RCSL No: 52-AA964

Edition: January 1980

Author: Jørgen Staunstrup

Title:

PASCAL80 REPORT

A/S REGNECENTRALEN
af 1979

Keywords:

High level language, Pascal, Concurrent programming, queue semaphores, LAMBDA.

Abstract:

This is a complete description of the programming language Pascal80. Pascal 80 extends the programming language Pascal with concurrent programming tools: processes, messages, and queue semaphores.

(80 printed pages)

Printed by A/S Regnecentralen af 1979, Copenhagen

42-i 1341

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS (continued) PAGE

TABLE OF CONTENTS (continued) PAGE

1. INTRODUCTION AND NOTATION 1.

1.1 Introduction 1.1

This report is a complete description of the programming language Pascal80. The description is intended to be short, yet precise and complete. Consequently, it is not a tutorial introduction.

Pascal80 is based on the two languages Pascal [1] and Platon [2] . Most language constructs are taken from Pascal except for the concurrent programming concepts which are similar to those found in Platon.

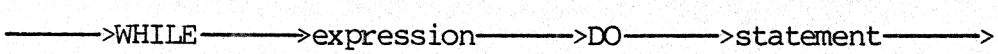
1.2 Notation 1.2

This manual is divided into a number of chapters and sections. The definitions given in a section also apply to all subsections; e.g. when the operator = is described in section 5.1 on simple types, this means that = can be applied to values of any of the simple types described in section 5.1.1 through 5.1.2.

1.2.1 Syntax Diagrams 1.2.1

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:



The syntax diagram defines the name (while statement) and syntax of a 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 distinguish between several occurrences of a construct, its name may be subscripted.

1.2.2 Separators

The separators are comments and non-printing symbols.

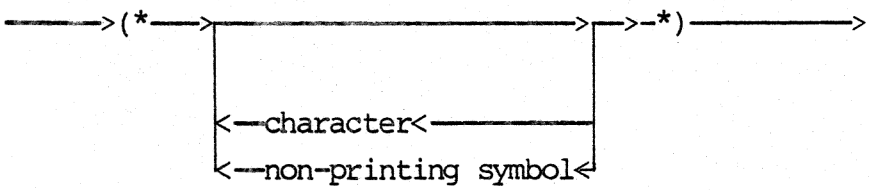
Separators can appear between any two consecutive language symbols.

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.

1.2.2.1 Comments

Comments are used to give explanatory text.

comment:



A comment does not affect the execution of the program.

1.2.2.2 Non-printing Symbols

The non-printing symbols are:

nl, sp, ff.

1.2.3 Semicolon

Semicolons (;) are used between declarations and between statements. Before the symbol END semicolon is allowed, but not mandatory. In all places where semicolon is allowed two or more semicolons are also allowed.

2. FUNDAMENTAL CONCEPTS 2.

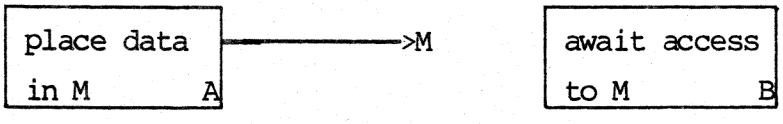
This chapter gives a brief explanation of a few concepts and the context in which they are used. The complete description of all Pascal80 concepts is given in chapters 3-11.

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.

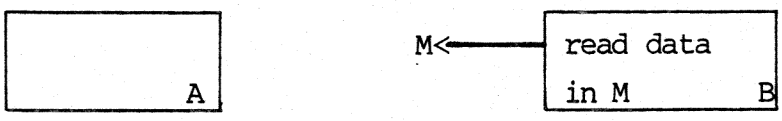
Actions are described by statements. The actions of one incarnation are executed one at a time in the order defined by the statements. The actions manipulate the data structure, which is described by a number of variables. A variable has a name and a type. The type describes the set of values the variable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

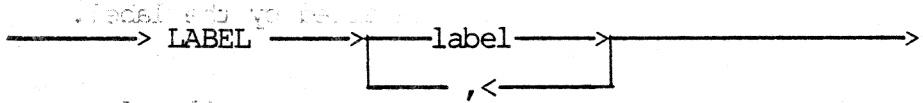
A process can have formal parameters. When an incarnation of the process is created a number of actual parameters is given. Incarnations communicate through common semaphore variables only. In this way a process determines the communication paths of sub-processes. Note, however, that the controlling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

3. LABELS

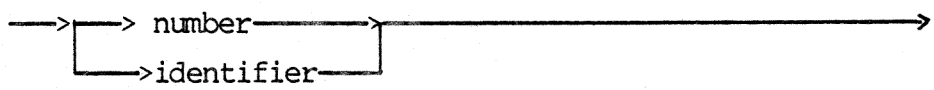
3.1 Label Declaration

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

label declaration:



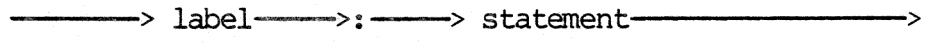
label:



A label is denoted by the identifier or the integer value of the number.

A label is defined by a labelled statement.

labelled statement:

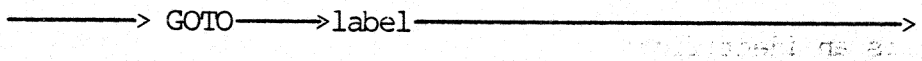


Labels must be defined and used in the scope (not block) where they are declared. A label may only be defined once in a scope.

3.2 Jumps

3.2

goto statement:



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.

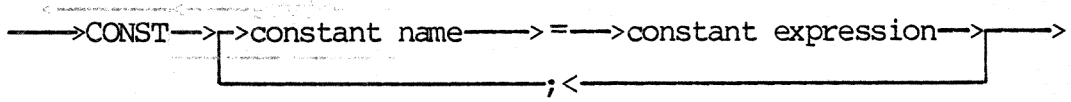
Jumps out of a channel or lock statement are not allowed.

4. CONSTANTS

4.

A constant declaration introduces one or more constants. A constant is an identifier denoting the value of a constant expression.

constant declaration:

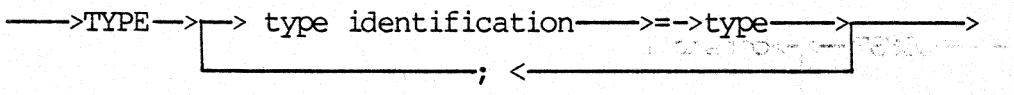


The type of the constant is determined by the constant expression. Constant declarations may not be recursive. A constant expression is an expression where all operands are symbolic values. Therefore, a constant expression has the same value throughout the whole execution of a block. Expressions are described in chapter 7.

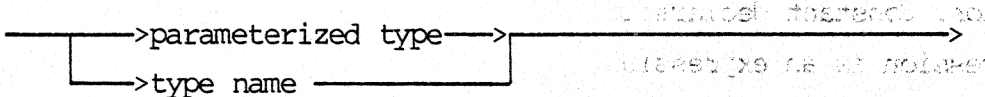
5. TYPES 5.

A type declaration introduces one or more types. A type is a set of values and a method of accessing these values.

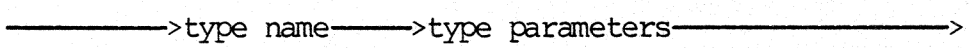
type declaration:



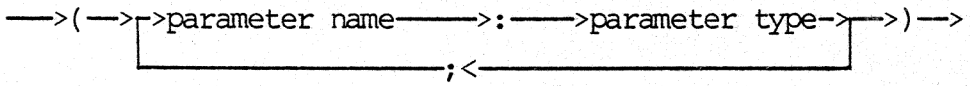
type identification:



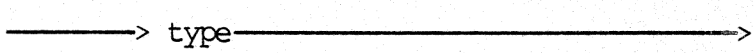
parameterized type:



type parameters:



parameter type:



The names of type parameters may be used in the type (on the right hand side of =) only.

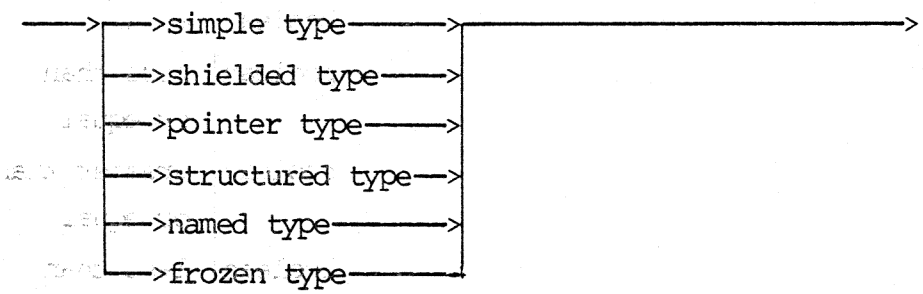
The parameter type must be an enumeration type or the name of an enumeration type.

The type string is a predefined parameterized type

```
TYPE string (n: integer) = ARRAY (1..n) OF char;
```

The type parameters are formal parameters. By binding different actual parameters to these, different named types can be obtained (section 5.6).

type:



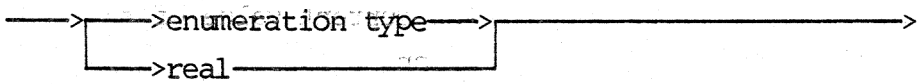
Type declarations may not be recursive, except in a type declaration:
TYPE name = t;
where t may contain the pointer type ↑name as an element or field type;

All types are described in this chapter; for each type is described: its values, how it is declared, how values of the type are accessed, which operators apply to the values, and which statements operate on the values.

5.1 Simple Types

A simple type is an enumeration type or real.

simple type:

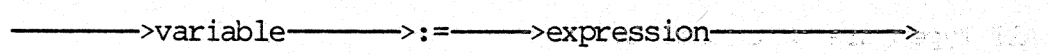


Operators:

operand	type of left operand	type of right operand	type of result	description
<>	any simple type T	T	boolean	not equal
=	any simple type T	T	boolean	equal
<=	any simple type T	T	boolean	less than or equal
>=	any simple type T	T	boolean	greater than or equal
<	any simple type T	T	boolean	less than
>	any simple type T	T	boolean	greater than

Assignment Statement

assignment statement:



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 (section 5.3)
- a structured type where all components are of a simple type or a pointer type (section 5.4)

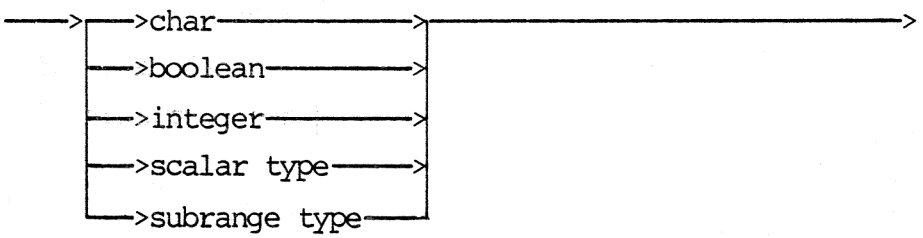
The assignment statement replaces the current value of the variable by the value of the expression.

5.1.1 Enumeration Types

5.1.1

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.

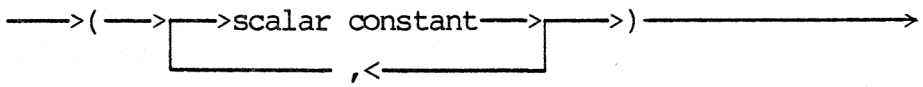
enumeration type:



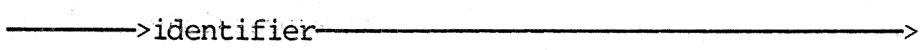
The three predefined types char, boolean, and integer are described below.

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:



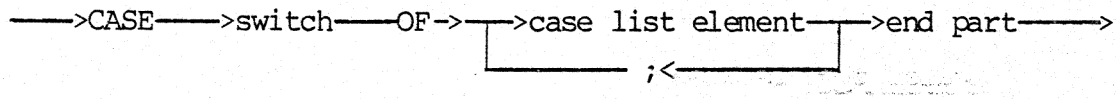
A scalar constant is an identifier appearing in the declaration of a scalar type T. The type of the scalar constant is T. Consider the following scalar type $(e_0, e_1, \ldots e_{n-1}, e_n, e_{n+1}, \ldots e_N)$, then e_{n-1} is the predecessor of e_n and e_{n+1} is the successor of e_n The ordinal value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

Predefined functions:

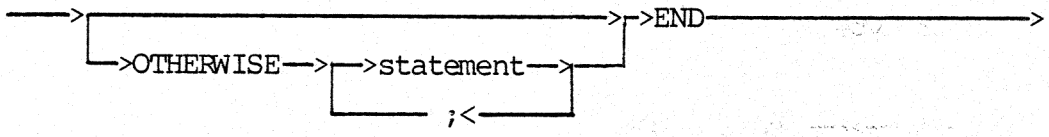
function name	type of parameter	type of result	description
succ	any enumeration type T	T	successor of the parameter
pred	any enumeration type T	T	predecessor of the parameter
ord	any enumeration type T	integer	ordinal value of the parameter

Case Statement

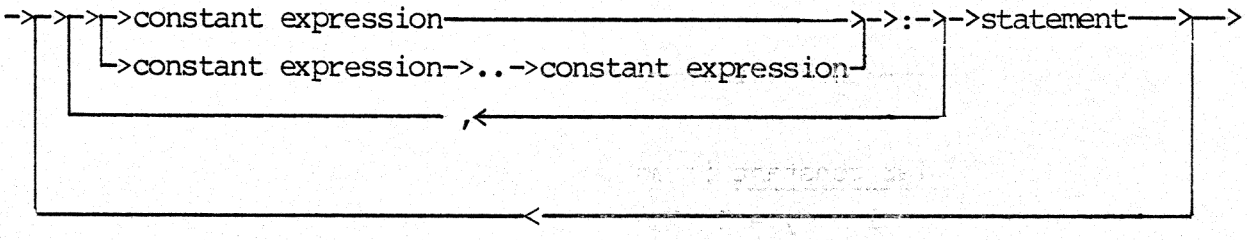
case statement:



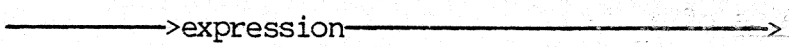
end part:



case list element:



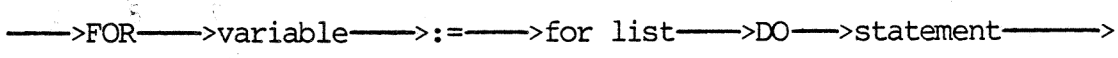
switch:



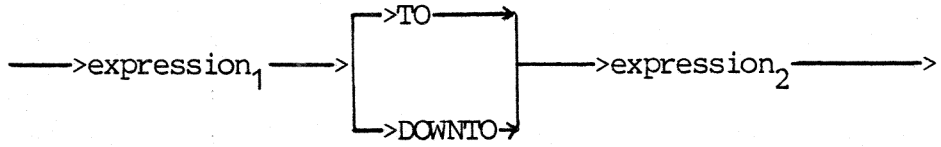
The values of the constant expressions in case list elements are called case labels. All case labels and the switch must be of the same enumeration type and all case labels must be distinct. The switch is evaluated and the statement labelled by the value of the switch is executed. If no such label is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

For Statement

for 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 variable can either be incremented (in steps of 1) from expression$_1$ TO expression$_2$, or decremented (in steps of 1) from expression$_1$ DOWNTO expression$_2$ .The two expressions are evaluated once, before the repetition. If expression$_1$ is greater than expression$_2$ and TO is specified, the statement is not executed.

Similarly, if expression$_1$ is less than expression$_2$ and DOWNTO is specified, the statement is not executed.

The value of the variable is unknown after the for statement.

5.1.1.1 The Type Char

5.1.1.1

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

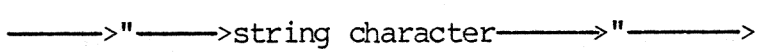
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
10	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
20	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
30	rs	us	sp	!	"	£	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	Æ	Ø	Å	↑	_	`	a	b	c
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.

Predefined function:

function name	type of parameter	type of result	description
chr	0..127	char	The character with the ordinal value of the parameter

char value:



The string characters are the characters: sp ..~, and nl, ff.

5.1.1.2 The Type Boolean

The type boolean is a predefined scalar type.

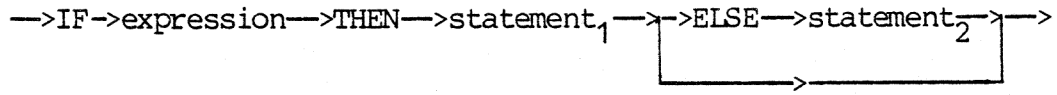
```
TYPE boolean = (false, true);
```

Operators:

operator	type of left operand	type of right operand	type of result	description
AND	boolean	boolean	boolean	logical conjunction
OR	boolean	boolean	boolean	logical disjunction
NOT	monadic	boolean	boolean	logical negation

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:

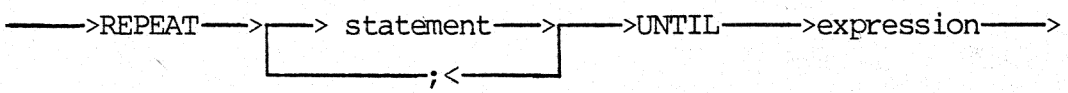
IF e_1 THEN IF e_2 THEN s_1 ELSE s_2

is equivalent to:

```
IF e1
   THEN BEGIN
        IF e2
           THEN s1
           ELSE s2
        END
```

Repeat Statement

repeat statement:

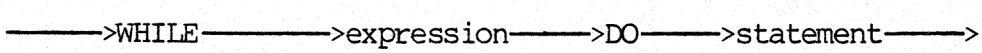


The result of the expression must be of type boolean.

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:



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.

5.1.1.3 The Type Integer

The type integer is a predefined enumeration type. Its values constitute a subrange of the integral numbers.

Operators:

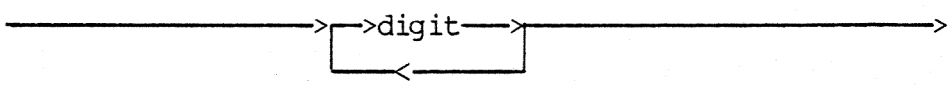
operator	type of left operand	type of right operand	type of result	description
+	integer	integer	integer	addition
+	monadic	integer	integer	monadic plus (redundant)
-	integer	integer	integer	subtraction
-	monadic	integer	integer	monadic minus
*	integer	integer	integer	multiplication
DIV	integer	integer	integer	integer division (truncated quotient)
MOD	integer	integer	integer	remainder of an integer division
/	integer	integer	real	division

Predefined functions:

function name	type of parameter	type of result	description
abs	integer	integer	absolute value
ord	integer	integer	redundant

Integer values are written symbolically as numbers.

number:



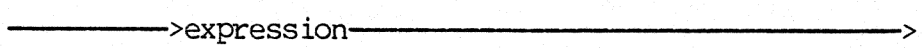
To indicate a radix different from 10,a number may be given an implementation dependent prefix.

5.1.1.4 Subrange Types 5.1.1.4

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

subrange type:

min bound,max bound:



The min and max bounds must be of the same enumeration type. All operands in the expressions min and max bound must either be symbolic values, value parameters of a frozen type, or formal type parameters. This ensures that the min and max bounds are the same throughout the execution of a block. The type of the min and max bounds is the base type of the subrange type. A subrange type is compatible with its base type and vice versa.

A static type is a type where the bounds of all subrange types are constant expressions. A dynamic type is a type where at least one bound in a subrange type contains a value parameter.

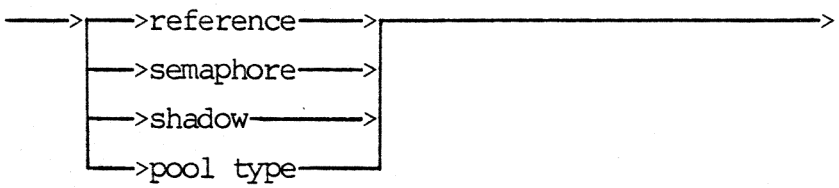
5.1.2 The Type Real 5.1.2

The predefined type real is a finite subset of the real numbers.

5.2 Shielded Types 5.2

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

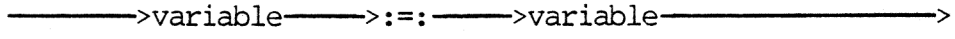
shielded type:



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 exchange statement is provided instead.

Exchange Statement

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.

5.2.1 Messages 5.2.1

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.

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

The message header consists of:

- pointers to the owner and answer semaphores of the message
- the size and kind of the message
- a pointer to the message data
- four user fields
- a number of invisible and implementation dependent fields

The type of the message header is:

```
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.

5.2.1.1 The Type Reference

5.2.1.1

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

Predefined function:

function name	type of parameter	type of result	description
nil	reference	boolean	true if the parameter is nil, false otherwise

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;
```

5.2.1.2 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 is non-empty. When the sequence is empty and the set of waiting incarnations is nonempty, the semaphore is locked. If both are empty, the semaphore is passive.

Predefined functions:

function name	type of parameter	type of result	description
open	semaphore	boolean	true if the semaphore is open, false otherwise
locked	semaphore	boolean	true if the semaphore is locked, false otherwise
passive	semaphore	boolean	true if the semaphore is passive, false otherwise.

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

5.2.1.3 Communication Routines

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 semaphore is not explicitly available.

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.

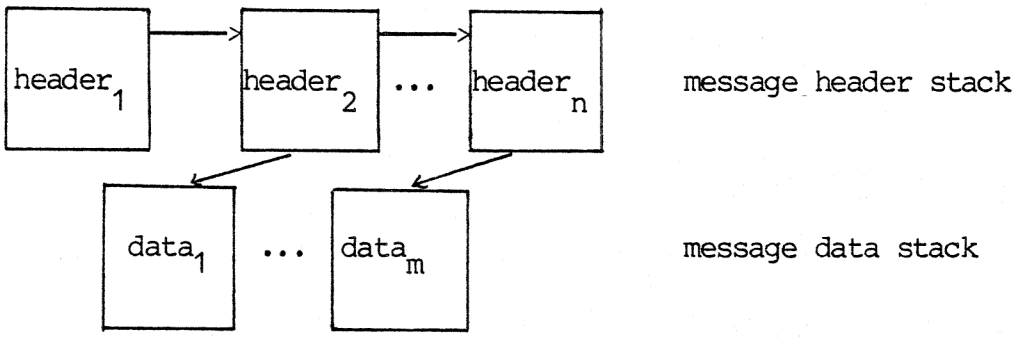
If the semaphore is open, the first message is removed from the semaphore's sequence of messages. If the semaphore is passive or locked, the incarnation waits and enters the set of incarnations waiting on the semaphore. It can be resumed by another incarnation calling signal or return.

5.2.1.4 Message Stack

5.2.1.4

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 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 a 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.

PROCEDURE 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.

Predefined function:

function name	type of parameter	type of result	description
empty	reference	boolean	true if the reference variable refers to a message with one header only, false otherwise

Let r be a reference variable which is not nil, then r↑.size, r↑.messagekind etc. denote fields of the top header. Similarly return and release use the answer and owner semaphores of the top header.

The top data of a message is manipulated by a lock statement.

Lock statement

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 and the type must not be parameterized. 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. The reference variable is nil within the statement. After the statement it refers to the same message.

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

Channel Statement

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 a parameter to wait, signal, return, release, alloc, pop, or push.

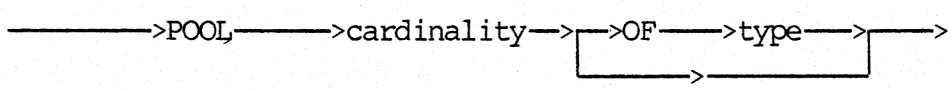
The channel statement controls the handling of peripherals in an implementation dependent way.

5.2.1.5 Pool Types

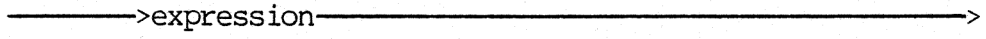
5.2.1.5

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.

PROCEDURE alloc (VAR r: reference; VAR p: pool type; VAR s: semaphore);

The pool variable can be of any pool type. (Note that this is a violation of the rules for routine parameters given in section 8.1).

The reference variable must be nil, otherwise an exception occurs. After the call it refers to a message. If the pool of messages 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.

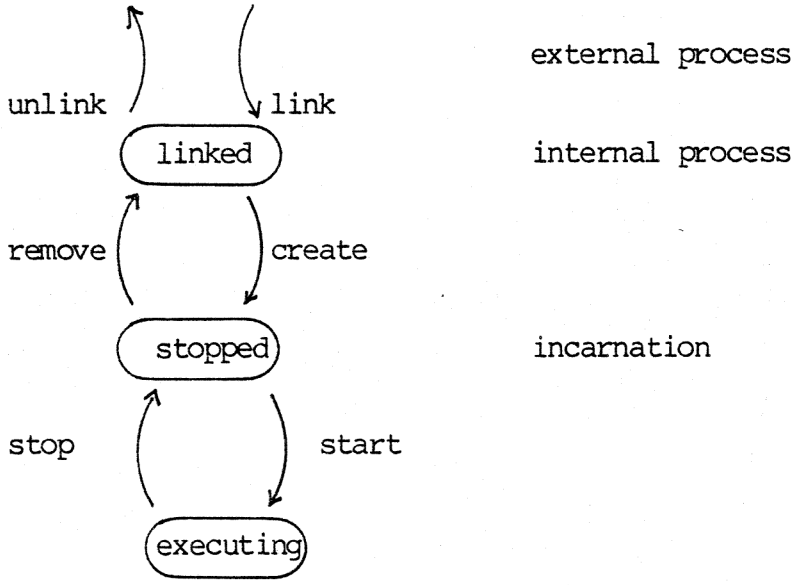
Variables of type pool (or variables with pool components) can only be declared in the declarations of a process and not in the declarations of a routine.

5.2.2 Process Control 5.2.2

A process is a description of some actions and a data structure. A process is realized as a program text which can be executed by a processor. A process can be linked to a process name. There are two predefined routines for linking and unlinking a process to a process name. When a process is declared, with an internal block, the process name (given in the heading) is initially linked to this process.

An incarnation of a process is the execution of the process actions on a private data structure. When an incarnation is stopped no actions are executed and the values in the data structure are not changed. Finally, an incarnation can be executing its actions as described by the program text.

The diagram below shows the state changes of a process and its incarnations. The predefined routines for doing this are described in section 5.2.2.2.



5.2.2.1 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. Below the predefined routines for controlling incarnations are described.

5.2.2.2 Control Routines


```
PROCEDURE link (external_name: string (n: integer);
                process name);                          +)
```

There must not be a process linked to the process name, otherwise an exception occurs. The process name must be the name of a process (see chapter 8: process heading). The process identified by the external name is linked to the process name. The external identification of processes is implementation dependent.

```
PROCEDURE unlink (process name);                        +)
```

A process must be linked to process name and no incarnations of the process may exist, otherwise an exception occurs. This link is deleted.

```
PROCEDURE create (process name (actual parameters);
VAR sh: shadow; storage, processor: integer);   +)
```

The shadow variable must be nil and the process name must be linked to a process. The binding of actual parameters to formal parameters is described in section 8.3.

+) Note that these procedure declarations violate the rules for routine parameters given in section 8.1.

A new incarnation of the process linked to process name is created. The storage and processor parameters specify the amount of storage for holding the runtime stack and the processor in which it is allocated. The store is initialized with the actual parameters and various administrative information 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.

```
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 etc.

After the call the shadow variable is nil.

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

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

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

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

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

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

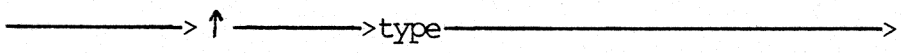
```
PROCEDURE break (VAR sh: shadow; exception: integer);
```

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

5.3 Pointer Types 5.3

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.

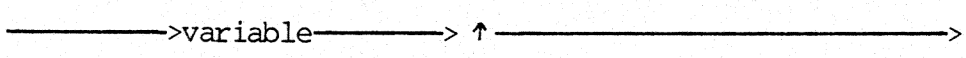
Assignments can be made to variables of pointer types.

Predefined function:

function name	type of parameter	type of result	description
nil	any pointer type	boolean	true if the parameter is nil false otherwise

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

pointed variable:



Let v be a variable of type ↑T, then the type of the pointed variable v ↑ is T. The selector of a pointed variable is ↑.

Predefined function:

function name	type of parameter	type of result
ref	semaphore	↑ semaphore

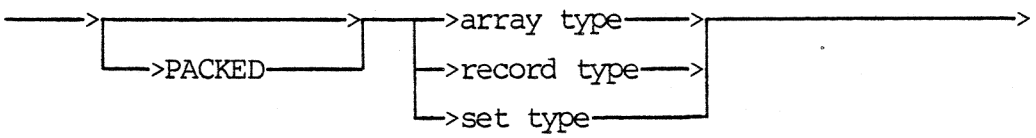
The parameter to the function ref must be a variable.

NOTE!! The predefined function ref can be applied to variables of type semaphore only. There is, therefore, no way of creating pointers to variables of other types.

5.4 Structured Types

A structured type is a composition of other types.

structured type:



A structured type has a number of component types. The component types of a simple, shielded, or pointer type are the type itself. The following operations are inherited by the structured type when they apply to all component types.

Operators:

operator	type of left operand	type of right operand	type of result	description
<>	any structured type T with simple components only	T	boolean	not all components are equal
=	any structured type T with simple components only	T	boolean	all components are equal

Assignments can be made to a variable of a structured type if assignment is allowed on all component types (section 5.1).

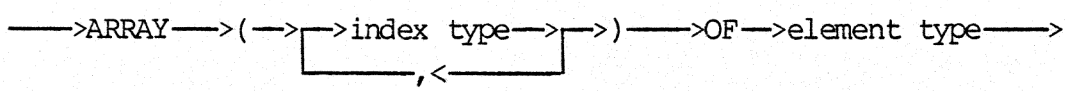
Array types are described in section 5.4.1.
Record types are described in section 5.4.2.
Set types are described in section 5.4.3.
Packed representation
is described in section 5.4.4.

5.4.1 Array Types

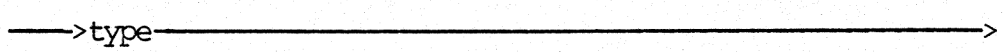
5.4.1

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.

The array type

ARRAY (t_1,t_2) OF t_3

is a shorthand for the type

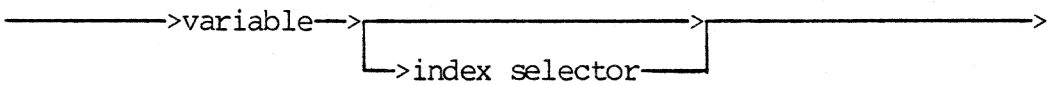
ARRAY (t_1) OF ARRAY (t_2) OF t_3.

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

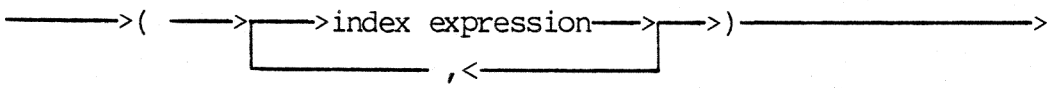
The name of an array variable denotes the whole array. An element is accessed by the array variable followed by an index enclosed in parentheses. An index consists of a number of index expressions. The number of index expressions must be less than or equal to the dimension of the array.

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.

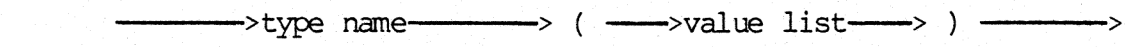
The following variables:

name (i_1,i_2) and name $(i_1)(i_2)$

access the same array element.

Array values are written symbolically as structured array values:

structured array value:



The type name specifies the type of the structured value. The type name must denote a one-dimensional array type. The number of elements in the value list must be the same as the number of elements in the index type and the type of the elements in the value list must be compatible with the element type of the array.

Value lists are described in section 7.2.

Note: The number of elements in a value list is static, so it is not possible to construct structured values of a dynamic type.

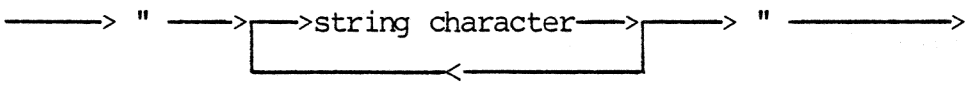
5.4.1.1 Strings

The type string is predefined.

```
TYPE string (n: integer) = ARRAY (1..n) OF CHAR;
```

A value of type string is a sequence of characters. Symbolically values of type string are written as character strings:

character string:



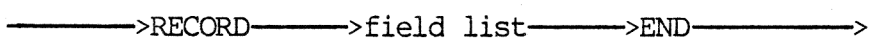
A string with n characters is of type string(n).

Char and string(1) are the same type.

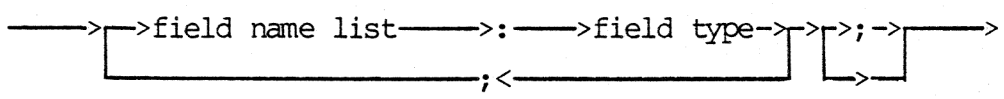
5.4.2 Record Types

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

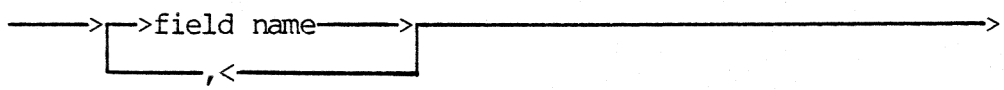
record type:



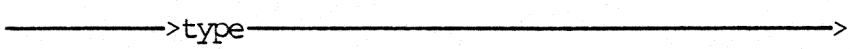
field list:



field name list:

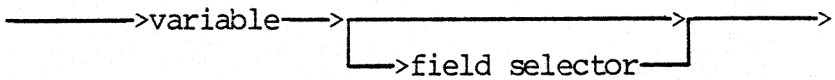


field type:

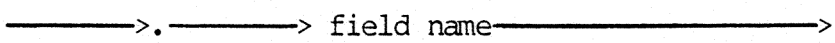


The component types are the component types of the field types. A record variable name denotes the whole record. A field is accessed by the variable which denotes the whole record and the field selector.

record variable:

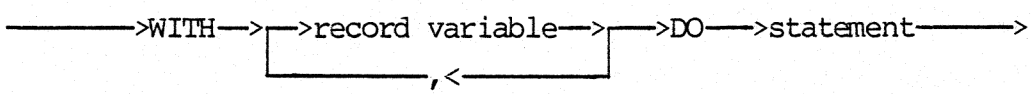


field selector:



With Statement

with statement:



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

The with statement

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

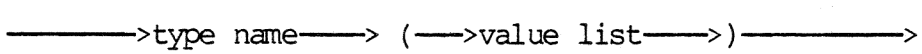
is a shorthand for the nested with statement:

```
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.

Record values are written as structured record values:

structured record value:



The type name specifies the type of the structured value.

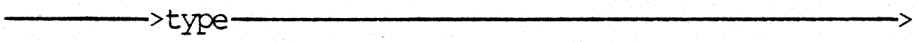
Each element in the value list specifies the value of the corresponding field. The value list must have an element for each field and the type of the element must be compatible with the type of the corresponding field.

Value lists are described in section 7.2.

5.4.3 Set Types 5.4.3

The values of a set type are the subsets of some enumeration type.
set type:

element type:



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

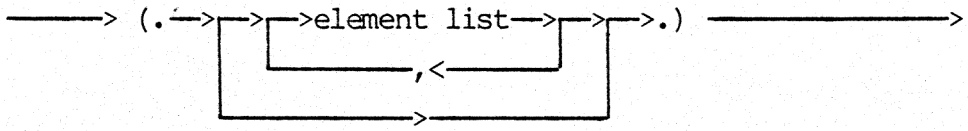
A set element is a value of the element type. The component type of a set is the element type.

Operators:

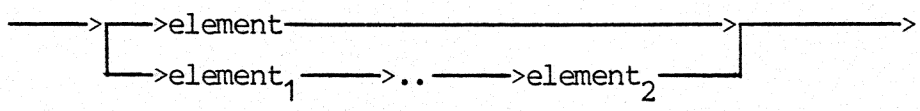
operator	type of left operand	type of right operand	type of result	description
+	any set type T	T	T	set union
*	any set type T	T	T	set intersection
-	any set type T	T	T	set difference
IN	any enumeration type T	SET OF T	boolean	set membership
<=	any set type T	T	boolean	set inclusion
>=	any set type T	T	boolean	set inclusion

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.

5.4.4 Packed Representation 5.4.4

The amount of storage needed to represent values of structured types can be reduced by prefixing the type definition with the symbol PACKED.

The packed representation may result in an increase in execution time.

The amount of space which is saved is implementation dependent, and in some implementations there may be no saving.

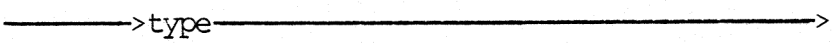
Note, the symbol PACKED is part of the type specification; a type and a corresponding packed type are therefore not compatible.

5.5 Frozen Types 5.5

The value of a variable of a frozen type cannot be changed.

frozen type:

base 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.

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

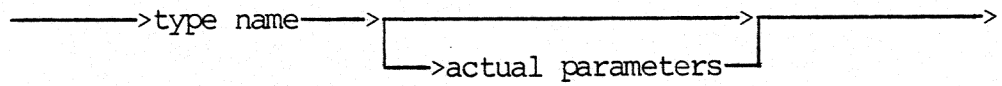
Parameters of a frozen type are given a value by a routine call or by creating an incarnation.

Variables of a frozen type are denoted as variables of the base type. Variables of frozen type can be initialized (see section 6.1).

5.6 Named Types

5.6

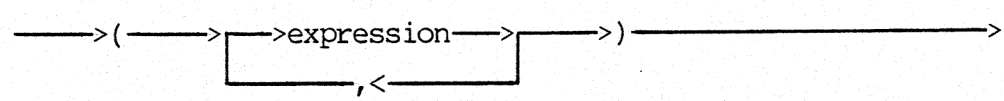
named type:



A named type is the type defined by the right hand side of a type declaration where type name occurs on the left hand side (type identification) (see the beginning of this chapter). If the type identification is a parameterized type the actual parameters of the named type are substituted for all occurrences of the corresponding formal parameters on the right hand side.

In this way a parameterized type may be used to generate a family of different types.

actual parameters:



Each actual parameter must be of the same type as the corresponding formal parameter. The number of formal and actual parameters must be the same.

Only values, frozen value parameters, and formal type parameters can be used as operands in the actual parameters.

A number of type names are predefined:

integer
char
boolean
real
semaphore
reference
shadow
string

5.7 Type Compatibility 5.7

Two named types are the same only if their type names are the same and if their actual parameters are pairwise the same. Two actual parameters are the same if their values are equal or if they are the same parameter name.

The types of two operands are the same if:

- the type names of the two operands are the same
- the two operands are declared in the same name list
- one of the operands is of type char and the other is of type string (1).

The type t_1 is <u>compatible</u> with the type t_2 if:

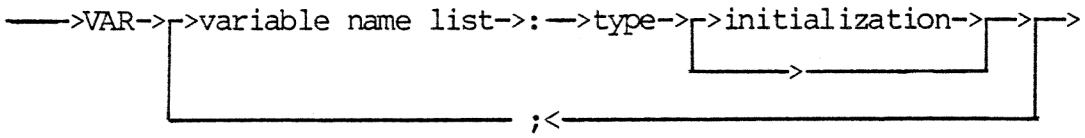
- t_1 and t_2 are the same named type
- t_1 is a subrange or t_2 or t_2 is a subrange of t_1
- t_1 is SET OF b_1 and t_2 is SET OF b_2 and
  b_1 is compatible with b_2
- t_1 is integer and t_2 is real
- t_1 is ! t_2
- t_1 is ↑t and t_2 is ↑t where t is a type name

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.

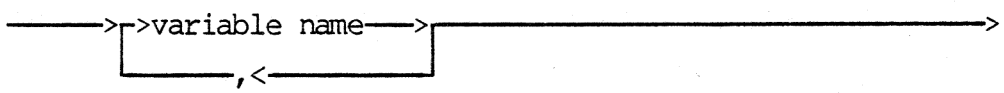
6. VARIABLE DECLARATION 6.

A variable is a named data structure that contains a value.

variable declaration:



variable name list:



The type specifies which values the variable can hold and how it is accessed.

The initial value of a variable is the value before execution of the compound statement of the block where the variable is declared. The initial value can be specified by an initialization.

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.

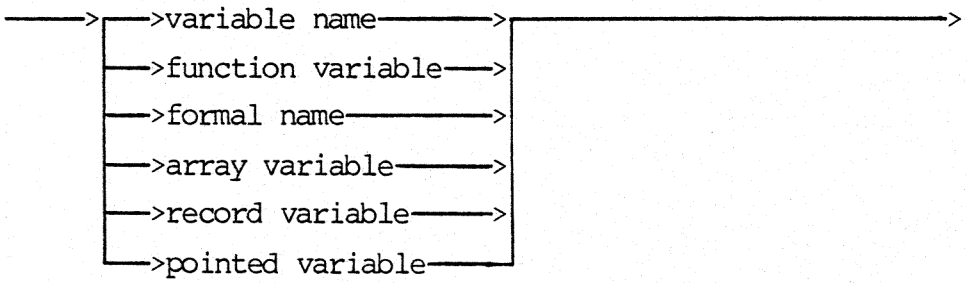
Variables of shielded and pointer types are implicitly given the following initial values:

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

6.1 Variables 6.1

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

variable:



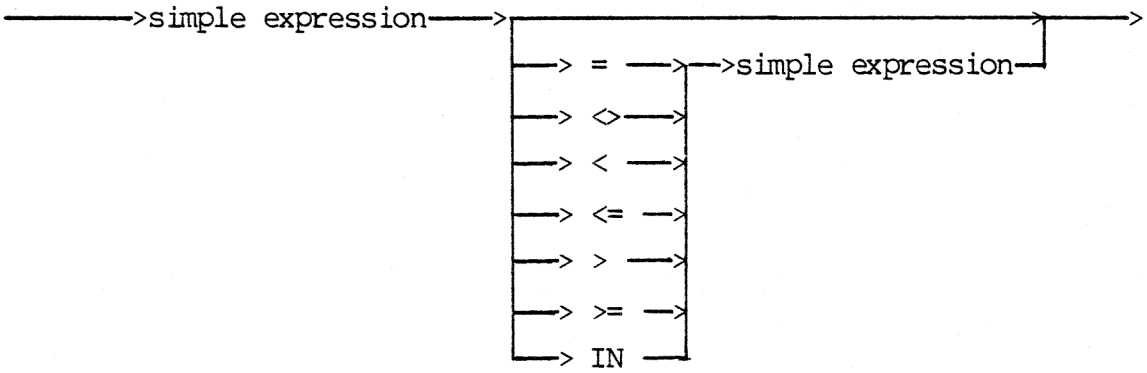
array variables are described in section 5.4.1
record variables are described in section 5.4.2
pointed variables are described in section 5.3
function variables are described in section 8.2.3.1

7. EXPRESSIONS 7.

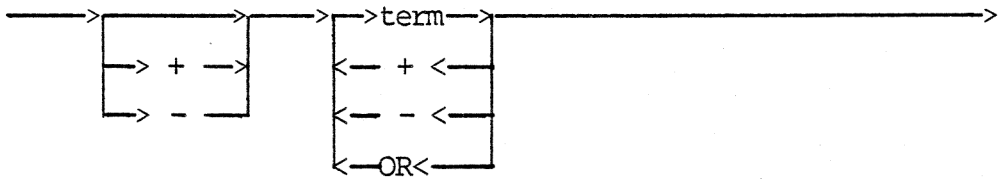
Expressions describe how values are computed. Expressions are evaluated from left to right using the following precedence rules:

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

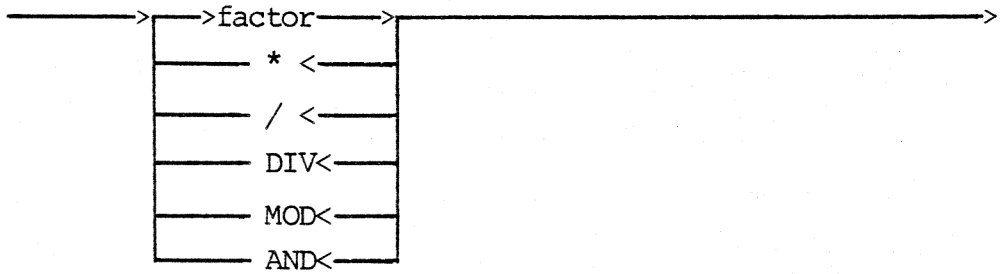
expression:



simple expression

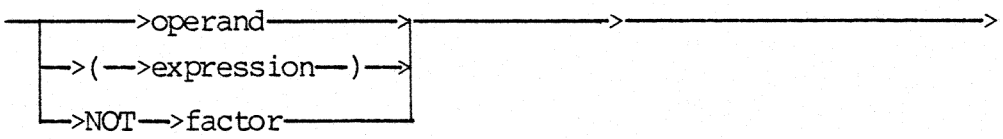


term:

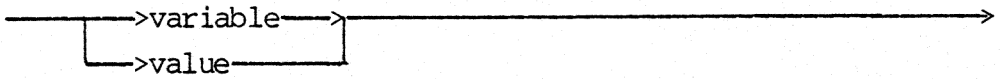


Note: All factors in an expression are evaluated.

factor:



operand:



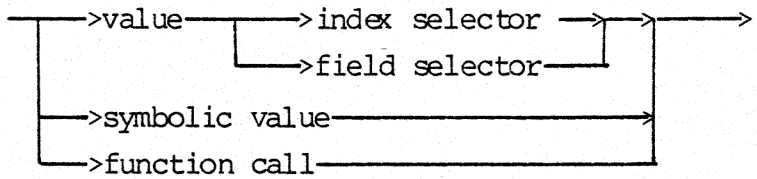
An expression where all operands are symbolic values is a constant expression.

7.1 Variable 7.1

Variables are described in chapter 6.

7.2 Value 7.2

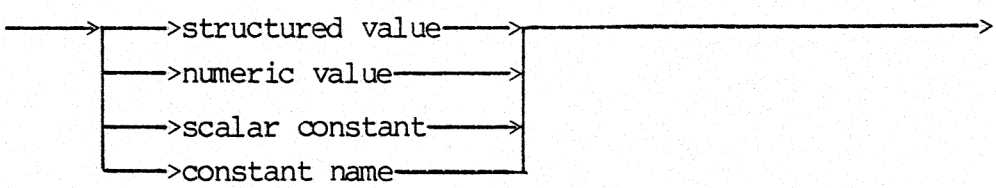
value:



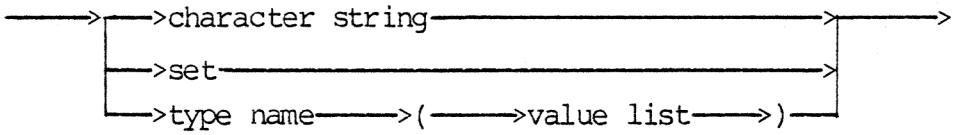
An index selector must only be applied to values of an array type. Similarly field selectors must only be applied to values of record type.

Symbolic values are given by their symbolic representation or as constants.

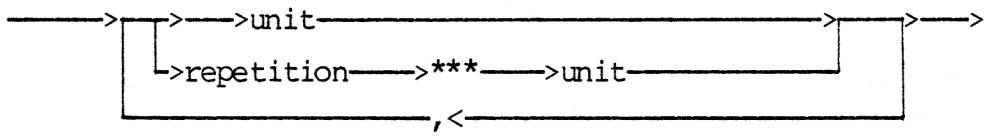
symbolic value:



structured value:



value list:



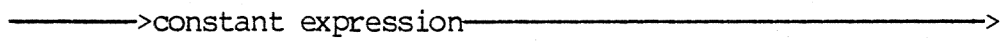
Units of a value list are given one at a time (separated by,) or by repeating a unit. The value of repetition specifies how many times the unit is repeated.

A structured value is built as follows:

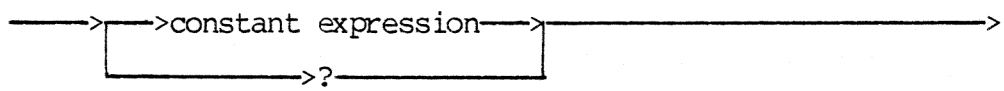
- type name denotes a record type:
  There must be a unit for each field and the first field gets the value of the first unit, the second field the value of the second unit etc. The repetition cannot be used.

- 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 field the value of the second unit etc.

repetition:



unit:

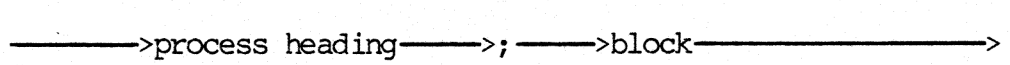


The unit "?" specifies no value, i.e. the component is skipped, its type is compatible with any type. This element is necessary to specify values of components which have no symbolic representation, e.g. values of shielded types. The no value element can only be used in value lists.

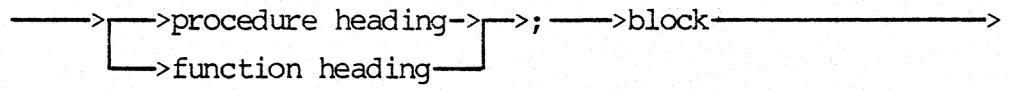
8. PROCESSES AND ROUTINES 8.

Process and routine declarations give a name to a block. This block can be executed by creating and starting a new incarnation of a process, or in the case of routines by a routine call. Routine is a generic term for procedures and functions.

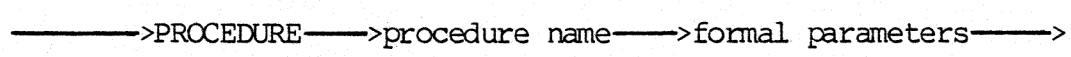
process declaration:



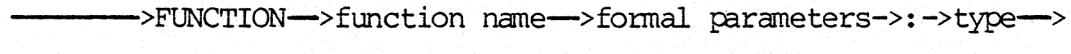
routine declaration:



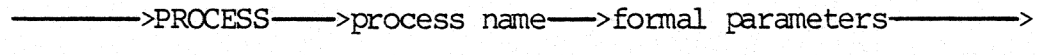
procedure heading:



function heading:



process heading:

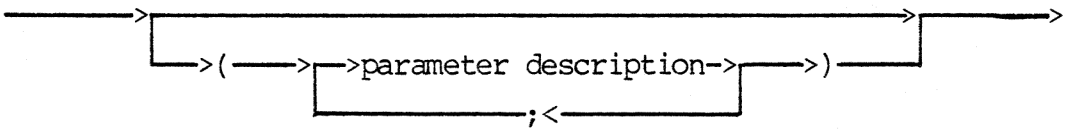


The type of a function cannot be a shielded type.

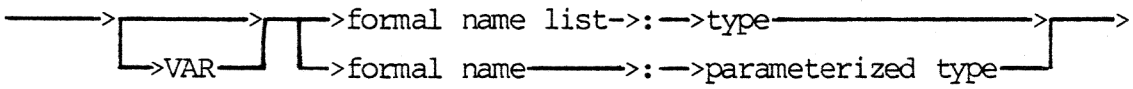
8.1 Formal Parameters 8.1

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

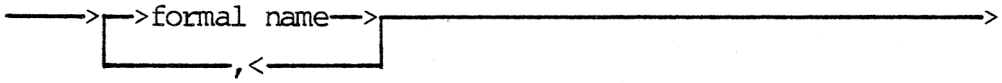
formal parameters:



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. Further differences between the two kinds of parameters are given in section 8.3.

8.1.1 Parameterized Type 8.1.1

A formal parameter can be of a parameterized type, but then the entire type identification must be given. The type of formal type parameters must be the same as in the declaration of the parameterized type.

The formal type parameters of the parameterized type are also formal parameters of the routine. They are of kind value and of a frozen type. These are called implicit parameters, to distinguish them from formal parameters which are not type parameters: explicit parameters.

8.1.2 Process Parameters 8.1.2

There are the following restrictions on the type of process parameters:

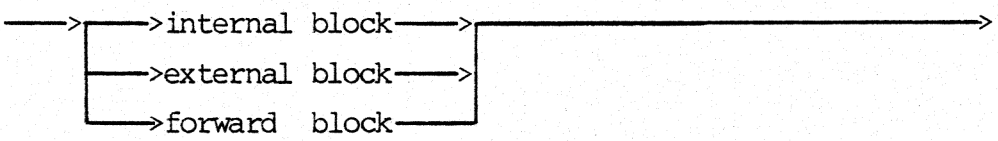
The component types of value parameters must be simple or pointer to semaphore, and the type must not be parameterized.

The component types of variable parameters must be semaphore or pointer to semaphore, and the type must not be parameterized.

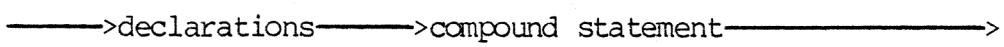
8.2 Block 8.2

A block consists of a number of declarations and a compound statement.

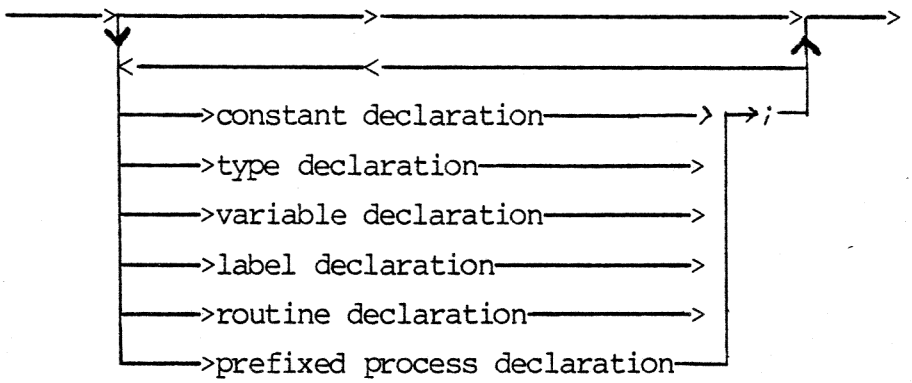
block:



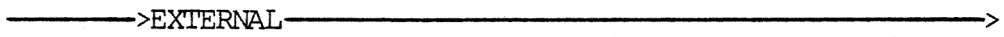
internal block:



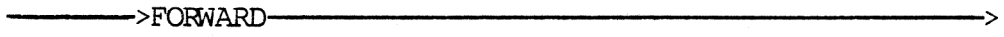
declarations:



external block:



forward block:



Constant declarations	are described in chapter 4.
Type declarations	are described in chapter 5.
Variable declarations	are described in chapter 6.
Label declarations	are described in chapter 3.
Process and routine declarations	are described in chapter 8.

An external block specifies a separately compiled block. The substitution of the external block of a routine by a separately compiled block is done implicitly by the compiler or runtime system. The substitution of the external block of a process is done explicitly by the predefined routines link and unlink (see section 5.7).

Routines with external blocks can only be declared in the declarations of a process and not in the declarations of a routine.

The scope rules (section 8.2.1) 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 same.

8.2.1 Scope Rules 8.2.1

A scope is one of the following:

- a field list excluding inner scopes,
- a declaration of a parameterized type 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, scalar constant, 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.

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 it not effective in the routine heading.

formal type parameter name: The declaration of a formal type parameter name is effective in the associated type specification only.

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

8.2.2 Process Blocks 8.2.2

The name of the process is not known in the process block.

8.2.3 Routine Blocks 8.2.3

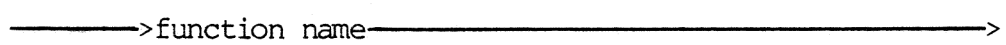
Within the block of a routine a recursive call of the routine can be made.

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

8.2.3.1 Functions 8.2.3.1

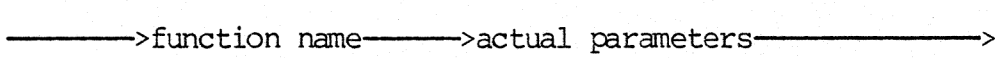
A function name may appear as a variable on the left hand side of an assignment or as a variable parameter in the block of the function. The type in the function heading is the function type, it specifies the range of the function. The value of a function is the dynamically last value assigned to the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

function variable:



A function call is a operand in an expression.

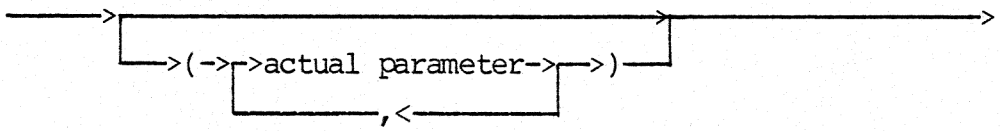
function call:



8.3 Actual Parameters 8.3

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

actual parameters:



actual parameter:

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.

If the formal parameter is of a parameterized type, only the type name of the formal and actual parameter has to be the same. The (initial) values of implicit formal parameters become the values of the corresponding parameters of the actual parameters.

8.4 Creation of a Process Incarnation 8.4

A process is a static description of a number of incarnations, which can be created and executed concurrently. Several incarnations of a process can be created and exist simultaneously.

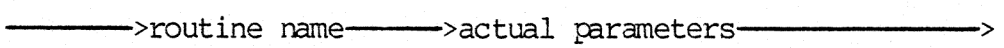
An incarnation is executed by executing the block. The execution is terminated either when the compound statement is completed or when the incarnation is removed.

Separate variables are associated with each incarnation of a process, they exist from the incarnation is created until it is terminated. When an incarnation is terminated:

- all incarnations controlled by it are terminated,
- all messages in semaphores or accessible through reference variables in the incarnation are released.
- all messages originating from pool variables in the incarnation continue to exist until they are released.

8.5 Routine Call 8.5

routine call:



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.

8.6 Exception Procedure 8.6

An exception procedure is called when an exception occurs. An exception procedure must be declared with the following heading:

```
PROCEDURE exception (excode: integer);
```

The formal parameter excode is an implementation dependent code for the exception.

It depends on the exception what happens when the block of the exception procedure is completed.

In the current implementation exception procedures cannot be declared in routine blocks.

8.7 Predefined Routines 8.7

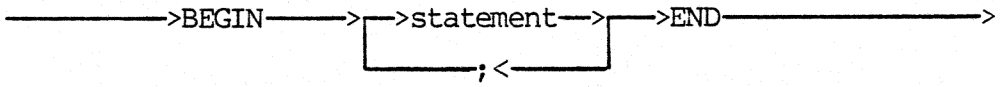
Below is given a list of predefined routines with references to the sections where they are defined.

Name	Section
abs	5.1.1.3
alloc	5.2.1.5
break	5.2.2.2
chr	5.1.1.1
create	5.2.2.2
empty	5.2.1.4
link	5.2.2.2
locked	5.2.1.2
nil	5.2.1.1/5.3
open	5.2.1.2
ord	5.1.1
passive	5.2.1.2
pop	5.2.1.4
pred	5.1.1
push	5.2.1.4
ref	5.3
release	5.2.1.3
remove	5.2.2.2
return	5.2.1.3
signal	5.2.1.3
start	2.2.2
stop	5.2.2.2
succ	5.1.1
wait	5.2.1.3
unlink	5.2.2.2

9. STATEMENTS 9.

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

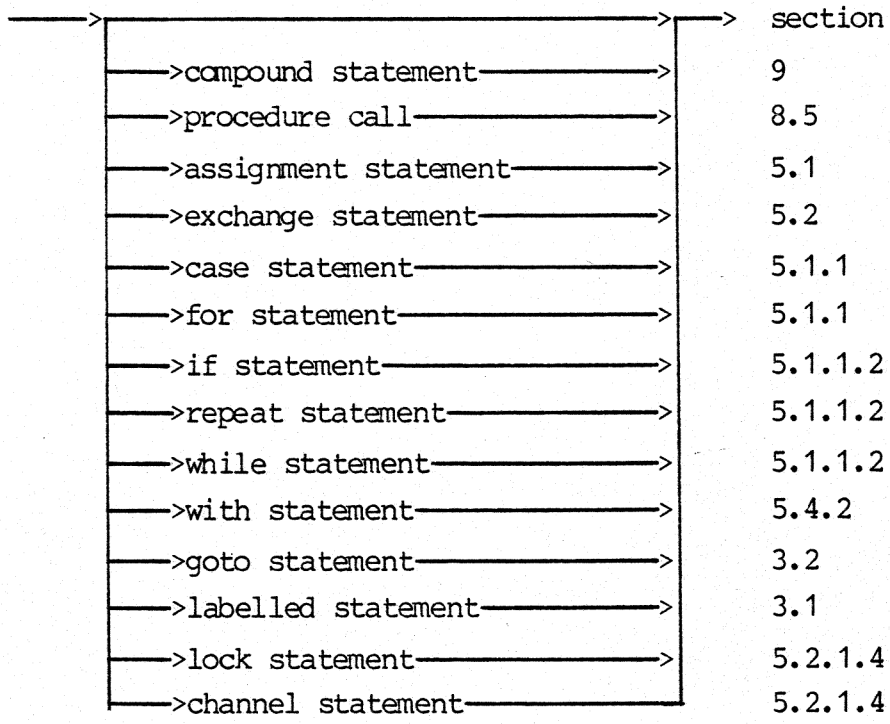
compound statement:



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:

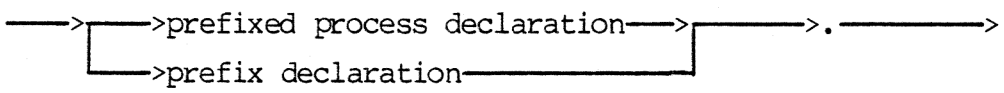
statement:



10. PROGRAM 10

A program is a process or prefix declaration.

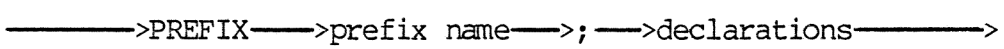
program:



10.1 Prefix 10.1

A prefix is a collection of declarations.

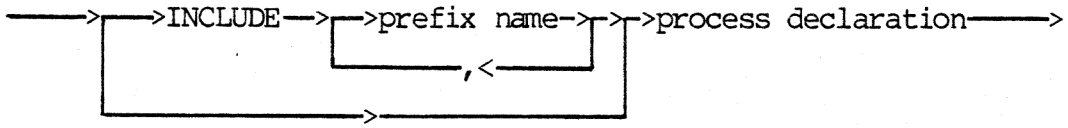
prefix declaration:



When a prefix is compiled it is included in a library, thereafter it can be referred to by its name. The administration of the library is implementation dependent.

The declarations described by one or more prefixes can be included in a process declaration.

prefixed process declaration:



The declarations in the prefix form a scope. The process declaration is an inner scope to the prefix scope.

11. VOCABULARY 11.

A Pascal80 program consists of a sequence of language symbols, identifiers, numeric values, character strings, and separators.

11.1 Language Symbols 11.1

The language symbols are punctuation marks and reserved identifiers. The reserved identifiers cannot be used otherwise.

PROCESS
CONST
TYPE
ARRAY
OF
RECORD
END
POOL
VAR
PROCEDURE
FUNCTION
LABEL
EXTERNAL
PREFIX
BEGIN
WITH
PACKED
SET
CASE
OTHERWISE
FOR
DOWNTO
DO
GOTO
IF
THEN
ELSE
TO

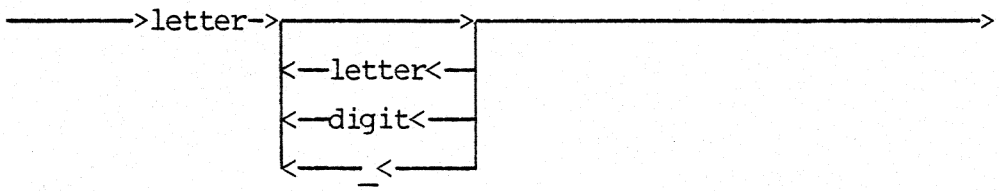
REPEAT
UNTIL
WHILE
IN
AS
CHANNEL
NOT
AND
OR
DIV
MOD
LOCK
INCLUDE
+
-
*
/
"
—
<
>
<>
<=
>=
(
)
(.
.)
↑
=
:=
:=:
.
,
;
:
..
***
(*

*)
!
?

11.2 Identifiers 11.2

All names of program quantities (variable name, type name etc.) are identifiers.

identifier:



letter is a, b, c,..., z

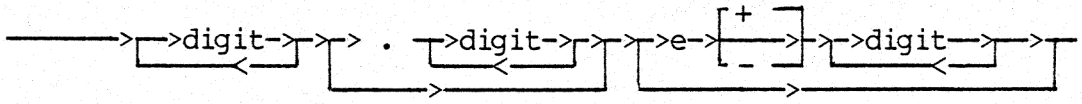
digit is 0, 1, 2, ..., 9

Identifiers can be arbitrarily long. An "_" (underline) is a significant character.

11.3 Numeric Value 11.3

A numeric value is a symbolic representation of a rational number.

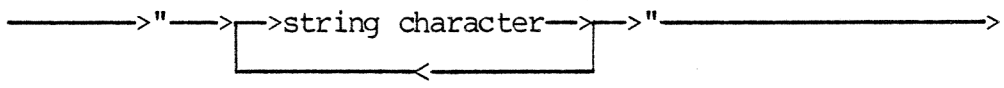
numeric value:



11.4 Character String

A character string is a sequence of characters:

character string:



string characters are the characters sp .. ~ , and nl, ff.

11.5 Separators 11.5

Separators are described in section 1.2.2.

A. REFERENCES A.

[1] PASCAL. User Manual and Report. Kathleen Jensen and Niklaus Wirth, Lecture Notes in Computer Science 18, Springer Verlag, Berlin 1974.

[2] Platon. Reference Manual, Sven Meiborg Sørensen and Jørgen Staunstrup, RECAU, University of Aarhus, Ny Munkegade, DK-8000 Aarhus C, Aarhus 1975.

B. INDEX B.

A PAGE

B

C

RETURN LETTER

Title: PASCAL80 REPORT RCSL No.: 52-AA964

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