RCSL No: 42-i1539

Edition: October, 1980

Author: Jan Bardino

Title:

PASCAL80 User's Guide

A/S REGNECENTRALEN af 1979

Keywords:

High level language, PASCAL80, Concurrent programming, Standard PASCAL.

Abstract:

This is a tutorial for the language PASCAL80. The manual contains a description of PASCAL80 and examples of programs and program constructs.

(88 printed pages).

42-i 1341

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS (continued) PAGE

1. INTRODUCTION 1.

This first edition of the PASCAL80 User's Guide is mainly based on extracts from earlier PASCAL80 papers such as the Report [1], and some preliminary introductions, and information published in Danish under the common little "PASCAL80 NYT".

This manual is directed to those who have previously acquired some familiarity with computer programming, and now wish to get acquainted with the programming language PASCAL80. The style of the manual is that of a tutorial, i.e. a demonstration of the language features by means of examples.
For a concise ultimate of the language definition the PASCAL80 REPORT [1] may be used and the actual implementations are described in xx-PASCAL80-REFERENCE manuals, by now xx is RC3502 and RC850.

Since PASCAL80 is based directly on Wirth's Standard Pascal [2] familiarity with that language means that the parts concerning sequential programs, i.e. most of the declarations and control statements, may be well known. PASCAL80 can be characterized as Standard Pascal without files but extended with communication primitives to be used to connect concurrent process incarnations.

For programmers acquainted with ALGOL, or FORTRAN it may prove helpful to glance at PASCAL80 in terms of these other languages. For this purpose we list the following characteristics of PASCAL80.

1. Declaration of variables is mandatory.
2. Certain key words (e.g. PROCESS, BEGIN) are "reserved" and cannot be used as identifiers. In this manual they are written with capital letters.
3. The semicolon (;) is considered as a statement separator, not a statement terminator.
4. The standard data types are those of whole numbers, the logical values, the characters, semaphores, shadows, references, and pools. The basic data structuring facilities include the array, the record (corresponding to COBOL's "structure"),

the pool, and the set. These structures can be combined and nested.

5. The facilities of the ALGOL switch and the computed go to of FORTRAN are represented by the case statement.
6. The for statement corresponding to the DO loop of FORTRAN, may only have steps of 1 (TO) or -1 (DOWNTO) and is executed only as long as the value of the control variable lies within the limits. Consequently, the controlled statement may not be executed at all.
7. There are no conditional expressions and no multiple assignments.
8. Procedures and functions may be called recursively.
9. There is no "own" attribute for variables (as in ALGOL). Parameters are called either by value or by reference; there is no call by name.
10. The "block structure" differs from that of ALGOL insofar as there are no anonymous blocks, i.e. each block is given a name, and thereby is made into a routine.
11. PASCAL80 is equipped with semaphores as a synchronizing tool and message buffers as a communication tool.
12. Concurrent process incarnations are synchronized by means of signal-wait primitives.

2. BASIC DEFINITIONS 2.

2.1 Vocabulary 2.1

The basic vocabulary consists of language symbols and user defined symbols. The language symbols are reserved words (key words) and punctuation marks:

AND	ELSE	LABEL	PROCESS
ARRAY	END	LOCK	RECORD
AS	EXPORT	MOD	REPEAT
BEGIN	EXTERNAL	NOT	SET
BEGINBODY	FOR	OF	THEN
CASE	FORWARD	OR	TO
CHANNEL	FUNCTION	OTHERWISE	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	POOL	VAR
DO	IN	PREFIX	WHILE
DOWNTO	INCLUDE	PROCEDURE	WITH

+	-	*	/	"	'	<	>
<>	<=	>=	(	)	(.	.)	↑
=	:=	:=:	.	,	:	;	..
***	(*	*)	!	?	<*	*>	#

The user may not use the reserved words in a context other than that explicit stated in the definition of PASCAL80; in particular, these words may not be used as identifiers.

2.2 Syntax Diagrams 2.2

The syntax of PASCAL80 is defined graphically by syntax diagrams. A syntax diagram consists of arrows, language symbols, and names of syntax diagrams. A PASCAL80 program is syntactically correct if it can be obtained by traversing the syntax diagrams. A traversal must follow the arrows. The name of a syntax diagram indicates a traversal of the corresponding diagram. The result of a traversal is the sequence of language symbols encountered in the traversal.

The following is an example of a syntax diagram.

while statement:

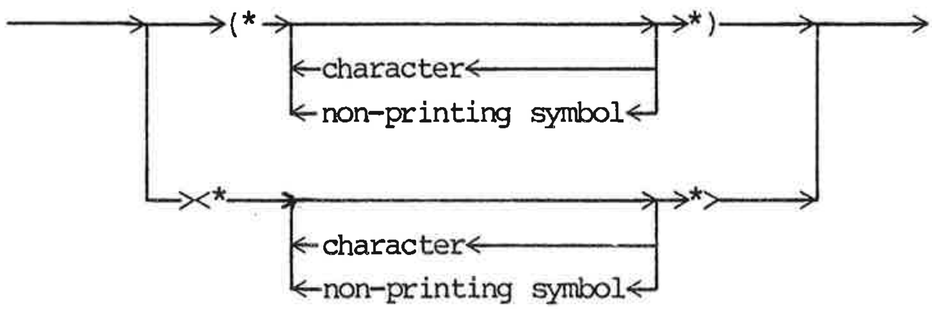
———>WHILE———>expression———>DO———>statement———>

The syntax diagram defines the name (while statement) and syntax of language construct. The name is used when the construct is referred to elsewhere in the text or in other syntax diagrams. Language symbols are either names in capital letters (e.g. WHILE) or punctuation marks (e.g. :=).

Constructs defined by other syntax diagrams are given by their names in small letters (e.g. expression). To be able to distinquish between several occurrences of a construct, its name my be subscripted.

2.2.1 Comments 2.2.1

Comment:

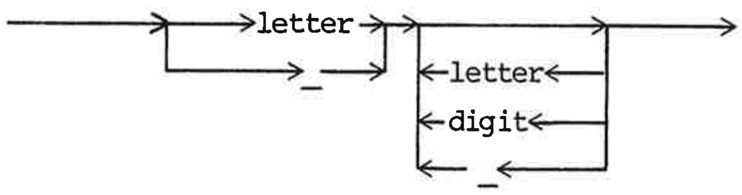


Comments may be inserted between any two identifiers, numbers or special symbols. A comment does not affect the execution of the program.

2.2.2 Identifiers 2.2.2

Names denoting labels, constants, types, variables, processes, and routines are called identifiers. They must begin with a letter or an underscore which may be followed by any combination and number of letters, digits, and underscores. Contrary to Standard PASCAL all the characters of an identifier are recognized as significant. Small and big letters are handled as being the same in identifiers.

identifier:



letter is A,B,...,Å,a,b,c,...,å
digit is 0,1,2,...,9

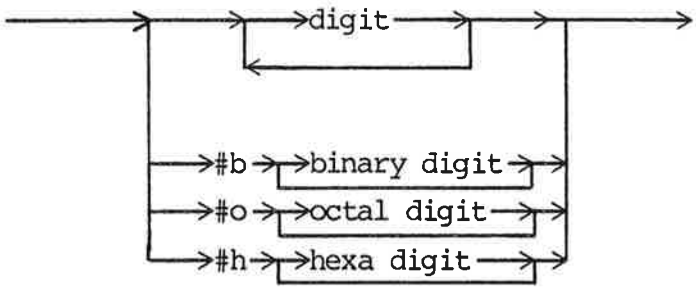
Examples of legal identifiers:
step use_count Local_Message
____very___special___defined___identifier

Note: "Local_Message" is identical to "local message", "LOCAL_MESSAGE", and any other combination of small and big letters.

2.2.3 Numbers 2.2.3

At label can be either an identifier or a numeric value in PASCAL80, this is in contrast to Standard Pascal where label is demanded to be an unsigned integer.

numeric value:



binary digits are 0..1
octal digits are 0..7
hexa digits are 0..9 and a..f

Example of legal numbers:
7913 0033 #b101 #hff00 #o7654

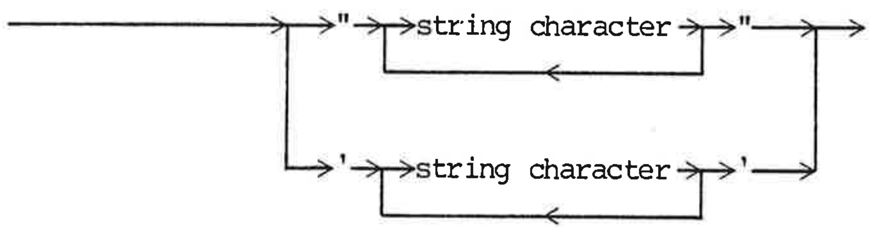
2.2.4 Separators 2.2.4

Blanks, nl's, ff's and comments are considered as separators. Separators can appear between any two consecutive language symbols.
No separator may occur within an identifier, number, numeric value, or language symbol. At least one separator must appear between any pair of consecutive identifiers, character strings, numbers, numeric values, or language symbols.

2.2.5 Strings of Characters 2.2.5

A character string is a sequence of characters enclosed by quote marks, both single and double quote marks are legal but the end mark must match the start mark.

Character string:



String characters are the printable subset of the alphabet, excluding newline (nl) and form feed (ff), i.e. ' ', '!', ...,'~'

Examples of legal strings:
"abcd", " '~' is a strange character", '"'

Note: If a string surrounded by single quote marks is to contain a quote mark or a string surrounded by double quote marks is to contain the surrounding quote mark, then this quote mark is to be written twice, for example """" is equivalent to '"', and '''' is equivalent to "'".

2.3 Fundamental Concepts 2.3

This section gives a brief explanation of a few concepts and the context in which they are used. The complete description of all PASCAL80 concepts is given in the following sections.

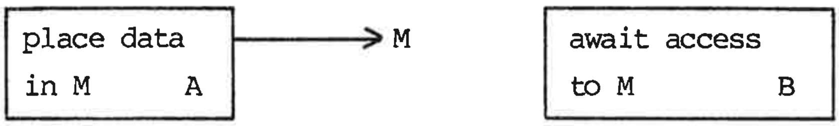
A program consists of a number of processes. Each process is a description of some actions and a description of a data structure. An incarnation of a process is the execution of the actions on a private data structure. Many incarnations can be executed concurrently.

Actions are described by statements. The actions of one incarnation are executed one at a time in the order defined by the statements. The actions manipulate the data structure, which is described by a number of variables. A variable has a name and a type. The type describes the set of values the variable can hold when the program is executed. There is a number of predefined types (integer, char, boolean, reference, semaphore, and shadow).

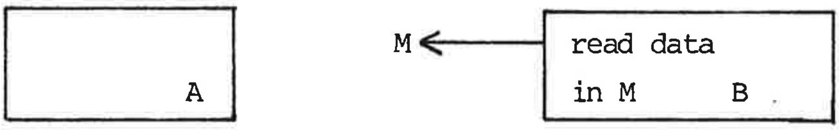
New types are defined either by listing their values or by combining several types into a structured type.

A number of statements and declarations can be combined into a routine declaration. Activation of a routine is described by routine calls (statement).

Process incarnations communicate by exchanging messages. A message can be accessed by one incarnation at a time.



Time T: A has exclusive access to the message M.



Time T + 1: B has exclusive access to M.

Variables of the two predefined types reference and semaphore are used for accessing and exchanging access to messages.

The value of a reference variable is either a reference to a message or nil (representing "no reference"). A message can be accessed through at most one reference variable at a time. Since process incarnations access messages through reference variables only, mutually exclusive access to messages is secured.

Incarnations exchange access to messages by means of queue semaphores. An incarnation places a message in a semaphore from which another incarnation can get access to it. Variables of type semaphore can be declared in any process. A semaphore variable may be accessible by many incarnations simultaneously.

Processes can be nested and a process which is declared within another process is a sub-process (of the surrounding process).

An arbitrary number of incarnations of sub-processes (children) can be created, they are all controlled by the parent. Incarnations are created and removed dynamically.

A process can have <u>formal parameters</u>. When an incarnation of the process is created a number of <u>actual parameters</u> is given. Incarnations communicate through common semaphore variables only. In this way a process determines the communication paths of sub-processes. Note, however, that the controlling process incarnation need not participate in the comminication.

3. THE PASCAL80 LANGUAGE 3.

This chapter consists of descriptions of the different components of a PASCAL80 process. First an example which shows the structure of a complete process definition, and after the example is given a more precise description of the syntactical definition, of the different parts of the process definition.

3.1 The Process Structure 3.1

A PASCAL80 process consists of declarations of constants, types, variables, routines, labels, and some statements that operate on the declared objects.

This is an outline of a PASCAL80 process:

```
PROCESS catalog;
  CONST
     idlength = 10;
     catalogsize = 256;
  TYPE
     identifier = ARRAY (1 .. idlength) OF char;
  .
  .
  .
  VAR
     name: identifier;
     found: boolean;
     index: integer;
  FUNCTION hash (id: identifier): integer;
     VAR
       key, next: integer;
       ch: char;
```

```
BEGIN (* body of function hash *)
   key:= 1;
   next:= 0;
   REPEAT
      next:= next + 1;
      ch:= id (next);
      IF ch <> sp
         THEN key:= key * ord (ch) MOD catalogsize + 1;
   UNTIL (ch = sp) OR (next >= idlength);
   hash:= key;
END; (* of hash *)
.
.
.
BEGIN (* main program *)
.
.
.
   index:= hash (name);
   REPEAT
   .
   .
   .
      found:= ——
      .
      .
      .
   UNTIL found;
   .
   .
   .
END.
```

The process contains a declaration of

- two contants: idlength with the value 10 and catalogsize with the value 256

- a type: identifier which is an array of characters

- three variables: name which can hold a value of type identifier, found which can hold a value of type boolean, and index which can hold an integer.

- a function hash which maps an identifier to an integer.

The function has a formal parameter id and three local variables key, next, and ch. The assignment statement: index:= hash (name) contains a call of the function; the result of the function is assigned to the variable index.

All declared objects have names: catalog, idlength, catalogsize, identifier, name, found, index, hash, id, key, next, and ch. These names are defined by declarations before they are used in statements.

3.2 The Process Heading 3.2

The process heading consists of an identification of the process to be declared and a parameter description. The process in the example of section 3.1 has no parameter, the identification is "catalog".

process heading:

——>PROCESS——>process name——>formal parameter——>

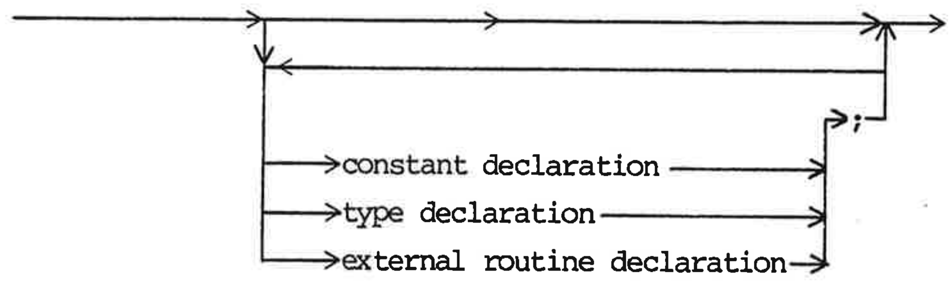
(formal parameters are described in subsection 3.3.5).

Declarations common to more processes may be defined in a context, and it may be specified in the call of the compiler which context(s) to include. The syntax is:

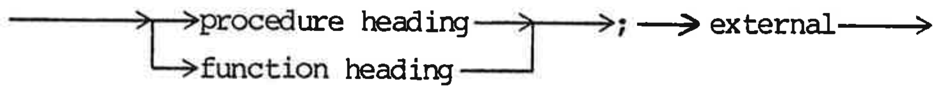
context:

——>context name——>;——>context declarations——>.——>

context declarations:



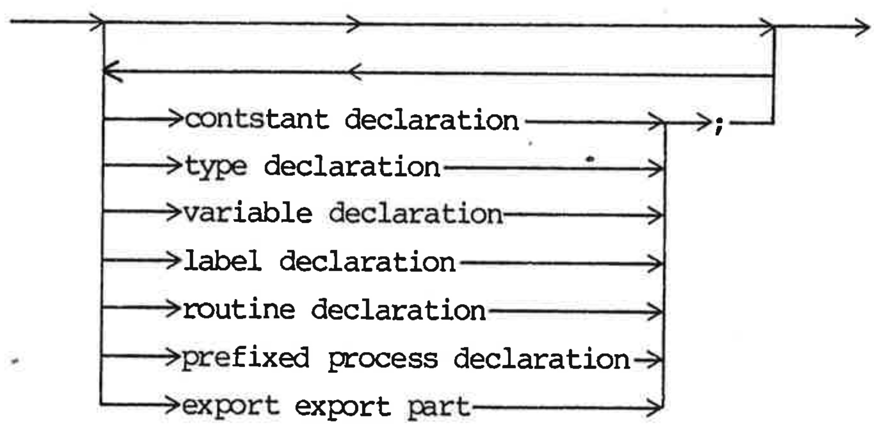
external routine declaration:



3.3 The Declaration Part 3.3

The declarations of a program serves as a description of the data which are manipulated by the actions performed by the program.

declarations:



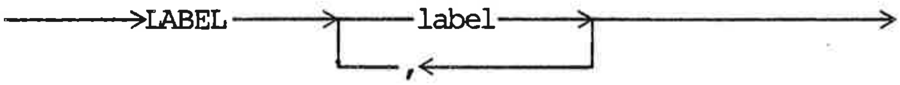
The order of declarations is only restricted of the demand for definition before use.

3.3.1 Label Declaration Part

3.3.1

A label is an identification of a statement, it can be either a number or an identifier. Every label must be declared.

label declaration:



label:

Example:

LABEL 7913, even_action;

A label is denoted by the identifier or the integer value of the number. (GOTO statement 3.4.10)

A label is defined by a labelled statement.

labelled statement:

Labels must be defined and used in the scope (not block) where they are declared. A label may only be defined once in a scope. Scope is defined in subsection 3.3.5.

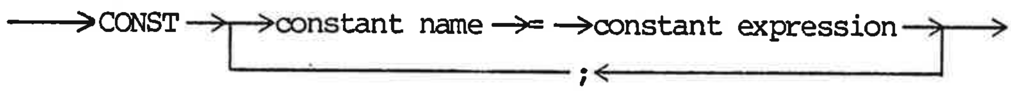
example:

error_action: exception (error_code);

3.3.2 Constants Definition Part

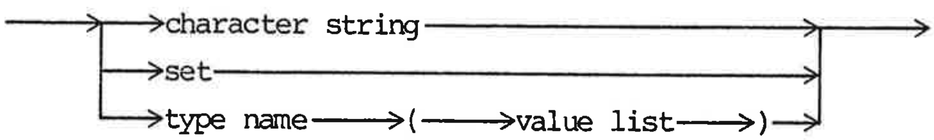
If a value is used serveral times in a program, it is useful to declare a constant with this value. In the program the constant is used to denote the value.

constant declaration:

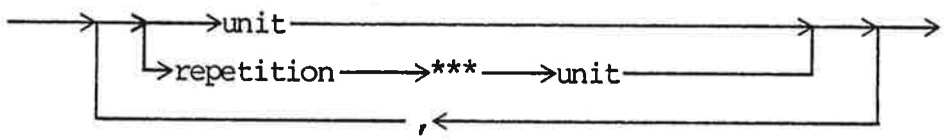


The constant expression is an expression the value of which may be computed at compile time, i.e. each operand must be a constant or a symbolic value. A very convenient feature of PASCAL80 is the so-called structured value which may be used for defining constants and for initialization of structured variables.

structured value:



value list:



Units of a value list are given one at a time (separated by ,) or by repeating a unit. The value of repetition specifies how many times the unit is repeated.

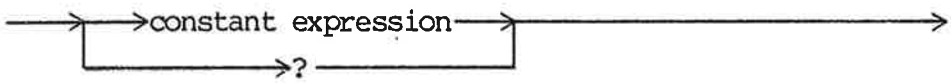
A structured value is built as follows:

- type name denotes a record type:
  There must be a unit for each field and the first field gets the value of the first unit, the second field the value of the second unit etc. The repetition cannot be used.

- type name denotes an array type:
  There must be a unit for each element and the first element gets the value of the first unit, the second element the value of the second unit etc.

repetition:

unit:



The unit "?" specifies <u>no value</u>, i.e. the component is skipped, its type is compatible with any type. This element is necessary to specify values of components which have no symbolic representation, e.g. values of shielded types. The no value element can only be used in value lists.

Example:

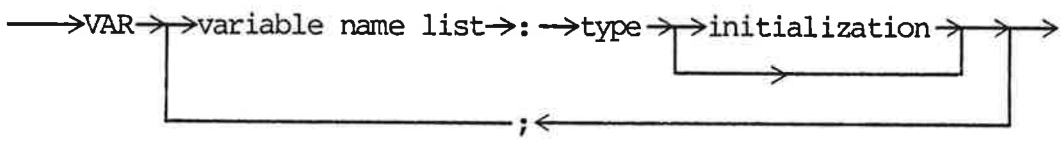
```
CONST
   catalogsize = 256;
   test = true;
   nul = 0;
```

If these values are changed, only the constant declaration needs to be changed.

3.3.3 Variable Declaration Part

A declaration of a variable must specify the name and type of the variable.

variable declaration:



variable declaration

initialization:

The type of the expression must be compatible with the type of the variable.

The value specified by the constant expression becomes the initial value of all variables in the variable name list.

Example:

```
VAR
    found: boolean:= false;
    index: 1 .. catalogsize;
    name: identifier:= identifier (idlength***sp);
```

The type defines which values a variable can hold. The variable found can hold the boolean values false and true, the initial value is false. The variable index can hold an integer in the range 1 to 256 (catalogsize = 256) the value of index is undefined until first assignment. The value of a variable is changed by an assignment:

```
found:= true;
index:= index MOD catalogsize + 1;
```

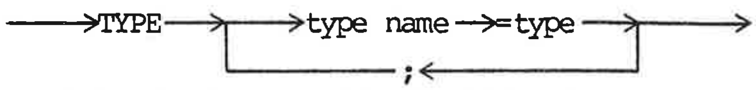
3.3.4 Type Definition Part 3.3.4

All data which is manipulated by a PASCAL80 program has a type. All operands (variables, constants, values etc.) have a fixed type and for each operator and statement there are strict rules defining which types of operands it accepts.

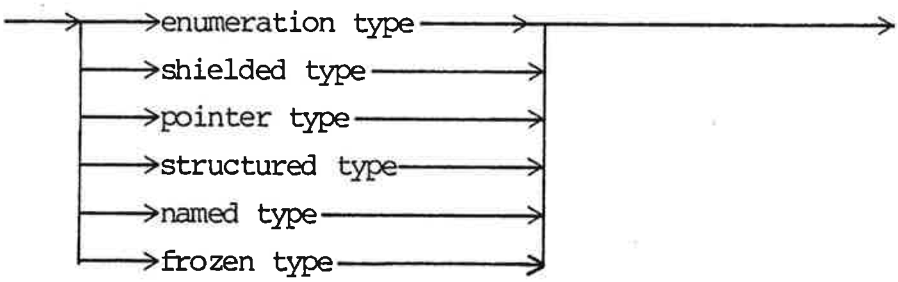
3.3.4.1 Types 3.3.4.1

A type is a set of values and a method of accessing these values. There is a number of predefined types: integer, char, boolean, semaphore, reference, and shadow. New types are named and defined by type declarations.

type declaration:



type:



Type declarations may not be recursive, except in a type declaration:

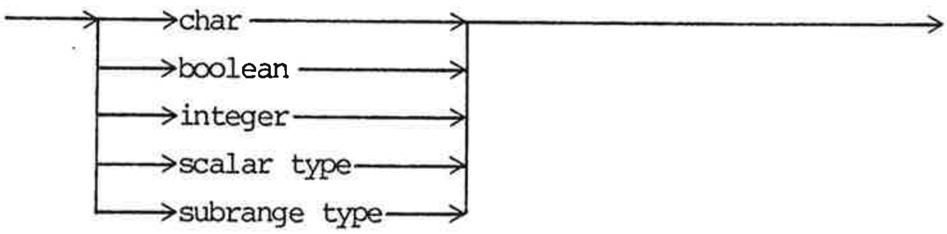
TYPE name = t;

where t may contain the pointer type ↑ name as an element or field type;

Enumeration Types

An enumeration type consists of a finite, totally ordered set of values. Furthermore, there is a mapping from the set of values to the integers.

enumeration type:

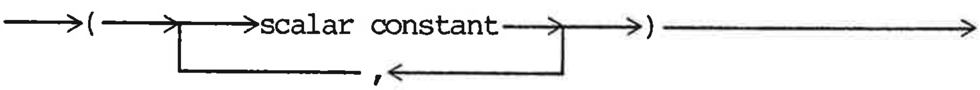


The three predefined types char, boolean, and integer are described below.

Scalar Types

A scalar type is a sequence of values (scalar constants). A scalar type is declared by listing its values in increasing order.

scalar type:



scalar constant:

    >identifier   >

A scalar constant is an identifier appearing in the declaration of a scalar type T. The type of the scalar constant is T. Consider the following scalar type $(e_0, e_1, \dots e_n, e_{n+1}, \dots e_N)$, then e_{n-1} is the predecessor of e_n and e_{n+1} is the successor of e_n the ordinal value of e_n is n. The predecessor of e_0 and the successor of e_N are undefined.

Example:

```
TYPE
     device = (drum, tape, disk);
```

The type device has the values drum, tape, and disk.

The Type Char

The type char is a predefined enumeration type. Its values are the (Danish) ISO characters.

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
10	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
20	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
30	rs	us	sp	!	"	£	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	Æ	Ø	Å	↑	_		a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	æ	ø	å	~	del		

The characters are numbered and the ordinal number of a character is the sum of its row and column number in the above table. The ordinal values define the ordering of the characters.

' ' (sp), '!' '"', ... ' ' are printing characters.

The Type Boolean

The type boolean is a predefined scalar type, defined as:

```
TYPE     boolean = (false, true);
```

Subrange Types

A type can also be declared as a subrange of an already defined type.

A subrange type is a sub-sequence of an enumeration type.

subrange type:

```
----------->min bound ------> .. ------>max bound ------------->
```

min bound, max bound:

```
----------->expression------------------------------------------>
```

The min and max bounds must be of the same enumeration type.

example:

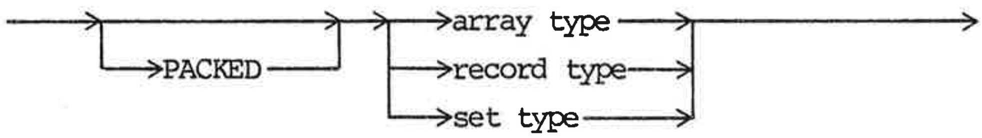
```
TYPE
     index = 1 .. catalogsize;
     small_letters = "a" .. "å";
     byte = 0 .. 255;
```

These declarations restrict the set of values of the type to the specified range.

Structured Types

A structured type is a composition of other types. There are three kinds of structured types: array, record, and set.

structured type:

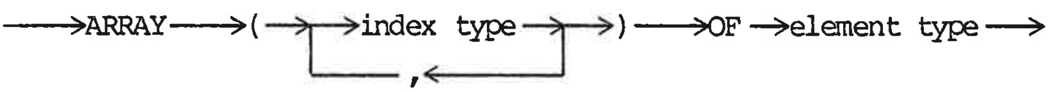


A structured type has a number of component types.

Array Types

An array consists of a number of elements of the same type. The number of elements is specified by an index type.

array type:



index type, element type:

The index type must be an enumeration type or the name of an enumeration type.

Example:

```
TYPE
     identifier = ARRAY (1 .. idlength) OF char;
     count = ARRAY (letters) OF integer;
VAR
     id: identifier;
```

The elements of the array "id" have indices from 1 to 10 (idlength). The value of an element can be changed:

```
id (5):= "x";
```

An array value (whole array) can also be constructed and manipulated:

```
CONST
  blank = "          ";
.
.
.
IF id <> blank
  THEN ...
```

Then array type

```
ARRAY (t1,t2) OF t3
```

is a shorthand for the type

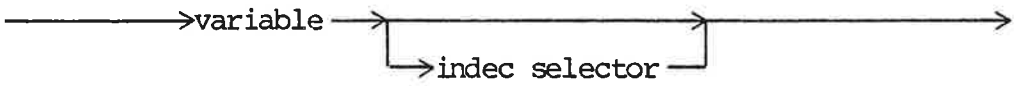
```
ARRAY (t1) OF ARRAY (t2) OF t3.
```

This is a multi-dimensional array. The number of index types is the dimension of the array.

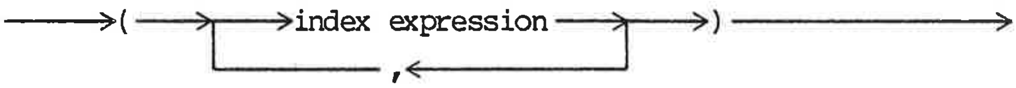
The name of an array variable denotes the whole array. An element is accessed by the array variable followed by an index enclosed in parentheses. An index consists of a number of index expressions. The number of index expressions must be less than or equal to the dimension of the array.

If the element type itself is structured, the component types of the array type are the component types of the element type.

array variable:



index selector:



index expression:

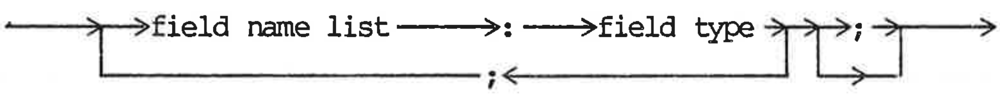
The type of each index expression must be compatible with the corresponding index type.

Record Types

A record consists of a number of fields. Each field has a name and a type.

record type:

field list:



field name list:

Example:

```
TYPE
   catalogentry = RECORD
                    id: identifier;
                    hashkey: integer;
                    medium: device;
                    addr: range;
                  END;
VAR
   element: catalogentry;
```

The record of type catalogentry has four fields: id, hashkey, medium, and addr. These fields are of the type identifier, integer, device, and range respectively. The value of a field can be changed or read:

```
element.medium:= drum;
IF element.id (1) ≤ "a" THEN ...
```

Values of record type are structured values (see subsection 3.4.1):

```
element:= catalogentry (blank, 0, drum, 16712);
```

Set Types

The values of a <u>set type</u> are the subset of some enumeration type.

set type:

——→ SET ——→ OF ——→ element type ——→

element type:

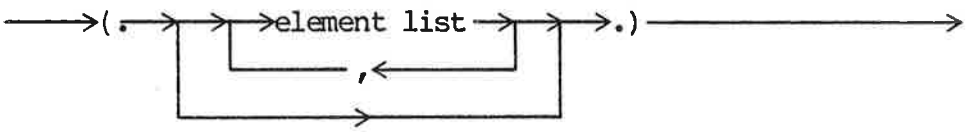
——→ type ——→

The element type must be an enumeration type or the name of an enumeration type.

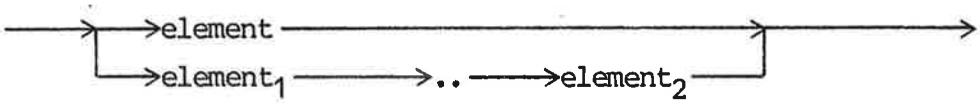
A set element is a value of the element type. The component type of a set is the element type.

Values of set type are written as a list of set elements.

set:



element list:



element:

All elements in a set must be of the same type and these must all be compatible with the element type. The empty set is denoted (..). The type of (..) is compatible with any set type.

A variable of type set can be given a value:

```
digits:= (. "0" .. "9".);
```

The operators on set operands are + (union), * (intersection), - (difference), and IN (membership).

Example:

```
TYPE
    characters = SET OF char;
VAR
    digits, letters: characters;
FUNCTION nextid: identifier;
    VAR
        i: 1 .. idlength;
        ch: char;
BEGIN (* body of nextid *)
  i:= 1;
  nextid:= blank;
  ch:= getchar;
  IF ch IN letters THEN
     WHILE (ch IN (letters + digits)) AND (i <= idlength) DO
     BEGIN
       nextid (i):= ch;
       i:= i + 1;
       ch:= getchar;
     END;
END; (* of nextid *)
```

(Note, getchar is not a PASCAL80 primitive).

As the above examples show, a type (predefined or programmer defined) can be used for constructing values of the type, defining constants, and declaring variables.

<u>Pointer Types</u>

The values of pointer type are pointers to variables or nil (no pointer).

pointer type:

The value nil belongs to every pointer type; it does not point to any variable.

Assignments can be made to variables of pointer types.

The variable pointed to by a pointer (value) is denoted by a variable of pointer type followed by an arrow (↑).

For the use of pointer variables and pointer types see the example in section 4.2 (under "ref").

Frozen Types

In PASCAL80 the programmer has the possibility of declaring variables and parameters as "read only" i.e. the variable/parameters cannot be changed inside the process/routine with the read-only declaration.

frozen type:

————>!————>base type————————————>

base type:

————>type ————————————————>

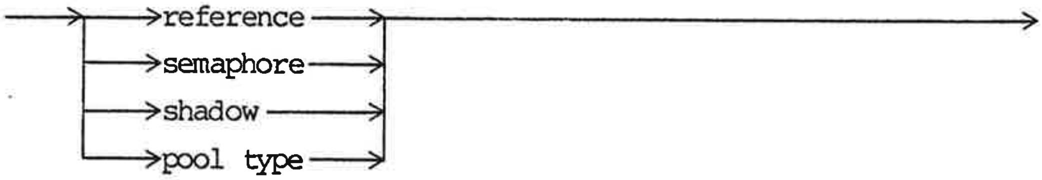
A variable of a frozen type must not be used as the lefthand side of an assignment, in an exchange statement, or as a variable parameter, unless the formal parameter is of the same frozen type!

A frozen type is compatible with its base type. The component types of a frozen type are the component types of the base type.

Shielded Types

Variables of shielded types enable a process incarnation to communicate with and control other incarnations.

shielded type:



The values of shielded types cannot be accessed directly. They are protected against malicious or accidental misuse. Therefore, the assignment statement cannot be applied to variables of shielded types. The exhange statement is provided instead (see subsection 3.4.3).

The Type Reference

The values of type reference are references to messages or nil (no reference). A message is always accessible through exactly one reference variable.

The syntax used to denote the accessible fields of a message header is derived from considering the type reference as a predefined pointer type (see section 5.3):

```
TYPE reference = ↑ message;
```

The type of the message header is:

```
TYPE
     message = RECORD (*message header*)
                 size, messagekind: !integer;
                 u1, u2, u3, u4: 0..255;
                 (*owner, answer:  semaphore;
                 data:  message data;
                 other implementation dependent fields*)
               END;
```

The interpretation of size and messagekind is implementation dependent. The owner, answer, and data fields cannot be used directly.

Messages

Process incarnations communicate by exchanging access to messages which hold data. When a process has access to a message it can place data in or read data from the message. A message can be accessed by one incarnation at a time (see section 2.3)

A message consists of a message header and message data (possibly empty). A header message is a message with no message data.

The Type Semaphore

A queue semaphore consists of a sequence (fifo) of messages and a set of waiting process incarnations. One of these is always empty. The values of type semaphore are queue semaphores.

The semaphore is open when the set of waiting incarnations is empty and the sequence of messages in non-empty. When the sequence is empty and the set of waiting incarnations is none-empty, the semaphore is locked. If both are empty, the semaphore is passive.

These concepts are described in details in the sections concerning process communication (chapter 4).

Variables of type semaphore (or variables with semaphore components) are restricted only to be declared in the declarations of a process and not in the declarations of a routine.

The Type Shadow

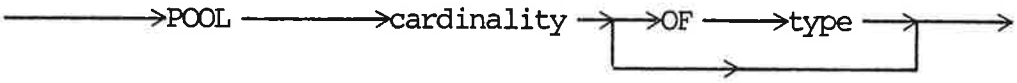
The values of type shadow are references to process incarnations or nil (no reference). Initially, a shadow variable is nil.

A shadow variable is given a value by creating a new incarnation. The incarnation is controlled through the shadow variable. The predefined routines for controlling incarnations are described in the sections concerning process control (chapter 5).

Pool Types

A pool consists of a number of messages.

pool type:



cardinality:

Initially a pool consists of a number of messages. The number is the value of cardinality (expression) which must be a positive integer. Each of the messages can hold a value from type. If no type is specified, the messages have headers only.

With each variable of type pool an anonymous semaphore is associated. This is the owner semaphore of all messages in the pool. A message is allocated from the pool by the predefined procedure alloc (see chapter 4).

3.3.4.2 Type Compatibility 3.3.4.2

In PASCAL80 any operand has a fixed type which can be determined statically. The type of constants, variables, and formal parameters is specified in their declaration.

```
CONST
   length = 16;
TYPE
   word = ARRAY  (0 .. length - 1) OF boolean;
CONST
   nul = word (length *** false);
VAR
   status: word;
```

The constant length is of type integer. The construct "word (length *** false)" is a value of type word where all elements are false. The constant nul and the variable status are both of type word.

The type of an expression is determined by the types of its operands and the way they are combined by operators. The addition of two integers, i.e. length + 1, gives a result of type integer, comparison of two integers, i.e. j <= length, gives a result of type boolean, conjunction of two booleans gives a boolean result, i.e. found AND (j <= length) etc. The operator AND can only be applied to boolean operands, the operator / can only be applied to integer operands etc. Similar restrictions are put on the operands used in all other constructs. In a while statement, for example, an expression of type boolean must be given:

```
WHILE found AND (j <= length) DO ...
```

A value of some type T can be assigned to a variable of the same type.

```
status:= nul;
```

<u>The types of two operands are the same only if their type names (identifiers) are the same, or the two operands are declared in the same list.</u>

```
VAR
     status: word;
     mask  : word;
     result: ARRAY (0 .. length - 1) OF boolean;
     trap  : ARRAY (0 .. length - 1) OF boolean;
     rec1, rec2: record ... end;
```

The types of status and mask are the same, but none of them are the same as the type of result. Consequently:

```
     status:= mask;
```

is a valid assignment, but

```
     status:= result;
```

is not a valid assignment. Furthermore, the type of trap is neither the same as the type of result nor the same as the type of status and mask. And the operands rec1 and rec2 are of the same type.

The type t_1 is compatible with the type t_2 if:

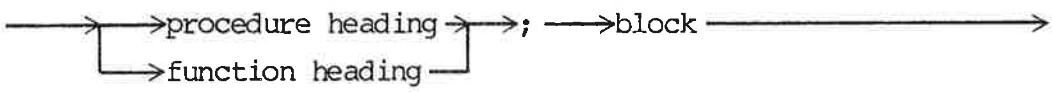
- t_1 and t_2 are the same named type
- t_1 is a subrange or t_2 or t_2 is a subrange of t_1
- t_1 is SET OF b_1 and t_2 is SET OF b_2 and b_1 is compatible with b_2
- t_1 is ! t_2
- t_1 is↑ t and t_2 is↑ t where t is a type name
- t_1 and t_2 are of pool type

Note, that the relation compatible is not symmetric. If the type t_1 is compatible with the type t_2, a value of type t_1 can be assigned to a variable of type t_2.

3.3.5 Routine Declaration Part

A number of statements and declarations can be combined into a routine. When the routine is called, the data structure defined by the declarations is allocated and the statements are executed. A routine is either a procedure or a function.

routine declaration:



procedure heading:

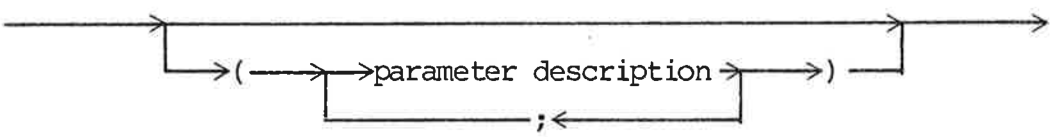
function heading:

The type of a function cannot be a shielded type.

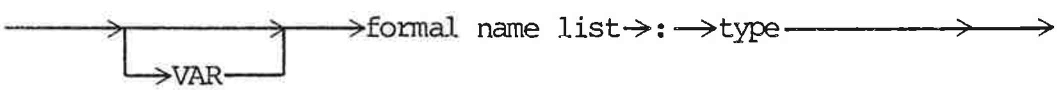
Formal Parameters

The formal parameters specify the interface between a block and the surrounding. For each formal parameter is given its kind, formal name, and type.

formal parameters:



parameter description:



formal name list:

If VAR is specified the parameter is of kind variable: a *var parameter*; otherwise the parameter is of kind value: a *value parameter*.

A formal parameter is used as a declared variable of the specified name and type.

Parameters with components of shielded type must be of kind variable.

Example:

```
TYPE
     parity = (even, odd);
     frame = 0 .. 31;
FUNCTION frame_parity (arg: frame): parity;
CONST
   table = (. 0,3,5,6,9,10,12,15,
              17,18,20,23,24,27,29,30 .);

            (* The set table contains all values of type
               frame with even parity *)

BEGIN
  IF arg IN table
     THEN frame_parity:= even
     ELSE frame_parity:= odd;
END;
```

A routine can have local declarations as in this case the constant table. A function returns a result, this result is the value assigned to the funcion name, e.g. frame_parity:= even. The function has a parameter with the name arg and the type frame. The type of the result is parity.

Routine declarations can be nested:

```
TYPE
   parity = (even, odd);
   byte = 0 .. 255;
FUNCTION byte_parity (arg: byte) : parity;
   TYPE
       frame = 0 .. 31;
   FUNCTION frame_parity (arg: frame) : parity;
       CONST
          table = (. 0,3,5,6,9,10,12,15,
                     17,18,20,23,24,27,30 .);

   BEGIN (* frame parity *)
      IF arg IN table
         THEN frame_parity:= even
         ELSE frame_parity:= odd;
   END;
BEGIN (* byte parity *)
   IF frame_parity (arg MOD 32) = frame_parity (arg DIV 32)
      THEN byte_parity:= even
      ELSE byte_parity:= odd;
END;
```

The declaration of a name in a routine is only valid inside the routine. Outside the routine it is invisible. The constant table can therefore only be applied in the function frame_parity where it is declared. But it cannot be applied in the function byte_parity. Similarly, the type frame is not known outside byte_parity. It can, however, be applied in inner routines such as the function frame_parity. The exact rules about valid contexts for a variable are called the scope rules (see the next subsection).

The scope rules require that a process or routine is declared before it is used. A declaration where the block is a forward block is an announcement of a routine or process declaration which is given textually later, this is a forward declaration. The heading of the declaration must be the same as the heading given in the forward declaration. That is the name, type, and order of the formal parameters must be the same.

3.3.5.1 Scope Rules 3.3.5.1

A scope is one of the following:

- a field list excluding inner scopes,
- a process or routine heading excluding inner scopes,
- a block excluding inner scopes,
- a prefix excluding inner scopes,
- a local declaration (in a lock statement) excluding inner scopes.

A name can be declared once in each scope only. All names must be declared before they are used. If a name is declared both in a scope and in an inner scope, it is always the inner declaration which is effective in the inner scope.

Generally the declaration of a name is effective in the rest of the block where it is declared. Further details for each kind of name is given below.

constant name, type name, variable name, and routine name: The declaration of these names is effective in the rest of the block excluding inner process blocks.

field name: The declaration of a field name is effective in the rest of the block excluding inner process blocks. But the field name can be used in record variables and with-statements only.

scalar constant: The declaration of a scalar constant name is effective in the rest of the block excluding inner process blocks. But used in a type definition of a fieldname the constant name can be used in with-statements only.

label: The declaration of a label is effective in the scope where it is declared.

routine parameter name (implicit and explicit): The declaration of a routine parameter name is effective in the routine block. Note that the declaration is not effective in the routine heading.

process name: The declaration of a process name is effective in the rest of the block where it is declared excluding inner process blocks.

3.3.5.2 Routine Blocks 3.3.5.2

Within the block of a routine a recursive call of the routine can be made.

Processes, exception routines, and variables with semaphore or pool components cannot be declared in a routine block.

3.3.5.3 Functions 3.3.5.3

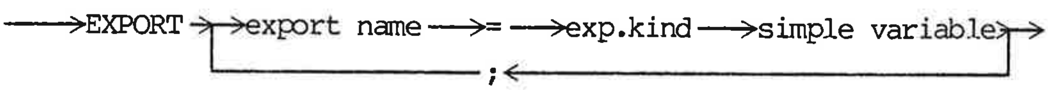
A function name may appear as a variable on the left head side of an assignment. The type in the function heading is the function type, it specifies the range of the function. The value of a function is the dynamically last value assigned to the function variable.

function variable:

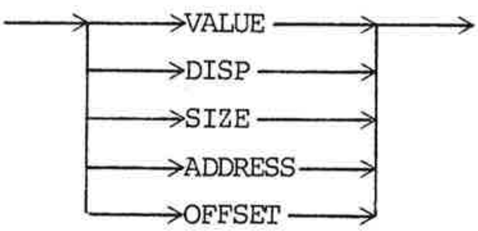
3.3.6 Export Part 3.3.6

Export part is an implementation dependent feature which may open for special linkage editor facilities (see chapter 4).

export part:



exp.kind:



Note:
The five words for exp.kind are not reserved words!

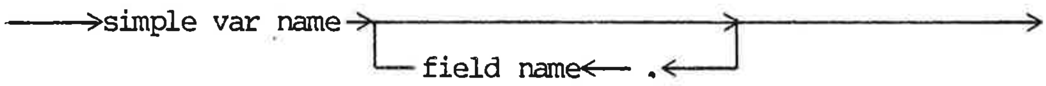
VALUE is for constants only

DISP is for fields only and means displacement relative to record start

SIZE is for constants, entire variables, and fields. SIZE means size (in bytes) of the type which is associated to the "simple variable"

ADDRESS indicates absolute address

OFFSET indicates relative offset in current stack frame simple variable:



simple var name can be either a constant name or a variable name.

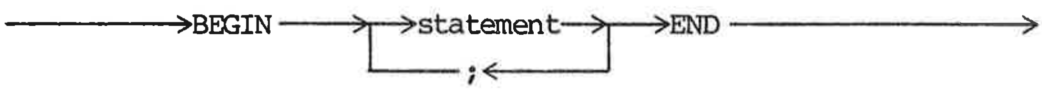
3.4 The Statement Part 3.4

This section contains subsections describing the syntax and the use of the different statements which are included in the language. Most of the statements are also found in Standard PASCAL and may be well known language elements.

3.4.1 Statements 3.4.1

The statements of a process describe the actions which are executed by a process incarnation. These statements are collected in a compound statement.

compound statement:



The statements are executed one at a time in the specified order.

Below, all statement forms are given together with references to their precise description:

statement:

statement	section
compound statement	3.4.1
procedure call	3.4.6
assignment statement	3.4.2
exchange statement	3.4.3
case statement	3.4.5
for statement	3.4.4
if statement	3.4.5
repeat statement	3.4.4
while statement	3.4.4
with statement	3.4.7
goto statement	3.4.10
labelled statement	3.4.10
lock statement	3.4.8
channel statement	3.4.9

3.4.2 Assignment Statement 3.4.2

assignment statement:

——>variable ——>:= ——>expression ——>

The type of the variable must be compatible with the type of the expression.

Assignments can be made to a variable of:

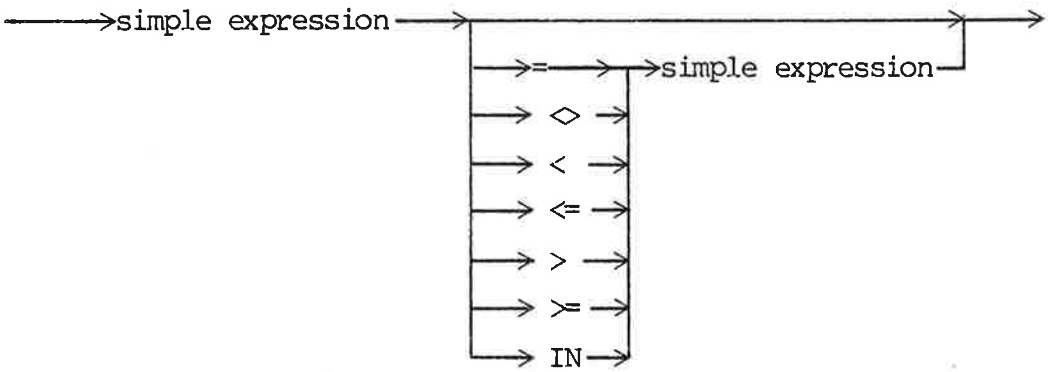
- a simple type,
- a pointer type,
- a structured type where all components are of a simple type or a pointer type.

The assignment statement replaces the current value of the variable by the value of the expression.

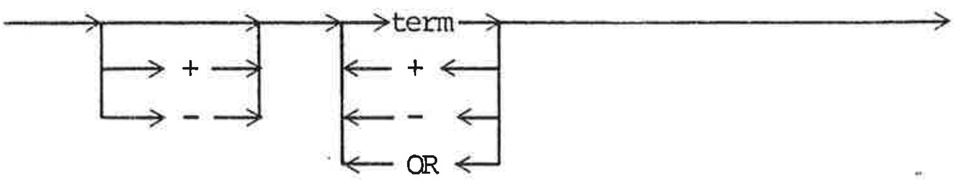
Expressions describe how values are computed. Expressions are evaluated from left to right using the following precedence rules:

NOT has the highest precedence	followed by
*, /, DIV, MOD, AND	followed by
+ , - , OR	followed by
=, <>, <, <=, >, >=, IN	

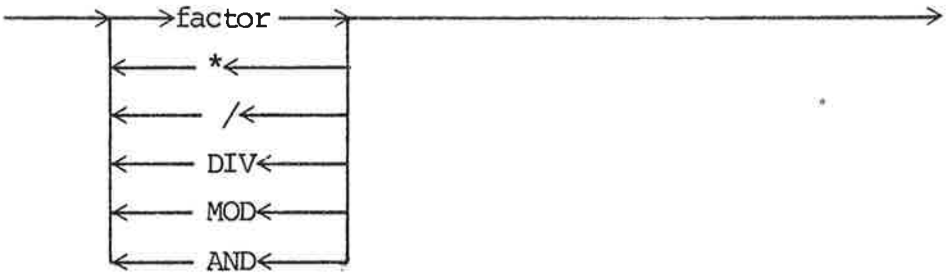
expression:



simple expression

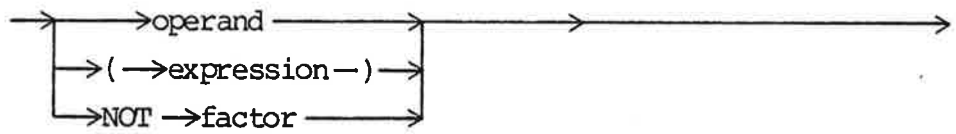


term:



<u>Note</u>: All factors in an expression are evaluated.

Factor:

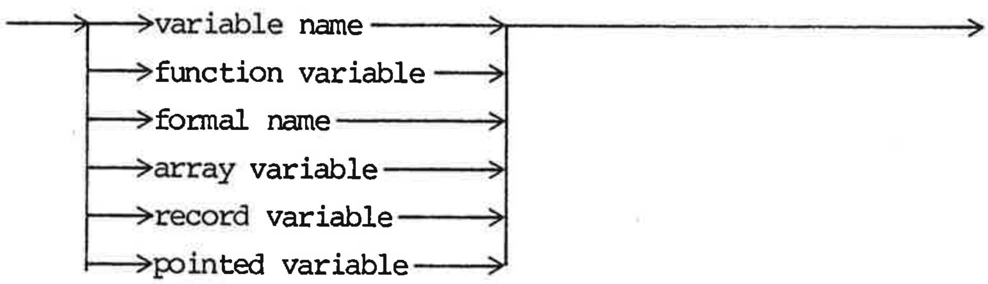


operand:

<u>Variables</u>

The term variable includes declared variables, formal parameters, and function variables. All variables are denoted by their name and possibly a selector.

variable:



array / record / pointed } variables are described in subsection 3.3.4

function variables are described in subsection 3.3.5.3

Variables of shielded and pointer types are implicitly given the following initial values:

semaphore: passive
shadow: nil
reference: nil
pool: a number of messages, determined by the cardinality expression; the contents of these messages are undefined
pointer: nil

3.4.3 Exchange Statement 3.4.3

exchange statement:

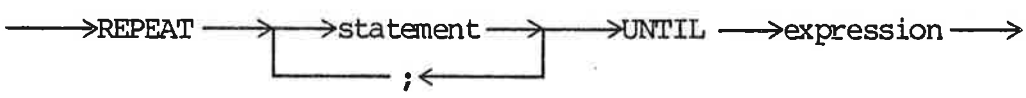
The two variables must either both be of type reference or both be of type shadow.

The exchange statement exchanges the values of the two variables.

3.4.4 Repetitive Statements 3.4.4

Repeat Statement

repeat statement:



The result of the expression must be of type boolean.

The statement sequence is executed one or more times. Every time the sequence has been executed, the expression is evaluated, when the result is true the repeat statement is completed.

While Statement

while statement:

——→WHILE———→expression——→DO——→statement——→

The result of the expression must be of type boolean.

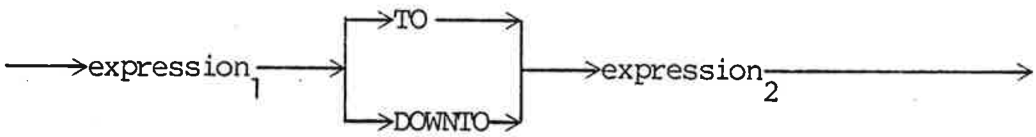
The statement is executed a number of times (possibly zero). The expression is evaluated before each execution, when the result is false, the while statement is completed.

For Statement

for statement:

——→FOR——→variable——→:=——→for list——→DO——→statement——→

for list:



The two expressions must be of the same enumeration type and the type of the variable must be compatible with this.

The selection of the variable cannot be changed in the statement. Hence, if the variable has array indices or pointers, changes to these (in the statement) will not affect the selection.

The statement is executed with consecutive values of the variable The ordinal value of the variable can either be incremented (in steps of 1 (succ)) from expression$_1$ TO expression$_2$, or descremted (in steps 1 (pred)) from expression$_1$ DOWNTO expression$_2$. The two expressions are evaluated once, before the repetition. If the value of expression$_1$ is greater than the value of expression$_2$ and TO is specified, the statement is not executed.

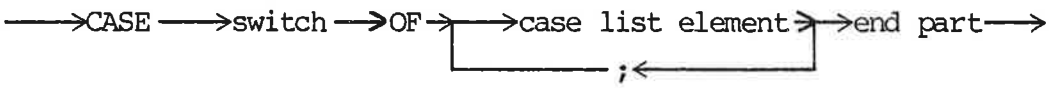
Similarly, if the value of $expression_1$ is less than the value of $expression_2$ and DOWNTO is specified, the statement is not executed.

The value of the variable is dependent of the expressions after the for statement.

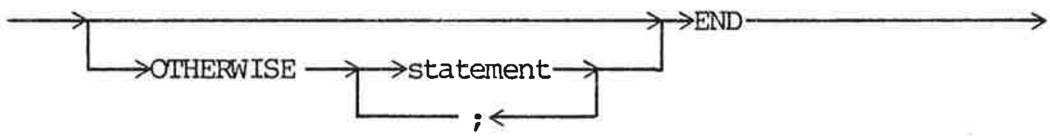
3.4.5 Conditional Statements 3.4.5

Case Statement

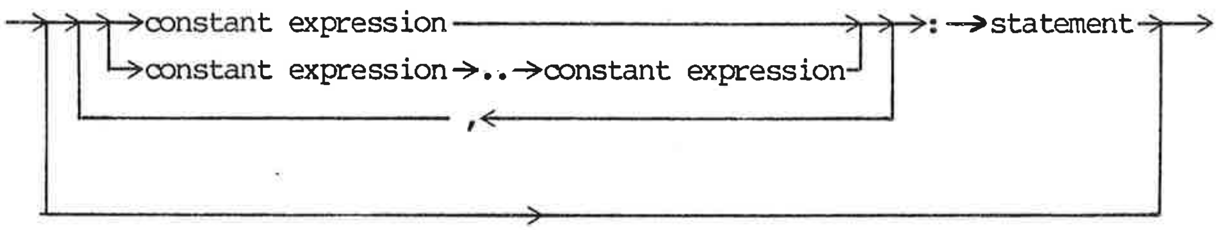
case statement:



end part:



case list element:



switch:

The values of the constant expressions in case list elements are called case labels. All case labels and the switch must be of the same enumeration type and all case labels must be distinct. The switch is evaluated and the statement labelled by the value of the switch is executed. If no such label is present, the statement following OTHERWISE is executed; if OTHERWISE is not specified, an exception occurs.

If Statement

if statement:

The result of the expression must be of type boolean.

Statement$_1$ is executed if the value of the expression is true. If it is false, statement$_2$ (if specified) is executed.

The statement:

IF e_1 THEN IF e_2 THEN s_1 ELSE s_2

is equivalent to:

IF e_1
THEN BEGIN
IF e_2
THEN s_1
ELSE s_2
END

3.4.6 Procedure Call 3.4.6

routine call:

→ routine name → actual parameters →

A routine call binds actual parameters to formal parameters, allocates local variables, and executes the compound statement of the block. When the compound statement is completed, local variables are deallocated and execution is resumed immediately after the routine call. All local reference and shadow variables must be nil when the compound statement is completed, otherwise an exception occurs.

The variables of a routine are associated with a specific call; they exist from the routine call until the compound statement (of the block) is completed. When a routine is called recursively, several versions of the variables exist simultaneously, one for each uncompleted call.
The difference between a procedure and a function is that a procedure call is a statement and a function call a factor (function variable) in an expression.

A function call is an operand in an expression.

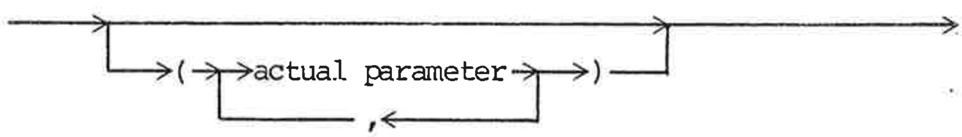
function call:

→ function name → actual parameters →

Actual Parameters

When a process incarnation is created or a routine is called actual parameters are bound to formal parameters.

actual parameters:



actual parameter:

———————→expression——————————————————→

There must be an actual parameter for each explicit formal parameter.

The binding of an actual parameter to a formal parameter depends on the parameter kind:

value: The type of the actual parameter must be compatible with the type of the formal parameter. The value of the actual parameter is evaluated and this value becomes the initial value of the formal parameter. Assignments to the formal parameter within the block does not affect the actual parameter (call by value).

variable: The type of the actual and formal parameter must be the same. The actual parameter must be a variable, the value of this variable becomes the initial value of the formal parameter. Changes to the value of the formal parameter within the block affects the actual parameter directly.

The actual parameter selects a variable, this selection cannot be changed in the block. Hence, if the variable has array indices or pointers, changes to these do not affect the selection (call by reference).

An element or a field of a packed variable cannot be an actual var parameter. The whole packed variable can, however, be an actual var parameter.

3.4.7 With Statement 3.4.7

with statement:

```
────────>WITH──>┬─>record variable─>┬────>DO──>statement────>
                └─────────── ,<─────┘
```

Within the statement fields can be accessed by giving their field names only.

The with statement

```
WITH v1, v2, ..., vn DO s;
```

is a shorthand for the nested with statement shown below.

```
WITH v1 DO
  WITH v2 DO
   .
     .
       .
        WITH vn DO s;
```

The record variable selects a record, this selection cannot be changed in the statement. Hence, if the record variable has array indices or pointers, changes to these (in the statement) will not affect the selection.

3.4.8 Lock Statement 3.4.8

lock statement:

```
───>LOCK─>reference variable─>AS─>local declaration─>DO─>statement──>
```

local declaration:

```
──────>local name──────>:────────>type───────────────────>
```

reference variable:

———>variable ———————————————————>

local name:

———>identifier ——————————————————>

The component types of the type must be simple. The reference variable must refer to a message (must not be nil), otherwise an exception occurs. If the message is too small to represent the specified type an exception occurs.

In the statement local name is a declared variable with the specified type. In the statement the reference variable must not be used as part of an exchange statement or as a parameter to signal, return, release, pop, or push.

The data part of a message is manipulated as a declared variable with the local name. It is always the top in the message stack which is manipulated.

3.4.9 Channel Statement 3.4.9

channel statement:

———>CHANNEL ———>reference variable ——>DO ——>statement ——>

The reference variable must refer to a message (must not be nil). Any implementation may place restrictions on this message. If the message is not of this restricted form an exception occurs.

In the statement the reference variable must not be used as part of an exchange statement or as a parameter to, signal, return, release, pop, or push.

The channel statement controls the handling of peripherals in an implementation dependent way.

3.4.10 Goto Statement 3.4.10

goto statement:

──────>GOTO ──────>label──────────────────────────>

The goto statement, the declaration of the label, and the definition of the label must be in the same scope.

Execution continues at the statement labelled by the label (labelled statement).

Jumps out of a channel or lock statement and jumps out of a routine are not allowed.

labelled statement:

──>label──>: ──>statement ──────>

3.4.11 Standard Routines abs, succ, pred, chr, ord 3.4.11

The absolute value of an integer variable is given as the result of:

```
FUNCTION abs (int: integer) : integer;
```

The successor and predecessor of a variable of scalar type is given as the result of:

```
FUNCTION succ (s: s_type): s_type

FUNCTION pred (s: s_type): s_type
```

s_type may be any scalar type.
succ taken on the last element and pred taken on the first element of a scalar type results in an exception.

The character with the ordinal value n is the result of a call chr(n) where chr is defined as:

```
FUNCTION chr (n: 0 .. 127): char;
```

The ordinal value of a scalar element is retrieved by the function ord:

```
FUNCTION ord (s: s_type): integer;
```

where s_type may be any scalar type.

4. PROCESS COMMUNICATION 4.

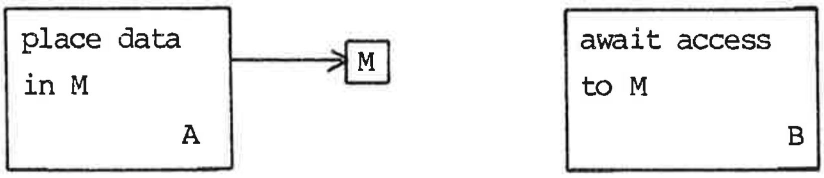
This chapter contains a general description of communication between incarnations, i.e. a description of the language concepts and the tables available for the programmer. After that is a more detailed description of the predefined routines intended for synchronization of the communication between process incarnations.

4.1

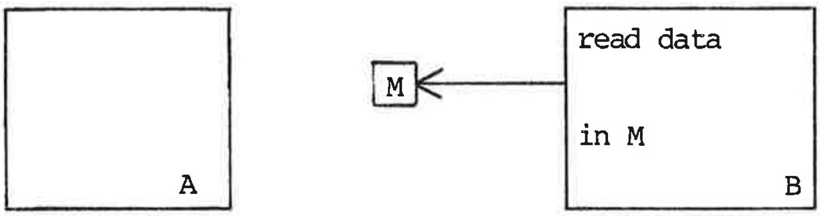
4.1 General Process Communication

A process consists of a number of statements and declarations. An incarnation of a process is the execution of the actions on a private data structure. Many incarnations can be executed concurrently.

Process incarnations communicate by exchanging messages. A message can be accessed by at most one incarnation at a time.



Time T: A has exclusive access to the message M.



Time T + 1: B has exclusive access to M.

The two predefined types reference and semaphore are used for referencing and exchanging access to messages.

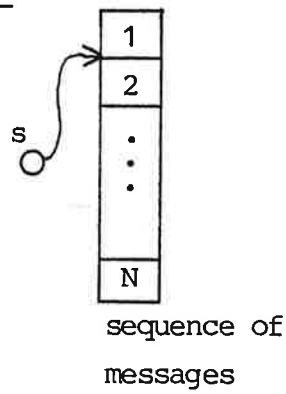
The value of a reference variable is either a reference to a message or nil (representing "no reference"). At most one variable references a message. Since process incarnations access messages through reference variables only, mutually exclusive access to messages is secured.

Queue semaphores are used for exchanging access to messages.

A queue semaphore consists of a sequence (fifo) of messages and a set of waiting process incarnations. One of these is always empty.

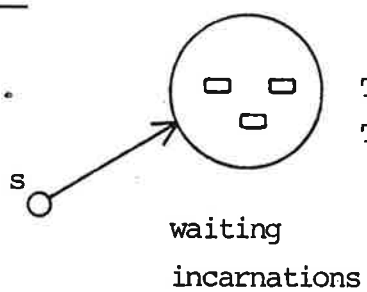
A queue semaphore s can be in one of three states:

open



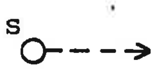
The sequence of messages is not empty.
The set of incarnations is empty.

locked



The sequence of messages is empty.
The set of incarnations is not empty.

passive



The sequence of messages is empty.
The set of incarnations is empty.

Any process may contain declarations of variables of type semaphore, and it may receive semaphore variables as parameters when it is created. All declared semephore variables are initially in the passive state. In constrast to variables of any other type a semaphore variable can be accessible by many process incarnations simultaneously.

example:

```
PROCESS converter (input, output: semaphore);
  VAR
     myown: ARRAY (1 .. 2) OF semaphore;
```

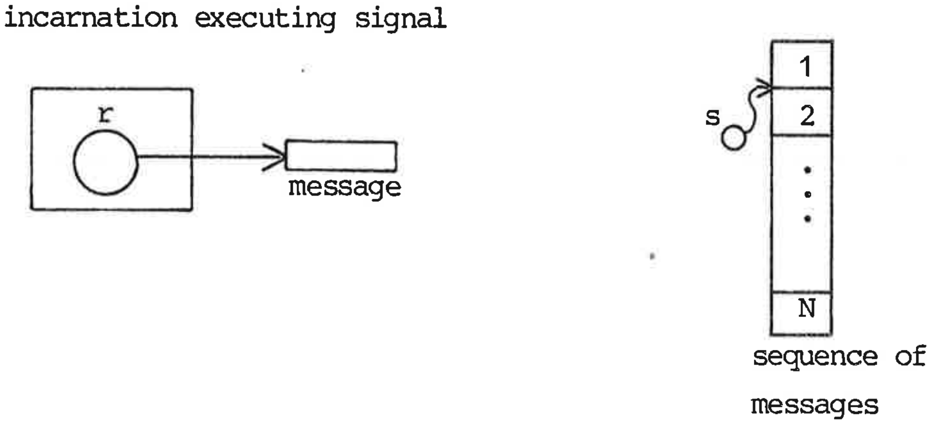
The process converter has access to four semaphores: input, output, myown(1), and myown(2).

The predefined routines signal and wait are used for exchanging access to messages.

```
PROCEDURE signal (VAR r: reference; VAR s: semaphore);
```

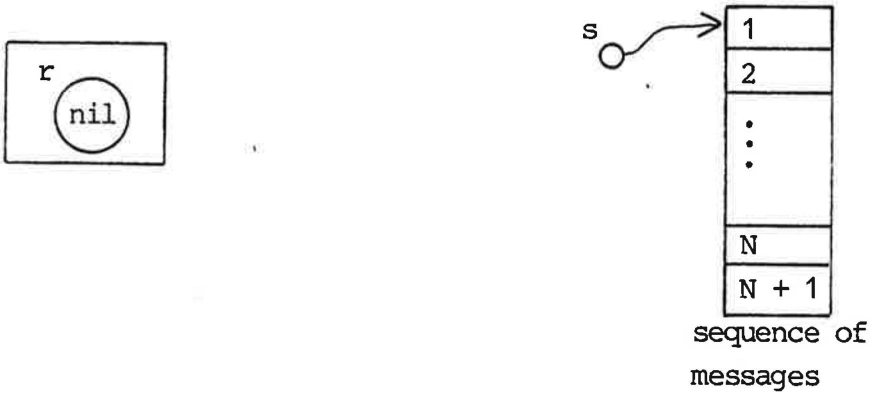
The reference r must reference a message. If the semaphore s is open or passive, the message referenced by r is entered in the sequence of messages belonging to s.

<u>open or passive</u>



<u>prior to signal (r, s)</u>

incarnation executing signal

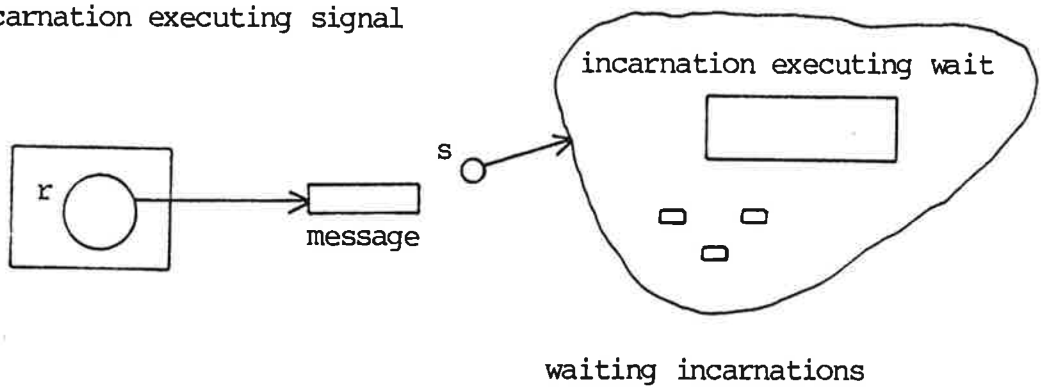


<u>after signal (r, s)</u>

If s is locked one incarnation is removed from the set of waiting incarnations and reactivated. That is, it will be allowed to complete the call of wait which caused it to wait.

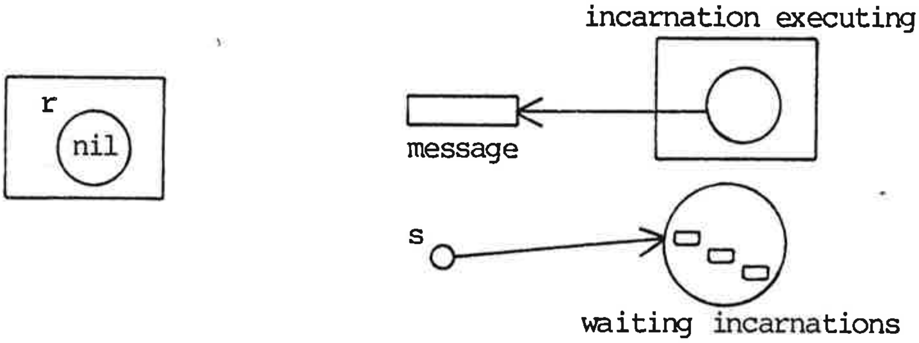
<u>locked</u>

incarnation executing signal



<u>prior to signal (r, s)</u>

incarnation executing signal

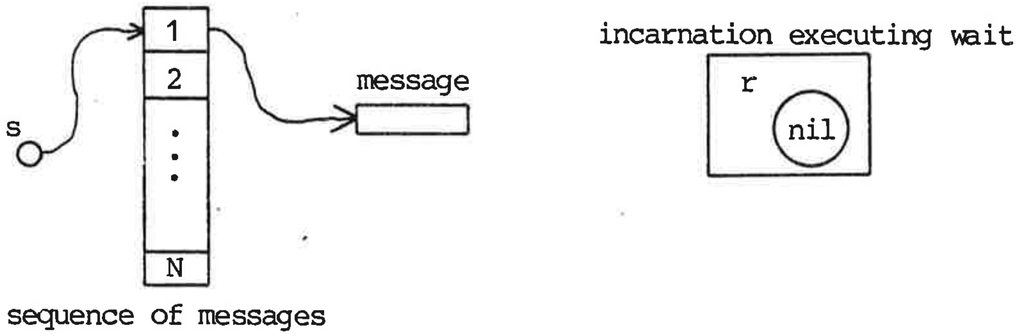


<u>after signal (r, s)</u>

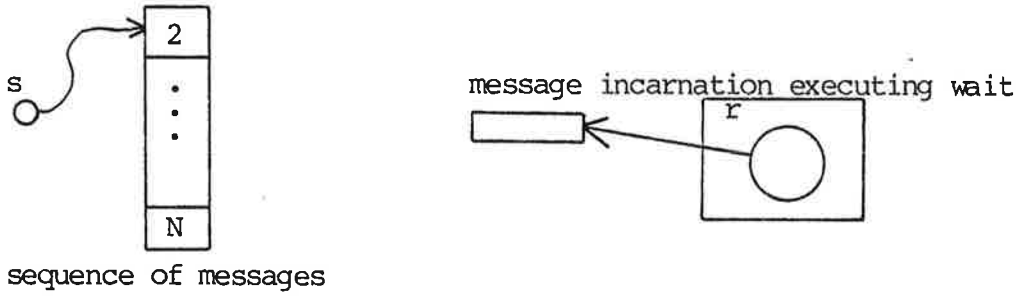
PROCEDURE wait (VAR r: reference; VAR s: semaphore);

The reference r must be nil. If the semaphore is open the first message in the sequence is removed and r becomes a reference to this message.

open



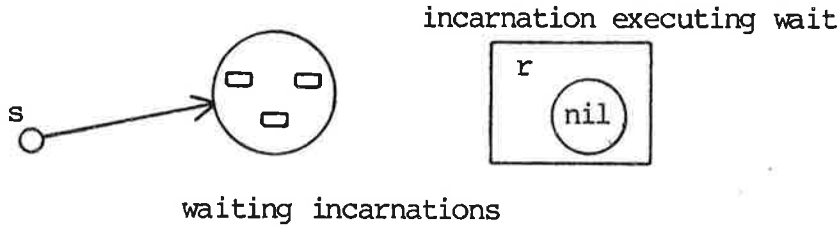
prior to wait (r, s)



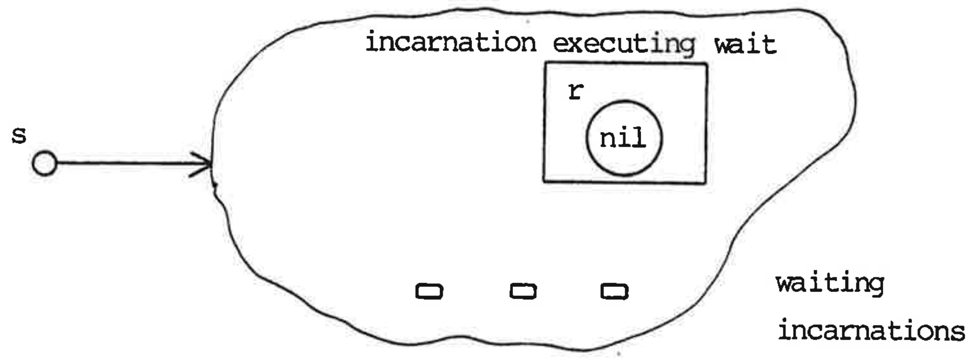
after wait (r, s)

If the state is locked or passive the incarnation is temporarily stopped and entered in the set of waiting incarnations.

locked or passive



prior to wait (r, s)



Incarnation has been stopped during wait (r, s)

It is implementation dependent which one of several waiting incarnations is selected for activation during execution of signal. However, the selection algorithm must be fair: no incarnation may remain waiting indefinitely on a semaphore, provided some other incarnations continue to signal messages to that semaphore.

Execution of wait and of signal is performed indivisibly: e.g. from the moment an incarnation starts execution of a signal on a given semaphore and until the execution is completed, any other incarnation trying to operate on that semaphore variable is delayed.

A process may only inspect or alter the contents of a message in a so-called lock statement. Let r be a reference variable which references a message:

```
LOCK r AS b: t DO s;
```

In the statement s the message referenced by r is manipulated as if it were a variable with the name b of type t.

In the following example there are two processes, one which produces data (e.g. input data) and one which consumes data (e.g. uses the input data in a computation).

```
PROCESS producer (full,
        void: semaphore);
  TYPE
     buffertype = ...;
  VAR
     r: reference;
BEGIN
   REPEAT
     wait (r, void);
     LOCK r AS b: buffertype DO
     BEGIN
        (*...produce data...*)
     END;
     signal (r, full);
   UNTIL...;
END;
```

```
PROCESS consumer (full,
        void: semaphore);
  TYPE
     buffertype = ...;
  VAR
     r: reference;
BEGIN
   REPEAT
     wait (r, full);
     LOCK r AS b: buffertype DO
     BEGIN
        (*...consume data...*)
     END;
     signal (r, void);
   UNTIL...;
END;
```

The allocation of messages is specified by declaring a variable of pool type.

```
VAR
   m: POOL cardinality OF type;
```

Initially, the variable m contains cardinality messages. These messages can hold a value from type. The predefined procedure alloc removes a message from a pool variable:

```
alloc (r, m, s);
```

The reference r must be nil. If the pool of messages is not empty, one of the messages is removed and r references this message. If the pool is empty the process incarnation waits until a message is released (by another incarnation calling the predefined procedure release).

Each message contains information about its origin. The third parameter to alloc must be a semaphore and it becomes the answer semaphore of the message. This is the equivalent of a return address on an envelope of a letter. The answer semaphore is used in the predefined procedure return:

```
PROCEDURE return (VAR r: reference);
```

A call of return is equivalent to a call of signal:

```
signal (r, "answer semaphore")
```

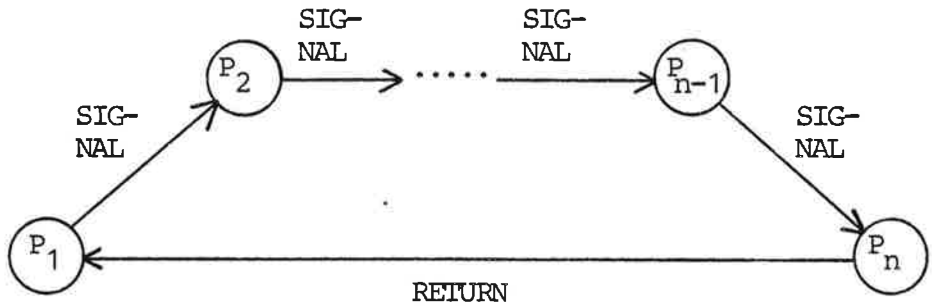
But the answer semaphore is only implicitly available through return.

The following is a revised version of the producer consumer example given above.

```
PROCESS proceducer (stream:
                  semaphore);
  TYPE
     buffertype = ...;
  VAR
     r: reference;
     m: POOL 1 OF buffertype;
     a: semaphore;
BEGIN
  alloc (r, m, a);
  REPEAT
     LOCK r AS b: buffertype DO
     BEGIN
       (*...produce buffer...*)
     END;
     signal (r, stream);
     wait (r, a);
  UNTIL...;
END;
```

```
PROCESS comsumer (stream:
                  semaphore);
  TYPE
     buffertype = ...;
  VAR
     r: reference;

BEGIN
  REPEAT
     wait (r, stream);
     LOCK r AS b: buffertype DO
     BEGIN
       (*...comsume buffer...*)
     END;
     return (r);
  UNTIL...;
END;
```

The following communication flow is possible by means of SIGNAL/RETURN:



A reference variable may point to none or a stack of messages. This is a generalization of the concept of reference variables as described earlier. In general terms a reference variable points to a stack of messages.

When the value of the reference variable is nil, the stack is empty.

A well-defined reference variable points to a stack of reference variables. The message header of the stack elements contains a field, which chains the messages together. This field is called the stack chain. This pointer is nil in the last element of the chain.

Two procedures:

PUSH(<reference variable>,<reference variable>)
POP (<reference variable>,<reference variable>)

are used to manipulate reference variables when interpreted as stack reference variables.

example:

```
VAR element: REFERENCE;
    stack:   REFERENCE;
.
.
.
  PUSH (element, stack);
.
.
.
  POP  (element, stack);
```

PUSH (ref1, ref2)

Before call

ref1 - well-defined reference to element.
The stack chain field in the element must be NIL.

ref2 - well-defined stackop element or
nil if stack is empty:

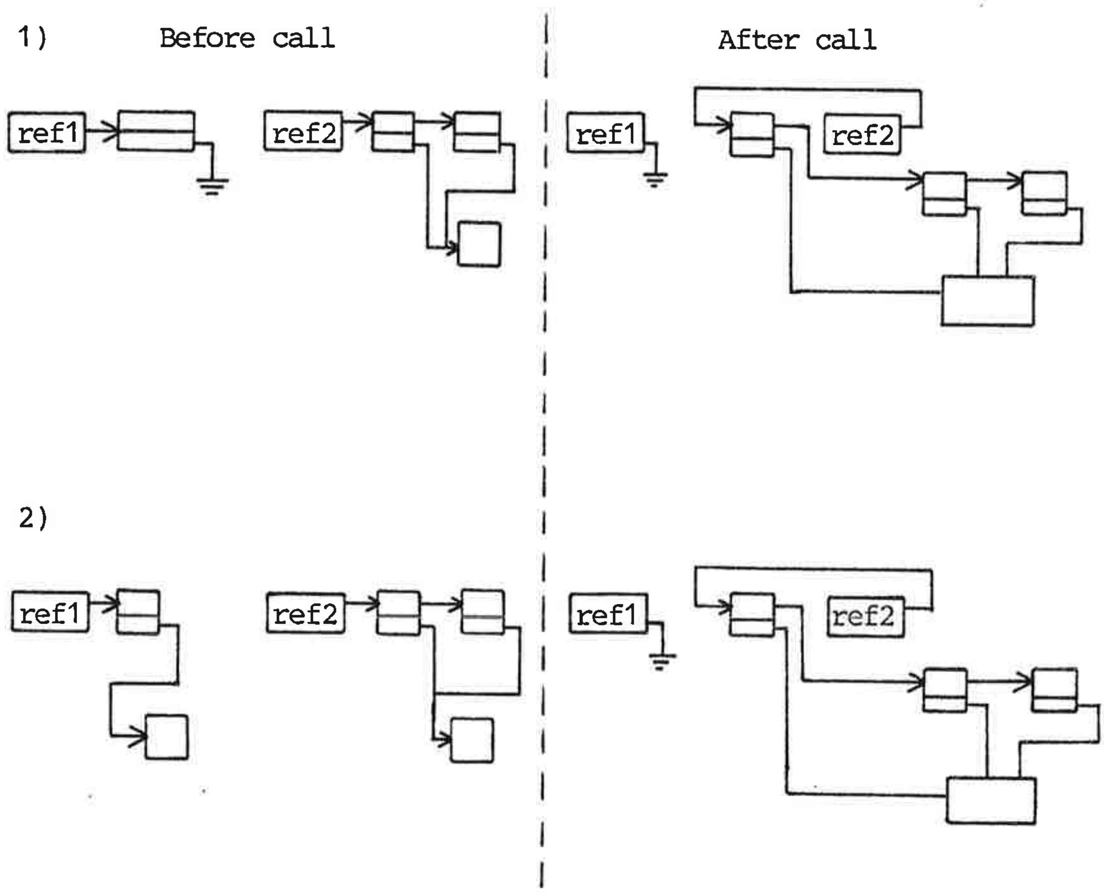
After call

ref1 - nil
ref2 - ref2:= old ref1

If the old ref1 has no associated message data, the message data associated the old ref2, if any, are assigned the old ref1, together with the pushing.

If the old ref1 has associated message data the old ref1 is just pushed.

Example



POP (ref1, ref2)

Before call

ref1 - nil
ref2 - well-defined stacktop element

<u>After call</u>

ref1 - ref1:= old ref2

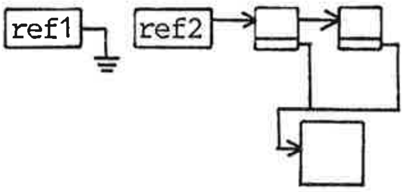
ref2 - new stacktopelement.
Note: ref2:= nil if the stack became empty during call.

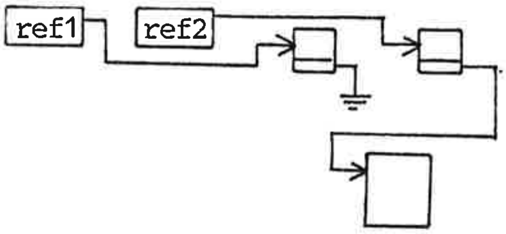
If the old stackop element has associated message data, and the new stackop element points to the same message data, the message data pointer in the popped element is set to nil.

This is not done if the new stackop element points either to another message data or points to nil.

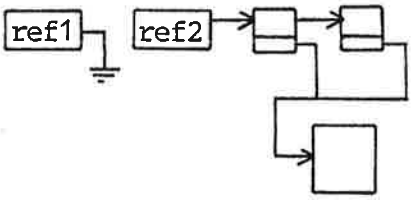
Examples

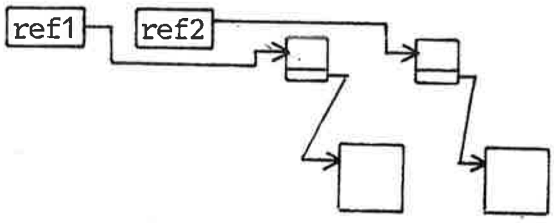
1) Before call / After call



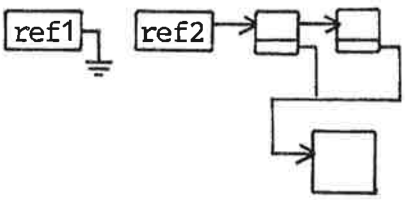


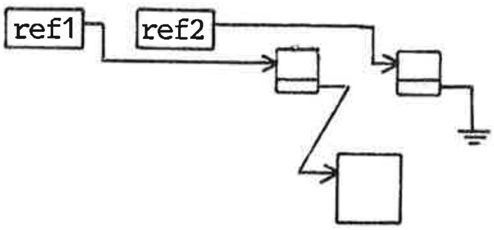
2)





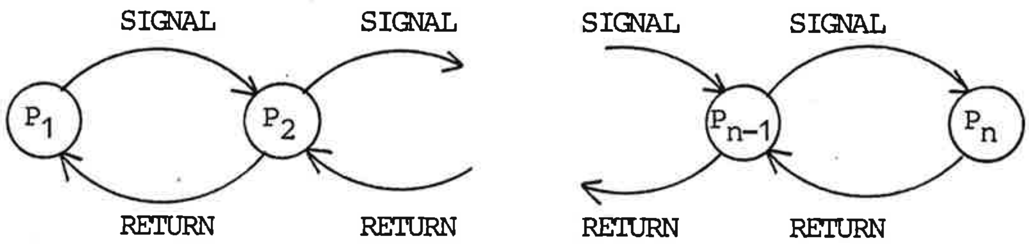
3)





The PUSH and POP procedures are especially suited to the following situations:

- to associate a new messageheader to received message data by the PUSH procedure in order to avoid copying of data, and to signal the message on to the next incarnation in the flow. The general answer mechanism will be to reestablish the original message header by a call of POP and return the message to the sender by calling RETURN.



- to pile together a number of messages and pass the whole batch to an incarnation by one call of SIGNAL.

Semaphore Pointers

A semaphore pointer variable is a variable of type:

↑semaphore

A semaphore pointer variable is a variable, whose value references a semaphore variable.

If the value is not a reference to a semaphore variable, the value is nil.

The only legal operations on semaphore pointer variables are:

nil, and := (assignment)

If p is a semaphore pointer variable, p denotes the semaphore pointed to by p.

signal, wait, return, and release.

There are four predefined communication routines, signal, return, wait, and release.

PROCEDURE signal (VAR r: reference; VAR s: semaphore);

The reference parameter must refer to a message (must not be nil), otherwise an exception occurs. The reference variable is nil after a call of signal.

If the semaphore is passive or open, the message referred to by r becomes the last element of the semaphore's sequence of messages. If the semaphore is locked, one of the incarnations waiting on the semaphore completes its wait call.

When several process incarnations are waiting, it is implementation dependent which one is resumed by a signal call. No process must, however, be waiting indefinitely if other incarnations continue to signal messages to the semaphore.

PROCEDURE return (VAR r: reference);

The parameter must refer to a message (must not be nil), otherwise an exception occurs.

The call:

```
return (r);
```

has the same effect as the call:

```
signal (r, r↑.answer↑);
```

The latter is, however, not a valid call because the answer semaphore is not explicitly available.

PROCEDURE release (VAR r: reference);

The parameter must refer to a message (must not be nil), otherwise an exception occurs.

The call:

    release (r);

has the same effect as the call:

    signal (r, r↑.owner↑);

The latter is, however, not a valid call because the owner samaphore is not explicitly avaiable.

PROCEDURE wait (VAR r: reference; VAR s: semaphore);

The reference parameter must be nil, otherwise an exception occurs. After a call of wait it refers to a message.

If the semaphore is open, the first message is removed from the semaphore's sequence of messages. If the semaphore is passive or locked, the incarnation waits and enters the set of incarnations waiting on the semaphore. It can be resumed by another incarnation calling signal or return.

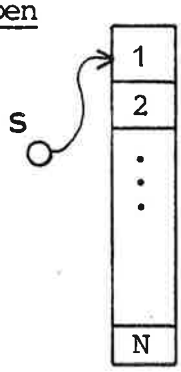
Open, Locked, Passive, and Sensesem

There are three predefined boolean functions to detect the state of a semaphore variable:

```
FUNCTION   open    (s: semaphore): boolean
FUNCTION   locked  (s: semaphore): boolean
FUNCTION   passive (s: semaphore): boolean
```

The three states may be depicted as:

open

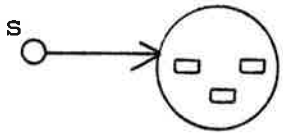


sequence of
messages

The sequence of messages is not empty.
The set of incarnations is empty.

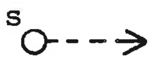
locked

The sequence of messages is empty.
The set of incarnations is not empty.



waiting
incarnations

passive



The sequence of messages is empty.
The set of incarnations is empty.

Sensesem

```
PROCEDURE sensesem (VAR r: reference;
                    VAR s: semaphore);
```

The body of sensesem is equivalent to:

```
IF open (s) THEN wait (r, s);
```

i.e. take a message from s if there is any, otherwise r remains nil.

Ref

Semaphore pointers may be assigned to denote a semaphore by means of the predefined routine ref:

```
FUNCTION ref (s: semaphore):  semaphore;
```

Semaphore pointers are initially set to nil by the system, this may be used to define a nilpointer which may be useful if semaphore pointers are used.

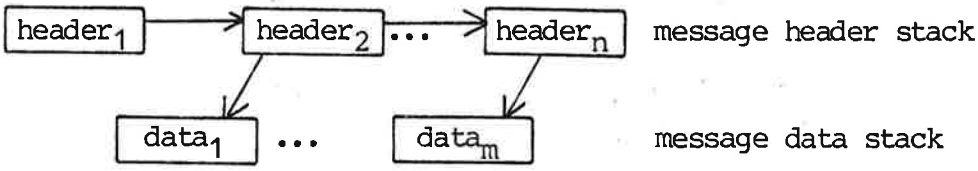
example:

```
var nil_pointer:!↑semaphore; (* nil_pointer cannot be
                                 changed since it is frozen*)
sem_arr: array (low .. high) of↑semaphore;

      .
      .

sem_arr (index):= ref (sem);

      .
      .

sem_arr (index):= nil_pointer;
```

Push, pop, and empty

A message may consist of a stack of headers and data areas. The stack of message headers is the message header stack, and the stack of the data areas, the message data stack.

message:



A header may or may not point to a data area ($m \leq n$). The top header of the message is $header_1$. The top data of a message is $data_1$.

The message is organized as a stack which is manipulated by the two predefined procedures push and pop.

PROCEDURE push (VAR r1, r2: reference);

The parameter r1 must refer to a message (must not be nil), and this message must have exactly one header, otherwise an exception occurs. The message accessible through r2 (possibly nil) is called the stack.

The header referred to by r1 becomes the new top header of the stack. After the call, r2 refers to the new stack.

If the new top message is a header message, the top data of r2 remains the same. After the call r1 is nil.

PROCEDRUE pop (VAR r1, r2: reference);

Reference variable r1 must be nil and r2 must refer to a message (must not be nil), otherwise an exception occurs.

The top header is removed from the message (accessed through r2) and after the call r2 refers to the remaining part, while r1 refers to the removed message.

It may be detected if a reference variable refers to a message with one header only by means of the predefined boolean function:

```
FUNCTION empty (r: reference): boolean
```

The body of empty may be:

```
        pop (local ref, r),
        empty:= nil (r)
```

where nil is another standard function:

<u>Nil</u>

```
FUNCTION nil (p: pointer_type): boolean
```

pointer_type may be any pointed type, for example reference which is defined like

```
TYPE reference = ↑ message;
```

<u>Alloc and Openpool</u>

With each variable of type pool an anonymous semaphore is associated. This is the owner semaphore of all messages in the pool. A message is allocated from the pool by the predefined procedure alloc.

```
PROCEDURE alloc (VAR r: reference, VAR p: pool 1; VAR s: semaphore);
```

The pool variable can be of any pool type.

The reference variable must be nil, otherwise an exception occurs. After the call it refers to a message. If the pool of message is not empty, one of the messages is removed. If the pool is empty the incarnation waits until a message is released to the pool by another process incarnation calling release. The answer semaphore of the removed message becomes s.

Variables of type pool (or variables with pool components) can only be declared in the declaration of a process and not in the declarations of a routine.

It may be detected if a pool is open (i.e. not empty) by means of

```
FUNCTION openpool (VAR p: pool 1): boolean;
```

the function result becomes true if the pool is not empty (cf. function open for semaphore).

5. PROCESS CONTROL 5.

Processes can be nested and a process declared within another process is a sub-process (of the surrounding process).

An arbitrary number of incarnations of sub-processes (children) can be created, they are all controlled by the parent. Incarnations are created and removed dynamically.

A process can have formal parameters. When an incarnation of the process is created a number of actual parameters is given. Incarnations communicate through common semaphore variables only. In this way a parent determines the communication paths of children. Note, however, that the controlling process incarnation need not participate in the communication.

Variables of the predefined type shadow are used to discern different incarnations of sub-processes. A shadow variable is the controlling process' link to an incarnation of a child. There is a number of predefined routines for exercising this control (start, stop, etc.).

5.1 The Predefined Routines for Process Control 5.1

Link

```
FUNCTION link (external_name: alfa;
               process name): integer;
```

There must not be a process linked to process name, process name must be the name of a process. The process identified by the external_name is linked to process name. The external identification of processes is implementation dependent.
Result 0 means success, other values are implementation dependent error codes.

```
FUNCTION create (incarnation_name: alfa;
        process name (actual parameters);
    VAR sh: shadow; storage: integer): integer;
```

The shadow variable must be nil and process name must be linked to a process. Result 0 means success, other values are implementation dependent error codes.

A new incarnation of the process linked to process name is created. The storage parameter specifies the amount of storage for holding the runtime stack. The store is initialized with the actual parameters and various administrative informations but the incarnation is stopped. The created incarnation is a child of the creating incarnation, the parent. After the call the shadow variable refers to the child.

Remove

```
PROCEDURE remove (VAR sh: shadow);
```

The shadow variable must refer to a process incarnation (child), otherwise an exception occurs.

Remove terminates execution of the child and deallocates all its resources. Execution of that incarnation cannot be resumed. Remove also removes all incarnations controlled by the child, their children ect.

After the call the shadow variable is nil.

Start, Stop, and Break

The following predefined procedures are used for controlling children between calls of create and remove.

```
PROCEDURE start (VAR sh: shadow; priority: integer);
```

Start initiates or resumes execution of a child which is stopped. The meaning of priority is implementation dependent.

PRODURE stop (VAR sh: shadow);

The shadow variable must refer to a process incarnation (child). The child is stopped.

PROCEDURE break (VAR sh: shadow; exception_code: integer);

The shadow variable must refer to a process incarnation (child). The call forces an exception upon the child. The meaning of the exception_code is implementation dependent.

Unlink

FUNCTION unlink (process name): integer;

At process must be linked to process name and no incarnations of the process may exist. After the call the link is deleted. Result 0 means success, other values are implementation dependent error codes.

6. UTILITY PROGRAMS 6.

6.1 Indent 6.1

Text formatting program

The program performs indention of source programs depending on the options specified in the call and on the keyswords (reserved words) of PASCAL/PASCAL80.

call:

```
                1
<outputfile>=       indent <input file>   <option>
                0                                 0
```

```
<option>::= lines     line numbers are added
            mark      the blockstructure is made clear by means
                      of ! between matching begin-end's
            list      the same as: lines mark
            noind     the output will be left justified
            myind     the output indention is the same as the
                      input indention
            lc        lists keywords in capital letters and
                      identifiers in small (lower case) letters
            uc        both key words and indentifiers are listed
                      in upper case letters
            help      produces a list of legal options
```

Storage requirements:

The core store required for indent is 16000 hW (size 16000).

Error messages:

??? illegal input-filename
input file must be specified

call: "indent help", for help

an error is detected in the program call, a new call "indent help" will produce a list of the valid options.

** warning, end(s) missing

an error in the begin-end structure has been detected.

** premature end of file

comment or string not terminated.

6.2 Cross Reference Program 6.2

Produces a cross reference listing of the identifiers and numbers and a use count of the PASCAL/PASCAL80 key words used in the input text.

The cross reference list is made with no regard to the block structure of the program. The list is sorted according to the ISO-alphabet, i.e. numbers before letters, but with no difference between upper and lower case letters.

The occurrence list for an identifier consists of a sequence of PASCAL/PASCAL80 line numbers. The occurrence kind is specified by means of the character following the line number:

* meaning the identifier or number is found in a declaration part.

= meaning the identifier is assigned to in the line specified.

: meaning the identifier or number occurred as a label.

blank all other uses

<<<<<<<<<<<<<<<in the list is a warning denoting that the name consists of more than 12 characters, which is the number of significant characters for PASCAL-identifiers.

Call:

<output file> = cross <input file> <option> $^{1}_{0}$

<option>::= bossline. <yes or no>

<yes or no>::= yes bossline's are added to the listing. (default).

no only PASCAL/PASCAL80 line numbers are generated.

Storage requirements:

The core store required for cross is at least 40000 hW (size 40000), but the requirement depends on the size of the input text.

Errormess:

??? illegal output-filename
left hand side of the call must be a name.

??? illegal input-filename
input file must be specified

??? yes or no expected
option 'bossline' must be 'bossline.yes' or 'bossline.no'

??? error in bracket structure, detected at line: xx
missing ")" ('s)

??? error in blockstructure, detected at line: xx
unmatched end

***** warning: hash table overflow at line: xx
the name table ran full at line xx, the cross referencing continues for the names met until line xx, new names in the following lines are ignored.

A. REFERENCES

[1] Staunstrup, J.:
PASCAL80 Report
RCSL No 52-AA964
A/S Regnecentralen af 1979

[2] Jensen, K. and Wirth, N.:
PASCAL User Manual and Report
Springer - Verlag
Berlin 1974

RETURN LETTER

Title: PASCAL80 User's Guide RCSL No.: 42-i1539

A/S Regnecentralen af 1979/RC Computer A/S maintains a continual effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback, your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

Do you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Name: Title:

Company:

Address:

Date:

Thank you

42-i 1288

. . . . Fold here . . . .

. . . . Do not tear - Fold here and staple . . . .

Affix
postage
here

REGNECENTRALEN
af 1979

Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark