONTROL envelope in fig. 1.4, consisted of a procedure and a type declaration. In general, any local identifier of an envelope may be made available to other blocks. It is considered to be undesirable though, to allow local data to be modified by the code outside the envelope. This principle is enforced by defining starred data of an envelope to be read-only to the code outside the envelope.

Similarly, it is considered undesirable to allow an envelope to modify directly, external data within its scope and so data in this position is defined to be read-only to the code of the envelope.

A side effect of defining interface data to be read-only is that an envelope can be used to extend the very limited constant facilities provided by PASCAL.

For example, consider a program which takes the date from the machine on which it is running and then determines and prints the day of the week. The day might be represented by an enumerated type

```
DAY = (MON,TUES,WED,THURS,FRI,SAT,SUN) ;
```

The corresponding names to be printed would be held in an array

```
DAYNAME : ARRAY [DAY] OF PACKED ARRAY [1..9] OF CHAR ;
```

and the array would be set up by a series of assignments of the form

```
DAYNAME [MON] := 'MONDAY   ' ;
```

Although DAYNAME is strictly a constant, PASCAL does not provide any means for indicating this or of protecting the data from modification following its initial setting.

If the data is declared in an envelope module as shown in fig. 1.6, it can be given the necessary protection.

It should be noted that there is no run-time overhead associated with protecting data in this way.

```
ENVELOPE MODULE DAYCONTROL ;
   TYPE
      *DAY = (*MON,*TUES,*WED,*THURS,*FRI,*SAT,*SUN) ;
   VAR
      *DAYNAME : ARRAY [DAY] OF PACKED ARRAY [1..9] OF CHAR ;
BEGIN
   DAYNAME [MON] := 'MONDAY   ' ;
            ...
   ***
END  (* DAY CONTROL *) ;
```

fig. 1.6 constant data protection

It should also be noted in the example that putting a star before DAY does not make available the values of the type; the values themselves must also be starred.

A final point to note is that the DAYCONTROL envelope actually contains an error. The fault is that there is no space between the opening bracket of the type list for DAY and the star before the value MON, giving as a result an open comment symbol!

1.6 Block nesting

Envelope definitions and instance sections are allowed to appear in any declaration position in a block. For example, the block format shown in fig. 1.7 is perfectly legal (if somewhat bizarre).

```
PROCEDURE P1 ;
   INSTANCE I1 : E1 ;
   LABEL 4 ;
   CONST C = 12 ;
   ENVELOPE E2 ;
   BEGIN *** END ;
   VAR V : INTEGER ;
   INSTANCE I2,I3,I4 : ARRAY [2..4] OF E2 ;
   PROCEDURE P2 ;
   BEGIN END ;
   ENVELOPE MODULE E3 ;
   BEGIN *** END ;
BEGIN END ;
```

fig. 1.7 : Sample block format

Envelope definitions and instance declarations may also be declared in envelope blocks and may as a consequence, be starred. An application of this facility is described in section 2.7.

1.7 Envelope instances as parameters

Envelope instances may be passed as variable parameters to any block. In this respect again, envelopes bear a close resemblance to PASCAL files.

As an illustration of this facility, consider the envelope in fig. 1.8 which represents the set of alphabetic characters.

```
ENVELOPE LETTERSET ;
   VAR LSET : PACKED ARRAY ['A'..'Z'] OF BOOLEAN ;

   PROCEDURE *ASSIGNEMPTY ;
      VAR CH : 'A'..'Z' ;
   BEGIN
      FOR CH := 'A' TO 'Z' DO LSET [CH] := FALSE
   END  (* ASSIGN EMPTY *)

   PROCEDURE *PLUS (CH : CHAR) ;
   BEGIN LSET [CH] := TRUE END ;

   PROCEDURE *MINUS (CH : CHAR) ;
   BECIN LSET [CH] :- FALSE END ;

   FUNCTION *PRESENT (CH : CHAR) : BOOLEAN ;
   BEGIN PRESENT := LSET [CH] END ;

BEGIN
   ASSIGNEMPTY ;
   ***
END  (* LETTER SET *) ;
```

fig. 1.8 : Envelope LETTERSET

This envelope provides four operations - ASSIGNEMPTY which assigns an empty value to the set, PLUS which adds a character to the set, MINUS which removes a character from the set and PRESENT which indicates if a specific character is a member of the set.

Blocks can be defined to provide operations, such as union and intersection, on this set. For example, the procedure in fig. 1.9, given two lettersets S1 and S2, returns as a result their intersection.

Note that although the two operands are conceptually value parameters to the intersection operation they must be passed as variables.

```
PROCEDURE INTERSECTION (VAR S1,S2 : LETTERSET ;
                        VAR RESULT : LETTERSET) ;
   VAR CH : 'A'..'Z' ;
BEGIN
   RESULT.ASSIGNEMPTY ;
   FOR CH := 'A' TO 'Z' DO
      IF S1.PRESENT (CH) AND S2.PRESENT (CH)
      THEN RESULT.PLUS (CH)
END  (* INTERSECTION *) ;
```

fig. 1.9 : Envelope instances as parameters

1.8 The inner mechanism

The description of the inner statement given in section 1.2 is somewhat deceptive in that it implies that the inner statement is a static separator dividing initialisation and finalisation code. It may, of course, be used in this way but its actual power is much greater than that suggested by this description.

The inner statement is like any other statement and may be executed conditionally or repeatedly. There may also be more than one inner statement in an envelope body or none at all.

When envelope instances are declared in a block, the effective body of the block is a combination of the explicit body and the bodies of the envelope instances. The effective body is determined by applying the inner substitution rule:-

> "The effective body of a parent block is formed by taking the effective body of each envelope instance in order of declaration in the block and replacing any inner statements by the effective body of the next envelope instance in the block, or if there is none, by the explicit body of the block"

In a vector instance declaration I[N] is defined to precede I[N+1].

As a simple example of the application of the inner substitution rule, consider the procedure shown in fig. 1.10. Procedure USER has two instances of the FILE envelope described in section 1.4.

```
PROCEDURE USER ;
   INSTANCE
      FILE1,FILE2 : FILE ;
BEGIN
   body of USER
END  (* USER *)
```

fig. 1.10 : Procedure USER

Fig. 1.11 shows the effective body of procedure USER, formed by applying the inner substitution rule.

```
BEGIN
   open FILE1 ;
   open FILE2 ;
   body of user ;
   close FILE2 ;
   close FILE1 ;
END  (* USER *) ;
```

fig. 1.11 : Effective body of procedure USER

In this example, it might be considered undesirable to execute the body of procedure USER if, for some reason, either file cannot be opened. This effect can be achieved by making the execution of the inner statement of the FILE envelope conditional on the successful opening of the file. The body of the envelope would take the form shown in fig. 1.12.

```
BEGIN
   open file ;
   IF file open THEN
   BEGIN
      *** ;
      close file
   END
END  (* file *) ;
```

fig. 1.12 : Conditional inner statement

and the effective body would have the form shown in fig. 1.13.

```
BEGIN
   open FILE1 ;
   IF file open THEN
   BEGIN
      open FILE2 ;
      IF file open THEN
      BEGIN
         body of USER ;
         close FILE2
      END ;
      close FILE1
   END
END  (* USER *) ;
```

fig. 1.13 : Effective body of USER with conditional FILE inner

The inner mechanism of an envelope is a useful tool in its own right. For example, the TIMER envelope in fig. 1.14 only makes use of the properties of the inner statement and does not provide any starred interface. The effect of declaring an instance of this envelope in any block is to have the lifetime of the block printed when its execution is complete.

The type ALFA in the envelope is defined as

```
PACKED ARRAY [1..8] OF CHAR
```

```
ENVELOPE TIMER (BLOCK : ALFA) ;
   VAR ENTERTIME,EXITTIME : INTEGER ;
BEGIN
   MILL (ENTERTIME) ;
   *** ;
   MILL (EXITTIME) ;
   WRITELN ('EXECUTION TIME : ',EXITTIME-ENTERTIME,
            'MILLISECONDS FOR ',BLOCK)
END  (* TIMER *) ;
```

fig. 1.14 : envelope TIMER

Fig. 1.15 shows a sample declaration of the TIMER envelope

```
PROCEDURE EXAMPLE ;
   INSTANCE
      I : TIMER ('EXAMPLE ') ;
      ...
BEGIN ... END  (* EXAMPLE *) ;
```

fig. 1.15 : Block timing

The corresponding output would be of the form

```
EXECUTION TIME : 128  MILLISECONDS FOR EXAMPLE
```

1.9 Effectively global variables

Variables which are declared local to an envelope module which is itself textually enclosed only by other envelope modules and/or the main program are said to be effectively global.

External files may be declared in such a position, as illustrated in fig. 1.16.

```
PROGRAM TEST (FILE1,FILE2) ;
   ENVELOPE MODULE E1 ;
      VAR FILE1 : TEXT ;
      ENVELOPE MODULE E2 ;
         VAR FILE2 : FILE OF INTEGER ;
      BEGIN *** END (* E2 *) ;
   BEGIN *** END (* E1 *) ;
BEGIN END  (* TEST *) .
```

fig. 1.16 : Effectively global files

1.10 Summary

definition: An envelope is a data structure which facilitates the grouping of related program declarations.

1.10.1 Form

An envelope definition has the same syntactic structure as a PASCAL procedure

```
e.g.    ENVELOPE E (formal parameter list) ;
          local declarations ;
        BEGIN
          body
        END ;
```

An envelope definition may appear in any declaration position in a block.

1.10.2 Instances

Instances of envelopes are declared in a section preceded by the word INSTANCE

```
e.g.    INSTANCE
          I : E (actual parameter list) ;
```

Several instances may be declared in the same statement

```
e.g.    I1,I2 : E (list for I1)(list for I2) ;
```

and instances may be declared as vectors

```
e.g.    I : array[0..1] of E ((list for I[0])(list for I[1])) ;
```

An envelope must be fully defined before an instance of it can be declared.

An instance section may appear in any declaration position in a block and several sections may appear in one block.

1.10.3 Modules

Envelopes of which only one instance is required may have their definition and declaration combined by preceding the block name with the word MODULE

e.g. ENVELOPE MODULE E ;

1.10.4 Access

The local identifiers of an envelope are invisible outside the envelope unless their declarations are preceded by a star.

Any local identifier may be starred.

Access to local envelope identifiers is made through the dot notation as with PASCAL records

e.g. E.STARREDNAME
E1.E2 [SUBSCRIPT].STARREDNAME

Again like PASCAL records, to avoid the repeated use of the access description a WITH statement may be used

e.g. WITH E DO .....

1.10.5 Instances as parameters

Envelope instances may be passed as variable parameters to any block

e.g. PROCEDURE P (VAR I : E) ;

1.10.6 Data scope

The normal scope rules for PASCAL identifiers, apply to envelopes with the additional restrictions that

(a) Starred data of an envelope is read only to code outside the envelope.

(b) Data outside an envelope is read only to the code of the envelope.

1.10.7 Inner statement

The body of an envelope will normally contain at least one inner statement. This is denoted by three stars - *** .

An inner statement represents the body of the next envelope instance in the block or if there is none, the body of the block in which the instance is declared.

In a vector instance declaration, I[N] precedes I[N+1].

2. Processes, Monitors and Conditions

A common way of introducing computer programming to complete novices is by analogy with the preparation of a list of instructions for some human activity such as crossing the road or baking a cake.

Although most problems can be approached in this way, a program in general may describe not a single activity but a set of interacting activities. By analogy, one might describe how two people would bake a cake or how a group of drivers and pedestrians would co-operate to cross each other's path.

The process, monitor and condition structures of PASCALPLUS provide a means for representing such problems.

2.1 Processes

A process is a representation of a program activity.

A process has the same syntax definition as a PASCALPLUS envelope and instances are declared in the same way.

The name of a process instance, like that of a program is never referenced and may be looked upon as a documentation aid.

Examples of process declarations are given in figs. 2.1 and 2.2

```
PROCESS MODULE PRINTA ;
BEGIN
   WRITE ('A')
END  (* PRINT A *) ;

PROCESS MODULE PRINTB ;
BEGIN
   WRITE ('B')
END  (* PRINT B *) ;
```

fig. 2.1 : process module declaration

```
PROCESS PRINT (CH : CHAR) ;
BEGIN
   WRITE (CH)
END  (* PRINT *) ;

INSTANCE
   PRINTA,PRINTB : PRINT ('A')('B') ;
```

fig. 2.2 : multiple instance process declarations

These two sample declarations are equivalent. Each consists of two processes - PRINTA which outputs the letter 'A' and PRINTB which outputs the letter 'B'. From the user's point of view, both processes are executed at the same time, or in parallel.

Processes are rarely independent of each other and occasionally need to access shared "resources", such as information held in a data structure or, like the examples, the output stream. Processes must co-operate in order to use a shared resource properly. The processes PRINTA and PRINTB, for instance, cannot print a character at the same time so any clash must be resolved to allow first one process to proceed to print its character and then the other. Such clashes are resolved in the case of the built-in PASCALPLUS procedures (READ,WRITE,NEW..etc.), by having each procedure guarantee exclusion during its execution. This means that only one process at a time may execute the code associated with that procedure. The underlying hardware ensures that two processes never invoke a built-in procedure at exactly the same time.

If one process attempts to use a built-in procedure while another process is executing it, the arriving process is delayed until the current process is finished. In the examples, the first process which attempts to output a character using the WRITE procedure will exclude the other process until the output is complete.

It is indeterminate which process will execute the WRITE procedure first. The output could therefore be either AB or BA and the result might vary from run to run!

2.2 Monitors

A monitor is an envelope which holds data accessed by more than one process and guarantees exclusion on that data.

Consider the example shown in fig. 2.3. This is part of a program which could be used to analyse (roughly) how processes actually progress relative to each other.

```
MONITOR MODULE TRACE ;
   VAR *TOTALCOUNT : INTEGER ;
   PROCEDURE *MARK ;
   BEGIN
      IF TOTALCOUNT <> 10000 THEN
      TOTALCOUNT := TOTALCOUNT + 1
   END  (* MARK *) ;
BEGIN
   TOTALCOUNT := 0 ;
   ***
END  (* TRACE *) ;

PROCESS MARKER ;
   VAR CYCLECOUNT : INTEGER ;
BEGIN
   CYCLECOUNT := 0 ;
   WHILE TRACE.TOTALCOUNT <> 10000 DO
   BEGIN
      TRACE.MARK ;
      CYCLECOUNT := CYCLECOUNT + 1
   END ;
   WRITELN (CYCLECOUNT)
END  (* MARKER *) ;

INSTANCE
   MARKERS : ARRAY [1..N] OF MARKER ;
```

fig. 2.3 : monitor exclusion for updating shared data

While the value of TOTALCOUNT in monitor TRACE is not equal to 10000 (say), each of the MARKER processes repeatedly calls procedure MARK to indicate that it has completed an execution cycle. Each process also keeps its own cycle count which it prints whenever TOTALCOUNT reaches the limit value.

The action of procedure MARK is to increment the value of TOTALCOUNT, provided it has not reached the limit value.

A monitor provides exclusion during the execution of its procedures and functions so that only one process at a time is capable of modifying its local data. In the case of procedure MARK, this exclusion ensures that there is no possibility of TOTALCOUNT being modified between being tested and then incremented.

Starred variables of a monitor may be inspected by more than one process at a time. Such variables though, can only be modified via a procedure of the monitor so exclusion is still preserved.

While several processes are inspecting a monitor variable, another may be modifying it. The hardware ensures that the inspecting processes do not get a meaningless value but it could be either the value just before modification or just after. In the example therefore, TOTALCOUNT can be modified between a MARKER process inspecting its value outside the monitor and then subsequently acting on it. The "real" total count in this program might be as large as 10000 + N - 1.

Monitor exclusion is achieved in PASCALPLUS by only allowing one process at a time to execute monitor code. Such a process is said to be in monitor mode. Once a process has entered monitor mode by invoking a monitor procedure (or function) it is then free to call any other monitor procedure or even the initial procedure recursively.

As all other processes are delayed if they try to enter monitor mode while it is occupied, it is essential that each process should spend as little time as possible in this state.

2.3 Monitor and process nesting

Processes may be defined in monitor blocks.

In order to maintain exclusion on the monitor data in such cases the data outside a process, but within its scope, is defined to be read only to the process.

The partially coded monitor in fig. 2.4 illustrates an application of this facility.

```
MONITOR MODULE INPUTCONTROL ;
   VAR BUFFER : ARRAY [1..N] OF CHAR ;

   PROCEDURE PUT (CH : CHAR) ;
   BEGIN
      put character into buffer
   END  (* PUT *) ;

   PROCEDURE *GET (VAR CH : CHAR) ;
   BEGIN
      get character from buffer
   END  (* GET *) ;

   PROCESS MODULE INPUT ;
      VAR CH : CHAR ;
   BEGIN
      REPEAT
         READ (CH) ;
         PUT (CH)
      UNTIL CH = TERMINATOR
   END  (* INPUT *) ;

BEGIN
   initialise buffer ;
   ***
END  (* INPUT CONTROL *) ;
```

fig. 2.4 : monitor with local process

The action of process INPUT is to repeatedly input characters and store them in the buffer, until some terminal character is detected.

The monitor provides the procedure GET which processes can use to remove characters from the buffer. Note that a process using the monitor need not be aware that the characters are coming from a buffer, rather than being input directly.

A fully coded version of this monitor is given in section 2.6 where the problems of what to do if the buffer becomes full during a PUT operation or empty during a GET operation, are considered.

executed up to an inner statement and is therefore initialised and ready for processes to use it.

Once activated the processes proceed in parallel until they all terminate. When all processes have stopped, the code following the inner statement in the program body is executed - in effect, the finalisation code in each monitor is executed.

The program shown in fig. 2.6 is a re-written version of the trace program described in section 2.2. The cycle count for each process is kept in the TRACE monitor and a count table printed when the finalisation code is executed, following the termination of the processes.

It should be noted that of the two versions of the trace program, the one in fig. 2.3 is preferable because it spends less time in monitor mode.

```
PROGRAM PROCESSTRACE ;
   TYPE PROCESSRANGE = 1..N ;

   MONITOR MODULE TRACE ;
      VAR
        *TOTALCOUNT : INTEGER ;
         COUNT : ARRAY [PROCESSRANGE] OF INTEGER ;
         INDEX : PROCESSRANGE ;
      PROCEDURE *MARK (P : PROCESSRANGE) ;
      BEGIN
         COUNT [P] := COUNT [P] + 1 ;
         IF TOTALCOUNT <> 10000 THEN
         TOTALCOUNT := TOTALCOUNT + 1
      END  (* MARK *) ;
   BEGIN
      TOTALCOUNT := 0 ;
      FOR INDEX := 1 TO N DO COUNT [INDEX] := 0 ;
      *** ;
      FOR INDEX := 1 to N DO WRITELN (COUNT [INDEX])
   END  (* TRACE *) ;

   PROCESS MARKER (P : PROCESSRANGE) ;
   BEGIN
      WHILE TRACE.TOTALCOUNT <> 10000 DO TRACE.MARK (P)
   END  (* MARKER *) ;

   INSTANCE
      MARKERS : ARRAY [PROCESSRANGE] OF MARKER ((1)(2)..(N)) ;

BEGIN *** END  (* PROCESS TRACE *) .
```

fig. 2.6 : trace program - version 2

2.6 Process synchronisation

If a process requires a resource which is not currently available it will usually suspend itself until some other process provides that resource. This process synchronisation is achieved using instances of a built-in monitor CONDITION. The interface provided by the condition monitor is shown in fig. 2.7.

```
MONITOR CONDITION ;
   TYPE RANGE = 0..MAXINT ;
   PROCEDURE *PWAIT (P : RANGE ) ;
   PROCEDURE *WAIT ;
   PROCEDURE *SIGNAL ;
   FUNCTION *LENGTH : RANGE ;
   FUNCTION *PRIORITY : RANGE ;
END  (* CONDITION *) ;
```

fig. 2.7 : monitor CONDITION interface

Associated with each instance of CONDITION is an ordered queue on which processes may be temporarily suspended until the resource they require is available.

The action of procedure PWAIT is to suspend the process calling it on the condition queue, in a position determined by the priority value P. A process is placed after processes with the same or smaller values of P.

When a process is suspended on a condition queue monitor mode is released.

The action of procedure WAIT is to call procedure PWAIT with a default priority of MAXINT DIV 2.

i.e. C.WAIT ≡ C.PWAIT (MAXINT DIV 2)

The action of procedure SIGNAL is to activate the top process on the condition queue. As only one process may be active in monitor mode, SIGNAL delays the process which calls it in order to allow the signalled process to proceed. The delayed process is activated again as soon as monitor mode is free.

If a condition queue is empty, calling SIGNAL has no effect.

Function LENGTH returns the number of processes on the queue.

Function PRIORITY returns the priority of the process at the head of the queue. If this function is invoked when the queue is empty, a run-time error is flagged.

The example shown in fig. 2.8 illustrates an application of the condition monitor.

```
MONITOR SINGLERESOURCE ;
   VAR BUSY : BOOLEAN ;
   INSTANCE NONBUSY : CONDITION ;

   PROCEDURE *ACQUIRE ;
   BEGIN
      IF BUSY THEN NONBUSY.WAIT ;
      BUSY := TRUE
   END  (* ACQUIRE *) ;

   PROCEDURE *RELEASE ;
   BEGIN
      BUSY := FALSE ;
      NONBUSY.SIGNAL
   END  (* RELEASE *) ;

BEGIN
   BUSY := FALSE ;
   ***
END  (* SINGLE RESOURCE *) ;
```

fig. 2.8 : single resource monitor

This monitor could be used to schedule a single resource such as the output stream.

```
e.g.   INSTANCE
          LP : SINGLERESOURCE ;
```

Any process which wants to use the output stream must first call LP.ACQUIRE and then when it has finished with it, call LP.RELEASE.

If the output stream is being used when a process attempts to acquire it, as indicated by the boolean BUSY, then that process is suspended on the condition NONBUSY. The process then remains on the condition queue until the output stream is released.

The action of calling procedure RELEASE is to set the boolean BUSY to false to indicate that the resource is now free, and then to "signal" the top process on the NONBUSY queue. Control is immediately transferred to the waiting process and it proceeds to execute the ACQUIRE procedure after the NONBUSY.WAIT statement i.e. it acquires the output stream by setting BUSY to true.

Control is subsequently returned to the signalling process as soon as monitor mode is free.

Fig. 2.9 shows a fully coded version of the INPUTCONTROL monitor described in section 2.3.

```
MONITOR MODULE INPUTCONTROL ;
   VAR
      COUNT : 0..N ;
      NEXTIN,NEXTOUT : 1..N ;
      BUFFER : ARRAY [1..N] OF CHAR ;
   INSTANCE
      NONEMPTY,NONFULL : CONDITION ;
   PROCEDURE PUT (CH : CHAR ) ;
   BEGIN
      IF COUNT = N THEN NONFULL.WAIT ;
      BUFFER [NEXTIN] := CH ;
      IF NEXTIN = N THEN NEXTIN := 1
      ELSE NEXTIN := NEXTIN + 1 ;
      COUNT := COUNT + 1 ;
      NONEMPTY.SIGNAL
   END  (* PUT *) ;
   PROCEDURE *GET (VAR CH : CHAR) ;
   BEGIN
      IF COUNT = 0 THEN NONEMPTY.WAIT ;
      CH := BUFFER [NEXTOUT] ;
      IF NEXTOUT = N THEN NEXTOUT := 1
      ELSE NEXTOUT := NEXTOUT + 1 ;
      COUNT := COUNT - 1 ;
      NONFULL.SIGNAL
   END  (* GET *) ;
   PROCESS MODULE INPUT ;
      VAR CH : CHAR ;
   BEGIN
      REPEAT
         READ (CH) ; PUT (CH)
      UNTIL CH = TERMINATOR
   END  (* INPUT *) ;
BEGIN
   COUNT := 0 ;
   NEXTIN := 1 ; NEXTOUT := 1 ;
   ***
END  (* INPUT CONTROL *) ;
```

fig. 2.9 : input control using a cyclic buffer

The INPUT process repeatedly reads characters and attempts to store them in the buffer. If the buffer is full, this process is suspended on the NONFULL condition until some other process removes a character from the buffer by calling procedure GET.

Correspondingly, if a process attempts to "get" a character when the buffer is empty it is suspended on the NONEMPTY condition until the INPUT process stores a character in the buffer by calling procedure PUT.

2.7 The envelope as a monitor interface

The initial acquisition and final release of a resource as illustrated in section 2.6, is a very common activity in any multiprocessing program. Unfortunately this mechanism is insecure as the responsibility for using the resource properly is left to the processes. A process, for instance, might "forget" to release a resource, or attempt to use it without first acquiring it.

A solution to this problem is to impose an envelope interface between the resource and the processes which wish to use it. Consider, for example, the monitor shown in fig. 2.10.

```
MONITOR MODULE OUTPUTFILECONTROL ;
   INSTANCE
      THISFILE : SINGLERESOURCE ;  (* fig. 2.8 *)
   ENVELOPE *INTERFACE ;
      PROCEDURE *PRINT (CH : CHAR) ;
      BEGIN
         IF CH = NEWLINE THEN WRITELN
         ELSE IF CH = NEWPAGE THEN PAGE
         ELSE WRITE (CH)
      END  (* PRINT *) ;
   BEGIN
      THISFILE.ACQUIRE ;
      *** ;
      THISFILE.RELEASE
   END  (* INTERFACE *) ;
BEGIN
   ***
END  (* OUTPUT FILE CONTROL *) ;
```

fig. 2.10 : envelope interface

This monitor might be used to control access to the OUTPUT file. The only starred component of the monitor is the envelope INTERFACE which makes available a procedure PRINT to transfer a character to the file. Any process which wishes to use the output file, first declares an instance of the INTERFACE envelope

```
INSTANCE
   LP : OUTPUTFILECONTROL.INTERFACE ;
```

The effect of declaring an instance of INTERFACE in any block is to acquire the output file before executing the block and then release it when the execution of the block has been completed. During the execution of the block, calls may be made on LP.PRINT to send characters to the output file.

This mechanism therefore both protects a resource by enforcing its correct acquisition and release and also simplifies its use by taking the task of scheduling it away from the processes.

Envelope instances may not be declared in either a monitor block or procedures and functions of a monitor block. (This restriction is partly due to the definition of a monitor as a shared envelope and partly due to implementation difficulties). As a consequence no envelope instance is ever "shared" and is always executed outside monitor mode.

In the example in 2.10 monitor mode is entered on calling both THISFILE.ACQUIRE and THISFILE.RELEASE.

2.8 Alternative program termination

When the end condition for a program is detected, several processes may, at that moment, be held up on condition queues. To avoid having to activate these processes and allow them to terminate by completing the execution of their code bodies, an alternative program termination is allowed: "a program is said to have terminated if all its processes are held up on condition queues."

For instance, the processes using the INPUTCONTROL monitor described in section 2.6, might terminate by being held up on the NONEMPTY condition following the detection of the terminal character by process INPUT.

2.9 Process priority

All processes proceed conceptually in parallel. In practice there are usually more processes than processors to run them, so some means is required to share processor time among the competing processes. To this end a run priority value in the range 0 to MAXINT is associated with each process and processors are allocated to those processes with the highest run priority (i.e. smallest value).

Processes waiting for a processor are suspended on a queue, ordered according to their run priority.

Initially all processes have a run priority of MAXINT DIV 2, and this value may be modified by calls on a built-in procedure SETPRIORITY.

e.g. SETPRIORITY (24)

2.10 Summary

definition: processes are program blocks which may be executed in parallel.

definition: a monitor is an envelope which holds data accessed by more than one process and guarantees exclusion on that data.

2.10.1 Process form, instances and modules

A process has the same syntax definition as an envelope and instances are declared in the same way.

Processes are allocated processor time according to their run priority. Each process has an initial priority of MAXINT DIV 2 which it can modify by calling the built-in procedure SETPRIORITY.

2.10.2 Process and monitor interaction

The program block is a monitor module.

Monitor and process definitions and declarations may only appear in a monitor.

Data declared outside a process is read only to the code of the process.

All the processes in a program are activated when the inner statement of the program block is executed. The processes then proceed (conceptually) in parallel until they terminate, after which the code following the inner statement is executed.

A process terminates when it has completed the execution of its code body or if all processes are suspended on condition queues.

When a process enters a monitor it is said to be in monitor mode and only one process at a time may be in this state.

Envelope instances may not be declared in a monitor.

All shared built-in procedures and functions such as READ and WRITE are executed with exclusion.

2.10.3 Process synchronisation

Processes synchronise using instances of the built-in monitor, CONDITION. The interface provided by this monitor is

```
MONITOR CONDITION ;
   TYPE RANGE = 0..MAXINT ;
   PROCEDURE *PWAIT (P : RANGE) ;
   PROCEDURE *WAIT ;
   PROCEDURE *SIGNAL ;
   FUNCTION *LENGTH : RANGE ;
   FUNCTION *PRIORITY : RANGE ;
END  (* CONDITION *) ;
```

3. Simulation

Simulation programs are concerned with analysing how interacting activities progress relative to each other in "time". Time in this context is pseudo, or simulated, time defined by the simulation program and is in no way related to the actual time required to execute the program.

3.1 The SIMULATION monitor module

Pseudo time control is provided in PASCALPLUS by a monitor module, SIMULATION. Its interface is shown in fig. 3.1.

```
MONITOR MODULE SIMULATION ;
   TYPE TIMERANGE = 0..MAXINT ;
   VAR *PSEUDOTIME : TIMERANGE ;
   PROCEDURE *HOLD (T : TIMERANGE) ;
BEGIN
   PSEUDOTIME := 0 ; ***
END  (* SIMULATION *) ;
```

fig. 3.1 : SIMULATION monitor interface

Associated with the SIMULATION monitor is an ordered queue, known as the time queue, on which processes may suspend themselves for a period of pseudo time, using procedure HOLD. The time queue is ordered so that processes with early wake-up times are at its head. The wake-up time for a suspended process is the sum of the specified delay time T and the value of PSEUDOTIME when that process invoked procedure HOLD.

Pseudo time only advances when all processes in the program are suspended, either on condition queues or the time queue. At this point, the value of PSEUDOTIME is set to the wake-up time for the process at the head of the time queue and all processes waiting for this value of PSEUDOTIME are activated.

If a process calls procedure HOLD with a delay time of zero, it is not suspended and continues its execution.

The program shown in fig. 3.2 illustrates the essential features of simulation programs.

```
PROGRAM EXAMPLE ;
   PROCESS MODULE TICK ;
   BEGIN
      WITH SIMULATION DO
         WHILE PSEUDOTIME < TIMELIMIT DO
         BEGIN
            WRITELN (PSEUDOTIME, '  SECONDS') ;
            HOLD (1)
         END
   END  (* TICK *) ;

   PROCESS MODULE JOBCONTROL ;
      VAR JOBTIME : 0..MAXINT ;
   BEGIN
      REPEAT
         READ (JOBTIME) ;
         SIMULATION.HOLD (JOBTIME) ;
         WRITELN ('END OF JOB')
      UNTIL JOBTIME = 0
   END  (* JOB CONTROL *) ;

   PROCESS MODULE ACTIVITY ;
   BEGIN
      execute activity
   END  (* ACTIVITY *) ;

BEGIN *** END  (* EXAMPLE *) .
```

fig. 3.2 : Sample simulation program

The action of process TICK is to repeatedly print each second of pseudo time up to some specified time limit. The actual units of simulated time are decided by the programmer and could equally well be milliseconds or days. It is of course important to ensure that the same units are adopted throughout the program.

The action of process JOBCONTROL is to input repeatedly a value denoting the time taken to execute some job or task, suspend the process for that length of time and then output a message indicating that the job is complete. This loop is repeated until a JOBTIME of zero is input.

The final process in the program is ACTIVITY which has been purposely left unspecified. Regardless of what this activity is, (provided it does not call HOLD) it will always be completed before process TICK is activated after its first delay of 1 sec. of pseudo time. The activity could be simply the addition of two numbers or perhaps the determination of π to several million decimal places.

What is the output from this program?

Basically it is a series of messages of the form "N SECONDS", interspersed with "END OF JOB" messages.

It is important to note that the delay time for each job cannot be determined from the output. This is because TICK and JOBCONTROL will always wake-up together and it is then indeterminate which will output its message first.

3.2 Simulation of a console room

A simulation program is used to model a system of interacting activities in order to provide a test-bed for evaluating either a proposed system or modifications to an existing one.

Consider, for example, a program to model the behaviour of users in a console room. The room is open for 12 hours a day, 5 days a week and users may come and go at any time during that period.

The basic form of the program is shown in fig. 3.3.

```
PROGRAM CONSOLEROOMSIMULATION ;
   MONITOR MODULE CONSOLECONTROL ;
                  ...
   PROCESS USER ;
                  ...
   INSTANCE
      USERS : ARRAY [USERRANGE] OF USER ;
BEGIN *** END.
```

fig. 3.3 : Console room simulation program - basic form

The time unit chosen for the program is the "minute".

The acquisition, use and subsequent release of the consoles by the users is represented by a monitor CONSOLECONTROL and each user represented by an instance of a process USER.

The form of the CONSOLECONTROL monitor is shown in fig. 3.4 (c.f. example in section 2.7).

```
MONITOR MODULE CONSOLECONTROL ;
   TYPE
      CONSOLERANGE = 1..CONSOLELIMIT ;
   VAR
      CONSOLESAVAILABLE : SET OF CONSOLERANGE ;
   INSTANCE
      CONSOLEFREE : CONDITION ;

   PROCEDURE ACQUIRE (VAR CONSOLE : CONSOLERANGE) ;
      VAR C : CONSOLERANGE ;
   BEGIN
      IF CONSOLESAVAILABLE = [] THEN CONSOLEFREE.WAIT ;
      C := 1 ;
      WHILE NOT (C IN CONSOLESAVAILABLE) DO C := C + 1 ;
      CONSOLESAVAILABLE := CONSOLESAVAILABLE - [C] ;
      CONSOLE := C
   END  (* ACQUIRE *) ;

   PROCEDURE RELEASE (CONSOLE : CONSOLERANGE) ;
   BEGIN
      CONSOLESAVAILABLE := CONSOLESAVAILABLE + [CONSOLE] ;
      CONSOLEFREE.SIGNAL
   END  (* RELEASE *) ;

   ENVELOPE *CONSOLE ;
      VAR
         ACONSOLE : CONSOLERANGE ;

      PROCEDURE *USE (T : TIMERANGE) ;
      BEGIN
         SIMULATION.HOLD (T)
      END  (* USE *) ;

   BEGIN
      ACQUIRE (ACONSOLE) ;
      *** ;
      RELEASE (ACONSOLE)
   END  (* CONSOLE *) ;

BEGIN
   CONSOLESAVAILABLE := [1..CONSOLELIMIT];
   ***
END  (* CONSOLE CONTROL *) ;
```

fig. 3.4 : CONSOLECONTROL monitor

When a USER process declares an instance of the interface envelope CONSOLE it is allocated one of the consoles or if none is available, it is suspended until one is free. It is assumed that the user is prepared to accept any console and will always wait until one is available even up to the last minute of any day.

A process "uses" a console for a time T minutes by calling procedure USE of envelope CONSOLE, which then suspends the process for the specified time.

The exact behaviour of a user is difficult (if not impossible) to describe and the process shown in fig. 3.5 should be regarded as a first approximation to it. It is assumed in this case that each user requires a console for the same fixed length of time each week - TIMEALLOCATION. A user enters the console room at some random time during the week and when he has acquired a console uses it for as long as possible that day. If insufficient time is available the user repeatedly attempts to use the remaining time at random intervals during the remainder of the week until either all the required time is used or the week "runs out".

```
PROCESS USER ;
   VAR
      STARTDELAY,TIMEREQUIRED,LIMITTIME : TIMERANGE ;

   PROCEDURE USECONSOLE (VAR TIMEREQUIRED : TIMERANGE) ;
      INSTANCE
         ACONSOLE : CONSOLECONTROL.CONSOLE ;
      VAR
         AVAILABLETIME : TIMERANGE ;
   BEGIN
      WITH SIMULATION
      DO AVAILABLETIME := DAYLENGTH - 1 - PSEUDOTIME MOD DAYLENGTH ;
      IF AVAILABLETIME > 0 THEN
      IF TIMEREQUIRED > AVAILABLETIME THEN
      BEGIN
         ACONSOLE.USE (AVAILABLETIME) ;
         TIMEREQUIRED := TIMEREQUIRED - AVAILABLETIME ;
      END ELSE
      BEGIN
         ACONSOLE.USE (TIMEREQUIRED) ;
         TIMEREQUIRED := 0 ;
      END
   END  (* USE CONSOLE *) ;

BEGIN
   TIMEREQUIRED := TIMEALLOCATION ;
   WITH SIMULATION DO
   REPEAT
      LIMITTIME := TIMELIMIT - TIMEREQUIRED ;
      STARTDELAY := RANDOMNUMBERCONTROL.RNI (0,LIMITTIME - PSEUDOTIME) ;
      IF PSEUDOTIME < LIMITTIME THEN HOLD (STARTDELAY) ;
      USECONSOLE (TIMEREQUIRED) ;
   UNTIL (TIMEREQUIRED = 0) OR (PSEUDOTIME = TIMELIMIT) ;
END  (* USER *) ;
```

fig. 3.5 : USER process

It should be noted that PASCALPLUS does not provide a built-in random number generator so this must always be supplied by the programmer. The USER process in this example assumes the existence of a monitor RANDOMNUMBERCONTROL, with a local function RNI which returns a random number in the range specified by its parameters.

i.e. FUNCTION RNI (MIN,MAX : INTEGER) : INTEGER ;

The basic form of the simulation program is now complete and the next step is to add code to extract and print the information it generates.

The OBSERVER process in fig. 3.6 would be defined local to the CONSOLECONTROL monitor. Its action is to first "observe" the consoles at one minute intervals noting which ones are free, and then at the end of the week to print tables showing the activity on each console during the week. A sample of the output is illustrated in fig. 3.7.

```
PROCESS MODULE OBSERVER ;
   VAR
      FREE : ARRAY [CONSOLERANGE] OF
                  PACKED ARRAY [0..TIMELIMIT] OF BOOLEAN ;
      CONSOLE : CONSOLERANGE ;
      TIME,FREETIME,TOTALFREETIME : TIMERANGE ;

BEGIN
   REPEAT
      FOR CONSOLE := 1 TO CONSOLELIMIT DO
         FREE [CONSOLE] [SIMULATION.PSEUDOTIME] :=
            CONSOLE IN CONSOLESAVAILABLE ;
      SIMULATION.HOLD (1) ;
   UNTIL SIMULATION.PSEUDOTIME > TIMELIMIT ;
   TOTALFREETIME := 0 ;
   FOR CONSOLE := 1 TO CONSOLELIMIT DO
   BEGIN
      WRITELN ; WRITELN ;
      WRITELN ('HISTORY OF CONSOLE',CONSOLE : 4) ;
      FREETIME := 0 ;
      FOR TIME := 0 TO TIMELIMIT DO
      BEGIN
         IF TIME MOD 60 = 0 THEN
         BEGIN
            WRITELN ;
            IF TIME MOD 480 = 0 THEN
            BEGIN
               WRITELN ; WRITELN ;
               WRITELN ('DAY', (TIME DIV 480) + 1 : 3) ;
            END ;
         END ;
         IF FREE [CONSOLE] [TIME] THEN
         BEGIN
            FREETIME := FREETIME + 1 ;
            WRITE ('-') ;
         END
         ELSE WRITE ('*') ;
      END ;
      WRITELN ;
      WRITE ('FREE TIME : ',FREETIME : 4,' MINUTES = ') ;
      WRITELN (FREETIME * 100) DIV TIMELIMIT : 3' %') ;
      TOTALFREETIME := TOTALFREETIME + FREETIME ;
   END ;
   WRITELN ; WRITELN ;
   WRITE ('TOTAL FREE TIME : ',TOTALFREETIME,' MINUTES = ') ;
   WRITELN((TOTALFREETIME * 100) DIV (TIMELIMIT * 4) : 3'1:') ;
END  (* OBSERVER *) ;
```

fig. 3.6 : OBSERVER process

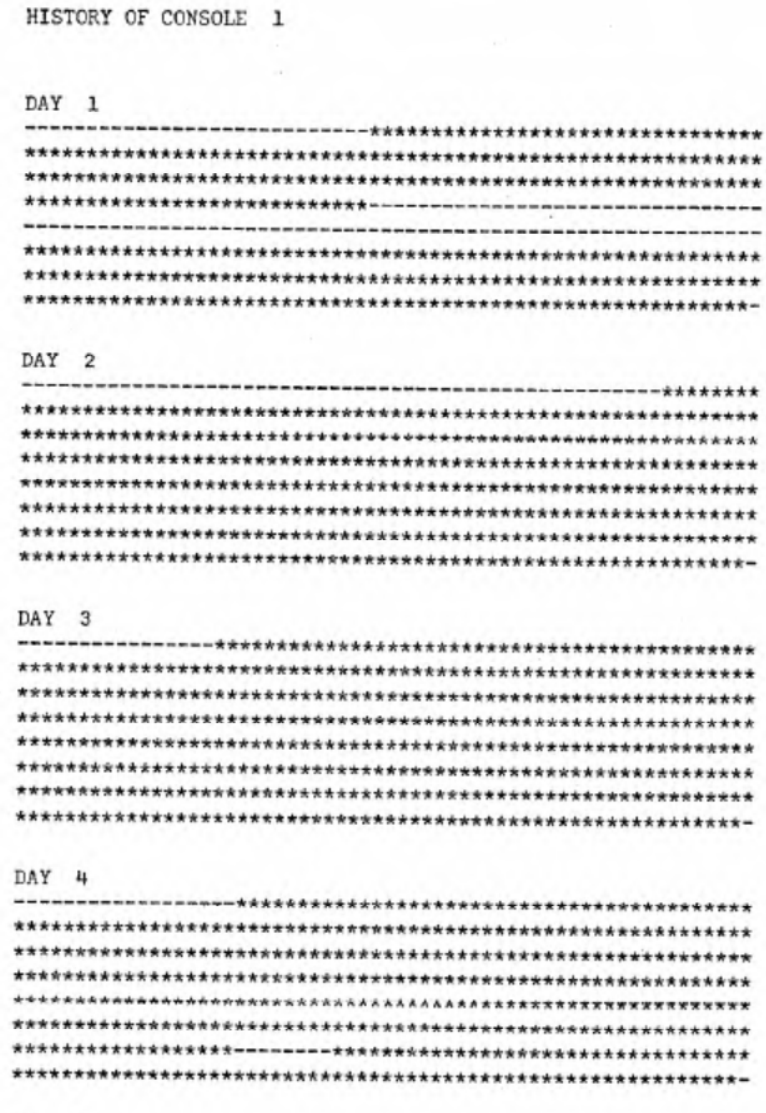


fig. 3.7 : Sample output of console usage

At this point the reader is invited to continue the development of the simulation program, providing for example

(i) details of the behaviour of the users, such as console waiting times, failures to complete the time allocation etc.

(ii) a more realistic user algorithm, incorporating differing time allocations, use of the console room for several short intervals rather than one long one, weighted arrival rates (e.g. more likely to arrive in the morning than the afternoon) etc.

(iii) attempt policy changes such as closing the console room over lunch hour.

The reader may even try to draw conclusions such as the console requirement for a fixed population of users or conversely the population limit for a fixed number of consoles.

4. 1900 PASCALPLUS

This section relates to the preparation and execution of PASCALPLUS programs in an ICL 1900 computer installation. It should be read in conjunction with the "1900 PASCAL user's guide".

4.1 Variable initialisation

All effectively global variables may be initialised in a VALUE section. An example of the use of this facility is shown in fig. 4.1.

```
PROGRAM EXAMPLE ;
   MONITOR MODULE M ;
      VAR I : INTEGER ;
      VALUE I = 20 ;
   BEGIN *** END (* M *)
BEGIN *** END (* EXAMPLE *) .
```

fig 4.1 VALUE initialisation

4.2 Error codes

PASCALPLUS has the following additional error codes.

COMPILE TIME ERRORS
****

500 : DATA IS READ-ONLY IN THIS CONTEXT.
501 : INNER STATEMENT MAY ONLY APPEAR IN THE BODY OF A MONITOR OR ENVELOPE BLOCK
502 : '.' EXPECTED.
503 : ENVELOPE INSTANCE CANNOT BE DECLARED IN A MONITOR.
504 : MONITOR AND ENVELOPE INSTANCES MAY ONLY BE PASSED AS VARIABLE PARAMETERS.
505 : MONITOR AND PROCESS INSTANCES MAY ONLY BE DECLARED IN A MONITOR.
506 : MONITOR PROCEDURES AND FUNCTIONS MAY NOT BE PASSED AS PARAMETERS.

RUN TIME ERRORS
****

520 : PROGRAM CONTAINS TOO MANY PROCESSES.
521 : PRIORITY OPERATION ATTEMPTED ON AN EMPTY CONDITION QUEUE.
522 : CONDITION INSTANCE REFERENCED IN THE BODY OF A MONITOR.
523 : PWAIT OR HOLD ATTEMPTED WITH NEGATIVE PARAMETER VALUE.

4.3 Diagnostic facilities

The diagnostic facilities of 1900 PASCAL are not implemented in 1900 PASCALPLUS.

4.4 Compiler directives

Most of the 1900 PASCAL directives are not implemented in 1900 PASCALPLUS. The directives provided are

```
% LISTING = {TRUE|FALSE}
% LISTING = {ON|OFF}
% CHECKS  = {TRUE|FALSE}
% CHECKS  = {ON|OFF}
% EBM = {TRUE|FALSE}
% CDM = {TRUE|FALSE}
```

The directives with TRUE/FALSE values may only be used at the head of a program.

Two additional directives are provided

(i) %NOPROCESSES = {TRUE|FALSE} (default : FALSE)
If %NOPROCESSES = TRUE is specified, the compiler assumes that the program which follows does not contain any processes or monitors.

(ii) %NOSLICING = {TRUE|FALSE} (default : FALSE)
To achieve the effect of processes executing in parallel on a single 1900 processor, control is regularly switched between the active processes. This switching mechanism is quite expensive and, if desired, may be suppressed by specifying %NOSLICING = TRUE at the head of a program.

4.5 Source library mechanism

A 1900 PASCAL enhancement is available which enables compilers running under George 3 or 4 operating systems to incorporate blocks from a source library during compilation. The blocks are held as normal text in file store files and the library may thus be created and maintained using the normal text filing and editing facilities of the George environment.

4.5.1 Retrieving a library block

To request the incorporation of a library block during compilation the user includes a block retrieval command (fig. 4.2) at the appropriate point in a block declaration part of his program.

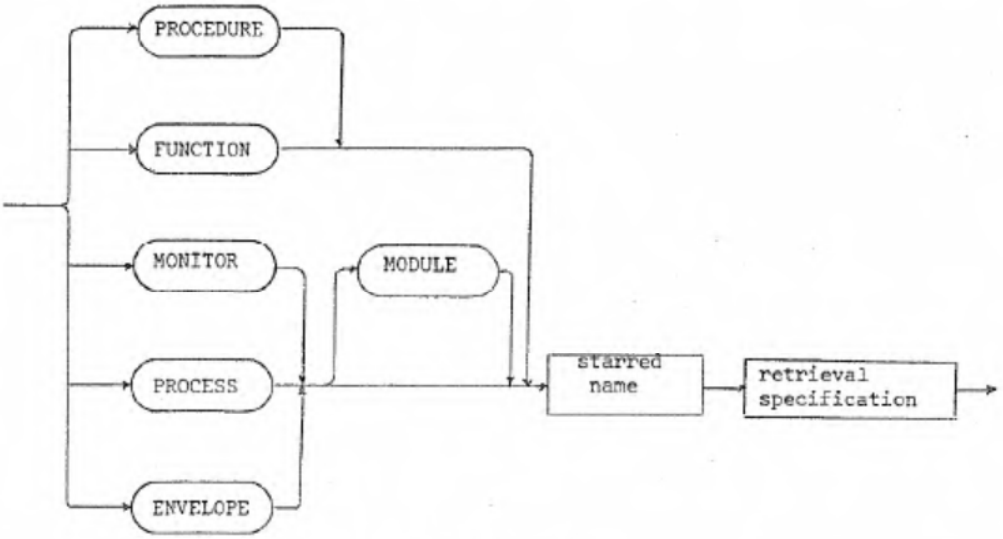


fig. 4.2 : retrieval command

The retrieval specification (fig. 4.3) determines the file from which the block is to be retrieved, whether listing is to continue during compilation of the block, and any adjustment of the non-local name scope necessary for its proper compilation.

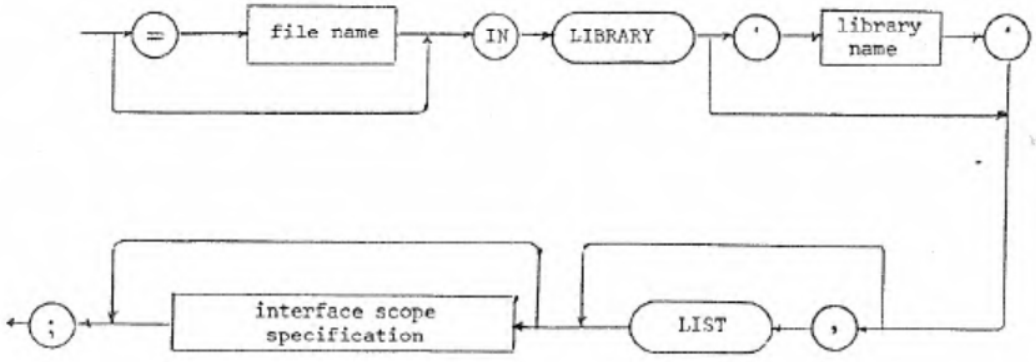


fig. 4.3 retrieval specification

If no file name is specified the file is assumed to have the same name as the block identifier specified by the retrieval command, otherwise it is the name following = . Note that since 1900 PASCAL identifiers are limited to 8 significant characters the names used for library filestore files must also be limited to 8 characters or less, in the form of a valid PASCAL name.

The library name is a 4 letter user code e.g. 'ABCD'. If a library name is specified the file is sought in the file store area under that user code, otherwise it is expected to be in the user's own file store area.

If, LIST is present listing continues during compilation of the retrieved library block but with line numbers restarting from zero, to indicate the source line position within the library file. (When compilation of the normal program text following the retrieval command is resumed, the previous sequence of line numbers is also resumed). If the listing specification is omitted listing is suppressed during compilation of the library block.

The meaning of a block often depends on names denoting types or constants which are assumed to be non-local to the block. It is often inconvenient, or impossible, to ensure that the required names are defined in the scope in which a block retrieval command occurs. Instead the user may define such names in an interface scope specification (fig. 4.4) within the retrieval command itself. Names defined in this way are available as non-locals during the compilation of the retrieved block, but have no effect on the scope in which the retrieval command occurs.

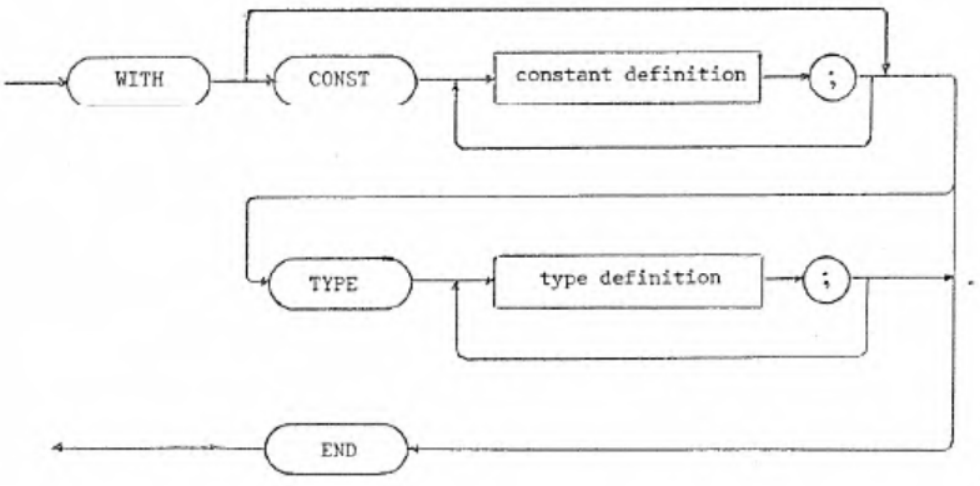


fig. 4.4 : interface scope specification

Examples of retrieval commands

```
PROCESS MODULE READER IN LIBRARY 'BDCN' ;

MONITOR *BUFFER IN LIBRARY , LIST ;

ENVELOPE INTERFACE IN LIBRARY
   WITH CONST N = 100 ;
   END ;

PROCEDURE SORT IN LIBRARY
   WITH CONST N = 100 ;
        TYPE  TABLE = ARRAY [1..N] OF REAL ;
              ELEMENT = REAL ;
   END ;
```

4.5.2 Storing library blocks

The contents of the filestore file retrieved by a retrieval command must be a 1900 PASCAL block declaration terminated by a semi-colon in the usual way. The name appearing in the block heading is irrelevant since the block compiled will be identified by the name appearing in the retrieval command. To avoid confusion it is recommended that the name appearing in the filed block heading should coincide with the filename by which it is retrieved. A recursive procedure or function must always be retrieved with the name by which it recursively calls itself.

Library blocks may contain retrieval commands for other library blocks and in general retrievals may be nested to any depth, provided a recursive retrieval cycle is not set up.

The type of filed block need not necessarily correspond to the type specified in the retrieval command. For instance a filed envelope may be retrieved as a monitor module, or a filed procedure as a process. The allowed mappings are given in the table in fig. 4.5.

filed block	permitted retrieval blocks
FUNCTION	FUNCTION
MONITOR	MONITOR, MONITOR MODULE
PROCESS	PROCESS, PROCESS MODULE
ENVELOPE	ENVELOPE, ENVELOPE MODULE
	MONITOR, MONITOR MODULE
ENVELOPE MODULE	ENVELOPE MODULE, MONITOR MODULE
MONITOR MODULE	MONITOR MODULE
PROCESS MODULE	PROCESS MODULE

fig. 4.5 : library retrieval mappings

4.5.3 Generalised blocks

The fact that the meaning of a block depends on the non-local type and constant names used within it enables quite general blocks to be stored in source libraries. The effect of the block in any particular compilation will be determined by the definition given to these names in the scope in which the retrieval command occurs, or by the retrieval command itself. To facilitate the programming of such general blocks one further extension is made to 1900 Pascalplus by the source library enhancement. The minimum and maximum values allowed by a simple type T may be denoted in an expression as T.MIN and T.MAX respectively. The type denoted by the type name may be any simple type other than REAL.

4.6 The PASCALPLUS macro

This macro performs the same function as the PASCAL macro. It differs from the PASCAL macro in the following ways

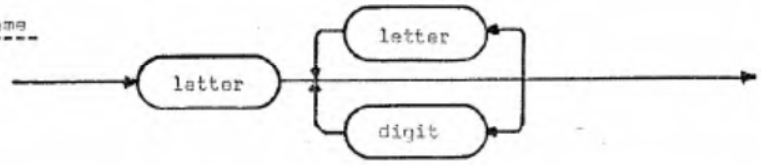
(i) Parameters associated with the diagnostic facilities of PASCAL do not apply i.e. TOKEN, TABLE, RUNOPTIONS and OPTIONSexcept for

CHECKS|NOCHECKS
LISTING|NOLISTING
EDM|CDM|DBM|EBM

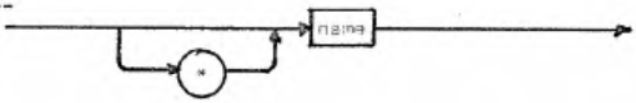
(ii) Two additional OPTIONS parameters are provided
PROCESSES|NOPROCESSES
SLICING|NOSLICING

Appendix : Syntax Diagrams .

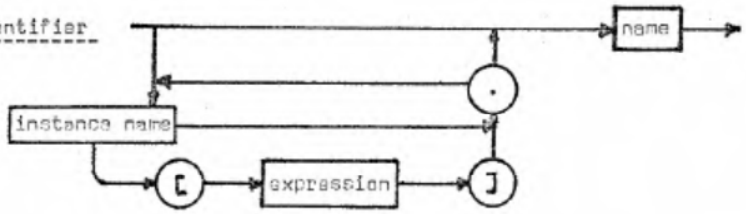
Name



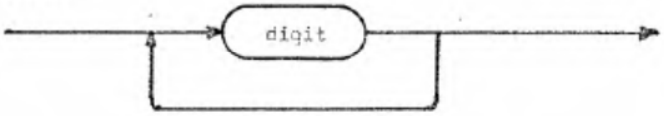
Starred name



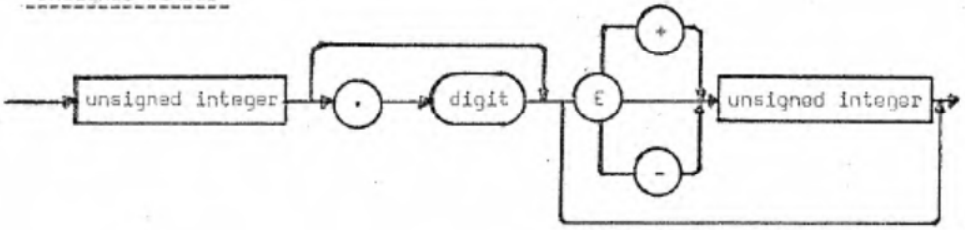
Identifier



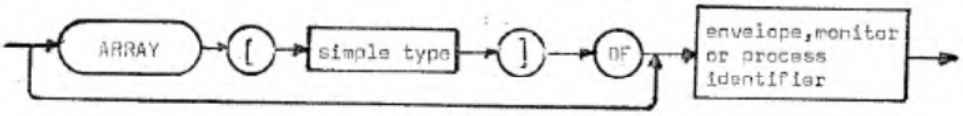
Unsigned integer



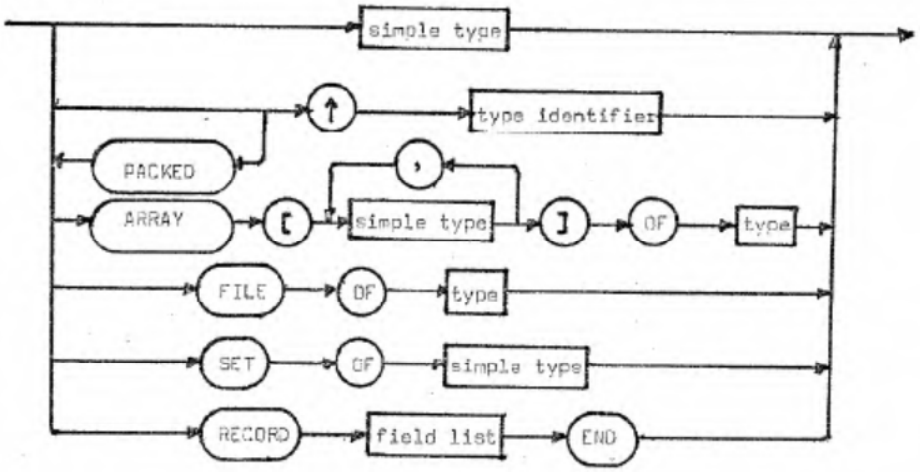
Unsigned number



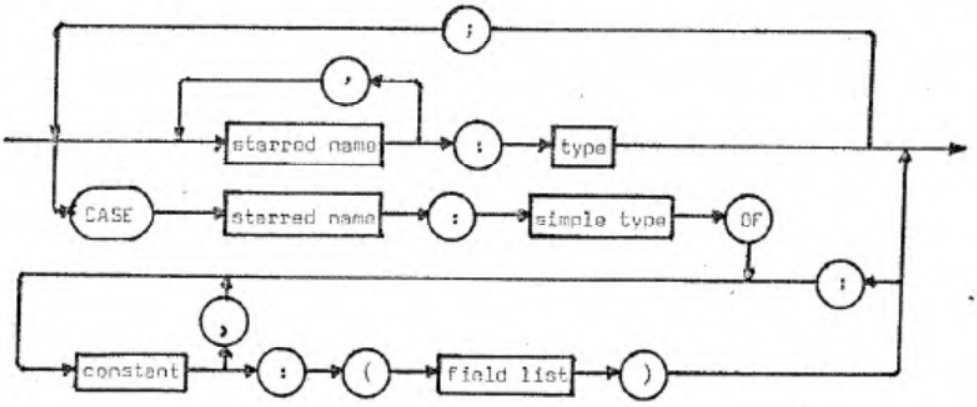
Instance type



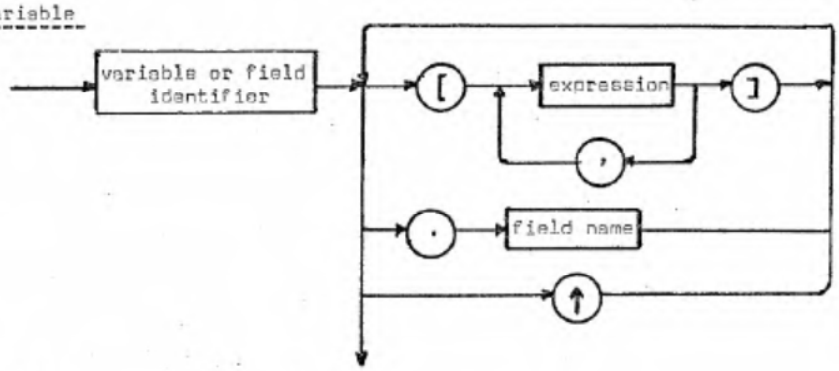
Type



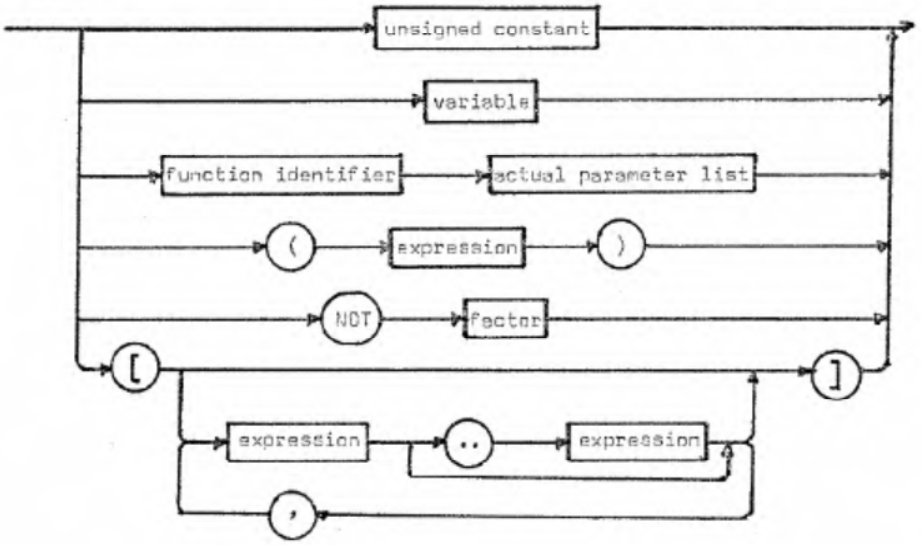
Field list



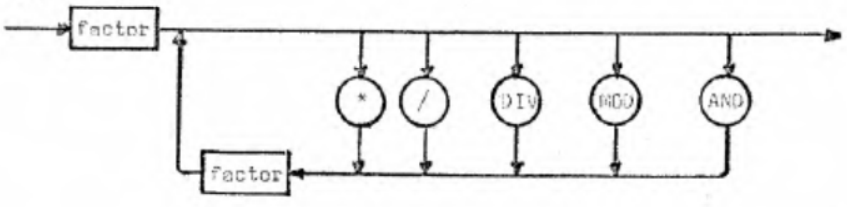
Variable



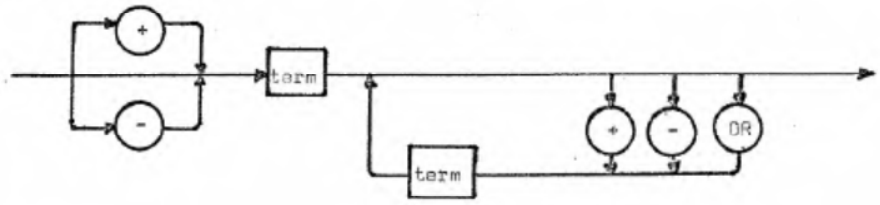
Factor



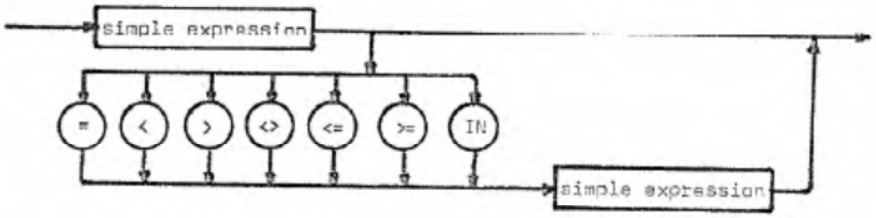
Term



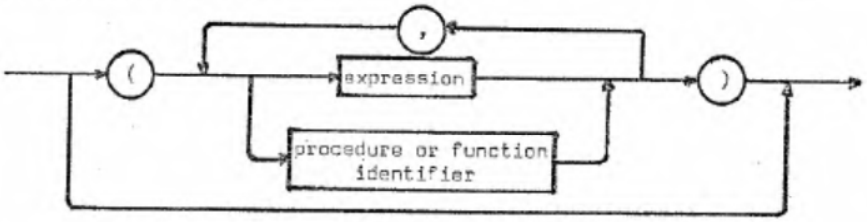
simple expression



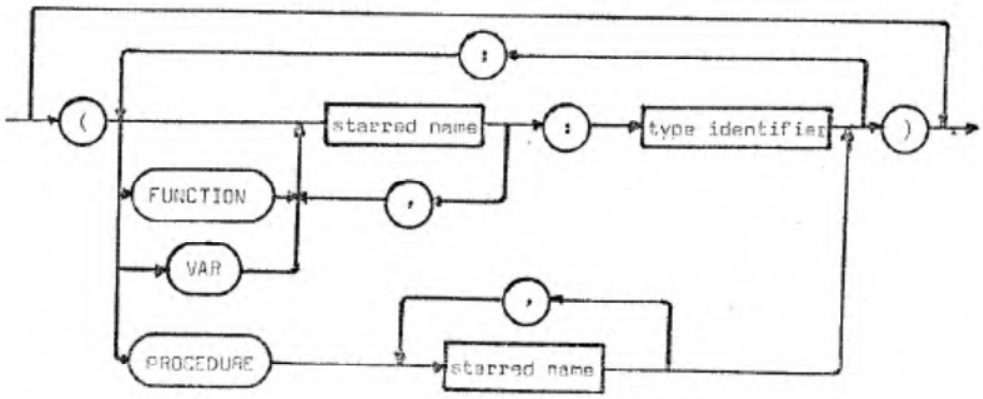
Expression

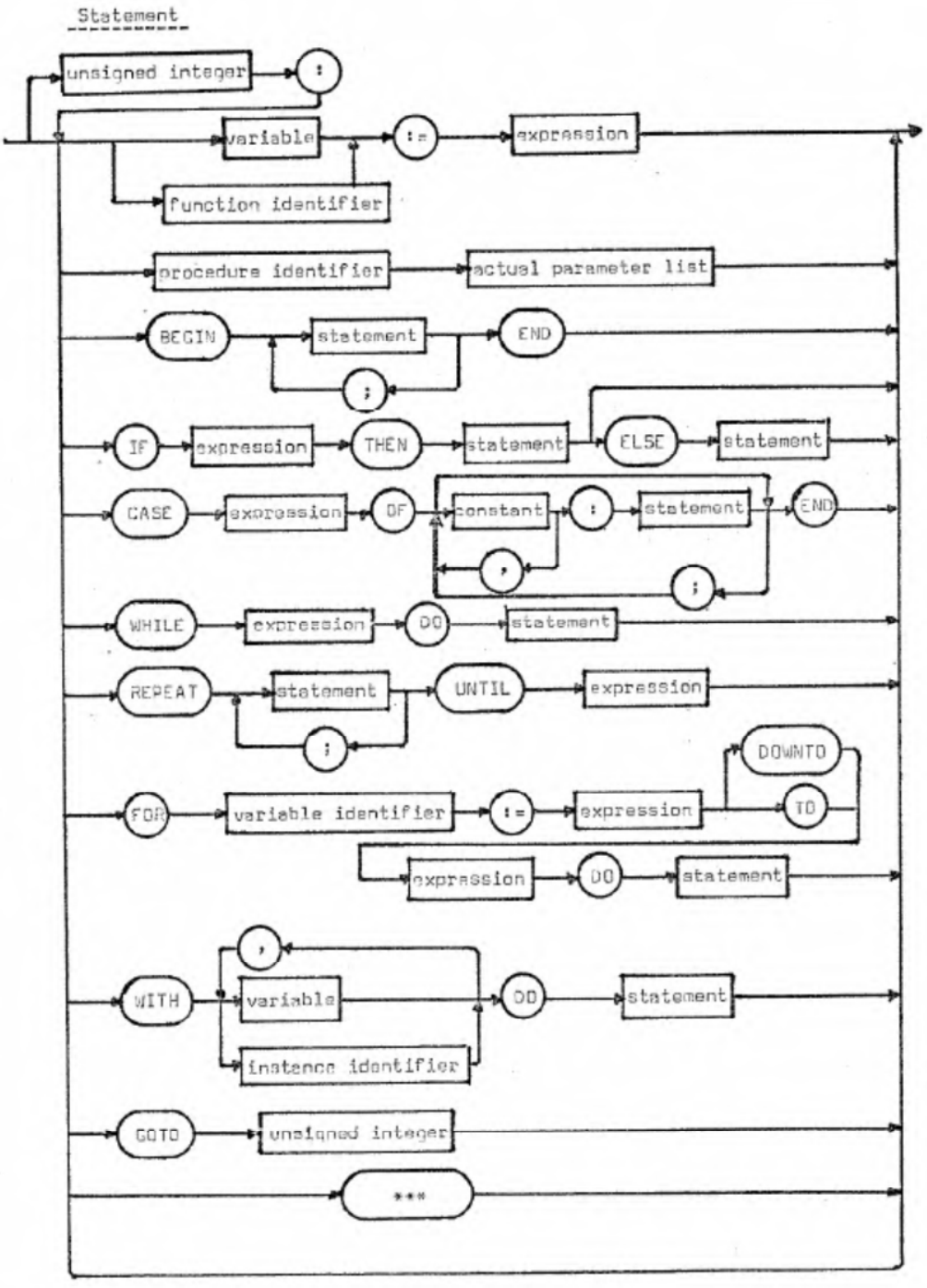


Actual parameter list

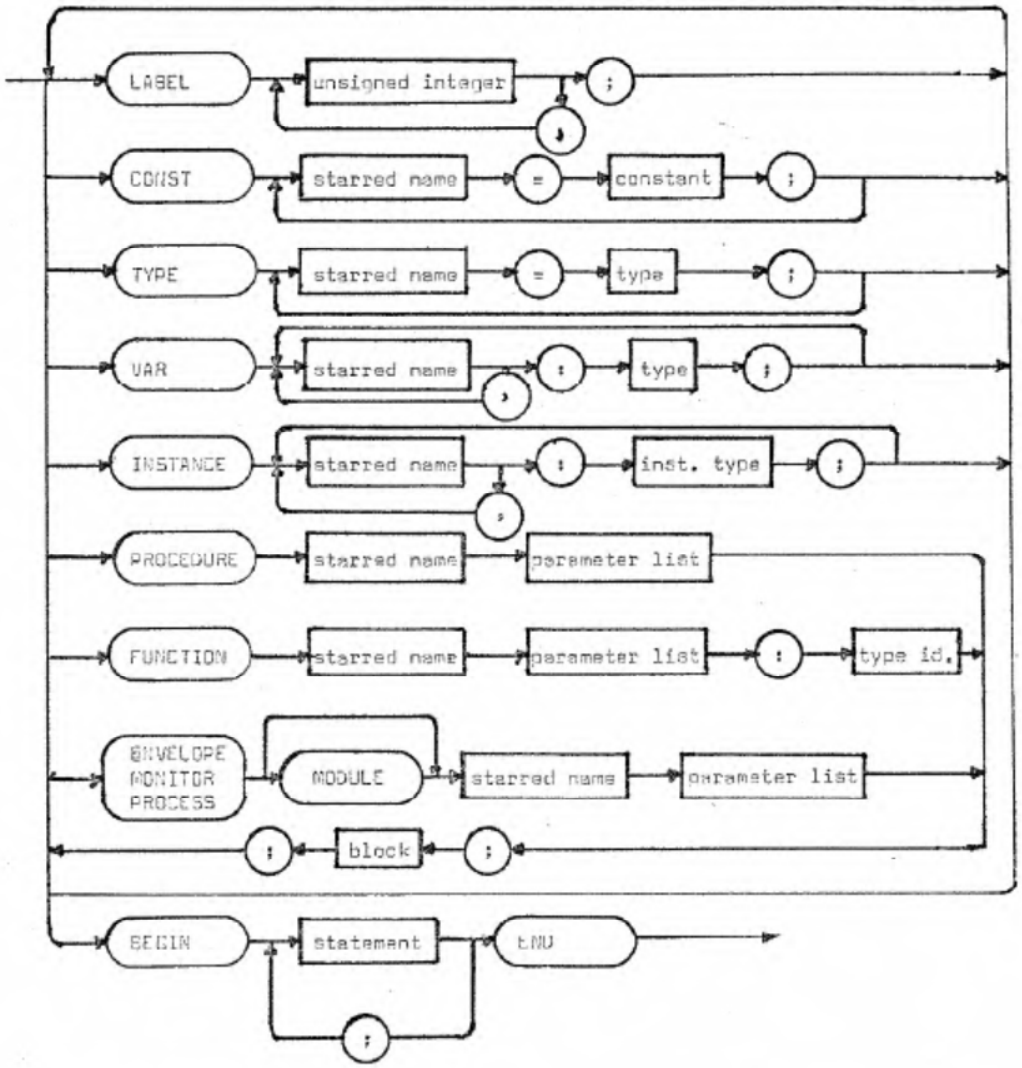


Parameter list




Statement
unsigned integer
:
variable
:=
expression
function identifier
procedure identifier
actual parameter list
BEGIN
statement
END
;
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
:
statement
END
,
;
WHILE
expression
DO
statement
REPEAT
statement
UNTIL
expression
;
DOWNTO
FOR
variable identifier
:=
expression
TO
expression
DO
statement
,
WITH
variable
DO
statement
instance identifier
GOTO
unsigned integer
***

Block



Program