TITLE: CR80 AMOS PASCAL COMPILER : REFERENCE MANUAL

DOCUMENT NO: CSS/450/RFM/0004

PREPARED BY: PER HØJMARK

APPROVED BY: JØRGEN HØG

AUTHORIZED BY: JØRGEN HØG

DISTRIBUTION:

ISSUE:	1							
DATE:	800619							

400-571-2

CR80 PASCAL
REFERENCE MANUAL

sign/date	page
PHØ/800619	i
repl	project

PAGE RECORD AND ISSUE LOG.

PAGE	ISSUE							
	1	2	3	4	5	6	7	8
01								
02								
03								
04								
05								
06								
07								
08								
09								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								
25								
26								
27								
28								
29								
30								
31								
32								
33								

PAGE	ISSUE							
	1	2	3	4	5	6	7	8
34								
35								
36								
37								
38								
39								
40								
41								
42								
43								
44								
45								
46								
47								
48								
49								
50								
51								
52								
53								
54								
55								
56								
57								
58								
59								
60								
61								
62								
63								
64								
65								
66								

PAGE	ISSUE							
	1	2	3	4	5	6	7	8
67								
68								
69								
70								
71								
72								
73								
74								
75								
76								
77								
78								
79								
80								
81								
82								
83								
84								
85								
86								
87								
88								
89								
90								
91								
92								
93								
94								
95								
96								
97								
98								
99								
100								

ISSUE	DATE	PREPARED BY	APPROVED BY	AUTHORIZED BY
1	800619			

PAGE RECORD AND ISSUE LOG.

PAGE	ISSUE 1	2	3	4	5	6	7	8
01								
02								
03								
04								
05								
06								
07								
08								
09								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								
25								
26								
27								
28								
29								
30								
31								
32								
33								

PAGE	ISSUE 1	2	3	4	5	6	7	8
34								
35								
36								
37								
38								
39								
40								
41								
42								
43								
44								
45								
46								
47								
48								
49								
50								
51								
52								
53								
54								
55								
56								
57								
58								
59								
60								
61								
62								
63								
64								
65								
66								

PAGE	ISSUE 1	2	3	4	5	6	7	8
67								
68								
69								
70								
71								
72								
73								
74								
75								
76								
77								
78								
79								
80								
81								
82								
83								
84								
85								
86								
87								
88								
89								
90								
91								
92								
93								
94								
95								
96								
97								
98								
99								
100								

ISSUE	DATE	PREPARED BY	APPROVED BY	AUTHORIZED BY
1	800619			

TABLE OF CONTENTS

1. Introduction

The main purposes of this document are:

- to define the CR80 PASCAL language
- to describe the system procedures and functions available to the programmer in a CR80 PASCAL program
- and to explain how the CR80 PASCAL compiler is operated

The document is not intended to be a tutorial on PASCAL.

2. Applicable documents

1. Jensen, Kathleen & Niklaus Wirth:
PASCAL User Manual and Report.
Second Edition.
Springer-Verlag. 1978.

2. CR80 AMOS, KERNEL
PRODUCT SPECIFICATION
CSS/302/PSP/0008

3. CR80 AMOS, I/O SYSTEM
PRODUCT SPECIFICATION
CSS/006/PSP/0006

4. CR FILE SYSTEM PSP
CSS/910/EWP/0001

5. CR80 AMOS, FILE NAME UTILITIES
PRODUCT SPECIFICATION
CSS/317/PSP/0014

6. CR80 AMOS, COMMAND INTERPRETER
USER´S MANUAL
CSS/381/USM/0037

7. CR80 Minicomputer Handbook
CSD/HDBK/0001

8. CR80 AMOS, ASSEMBLER
USER´S MANUAL
CSS/401/USM/0042

3. THE CR80 PASCAL LANGUAGE

3.0 Introduction to CR80 PASCAL

The following text is intended for readers with some experience in high level languages. For readers not acquainted with other programming languages it will be hot stuff. This introductory section tries to present an overview of CR80 PASCAL so that the reader can view the forest before examining individual trees.

A CR80 PASCAL program consists of two essential parts, a description of the actions to be performed, and a description of the data which are manipulated by these actions. Actions are described by statements, and data are described by declarations and definitions.

The data are represented by values of variables. Every variable occurring in a statement must be introduced by a variable declaration which associates an identifier and a data type with that variable. The data type essentially defines the set of values which may be assumed by that variable. A data type may in CR80 PASCAL be either directly described in the variable declaration, or it may be referenced by a type identifier, in which case this identifier must be described by an explicit type definition, or be one of the standard type identifiers BOOLEAN, INTEGER, CHAR or LONG_INTEGER.

Enumerated types are defined by indication of an ordered set of values, i.e. by introduction of identifiers standing for each value of the type.

A type may also be defined as a subrange of one of the types INTEGER, CHAR or BOOLEAN.

Structured types are defined by describing the types of their components and by indicating a structuring method. The various structuring methods differ in the selection mechanism serving to select the components of a variable of the structured type. In CR80 PASCAL there are three basic structuring methods available: array structure, record structure, and set structure.

In an array structure all components are of the same type. A component is selected by an array selector, or index, whose type is indicated in the array type definition. Given a value of the index type, an array selector yields a value of the component type. Every array variable can therefore be regarded as a mapping of the index type onto the component type.

In a record structure the components (called fields) are not necessarily of the same type. Each component has attached to it an identifier (declared in the record type definition) which is used when the component is selected.

A record type may be specified as consisting of several variants. This implies that different variables, although said to be of the same type, may assume structures which differ in a certain manner. The difference may consist of a different number and different types of components. The variant which is assumed by the current value of a record variable is indicated by a component field which

is common to all variants and is called the tag field.

A set structure defines the set of values which is the powerset of its base type, i.e. the set of all subsets of values of the base type.

Variables declared in explicit declarations are called static. The declaration associates an identifier with the variable which is used to refer to the variable. In contrast, variables may be generated by an executable statement. Such a dynamic generation yields a pointer, which subsequently serves to refer to the variable. This pointer may be assigned to other variables, namely variables of type pointer. Every pointer variable may assume values pointing to variables of the same type T only, and it is said to be bound to this type T. It may, however, also assume the value NIL, which points to no variable.

The most fundamental statement is the assignment statement. It specifies that a newly computed value be assigned to a variable, or a component of a variable. The value is obtained by evaluating an expression. Expressions consist of variables, constants, sets, operators, and functions operating on the denoted quantities and producing new values. CR80 PASCAL defines a fixed set of operators, each of which can be regarded as describing a mapping from the operand types into the result type. The set of operators is subdivided into groups of arithmetic operators (addition, subtraction, sign inversion, multiplication, division, and computing the remainder), boolean operators (negation, union, and

set difference), and relational operators (equality, inequality, ordering, set membership, and set inclusion).

The procedure statement causes the execution of the designated procedure (see below). Assignment and procedure statements are the components or building blocks of structured statements, which specify sequential, selective, or repeated execution of their components. Sequential execution of statements is specified by the compound statement, conditional or selective execution by the if statement and the case statement, and repeated execution by the repeat statement, the while statement, and the for statement. The if statement serves to make the execution of a statement dependent on the value of a boolean expression, and the case statement allows for the selection among many statements according to the value of a selector. The for statement is used when the number of iterations is known beforehand, and the repeat and while statements are used otherwise.

A statement can be given a name, and be referenced through that name. The statement is then called a procedure, and its declaration a procedure declaration. Such a declaration may additionally contain a set of variable declarations and type definitions. The variables and types thus introduced can be referenced only within the procedure itself, and are therefore called local to the procedure. Their identifiers have significance only within the program text which constitutes the procedure declaration and which is called the scope of these identifiers. Entities which are declared in the main program, i.e. not local to some procedure, are called global. A procedure has a fixed number

of parameters (if any), each of which is denoted within the procedure by an identifier called the formal parameter. Upon an activation of the procedure statement, an actual quantity has to be indicated for each formal parameter. This quantity is called the actual parameter.

A function is declared analogously to a procedure. The only difference lies in the fact that a function yields a result the type of which must be specified in the function declaration. Functions may therefore be used as constituents of expressions.

3.1 Notation, terminology, and vocabulary

Syntactic constructs are denoted by English words enclosed between angular brackets < and >. Zero or more repetitions of a construct is indicated by enclosing the construct within metabrackets { and } . The brackets [and] are part of the CR80 PASCAL language, but are also used as metasymbols. When used as metasymbols, they will be underlined ([,]). [X] means 0 or 1 instance of X. A bar (|) is used to indicate alternatives.

A "shorthand" will be used to avoid repetition that is more distracting than illuminating: constructs of the form:

<X identifier> ::= <identifier>

will not be shown. All particular identifiers are instances of identifier. Also, in the verbal description we will write q for the non-terminal symbol <q>.

The basic vocabulary of CR80 PASCAL consists of letters, digits, and special symbols.

```
<letter>  ::=  A|B|C|D|E|F|G|H|I|J|
               K|L|M|N|O|P|Q|R|S|T|
               U|V|W|X|Y|Z|_

<digit>   ::=  0|1|2|3|4|5|6|7|8|9|
```

Note: Underscore is a letter.

```
<special symbol> ::=  +|-|*|/|=|<|
                      >|[|(.|]|.)|
                      .|,|:|;|∂|(|
                      )|<>|<=|>=|:=|
                      ..| <word symbol>

<word symbol>    ::=  AND|ARRAY|BEGIN|CASE|
                      CONST|DIV|DO|DOWNTO|
                      ELSE|END|FOR|FORWARD|
                      FUNCTION|IF|IN|MOD|
                      NOT|OF|OR|PROCEDURE|
                      PROGRAM|RECORD|REPEAT|
                      SET|THEN|TO|TYPE|UNIV|
                      UNTIL|VAR|WHILE|WITH|
```

Special symbols have fixed meanings (except within character strings and comments). Thus, word symbols cannot be used as identifiers.

The construct

"<any sequence of characters not containing a double quote>"

is a comment if it does not occur within a character string. The substitution of a space for a comment will not alter the meaning of a CR80 PASCAL program.

Lexical tokens used to construct CR80 PASCAL programs can be classified into special symbols, identifiers, unsigned numbers and character strings. Comments, spaces,

and the NL- and FF-character are token separators. Zero or more token separators may occur between any two consecutive tokens, or before the first token of a program text. There shall be at least one separator between any pair of consecutive tokens made up of identifiers, word symbols or unsigned numbers. No separator may occur within tokens.

3.2 Identifiers, Numbers and Character Strings

Identifiers denote constants, types, variables, formal parameters, procedures, functions, programs, and fields and tag fields in records.

<identifier> ::= <letter> { <letter or digit> } $_{0}^{139}$

<letter or digit> ::= <letter> | <digit>

All characters of an identifier are significant.
Examples of identifiers:

X _A_FUNNY_ONE_ A38

Numbers are the constants of the standard data types INTEGER and LONG_INTEGER.

<digit sequence> ::= <digit> {<digit>}
<hexa digit> ::= A|B|C|D|E|F| <digit>
<hexa digit sequence> ::= <hexa digit> {<hexa digit>}
<unsigned integer> ::= <digit sequence> |
#<hexa digit sequence>
<unsigned long_integer> ::= <digit sequence>L |
#<hexa digit sequence>L
<signed integer> ::= [<sign>] <unsigned integer>
<signed long_integer> ::= [<sign>] <unsigned long_integer>
<sign> ::= + | -

Examples of signed integer numbers:

1 +100 #FFFF -#4711

Examples of signed long integer numbers:

1L +100L #ABEL -#4711FFFFL

Numbers without a preceding #-character are in base 10. Numbers with a preceding #-character are in base 16.

Character strings are sequences of string elements enclosed by apostrophes. Character strings consisting of a single string element are the constants of the standard type CHAR. Character strings consisting of n (1 <n <=80) enclosed string elements are constants of the type

ARRAY [1..n] OF CHAR

If the character string is to contain an apostrophe, this apostrophe must be written twice. Empty strings are not allowed.

```
<character string> ::= '<string element>
                           {<string element>}  79 '
                                               0
<string element>   ::= <apostrophe image> |
                       <string character> |
                       (:<digit sequence>:)
<apostrophe image> ::= ''
<string character> ::= any ASCII character except
                         EM and NL.
```

A string element of the form (:<digit sequence>:) can be used if a character is difficult to punch or unprintable

(or if it equals EM or NL). The digit sequence must equal the ordinal value of an ASCII character, i.e. be in the closed interval from 0 to 127.

Examples of character strings:

```
'A'   ';'   ''''   '(:10:):)(:25:)'
```

The last character string has the length 4 and contains an NL-character followed by a colon, a right parenthesis, and an EM-character.

3.3 Constant definitions

A constant definition introduces an identifier to denote a constant.

```
<constant definition> ::= <identifier> = <constant>
<constant> ::= <unsigned integer> |
               <unsigned long integer> |
               <constant identifier> |
               <character string>
```

Note: If an identifier is to denote a negative number, this number must be written in the hexadecimal notation (e.g. -1 must be written #FFFF and -2L must be written #FFFFFFFEL, because the representation is 2's complement).

3.4 Type definitions

A type determines the set of values which variables of that type may have and the operations which may be performed upon values of that type. A type definition associates an identifier with a type.

```
<type definition> ::= <identifier> = <type>
<type> ::= <simple type> |
           <structured type> |
           <pointer type>
```

3.4.1 Simple types

```
<simple type> ::= <ordinal type>|
                  <long_integer type>
<ordinal type> ::= <enumerated type>|
                   <subrange type>   |
                   INTEGER |
                   BOOLEAN |
                   CHAR    |
                   <ordinal type identifier>
<long_integer type> ::= LONG_INTEGER |
                        <long_integer type identifier>
```

3.4.1.1 Standard simple types

The values belonging to the standard types may be manipulated by means of predefined primitive operations. The following types are standard:

INTEGER : The values are the subset -32768 to 32767 of the whole numbers, denoted as specified in 3.2 by the signed integer values.

LONG_INTEGER : The values are the subset -2147483648L to 2147483647L of the signed long_integer values as specified in 3.2

BOOLEAN : The values are the truth values denoted by the predefined constant identifiers FALSE and TRUE, where FALSE is the predecessor of TRUE. The ordinal numbers of FALSE and TRUE are 0 and 1 respectively.

CHAR : The values are the character strings of length 1 as specified in 3.2. The ordering relationship between two character values is the same as between their ordinal numbers.

Note: Operators applicable to standard types are specified in 3.7.1.

3.4.1.2 Enumerated types

An enumerated type defines an ordered set of values by enumeration of the identifiers which denote these values. The ordering of the values is determined by the sequence in which their identifiers are listed, i.e. if x precedes y then x<y. The identifiers in the identifier list are mapped onto consecutive nonnegative integer values starting from zero.

```
<enumerated type> ::= (<identifier list>)
<identifier list> ::= <identifier> {,<identifier>}
```

Examples of enumerated types:

```
(RED, YELLOW, GREEN, BLUE, TARTAN)
(MARRIED, DIVORCED, WIDOWED, SINGLE)
```

3.4.1.3 Subrange types

A type may be defined as a subrange of an ordinal type (the host type) by indication of the smallest and the largest value in the subrange. The first constant specifies the lower bound which shall be less than or equal to the upper bound.

```
<subrange type>  ::= <constant>..<constant>
```

Examples of subrange types:

```
1..100
'A'..'Z'
RED..GREEN
```

3.4.2 Structured types

A structured type is characterized by the type(s) of its components and by its structuring method. If a component type is itself structured, the resulting structured type exhibits several levels of structuring.

```
<structured type> ::= <array type>           |
                      <record type>          |
                      <set type>             |
                      <structured type identifier>
```

3.4.2.1 Array types

An array type is a structured type consisting of a fixed number of components that are all of one type, called the component type. The elements of the array are designated by one or more indices, which are values of the corresponding index types. The array type definition specifies both the component type and the index types.

```
<array type> ::=
    ARRAY <lb> <index type> {,<index type>} <rb>
          OF <component type>
<index type> ::= <ordinal type>
<component type> ::= <type>
<lb> ::= [ | (.
<rb> ::= ] | .)
```

Examples of array types:

```
ARRAY [1..100, 'A'..'Z'] OF INTEGER
ARRAY [BOOLEAN] OF COLOUR
```

3.4.2.2 Record types

A record type is a structured type consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a field, its type and an identifier that denotes it. The scope of these field identifiers is the record definition itself, and they are also accessible within a field designator (see 3.5.2.2) referring to a record variable of this type.

The syntax of a record type permits the specification of a variant part. This enables different values, although of identical type, to exhibit structures which differ in the number and/or types of their components. A certain field in the variant part is designated as the tag field. The value of the tag field at any time indicates which variant is assumed by the record variable at that time. Each variant is introduced by one or more case labels, which must be distinct and of a type compatible (see 3.11) with the tag type. A change of variant occurs only when a value is assigned to the tag field. Assignment to the tag field causes the rest of the fields in the variant part to be filled with zerobits. A runtime error will result if a reference is made to a field of a variant other than the current variant.

<record type> ::= RECORD <field list> END
<field list> ::= <fixed part> [; <variant part>]
<variant part>

```
<fixed part> ::= <record section>{; <record section>}
<record section> ::= <identifier list> : <type>
<variant part> ::= CASE <tag field> : <tag type> OF
                        <variant>{ ;<variant> }
<tag field> ::= <identifier>
<tag type> ::= <ordinal type identifier>
<variant> ::= <case constant list>
                  : (<field list>)
<case constant list> ::= <case constant>
                           {, <case constant>}
<case constant> ::= <constant>
```

The ordinal value of the case constants must be contained in the closed interval from 0 to 15. The type in a record section must not be the defining occurence of an enumerated type.

Note: It is a syntax error to place a semicolon in front of the final END in a record type definition.

Examples of record types:

```
RECORD
  YEAR:   INTEGER;
  MONTH:  1..12;
  DAY:    1..31
END

RECORD
  NAME, FIRSTNAME: ARRAY [1..20] OF CHAR;
  CASE S:  SEX OF
    MALE:   (ENLISTED, BEARDED: BOOLEAN);
    FEMALE: (PREGNANT:  BOOLEAN)
END
```

This record type contains a defining occurrence of an enumerated type and is therefore not allowed:

```
RECORD
  COLOUR: (RED, GREEN, BLUE)
END
```

3.4.2.3 Set types

A set type defines the range of values which is the powerset of its base type. Thus each value of a set type is a set whose members are unique values of the base type. The largest and smallest values of integer type permitted as members of a value of a set type are 127 and 0. The base type appearing in a set type must not possess a value outside these limits.

```
<set type> ::= SET OF <base type>
<base type> ::= <ordinal type>
```

Operators applicable to values of set types are specified in 3.7.1.

3.4.3 Pointer types

A pointer type consists of a set of values each identifying one variable of a given type. This set of values is dynamic, in that the variables and the values pointing to them may be created and destroyed during the execution of the program. No operators are specified regarding pointers except the tests for equality and

inequality.

Pointer values are created by the standard procedure NEW. NEW(P) allocates a new variable V in the so-called heap. The programmer is able to reallocate already allocated variables in the heap. This is done through use of the two prefix procedures (see 3.9) MARK and RELEASE:

PROCEDURE MARK (VAR TOP: INTEGER);
Returns in TOP information to be used by RELEASE in recollecting storage allocated by subsequent calls of the standard procedure NEW.

PROCEDURE RELEASE (TOP: INTEGER);
Releases storage allocated by the standard procedure NEW since the call of MARK which returned the value of TOP.

The heap is thus (despite the name) manipulated as a stack.

The pointer value NIL belongs to every pointer type; it points to no element at all.

<pointer type> ::= ∂ <type identifier> |
<pointer type identifier>

3.5 Declaration and denotation of variables

Variable declarations consist of a list of identifiers denoting the new variables, followed by their type.

```
<variable declaration> ::=
                    <identifier list> : <type>
```

A variable declared in the program block (see 3.9) or in a routine block (see 3.6.1) exists from the time the block is activated, until its statement part is completed. This implies that each activation of a block introduces a distinct set of local variables.

Example:

```
A: ARRAY[0..63] OF LONG_INTEGER
C: COLOUR
M: ARRAY[1..10, 1..10] OF INTEGER
HUE1, HUE2: SET OF COLOUR
```

A denotation of a variable designates either an entire variable, a component of a variable, or a variable referenced by a pointer.

```
<variable> ::= <entire variable> |
               <component variable>  |
               <referenced variable>
```

3.5.1 Entire variables

An entire variable is denoted by its identifier:

<entire variable> ::= <variable identifier>

3.5.2 Component variables

A component of a variable is denoted by the variable followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable.

<component variable> ::= <indexed variable> |
<field designator>

3.5.2.1 Indexed variables

A component of an n-dimensional array variable is denoted by the variable followed by n index expressions.

<indexed variable> ::= <array variable>
<lb> <expression>{,<expression>}<rb>
<array variable> ::= <variable>

An array variable shall be a variable of an array type. Each index expression must be compatible (see 3.11) with the corresponding index type specified in the definition of the array type. A runtime error will occur if an index expression is out of range.

Example:

Suppose we have

```
TYPE NAME: ARRAY [1..7] OF CHAR;
VAR
  A: NAME;
  B: ARRAY [BOOLEAN] OF NAME;
  M: ARRAY [1..10, 1..10] OF INTEGER;
```

Then the following are indexed variables:

```
A [7]
B [FALSE] [3]
M [5, 5]
```

The second indexed variable cannot be written B [FALSE,3].

3.5.2.2 Field designators

A component of a record variable is denoted by the record variable followed by the field identifier of the component.

```
<field designator> ::=
      <record variable>.<field identifier>
<record variable> ::= <variable>
```

Example:

If the variable REC is declared

```
REC:  RECORD
        MONTH, YEAR: INTEGER
      END
```

then

REC.YEAR

is a field designator.

3.5.3 Referenced variables

<referenced variable> ::= <pointer variable> ∂
<pointer variable> ::= <variable>

If P is a pointer variable which is bound to a type T, then P denotes that variable and its pointer value, whereas P∂ denotes the variable of type T referenced by P. A runtime error will occur if a pointer variable has the value NIL when it is de-referenced.

3.6 Procedure and function declarations

3.6.1 Procedure declarations

A procedure declaration associates an identifier with a part of a program so that it can be activated by a procedure statement. If a procedure is referenced textually before its procedure block is defined (i.e. referenced within another preceding procedure or function declaration), it must be introduced first by means of its heading followed by the symbol FORWARD. The procedure can then be completed later by repeating its heading without the formal parameter list but followed by the procedure block.

```
<procedure declaration> ::=
            <procedure heading><procedure block> |
            <procedure heading> FORWARD
```

The procedure heading specifies the identifier naming the procedure and the formal parameters (if any).

```
<procedure heading> ::=
     PROCEDURE <identifier> [<formal parameter list>] ;
<procedure block> ::= <routine block>
<routine block> ::= [<definition part>]
                    [<variable declaration part>]
                     <statement part>
<definition part> ::=
          <constant definitions>   |
          <type definitions>|
          <definition part> <constant definitions> |
          <definition part> <type definitions>
```

<constant definitions> ::=
CONST <constant definition>;{ <constant definition;}

<type definitions> ::=
TYPE <type definition> ;{ <type definition>; }

<variable declaration part> ::=
VAR <variable declaration>;{<variable declaration>;}

<statement part> ::= <compound statement>

(<formal parameter list> is defined in 3.6.3)

The algorithmic actions that will be executed upon activation of the procedure by a procedure statement are specified by the statement part of the procedure block.

All identifiers introduced in the formal parameter list and the procedure block are local to the procedure declaration which is called the scope of these identifiers. They are not known outside their scope. In the case of local variables, their values are unpredictable at the beginning of the statement part.

The use of the procedure identifier in a procedure statement within its declaration implies recursive execution of the procedure.

Example:

```
PROCEDURE MINMAX (A:LIST;VAR MIN,MAX:INTEGER);
VAR TEMP, I: INTEGER;
BEGIN
   MIN :=  A [1] ;
   MAX :=  MIN;
   FOR I := 2 TO LIST_MAX DO
     BEGIN
       TEMP :=  A [I] ;
       IF TEMP > MAX THEN
         MAX := TEMP
       ELSE
       IF TEMP < MIN THEN
         MIN := TEMP
     END
END
```

3.6.2 Function declarations

A function declaration serves to define a part of the program that computes a value of a simple type or a pointer type. A function is activated by evaluation of a function designator (see 3.7.2), that is a constituent of an expression.

If a function is referenced textually before its function block is defined, it must be introduced first by means of its heading followed by the symbol FORWARD, and then completed later by repeating its heading without the formal parameter list, and without the result type, followed by the function block.

```
<function declaration> ::=
        <function heading><function block>  |
        <function heading> FORWARD
```

The function heading specifies the identifier naming the function, the formal parameters (if any), and the type of the function.

```
<function heading> ::= FUNCTION <identifier>
  [<formal parameter list>] : <result type>;
<result type> ::= <simple type identifier>  |
                  <pointer type identifier>
<function block> ::= <routine block>
```

The algorithmic actions that will be executed upon activation of the function by a function designator are specified by the statement part of the function block.

The function block should contain at least one assignment statement that assigns a value to the function identifier. The result of the function is the last value assigned. If no assignment occurs the value of the function is unpredictable.

Occurrence of the function identifier in a function designator within its declaration implies recursive execution of the function.

Examples:

```
FUNCTION GCD(M,N: INTEGER):INTEGER;FORWARD

FUNCTION LONG_MOD(A,B:LONG_INTEGER): LONG_INTEGER;
BEGIN
  LONG_MOD := A-(A/B)*B
END

FUNCTION GCD"(M,N:INTEGER):INTEGER";
BEGIN
  IF N = 0 THEN
   GCD := M
  ELSE
   GCD := GCD(N, M MOD N)
END
```

3.6.3 Parameters

In the following the term routine will be used for a procedure or function.

There are two kinds of parameters:

Value parameters and variable parameters. A parameter section without a preceding specifier is a list of value parameters; a parameter section with the specifier VAR preceding is a list of variable parameters.

```
<formal parameter list>  ::=
    (<formal parameter section>
    { ;  <formal parameter section } )

<formal parameter section>  ::=
    [VAR]  <identifier list> :
    [UNIV] <type identifier>
```

A value parameter represents an expression that is evaluated when the routine is called. Its value cannot be changed by the routine. In the case of a variable parameter, the actual parameter must be a variable, and the corresponding formal parameter represents this actual variable during the entire execution of the routine. Variable parameters are called by reference, and the address is evaluated when the routine is called.

In general formal and actual parameters must be compatible. However, there are 2 exceptions:

1) The word UNIV in front of the type identifier in a formal parameter section suppresses compatibility checking. An actual parameter of

type T1 is compatible with a formal UNIV-parameter of type T2 if both types are not of pointer type (or do not contain a component of pointer type) and if variables of both types take up the same number of machine words. In addition, if the formal parameter is of record or array type, the actual parameter must be a variable. The type checking is only suppressed in routine calls. Inside the given routine the formal parameter is considered to be of type T2, and outside the routine call the actual parameter is considered to be of type T1. The strict type checking in CR80 PASCAL is generally a great advantage to the programmer. Therefore the UNIV-loophole should be used with care. However, situations may occur, especially when CR80 PASCAL is being used as a systems programming language, in which the UNIV-facility is applicable - or even indispensable.

2) An actual parameter corresponding to a formal value parameter of type ARRAY [1..N] OF CHAR may be a character string (i.e. of type ARRAY [1..M] OF CHAR) of any length. This relaxation makes it possible e.g. to write one single procedure to print character strings of any length.

3.6. 4 Standard Procedures

The only standard procedure in CR80 PASCAL is the procedure NEW. NEW(P) allocates a variable V with unpredictable contents in the heap. A pointer to V will be assigned to the pointer variable P. A runtime error will occur if allocation is impossible.

3.6. 5 Standard Functions

ABS(X) X must be an expression of integer or long_integer type. The result (of the same type as X) is the absolute value of X. Overflow may occur.

CHR(X) X must be an expression of integer type. The result is the value of char type whose ordinal number is equal to the expression X.

LONG(X) X must an expression of integer type. The result is the long_integer with the same value as X.

ORD(X) X must be an expression of char type. The integer type result is equal to the ordinal value of the character.

PRED(X) X must be an expression of ordinal type. The function will yield a value of the same type as X whose ordinal number is one less than that of the expression X. No check for "underflow".

SHORT(X) X must be an expression of long_integer type. The result is the integer with the same value as X. Overflow may occur.

SUCC(X) X must be an expression of ordinal type. The function will yield a value of the same type as X whose ordinal number is one greater than that of the expression X. No check for "overflow".

3.7 Expressions

Expressions consist of operators and operands, i.e. variables, constants and function designators. The rules of composition specify operator precedences according to four classes of operators. The operator NOT has the highest precedence, followed by the multilying operators, then the adding operators and signs, and finally, with the lowest precedence, the relational operators. Sequences of two or more operators of the same precedence separated by operands are executed from left to right. The rules of precedence are reflected by the following syntax:

```
<factor>  ::=  <variable> |
               <constant> |
               NIL  |
               <function designator> |
               <set> |
               (  <expression>  )  |
               NOT  <factor>

<set>  ::=     <lb>  <expression list>  <rb>

<expression list>::=   [<expression> { , <expression>} ]

<term>              ::= <factor> |
                        <term> <multiplying operator> <factor>


<simple expression>  ::= <term>  |
        <simple expression> <adding operator>  <term>  |
        <sign>  <term>

<expression>  ::=  <simple expression>  |
     <simple expression> <relational operator> <simple expression>
```

Expressions which are members of a set must be compatible with the base type of the set. [ ] or (..) denote the empty set.

Examples are as follows:

Factors:
```
X
15
(X + Y + Z)
ORD ('O')
[RED, GREEN]
NOT P
```

Terms:
```
X * Y
I/(1-I)
(X <= Y) AND (Y < Z)
```

Simple expressions:

```
X + Y
- X
I * J + 1
```

Expressions:
```
X = 15
P <= Q
(I < J)  =  (J < K)
```

3.7.1 Operators

<multiplying operator> ::= * | / | DIV | MOD | AND

<adding operator> ::= + | - | OR

<relational operator> ::= = | <> | < | > | <= | >= | IN

3.7.1.1 Aritmetic operators

The types of operands and results for dyadic and monadic operations are shown in the following two tables:

Dyadic operations

operator	operation	type of operands	type of result
+	addition	INTEGER LONG_INTEGER	INTEGER LONG_INTEGER
-	subtraction	INTEGER LONG_INTEGER	INTEGER LONG_INTEGER
*	multiplica- tion	INTEGER LONG_INTEGER	INTEGER LONG_INTEGER
/	division	LONG_INTEGER	LONG_INTEGER
DIV	division	INTEGER	INTEGER
MOD	remainder	INTEGER	INTEGER

Both / and DIV are division with truncation (e.g. 3 DIV 2 = 1, (-3) DIV 2 = -1, (-3) DIV (-2) = 1, 3L / 2L = 1L, and 4L / (-2L) = -2L).

Monadic operations

operator	operation	type of operand	type of result
+	identity	INTEGER LONG_INTEGER	INTEGER LONG_INTEGER
-	sign. inversion	INTEGER LONG_INTEGER	INTEGER LONG_INTEGER

Note: The symbol - is also used as a set operator.

3.7.1.2 Boolean operators

The types of operands and results for boolean operations are shown in the following table:

operator	operation	type of operands(s)	type of results
OR	logical or	BOOLEAN	BOOLEAN
AND	logical and	BOOLEAN	BOOLEAN
NOT	logical negation	BOOLEAN	BOOLEAN

Note: The symbols AND and OR are also used as set operators.

3.7.1.3 Set operators

The types of operands and results for set operations are shown in the following table:

operator	operation	type of operands	type of result
OR - AND	set union set difference set intersection	Any set type T	T

3.7.1.4 Relational operators

The types of operands and results for relational operations are shown in the following table:

operator	type of operands	type of result
= <>	any type	BOOLEAN
< >	any simple or string type	BOOLEAN
<= >=	any set, simple or string type	BOOLEAN
IN	left operand: any ordinal type T right operand: set of T	BOOLEAN

The operands of =, <>, <, >, <= and >= shall be of compatible type.

The operators =, <>, <, > stand for "equal to", "not equal to", "less than" and "greater than" respectively. Except when applied to sets, the operators <= and >= stand for "less than or equal to" and "greater than or equal to" respectively.

If U and V are operands of set type, U <= V denotes the inclusion of U in V, and U >= V denotes the inclusion of V in U.

Since the BOOLEAN type is an ordinal type with FALSE < TRUE, then if P and Q are operands of BOOLEAN type, P = Q denotes their equivalence and P <= Q means P implies Q.

When the relational operators =, <>, <, >, <=, >= are used to compare strings, they denote lexicographic ordering according to the ordering of the character set.

The operator IN yields the value TRUE if the value of the operand of ordinal type is a member of the set, otherwise it yields the value FALSE. In particular, if the ordinal value is outside the closed interval from 0 to 127 a runtime error will occur.

3.7.2 Function designator

A function designator specifies the activation of a function denoted by the function identifier. If necessary the function designator shall contain a list of actual parameters that are bound to their corresponding formal parameters defined in the function declaration. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively, and the number of actual parameters must be equal to the number of formal parameters. An actual parameter must be compatible with its corresponding formal parameter. The selection of an actual variable parameter and the evaluation of an actual value parameter are done once before the function is activated.

```
<function designator>  ::=
          <function identifier> [ <actual parameter list> ]

<actual parameter list>  ::=
          (<actual parameter> { , <actual parameter> })

<actual parameter>     ::=    <expression> |
                              <variable>
```

Examples:

```
LONG_MOD(A, 10L)
GCD(147, K)
ORD(F'a)
```

3.8 Statements

Statements denote algorithmic actions, and are said to be executable.

```
<statement>  ::=  <simple statement> |
                  <structured statement>
```

3.8.1 Simple statements

A simple statement is a statement of which no part constitutes another statement. The empty statement consists of no symbols and denotes no action.

```
<simple statement>  ::=  <assignment statement> |
                         <procedure statement>  |
                         <empty statement>

<empty statement>   ::=
```

3.8.1.1 Assignment statements

The assignment statement serves to replace the current value of a variable or function identifier by a new value specified as an expression.

```
<assignment statement>  ::=
        <variable>  :=  <expression>  |
        <function identifier>  :=  <expression>
```

The expression must be compatible with the variable or the function identifier.

Examples:

```
X :=  X + Y
P :=  (1  <=  I) AND (C IN [RED, BLUE])
```

3.8.1.2 Procedure Statements

A procedure statement serves to execute the procedure denoted by the procedure identifier. If necessary the procedure statement shall contain a list of actual parameters that are bound to their corresponding formal parameter defined in the procedure declaration. The correspondence is established by the positions of the parameters in the list of actual and formal parameters respectively, and the number of actual parameters must be equal to the number of formal parameters.

An actual parameter must be compatible with its corresponding formal parameter. The selection of an actual variable parameter and the evaluation of an actual value parameter are done once before the procedure is activated.

<procedure statement> ::=
<procedure identifier> [<actual parameter list>]

Examples:

```
MINMAX(LIST, MIN, MAX)
NEW(P)
```

3.8.2 Structured Statements

Structured statements are constructs composed of other statements which have to be executed either in sequence (compound statement), conditionally (conditional statements), repeatedly (repetitive statements), or within a special scope (with statements).

```
<structured statement>  ::=
                <compound statement>     |
                <conditional statement>  |
                <repetitive statement>   |
                <with statement>
```

3.8.2.1 Compound statements

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. The symbols BEGIN and END act as statement brackets, and the semicolon is used as a statement separator.

<compound statement> ::=
BEGIN <statement>{ ; <statement>} END

Example:

BEGIN Z := X; X := Y END

3.8.2.2 Conditional statements

A conditional statement selects for execution a single one of its component statements.

<conditional statement> ::=
<if statement> |
<case statement>

3.8.2.2.1 If statements

<if statement> ::= IF <boolean expression>
THEN <statement> [<else-part>]

<else-part> ::= ELSE <statement>

<boolean expression> ::= <expression>

If the boolean expression yields the value TRUE, the statement following the THEN is executed. If the boolean expression yields FALSE, the action will depend on the existence of an else-part. If the else-part is present the statement following the ELSE is executed, otherwise an empty statement is executed.

The so-called "dangling else" ambiguity is resolved by pairing an else-part with the nearest preceding unpaired THEN. Thus the construct

```
IF <expression 1>  THEN
 IF <expression 2>  THEN  <statement 1>
    ELSE  <statement 2>
```

is equivalent to

```
IF  <expression 1>  THEN
    BEGIN
      IF <expression 2> THEN <statement 1>
      ELSE <statement 2>
    END
```

Example:

```
IF I < J THEN I  :=  J ELSE I  :=  J - 1
```

3.8.2.2.2 Case statements

The case statement consists of a case index and a list of statements, each being preceded by one or more case constants. All case constants shall be distinct and shall be of the same ordinal type as the case index.

The case statement specifies execution of the statement whose case constant is equal to the value of the case index upon entry to the case statement.

```
<case statement>  ::=  CASE  <case index>  OF
      <case list element> { ;  <case list element>} END

<case list element>  ::=
      <case constant list>  :  <statement>

<case index>  ::=  <expression>
```

Note: It is a syntax error to place a semicolon immediately before the last END of a case statement.

The ordinal value of a case constant must belong to the closed interval from 0 to 127.

If the case index does not match a case constant, one of two things will happen:

1) There is a case constant with a larger ordinal value than the case index value, and there is also a case constant with a smaller ordinal value than the case index value: The empty statement will be executed.

2) The conditions in 1) are not fulfilled: A runtime error will result.

Example:

```
CASE OPERATOR OF
     PLUS:  X  :=  X  +  Y;
     MINUS: X  :=  X  -  Y;
     TIMES: X  :=  X  *  Y
END
```

3.8.2.3 Repetitive statements

Repetitive statements specify that certain statements are to be executed repeatedly. If the number of repetitions is known before the repetions are started, the for statement is the appropriate construct to express this situation; otherwise the while or repeat statement should be used.

<repetitive statement> ::= <repeat statement> |
<while statement> |
<for statement>

3.8.2.3.1 Repeat statements

<repeat statement> ::=
REPEAT <statement sequence> UNTIL <boolean expression>

<statement sequence> ::=
<statement> {; <statement>}

The sequence of statements between the tokens REPEAT and UNTIL is repeatedly executed until the boolean expression yields the value TRUE on completion of the statement sequence. The statement sequence is executed at least once, because the boolean expression is evaluated after execution of the statement sequence.

Example:

```
REPEAT
   K  :=  I MOD J;
   I  :=  J;
   J  :=  K
UNTIL J  =  0
```

3.8.2.3.2 While statements

```
<while statement>  ::=
    WHILE <boolean expression> DO <statement>
```

The statement is repeatedly executed until the expression becomes FALSE. If its value is FALSE at the beginning, the statement is not executed at all.

Example:

```
WHILE A [I] <>  X  DO
  BEGIN
    A [I] :=  Y;
    I   :=  I + 1
  END
```

3.8.2.3.3 For statements

The for statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the control variable of the for statement.

```
<for statement>   ::=
     FOR <control variable>   :=  <initial value>
         TO <final value>  DO  <statement>  |

     FOR <control variable>  :=  <initial value>
     DOWNTO <final value>  DO  <statement>

<control variable> ::=  <entire variable>
<initial value>    ::=  <expression>
<final value>      ::=  <expression>
```

The control variable shall be of an ordinal type, and the initial value and final value shall be of a type compatible with this type. The final value is only calculated once.

The programmer is not allowed to change the value of the control variable within the statement of the for statement.

The for statement

```
FOR V  :=  E1 TO E2 DO BODY
```

where E1 and E2 are general expressions, is equivalent to

```
BEGIN
  TEMP1  :=  E1;
  TEMP2  :=  E2;
  IF TEMP1  <=  TEMP2  THEN
    BEGIN
      V  :=  TEMP1;
      BODY;
      WHILE V  <>  TEMP2 DO
        BEGIN
          V  :=  SUCC(V);
          BODY
        END
    END
END
```

and the for statement

```
FOR V  := E1 DOWNTO E2 DO BODY
```

is equivalent to

```
BEGIN
  TEMP1  :=  E1;
  TEMP2  :=  E2;
  IF TEMP1 >= TEMP2 THEN
     BEGIN
       V  :=  TEMP1;
       BODY;
       WHILE V  <>  TEMP2 DO
         BEGIN
           V  :=  PRED(V);
           BODY
         END
     END
END
```

where TEMP1 and TEMP2 are auxiliary variables that do not occur elsewhere in the program.

Examples of for statements are:

```
FOR I  :=  2 TO 63 DO
  IF A [I] >  MAX THEN MAX  :=  A [I]
```

```
FOR I  :=  1  TO N DO
  FOR J  :=  1  TO N DO
    BEGIN
      X  := 0;
      FOR K  := 1 TO N DO
        X  :=  X + M1 [I,  K ]*  M2 [K,  J] ;
      M [I,   J ] :=  X
    END
```

3.8.2.4 With statements

```
<with statement>  ::=
 WITH  <record variable list> DO  <statement>

<record variable list>  ::=
   <record variable> { ,   <record variable>}
```

Within the component statement of the with statement, the components (fields) of the record variable(s) specified by the record variable list can be denoted by their field identifier only, i.e. without preceding them with the denotation of the entire record variable. The scope containing the field identifiers of the specified record variable(s) is effectively opened, so that the field identifiers may occur as variable identifiers.

The statement

```
WITH V1, V2, .., VN DO S
```

is equivalent to

```
WITH V1 DO
  WITH V2 DO, ..., VN DO S
```

Example:

The statement

```
IF DATE.MONTH = 12  THEN
   BEGIN
     DATE.MONTH  :=  1;
     DATE.YEAR   :=  DATE.YEAR + 1
   END
 ELSE
   DATE.MONTH  :=  DATE.MONTH + 1
```

is equivalent to the with statement

```
WITH DATE DO
  IF MONTH = 12 THEN
    BEGIN
      MONTH :=  1;
      YEAR  :=  YEAR + 1
    END
  ELSE
    MONTH  :=  MONTH + 1
```

3.9 Programs and prefix

A CR80 PASCAL program consists of a prefix followed by a program block.

<program > ::= <prefix> <program block>

A CR80 PASCAL program interacts with its runtime environment by means of procedures and functions implemented within that environment. These interface procedures and functions together with their parameter types are declared in the prefix. The prefix enables the compiler to make complete type checking of calls to the runtime environment.

The compiler has virtually no inherent knowledge about the runtime milieu for which it generates code. Instead the programmer supplies the neccessary information by giving a prefix tailored to that milieu. Needless to say that the programmer should be on the alert that he is supplying the right prefix.

<prefix> ::= [<definition part>]
[<prefix routines>]
<program heading>

<prefix routines> ::= <procedure heading> |
<function heading> |
<prefix routines> <procedure heading> |
<prefix routines> <function heading>

```
<program heading>  ::=
    PROGRAM  <identifier>  <formal parameter list>;

<program block>  ::=
          [<definition part>]
          [<variable declaration part>]
          [<routine declarations>]
           <compound statement> .

<routine declarations>  ::=
          <procedure declaration>  |
          <function declaration>  |
          < routine declarations>  <procedure declaration>  |
          <routine declarations>  <function declaration>
```

The variables declared in the program block exist throughout the execution of the program. They are called global variables. Their values are unpredictable at the beginning of the compound statement.

The formal parameter list in the program heading can be used by a loader process to pass information to the program. The program may pass information back to the loader process, if the parameter list contains a variable parameter.

Examples of CR80 PASCAL programs:

```
PROGRAM P;
BEGIN END.
```

```
CONST
  TABMAX = 15;
TYPE
  INDEX  = 1..TABMAX;
  BUFPTR = @BUFFER;
  BUFFER = ARRAY [INDEX] OF INTEGER;
FUNCTION IAND(M1, M2: INTEGER) : INTEGER;
PROGRAM PIP(VAR PTR: BUFPTR);
TYPE
  REC = RECORD
          A, B: CHAR;
          C: BUFFER
        END;
CONST
  PAP = 4711;
VAR
  I: INTEGER;
  POP, PUP: REC;
PROCEDURE INIT;
BEGIN
  WITH POP DO
    BEGIN
      A := 'A';
      B := 'C';
      C := PTR@;
      I := C[7];
    END;
  PUP := POP;
END;

BEGIN
  INIT;
  IF PTR@[1] = PAP THEN
    I := IAND(POP.C[2], #00FF);
  PTR@[TABMAX] := I;
END.
```

3.10 Scope rules

A scope is a region of program text in which an identifier is used with a definite meaning. More precisely a scope is (a part of) the program, (a part of) the prefix, (a part of) the program block, (a part of) a procedure or function, or a record variable or a with-statement.

The general rule is that an identifier must be introduced before it is used. However, in order to make list processing feasible, it is allowed in pointer type definitions to refer to a type that has not yet been defined.

When a scope is defined within another scope we have an outer scope and an inner scope that are nested. An identifier can only be introduced with one meaning in a scope. It can, however, be introduced with another meaning in an inner scope. In that case the inner meaning applies in the inner scope, and the outer meaning applies in the outer scope.

Components of a record variable can be referenced through field designators or with-statements. The record variable within which components are selected must be known in the scope in which the selection is indicated.

The hierarchy of scopes can be illustrated like this:

Universal level
Prefix level
Program block level
Routine level
Nested with statements within routines
Nested with statements in the compound statement of the program block.

At the universal level the following standard identifiers exist: FALSE, TRUE, INTEGER, LONG_INTEGER, BOOLEAN, CHAR, NIL, NEW, ABS, CHR, ORD, LONG, SHORT, PRED, and SUCC.
The predefined meaning of these standard identifiers may be overruled by declarations and/or definitions in inner scopes.

The following program tries to illustrate how the interpretation of an identifier may change within a program:

```
"ILLUSTRATION OF SCOPE RULES"

CONST CHAR = 13;
"IT IS NOW IMPOSSIBLE TO DECLARE VARIABLES OF STANDARD TYPE CHAR"
TYPE PTR = @REC;
"THE TYPE REC HAS NOT BEEN DEFINED YET"  -
     REC = RECORD
             NIL : INTEGER;
             A: BOOLEAN;
             CHAR: LONG_INTEGER
           END;

PROCEDURE PROC(I, J: INTEGER);
"ASSUME THIS IS THE ONLY PREFIX ROUTINE"

PROGRAM MAIN(PARM: INTEGER);
TYPE SUB1 = 1..CHAR; "CORRESPONDS TO 1..13"
CONST CHAR = 'C';
TYPE SUB2 = 'A'..CHAR; "'A'..'C'"
     P = @REC; "POINTER TO REC IN PREFIX"
     REC = RECORD
             A: LONG_INTEGER;
             CHAR: PTR
           END;
     Q = @REC; "POINTER TO REC IMMEDIATELY ABOVE"
VAR P_PTR: P;
    Q_PTR: Q;
    SUB1_VAR: SUB1;
    SUB2_VAR: SUB2;
    A: INTEGER;

PROCEDURE PROC1(P: INTEGER);
VAR
   A: REC;
BEGIN
   A.A := 4711L;
   NEW(A.CHAR);
   WITH A.CHAR@ DO
      CHAR := 4712L;
   WITH A DO
      CHAR := NIL;
   PROC(1, 0); "CALL OF PREFIX ROUTINE"
END;

PROCEDURE PROC(I, J: INTEGER);
BEGIN
   "IN THIS PROCEDURE THE IDENTIFIER A HAS 3 MEANINGS"
   A := 1;


   WITH Q_PTR@ DO
      BEGIN
         A := 1L;
         WITH P_PTR@ DO
            A := TRUE;
      END;
END;

PROCEDURE PROC2;
BEGIN
   PROC(1, 0); "CALL OF THE ABOVE DEFINED PROCEDURE"
END;

BEGIN
   SUB1_VAR := 1;
   SUB2_VAR := 'B';
   NEW(P_PTR);
   NEW(Q_PTR);
   Q_PTR@.CHAR@.CHAR := LONG(ORD(CHAR));
   "IN THE STATEMENT ABOVE CHAR HAS 3 MEANINGS"
   P_PTR@.NIL := 17;
   P_PTR := NIL;
END.
```

3.11 Type Compatibility

Two types are compatible if

1) they are defined by the same type definition

or

2) both are subranges of a single type

or

3) they are string types of the same length

or

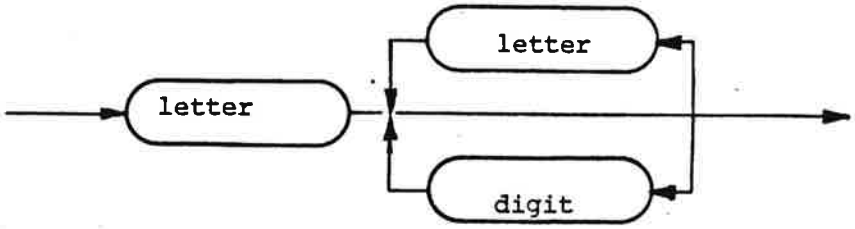
4) they are set types whose members are of the same ordinal type

or

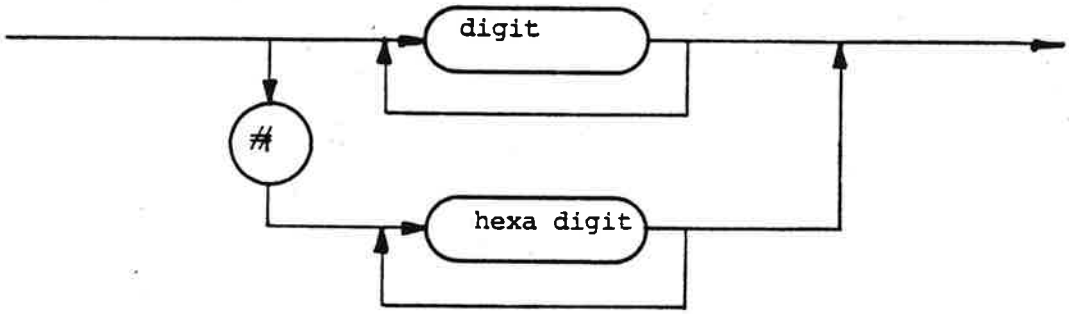
5) they are set types, one (or both) of which is the null set type.

3.12 Syntax graphs

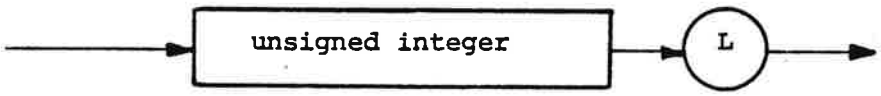
identifier



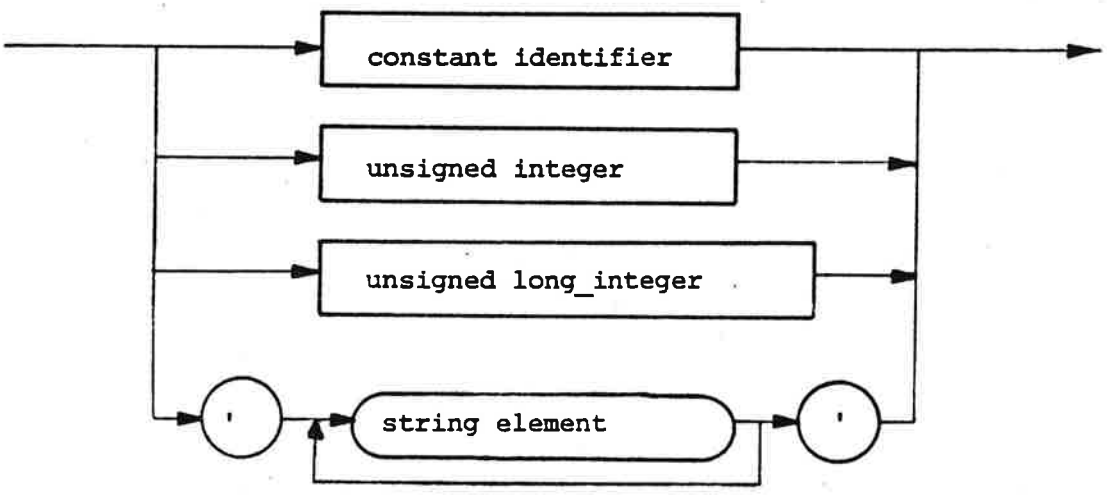
unsigned integer



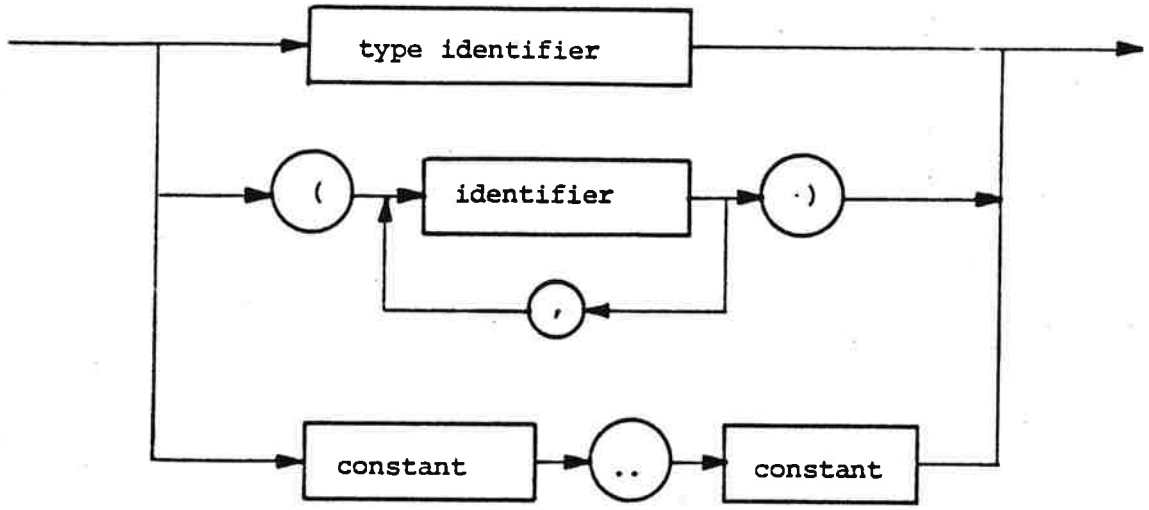
unsigned long_integer



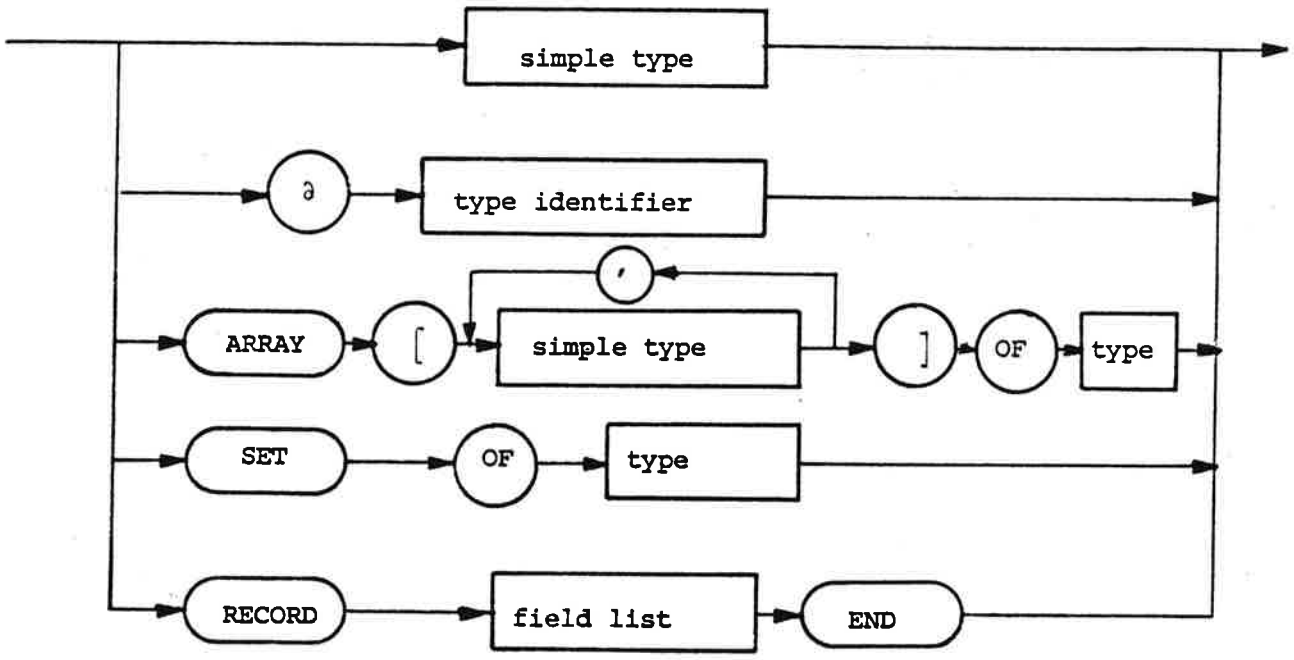
constant



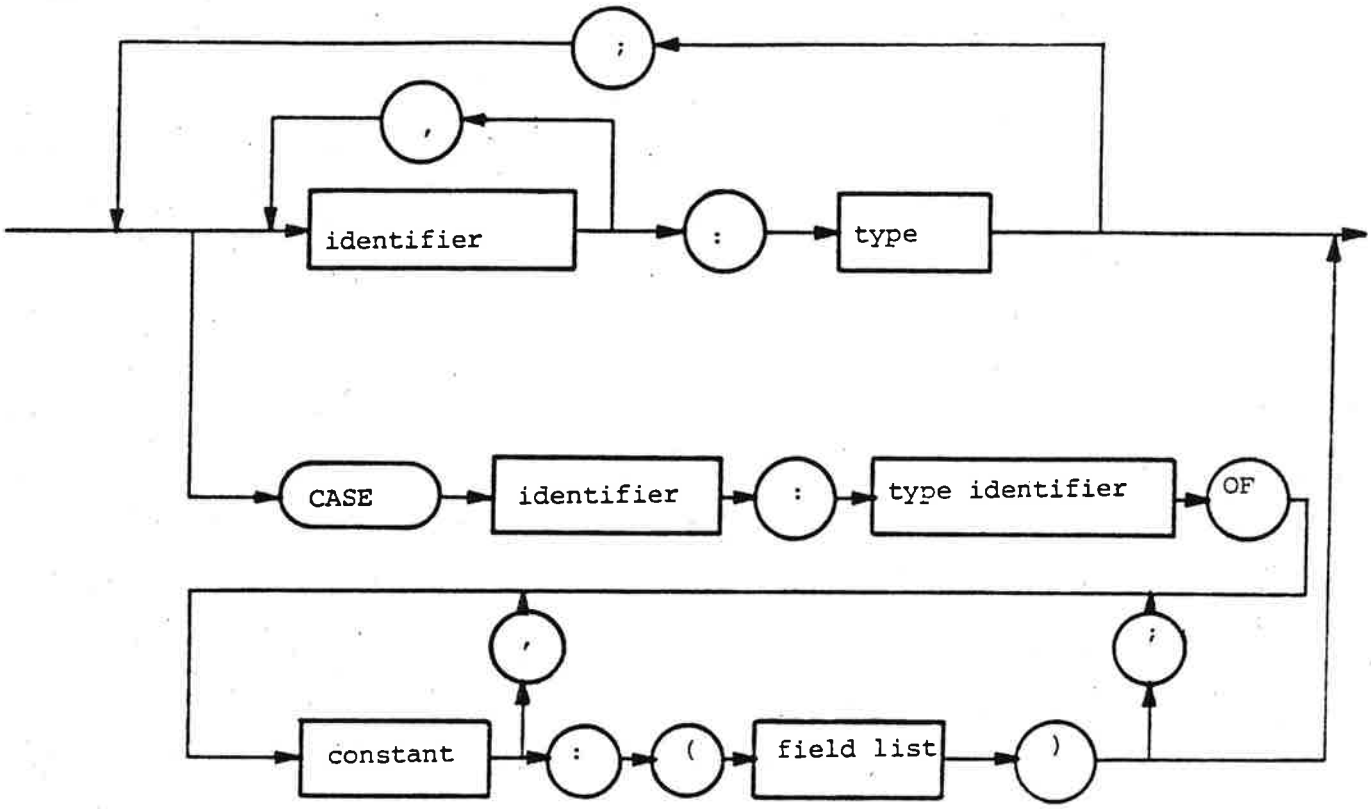
simple type



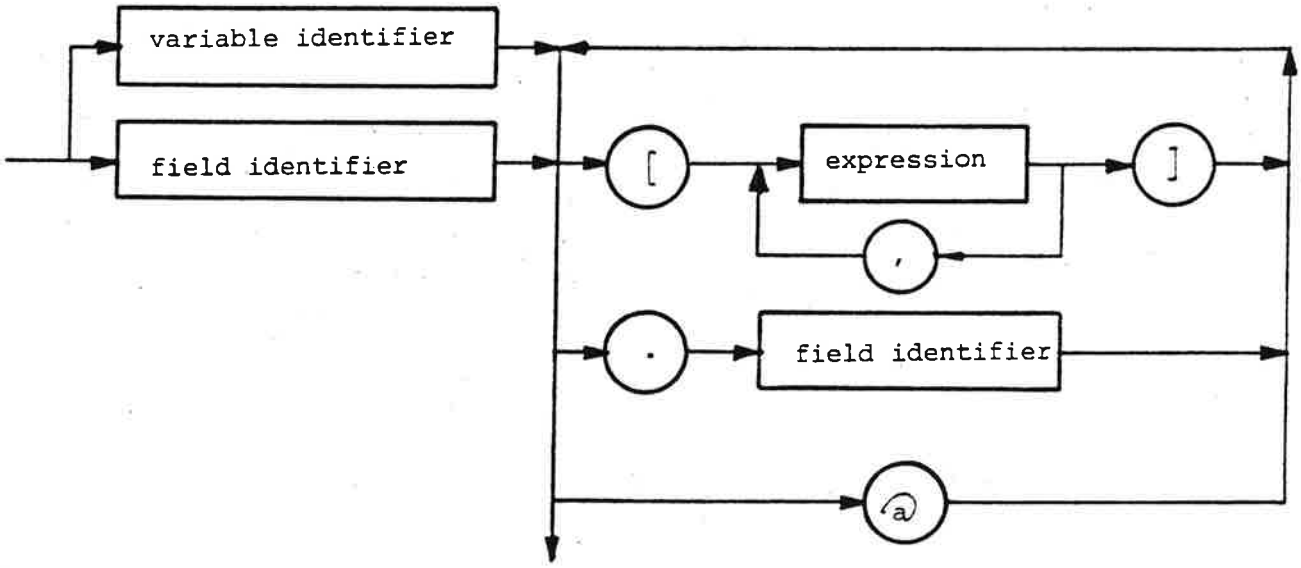
type



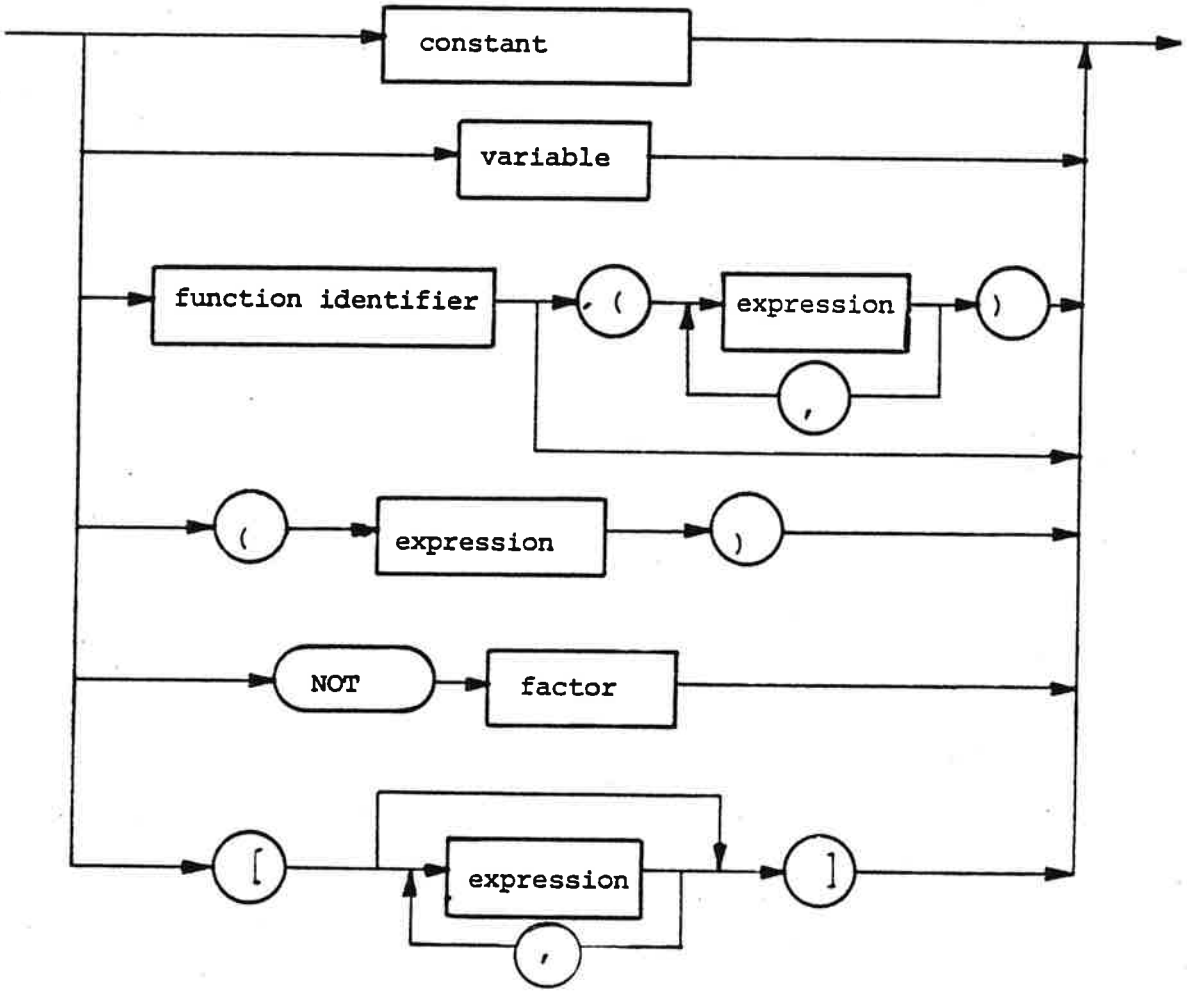
field list



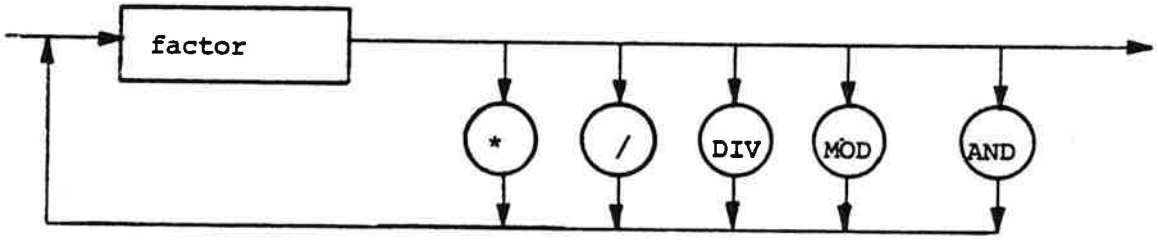
variable



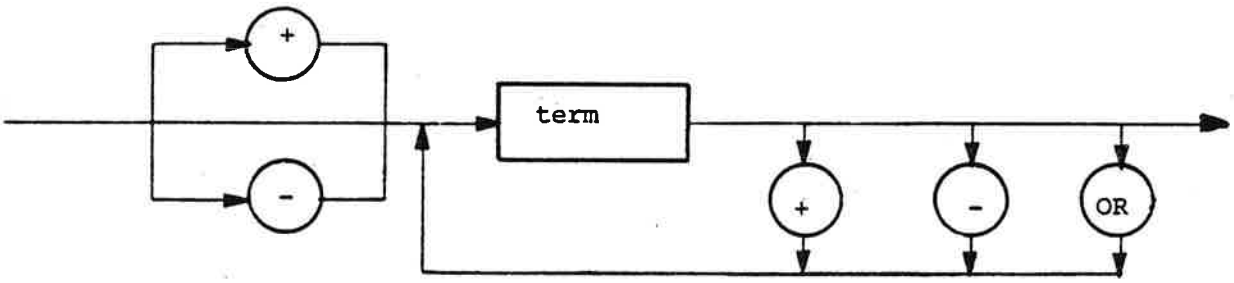
factor


constant
variable
function identifier
(
expression
)
,
(
expression
)
NOT
factor
[
expression
]
,

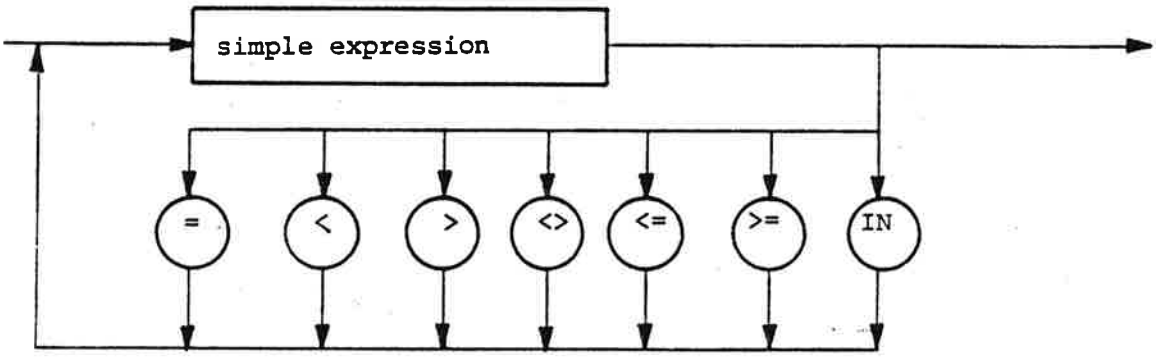
term


factor
*
/
DIV
MOD
AND

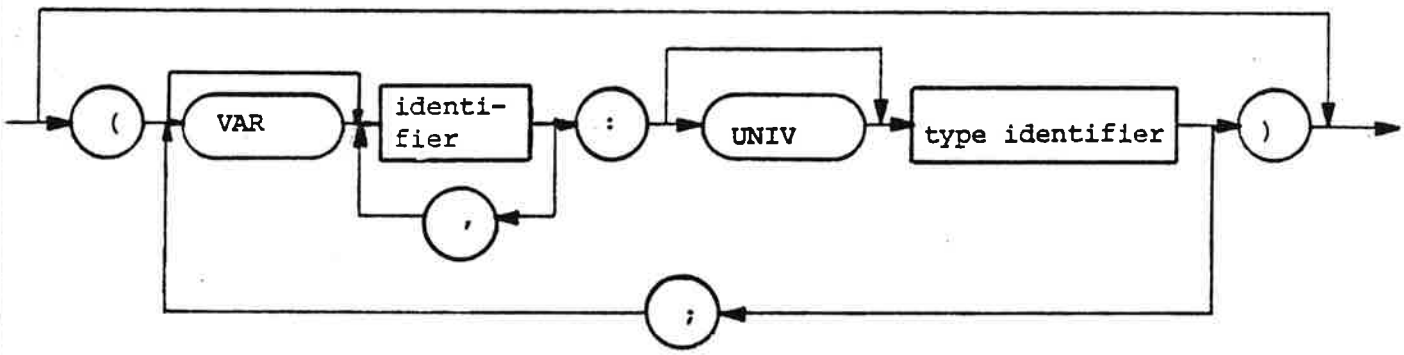
simple expression



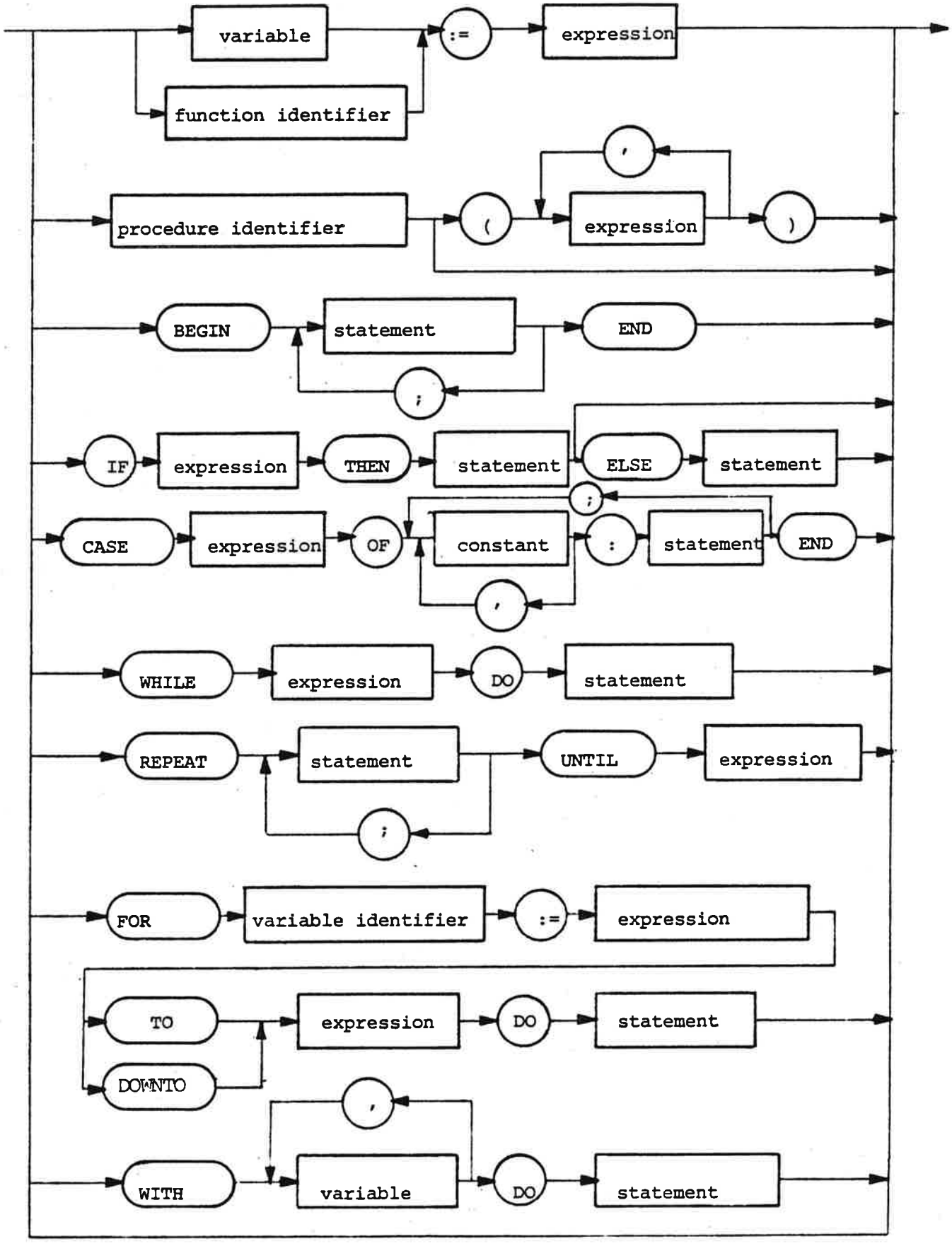
expression



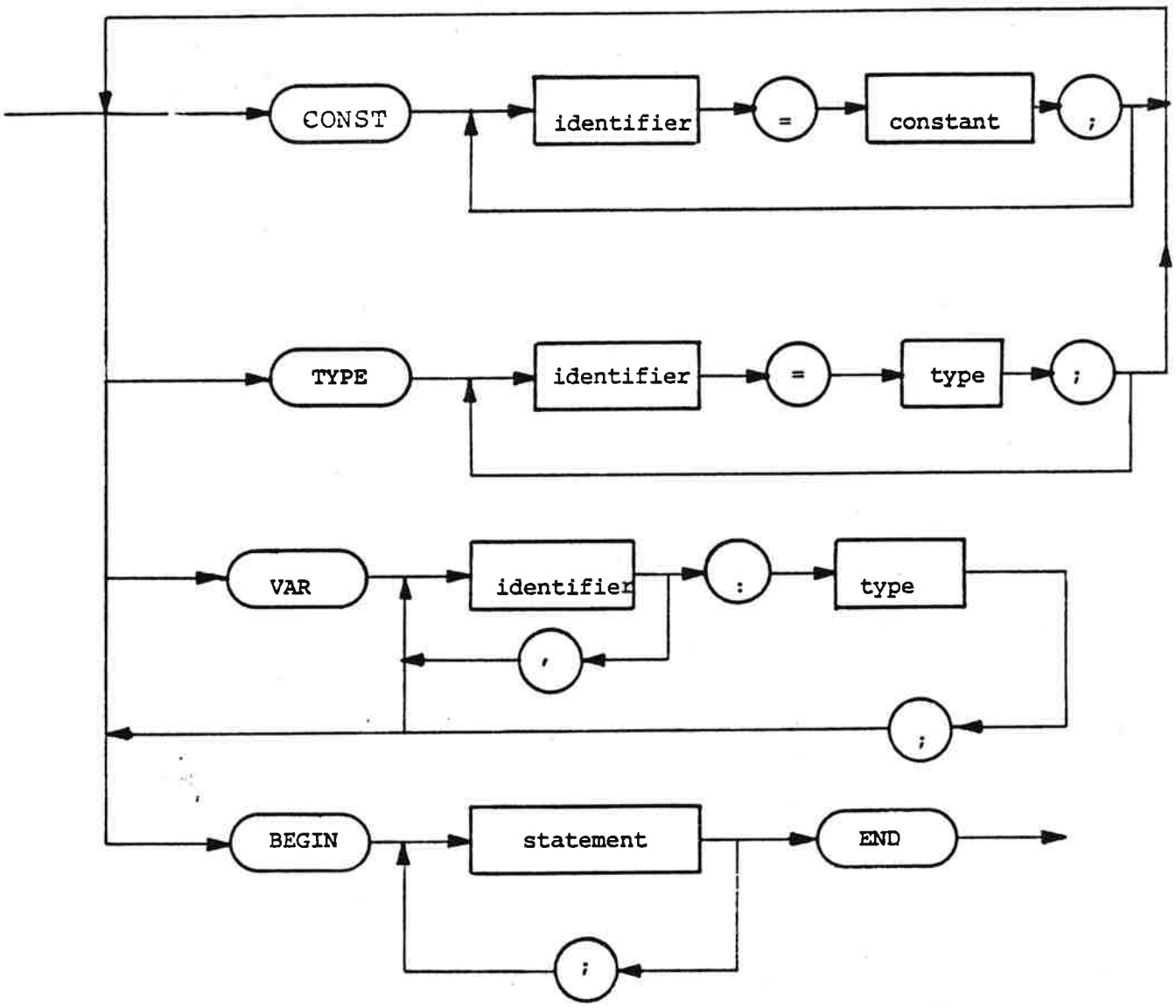
parameter list



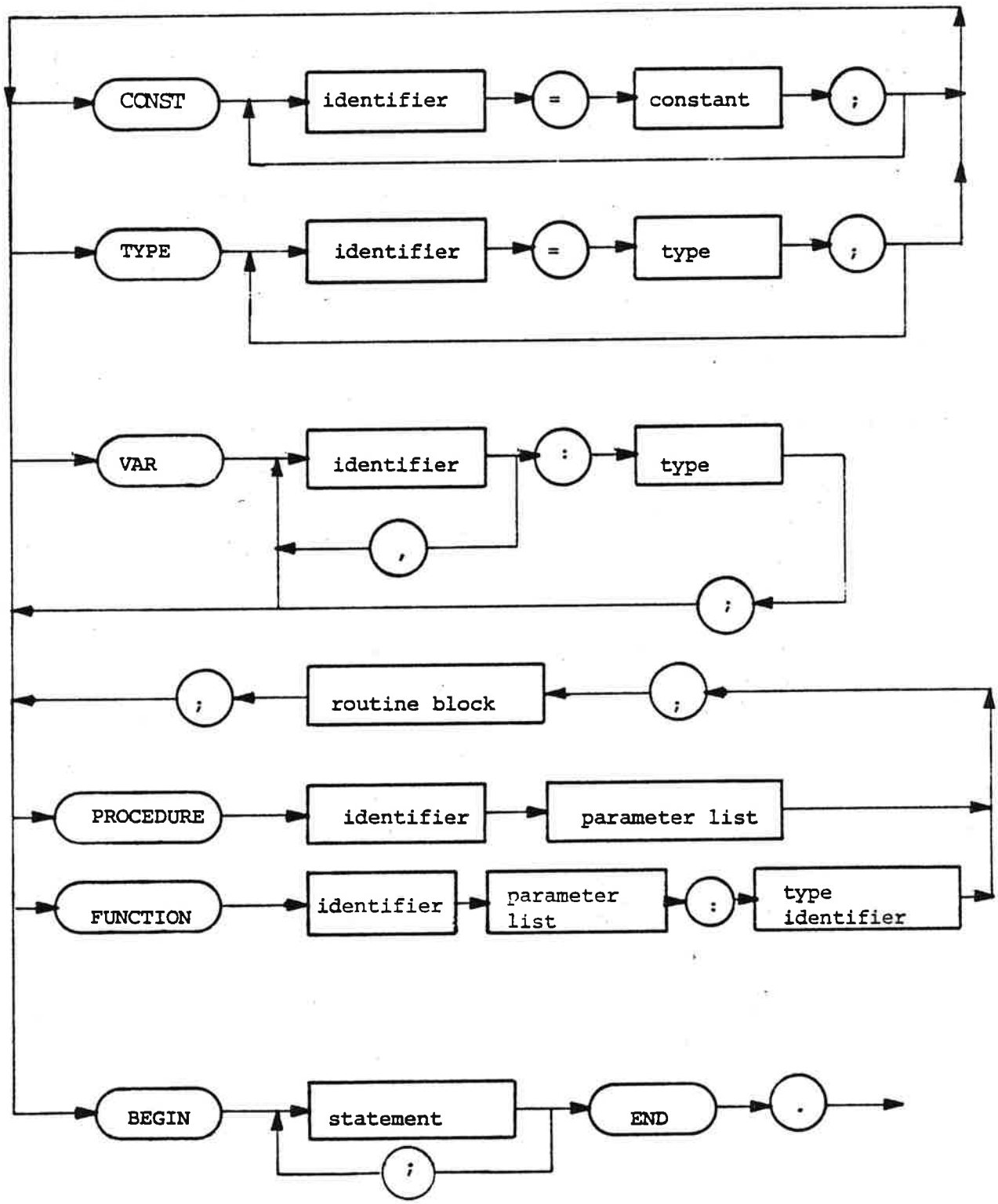
statement


variable
:=
expression
function identifier
,
procedure identifier
(
expression
)
BEGIN
statement
END
;
IF
expression
THEN
statement
ELSE
statement
;
CASE
expression
OF
constant
:
statement
END
,
WHILE
expression
DO
statement
REPEAT
statement
UNTIL
expression
;
FOR
variable identifier
:=
expression
TO
expression
DO
statement
DOWNTO
,
WITH
variable
DO
statement

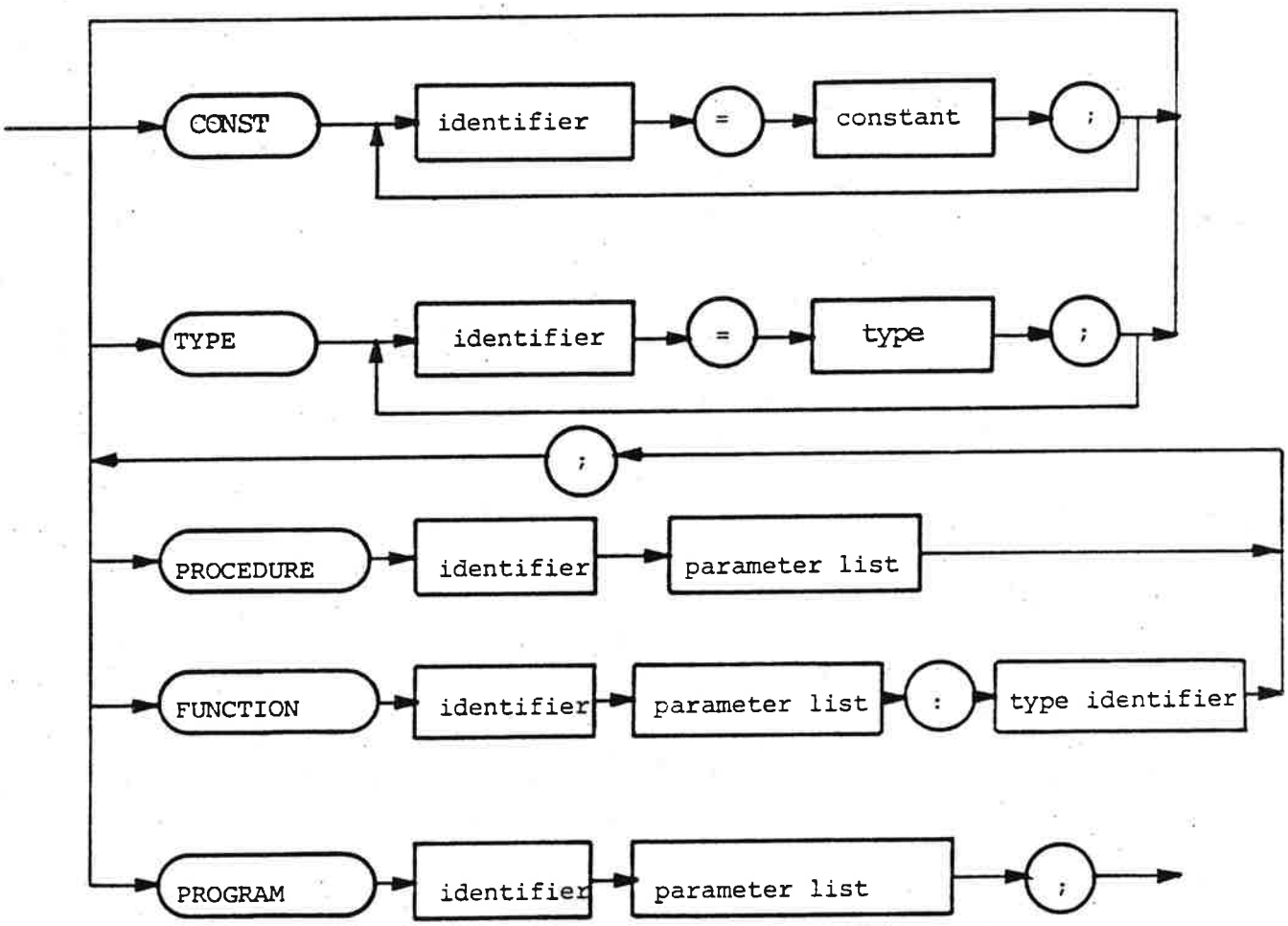
routine block



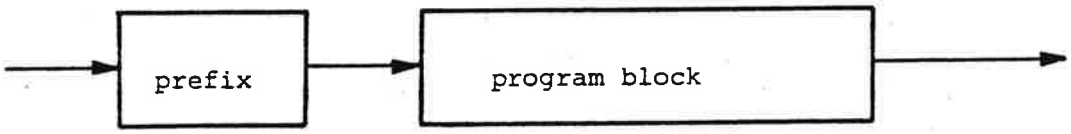
program block



prefix



program



4. Differences between CR80 PASCAL and JW PASCAL

The term JW PASCAL will be used for the language defined in Jensen, Kathleen & Niklaus Wirth: PASCAL User Manual and Report. Second edition. Springer-Verlag. 1978.

Letters and special symbols:

The underscore character _ is a letter in CR80 PASCAL.

The character ∂ is used in CR80 PASCAL instead of the character ↑ in JW PASCAL.

In CR80 PASCAL the brackets [and] can also be written (. and .) respectively.

Word symbols:

The word symbols of JW PASCAL FILE, GOTO, LABEL, PACKED and NIL are not word symbols in CR80 PASCAL.

The word symbols of CR80 PASCAL FORWARD and UNIV are not word symbols of JW PASCAL.

Prefix:

The notion of a prefix is not known in JW PASCAL.

Comments:

CR80 PASCAL: Uses the character " to begin and to end a comment.

JW PASCAL: Uses { to begin a comment and } to end a comment.

Labels and label definition part:

There are no labels and no label definition part in CR80 PASCAL.

Constant and type definitions:

Constant and type definitions can be intermixed and appear any number of times in CR80 PASCAL, whereas JW PASCAL requires the constant definitions (if any) to appear before the type definitions (if any).

Constants in constant definitions must be unsigned in CR80 PASCAL.

It is not allowed in CR80 PASCAL to define an enumerated type within a record type definition.

CR80 PASCAL requires a variant record to contain a tag field, and the ordinal value of the case labels in the variant part must be contained in the closed interval from 0 to 15.

A record type definition in CR80 PASCAL cannot have a semicolon immediately before the final END of the definition.

Standard types:

The type REAL of JW PASCAL is replaced by LONG_INTEGER in CR80 PASCAL. The type FILE is not an inherent type in CR80 PASCAL.

Procedures and functions:

Procedures and functions cannot be nested in CR80 PASCAL (i.e. it is not possible to declare a routine within another routine).

Assignment to formal value parameters is not allowed in CR80 PASCAL.

The UNIV-facility does not exist in JW PASCAL.

Procedures and functions cannot be used as formal parameters in CR80 PASCAL.

Standard procedures and functions:

The only standard procedure in CR80 PASCAL is NEW. The standard functions are LONG, SHORT, ORD, CHR, SUCC and PRED.

Statements:

GOTO-statements are not part of the CR80 PASCAL language.

A case statement in CR80 PASCAL cannot have a semicolon immediately before the final END of the statement.

Sets and set operators:

The ..-notation used in JW PASCAL is not allowed in CR80 PASCAL (e.g. the set ['A', 'B', 'C'] cannot be written ['A'..'C']).

Set union is indicated by OR in CR80 PASCAL and + in JW PASCAL.

Set intersection is indicated by AND in CR80 PASCAL in contrast to * in JW PASCAL.

Comparison operators:

Records: Comparison for equality and inequality between records are allowed in CR80 PASCAL, but not in JW PASCAL.

Arrays: It is possible to compare arrays of other types than CHAR for equality and inequality in CR80 PASCAL.

Integer constants:

The hexadecimal notation possible in CR80 PASCAL is not allowed in JW PASCAL.

Character strings:

JW PASCAL does not relax the type checking for value parameters of the type ARRAY [1..N] OF CHAR.

String elements of the form (:<number>:) are not allowed in JW PASCAL.

Program heading:

The parameter list of a CR80 PASCAL program heading can be empty, whereas JW PASCAL requires at least one formal parameter to be specified.

Scope Rules:

In JW PASCAL, the scope of an identifier is directly related to the block structure. A definition/declaration of an identifier prohibits that identifier from indicating another object throughout the entire procedure.

CR80 PASCAL uses a subtle different rule, called 'one pass scope', in which a definition of an identifier prohibits only subsequent uses of the identifier within the block form indicating an object outside the block.

5. Data Representation in CR80 PASCAL

This chapter provides information which is useful when calculating the size of the needed runtime stack. It is also a prerequisite for using the UNIV-facility (see 3.6.3) and for inserting assembly code into a CR80 PASCAL program by the %CODE compile time directive (see chapter 8).

In the following a word is a CR80 machine word of 16 bits, and a byte is the 8 rightmost or leftmost bits of a word. All addresses shown are word addresses.

INTEGER : Integer variables are represented in 2's complement. They are contained in 1 word. The range is -32768 to 32767.

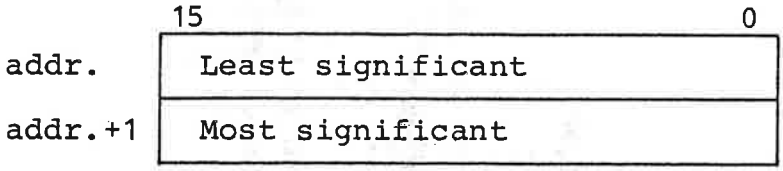
BOOLEAN : A boolean variable is contained in 1 word. The value of the word is either 0 corresponding to FALSE, or 1 corresponding to TRUE.

CHAR : Contained in 1 word. The rightmost byte holds the ASCII value of the character, and the leftmost byte is 0.

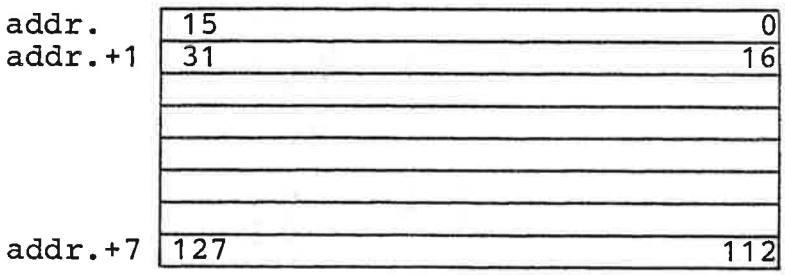
POINTER : Contained in 1 word. The special value NIL is represented as 0. Otherwise it contains a process base relative address of the first word (i.e. the word with the lowest machine word address) of the object pointed at.

ENUMERATED TYPE : A variable of enumerated type is represented in 1 word. In a declaration T = (C0, C1, C2,...., Cn) will C0 correspond to 0, C1 to 1, ... , and Cn to n.

LONG_INTEGER : Long_integer variables are contained in 2 words. The representation is 2's complement, and the range is -2147483648L to 2147483647L.



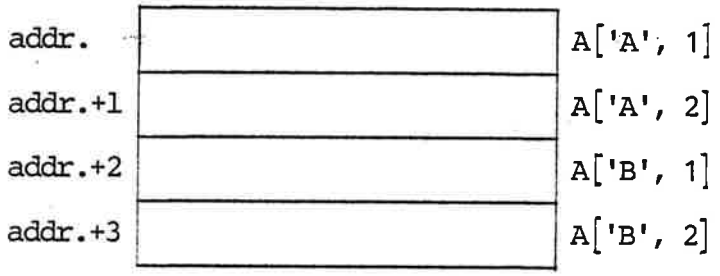
SET : A set variable is always layed out as 8 words:



Member no. n is included in the set if and only if bit no. n is 1.

ARRAY : Arrays are layed out in lexicographical order. Example:

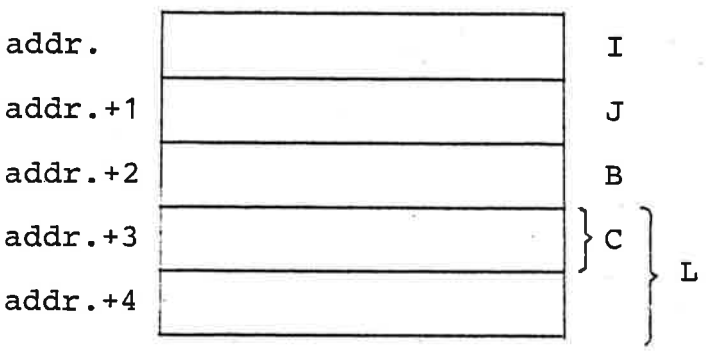
```
A: ARRAY ['A'.. 'B'; 1.. 2]
          OF INTEGER;
```



RECORD : Space is always allocated for the largest possible variant. The first field in the record gets the lowest address, and the last field gets the highest address.

Example:

```
R: RECORD
   I, J: INTEGER;
   CASE B: BOOLEAN OF
     FALSE: (C: CHAR);
     TRUE:  (L: LONG_INTEGER)
   END
```



6. The runtime system: An inner look

The scope of this section is to provide the necessary information to enable a programmer with previous experience in CR80 assembly language programming to insert native CR80 machine code into a CR80 PASCAL program by utilizing the %CODE compile time directive. Assembly code can be inserted to make a monitor function not supported by the prefix accessible, or to minimize the CPU-time used at certain bottle-necks in a program.

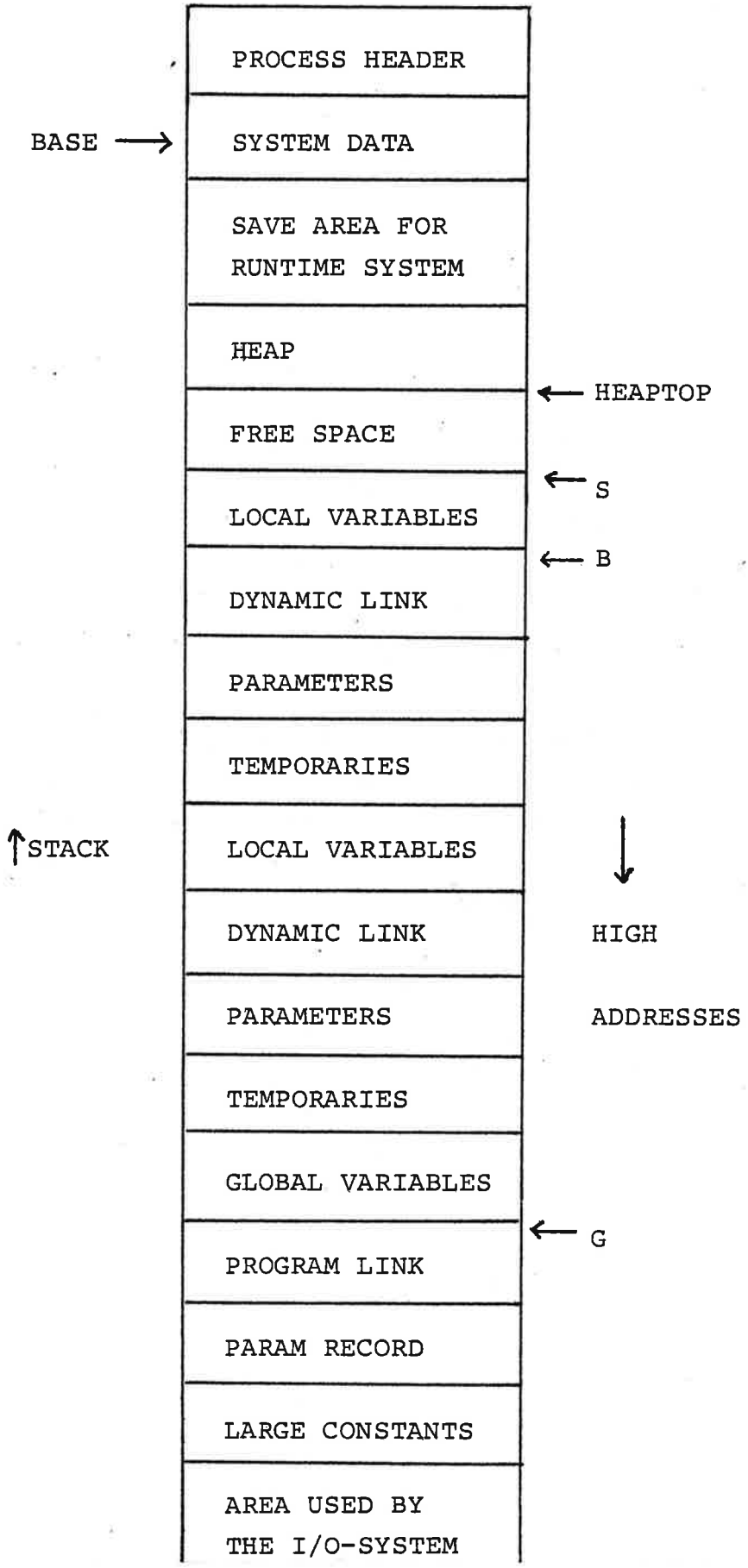
Although insertion of assembly code should not be the province of the ordinary programmer, he might skim this section to gain further insight into the CR80 PASCAL system.

6.1 The runtime stack

When a CR80 PASCAL program is executing, it uses a stack and a heap. The stack contains variables, temporary results, and parameters and return information for procedures and functions. The heap contains variables allocated by the standard procedure NEW.

The following figure illustrates the layout of the data part of an executing CR80 PASCAL program. A procedure has just been called by another procedure, which was called in the program block:

PROCESS:


PROCESS HEADER
BASE
SYSTEM DATA
SAVE AREA FOR RUNTIME SYSTEM
HEAP
HEAPTOP
FREE SPACE
S
LOCAL VARIABLES
B
DYNAMIC LINK
PARAMETERS
TEMPORARIES
STACK
LOCAL VARIABLES
DYNAMIC LINK
HIGH
PARAMETERS
ADDRESSES
TEMPORARIES
GLOBAL VARIABLES
G
PROGRAM LINK
PARAM RECORD
LARGE CONSTANTS
AREA USED BY THE I/O-SYSTEM

6.2 Register Allocation in the Runtime System

The PASCAL runtime system maintains 4 registers:

G: The global base register. R3
is used.

B: The local base register. R6
is used.

Q: The program counter in
the virtual code. R4
is used.

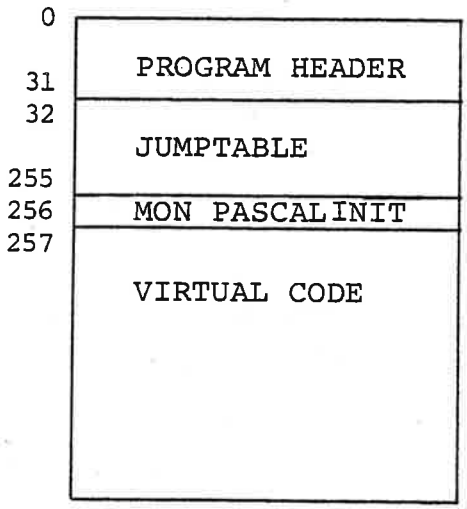
S: The stack top pointer. R5
is used.

A programmer inserting assembly code into a CR80 PASCAL program can use registers R0, R1, R2, R4, and R7 as work registers. He can use R4 because this register is only used to contain the return link when a jump has been performed to the runtime system.

Registers R3, R5, and R6 shall after the execution of the inserted code have the same contents as before the code was executed.

6.3 The Virtual Code

The program part of a CR80 PASCAL program has this format:



The execution of a CR80 PASCAL program starts with a monitor call which initializes the jumptable with the addresses of those subroutines in the runtime system which emulates the virtual instructions. The virtual code consists of JMPI S4 P8 instructions, possibly followed by parameters. Thus although the code is said to be virtual, it is basically CR80 machine code, and this fact makes insertion of "normal" CR80 machine code feasible.

6.4 Addressing and Layout of Variables

All addresses of variables or parameters are normal CR80 process base relative word addresses. The address of a variable or parameter that takes up more than 1 word is the address of the first word, i.e. the word with the lowest address. See also chapter 5 of this document.

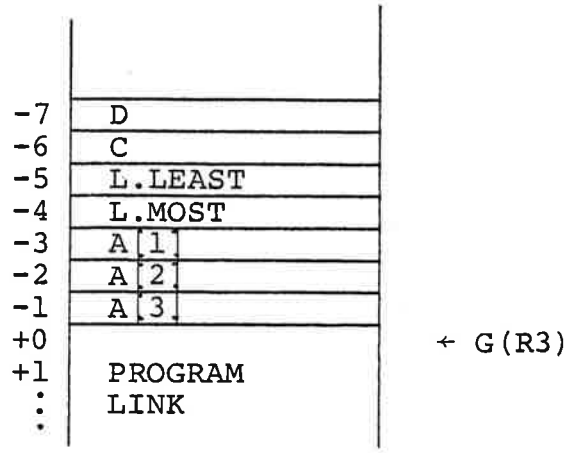
6.4.1 Global Variables

Global variables are allocated in the order in which their declarations are met. They are addressed relative to the global base G (R3) with negative displacements such that the absolute value of the displacement is least for the first declared variable.

Example:

```
PROGRAM P;
VAR
  A:  ARRAY [1..3] OF INTEGER;
  L:  LONG_INTEGER;
  C, D:  CHAR;
BEGIN
  :
  :
END.
```

When the program is executing, we will have this situation:



If the inserted code among other things had to move the variable C to R0, the code might include:

```
MOV        R3     R7
ADDC  -6          R7
MOV   0.   X7     R0
```

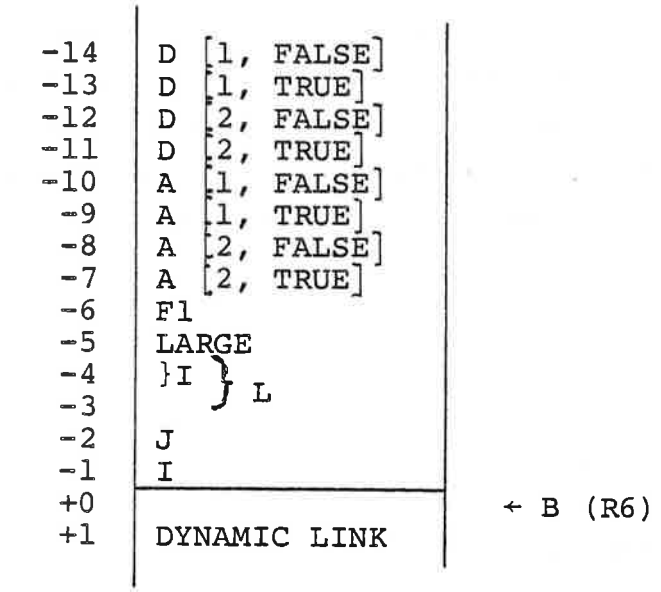
6.4.2 Local Variables

Local variables are dynamically allocated at procedure or function entry and deallocated at exit. They are laid out in the order in which their declarations are met, and are addressed relative to the local base B (R6) with negative displacements, such that the absolute value of the displacement is least for the first declared variable.

Example:

```
PROCEDURE PIP;
VAR
  I, J:  INTEGER;
  REC:  RECORD
          F1:  CHAR;
          CASE LARGE:  BOOLEAN OF
            TRUE:  (L:  LONG_INTEGER);
            FALSE:  (I:  INTEGER)
        END;
  A, D:  ARRAY [1..2, BOOLEAN] OF 1..3;
BEGIN
  "PROCESSING"
END;
```

Within the procedure block we will have this picture:

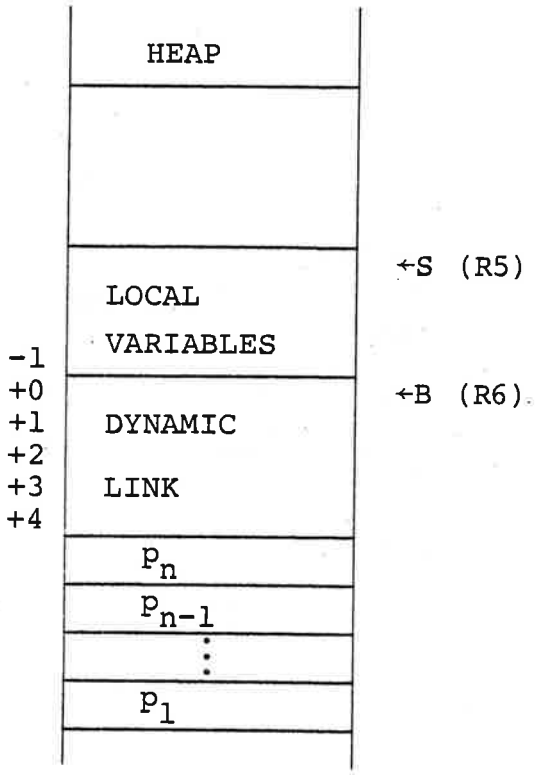


If the inserted code should include a move of the 2 first elements of A to the 2 first elements of D we might have

```
MOV       R6      R7; B
ADDC  -14         R7; -14 => REF D;
MOVL    4.X7     R01;  A[1, *]
MOVL      R01   0.X7; => D [1, *];
```

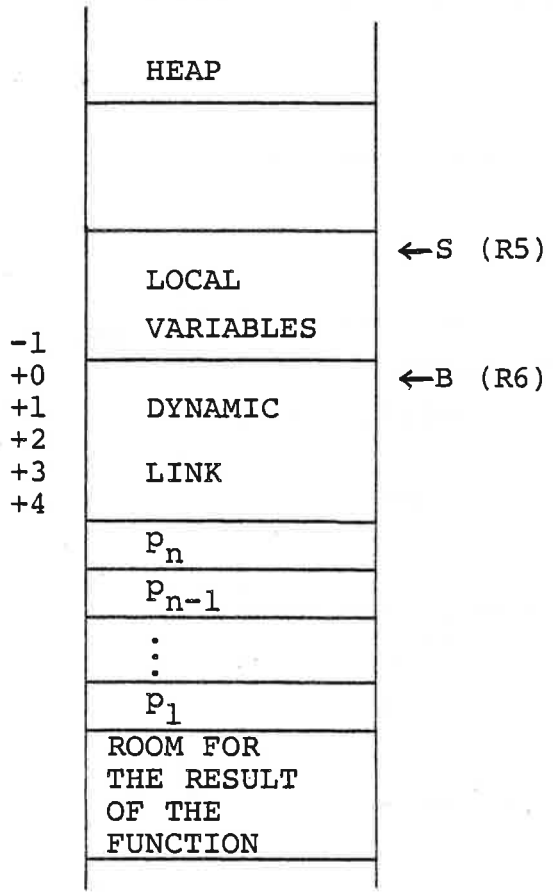
6.4.3 Parameter Passing

Suppose the procedure PAP(parm$_1$, parm$_2$,..., parm$_n$) has just been called. When the procedure is entered, the stack will contain at the point just before the first statement in the procedure block:



The symbol p_i either represents the value of the actual parameter parm$_i$ or the process base relative address of parm$_i$. Each p_i can take up 1, 2 or 8 words.

When a function is called, e.g. by A := PEP ($parm_1$, $parm_2$, ..., $parm_n$), the stack will contain at the point just before the first statement in the function block:



Again each p_i can be in 1, 2 or 8 words. The value of a function is either of simple type or pointer type. Thus the space for the result is either 1 or 2 words (2 for LONG_INTEGER type).

A parameter can be declared as a variable or value parameter. A variable parameter is prefixed by the word VAR and represents a variable to which the routine may assign a value. An actual variable parameter will have its process base relative address pushed on the stack before the routine is entered. Hence, p_i being an address.

A value parameter is not prefixed by the word VAR, and it is not supposed to have its value changed in the routine. Actual value parameters of simple type, LONG_INTEGER type, pointer type and set type will have their value pushed on the stack before the routine is entered. Hence, p_i being a value.

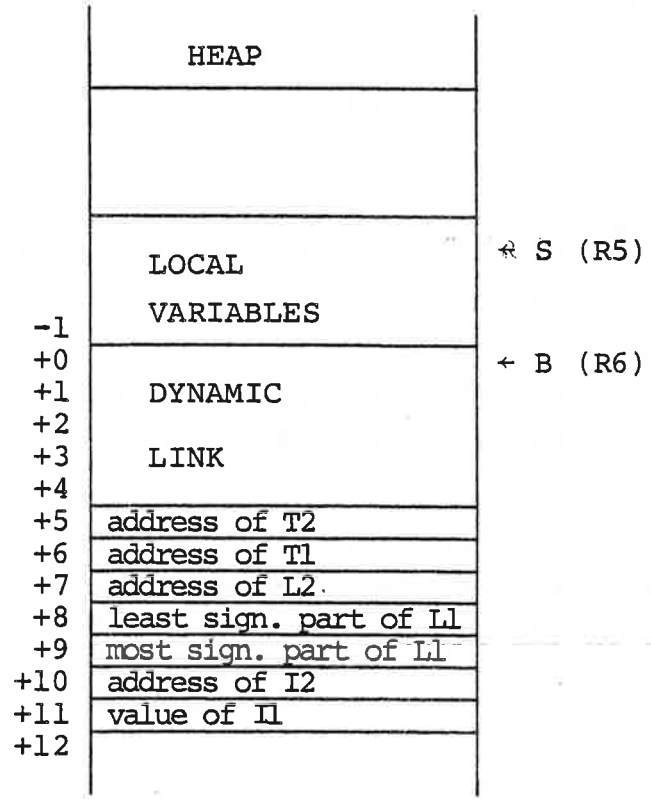
Actual value parameters of array type and record type will always have their process base relative address pushed on the stack.

Example:

```
PROCEDURE P(
  I:          INTEGER;
  VAR J:      INTEGER;
  LONG1:      LONG_INTEGER;
  VAR LONG2:  LONG_INTEGER;
  TXT1:       IDENTIFIER;
  VAR TXT2:   IDENTIFIER);
```

where TYPE IDENTIFIER = ARRAY [1..10] OF CHAR;

The call P(I1, I2, L1, L2, T1, T2) will result in the following stack just before the first statement in the procedure block:



6.5 Work Areas

When assembly code is inserted, a need for work areas (save areas) may arise. The programmer can either declare variables in the CR80 PASCAL program to be used as register save areas (see 6.6), or he can use the space between the stacktop and the heaptop. To get the size of the free area between the stack and the heap, it is necessary to use the merge file @**GENS.D*PASASM defining the address of the memory location where the current value of the heaptop is stored. The label is HEAPTOP, and the following sequence can be used to get the size of the free area into R1:

```
MOV    HEAPTOP     R0 ;
MOV        R5      R1 ;
SUB        R0      R1 ;
```

It should be noted that HEAPTOP contains the address of the first word after the heap, while R5 contains the address of the top stack element.

The merge file @**GENS.D*PASASM also defines a consecutive register save area of 8 words, which can be freely used by the programmer. The layout of the save area is:

```
REGS0:      0 ;
REGS1:      0 ;
  :
REGS7:      0 ;
```

6.6 Inserting Assembly Code

As an example it will be shown how a monitor function can be made accessible in a CR80 PASCAL program.

We will implement:

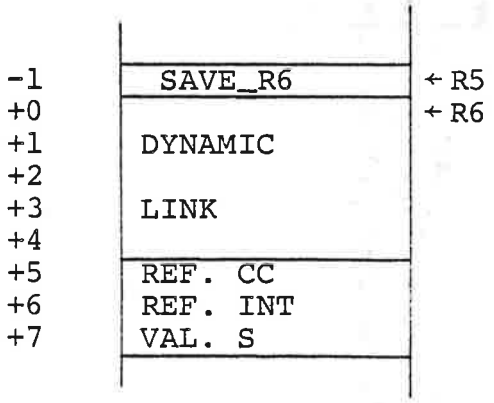
```
PROCEDURE READ_INTEGER(
  S:         STREAM;
  VAR INT:   INTEGER;
  VAR CC:    COMPLETION_CODE);
```

"THIS PROCEDURE IS USED FOR READING INTEGERS.
IT CALLS THE MONITOR FUNCTION STREAM, INELEMENT
(REF. 3) AND SKIPS ANYTHING ENCOUNTERED WHICH
IS NOT AN INTEGER. HOWEVER, IF A NUMBER OUTSIDE
THE INTERVAL -32768..32767 IS READ, THE PROCEDURE
WILL RETURN WITH CC INDICATING 'ELEMENT OVERFLOW'"

Assuming that the binary assembly code is contained in the file @**READ_INT, the input to the CR80 PASCAL compiler will have this outline:

```
"PREFIX"
"CONST AND TYPE DECLARATIONS"
"VARIABLE DECLARATIONS"
"PROCEDURE DECLARATIONS"
PROCEDURE READ_INTEGER(
  S:            STREAM;
  VAR INT:      INTEGER;
  VAR CC:       COMPLETION_CODE);
VAR SAVE_R6:    INTEGER;
BEGIN
%CODE =@**READ_INT
END;
"MORE PROCEDURE DECLARATIONS"
BEGIN
  "PROCESSING"
END.
```

At the point where the code is inserted, the picture is:



The input source file to be merged and then assembled out into @**READ_INT will contain:

```
LIST   "IMPLEMENTATION OF THE PROCEDURE READ_INTEGER"
BEGIN MODULE
XDATA:=     FALSE;
XPROGRAM:= FALSE;
NOLIST
$@**GENS.D*AMOSG1;  USED TO DEFINE STREAM AND INELEMENT;
LIST
USE BASE
SAVER6:=    0;
REFCC:=     SAVER6+6;
REFINT:=    REFCC+1;
SVALUE:=    REFINT+1;
USE PROG
         MOV           R5           R0 ;   SAVE(S);
         MOV           R3           R2 ;   SAVE(G);
         MOV           R6   SAVER6.X5  ;   B => SAVE_R6;
         MOV   SVALUE.X5            R4 ;   S => STREAM;
REP:                                    ; REP:
         MOVC  2                    R1 ;   2 => BYTE_COUNT;
         MOV   REFINT.X5            R6 ;   REF.INT => ADDR;
         MON   STREAM, INELEMENT       ;   INELEMENT(
         JMP   &             IERROR    ;    ERROR:      GOTO IERROR;
         JMP   &             NUMBER    ;    NUMBER:     GOTO NUMBER;
         JMP   &             IDENTI    ;    IDENTIFIER:GOTO IDENTI;
         JMP   &             SPECIA    ;    SPECIAL:    GOTO SPECIA);
IDENTI:                                 ; IDENTI:
SPECIA:                                 ; SPECIA:
         MOV           R0           R5 ;   RESTORE(S);
         JMP                 REP       ;   GOTO REP;
NUMBER:                                 ; NUMBER:
IERROR:                                 ; IERROR:
         MOV           R0           R5 ;   RESTORE(S);
         MOV           R2           R3 ;   RESTORE(G);
         MOV   SAVER6.X5            R6 ;   RESTORE(B);
         MOV   REFCC. X5            R4 ;   REF.CC;
         MOV           R7           X4 ;   COMPLCODE => CC;
; SEE?
END
```

7. The AMOS standard prefix

This chapter describes the AMOS standard prefix. The prefix, which is listed in appendix A, contains a number of type and constant definitions and a list of 133 assembly coded procedures and functions. These prefix routines are included in the PASCAL runtime system and are directly available to the programmer.

Most of the data types are introduced to mirror data structures in the AMOS kernel, the I/O system or the file system. These data types should be regarded as a sort of intrinsic data types. E.g. it should be ignored that a variable of type FILE actually is nothing but an integer variable.

A few of the data types in the prefix may need a little explanation:

1)
```
TYPE ELEMENT =
  ARRAY [1..1]  OF INTEGER;
```

This type is e.g. used in the prefix routine OUTREC:

```
PROCEDURE OUTREC(
  S: STREAM;
  FIRST_ELEMENT: UNIV ELEMENT;
  VAR RECORD_LENGTH_IN_BYTES: INTEGER;
  VAR CC: COMPLETION_CODE);
```

The ideal would have been a procedure with the following outline:

```
PROCEDURE OUTREC(
  S: STREAM;
  REC: ANY TYPE;
  VAR CC: COMPLETION_CODE);
```

This could have been achieved by making OUTREC a standard procedure. Now it is a prefix procedure, and the compiler does all its type checking. The way OUTREC is declared makes it possible to output most variables and subparts of structured variables. A variable declared as

```
A: ARRAY [4711..5001] OF CHAR
```

can be output like this:

```
BYTELENGTH: = (5001-4711+1) *2;
OUTREC(STRM, A [4711], BYTELENGTH, CC);
```

We simply use the fact that the address of any variable is the address of its first "element".

The type ELEMENT is introduced because OUTREC needs the address of the first word to be transferred. If the second formal parameter of OUTREC had been declared FIRST_ELEMENT: UNIV INTEGER, then OUTREC would have received the value of the first word.

2)
```
PACKED_NAME   = ARRAY [0..7] OF INTEGER;
PACKED_NAME2  = ARRAY [0..1] OF INTEGER;
PACKED_NAME3  = ARRAY [0..2] OF INTEGER;
FILE_NAME     = PACKED_NAME;
```

In CR80 PASCAL arrays of CHAR are not packed as in the kernel or file system. This makes it a little inconvenient e.g. to build a file name. Example:

Lookup the file named PASCALCOMPILER in the directory contained in the file variable DIR:

```
BUF = 'PASCALCOMPILER(:0:)(:0:)';
PACK (BUF[1], NAME[0], 16);
LOOKUP (DIR, NAME, F, CC);
```

where

```
  BUF: ARRAY [1..16] OF CHAR;
  NAME: FILE_NAME;
```

and PACK and LOOKUP are prefix routines.

The 4 "packed" types above only contains packed characters when the programmer does the packing, or when he assigns/reads something which is already packed. Because an INTEGER variable and a variable of type CHAR both take up 1 word, PACKED_NAME could also have been an ARRAY [0..7] OF CHAR. However, then the compiler would not have detected an erroneous call such as LOOKUP (DIR, 'OBJECT', F, CC), because of the relaxed type checking in procedure and function calls (see 3.6.3).

3)
```
BUFFER_LOCATION = (LOCAL, EXTERNAL);
BLEPTR = @BLE;
BLE =
  RECORD
    LINK: BLEPTR;
    CASE XL: BUFFER_LOCATION OF
  LOCAL:
      (BUFADDR, BUFSIZE_IN_BYTES: INTEGER);
  EXTERNAL:
      (MEMORY: MEMORY_PARM)
END;
```

These types are used when direct I/O (as opposed to stream I/O) is performed. Their use should be deducible from the following program example:

```
PROCEDURE DIRECT_IO(F: FILE);
"THIS PROCEDURE READS 1016 WORDS FROM F."
"THE FIRST 16 ARE DELIVERED IN HEADER,"
"AND THE LAST 1000 ARE READ INTO AN EXTERNAL BUFFER."

VAR
  HEADER: ARRAY[1..16] OF INTEGER;
  BLE_POINTER: BLEPTR;
  MEM: MEMORY_PARM;
  CC: COMPLETION_CODE;
  TOP, ALLOC: INTEGER;
  WA: WORD_ADDRESS;
  FA: FILE_ADDRESS;
  OK: BOOLEAN;

BEGIN
  GET_BUFFER(1000, MEM, WA, ALLOC, OK); "PREFIX ROUTINE"
  IF NOT OK THEN ERROR; "ERROR DECLARED ELSEWHERE"
  MARK(TOP); "PREFIX ROUTINE"
  NEW(BLE_POINTER);

  WITH BLE_POINTER@ DO
    BEGIN
      XL := LOCAL;
      BUFADDR := REL_ADDR(HEADER[1]); "PREFIX ROUTINE"
      BUFSIZE_IN_BYTES := 16*2;
      NEW(LINK);
      WITH LINK@ DO
        BEGIN
          XL := EXTERNAL;
          MEMORY := MEM;
          LINK := NIL; "LAST IN CHAIN"
        END;
    END;

  WITH FA DO
    BEGIN
      FIRST_BYTE := 0L;
      BYTE_COUNT := (16L + 1000L)*2L;
    END;

  READ_BYTES(F, FA, BLE_POINTER, CC);
  IF (CC <> IO_OK) OR
     (FA.BYTE_COUNT <> FA.TRANSFERRED_BYTES) THEN ERROR;
  "DO THE INTENDED PROCESSING"
  RELEASE_BUFFER(MEM, OK);
  IF NOT OK THEN ERROR;
  RELEASE(TOP);


END "DIRECT_IO";
```

```
4) PARAMTYPE =
     RECORD
       "CURRENT FILE SYSTEM NAME"
       FSN: FILE_SYSTEM_NAME;
       "CURRENT VOLUME NAME"
       VOL: VOLUME_NAME;
       "CURRENT PARAMETER FILE"
       PFILE: FILE;
       "CURRENT DIRECTORY FILE"
       DFILE: FILE;
       "CURRENT INPUT FILE"
       IFILE: FILE;
       "CURRENT OUTPUT FILE"
       OFILE: FILE;
       "PARENT OF PROCESS"
       PARENT: PROCESS_NAME;
       PTR:    POINTER
     END;
```

PASCAL utility programs (i.e. programs not compiled with the %NONUTILITY toggle on) will have their PARAM-record initialized by the runtime system when they are loaded. For nonutility programs the contents of this record are undefined.

More information can be found in ref. 6 under the START command. The use of the PTR-field is explained under the description of the prefix routine RUN.

The rest of this chapter gives a brief description of each prefix procedure and function. The routines are described in order of appearance in the prefix.

Two things should be noted:

1) The outcome of an I/O-procedure is only as indicated when the returned completion code equals IO_OK.

2) Actual variable parameters will most likely have their old contents destroyed, even though the routine call was not successful. For example, if INBYTE is called with a non-connected stream, the CC parameter will indicate this fact, but the second parameter will nevertheless receive some unpredictable garbage "byte".

```
FUNCTION IAND  (
  MASK1,  MASK2 : UNIV INTEGER): INTEGER;
```

The two masks are and'ed logically.

```
FUNCTION IOR (
  MASK1,  MASK2 : UNIV INTEGER): INTEGER;
```

The two masks are or'ed logically.

```
FUNCTION XOR (
  MASK1,  MASK2 : UNIV INTEGER): INTEGER;
```

The two masks are exclusive or'ed logically.

```
FUNCTION INV(
  MASK: UNIV INTEGER): INTEGER;
```

The result is the mask inverted (i.e. 1 bits are changed to 0 bits and 0 bits are changed to 1 bits).

```
FUNCTION LEFTSHIFT(
  BITS: UNIV INTEGER; SHIFTS: INTEGER): INTEGER;
```

The result is BITS shifted logically to the left as indicated by SHIFTS. If SHIFTS >= 16 the result is 0. If SHIFTS <= 0 no shifts are performed.

```
FUNCTION RIGHTSHIFT(
  BITS: UNIV INTEGER; SHIFTS: INTEGER): INTEGER;
```

The result is BITS shifted logically to the right as indicated by SHIFTS. If SHIFTS >= 16 the result is 0. If SHIFTS <= 0 no shifts are performed.

```
FUNCTION ADD(
  A, B: INTEGER): INTEGER;
```

The two integers A and B are added. There is no test for overflow. If A and B were added by using the normal plus operator +, overflow would have caused a runtime error.

```
FUNCTION SUBSTRACT (
  A, B: INTEGER): INTEGER;
```

The integer B is subtracted from A. There is no test for overflow. If B was subtracted from A by using the normal minus operator -, overflow would have caused a runtime error.

```
FUNCTION GETBITS (
  BITS: UNIV INTEGER; LEFTMOST: BITPOSITION;
  FIELDLENGTH: BITFIELDLENGTH): INTEGER;
```

Let M = MINIMUM (LEFTMOST + 1, FIELDLENGTH). GETBITS extracts a bit field from BITS of length M. The bit numbers selected are LEFTMOST, LEFTMOST - 1, ..., LEFTMOST + 1 - M. The result is the extracted bit field right justified possibly (if M < 16) extended to the left with 0-bits. If FIELDLENGTH = 0 the result is 0. If LEFTMOST or FIELDLENGTH are not within range (0 < = LEFTMOST < = 15 ; 0 < = FIELDLENGTH < = 16) a runtime error (range error) will occur.

```
PROCEDURE PUTBITS (
  FROM: UNIV INTEGER; VAR TO_: UNIV INTEGER;
  LEFTTO: BITPOSITION; FIELDLENGTH: BITFIELDLENGTH);
```

Let M = MINIMUM(LEFTTO + 1, FIELDLENGTH). PUTBITS extracts a bit field of M bits from the parameter FROM consisting of the least significant M bits and inserts this bit field from bit number LEFFTO to bit number LEFTTO - M + 1 in the second parameter TO_. The other bits of TO_ are left unchanged. If FIELDLENGTH = 0, TO_ is not changed. If LEFTTO or FIELDLENGTH are not within range (0 < = LEFFTO < = 15; 0 < = FIELDLENGTH < = 16) a runtime error (range error) will occur.

```
FUNCTION TESTBIT(
  BITS: UNIV INTEGER; BITNUMBER: BITPOSITION): BOOLEAN;
```

If bit number BITNUMBER is set (i.e. equal to 1) in BITS, the result is TRUE. Otherwise it is FALSE. A runtime error (range error) will occur, if BITNUMBER is not within range (0 < = BITNUMBER < = 15).

```
PROCEDURE SETBIT(
  VAR BITS: UNIV INTEGER; BITNUMBER: BITPOSITION);
```

The procedure sets to 1 the bit in BITS with number BITNUMBER. All other bits in the parameter BITS are left unchanged. A runtime error will be produced if BITNUMBER is not within range (0 < = BITNUMBER < = 15).

```
PROCEDURE CLEARBIT(
  VAR BITS: UNIV INTEGER; BITNUMBER: BITPOSITON);
```

The procedure clears to 0 the bit in BITS with number BITNUMBER. All other bits in the parameter BITS are left unchanged. A runtime error will be produced if BITNUMBER is not within range (0 < = BITNUMBER < = 15).

```
PROCEDURE SENSE_IO(
  DEVICE:  INTEGER; VAR STATUS: UNIV INTEGER);
```

The 6 least significant bits of DEVICE are the address of an I/O-device, while the remaining 10 bits may be used as a command with a device dependent meaning. The contents of the device control register of the selected device are delivered in the parameter STATUS.

```
PROCEDURE READ_IO(
  DEVICE: INTEGER; VAR DATA: UNIV INTEGER);
```

The 6 least significant bits of DEVICE are the address of an I/O-device, while the remaining 10 bits may be used as a command with a device dependent meaning. The procedure reads a data word from the selected device and delivers it in the parameter DATA.

```
PROCEDURE CONTROL_IO(
  DEVICE: INTEGER; STATUS: UNIV INTEGER);
```

The 6 least significant bits of DEVICE are the address of an I/O-device, while the remaining 10 bits may be used as a command with a device dependent meaning. The control word contained in the parameter STATUS is transferred to the selected device.

```
PROCEDURE WRITE_IO(
  DEVICE: INTEGER; DATA: UNIV INTEGER);
```

The 6 least significant bits of DEVICE are the address of an I/O-device, while the remaining 10 bits may be used as a command with a device dependent meaning. The procedure transfers the word contained in the second parameter DATA to the selected device.

```
PROCEDURE RESERVE_INTERRUPT(
  DEVPR: INTEGER; VAR INTRPT: INTEGER);
```

The monitor function RESERVEINTERRUPT is called. The parameter DEVPR shall contain the concatenation of a 6 bit I/O-device address (in bits 2 to 7) and a priority (in bits 0 to 1). If DEVPR is valid and not reserved by another process, an identification of the interrupt will be returned in the parameter INTRPT. This identification must be used for all other interrupt functions. If the reservation was not successful, INTRPT will contain -1.

```
PROCEDURE RELEASE_INTERRUPT(
  INTRPT: INTEGER);
```

The monitor function RELEASEINTERRUPT is called. The interrupt is released, if INTRPT is an identification of an interrupt reserved by the calling process. Otherwise nothing happens.

```
PROCEDURE CLEAR_INTERRUPT(
  INTRPT: INTEGER);
```

The monitor function CLEARINTERRUPT is called. The interrupt counter is cleared to zero, if INTRPT contains an identification of an interrupt reserved by the calling process.

```
PROCEDURE WAIT_INTERRUPT(
  DELAY, INTRPT: INTEGER; VAR TIMED_OUT: BOOLEAN);
```

First the monitor function SETINTERRUPT is called, followed by a call of the monitor function WAITEVENT waiting for interrupts and time outs. The procedure returns when the interrupt identified by INTRPT (and previously reserved by the process) is or already has been received, or when DELAY*100 ms have elapsed, whatever happens first. At return the parameter TIMED_OUT indicates whether the process was timed out or the interrupt was received.

```
PROCEDURE SET_INTERRUPT(
  INTRPT: INTEGER);
```

The monitor function SETINTERRUPT is called. If INTRPT is an identification of an interrupt reserved by the calling process, this interrupt will be the one waited for when the prefix procedure WAIT_EVENT is called with an event mask specifying that interrrupts are awaited.

```
PROCEDURE SET_CYCLE(
  CYCLE: INTEGER);
```

The monitor function SETCYCLE is called.

```
PROCEDURE SEND_MESSAGE(
   VAR RECEIVER: PROCESS_NAME;
   MSG: UNIV MESSAGE_BUFFER; VAR EVENT: INTEGER);
```

The monitor function SENDMESSAGE is called. The contents of the parameter MSG are copied to a message buffer, the contents of which are delivered to the process identified by RECEIVER, when this process calls the prefix procedure WAIT_EVENT or WAIT_MESSAGE. An identification of the message is delivered in the last parameter EVENT. Note: RECEIVER. NAME must contain the name of the process in packed form (first character in byte 0, second character in byte 1, ...). The first parameter RECEIVER is a variable parameter, because RECEIVER. NAME_IDENT might be updated to allow faster lookup next time this process is referenced.

```
PROCEDURE SEND_SYSTEM_MESSAGE(
  VAR RECEIVER: PROCESS_NAME;
  MSG: UNIV MESSAGE_BUFFER; VAR EVENT: INTEGER);
```

The monitor function SENDSYSTEMMESSAGE is called. Analoguos to SEND_MESSAGE above.

```
PROCEDURE SEND_ANSWER(
  ANS: UNIV MESSAGE_BUFFER; EVENT: INTEGER);
```

The monitor function SENDANSWER is called. The contents of ANS are copied to a message buffer, the contents of which are sent to the originator of the event contained in EVENT.

```
PROCEDURE SEND_SYSTEM_ANSWER(
  ANS: UNIV MESSAGE_BUFFER; EVENT: INTEGER);
```

The monitor function SENDSYSTEMANSWER is called. The contents of ANS are copied to a message buffer, the contents of which are sent to the originator of the event contained in EVENT.

```
PROCEDURE SEND_SIGNAL(
  VAR RECEIVER: PROCESS_NAME);
```

The monitor function SENDSIGNAL is called. The signal boolean in the process identified by RECEIVER is set, and if the receiving process was awaiting the signal, it is linked to its CPU ready queue.

```
PROCEDURE INDENTIFY_SENDER(
  EVENT: INTEGER; VAR PROC: INTEGER; VAR OK: BOOLEAN);
```

The monitor function IDENTIFYSENDER is called. The PCB index of the process originating the EVENT is delivere in PROC. If the EVENT was received by the calling process, OK is set to TRUE, otherwise to FALSE. (The process name of the originator can be found by callin the prefix procedure GET_PROC_NAME).

```
PROCEDURE GET_PROC_NAME(
  VAR PROC_NAME: PROCESS_NAME);
```

The monitor function IDENTIFYPROCESS is called. If a process with PCB index (e.g. returned by IDENTIFY_SENDER) equal to PROC_NAME. NAME_IDENT exists, the name of the process is returned in PROC_NAME. NAME, otherwise a dummy name equal to '??????' is returned.

```
PROCEDURE GET_PROC_IDENT(
  VAR PROC_NAME: PROCESS_NAME; VAR FOUND: BOOLEAN);
```

The monitor function LOOKUPPROCESS is called. If a process with name PROC_NAME. NAME exists, its PCB index is returned in PROC_NAME. NAME_IDENT and FOUND is set to TRUE. Otherwise FOUND is set to FALSE.

```
PROCEDURE WAIT_MESSAGE(
  DELAY: INTEGER; VAR MSG: UNIV MESSAGE_BUFFER;
  VAR EVENT: INTEGER; VAR EVTTYPE: EVENT_TYPE);
```

The monitor function WAITEVENT is called, waiting for messages and time outs. The procedure returns when a message is or already has been received, or when DELAY*100 ms have elapsed, whatever happens first. If a message is received, it is delivered in MSG and an indentification of the message is delivered in EVENT. The last parameter EVTTYPE tells which event actually happened first.

```
PROCEDURE WAIT_SYSTEM_MESSAGE(
  DELAY: INTEGER; VAR MSG: UNIV MESSAGE_BUFFER;
  VAR EVENT: INTEGER; VAR EVTTYPE: EVENT_TYPE);
```

The monitor function WAITEVENT is called, waiting for system messages and time outs. Analogous to WAIT_MESSAGE above.

```
PROCEDURE WAIT_ANSWER(
  DELAY: INTEGER; EVENT; INTEGER;
  VAR ANS: UNIV MESSAGE_BUFFER; VAR EVTTYPE: EVENT_TYPE);
```

The monitor function AWTANSWER is called. The procedure returns when the specific answer corresponding to EVENT is received, or when DELAY*100 ms have elapsed, whatever happens first. If an answer is received, it will be delivered in ANS and EVTTYPE will be equal to ANSWER. Otherwise EVTTYPE will be equal to TIME_OUT.

```
PROCEDURE WAIT_SYSTEM_ANSWER(
  DEALY: INTEGER; EVENT: INTEGER;
  VAR ANS: UNIV MESSAGE_BUFFER; VAR EVTTYPE: EVENT_TYPE);
```

The monitor function AWTSYSTEMANSWER is called. The procedure is analogous to WAIT_ANSWER above.

```
PROCEDURE WAIT_EVENT(
  DELAY: INTEGER; EVTMSK: EVENT_MASK;
  VAR MSG: UNIV MESSAGE_BUFFER;
  VAR EVENT: INTEGER; VAR EVTTYPE: EVENT_TYPE);
```

The monitor function WAITEVENT is called. EVTMSK is a bit mask specifying those eventtypes to be awaited. If none of the eventtypes specified have yet occured, the process is suspended until an occurence. It returns with the most urgent event. The resulting eventtype is delivered in EVTTYPE, and if the eventtype is of message or answer type an identification of the message/answer is delivered in EVENT, and the contents of the message/answer are delivered in MSG. The value of DELAY is only used if time outs are awaited.

```
PROCEDURE SAVE_EVENT(
  EVENT: INTEGER);
```

The monitor function SAVEEVENT is called. The message or answer (ordinary, system, or path) corresponding to EVENT is queued such that it later on can be retrieved by calling the prefix procedure RESTORE_EVENTS followed by a call of WAIT_EVENT, WAIT_MESSAGE, or WAIT_SYSTEM_MESSAGE.

```
PROCEDURE RESTORE_EVENTS(
  EVTTYPE: EVENT_TYPE);
```

The monitor function RECOVEREVENTS is called. If the eventtype specified is of message or answer type (ordinary, system, or path) the corresponding list of saved events is transferred to the front of the corresponding event queue.

```
PROCEDURE TERMINATE(
  CC: COMPLETION_CODE);
```

The monitor function TERMINATE is called. The value of CC should be between 0 and 255, because the upper byte is reserved. The process is suspended with SSTATE = STOPPED, and the value of CC (with bit 15 set by the kernel) and the program source line number of the procedure call are stored in SERROR in the PCB. If the program was compiled with the %NONUMBER toggle on, the line number stored is that of the line containing the first BEGIN of the program or routine block, in which TERMINATE is called. A parent signal is sent to the parent of the calling process. If CC <> 0 a line number trace is written on current output. The trace makes it possible to follow on a procedure/function basis the execution which lead to this severe situation requiring a forced abend.

```
PROCEDURE READ_TIME(
  VAR TIME: DATE_TIME_GROUP);
```

The monitor function READRTC is called. The clock and date are returned. TIME.YEAR will be the actual year (e.g. 1980).

```
PROCEDURE START_PROCESS(
  PROC: INTEGER; VAR ILLEGAL: BOOLEAN);
```

The monitor function STARTPROCESS is called. If PROC is the PCB index of a child process of the calling process and its state is stopped (or to be stopped), ILLEGAL is set to FALSE, and the state of the child is set to preemted. Otherwise ILLEGAL is set to TRUE.

```
PROCEDURE STOP_PROCESS(
  PROC: INTEGER; VAR ILLEGAL: BOOLEAN);
```

The monitor function STOPPROCESS is called. If PROC is the PCB index of a child process of the calling process, ILLEGAL is set to FALSE and the child is stopped. Otherwise ILLEGAL is set to TRUE.

```
PROCEDURE PROCESS_STATUS(
  PROC: INTEGER; VAR ILLEGAL: BOOLEAN;
  VAR PROC_ATTR: PROCESS_ATTRIBUTES);
```

The monitor function GETATTRIBUTES is called. If PROC is a PCB index, ILLEGAL is set to FALSE and the attributes are delivered in PROC_ATTR. Otherwise ILLEGAL is set to TRUE.

```
PROCEDURE REMOVE_PROCESS(
  PROC: INTEGER; VAR ILLEGAL: BOOLEAN);
```

The monitor function REMOVEPROCESS is called. If PROC is the PCB index of a child of the calling process, ILLEGAL is set to FALSE and the process is removed. Otherwise ILLEGAL is set to TRUE. (As can be seen, the memory parameter of the removed process is not returned as in the assembly language version).

```
PROCEDURE GET_NEXT_PROCESS(
  VAR PROC: INTEGER; VAR NONE: BOOLEAN);
```

The monitor function GETCHILD is called. Successive calls of GET_NEXT_PROCESS will step through the circular list of child processes, delivering their PCB index in PROC. If the list is empty, NONE is set to TRUE, otherwise to FALSE.

```
PROCEDURE ADOPT_PROCESS(
  PROC: INTEGER; VAR ILLEGAL: BOOLEAN);
```

The monitor function ADOPTPROCESS is called. If PROC is the PCB index of a child of the calling process, and if the calling process has a parent, the parenthood of the child is transferred to the grandparent of the child and ILLEGAL is set to FALSE. Otherwise ILLEGAL is set to TRUE.

```
PROCEDURE CREATE_PROCESS(
  VAR CB: CREATION_BLOCK; VAR RESULT: INTEGER);
```

The monitor function CREATEPROCESS is called. The completion code (see ref. 2) is delivered in RESULT.

```
PROCEDURE GET_CPU_PARAMETER(
  CPU: INTEGER; PAR: CPUPARAMETER;
  PRIORITY: INTEGER; VAR VAL: INTEGER; VAR OK: BOOLEAN);
```

The monitor function GETCPUPARAMETER is called (see ref 2).

```
PROCEDURE SET_CPU_PARAMETER(
  CPU: INTEGER; PAR: CPUPARAMETER;
  PRIORITY: INTEGER; VAL: INTEGER; VAR OK: BOOLEAN);
```

The monitor function SETCPUPARAMETER is called (see ref 2).

```
PROCEDURE LOOKUP_CPU(
  VAR CPU_NAME: PROCESS_NAME; VAR FOUND: BOOLEAN);
```

The monitor function LOOKUPCPU is called. The CPU identified by CPU_NAME.NAME is looked up. If found the CPUCB index is returned in CPU_NAME.NAME_IDENT, and FOUND is set to TRUE. Otherwise FOUND is set to FALSE.

```
PROCEDURE GET_BUFFER(
  WORD_CLAIM: INTEGER; VAR MEMORY: MEMORY_PARM;
  VAR ADDR: WORD_ADDRESS;
  VAR WORDS_ALLOCATED: INTEGER; VAR OK: BOOLEAN);
```

The monitor function GETBUF is called. The memory manager is asked to allocate a contiguous memory area at least containing WORD_CLAIM words. If the allocation is successful, OK is set to TRUE and 1) an identification of the area is returned in MEMORY, 2) the start address of the area is returned in ADDR such that ADDR.MEMORY_SECTION may be used directly as a PSW value, i.e. the page number is contained in bits 2 and 3, and 3) the number of words actually allocated is returned in WORDS_ALLOCATED. Otherwise OK is set to FALSE.

```
PROCEDURE GET_BUFFER_ADDR(
  MEMORY:MEMORY_PARM; VAR ADDR: WORD_ADDRESS;
  VAR SIZE_IN_WORDS: INTEGER; VAR OK: BOOLEAN);
```

The monitor funciton ADRBUF is called. If MEMORY identifies a memory area belonging to the calling process, the start address of this area is delivered in ADDR (such that ADDR.MEMORY_SECTION may be used as a PSW), the size of the area is delivered in SIZE_IN_WORDS, and OK is set to TRUE. Otherwise OK is set to FALSE.

```
PROCEDURE RELEASE_BUFFER(
  MEMORY: MEMORY_PARM; VAR OK: BOOLEAN);
```

The monitor function RELBUF is called. If the memory area identified by MEMORY belongs to the calling process, the area is returned to the vacant area pool, and OK is set to TRUE. Otherwise OK is set to FALSE.

```
PROCEDURE CREATE(
  FSN: FILE_SYSTEM_NAME; ATTRIBUTES: FILE_ATTRIBUTES;
  VAR F: FILE; VAR CC: COMPLETION_CODE);
```

The monitor function IO, CREATE is called. A new file is created on the file system FSN with the attributes specified. The file is returned through F as an "open" file.

```
PROCEDURE DISMANTLE(
  F: FILE; VAR CC: COMPLETION_CODE);
```

The monitor function IO, DISMANTLE is called. The file F is dismantled.

```
PROCEDURE PROTECT(
  F: FILE; ACCESS: ACCESS_DESCRIPTION;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, PROTECT is called (see ref 3).

```
PROCEDURE RESET(
  F: FILE; VAR CC: COMPLETION_CODE);
```

The monitor function IO, RESET is called. All storage allocated to the file is deallocated.

```
PROCEDURE OFFER(
  F: FILE; USER: USERID; VAR CC: COMPLETION_CODE);
```

The monitor function IO, OFFER is called. The file F is made available for ACCEPTing (see below) to the USER.

```
PROCEDURE ACCEPT(
  FSN: FILE_SYSTEM_NAME;
  VAR F: FILE; VAR CC: COMPLETION_CODE);
```

The monitor function IO, ACCEPT is called. A file, which was previously OFFERed by another user, from the specified file system is delivered in F as an "open" file.

```
PROCEDURE GET_FILE_INFORMATION(
  F: FILE; INF_TYPE: FILE_INFORMATION_TYPE;
  VAR INF: UNIV LONG_INTEGER; VAR CC: COMPLETION_CODE);
```

The monitor function IO, GETFILEINFORMATION is called. The information about the file F specified by INF_TYPE is returned in INF. If the requested information only occupies one word, it is returned in the least significant word of INF, and the most significant word is set to 0.

```
PROCEDURE ASSIGN(
  FSN: FILE_SYSTEM_NAME;
  DESCRIPTION: DEVICE_DESCRIPTION;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, ASSIGN is called. The device identified by DESCRIPTION is included in the set of devices which can be used by the file system, and the device can now be referrred to by DESCRIPTION. DEVICE.

```
PROCEDURE DEASSIGN(
  FSN: FILE_SYSTEM_NAME;
  DEVICE: DEVICE_NAME; VAR CC: COMPLETION_CODE);
```

The monitor function IO, DEASSIGN is called. The DEVICE is deassigned from the specified file system.

```
PROCEDURE MOUNT(
  FSN: FILE_SYSTEM_NAME; DEVICE: DEVICE_NAME;
  VOLUME: VOLUME_NAME; VAR CC: COMPLETION_CODE);
```

The monitor function IO, MOUNT is called. The volume VOLUME is connected to the device DEVICE on the specified file system.

```
PROCEDURE DISMOUNT(
  FSN: FILE_SYSTEM_NAME;
  VOLUME: VOLUME_NAME; VAR CC: COMPLETION_CODE);
```

The monitor function IO, DISMOUNT is called. The volume VOLUME is dismounted from the file system.

```
PROCEDURE FORMAT(
  FSN: FILE_SYSTEM_NAME; DEVICE: DEVICE_NAME;
  VAR SECTORADDR: SECTOR_ADDRESS;
  BLE_POINTER:BLEPTR; VAR CC: COMPLETION_CODE);
```

The monitor function IO, FORMAT is called. The volume on the specified DEVICE is formatted. The BLE_POINTER is a dummy parameter. The sectors to be formatted are selected by SECTORADDR. At return the number of sector actually formatted is delivered in SECTORADDR.TRANSFERRED_SECTORS.

```
PROCEDURE GET_ROOT(
  FSN: FILE_SYSTEM_NAME; VOLUME: VOLUME_NAME;
  VAR ROOT_DIRECTORY: FILE; VAR CC: COMPLETION_CODE);
```

The monitor function IO, GETROOT is called. The root directory of the indicated volume is returned in ROOT_DIRECTORY as an "open" file (e.g. ready to ENTER a file into).

```
PROCEDURE USER_ON(
  FSN: FILE_SYSTEM_NAME;
  USER: USERID; VAR CC: COMPLETION_CODE);
```

The monitor function IO, USERON is called. The USER can now use the file system FSN.

```
PROCEDURE USER_OFF(
  FSN: FILE_SYSTEM_NAME;
  USER: USERID; VAR CC: COMPLETION_CODE);
```

The monitor function IO, USEROFF is called. The USER is logged off the file system FSN.

```
PROCEDURE ENTER(
  DIRECTORY: FILE; SUBJECT: FILE;
  NAME: FILE_NAME; VAR CC: COMPLETION_CODE);
```

The monitor function IO, ENTER is called. The SUBJECT file is entered (or catalogged) in the DIRECTORY file under the specified NAME.

```
PROCEDURE LOOKUP(
  DIRECTORY: FILE; NAME: FILE_NAME;
  VAR F: FILE; VAR CC: COMPLETION_CODE);
```

The monitor function IO, LOOKUP is called. If the returned CC = IO_OK, a file with name NAME was found in DIRECTORY. The file is returned as an "open" file through F.

```
PROCEDURE DESCENT(
  VAR F: FILE; NAME: FILE_NAME;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, DESCENT is called. If the returned CC = IO_OK, a file with name NAME was found in the directory F. The directory F was then "closed", and the file with name NAME is returned as an "open" file through F.

```
PROCEDURE FIND_FILE(
  FROM_ADAM: BOOLEAN; FSN: FILE_SYSTEM_NAME;
  VOLUME: VOLUME_NAME; NAMELIST: NAMELISTTYPE;
  NAME_NO: INTEGER; DIRECTORY: FILE;
  VAR F: FILE; VAR CC: COMPLETION_CODE);
```

(In retrospect: The layout of this procedure is a blunder). The monitor function FINDFILE is called (see ref 5).

```
PROCEDURE RENAME(
  DIRECTORY: FILE; OLDNAME: FILE_NAME;
  NEWNAME: FILE_NAME; VAR CC: COMPLETION_CODE);
```

The monitor function IO, RENAME is called. If the returned CC = IO_OK, the file with name OLDNAME was found in the DIRECTORY and was renamed to the name NEWNAME.

```
PROCEDURE REMOVE(
  DIRECTORY: FILE; NAME: FILE_NAME;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, REMOVE is called. The file with name NAME is no longer catalogged in DIRECTORY.

```
PROCEDURE READ_SECTORS(
  FSN: FILE_SYSTEM_NAME; DEVICE: DEVICE_NAME;
  VAR SECTORADDR: SECTOR_ADDRESS; BLE_POINTER: BLEPTR;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, READSECTORS is called. The sectors specified by SECTORADDR on DEVICE are transferred to the memory area(s) specified by BLE_POINTER. The number of sectors actually transferred is delivered in SECTORADDR.TRANSFERRED_SECTORS.

```
PROCEDURE WRITE_SECTORS(
  FSN: FILE_SYSTEM_NAME; DEVICE: DEVICE_NAME;
  VAR SECTORADDR: SECTOR_ADDRESS; BLE_POINTER: BLEPTR;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, WRITESECTORS is called. The memory area(s) specified by BLE_POINTER is (are) written on the sectors specified by SECTORADDR on DEVICE. The number of sectors actually transferred is delivered in SECTORADDR.TRANSFERRED_SECTORS.

```
PROCEDURE WRITE_AND_PROTECT(
  FSN: FILE_SYSTEM_NAME; DEVICE: DEVICE_NAME;
  VAR SECTORADDR: SECTOR_ADDRESS; BLE_POINTER: BLEPTR;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, WRITEANDPROTECT is called.

```
PROCEDURE WRITE_AND_MARK(
  FSN: FILE_SYSTEM_NAME; DEVICE: DEVICE_NAME;
  VAR SECTORADDR: SECTOR_ADDRESS; BLE_POINTER: BLEPTR;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, WRITEANDMARK is called.

```
PROCEDURE READ_BYTES(
  F: FILE; VAR FILE_ADDR: FILE_ADDRESS;
  BLE_POINTER: BLEPTR; VAR CC: COMPLETION_CODE);
```

The monitor function IO, READBYTES is called. Data is read into the buffer (s) specified by BLE_POINTER from the specified file address. The number n of bytes actually read is returned in FILE_ADDR.TRANSFERRED_BYTES. (n = MINIMUM(FILE_ADDR. BYTE_COUNT, sum of buffer lengths, no. of bytes in the file from FILE_ADDR. FIRST_BYTE and to the end of the file)).

```
PROCEDURE MODIFY_BYTES(
  F: FILE; VAR FILE_ADDR: FILE_ADDRESS;
  BLE_POINTER: BLEPTR; VAR CC: COMPLETION_CODE):
```

The monitor function IO, MODIFYBYTES is called. Data is written from the buffers specified by BLE_POINTER onto the file from the specified file address. The number n of bytes actually written is returned in FILE_ADDR. TRANSFERRED_BYTES. (n = MINIMUM (sum of buffer lengths, FILE_ADDR. BYTE_COUNT)).

```
PROCEDURE APPEND_BYTES(
  F: FILE; VAR FILE_ADDR: FILE_ADDRESS;
  BLE_POINTER: BLEPTR; VAR CC: COMPLETION_CODE);
```

The monitor function IO, APPENDBYTES is called. Analogous to MODIFY_BYTES above, but data is appended (i.e. the value of FILE_ADDR. FIRST_BYTE is irrelevant).

```
PROCEDURE INIT_READ_BYTES(
  F: FILE; VAR FILE_ADDR: FILE_ADDRESS;
  BLE_POINTER: BLEPTR; VAR OPREF: OPERATION_REFERENCE;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, INITREADBYTES is called. Analogous to READ_BYTES, but the transfer is only initiated. The operation may be awaited/tested for completion by calls of WAIT_OPERATION or TEST_OPERATION. An identification of the initiated transfer is delivered in OPREF. Note: Because FILE_ADDR and the buffers are updated after return from the procedure, they should not be used, or implicitly or explicitly deallocated, until the operation is finished.

```
PROCEDURE INIT_MODIFY_BYTES(
  F: FILE; VAR FILE_ADDR: FILE_ADDRESS;
  BLE_POINTER: BLEPTR; VAR OPREF: OPERATION_REFERENCE;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, INITMODIFYBYTES is called. Analogous to MODIFY_BYTES, but the transfer is only initiated. See also INIT_READ_BYTES above.

```
PROCEDURE INIT_APPEND_BYTES(
  F: FILE; VAR FILE_ADDR: FILE_ADDRESS;
  BLE_POINTER: BLEPTR; VAR OPREF: OPERATION_REFERENCE;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, INITAPPENDBYTES is called. Analogous to APPEND_BYTES, but the transfer is only initiated. See also INIT_READ_BYTES.

```
PROCEDURE WAIT_OPERATION(
  OPREF: OPERATION_REFERENCE;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, WAITOPERATION is called. If the returned CC = IO_OK, the operation identified by OPREF is successfully completed. The TRANSFERRED_BYTES field of the FILE_ADDRESS variable used when initiating the operation now contains the number of bytes actually transferred.

```
PROCEDURE TEST_OPERATION(
  OPREF: OPERATION_REFERENCE; VAR FINISHED: BOOLEAN;
  VAR CC: COMPLETION_CODE);
```

The monitor function IO, TESTOPERATION is called. If the returned CC = IO_OK, then FINISHED indicates whether the operation identified by OPREF is finished or not. The difference between WAIT_OPERATION and TEST_OPERATION is that TEST_OPERATION returns immediately, but WAIT_OPERATION returns when the operation is finished.

```
PROCEDURE CANCEL_OPERATION(
  OPREF: OPERATION_REFERENCE; VAR CC: COMPLETION_CODE);
```

The monitor function IO, CANCEL is called. The operation identified by OPREF is cancelled (if it was not already finished).

```
PROCEDURE CONNECT(
  F: FILE; M: MODE; VAR S: STREAM;
  VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, CONNECT is called. The stream S is connected to the file F for either input or output as specified by M.

```
PROCEDURE DISCONNECT(
  S: STREAM; VAR F: FILE; VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, DISCONNECT is called. The stream S is disconnected, and the file to which the stream was connected is returned in F.

```
PROCEDURE GET_POSITION(
  S: STREAM;  VAR POSITION: STREAM_POSITION;
  VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, GETPOSITION is called. The current position on the stream S is returned in POSITION (for later user by SET_POSITION).

```
PROCEDURE SET_POSITION(
  S: STREAM; POSITION: STREAM_POSITION;
  VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, SETPOSITION is called. The current position on the stream S is now as specified by POSITION.

```
PROCEDURE INBYTE(
  S: STREAM; VAR B: UNIV BYTE; VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, INBYTE is called. The next byte on the stream S is delivered in B (the high order byte of B contains 0).

```
PROCEDURE INWORD(
  S: STREAM; VAR WORD: UNIV INTEGER;
  VAR CC: COMPLETION_CODE);
```

The next two bytes on the stream S are delivered in WORD with the first byte in the rightmost byte of WORD.

```
PROCEDURE BACKSPACE(
  S: STREAM; VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, BACKSPACE is called. The effect is, no matter how many times it is called, that INBYTE will deliver the same byte as the last call of INBYTE.

```
PROCEDURE INREC(
  S: STREAM; VAR FIRST_ELEMENT: UNIV ELEMENT;
  VAR RECORD_LENGTH_IN_BYTES: INTEGER;
  VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, INREC is called. The next RECORD_LENGTH_IN_BYTES bytes from the stream S are delivered in the memory locations that start at FIRST_ELEMENT. At return RECORD_LENGTH_IN_BYTES contains the number of bytes actually transferred. This number may be less than requested, if the end of the stream is reached. CC will be IO_OK if any bytes are delivered.

Example: The two fields T and L in the record

```
REC: RECORD
       I: INTEGER;
       T: ARRAY [7..18] OF CHAR;
       L: LONG_INTEGER;
       B: BOOLEAN
     END
```

shall be initialized by a call of INREC:

```
LENGTH: = (18-7+1) *2 + 4;
INREC (STRM, REC. T[7], LENGTH, CC);
```

```
PROCEDURE OUTBYTE(
  S: STREAM; B: UNIV BYTE; VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, OUTBYTE is called. The rightmost byte of B is written to the next position on the stream.

```
PROCEDURE OUTWORD(
  S: STREAM; WORD: UNIV INTEGER;
  VAR CC: COMPLETION_CODE);
```

The contents of the WORD are written on the stream S. The least significant byte is written first.

```
PROCEDURE OUTREC(
  S: STREAM; FIRST_ELEMENT: UNIV ELEMENT;
  VAR RECORD_LENGTH_IN_BYTES: INTEGER;
  VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, OUTREC is called. Analogous to INREC above.

```
PROCEDURE FLUSH(
  S: STREAM; VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, FLUSH is called. The currently buffered data is output to (the file connected to) the stream S. FLUSH is used e.g. in an interactive program when the user is prompted.

```
PROCEDURE INTYPE(
  S: STREAM; VAR CH: CHAR; VAR CH_TYPE: CHAR_TYPE;
  VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, INTYPE is called. The next character (byte) is read from the stream and delivered in CH. CH_TYPE contains the type of the character:

CHARACTER	CHAR_TYPE
SPACE	TSPACE
'0'..'9'	TDIGIT
'A'..'Z', '.', '-', NULL	TLETTER
OTHER CHARACTERS	TOTHER

```
PROCEDURE INELEMENT(
  S: STREAM; VAR ELEM: ELEM_REC;
  VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, INELEMENT is called. The next "element" (i.e. integer, long_integer, identifier or special character) from the stream S is delivered in the variant record ELEM. The declaration of an ELEM_REC is

```
RECORD
  DELIM: CHAR;
  BYTE_COUNT: INTEGER;
  CASE ELEM_TYPE: ELEMENT_TYPE OF
    TINTEGER:          (INT: INTEGER);
    TLONG_INTEGER:     (LINT: LONG_INTEGER);
    TIDENTIFIER:       (NAME: PACKED_NAME);
    TSPECIAL:          (SPEC_CHAR: CHAR)

END;
```

The syntax of the various constructs is

```
<integer> :: = [+|-] <digit> {<digit>} |
               #<hexadigit> {<hexadigit>}
<long_integer>::=<integer> "outside  -32768..32767"
<identifier> :: = <letter> {<letter>| <digit>}
<letter> :: = _ |.| NULL CHAR| A | B | ... | Y | Z
(<digit> and <hexadigit> are as expected).
```

At return the stream is always positioned such that the next byte will be that immediately after the "element" delivered. The contents of the record fields depend on the value of the tag field ELEM_TYPE:

TINTEGER:

DELIM:	Contains the character immediately after the number. This character will be the one delivered if INBYTE is called next.
BYTE_COUNT:	2.
INT:	The integer value.

TLONG_INTEGER:

DELIM:	As for TINTEGER.
BYTE_COUNT:	4.
LINT:	The long_integer value.

TIDENTIFIER:

DELIM:	Contains the character immediately after the identifier. This character will be the one delivered if INBYTE is called next.
BYTE_COUNT:	The number of characters in the identifier. If the identifier is longer than 16 characters, it will be truncated such that only the first 15 an the last character are delivered. In this case BYTE_COUNT will be 16.
NAME:	The identifier (packed). If BYTE_COUNT is less than 16, the last 16 - BYTE_COUNT characters will be null-characters.

TSPECIAL:

DELIM: The special character itself (not the following).

BYTE_COUNT: 1.

SPEC_CHAR: If the "element" in the stream was not an integer, long_integer or identifier, the first character read is delivered in SPEC_CHAR. It should be noted that

1) a semicolon will never be returned. If a semicolon is encountered in the stream, skipping to the next NL-character takes place, and this character is returned.
2) if a space character is returned, the next "element" cannot be a space because INELEMENT skips spaces and only delivers the last in a sequence.

```
PROCEDURE ININTEGER(
  S: STREAM; VAR INT: INTEGER; VAR CC: COMPLETION_CODE);
```

If the returned CC = IO_OK, the stream contained an integer (in the notation specified under INELEMENT), the value of which is delivered in INT. No other characters but spaces are allowed in front of the number.

```
PROCEDURE INLONG_INTEGER(
  S: STREAM; VAR LINT: LONG_INTEGER;
  VAR CC: COMPLETION_CODE);
```

If the returned CC = IO_OK, the stream contained a long_integer (in the notation specified under INELEMENT), the value of which is delivered in LINT. No other characters but spaces are allowed in front of the number.

```
PROCEDURE INNAME(
  S: STREAM; VAR N: PACKED_NAME;
  VAR CC: COMPLETION_CODE);
```

If the returned CC = IO_OK, the stream contained an identifier (in the notation specified under INELEMENT); which is delivered in N in packed form. No other characters but spaces are allowed in front of the identifier. If the identifier read is shorter than 16 characters, the rest of the characters in N will be null-characters. If the identifier was longer than 16 characters, the first 15 and the last are delivered.

```
PROCEDURE INFILEID(
  S: STREAM; VAR FROM_ADAM: BOOLEAN;
  VAR FSN: FILE_SYSTEM_NAME; VAR VOLUME: VOLUME_NAME;
  VAR NAMELIST: NAMELISTTYPE; VAR NAME_NO: INTEGER;
  VAR CC: COMPLETION_CODE);
```

(In retrospect: The layout of this procedure is a blunder). The monitor function INFILEID is called (see ref 5).

```
PROCEDURE OUTTEXT(
  S: STREAM; UNPACKED_TEXT: TEXT;
  VAR CC: COMPLETION_CODE);
```

The UNPACKED_TEXT is packed and then output to the stream (or stated alternatively: the rightmost byte of each CHAR is output). The last character written is the one immediately before the first null-character. Because of the relaxed type checking concerning character arrays as actual parameter, the second parameter only needs to be a one-dimensional array of CHAR or a character string.

```
PROCEDURE OUTSTRING(
  S: STREAM; UNPACKED_TEXT: TEXT;
  NO_OF_CHARS: INTEGER; VAR CC: COMPLETION_CODE);
```

Analogous to OUTTEXT above. However, exactly NO_OF_CHARS characters (null-characters and all) are output.

```
PROCEDURE OUTHEXA(
  S: STREAM; INT: UNIV INTEGER; PAD_CHAR: CHAR;
  VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, OUTHEXA is called. The value in INT is output to the specified stream as 4 hexadecimal characters preceded by the character in PAD_CHAR. However, if this character equals NL, only the 4 hexadecimal characters are output.

```
PROCEDURE OUTINTEGER(
  S: STREAM; INT: UNIV INTEGER;
  FORMAT: UNIV INTEGER; VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, OUTINTEGER is called. The value of INT is output as a decimal number to the specified stream. The format of the number is governed by FORMAT:

BIT 15:	Set:	the number is treated as an unsigned number (0..65535).
	Reset:	the number is treated as a normal signed integer (-32768..32767).
BIT 14-8:		Field with. If the number cannot be accomodated in the field, the field is expanded.

BIT 7-0: Padding character. If the field is longer than needed to contain the number, the number is right justified padded to the left with this character.

The sign is only printed for negative numbers.

```
PROCEDURE OUTLONG_INTEGER(
  S: STREAM; LINT: UNIV LONG_INTEGER;
  FORMAT: UNIV INTEGER; VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, OUTLONG INTEGER is called. Analogous to OUTINTEGER above.

```
PROCEDURE OUTNL(
  S: STREAM; VAR CC: COMPLETION_CODE);
```

The monitor function STREAM, OUTNL is called. A NL-character is output to the specified stream.

```
PROCEDURE MARK(
  VAR TOP: INTEGER);
```

Returns in TOP information to be used by the prefix procedure RELEASE in recollecting storage in the heap allocated by subsequent calls of the standard procedure NEW.

```
PROCEDURE RELEASE(
  TOP: INTEGER);
```

Releases storage allocated in the heap by the standard procedure NEW since the call of the prefix procedure MARK which returned the value of TOP.

FUNCTION FREE_SPACE: INTEGER;

The stack (which contains global and local variables) and the heap (which contains variables allocated by the standard procedure NEW) grow towards each other. The function FREE_SPACE delivers the number of unused words between the stack and the heap.

FUNCTION CONTENTS(
BASE_REL_ADDR: LONG_INTEGER): INTEGER;

Delivers the contents of the memory location with the indicated process base relative address. The address is taken modulo 64K.

PROCEDURE EXIT;

This is a very useful procedure because of the lack of GOTO-statements in CR80 PASCAL. When the procedure is called in the program block, the program terminates as if the last END. had been reached. When EXIT is called in a procedure or function, the execution of the routine is terminated as if the last END in the routine had been reached.

PROCEDURE CURRENT_LEVEL(
VAR LEVEL: INTEGER);

Returns in LEVEL information to be used by the prefix procedure LONG_EXIT.

```
PROCEDURE LONG_EXIT(
  LEVEL: INTEGER);
```

This procedure is perhaps best introduced by an example. Suppose we have the following sequence

```
.
.
CURRENT_LEVEL(LEVEL);
A : = A + B;
REPEAT
  READ_COMMAND (OK);
  IF NOT OK THEN ERROR;
UNTIL OK;
EXECUTE_COMMAND;
FOR I : = 1 TO 4711 DO
.
.
```

and suppose READ_COMMAND calls other routines which in turn may call other routines, and so on. Then a call LONG_EXIT (LEVEL) in READ_COMMAND or any of the routines reached from READ_COMMAND will force the execution to continue with the IF-statement immediately after the call of READ_COMMAND (assuming, of course, that LEVEL has not been changed). A call LONG_EXIT (LEVEL) in EXECUTE_COMMAND or any routine reached from EXECUTE_COMMAND will force execution to continue with the FOR-statement. Generally speaking LONG_EXIT (LEVEL) performs a sequence of EXIT calls until the routine or program block in which LEVEL was initialized by a call of CURRENT_LEVEL is reached. A run-time error (range error) occurs if LEVEL does not contain the value of an active "level".

```
FUNCTION CURRENT_LINE: INTEGER;
```

Returns the program source line number in which it is called. However, if the program was compiled with the %NONUMBER toggle on, the line number delivered is that of the line containing the first BEGIN of the (program or routine) block in which the function is called.

```
FUNCTION REL_ADDR(
  FIRST_ELEMENT: UNIV ELEMENT): INTEGER;
```

Returns the process base relative address of the parameter. The function is primarily intended for use when setting up BLE's specifying local buffers.

```
PROCEDURE GET_ABS_ADDR(
  FIRST_ELEMENT: UNIV ELEMENT;
  VAR WORD_ADDR: WORD_ADDRESS);
```

The absolute address of the first parameter is delivered. WORD_ADDR.MEMORY_SECTION contains in bits 3-2 the page number, and the 3 leftmost bits are all ones (i.e. the word can be used directly as a PSW). WORD_ADDR.WORD_DISPLACEMENT contains the word address within the page.

```
PROCEDURE COPY(
  SOURCE, DEST: BYTE_ADDRESS; NO_OF_BYTES: INTEGER);
```

This procedure can be used for inter page copying. The number of bytes specified by the last parameter are copied from the source to the destination. Only non-negative BYTE_DISPLACEMENTs should be used in the two BYTE_ADDRESSes.

```
PROCEDURE PACK(
  FIRST_ELEMENT_OF_UNPACKED: UNIV ELEMENT;
  VAR FIRST_ELEMENT_OF_PACKED: UNIV ELEMENT;
  NO_OF_BYTES: INTEGER);
```

The rightmost bytes (0, 2, 4, ...) of UNPACKED are packed into PACKED like this:

```
    FOR I := 0 TO NO_OF_BYTES - 1 DO
      PACKED.BYTE [I] := UNPACKED.BYTE [2 * I];
```

Example:

```
A, B: ARRAY [1..5] OF CHAR;
```

Old contents:

A:	'A'	'B'	'C'	'D'	'E'
B:	'F'	'G'	'H'	'I'	'K' 'J'

```
PACK (A [3] , B [4] , 3);
```

New contents (A unchanged):

B:	'F'	'G'	'H'	D 'C'	K 'E'

```
PROCEDURE UNPACK(
  FIRST_ELEMENT_OF_PACKED: UNIV ELEMENT;
  VAR FIRST_ELEMENT_OF_UNPACKED: UNIV ELEMENT;
  NO_OF_BYTES: INTEGER);
```

The bytes in PACKED are unpacked into UNPACKED like this:

```
    FOR I := NO_OF_BYTES - 1 DOWNTO 0 DO
      BEGIN
       UNPACKED.BYTE [2 * I] := PACKED.BYTE [I];
       UNPACKED.BYTE [2 * I + 1] := 0;
      END;
```

```
PROCEDURE PACK_SWAPPED(
  FIRST_ELEMENT_OF_UNPACKED: UNIV ELEMENT;
  VAR FIRST_ELEMENT_OF_PACKED: UNIV ELEMENT;
  NO_OF_BYTES: INTEGER);
```

The bytes of UNPACKED are packed into PACKED like this:

```
   FOR I :=  0 TO NO_OF_BYTES - 1 DO
    BEGIN
     IF I MOD 2 = 0 THEN J :=  I + 1 ELSE J :=  I - 1;
     PACKED. BYTE [J] :=  UNPACKED. BYTE [2 * I];
    END;
```

```
PROCEDURE UNPACK_SWAPPED(
  FIRST_ELEMENT_OF_PACKED: UNIV ELEMENT;
  VAR FIRST_ELEMENT_OF_UNPACKED: UNIV ELEMENT;
  NO_OF_BYTES: INTEGER);
```

The bytes of PACKED are unpacked into UNPACKED like this:

```
  FOR I :=  0 TO NO_OF_BYTES - 1 DO
   BEGIN
    IF I MOD 2 = 0 THEN J :=  I + 1 ELSE J :=  I - 1;
    UNPACKED. BYTE [2 * I] :=  PACKED. BYTE [J];
    UNPACKED. BYTE [2 * I + 1] := 0;
   END;
```

```
PROCEDURE RUN(
  F: FILE;  VAR PARAM: PARAMTYPE; VAR LINE: INTEGER;
  VAR RESULT: PROGRESULT);
```

This procedure makes it possible from a CR80 PASCAL program to execute another CR80 PASCAL program - almost as if the called program was a procedure. The operations of RUN are:

1) Create a temporary file and save the calling program. Only the program code is saved; its variables are still in memory.

2) Load the new program from the file F into the locations which previously held the calling program. The new program's program code requirement must not be larger then the requirement of the initially loaded program. (Small programs that call RUN can adjust their size by the %OVERLAY-directive).

3) Give the loaded program access to the PARAM-record, and start its execution.

4) When the loaded program terminates, the caller is reloaded and continues execution.

A program and the program it RUNs exchange information through the PARAM-type record and the heap. In the standard prefix the PTR field in a PARAMTYPE record points to an integer. However, because a pointer is always contained in one word, the PTR field could just as well e.g. be edited to be a pointer to a record containing a number of pointers (and other fields, too). Any conceiveable data structure can thus be made common to a program and the program it RUNs, the only restriction being that the data structure must be contained in the heap.

The local and global variables of a RUNned program are deallocated when the program returns to the caller. But the variables allocated in the heap continue to exist, because the program might have linked these new variables to the common data structure. If the program that calls RUN has no interest in variables allocated in the heap by the RUNned program, it should surround RUN by calls of MARK and RELEASE.

It is allowed for a RUNned program to call RUN.

At return the last parameter RESULT should be tested. A value different from TERMINATED indicates an error. The parameter LINE contains the number of the last program source line executed in the called program.

```
FUNCTION CREATE_LONG(
  LEAST, MOST: UNIV INTEGER): LONG_INTEGER;
```

A long_integer value is created by concatenating the two parameter values.

```
PROCEDURE SPLIT_LONG(
  L: LONG_INTEGER; VAR LEAST, MOST: UNIV INTEGER);
```

The least significant word of L is delivered in LEAST, and the most significant word of L is delivered in MOST.

```
PROCEDURE ASSIGNBITS(
  VALUE: UNIV BITVALUE; VAR P: UNIV PAGE;
  FIRSTBIT, NO_OF_BITS: INTEGER);
```

This rather special procedure was tailored to the file system. The bits in P are numbered 0, 1, 2, ..., 16*256-1 from right to left. ASSIGNBITS puts VALUE in bitnumber FIRSTBIT to bitnumber FIRSTBIT+NO_OF_BITS-1 of P. All other bits in P are left unchanged.

```
PROCEDURE SKIPBITS(
  VALUE: UNIV BITVALUE; P: UNIV PAGE;
  VAR FIRSTBIT: INTEGER; NO_OF_BITS: INTEGER;
  VAR BITSSKIPPED: INTEGER);
```

This rather special procedure was tailored to the file system. The bits in P are numbered 0, 1, 2, ..., 16*256-1 from right to left. SKIPBITS searches for a bit in P with value VALUE. The search starts at bitnumber FIRSTBIT and upto NO_OF_BITS are investigated. At return FIRSTBIT is the bitnumber of the first matching bit, and BITSSKIPPED is the number of bits skipped until the match. When there is no match, BITSSKIPPED will equal NO_OF_BITS at return.

```
PROCEDURE SET_TRACE(
  S: STREAM; MASK: INTEGER);
```

In extreme debugging situations this procedure may be helpful, because it can provide a trace of the program execution. It can also be used to find the optimal value in the %STACK directive. A call of SET_TRACE with a MASK different from 0 will slow execution down with a factor 3.

MASK:

BIT 1: When set a line is output every time a user-declared routine is called. The information printed contains the line no. of the entered routine, the line no. of the call, and the value or process base relative address of each parameter.

BIT 2: When set a line is output every time a prefix routine is called. The information printed indicates which routine is called (they are numbered 0, 1, ... in order of appearance in the prefix) and from where it was called.

BIT 3: When set a line of information is output every time a user-declared routine is exited.

BIT 6: When set the minimum number of free words between the stack and the heap during the rest of the execution will be printed on current output when the program terminates.

See also PRINT_TRACE below.

```
PROCEDURE PRINT_TRACE(
  ON:BOOLEAN);
```

The first time SET_TRACE is called with one or more of bits 1, 2 or 3 set, output will be written on the trace stream until PRINT_TRACE is called with ON equal to FALSE. Trace output is resumed when PRINT_TRACE is called with ON equal to TRUE.

8. Compile Time Directives

All directives to the compiler begin with a %-character, which can be placed in any character position on the line. The characters from the end of the directive and until the first NL-character are skipped, i.e. this field can be used for a comment without enclosing the comment between "-characters. Syntactically directives are equivalent to a single NL-character. They must appear before the final END. in the program source.

Excepting %LIST, %NOLIST, and %CODE, directives have a global influence and can be placed in any line with the same effect. If a 'global' directive is encountered more than once, the last occurrence applies.

Some of the directives include a <number> or a <name>:

```
<number>   ::=  <digit>   {<digit>} |
                #<hexa>    {<hexa>}
<name>     ::=  <letter> {<letter>|<digit>}
<digit>    ::=   0|1|2|3|4|5|6|7|8|9
<hexa>     ::=   <digit>|A|B|C|D|E|F
<letter>   ::=   A|B|C|...Y|Z|_
```

Negative numbers in a directive can only be written in the hexa-decimal notation.

If a name in a directive is longer than 6 characters only the first 5 and the last character are read.

The following directives are implemented:

8.1 %LIST and %NOLIST

By default the source text is listed on the print file. If the %NOLIST directive is used, the source text will not be printed until a %LIST directive is encountered. The change in listing state is effective in the line immediately after the directive.

8.2 %NUMBER and %NONUMBER

The compiler generates for every line in the program block and every line in the routine blocks a special NEWLINE-instruction, when the program is compiled with the %NUMBER toggle. This makes it possible for the PASCAL runtime system to specify exactly which program line was executing, when a runtime error occurred. This feature is vital in program testing. However, the program will take up more memory space (usually about one third) and run slower. By default NEWLINE-instructions are generated. %NONUMBER will tell the compiler not to generate these.

8.3 %CHECK and %NOCHECK

By default the compiler generates special runtime checks:

- Range checks of actual procedure or function value parameters of enumerated type, subrange type, BOOLEAN type, and CHAR type. The checks are done at the point of call.

- pointer checks to ensure that NIL-valued pointers are not used as references.

- Variant checks to ensure that only currently defined variant fields in a record are referenced.

The code generated will also initialize global variables at program entry and local variables at routine entry to contain only 0-bits.

Runtime checks will not be generated, and initialization will not take place, if %NOCHECK is used.

8.4 %SUMMARY and %NOSUMMARY

Some statistics (compiler release, size of program part, directive values used) on the object program will by default be written on the print file after the source listing. The %NOSUMMARY will tell the compiler not to generate the summary.

8.5 %STACK = <number>

The number specifies how many words of memory the program needs for its variables in the stack (and the heap) at runtime. The default memory claim is 2048 words. The total process size of a CR80 PASCAL program cannot exceed 64K. If the stack claim is so large that this limit will be violated, the claim will be adjusted by the compiler such that the process size will be exactly 64K. The directive %WORKAREA = <number> has precisely the same effect as %STACK= <number>.

8.6 %OVERLAY = <number>

The directive enlarges the program part of a CR80 PASCAL object program with a so-called overlay area of <number> words. It is only relevant to create an overlay area for a program that uses the prefix routine RUN. When a program calls RUN, the size of the program part of the loaded program must not be greater than the size of the calling program, and it is therefore necessary to create an overlay area, when a program RUNs a program with a larger program part. The default overlay area size is 0.

8.7 %REENTRANT and %NONREENTRANT

The object program is marked reentrant (the default) or nonreentrant. A program that calls the prefix procedure RUN should be marked nonreentrant.

8.8 %UTILITY and %NONUTILITY

By default an object program is marked as being a utility program. A utility program will be loaded by the CMI (ref. 6) and will have its PARAM record initialized by the PASCAL runtime system. Programs to be loaded otherwise, for example as part of a boot module, should use the %NONUTILITY toggle.

8.9 %CODE = <file-id>

The file-id shall be written in the format specified in ref. 6. The total contents of the file - hopefully machine or virtual PASCAL code - are inserted at this point in the object code, and the virtual location counter is incremented by the number of words in the file. More information about the use of %CODE can be found in chapter 6 of this document.

8.10 %UNIVCHECK and %NOUNIVCHECK

The word UNIV in front of the type identifier in a formal parameter section suppresses compatibility checking. However, the formal and the actual parameter must take up the same number of machine words, and none of them may contain or be a pointer. These two restrictions are removed if the %NOUNIVCHECK option is used. The default is %UNIVCHECK.

8.11 %CODESTATISTICS and %NOCODESTATISTICS

If %CODESTATISTICS is used, an area of 226 words is layed out in the process part of the program. This area is intended for counting the number of times each virtual instruction is executed. At present, however, the PASCAL runtime system does not make any use of this area. The default is %NOCODESTATISTICS.

8.12 %LINESTATISTICS and %NOLINESTATISTICS

If %LINESTATISTICS is used, an area is layed out in the process part of the program. This area is intended for counting the number of times each line of the program is executed in order to provide a runtime profile of the program. At present, however, the PASCAL runtime system does not make any use of this area. The default is %NOLINESTATISTICS.

8.13 %PROGRAMNAME = <name>

The name is inserted in the program name field of the program header. Default: 6 NULL-characters.

8.14 %PROCESSNAME = <name>

The name is inserted in the process name field of the process header. Default: 6 NULL-characters.

8.15 %CPUNAME = <name>

The name is inserted in the CPU name field of the process header. Default: 6 NULL-characters.

8.16 %VERSION = <number>

The number is inserted in the version field of the program header. Default: 0.

8.17 %PRIORITY = <number>

The number is inserted in the priority field of the process header. Default: 1.

8.18 %CAPABILITIES = <number>

The number is inserted in the capability requirement field of the process header. Default: 0.

8.19 %FDS = <number>

The number of file descriptions used by the program. Default: 4. The number of FDS should equal the maximum number of simultaneously 'open' files during the program execution.

8.20 %STREAMS = <number>

The specified number should equal the maximum number of streams simultaneously connected during the program execution. Default: 2.

8.21 %IOCBS = <number>

The number of I/O control blocks required by the program. There should be 2 IOCBS for each input stream and 1 for each output stream plus one for each outstanding direct I/O request. Default: 4.

8.22 %TLES = <number>

The number of transfer list elements used by the program. Usually 3*IOCBS. Default: 12.

8.23 %MESSAGES = <number>

The number of message buffers required by the program. At least 1 + IOCBS. Default: 5.

8.24 %USERID0 = <number>
%USERID1 = <number>

The 2 numbers are inserted in the user id field of the process header with that after %USERID0 in the word with the lowest address. Both values default to 0.

8.25 %EXECLEVEL = <number>

The number is inserted in the execution level field of the process header. Programs doing I/O need an execution level of 2. Default: 0.

9. The CR80 PASCAL Compiler

The compiler consists of 9 separate programs:

1) PASCAL
   Reads the parameters and calls

2) SPASCA.OBJECT
   This is the pass driver that invokes the 7 passes of the compiler one by one.

3) - 9) The passes of the compiler are named
   SPASS1.OBJECT,
   SPASS2.OBJECT,
   :
   SPASS7.OBJECT.

9.1 Activating the Compiler

The syntax of a call of the CR80 PASCAL compiler is as follows:

PASCAL {<file parameter>}{/<control letter>}

where

<file parameter> ::= <source file> |
<object file> |
<print file>

<source file> ::= $\{\begin{matrix}S\\I\end{matrix}\}_1^1$: <file id>
<object file> ::= O : <file id>
<print file> ::= P : <file id>
<control letter> ::= L|N|T

For the syntax of a file id, please see ref. 6.
If a file id is not a complete description of a file, the file is searched relative to the current directory.

source file : If this parameter is not specified, the compiler will use the current input file as source file. The user will be prompted, when he has to enter the program text.

object file : The compiler generates the object program into this file. If the parameter is not present, the compiler will use (and create when non-existent) the file PASCAL.OBJECT in the current directory. Otherwise the file must exist beforehand.

print file : If this parameter is not specified, the compiler will use the current output file as the print file. Otherwise the indicated file must exist beforehand.

The old contents, if any, of the object file and the print file are deleted.

The meaning of the control letters are:

L: Ignore %LIST and %NOLIST directives and list the whole source.

N: Ignore %LIST and %NOLIST directives and do not list the source.

T: Generate test output for compiler maintenance purposes.

9.2 Preparing the Program Source

The compiler is able to take input from any number of source files. When it encounters a $-sign (ASCII 36) as the first character on a line, it either expects a < character (less than character) or a file id. Anything else will terminate the compilation:

- A < character is read:

  The rest of the line is skipped, and the compiler begins reading from the current input file. If this file is the terminal, the user will be prompted. When the end of the file is met, the compiler returns to the line in the old source file just after the line that contained $< as the first 2 characters.

- A file id is read:

  The rest of the line is skipped, and the compiler begins to read from the indicated file. When the end of this file is reached, the compiler resumes reading in the old source file just after the line that contained the $file id.

This file merge may continue to a level of 3.

9.3 Example:

```
COPY I:EXAMPLE
P00014 LOADED

FM: P00014
$DIRECTIVES    "CONTAINED IN CDIR"
$<
%NOLIST
$D↔↔GENS.D↔PREFIX        "THE STANDARD PREFIX"
%LIST
$<

FM: PHO
P00014 TERMINATED, RESULT= #8000 AT LINE    885      189 CPU MSECS USED

:PASCAL I:EXAMPLE O:OBJECT P:P
P00014 LOADED

FM: P00014
PASS 1 IS EXECUTING.
PLEASE ENTER THE PROGRAM TEXT
"THIS IS ONLY A TEST"              (LAST 2 CHARACTERS: NL and EM)
PLEASE PROCEED
$BEGINEND        "FILE IN CDIR"
PASS 2 IS EXECUTING.
PASS 3 IS EXECUTING.
PASS 4 IS EXECUTING.
PASS 5 IS EXECUTING.
PASS 6 IS EXECUTING.
PASS 7 IS EXECUTING.
COMPILATION SUCCESSFUL

FM: PHO
P00014 TERMINATED, RESULT= #8000 AT LINE   1106    52647 CPU MSECS USED

:COPY I:P
P00014 LOADED

FM: P00014

↔↔↔
↔↔↔ CR80 PASCAL COMPILER (VERSION: 80/01/23)
↔↔↔ COMPILATION STARTED 80/06/02 AT 19:44
↔↔↔ SOURCE FILE: @DMA000-FILE↔CRP0001↔MD↔UTILITY.D↔PASCAL.D↔EXAMPLE
↔↔↔ OBJECT FILE: @DMA000-FILE↔CRP0001↔MD↔UTILITY.D↔PASCAL.D↔OBJECT
↔↔↔


0001 %NOSUMMARY
0002 "THIS IS ONLY A TEST"
0003 %NOLIST
0534 BEGIN
0535 END.

FM: PHO
P00014 TERMINATED, RESULT= #8000 AT LINE    885      188 CPU MSECS USED

:
```

10. Runtime Error Codes

When a CR80 PASCAL program terminates, the CMI (ref. 6) will indicate a completion code, and which program line was executed last. However, if the program was compiled with the %NONUMBER toggle, the line number indicated will be the number of the line containing the first BEGIN of the (program or routine) block executed last.

If the completion code is different from # 8000, the runtime system will print a dynamic line number trace on the current output. This requires that the runtime system can connect the current output file to a free stream. The programmer should therefore always specify one stream more than he actually uses himself in the %STREAMS directive to be sure getting this trace.

List of completion codes generated by the runtime system:

#8000: OK

#8701: Arithmetic overflow.

#8702: Pointer error. A NIL-valued pointer was used when referencing a variable.

#8703: Range error. For example when indexing an array out of bounds.

#8704: Variant error. A field of a variant other than the current variant was referenced in a record.

#8705: Heap limit. A call of the standard procedure NEW was unsuccessful due to lack of memory. Recompile with a larger value in the %STACK directive.

#8706: Stack limit. Not enough memory to the runtime stack. Recompile with a larger value in the %STACK directive.

#8720: Mismatch. The program might not run successfully under the current runtime system. Recompile.

#8721: Trace error. Something went wrong after a call of the prefix routine SET_TRACE. The 'line number' printed by the CMI will be the completion code received from the I/O system indicating the nature of the error.

#8722: Not a PASCAL program. A non-PASCAL program has called the monitor function PASCALINIT.

#8723: Initialization error. During program initialization the runtime system received a completion code <> IO_OK. This completion code is returned as the line number.

#8724: I/O error in prefix routine RUN. The completion from the I/O system is returned as the line number.

When a CR80 PASCAL program terminates with a completion code not contained in the above list, the completion code has been generated by the prefix procedure TERMINATE.

APPENDIX A. LISTING OF CR80 PASCAL STANDARD PREFIX.

```
 1: "CR80 PASCAL STANDARD PREFIX. PHØ-800522"
 2: "########################################"
 3: "**********    A M O S     **********"
 4: "########################################"
 5:
 6: CONST NL = '(:10:)';  FF = '(:12:)';  CR = '(:13:)';  EM = '(:25:)';
 7: CONST NULL = '(:0:)';  SP = ' ';
 8:
 9: CONST LINELENGTH = 132;
10: TYPE LINE = ARRAY [1..LINELENGTH] OF CHAR;
11: TYPE TEXT = LINE;
12:
13: TYPE PROGRESULT = (TERMINATED, OVERFLOW, POINTERERROR,
14:                    RANGEERROR, VARIANTERROR, HEAPLIMIT,
15:                    STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERROR);
16:
17: TYPE BITPOSITION = 0..15;
18: TYPE BITFIELDLENGTH = 0..16;
19: TYPE BITVALUE = (LOW, HIGH);
20:
21: TYPE MESSAGE_BUFFER = ARRAY [1..5] OF INTEGER;
22:
23: TYPE WORD_ADDRESS = RECORD
24:                        MEMORY_SECTION: INTEGER;
25:                        WORD_DISPLACEMENT:INTEGER
26:                      END;
27:
28: TYPE BYTE_ADDRESS = RECORD
29:                        BYTE_DISPLACEMENT: INTEGER;
30:                        WORD_ADDR: WORD_ADDRESS
31:                      END;
32:
33: TYPE FILE = INTEGER;
34: TYPE COMPLETION_CODE = INTEGER;
35: TYPE ELEMENT = ARRAY [1..1] OF INTEGER;
36: TYPE PACKED_NAME = ARRAY [0..7] OF INTEGER;
37: TYPE PACKED_NAME3 = ARRAY [0..2] OF INTEGER;
38: TYPE PACKED_NAME2 = ARRAY [0..1] OF INTEGER;
39:
40: TYPE PROCESS_NAME = RECORD
41:                         NAME: PACKED_NAME3;
42:                         NAME_IDENT: INTEGER
43:                       END;
44:
45: TYPE FILE_SYSTEM_NAME = RECORD
46:                            PNAME: PROCESS_NAME;
47:                            GNAME: PACKED_NAME2
48:                          END;


49: TYPE VOLUME_NAME = PACKED_NAME;
50: CONST DIRECTORY = 10;  CONTIGUOUS = 12;  RANDOM = 14;
51: TYPE FILE_ORGANIZATION = DIRECTORY..RANDOM;
52: TYPE FILE_ATTRIBUTES = RECORD
53:                           VOLUME: VOLUME_NAME;
54:                           ORGANIZATION: FILE_ORGANIZATION;
55:                           ALLOC_SIZE: LONG_INTEGER;
56:                           AREA_SIZE: INTEGER
57:                         END;
58: TYPE USERID = ARRAY [0..1] OF INTEGER;
59: TYPE ACCESS_DESCRIPTION = RECORD
60:                              USER: USERID;
61:                              RIGHTS: ARRAY [0..1] OF INTEGER
62:                            END;
63: TYPE FILE_INFORMATION_TYPE = (F_ORGANIZATION, F_SIZE, F_ALLOCSIZE,
64:                               F_BODYADDR, F_AREASIZE, F_THRESHOLD,
65:                               F_LINKS, F_INBFD, F_BFDNBR);
66: TYPE DEVICE_NAME = PACKED_NAME2;
67: TYPE DEVICE_DESCRIPTION = RECORD
68:                              DEVICE_KIND: INTEGER;
69:                              DEVICE_ADDR: INTEGER;
70:                              UNIT: INTEGER;
71:                              SUBUNIT: INTEGER;
72:                              DEVICE: DEVICE_NAME
73:                            END;
```

```
 74: TYPE FILE_NAME = PACKED_NAME;
 75: TYPE FILE_ADDRESS = RECORD
 76:                        FIRST_BYTE: LONG_INTEGER;
 77:                        BYTE_COUNT: LONG_INTEGER;
 78:                        TRANSFERRED_BYTES: LONG_INTEGER
 79:                      END;
 80: TYPE SECTOR_ADDRESS = RECORD
 81:                          FIRST_SECTOR: LONG_INTEGER;
 82:                          SECTOR_COUNT: LONG_INTEGER;
 83:                          TRANSFERRED_SECTORS: LONG_INTEGER
 84:                        END;
 85: TYPE MODE = (INPUT_MODE, OUTPUT_MODE);
 86: TYPE STREAM = INTEGER;
 87: TYPE STREAM_POSITION = LONG_INTEGER;
 88: TYPE BYTE = 0..255;
 89: TYPE OPERATION_REFERENCE = INTEGER;
 90: TYPE MEMORY_PARM = INTEGER;
 91:
 92: TYPE BUFFER_LOCATION = (LOCAL, EXTERNAL);
 93: TYPE BLEPTR = @BLE;
 94: TYPE BLE = RECORD
 95:               LINK: BLEPTR;
 96:               CASE XL: BUFFER_LOCATION OF
 97:                  LOCAL: (BUFADDR, BUFSIZE_IN_BYTES: INTEGER);
 98:                  EXTERNAL: (MEMORY: MEMORY_PARM)
 99:            END;
100:
101: TYPE EVENT_TYPE = (SIGNAL, MESSAGE, ANSWER, SYSTEM_MESSAGE,
102:                    SYSTEM_ANSWER, PATH_MESSAGE, PATH_ANSWER,
103:                    INTERRUPT, TIME_OUT, PARENT_SIGNAL);
104:
105: TYPE EVENT_MASK = INTEGER;
106:
107: TYPE DATE_TIME_GROUP = RECORD
108:                          YEAR, MONTH, DAY: INTEGER;
109:                          HOUR, MIN, SEC: INTEGER
110:                        END;
111:
112: TYPE PROC_TIME = ARRAY [0..2] OF INTEGER;
113:
114: TYPE PROCESS_ATTRIBUTES = RECORD
115:                             ACCESS_RIGHTS, STATE: INTEGER;
116:                             ERROR_CODE, ERROR_LOC: INTEGER;
117:                             CONSUMED_TIME, CREATION_TIME: PROC_TIME
118:                           END;
119:
120: TYPE CREATION_BLOCK = RECORD
121:                          VNAME: PROCESS_NAME;
122:                          VPROG, VINIT, VMICRO, VCAPAB: INTEGER;
123:                          VCPU, VPRIO, VLEVEL, VBASE: INTEGER;
124:                          VSIZE, VBOUND, VMEMORY, VMSGS: INTEGER;
125:                          VUSER: USERID
126:                        END;
127:
128: TYPE CPUPARAMETER = (VCPUNMB, VINTERRUPTMASK, VSCHEDULERESETCOUNT,
129:                     VSLICESIZE, VELAPSEDTIME, VHWPRIORITY);
130:
131: TYPE CHAR_TYPE = (TSPACE, TDIGIT, TLETTER, TOTHER);
132:
133: CONST PAGELENGTH = 256;
134: TYPE PAGE = ARRAY [1..PAGELENGTH] OF INTEGER;
135:
136: TYPE ELEMENT_TYPE = (TERROR, TINTEGER, TIDENTIFIER,
137:                      TSPECIAL, TLONG_INTEGER);
138:
139: TYPE ELEM_REC = RECORD
140:                    DELIM: CHAR;
141:                    BYTE_COUNT: INTEGER;
142:                    CASE ELEM_TYPE: ELEMENT_TYPE OF
143:                       TINTEGER:      (INT: INTEGER);
144:                       TLONG_INTEGER: (LINT: LONG_INTEGER);
145:                       TIDENTIFIER:   (NAME: PACKED_NAME);
146:                       TSPECIAL:      (SPEC_CHAR: CHAR)
147:                  END;
148:
```

```
149: TYPE POINTER = @INTEGER;
150: TYPE PARAMTYPE = RECORD
151:                     FSN:      FILE_SYSTEM_NAME;  "CURRENT FILE SYSTEM NAME"
152:                     VOL:      VOLUME_NAME;       "CURRENT VOLUME NAME"
153:                     PFILE:    FILE;              "CURRENT PARAMETER FILE"
154:                     DFILE:    FILE;              "CURRENT DIRECTORY FILE"
155:                     IFILE:    FILE;              "CURRENT INPUT FILE"
156:                     OFILE:    FILE;              "CURRENT OUTPUT FILE"
157:                     PARENT:   PROCESS_NAME;      "PARENT OF PROCESS"
158:                     PTR:      POINTER
159:                 END;
160:
161: CONST
162:    IO_OK                      =    0;     EOF                        = #201;
163:    NO_FDS_AVAILABLE           = #202;     ILLEGAL_FD                 = #203;
164:    NO_IOCBS_AVAILABLE         = #204;     ILLEGAL_IOCB               = #205;
165:    NO_STREAMS_AVAILABLE       = #206;     ILLEGAL_STREAM             = #207;
166:    NO_XFELEMS_AVAILABLE       = #208;     ILLEGAL_ADDRESS            = #209;
167:    ILLEGAL_BLE                = #20A;     FILE_NOT_OPEN              = #20B;
168:    DIFFERENT_FILE_SYSTEMS     = #20C;     UNKNOWN_FILE_SYSTEM        = #20D;
169:    ILLEGAL_COMMAND            = #20E;     IO_SYSTEM_ERROR            = #20F;
170:    NOT_ENOUGH_SPACE           = #210;     ILLEGAL_MODE               = #211;
171:    ILLEGAL_MEMORY_PARM        = #212;     NO_BUFFER_SPACE            = #213;
172:    NOT_CONNECTED              = #214;     NOT_OUTPUT_MODE            = #215;
173:    NOT_INPUT_MODE             = #216;     ELEMENT_OVERFLOW           = #217;
174:    SYNTAX_ERROR               = #218;
175:
176:    NONEXISTING_DEVICE         = #400;     ILLEGAL_DEVICE_KIND        = #401;
177:    ILLEGAL_CR80_ADDR          = #402;     DEVICE_NAME_IN_USE         = #403;
178:    ILLEGAL_UNIT               = #404;     ILLEGAL_SUBUNIT            = #405;
179:    WRONG_VOLUME_NAME          = #406;     NONEXISTING_VOLUME         = #407;
180:    VOLUME_MOUNTED             = #408;     DIFFERENT_VOLUMES          = #409;
181:    ILLEGAL_FILE               = #40A;     ILLEGAL_ORGANIZATION       = #40B;
182:    ILLEGAL_ALLOC_SIZE         = #40C;     ILLEGAL_AREA_SIZE          = #40D;
183:    ILLEGAL_RESET              = #40E;     ALLOC_TO_CONTIGUOUS_FILE   = #40F;
184:    FILES_OPEN                 = #410;     NO_FILE_TO_ACCEPT          = #411;
185:    NONEXISTING_USER           = #412;     USER_ALREADY_ACTIVE        = #413;
186:    NO_CONNECTION              = #414;     ILLEGAL_USER               = #415;
187:    ILLEGAL_CALLER             = #416;     OTHER_USERS                = #417;
188:    DISK_COMMAND               = #418;     OUT_OF_RANGE               = #419;
189:    DISK_DRIVER_FAILURE        = #41A;     FILE_FULL                  = #41B;
190:    ACL_FULL                   = #41C;     PROTECTION_FAILURE         = #41D;
191:    NO_ACCESS_RIGHTS           = #41E;     BFD_ERROR                  = #41F;
192:    ILLEGAL_DIRECTORY          = #420;     NAME_EXISTS                = #421;
193:    NONEXISTING_NAME           = #422;     NOT_ALLOCATABLE            = #423;
194:
195: CONST NAMELISTMAXINDEX = 10;
196: TYPE NAMELISTTYPE = ARRAY [1..NAMELISTMAXINDEX] OF PACKED_NAME;
197:
```

```
198: FUNCTION IAND(MASK1, MASK2: UNIV INTEGER): INTEGER;
199: FUNCTION IOR(MASK1, MASK2: UNIV INTEGER): INTEGER;
200: FUNCTION XOR(MASK1, MASK2: UNIV INTEGER): INTEGER;
201: FUNCTION INV(MASK: UNIV INTEGER): INTEGER;
202:
203: FUNCTION LEFTSHIFT(BITS: UNIV INTEGER; SHIFTS: INTEGER): INTEGER;
204: FUNCTION RIGHTSHIFT(BITS: UNIV INTEGER; SHIFTS: INTEGER): INTEGER;
205:
206: FUNCTION ADD(A, B: INTEGER): INTEGER;
207: FUNCTION SUBTRACT(A, B: INTEGER): INTEGER;
208:
209: FUNCTION GETBITS(BITS: UNIV INTEGER; LEFTMOST: BITPOSITION;
210:                  FIELDLENGTH: BITFIELDLENGTH): INTEGER;
211: PROCEDURE PUTBITS(FROM: UNIV INTEGER; VAR TO_: UNIV INTEGER;
212:                   LEFTTO: BITPOSITION; FIELDLENGTH: BITFIELDLENGTH);
213: FUNCTION TESTBIT(BITS: UNIV INTEGER; BITNUMBER: BITPOSITION): BOOLEAN;
214: PROCEDURE SETBIT(VAR BITS: UNIV INTEGER; BITNUMBER: BITPOSITION);
215: PROCEDURE CLEARBIT(VAR BITS: UNIV INTEGER; BITNUMBER: BITPOSITION);
216:
217: PROCEDURE SENSE_IO(DEVICE: INTEGER; VAR STATUS: UNIV INTEGER);
218: PROCEDURE READ_IO(DEVICE: INTEGER; VAR DATA: UNIV INTEGER);
219: PROCEDURE CONTROL_IO(DEVICE: INTEGER; STATUS: UNIV INTEGER);
220: PROCEDURE WRITE_IO(DEVICE: INTEGER; DATA: UNIV INTEGER);
221:
222: PROCEDURE RESERVE_INTERRUPT(DEVPR: INTEGER; VAR INTRPT: INTEGER);
223: PROCEDURE RELEASE_INTERRUPT(INTRPT: INTEGER);
224: PROCEDURE CLEAR_INTERRUPT(INTRPT: INTEGER);
225: PROCEDURE WAIT_INTERRUPT(DELAY, INTRPT: INTEGER; VAR TIMED_OUT: BOOLEAN);
226: PROCEDURE SET_INTERRUPT(INTRPT: INTEGER);
227: PROCEDURE SET_CYCLE(CYCLE: INTEGER);
228:
229: PROCEDURE SEND_MESSAGE(VAR RECEIVER: PROCESS_NAME;
230:                        MSG: UNIV MESSAGE_BUFFER;
231:                        VAR EVENT: INTEGER);
232: PROCEDURE SEND_SYSTEM_MESSAGE(VAR RECEIVER: PROCESS_NAME;
233:                               MSG: UNIV MESSAGE_BUFFER;
234:                               VAR EVENT: INTEGER);
235: PROCEDURE SEND_ANSWER(ANS: UNIV MESSAGE_BUFFER; EVENT: INTEGER);
236: PROCEDURE SEND_SYSTEM_ANSWER(ANS: UNIV MESSAGE_BUFFER; EVENT: INTEGER);
237: PROCEDURE SEND_SIGNAL(VAR RECEIVER: PROCESS_NAME);
238: PROCEDURE IDENTIFY_SENDER(EVENT: INTEGER; VAR PROC: INTEGER; VAR OK: BOOLEAN);
239:
240: PROCEDURE GET_PROC_NAME(VAR PROC_NAME: PROCESS_NAME);
241: PROCEDURE GET_PROC_IDENT(VAR PROC_NAME: PROCESS_NAME; VAR FOUND: BOOLEAN);
242:
243: PROCEDURE WAIT_MESSAGE(DELAY: INTEGER; VAR MSG: UNIV MESSAGE_BUFFER;
244:                        VAR EVENT: INTEGER; VAR EVTTYPE: EVENT_TYPE);


245: PROCEDURE WAIT_SYSTEM_MESSAGE(DELAY: INTEGER;
246:                               VAR MSG: UNIV MESSAGE_BUFFER;
247:                               VAR EVENT: INTEGER;
248:                               VAR EVTTYPE: EVENT_TYPE);
249: PROCEDURE WAIT_ANSWER(DELAY: INTEGER; EVENT: INTEGER;
250:                       VAR ANS: UNIV MESSAGE_BUFFER;
251:                       VAR EVTTYPE: EVENT_TYPE);
252: PROCEDURE WAIT_SYSTEM_ANSWER(DELAY: INTEGER; EVENT: INTEGER;
253:                              VAR ANS: UNIV MESSAGE_BUFFER;
254:                              VAR EVTTYPE: EVENT_TYPE);
255: PROCEDURE WAIT_EVENT(DELAY: INTEGER; EVTMSK: EVENT_MASK;
256:                      VAR MSG: UNIV MESSAGE_BUFFER;
257:                      VAR EVENT: INTEGER;
258:                      VAR EVTTYPE: EVENT_TYPE);
259: PROCEDURE SAVE_EVENT(EVENT: INTEGER);
260: PROCEDURE RESTORE_EVENTS(EVTTYPE: EVENT_TYPE);
261:
262: PROCEDURE TERMINATE(CC: COMPLETION_CODE);
263:
264: PROCEDURE READ_TIME(VAR TIME: DATE_TIME_GROUP);
265:
```

```
266: PROCEDURE START_PROCESS(PROC: INTEGER; VAR ILLEGAL: BOOLEAN);
267: PROCEDURE STOP_PROCESS(PROC: INTEGER; VAR ILLEGAL: BOOLEAN);
268: PROCEDURE PROCESS_STATUS(PROC: INTEGER; VAR ILLEGAL: BOOLEAN;
269:                          VAR PROC_ATTR: PROCESS_ATTRIBUTES);
270: PROCEDURE REMOVE_PROCESS(PROC: INTEGER; VAR ILLEGAL: BOOLEAN);
271: PROCEDURE GET_NEXT_PROCESS(VAR PROC: INTEGER; VAR NONE: BOOLEAN);
272: PROCEDURE ADOPT_PROCESS(PROC: INTEGER; VAR ILLEGAL: BOOLEAN);
273: PROCEDURE CREATE_PROCESS(VAR CB: CREATION_BLOCK; VAR RESULT: INTEGER);
274: PROCEDURE GET_CPU_PARAMETER(CPU: INTEGER; PAR: CPUPARAMETER;
275:                           PRIORITY: INTEGER; VAR VAL: INTEGER;
276:                           VAR OK: BOOLEAN);
277: PROCEDURE SET_CPU_PARAMETER(CPU: INTEGER; PAR: CPUPARAMETER;
278:                           PRIORITY: INTEGER; VAL: INTEGER;
279:                           VAR OK: BOOLEAN);
280: PROCEDURE LOOKUP_CPU(VAR CPU_NAME: PROCESS_NAME; VAR FOUND: BOOLEAN);
281:
282: PROCEDURE GET_BUFFER(WORD_CLAIM: INTEGER;
283:                           VAR MEMORY: MEMORY_PARM;
284:                           VAR ADDR: WORD_ADDRESS;
285:                           VAR WORDS_ALLOCATED: INTEGER;
286:                           VAR OK: BOOLEAN);
287: PROCEDURE GET_BUFFER_ADDR(MEMORY: MEMORY_PARM;
288:                                VAR ADDR: WORD_ADDRESS;
289:                                VAR SIZE_IN_WORDS: INTEGER;
290:                                VAR OK: BOOLEAN);
291: PROCEDURE RELEASE_BUFFER(MEMORY: MEMORY_PARM; VAR OK: BOOLEAN);
292:



293: PROCEDURE CREATE(FSN: FILE_SYSTEM_NAME;
294:                  ATTRIBUTES: FILE_ATTRIBUTES;
295:                  VAR F: FILE;
296:                  VAR CC: COMPLETION_CODE);
297: PROCEDURE DISMANTLE(F: FILE; VAR CC: COMPLETION_CODE);
298: PROCEDURE PROTECT(F: FILE;
299:                   ACCESS: ACCESS_DESCRIPTION;
300:                   VAR CC: COMPLETION_CODE);
301: PROCEDURE RESET(F: FILE; VAR CC: COMPLETION_CODE);
302: PROCEDURE OFFER(F: FILE; USER: USERID; VAR CC: COMPLETION_CODE);
303: PROCEDURE ACCEPT(FSN: FILE_SYSTEM_NAME;
304:                  VAR F: FILE;
305:                  VAR CC: COMPLETION_CODE);
306: PROCEDURE GET_FILE_INFORMATION(F: FILE; INF_TYPE: FILE_INFORMATION_TYPE;
307:                                VAR INF: UNIV LONG_INTEGER;
308:                                VAR CC: COMPLETION_CODE);
309: PROCEDURE ASSIGN(FSN: FILE_SYSTEM_NAME;
310:                  DESCRIPTION: DEVICE_DESCRIPTION;
311:                  VAR CC: COMPLETION_CODE);
312: PROCEDURE DEASSIGN(FSN: FILE_SYSTEM_NAME;
313:                    DEVICE: DEVICE_NAME;
314:                    VAR CC: COMPLETION_CODE);
315: PROCEDURE MOUNT(FSN: FILE_SYSTEM_NAME;
316:                 DEVICE: DEVICE_NAME;
317:                 VOLUME: VOLUME_NAME;
318:                 VAR CC: COMPLETION_CODE);
319: PROCEDURE DISMOUNT(FSN: FILE_SYSTEM_NAME;
320:                    VOLUME: VOLUME_NAME;
321:                    VAR CC: COMPLETION_CODE);
322: PROCEDURE FORMAT(FSN: FILE_SYSTEM_NAME;
323:                  DEVICE: DEVICE_NAME;
324:                  VAR SECTORADDR: SECTOR_ADDRESS;
325:                  BLE_POINTER: BLEPTR;
326:                  VAR CC: COMPLETION_CODE);
327: PROCEDURE GET_ROOT(FSN: FILE_SYSTEM_NAME;
328:                    VOLUME: VOLUME_NAME;
329:                    VAR ROOT_DIRECTORY: FILE;
330:                    VAR CC: COMPLETION_CODE);
331: PROCEDURE USER_ON(FSN: FILE_SYSTEM_NAME;
332:                   USER: USERID;
333:                   VAR CC: COMPLETION_CODE);
334: PROCEDURE USER_OFF(FSN: FILE_SYSTEM_NAME;
335:                    USER: USERID;
336:                    VAR CC: COMPLETION_CODE);
337: PROCEDURE ENTER(DIRECTORY: FILE;
338:                 SUBJECT: FILE;
339:                 NAME: FILE_NAME;
340:                 VAR CC: COMPLETION_CODE);
```

```
341: PROCEDURE LOOKUP(DIRECTORY: FILE;
342:                  NAME: FILE_NAME;
343:                  VAR F: FILE;
344:                  VAR CC: COMPLETION_CODE);
345: PROCEDURE DESCENT(VAR F: FILE;
346:                   NAME: FILE_NAME;
347:                   VAR CC: COMPLETION_CODE);
348: PROCEDURE FIND_FILE(FROM_ADAM: BOOLEAN;
349:                     FSN: FILE_SYSTEM_NAME;
350:                     VOLUME: VOLUME_NAME;
351:                     NAMELIST: NAMELISTTYPE;
352:                     NAME_NO: INTEGER;
353:                     DIRECTORY: FILE;
354:                     VAR F: FILE;
355:                     VAR CC: COMPLETION_CODE);
356: PROCEDURE RENAME(DIRECTORY: FILE;
357:                  OLDNAME: FILE_NAME;
358:                  NEWNAME: FILE_NAME;
359:                  VAR CC: COMPLETION_CODE);
360: PROCEDURE REMOVE(DIRECTORY: FILE;
361:                  NAME: FILE_NAME;
362:                  VAR CC: COMPLETION_CODE);
363: PROCEDURE READ_SECTORS(FSN: FILE_SYSTEM_NAME;
364:                        DEVICE: DEVICE_NAME;
365:                        VAR SECTORADDR: SECTOR_ADDRESS;
366:                        BLE_POINTER: BLEPTR;
367:                        VAR CC: COMPLETION_CODE);
368: PROCEDURE WRITE_SECTORS(FSN: FILE_SYSTEM_NAME;
369:                         DEVICE: DEVICE_NAME;
370:                         VAR SECTORADDR: SECTOR_ADDRESS;
371:                         BLE_POINTER: BLEPTR;
372:                         VAR CC: COMPLETION_CODE);
373: PROCEDURE WRITE_AND_PROTECT(FSN: FILE_SYSTEM_NAME;
374:                             DEVICE: DEVICE_NAME;
375:                             VAR SECTORADDR: SECTOR_ADDRESS;
376:                             BLE_POINTER: BLEPTR;
377:                             VAR CC: COMPLETION_CODE);
378: PROCEDURE WRITE_AND_MARK(FSN: FILE_SYSTEM_NAME;
379:                          DEVICE: DEVICE_NAME;
380:                          VAR SECTORADDR: SECTOR_ADDRESS;
381:                          BLE_POINTER: BLEPTR;
382:                          VAR CC: COMPLETION_CODE);
383: PROCEDURE READ_BYTES(F: FILE;
384:                      VAR FILE_ADDR: FILE_ADDRESS;
385:                      BLE_POINTER: BLEPTR;
386:                      VAR CC: COMPLETION_CODE);
387: PROCEDURE MODIFY_BYTES(F: FILE;
388:                        VAR FILE_ADDR: FILE_ADDRESS;
389:                        BLE_POINTER: BLEPTR;
390:                        VAR CC: COMPLETION_CODE);
391: PROCEDURE APPEND_BYTES(F: FILE;
392:                        VAR FILE_ADDR: FILE_ADDRESS;
393:                        BLE_POINTER: BLEPTR;
394:                        VAR CC: COMPLETION_CODE);
395: PROCEDURE INIT_READ_BYTES(F: FILE;
396:                           VAR FILE_ADDR: FILE_ADDRESS;
397:                           BLE_POINTER: BLEPTR;
398:                           VAR OPREF: OPERATION_REFERENCE;
399:                           VAR CC: COMPLETION_CODE);
400: PROCEDURE INIT_MODIFY_BYTES(F: FILE;
401:                             VAR FILE_ADDR: FILE_ADDRESS;
402:                             BLE_POINTER: BLEPTR;
403:                             VAR OPREF: OPERATION_REFERENCE;
404:                             VAR CC: COMPLETION_CODE);
405: PROCEDURE INIT_APPEND_BYTES(F: FILE;
406:                             VAR FILE_ADDR: FILE_ADDRESS;
407:                             BLE_POINTER: BLEPTR;
408:                             VAR OPREF: OPERATION_REFERENCE;
409:                             VAR CC: COMPLETION_CODE);
410: PROCEDURE WAIT_OPERATION(OPREF: OPERATION_REFERENCE;
411:                          VAR CC: COMPLETION_CODE);
412: PROCEDURE TEST_OPERATION(OPREF: OPERATION_REFERENCE;
413:                          VAR FINISHED: BOOLEAN;
414:                          VAR CC: COMPLETION_CODE);
415: PROCEDURE CANCEL_OPERATION(OPREF: OPERATION_REFERENCE;
416:                            VAR CC: COMPLETION_CODE);
417:
```

```
418: PROCEDURE CONNECT(F: FILE;
419:                   M: MODE;
420:                   VAR S: STREAM;
421:                   VAR CC: COMPLETION_CODE);
422: PROCEDURE DISCONNECT(S: STREAM;
423:                      VAR F: FILE;
424:                      VAR CC: COMPLETION_CODE);
425: PROCEDURE GET_POSITION(S: STREAM;
426:                        VAR POSITION: STREAM_POSITION;
427:                        VAR CC: COMPLETION_CODE);
428: PROCEDURE SET_POSITION(S: STREAM;
429:                        POSITION: STREAM_POSITION;
430:                        VAR CC: COMPLETION_CODE);
431: PROCEDURE INBYTE(S: STREAM; VAR B: UNIV BYTE; VAR CC: COMPLETION_CODE);
432: PROCEDURE INWORD(S: STREAM;
433:                  VAR WORD: UNIV INTEGER;
434:                  VAR CC: COMPLETION_CODE);
435: PROCEDURE BACKSPACE(S: STREAM; VAR CC: COMPLETION_CODE);
436: PROCEDURE INREC(S: STREAM;
437:                 VAR FIRST_ELEMENT: UNIV ELEMENT;
438:                 VAR RECORD_LENGTH_IN_BYTES: INTEGER;
439:                 VAR CC: COMPLETION_CODE);
440: PROCEDURE OUTBYTE(S: STREAM; B: UNIV BYTE; VAR CC: COMPLETION_CODE);
441: PROCEDURE OUTWORD(S: STREAM;
442:                   WORD: UNIV INTEGER;
443:                   VAR CC: COMPLETION_CODE);
444: PROCEDURE OUTREC(S: STREAM;
445:                  FIRST_ELEMENT: UNIV ELEMENT;
446:                  VAR RECORD_LENGTH_IN_BYTES: INTEGER;
447:                  VAR CC: COMPLETION_CODE);
448: PROCEDURE FLUSH(S: STREAM; VAR CC: COMPLETION_CODE);
449: PROCEDURE INTYPE(S: STREAM;
450:                  VAR CH: CHAR;
451:                  VAR CH_TYPE: CHAR_TYPE;
452:                  VAR CC: COMPLETION_CODE);
453: PROCEDURE INELEMENT(S: STREAM;
454:                     VAR ELEM: ELEM_REC;
455:                     VAR CC: COMPLETION_CODE);
456: PROCEDURE ININTEGER(S: STREAM;
457:                     VAR INT: INTEGER;
458:                     VAR CC: COMPLETION_CODE);
459: PROCEDURE INLONG_INTEGER(S: STREAM;
460:                          VAR LINT: LONG_INTEGER;
461:                          VAR CC: COMPLETION_CODE);
462: PROCEDURE INNAME(S: STREAM;
463:                  VAR N: PACKED_NAME;
464:                  VAR CC: COMPLETION_CODE);
465: PROCEDURE INFILEID(S: STREAM;
466:                    VAR FROM_ADAM: BOOLEAN;
467:                    VAR FSN: FILE_SYSTEM_NAME;
468:                    VAR VOLUME: VOLUME_NAME;
469:                    VAR NAMELIST: NAMELISTTYPE;
470:                    VAR NAME_NO: INTEGER;
471:                    VAR CC: COMPLETION_CODE);
```

```
472: PROCEDURE OUTTEXT(S: STREAM;
473:                   UNPACKED_TEXT: TEXT;
474:                   VAR CC: COMPLETION_CODE);
475: PROCEDURE OUTSTRING(S: STREAM;
476:                     UNPACKED_TEXT: TEXT;
477:                     NO_OF_CHARS: INTEGER;
478:                     VAR CC: COMPLETION_CODE);
479: PROCEDURE OUTHEXA(S: STREAM;
480:                   INT: UNIV INTEGER;
481:                   PAD_CHAR: CHAR;
482:                   VAR CC: COMPLETION_CODE);
483: PROCEDURE OUTINTEGER(S: STREAM;
484:                      INT: UNIV INTEGER;
485:                      FORMAT: UNIV INTEGER;
486:                      VAR CC: COMPLETION_CODE);
487: PROCEDURE OUTLONG_INTEGER(S: STREAM;
488:                           LINT: UNIV LONG_INTEGER;
489:                           FORMAT: UNIV INTEGER;
490:                           VAR CC: COMPLETION_CODE);
491: PROCEDURE OUTNL(S: STREAM; VAR CC: COMPLETION_CODE);
492:
493: PROCEDURE MARK(VAR TOP: INTEGER);
494: PROCEDURE RELEASE(TOP: INTEGER);
495: FUNCTION FREE_SPACE: INTEGER;
496: FUNCTION CONTENTS(BASE_REL_ADDR: LONG_INTEGER): INTEGER;
497: PROCEDURE EXIT;
498: PROCEDURE CURRENT_LEVEL(VAR LEVEL: INTEGER);
499: PROCEDURE LONG_EXIT(LEVEL: INTEGER);
500: FUNCTION CURRENT_LINE: INTEGER;
501: FUNCTION REL_ADDR(FIRST_ELEMENT: UNIV ELEMENT): INTEGER;
502: PROCEDURE GET_ABS_ADDR(FIRST_ELEMENT: UNIV ELEMENT;
503:                        VAR WORD_ADDR: WORD_ADDRESS);
504: PROCEDURE COPY(SOURCE, DEST: BYTE_ADDRESS; NO_OF_BYTES: INTEGER);
505: PROCEDURE PACK(FIRST_ELEMENT_OF_UNPACKED: UNIV ELEMENT;
506:                VAR FIRST_ELEMENT_OF_PACKED: UNIV ELEMENT;
507:                NO_OF_BYTES: INTEGER);
508: PROCEDURE UNPACK(FIRST_ELEMENT_OF_PACKED: UNIV ELEMENT;
509:                  VAR FIRST_ELEMENT_OF_UNPACKED: UNIV ELEMENT;
510:                  NO_OF_BYTES: INTEGER);
511: PROCEDURE PACK_SWAPPED(FIRST_ELEMENT_OF_UNPACKED: UNIV ELEMENT;
512:                        VAR FIRST_ELEMENT_OF_PACKED: UNIV ELEMENT;
513:                        NO_OF_BYTES: INTEGER);
514: PROCEDURE UNPACK_SWAPPED(FIRST_ELEMENT_OF_PACKED: UNIV ELEMENT;
515:                          VAR FIRST_ELEMENT_OF_UNPACKED: UNIV ELEMENT;
516:                          NO_OF_BYTES: INTEGER);
517: PROCEDURE RUN(F: FILE; VAR PARAM: PARAMTYPE;
518:               VAR LINE: INTEGER; VAR RESULT: PROGRESULT);
519: FUNCTION CREATE_LONG(LEAST, MOST: UNIV INTEGER): LONG_INTEGER;
520: PROCEDURE SPLIT_LONG(L: LONG_INTEGER; VAR LEAST, MOST: UNIV INTEGER);
521: PROCEDURE ASSIGNBITS(VALUE: UNIV BITVALUE; VAR P: UNIV PAGE;
522:                      FIRSTBIT, NO_OF_BITS: INTEGER);
523: PROCEDURE SKIPBITS(VALUE: UNIV BITVALUE; P: UNIV PAGE;
524:                    VAR FIRSTBIT: INTEGER; NO_OF_BITS: INTEGER;
525:                    VAR BITSSKIPPED: INTEGER);
526: PROCEDURE SET_TRACE(S: STREAM; MASK: INTEGER);
527: PROCEDURE PRINT_TRACE(ON: BOOLEAN);
528:
529: PROGRAM MAIN(VAR PARAM: PARAMTYPE);
```