
RCSL No: 42-i1786

Edition: August, 1981

Author: Jan Bardino

Title:

RC8000 PASCAL, User's Guide

Copyright © 1981, A/S Regnecentralen at 1979
RC Computer A/S
Printed by A/S Regnecentralen at 1979, Copenhagen
Users of this manual are cautioned that the specifications contained herein are subject to change by RC at any time without prior notice. RC is not responsible for typographical or arithmetic errors which may appear in this manual and shall not be responsible for any damages caused by reliance on any of the materials presented.

(132 printed pages)

This manual is a tutorial for the language Pascal as implemented for the RC8000 computer.
The manual contains a complete description of RC8000-Pascal and examples of program constructs.

Abstract:

High level language, Standard Pascal

Keywords:

FOREWORD

First edition: RCSL No 42-i1786.

This manual provides a complete description of the programming language Pascal as implemented for the RC8000 computer.

The manual is directed to those who have previously acquired some familiarity with computer programming, and now wish to get acquainted with the programming language Pascal. The style of the manual is that of a tutorial, i.e. a demonstration of the language features by means of examples.

For a concise ultimate of the language definition ref. [1] or ref. [2] may be used.

Summarizing tables and syntax diagrams are added as appendices.

Jan Bardino

A/S REGNECENTRALEN af 1979, August 1981

1. Introduction

2. Literature Review

3. Methodology

4. Results

5. Discussion

6. Conclusion

7. References

8. Appendix

9. Bibliography

10. Index

11. Glossary

12. Acknowledgements

13. Author's Note

14. Contact Information

15. Declaration of Interest

16. Funding Sources

17. Data Availability

18. Ethical Approval

19. Conflicts of Interest

TABLE OF CONTENTS	PAGE
1. INTRODUCTION	1
2. BASIC DEFINITION	2
2.1 Vocabulary	2
2.2 Program Elements	3
2.2.1 Syntax Diagrams	3
2.2.2 Comments and Separators	4
2.2.3 Identifiers	5
2.2.4 Numbers	6
2.2.5 Real Literal	6
2.2.6 Strings of Characters	7
2.2.7 Boolean Literal	8
3. THE PASCAL LANGUAGE	9
3.1 The Program Outline	9
3.2 The Program Structure	11
3.3 The Declaration Part	13
3.3.1 Labels	14
3.3.2 Constants	14
3.3.3 Types	16
3.3.3.1 Enumeration Types	17
3.3.3.2 Subrange Types	19
3.3.3.3 Structured Types	27
3.3.3.4 Type Compatibility	48
3.3.4 Variables	49
3.3.5 Value Part	50
3.3.6 Routine Declaration	53
3.4 The Statement Part	58
3.4.1 Statements	58
3.4.2 Assignment Statement	59
3.4.2.1 Expressions	60
3.4.3 Goto Statement	63
3.4.4 Repetitive Statements	63
3.4.5 Conditional Statements	66
3.4.6 Procedure Call	69
3.4.7 With Statement	72

<u>TABLE OF CONTENTS (continued)</u>	<u>PAGE</u>
4. DETAILED SCOPE RULES	75
5. PREDEFINED ROUTINES	76
5.1 Standard Procedures	76
5.1.1 File Handling Procedures	76
5.1.2 Dynamic Allocation Procedures	77
5.1.3 Transfer Procedures	77
5.1.4 Date and Time	77
5.1.5 Program Control Procedure	77
5.2 Standard Functions	78
5.2.1 Arithmetic Functions	78
5.2.2 Transfer Functions	79
5.2.3 Ordinal Functions	79
5.2.4 Predicates	80
5.2.5 Processing Time Function	80
5.2.6 Monitor Functions	81
5.2.7 Access to File Processor Parameters	82
5.3 Complete List of Predefined Routines	84
6. COMPILER DIRECTIVES	86
7. CALL OF THE PASCAL COMPILER	89
7.1 How to Compile a Pascal Program	89
8. RUNTIME ENVIRONMENT	91
8.1 The Pascal Process at Runtime	91
8.1.1 Resident Procedures	92
9. ERROR MESSAGES	93
10. SOME PROGRAMMING HINTS AND WARNINGS	95

TABLE OF CONTENTS (continued)	PAGE
-------------------------------	------

APPENDICES:

A. REFERENCES	99
B. RC8000 PASCAL SYNTAX DIAGRAMS	100
C. UTILITY PROGRAMS	106
C.1 Indent	106
C.2 Cross Reference Program	107
C.3 Use of Indent and Cross	109
C.4 Performance Measurement	114
D. ERROR MESSAGES	116
D.1 Error Messages from First Pass	116
D.2 Error Messages from Second Pass	119
D.3 Runtime Error Messages	120
D.3.1 Start Up Errors	120
D.3.2 Errors During Program Execution	120

1. INTRODUCTION

1.

The language Pascal was designed by Professor Niklaus Wirth to satisfy two principal aims.

- 1) To make available a language suitable for teaching programming as a systematic discipline.
- 2) To define a language whose implementations could be both reliable and efficient on then available computers.

A preliminary version was drafted in 1968, and the first compiler became operational in 1970. After some revisions, dictated by two years of experience in the use of the language, a Revised Report was published in 1973.

2. BASIC DEFINITION

2.

Any Pascal program consists of a sequence of Pascal symbols. This chapter defines this set of symbols. The Pascal symbols can be divided into the following classes: reserved symbols; identifiers; literals and separators.

An algorithm can be written as a Pascal program which is divided into two main parts: a declaration part and a statement part. The declaration part defines a number of objects which can be manipulated by the statement part. The data items used in an algorithm are called variables and these are introduced by variable declarations. The values that these data items can assume are defined by type declarations. A number of variables constituting a single entity may be combined into a structured data type. A number of declarations and operations which form a closed entity may be combined into a routine by a procedure or function declaration. The statement part defines the main flow of the algorithm and consists of a sequence of statements.

2.1 Vocabulary

2.1

The basic vocabulary consists of language symbols and user defined symbols. The language symbols are reserved words (key words) and punctuation marks. Throughout this manual reserved symbols will be written in capital letters (e.g. BEGIN). The reserved symbols are all listed below:

AND	END	IN	PACKED	TO
ARRAY	EXTERNAL	LABEL	PASCAL	TYPE
BEGIN	FILE	MOD	PROCEDURE	UNTIL
CASE	FOR	MODULE	PROGRAM	VAR
CONST	FORTRAN	NIL	RANDOM	VALUE
DIV	FORWARD	NOT	RECORD	WHILE
DO	FUNCTION	OF	REPEAT	WITH
DOWNTO	GOTO	OR	SET	
ELSE	IF	OTHERWISE	THEN	

+	-	*	/	"	'	<	>
◇	⇐	⇒	()	[]	↑
=	:=	#	.	,	:	;	..
(*	*)	{	}				

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

It should be noted that the following reserved symbols are not used in the current version: RANDOM, EXTERNAL, FORTRAN, PASCAL and MODULE.

2.2 Program Elements

2.2

2.2.1 Syntax Diagrams

2.2.1

The syntax of the various language constructions is defined by means of syntax diagrams. A syntax diagram is a graphical representation of a syntactical rule, every traversal of such a graph corresponds to a particular application of that rule. Any such traversal must follow the direction indicated by the arrows, i.e. no legal traversal may encounter an arrow pointing in the opposite direction.

The following is an example of a syntax diagram.

While statement:

→ WHILE → expression → DO → statement →

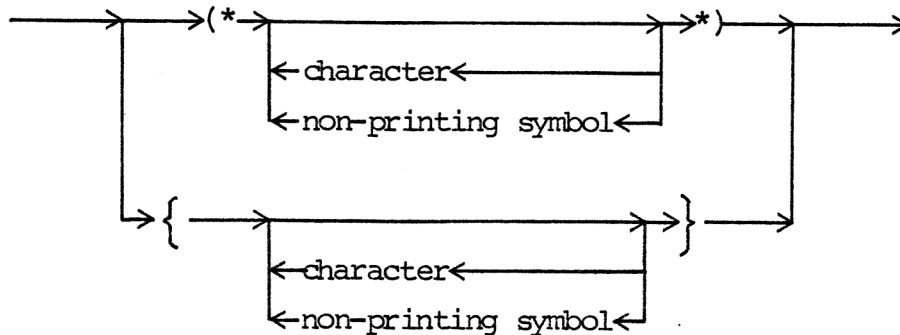
The syntax diagram defines the name (while statement) and syntax of the language constructions. The name is used when the construction is referred to elsewhere in the text or in other syntax diagrams. Language symbols are either names in capital letters (e.g. WHILE) or punctuation marks (e.g. :=).

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

.2 Comments and Separators

2.2.2

Comment:



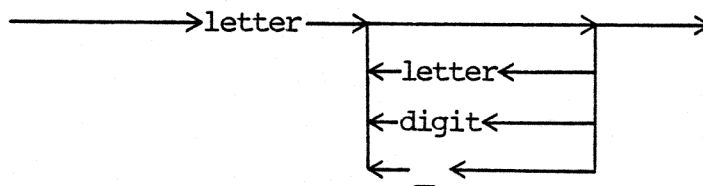
Comments may be inserted between any two identifiers, numbers or special symbols. A comment does not affect the execution of the program.

If the first character after the (*) is a \$ (dollar), the comment is interpreted as a list of compiler options. For a complete description of the available options the reader is referred to chapter 6.

Comments, spaces and ends of lines are considered to be token separators. An arbitrary number of separators are permitted between any two consecutive tokens, or before the first token of a program text. At least one separator is required between any consecutive pair of tokens made up of identifiers, word-symbols or numbers. Apart from the use of the space character in character strings, no separators occur within tokens.

Names denoting constants, types, variables, programs and routines are called identifiers. They must begin with a letter which may be followed by any combination and number of letters, digits and underscores. Identifiers are permitted to be of any length, but only the first twelve are recognized as significant. Matching upper and lower case letters are equivalent in identifiers.

identifier:



letter is A,B,C,...,Z,a,b,c,...,z

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

Examples of legal identifiers:

step use_count Local_Message
 very__special__defined__identifier

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

Whereas none of the following are identifiers.

1a
 __day

The following are some of the predefined identifiers.

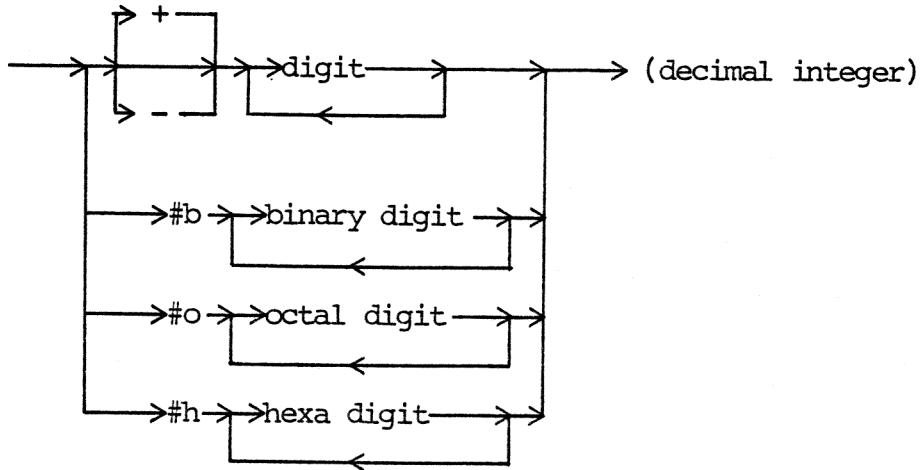
integer
 real
 text
 succ
 false

2.2.4 Numbers

2.2.4

Numbers are integer literals (numeric values) and real literals.

numeric value:



binary digits are 0..1

octal digits are 0..7

hexa digits are 0..9 and a..f

Example of legal numbers:

7913 0033 #b101 #hff00 #o7654

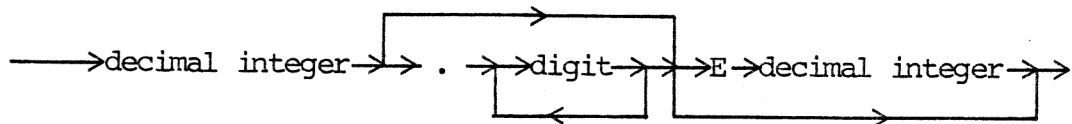
Note: Blanks are not allowed between #b, #o and #h, and the following number.

2.2.5 Real Literal

2.2.5

A real literal is a real number with an optional scale factor.

real literal:



Note that if the real literal contains a decimal point, at least one digit must precede and succeed the point. Also, no comma may occur in a number.

Example of legal real numbers:

3.141592

0.31415E1

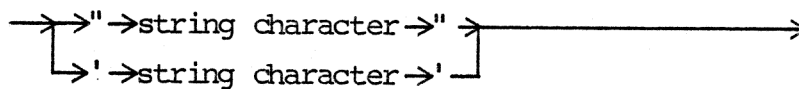
314E-2

2.2.6 Strings of Characters

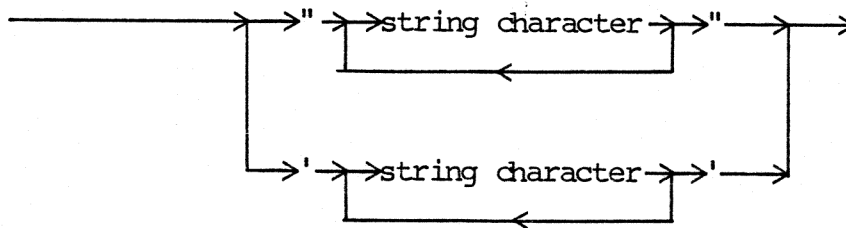
2.2.6

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

char literal:



string literal:



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

Examples of legal strings:

"abcd" " '~' is a strange character" ""

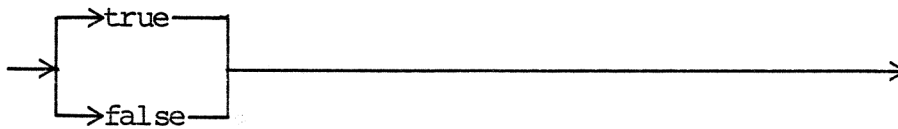
Note: If a string surrounded by single quote marks is to contain a quote mark or a string surrounded by double quote marks is to contain the surrounding quote mark, then this quote mark is to be written twice, for example """" is equivalent to '"', and '''' is equivalent to '"'.

2.2.7 Boolean Literal

2.2.7

A boolean literal is one of the predefined constants true and false.

boolean literal:



3. THE PASCAL LANGUAGE

3.

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

3.1 The Program Outline

3.1

A Pascal program consists of declarations of labels, constants, types, variables, routines, some initializations (VALUE-part) and some statements which operate on the declared objects.

This is an outline of a Pascal program:

```
PROGRAM catalog (output);
  CONST
    idlength = 10;
    catalogsize = 256;
  TYPE
    identifier = ARRAY [1..idlength] OF char;
    .
    .
    .
  VAR
    name: identifier;
    found: boolean;
    index: integer;
  FUNCTION hash (id: identifier): integer;
  VAR
    key, next: integer;
    ch: char;
  VALUE
    key = 1;
    next = 0;
```

```

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

```

The program contains a declaration of

- two constants: idlength with the value 10 and catalogsize with the value 256,
- a type: identifier which is an array of characters,

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

The function has a formal parameter id and three local variables key, next and ch. The value-part specifies initial values for key and next.

The assignment statement: `index := hash (name)` contains a call of the function; the result of the function is assigned to the variable index.

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

3.2 The Program Structure

3.2

The syntax of a Pascal program is

program:

→program heading →block → . →

The program heading specifies the interface to the environment in which the program is executed.

program heading:

→PROGRAM →program identifier →(→file name →) →; →

program identifier:

→identifier →

file name:

→identifier →

→= →external name →

The files denoted by the file names must be declared as file variables in the block of the program, an exception to this is input and output. The files listed in the program heading are called external files. The external name, if present, is an RC8000 catalog entry name in quotes. An external file which has a file specification is automatically opened at the start of the program, as if there had been an explicit `open(file, external name)` (see 3.3.3.3), but there is no automatic call of 'reset' or 'rewrite'

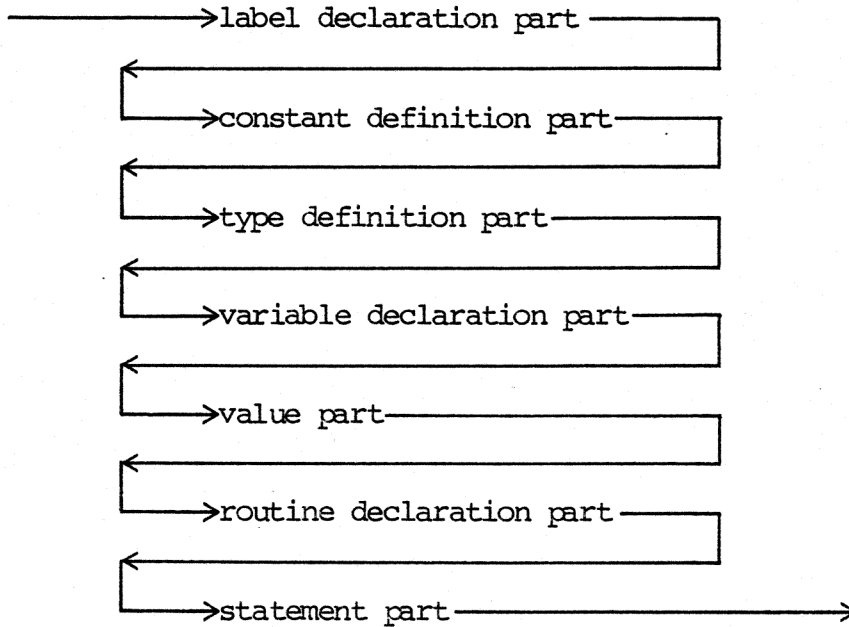
On the RC8000 input and output are initially connected to current input and output allocated by FP. If other files are used for input and/or output by a program there must be an explicit call of `close` before the program terminates.

Note: The program heading must contain the file name output.

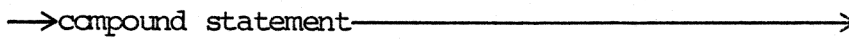
Example of program heading

PROGRAM catalog (output, input= 'pip');

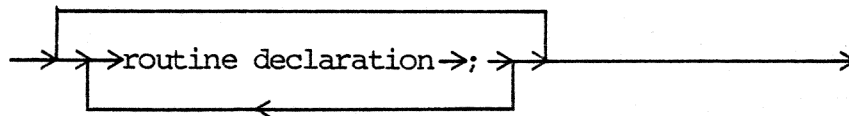
block:



statement part:



routine declaration part:



The following sections will define and show examples of the different elements of a block.

3.3 The Declaration Part

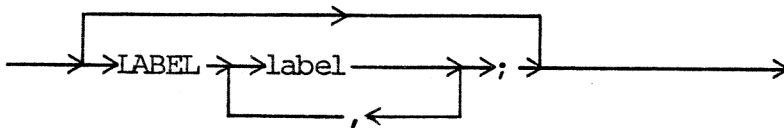
The declarations of a program serve as a description of the data which are manipulated by the actions performed by the program.

3.3.1 Labels

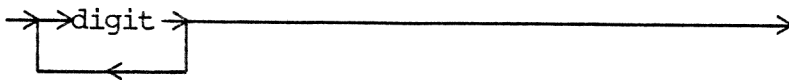
3.3.1

A label is a non negative number less than 10000. Labels must be declared prior to their use. A label is defined in the compound statement of a routine or program. Any such label must be declared in the label declaration part of the routine or program where it is defined.

label declaration part:



label:



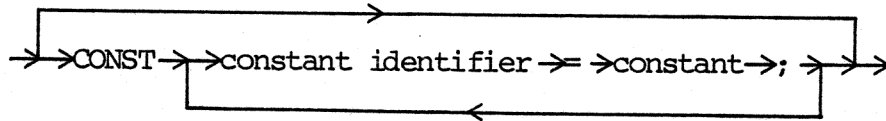
Two labels which denote the same number are considered identical. Labels follow the same rules of scope as other quantities; i.e. they can be used in the rest of the program or routine where they are declared.

3.3.2 Constants

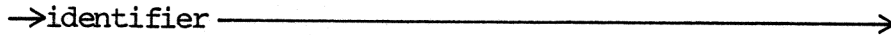
3.3.2

The constant definition part consists of a number of definitions of constants. Each of these definitions introduces an identifier as a synonym for the value of a literal or as a synonym for an enumeration constant from a scalar type.

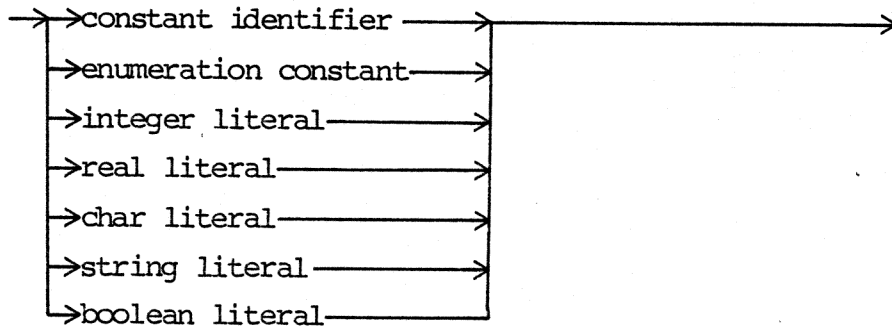
constant definition part:



constant identifier:



constant:



The use of constant identifiers generally makes a program more readable and acts as a convenient documentation aid. It also allows the programmer to group machine or example dependent quantities at the beginning of the program where they can be easily noted and/or changed. (Thereby aiding the portability and modularity of the program).

Example of constant definition part:

```

CONST
    idlength= 10;
    catalogsize= 256;
    version_date = '81.07.17';
  
```

There are some predefined constants:

```

alfalength = 12; (* number of characters in a variable of
                  type alfa (see 3.3.3.3) *)
  
```

```

maxint = 8388607; (* 223-1, the largest possible integer
                  value *)
  
```

```

firstch = ' ';      (* first character of the standard
                    type char (see 3.3.3.1) *)

lastch = '~';      (* last character of the standard
                    type char (see 3.3.3.1) *)

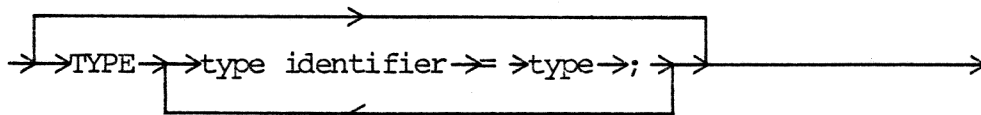
setmax = 143;      (* largest index allowed in a set
                    (see 3.3.3.3) *)
    
```

3.3.3 Types

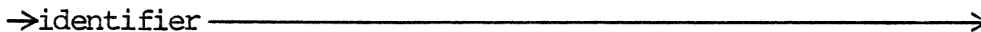
3.3.3

A data type defines the set of values which may be assumed by variables and expressions (in the following called instances) of that type. New data types may be defined in a type definition part.

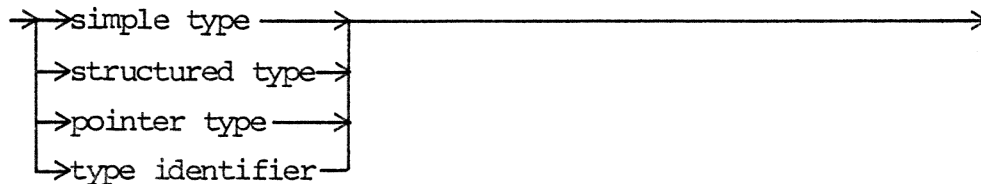
type definition part:



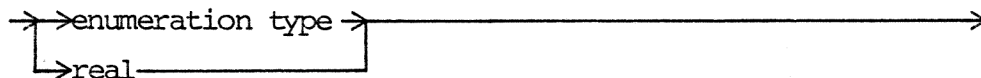
type identifier:



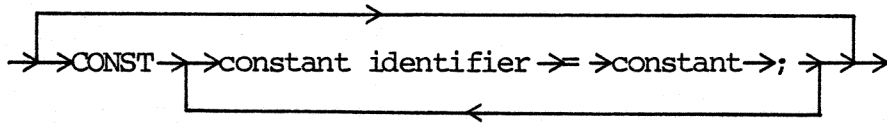
type:



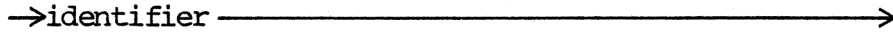
simple type:



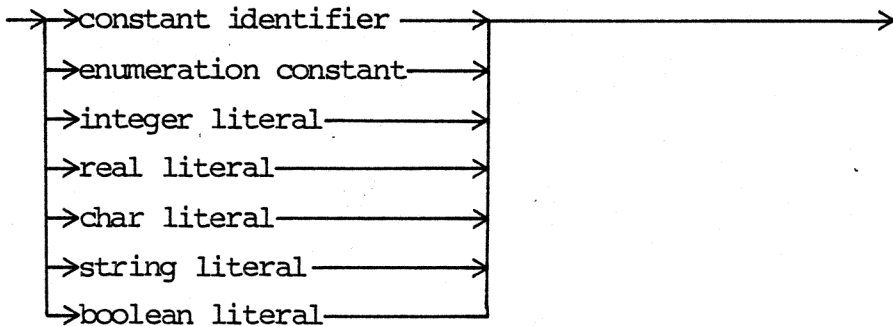
constant definition part:



constant identifier:



constant:



The use of constant identifiers generally makes a program more readable and acts as a convenient documentation aid. It also allows the programmer to group machine or example dependent quantities at the beginning of the program where they can be easily noted and/or changed. (Thereby aiding the portability and modularity of the program).

Example of constant definition part:

```

CONST
    idlength= 10;
    catalogsize= 256;
    version_date = '81.07.17';
  
```

There are some predefined constants:

```

alfalength = 12; (* number of characters in a variable of
                  type alfa (see 3.3.3.3) *)
  
```

```

maxint = 8388607; (* 223-1, the largest possible integer
                  value *)
  
```

```

firstch = ' ';      (* first character of the standard
                    type char (see 3.3.3.1) *)

lastch = '~';      (* last character of the standard
                    type char (see 3.3.3.1) *)

setmax = 143;      (* largest index allowed in a set
                    (see 3.3.3.3) *)

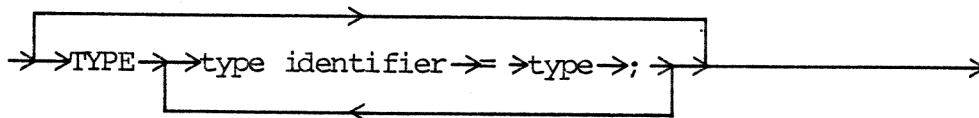
```

3.3.3 Types

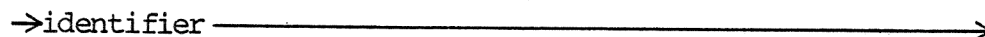
3.3.3

A data type defines the set of values which may be assumed by variables and expressions (in the following called instances) of that type. New data types may be defined in a type definition part.

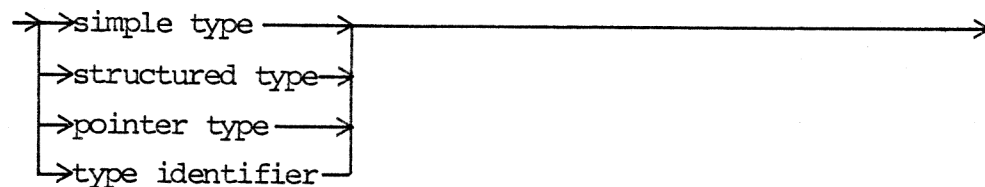
type definition part:



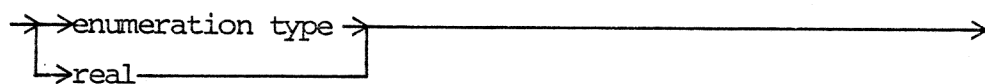
type identifier:



type:



simple type:

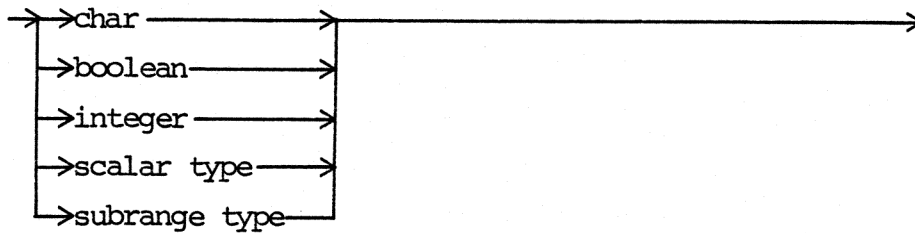


3.3.3.1 Enumeration Types

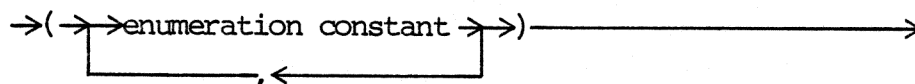
3.3.3.1

An enumeration type consists of a finite, totally ordered set of values.

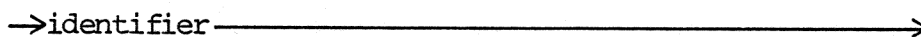
Enumeration type:



scalar type:



enumeration constant:



A scalar type is defined by listing all the possible values in increasing order as a list of identifiers.

Standard simple types

A standard type is denoted by a predefined type-identifier. The values belonging to a standard type are manipulated by means of predefined primitive operations. The following types are standard in Pascal:

integer The values are a subset of the whole numbers, denoted as described in 2.2.4. The predefined integer constant `maxint` defines the subset of the integers available in an implementation over which the integer operations are defined.

The range is the set of values:

`-maxint-1, -maxint, ..., -1, 0, 1, ..., maxint-1, maxint.`

- real** The values are a subset of the real numbers denoted as defined in 2.2.5. The real values are in the range $[-2^{2047} \dots -0.5 \cdot 2^{-2048}, 0, 0.5 \cdot 2^{-2048} \dots 2^{2047}]$ or approximately in $[10^{-616} \dots 10^{616}]$ or the corresponding negative range. For more details see ref. [3] chapter 5.
- boolean** The values are truth values denoted by the identifiers false and true, such that false is less than true.
- char** The values are a set of characters. The denotation of character values is described in 2.2.6. The ordering properties of the character values are defined by the ordering of the ordinal values of the characters, i.e. the relationship between the character variables c_1 and c_2 is the same as the relationship between $\text{ord}(c_1)$ and $\text{ord}(c_2)$. In all Pascal implementations the following relations hold:
- (1) The subset of character values representing the digits 0 to 9 is ordered and contiguous.
 - (2) The subset of character values representing the upper-case letters A to Z is ordered but not necessarily contiguous.
 - (3) The subset of character values representing the lower-case letters a to z, if available, is ordered but not necessarily contiguous.

Integer, boolean and char are enumeration-types. Real is a real-type.

Operators applicable to standard types are defined in the following.

3.3.3.2 Subrange Types

3.3.3.2

An enumeration type can also be defined as a subrange of another enumeration type by specifying its min and max values (separated by .. (double period)). A subrange of the type real is not allowed.

subrange type:

→min value →.. →max value →————→

min value:

→constant————→

max value:

→constant————→

The min value must not exceed the max value and they must be of compatible enumeration types.

A subrange type is in fact a synonym for an enumeration type with a range check included.

Often in this manual, the phrase 'or subrange thereof' is assumed to be implied but is not always mentioned explicitly.

The predefined function ord can also be applied to an instance of a subrange type.

As a consequence of the ordering the following dyadic operators are defined on operands of any enumeration type. They all take two operands of compatible types and yield a boolean result.

- < less than
- <= less than or equal
- = equal
- ◇ not equal
- > greater than
- >= greater than or equal

The following predefined functions apply to instances of all enumeration types. They take one argument and for succ and pred the type of their result is compatible with the type of their argument, if the result is defined.

succ The result is the successor of the argument. If the argument is the last (greatest) value of the type the result is undefined.

pred The result is the predecessor of the argument. If the argument is the first (smallest) value of the type the result is undefined.

ord The result is of type integer and is the ordinal number of the argument in the set of values defined by the type of the argument.

The types iso and char

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

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

char is defined as

```
char = firstch.. lastch; (* 'L'..'~' *)
```

Note: The Danish characters \mathbb{E} , ϕ , \mathbb{A} , \mathbb{a} , ϕ and \mathbb{a} are special symbols, they are not part of the set of characters used for identifiers, but they are used as $[\backslash,], \{, |$ and $\}$. And '#' is used instead of 'f'.

Examples of enumeration and subrange types and their use:

Given the declarations

TYPE

```
suits=(club, diamond, heart, spade);
days=(monday, tuesday, wednesday, thursday, friday,
saturday, sunday);
week_end=friday..sunday; (* subrange type *)
months=(january, february, march, april, may, june, july,
august, september, october, november, december);
seasons=(winter, spring, summer, autumn);
colours=(black, red);
```

Then the following relations are all true.

```
diamond<=heart
monday<sunday
december>=april
wednesday=succ(tuesday)
november=pred(december)
```

Whereas the following relations are all false.

```
club>=diamond
january=february
succ(november)=october
```

The Type Boolean

The type boolean is a predefined enumeration type. Boolean is predefined as `TYPE boolean=(false, true);`

The following operators can be applied to instances of type boolean. They all yield a boolean result.

AND dyadic logical conjunction of the two operands.

OR dyadic logical disjunction of the two operands.

NOT monadic logical negation of the operand.

When the predefined function `ord` is applied to a boolean value the result is the following.

`ord(false)=0`

`ord(true)=1`

Each of the relational operators (`=`, `<`, `<=`, `>`, `>=`) yields a boolean value. Furthermore, the type boolean is defined so that `false < true`. Hence, it is possible to define each of the 16 boolean operations using the above logical and relational operators. For example, if `p` and `q` are boolean values, one can express

implication as `p <= q`
 equivalence as `p = q`
 exclusive or as `p <> q`

The following table shows the value of some boolean expressions.

expression	value
true AND true	true
true AND false	false
false AND true	false
false AND false	false

expression	value
true OR true	true
true OR false	true
false OR true	true
false OR false	false

NOT true	false
NOT false	true

true<true	false
true<false	false
false<true	true
false<false	false

true=true	true
true=false	false
false=true	false
false=false	true

true◇true	false
true◇false	true
false◇true	true
false◇false	false

true<=true	true
true<=false	false
false<=true	true
false<=false	true

true>=true	true
true>=false	true
false>=true	false
false>=false	true

The type integer

The following operators can be applied to instances of type integer. They all yield an integer result.

- + dyadic integer addition of the values of the two operands.
- + monadic monadic plus (redundant).
- dyadic integer subtraction of the value of the right operand from the value of the left operand.
- monadic monadic minus.
- * dyadic integer multiplication of the values of the two operands.
- DIV dyadic the value of the left operand is divided by the value of the right operand. The result is the quotient truncated (i.e. the quotient is not rounded) to integer.
- MOD dyadic $a \text{ MOD } b$ is defined as $a - ((a \text{ DIV } b) * b)$

The following predefined functions all take a single integer argument.

- abs The integer result is the absolute value of the argument.
- sqr The integer result is the square of the argument.
- odd The boolean result is true if the argument is odd; otherwise it is false.
- chr The result (of type char) is the character which has the ordinal value of the argument. As a consequence chr is only defined in the subrange [0..127].
- ord The result (of type integer) is equal to the value of the argument.

The following relations are all true.

$$2+2=4$$

$$-2-2=-4$$

$$5*3=15$$

$$15 \text{ DIV } 3=5$$

$$15 \text{ DIV } 7=2$$

$$11 \text{ DIV } 4=2$$

$$15 \text{ MOD } 3=0$$

$$15 \text{ MOD } 7=1$$

$$11 \text{ MOD } 4=3$$

$$-15 \text{ DIV } 3=-5$$

$$-11 \text{ DIV } 4=-2$$

$$-11 \text{ MOD } 4=-3$$

$$-15 \text{ DIV } (-7)=2$$

$$-15 \text{ MOD } (-7)=-1$$

$$\text{abs}(-3)=3$$

$$\text{sqr}(4)=16$$

$$\text{sqr}(-4)=16$$

$$\text{odd}(3)=\text{true}$$

$$\text{odd}(-3)=\text{true}$$

$$\text{odd}(4)=\text{false}$$

$$\text{odd}(-4)=\text{false}$$

$$\text{odd}(0)=\text{false}$$

$$\text{chr}(65)='A'$$

The Type Real

The predefined type real consists of a finite subset of the real numbers. A value of type real is represented in the RC8000 floating point format [3] the mantissa has 36 bits including a sign and the exponent 12 bits; thus there are at least 11 significant decimal digits.

The following operators can be applied to instances of type real.

+	dyadic	Floating point addition of the values of the two operands.
+	monadic	Monadic plus (redundant).
-	dyadic	Floating point subtraction of the value of the right operand from the value of the left operand.
-	monadic	Monadic minus.
*	dyadic	Floating point multiplication of the values of the two operands.
/	dyadic	Floating point division of the value of the left operand by the value of the right operand.
<=	dyadic	The boolean result is true if the specified relation holds between the two operands, otherwise it is false.
>=	dyadic	
◇	dyadic	
<	dyadic	
>	dyadic	
=	dyadic	

The following predefined functions can be applied to a real argument:

sin, cos, arctan, ln, exp, sqrt, abs, sqr, sinh, arcsin

The result (of type real) is the result of applying the specified mathematical function to the argument.

round

The result (of type integer) is the argument rounded (not truncated) according to the standard mathematical conventions.

trunc

The result is the integer, with the same sign as the argument, whose absolute value is the greatest among the integers less than or equal to the absolute value of the argument.

The difference between trunc and round is illustrated by the following examples

trunc(1.6)=1, trunc(-1.6)=-1, trunc(2.4)=2,
round(1.6)=2, round(-1.6)=-2, round(2.4)=2.

The operators = and \diamond should be used with great care on real arguments. This is due to the round-off error which often results from the representation of real values.

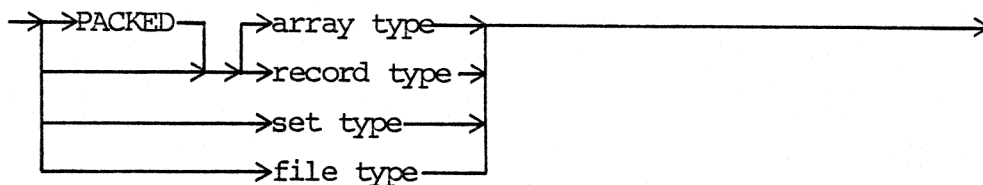
The relative precision of a real number lies between $3 \cdot 10^{-11}$ and $6 \cdot 10^{-11}$.

3.3.3.3 Structured Types

3.3.3.3

A structured type is a composition of other types. The specification of a structured type specifies the structuring method and the component types.

structured type:



Array Types

An array consists of a fixed number of components all of which have the same type. The number of components is specified by an enumeration type (index type). The index type must not be integer, but a subrange of type integer is allowed.

Note: The index type is static and cannot be varied dynamically. This implies that the index type must be known at the compilation time.

array type:

→ARRAY → [→ → index type → →] → OF → component type →

index type:

→enumeration type →

component type:

→type →

Arrays can either be used as a whole or component-wise. A whole array is denoted by its array variable. A component of an array is denoted by the array variable followed by one or more indices separated by commas and enclosed in brackets. An index consists of a number of index expressions. The total number of index expressions must not exceed the dimension of the array. Furthermore the value of each index expression must be of a type compatible with the declaration of the corresponding index.

indexed variable:

→array variable → [→ → expression → →] →

array variable:

→variable →

Examples of array declarations and denotations:

Assume the declarations

TYPE

hours=8..16;

matrix=ARRAY[1..n,1..n] OF real; (*n is an integer constant *)

counter=ARRAY['a'..'z'] OF integer;

name_of_day=ARRAY[days] OF alfa;

occupied_type=ARRAY[days,hours] OF boolean;

VAR

a,b,c: matrix;
 occupied: occupied_type;

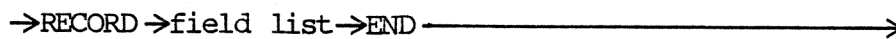
Then the following lines give examples of correct array-denotations.

a := b; (* the entire matrix b is copied into a *)
 c[i]:=a[i]; (* one row of a is copied into the corresponding row
 in c *)
 c[i,j]:=a[k,l]; (* one component of a is copied into one compo-
 nent of c *)
 occupied[wednesday,9]:=true;
 occupied[friday,15]:=false;

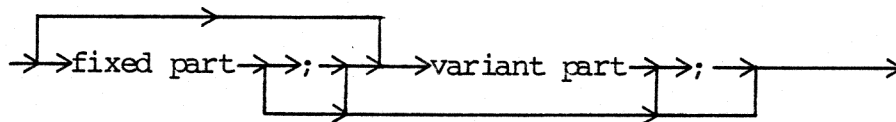
Record Types

A record consists of a fixed number of components called fields, which may be of different types. For each field its field identifier and its type must be specified. A record can be divided into a fixed part and a variant part, either or both of these parts may be empty.

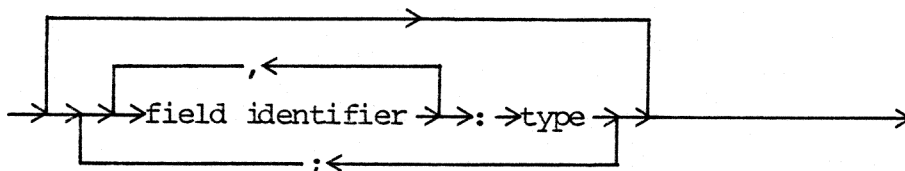
record type:



field list:

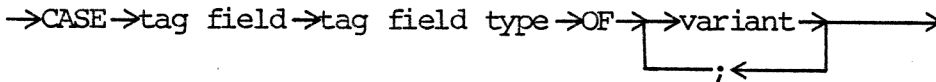


fixed part:

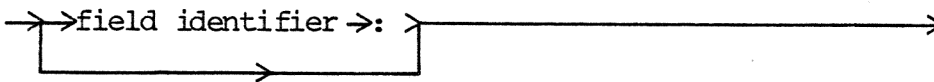


A field list may have a number of variants, in which case a certain field may be designated as the tag field, whose value indicates which variant is assumed by the field list at a given time. The tag field may be empty.

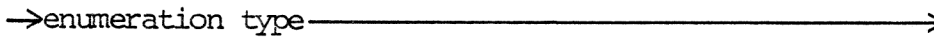
variant part:



tag field:



tag field type:



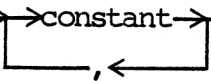
Note:

1. All field names must be distinct - even if they occur in different variants.
2. If the field list for a label L is empty, the form is:
L : ()
3. A field list can have only one variant part and it must succeed the fixed part(s). (However, a variant part may itself contain variants. Hence, it is possible to have nested variants).
4. The tag field type must be an enumeration type. Each variant must be labelled with one or more constants of a type compatible with the tag field type. All such labels must be distinct.

variant:

→case label list→: →(→field list →)→

case label list:

→constant→

 A horizontal arrow points from the word 'constant' to the right. Below the arrow, a bracket starts under 'constant' and ends under a comma, with an arrow pointing back to the left.

The value of the tag field determines which variant can be manipulated.

Records can either be used as a whole or component-wise. A component of a record is denoted by the record variable followed by the field identifier of the component separated by a period.

field designator:

→record variable→.→field identifier→

record variable:

→variable→

field identifier:

→identifier→

Note: It is not checked that the tag field has the correct value when a component of a variant part is referred to.

Examples of record definitions:

TYPE

date=RECORD

year: integer;

month: 1..12;

day:1..31

END;

```

person=RECORD
    name, firstname: alfa;
    age: 0..99;
    CASE married: boolean OF
    true:(spousesname: alfa);
    false:()
END;

```

```

figure=RECORD
    x,y: real;
    area: real;
    CASE s: shape OF
    triangle:(side:real;
                inclination, angle1, angle2: angle);
    rectangle: (side1, side2: real;
                skew, angle3: angle);
    circle:(diameter: real);
END;

```

Packed Representation

In order to reduce storage requirements a definition of an array or record type can be prefixed by the symbol PACKED.

Note: The packed representation may result in an increase in execution time and of the size of the compiled code. This is due to the packing and unpacking operations which must be performed every time a component is accessed.

Two predefined procedures are provided for the packing and unpacking of an array of type char.

Assume that a and p are variables of the following types:

```
a:ARRAY[m..n]OF char; p:PACKED ARRAY[u..v]OF char;
```

where $(\text{ord}(n) - \text{ord}(m)) \geq (\text{ord}(v) - \text{ord}(u))$;

$\text{ord}(m) \leq \text{ord}(i) \leq (\text{ord}(n) - \text{ord}(v) + \text{ord}(u))$;

and the index types of the arrays a and p and the type of i are compatible.

Then `pack(a,i,p)` is equivalent to

```

k:=i;
FOR j:=u TO v DO
BEGIN
  p[j]:=a[k];
  k:=succ(k)
END

```

and `unpack(p,a,i)` is equivalent to

```

k:=i;
FOR j:=u TO v DO
BEGIN
  a[k]:=p[j];
  k:=succ(k)
END

```

where `j` denotes an auxiliary variable not occurring elsewhere in the program.

Use of the predefined procedures should be preferred because of their more efficient implementation.

Note: No component of a packed structure may be used as a variable parameter to a routine.

Strings

In 2.2.6 string literals were defined as sequences of characters enclosed by quotes. Strings consisting of a single character are constants of the predefined type `char`, those of `n` characters ($n > 1$) are constants of the type defined as: `PACKED ARRAY [1..n] OF char`; furthermore the type `alfa` is predefined as: `PACKED ARRAY [1..alfalength] OF char`; (on RC8000 `alfalength` is 12).

The relational operators `<`, `>`, `<=` and `>=` are applicable to strings of the same length. The ordering is the lexicographic ordering based on the ordering of the characters.

Set Types

A set type consists of the set of all subsets of some enumeration type. A set type definition is written as follows.

set type:

→SET OF →base type →

base type:

→enumeration type →

The ordinal number of the largest element must not exceed 143, and the ordinal number of the smallest must not be negative. It follows that a set type can contain at most 144 elements.

set:

→ [→ → element → →] →

element:

→ [→ expression →] →

→ [→ expression₁ → .. → expression₂ →] →

A set denotes a set consisting of the expression values. The form $m..n$ denotes the set of all elements i of the base type so that $m \leq i \leq n$. If $m > n$ then $[m..n]$ denotes the empty set. The set expressions must all be of compatible enumeration types. The empty set is denoted $[]$ and is compatible with any set type.

The following three operators take two operands of compatible set types and their result is of a set type compatible with the operand type.

- + The result is the union of the two operand sets.
- * The result is the intersection of the two operand sets.
- The result is the set difference of the two operand sets (i.e. the elements which belong to the left operand but not to the right operand).

The following two operators take two operands of compatible set types and give a boolean result.

- <= The result is true if the left operand is included in the right operand; otherwise it is false.
- >= The result is true if the right operand is included in the left operand; otherwise it is false.

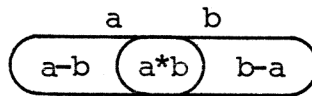
The following operator takes two operands.

- IN The result is true if the left operand is a member of the set specified as the right operand. The left operand must be an instance of an enumeration type compatible with the base type of the right operand.

Assume a and b are of type t and assume t is a set type.

Then the following expression is true.

$$(a-b)+(b-a)=a+b-a*b$$



Assume the declarations

TYPE

```
workingdays=SET OF days;
characters=SET OF "!"..."~";
```

VAR

```
workingday:workingdays;
letters, digits, first, following: characters;
lazy: boolean;
```

Then the following lines are examples of applications of set and set operators.

```
workingday:=[monday..friday];
lazy:=NOT(saturday in workingday);
letters:=["A".."Z","a".."z"];
digits:["0".."9"];
first:=letters;
following:=first+digits;
```

The following relations are all true.

```
first*digits=[]
following*first=letters
```

File Types

A file-type is a structured-type consisting of a sequence of components which are all of one type. The number of components, called the length of the file, is not fixed by the file-type definition. A file with zero components is empty.

At any time only one component of the file is accessible. The other components can be reached by sequencing through the file.

A file type can be defined as follows.

file type:

→FILE OF →type →

The declaration of a file variable introduces a file buffer of the component type. The file buffer is denoted by the file variable followed by an up arrow (↑).

file buffer:

→file variable→↑ →

file variable:

→variable →

The file buffer can be considered as a window through which existing components of the file can be inspected (read) or new components appended (written). A file position is implicitly associated with this window (the file buffer). The window is automatically moved by certain file operations. It is, however, not possible to alternate between reading and writing a file. In a single pass the file can be either read or written.

The sequential processing and the existence of a file buffer suggest that files are associated with secondary storage and peripherals. Exactly how the components are allocated varies, but usually only a few components are present in primary storage at any given time, and only the component denoted by the file buffer is directly accessible.

A special mark is placed after the last component of the file. This mark is called the end-of-file mark (eof).

The predefined routines for file handling are given below. It is assumed that *f* is a file variable and *x* is of a type compatible with the type of the components in the file *f*.

`eof(f)` This boolean function is true if the file is positioned at the end-of-file mark, otherwise it is false.

`reset(f)` The file is repositioned at the start, i.e. the file buffer *ft* contains the first component of the file. The file can now be read. If the file is empty the value of *ft* is undefined, and `eof(f)` is true.

`rewrite(f)` The file is positioned at the start for re-writing. The value of *f* becomes the empty file, *ft* becomes undefined, and `eof(f)` becomes true.

`open(f,<file name>)` Opens the file `f` specified by the `<filename>` of type `PACKED ARRAY [1..n]` of char ($1 \leq n \leq 11$). Only external files (see 3.2) may be opened, and only if they are not already opened.

`close(f)` Closes the file `f`. Only external files may be closed and only if they have been opened.

`get(f)` The position of the file is advanced to the next component. The value of the file buffer becomes the contents of this component. If no next component exists `eof(f)` becomes true, and the value of `f↑` is undefined. If `eof(f)` is true prior to the execution of `get(f)` the call will result in the runtime error message 'try to read past eof'. The call `get(f)` presupposes that the immediately preceding operation on `f` was either `get(f)` or `reset(f)` or equivalent forms.

`put(f)` The value of the buffer variable `f↑` is appended to the file `f`. The value of `f↑` becomes undefined. If the value of `eof(f)` is false prior to the execution the call will result in the runtime error message 'illegal zone-state'. Otherwise the value of `eof` remains true. The call `put(f)` presupposes that the immediately preceding operation on `f` was either `put(f)` or `rewrite(f)` or equivalent forms.

`read(f,x)` A call of `read` is exactly equivalent to executing: `x:=f↑; get(f);`
`x` must be of a type compatible with the type of the components in the file `f`. If `f` is a textfile the reader is referred to the following part about textfiles.

`write(f,x)` A call of `write` is exactly equivalent to executing: `ft:=x; put(f);`
`x` must be of a type compatible with the type of the components in the file `f`.

Note: An open file needs one area process, hence the maximum number of simultaneous open files are limited by the number of area processes of the job.

Note: Routines which have local files should not be called recursively.

Textfiles

A file of characters is called a textfile. Accordingly, the predefined type `text` is defined as: `FILE OF char`.

Texts can be subdivided into lines. The following predefined routines are provided for manipulating the end-of-line mark (`nl`). It is assumed that `t` is a variable of type `text`.

`writeln(t)` Terminate the current line of `t` i.e. write an `nl` character.

`readln(t)` Skip to the beginning of the next line of `t`. Subsequently `t↑` becomes the first character of the next line if any. Thus `readln(t)` has the same effect as the following statements:
`WHILE NOT eoln(t) DO get(t); get(t);`

`eoln(t)` The result of this boolean function is true if `t` is positioned at an end-of-line mark, and false otherwise. If true, `t↑` contains a blank.

page(t) The parameter must be a textfile. page(t) is equivalent to the statement:

```
write(t,ff); (* form feed *)
```

(This will usually force a lineprinter to start on a new page).

To facilitate the manipulation of textfiles, the predefined procedures read and write have some built-in transformation procedures. These translate numbers from the internal binary representation into a character sequence of decimal digits and vice versa. These procedures are called in a non-standard way, since they can be called with a variable number of parameters of various types.

Let t denote a textfile and v, v_1, v_2, \dots, v_n variables of type char, integer or real:

read(t,v) A sequence of characters are read from the file t through the file buffer $t\uparrow$ by means of `get(t)`. The first significant character is the character in $t\uparrow$.

If v is of type char, then `read(t,v)` is exactly equivalent to executing `v:=t↑; get(t);`

If v is of type integer a sequence of digits is transformed into a (decimal) value which is assigned to v . Preceding non-digits are skipped. The character sequence which follows must be consistent with the syntax for decimal integers given in chapter 2. If not, the execution is terminated and a runtime error message is given.

If v is of type real, a sequence of characters is transformed into a real value which is assigned to v . Preceding characters are skipped. The character sequence which follows must be consistent with the syntax for real literals given in chapter 2; with the extension: both ' (quote) and E are accepted as

exponent part indicator. If not, execution is terminated and a runtime error message is given.

If v is of type char, then all preceding non-char characters are skipped, except if $t \neq \text{nl}$ then $\text{eoln}(t)$ becomes true, v becomes ' ' (space), and the next character is moved into the file buffer.

$\text{read}(t, v_1, v_2, \dots, v_n)$ Is a shorthand notation for `BEGIN read(t, v1); read(t, v2); read(t, vn) END`

$\text{readln}(t, v)$ Is a shorthand notation for `BEGIN read(t, v); readln(t) END`

$\text{readln}(t, v_1, \dots, v_n)$ Is a shorthand notation for `BEGIN read(t, v1, v2, ..., vn); readln(t) END`

The predefined procedure `write` is extended in a similar way. Let p, p_1, p_2, \dots, p_n be parameters of the form defined below, and let t be a textfile:

$\text{write}(t, p)$ The parameter p is transformed into a sequence of characters (according to the rules given below). This sequence is written on t .

$\text{write}(t, p_1, p_2, \dots, p_n)$
This is just a shorthand notation for `BEGIN write(t, p1); write(t, p2); ; write(t, pn) END`

$\text{writeln}(t, p_1, \dots, p_n)$
This is just a shorthand notation for `BEGIN write(t, p1, ..., pn); writeln(t) END`

The parameters to the predefined procedures `write` and `writeln` must have the following form.

parameter:

→ expression → : → field width → : → fraction length →

field width:

→ expression →

fraction length:

→ expression →

The first expression (which is the value to be written) must be of one of the following types: integer, boolean, char, real or string. The field width indicates the minimum number of characters to be written. If the field width is longer than needed, the value is written right justified. The field width must be an integer expression with value greater than or equal to 0. If omitted a default value is chosen.

TYPE	DEFAULT FIELD WIDTH	REMARKS
integer	8	
boolean	6	The string "true" or "false" is written.
char	1	
real	14	If fraction length is not specified, the value will be written with 1 digit before the decimal point; 7 digits after the decimal point; and a scaling exponent written as '+ddd' (floating point notation).

TYPE	DEFAULT FIELD WIDTH	REMARKS
		<p>If fraction length is specified, the fraction length must be at least two less than the field width. The fraction length specifies the number of digits to follow the decimal point. If the fraction length is specified, no exponent is written (fixed point notation).</p> <p>If the field width is too short, the necessary number of additional character positions are used.</p>
string	length of string	<p>If a non-zero field width less than the length of the string is specified, the right part of the string is truncated.</p>
alfa	12	

A textfile *t* subdivided into lines can be scanned by the following piece of program.

```

WHILE NOT eof(t) DO
  BEGIN
    WHILE NOT eoln(t) DO
      BEGIN
        read(t,ch);
        q(ch)(* process single character *)
      END
      readln(t);
      r(* process line *)
    END;
  END;

```

A textfile *t* subdivided into lines with maximum *n* significant characters in each line can be scanned by the following piece of program.

```

WHILE NOT eof(t) DO
BEGIN
  i:=0;
  WHILE (i<n)>eoln(t) DO
  BEGIN
    i:=i+1;
    read(t,line[i]);
  END
  readln(t);
  r(* process line *)
END;

```

The Predefined Textfiles Input and Output

Two textfiles named input and output are predefined as

```
VAR input, output: text;
```

The first parameter to read, readln, write or writeln can be omitted, in which case input or output respectively is used.

Let v denote a variable of type char, integer or real. Let e denote an expression of type char, integer, real, boolean or string.

write(e)	is equivalent to	write(output,e)
writeln(e)	is equivalent to	writeln(output,e)
read(v)	is equivalent to	read(input,v)
readln(v)	is equivalent to	readln(input,v)

On the RC8000, input and output are initially connected to current input and output allocated by FP. If disc files are used for input and/or output by a program, there must be an explicit call of close before the program terminates, and if input and/or output have not been connected to disk files, they must not be closed.

Pointer Types

A static variable (statically allocated) is one that is declared in a program and subsequently denoted by its identifier. It is called static, for it exists (i.e. memory is allocated for it) during the entire execution of the block to which it is local. A variable may, on the other hand, be generated dynamically (without any correlation to the static structure of the program) by the procedure new. Such a variable is consequently called a dynamic variable.

Dynamic variables do not occur in an explicit variable declaration and cannot be referred directly by identifiers. Instead, generation of a dynamic variable introduces a pointer value (which is nothing other than the storage address of the newly allocated variable). Hence, a pointer type consists of an unbounded set of values pointing to variables of a type. No operators are defined on pointers except the tests for equality and inequality.

Pointer values are created by the standard procedure new. The pointer value NIL belongs to every pointer type; it does not point to a variable.

pointer type:

→↑→identifier →

The identifier must denote a type which must not be a file type.

The value of a pointer variable is either undefined, NIL or a reference to a variable of specified type. The variable referred by a pointer is denoted by the pointer variable followed by an up arrow (↑).

referred variable:

→pointer variable →↑

pointer variable:

→variable—————→

The declaration of a pointer variable will only cause the computer to allocate space for the pointer, hence no space is allocated for any referred variable before this is explicitly denoted by calling the predefined procedure `new`.

The type of a referred variable is the type specified in the declaration of the pointer type.

The predefined procedures on RC8000 provided for manipulating pointer variables are `new` and `dispose`.

`new(p)` A new variable of the type associated with `p` is allocated on the top of the core area for dynamic variables and a reference to this variable is assigned to `p`.

`new(p,c1,c2,...,cn)` In case the type associated with `p` is a record type with variants, the form `new(p,c1,...,cn)` can be used. `c1,c2,...,cn` is a list of constant selectors used to determine the size of the allocated variable. The size is as if the variable was declared of a record type with the field list formed by the following rule of selection: First, the variant corresponding to the selector `c1` is selected. Then, the field list of this variant is formed by using the selectors `c2,...,cn` (by a recursive application of this rule). Finally the so far formed field list is prefixed by the tag field (if non-empty) and is then substituted for the variant part.

The above description does not imply any assignment to the tag fields.

Note: The variant of the allocated variable must not be changed, and assignment to the entire variable is not allowed. However, the value of single components can be altered.

dispose(p)

dispose(p,c1,c2,...,cn)

In the RC8000 implementation the area used for dynamic variables is handled as a stack, i.e. a call of new(p) is a stacking of a new element of type p†. The unstacking is performed by means of the procedure dispose. The call dispose(p) implies that the core reserved for p† and later allocated variables will be released and reused on later calls of new.

Examples of use of pointer variables.

A list structure can be declared as follows.

```

TYPE
  list= RECORD
    inf: ...;
    next: †list
  END;
VAR
  head: †list;

```

A list structure with two elements can be created as follows.

```

new(head);
head†.inf:= ...;
new(head†.next);
head†.next†.inf:= ...;
head†.next†.next:=NIL;

```

Assume the declarations:

```

CONST
  maxval=50;
TYPE
  atom=RECORD
    name: alfa;
    number: integer;
    weight: real;
    occupied: SET OF 1..maxval;
    bindings: ARRAY[1..maxval] OF ↑atom;
    charge: (plus, minus, neutral);
    saturated: boolean
  END;

VAR
  a: atom;

```

Then the following statements give all names of atoms to which a is bound.

```

WITH a DO
  FOR i:=1 TO maxval DO
    IF i IN occupied THEN writeln(i,bindings[i]↑.name);

```

3.3.3.4 Type Compatibility

3.3.3.4

Compatibility of types is defined by so-called "name equivalence" as follows:

Any type is compatible with itself.

Any two types are compatible if a type exists that is compatible with both of them.

Any two set types are compatible if their base types are compatible. The type of the empty set [] is compatible with any set type.

Any subrange type is compatible with the type of which it is a subrange.

Any two file types are compatible if their component types are compatible.

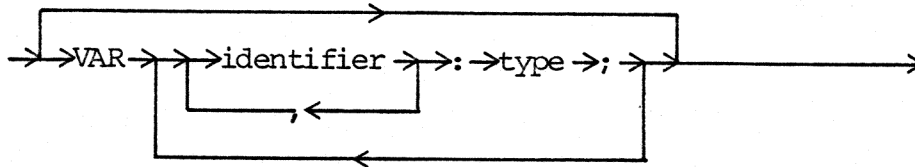
Any two pointer types are compatible if the variables referred by the pointers are of compatible types. The type of the pointer value NIL is compatible with any pointer type.

3.3.4 Variables

3.3.4

A variable is a named data structure that contains a value. Each variable must be declared in a variable declaration part prior to its use. The name and data type of each variable must be specified.

variable declaration part:

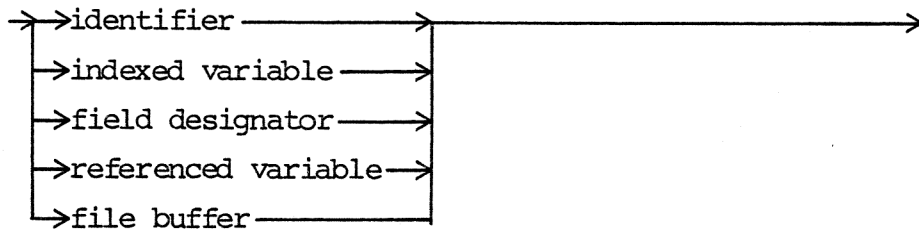


Several variables of the same type may be declared in a single list of identifiers followed by the type.

An entire variable is denoted by its identifier.

If a variable is of array type or record type, a single component is denoted by the identifier, followed by a selector specifying that component (see subsection 3.3.3.3).

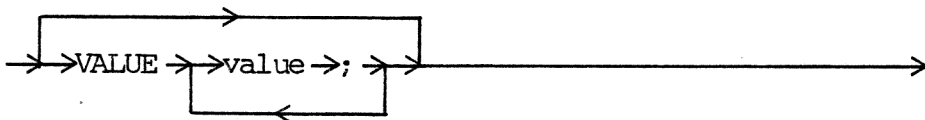
variable:



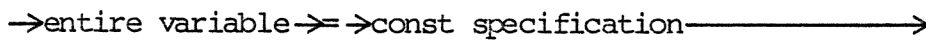
3.3.5 Value Part

3.3.5

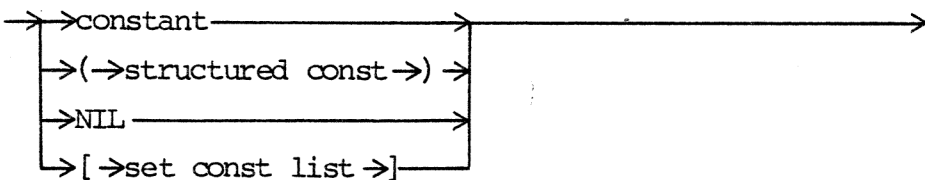
value part:



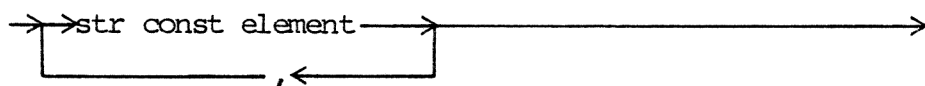
value:



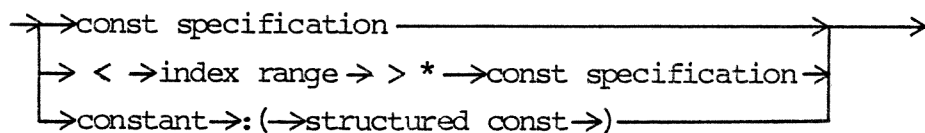
const specification:



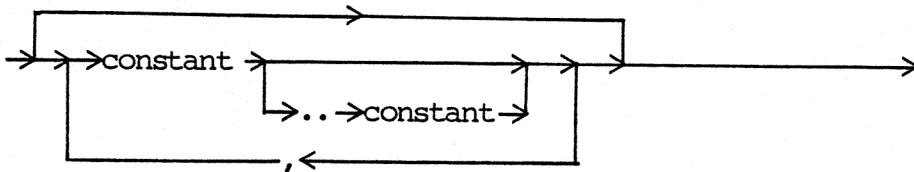
structured const:



str const element:



set const list:



The value-part is used to give local variables initial values on entry to a block, each variable in the value part is initialized according to the const specification on the right hand side of the equal sign.

For structured variables all parts must be specified, and the tag field in a RECORD with a CASE must be specified even if the tag field is empty in the definition of the RECORD. The initialization of a tag field and the associated variant are specified by

value of the tag field : (value of variant)

Examples of value specifications:

Let x be declared as

```
x: RECORD
  a1: char;
  CASE integer OF
    1: (a2 : boolean;
        a21 : SET OF 0..10);
    2: (a3: 0..25; a4, a5: char);
    3: (a6: real);
  END;
```

If the value part contains

```
x=('A',1:(true,[0,3..5])) then
```

x.a1 is initialized to 'A'

x.a2 is initialized to true

x.a21 is initialized to [0,3,4,5]

or if the value part contains

```
x=('B',2:(5,'C','D')) then
```

x.a1 is initialized to 'B'

x.a3 is initialized to 5

x.a4 is initialized to 'C'

x.a5 is initialized to 'D'

As a compact notation for giving the same value to a number of consecutive array elements it is possible to specify the index range followed by the specification of one value.

e.g. If b is declared as

```
b: ARRAY [2..25] OF integer
```

and if the valuepart contains

```
b=(5,<3..10> * 0,3,4,<13..25> * 10) then
```

b[2] is initialized to 5

b[3] is initialized to 0

.

.

.

b[10] is initialized to 0

b[11] is initialized to 3

b[12] is initialized to 4

b[13] is initialized to 10

.

.

.

b[25] is initialized to 10

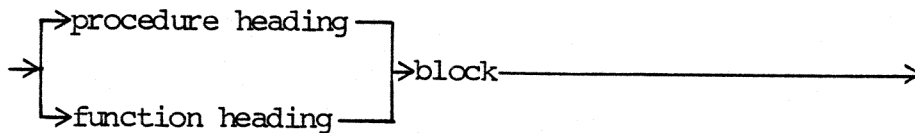
Note: Pointer variables may only be initialized to NIL. Each variable may only occur once in the valuepart.

3.3.6 Routine Declaration

3.3.6

A routine declaration serves to associate an identifier with a set of definitions, declarations and a statement. The execution of this statement can be invoked by a routine call. Routine is a generic term for procedures and functions.

routine declaration:



procedure heading:

→PROCEDURE →procedure identifier →formal parameters →; →

function heading:

→FUNCTION →function identifier →formal parameters →: →
 →type identifier →; →

procedure identifier:

→identifier →

function identifier:

→identifier →

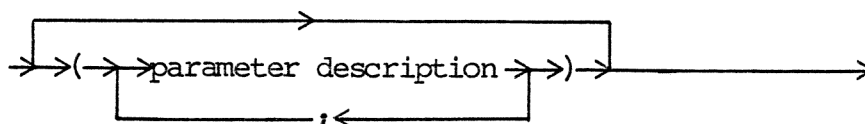
type identifier:

→type →

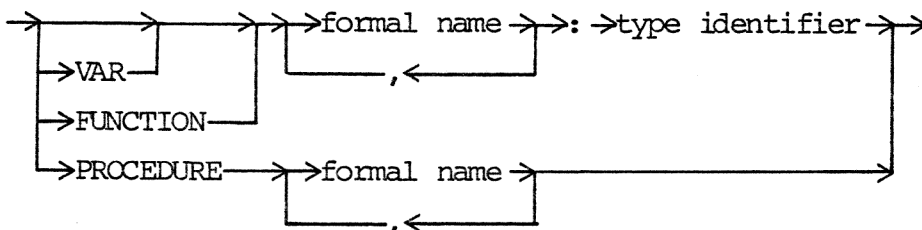
A list of formal parameters may be specified in the routine heading. For each formal parameter is specified its name (formal name) and its kind. There are the following four parameter kinds: value, variable, procedure and function. The kind value is as-

sumed if nothing else is specified. The kinds variable, procedure and function are specified by the symbols VAR, PROCEDURE and FUNCTION respectively. In addition the types of all value, variable or function parameters must be specified. The parameter kind defines the binding between actual parameters and formal parameters in a routine (see 3.4.6).

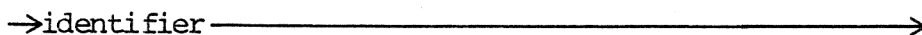
formal parameters:



parameter description:



formal name:



In the block of the routine formal parameters are denoted by their formal names.

A formal parameter of kind value may be used as a local variable of the specified name and type, the value of which is initialized to the value of the actual parameter at the routine call.

A formal parameter of kind variable denotes a variable of the specified name and type. The denoted variable is the actual parameter.

A formal parameter of kind procedure or function may be used as if it was locally declared with all formal parameters of kind value.

The difference between the various kinds of parameters is explained in subsection 3.4.6.

The following are all examples of routine headings.

```
FUNCTION my_own_sqrt(x:real):real;  
FUNCTION zero(lower,upper:real;FUNCTION f:real):real;  
PROCEDURE insert(element:component_type);  
PROCEDURE update(VAR element: component_type);
```

The block of a routine consists of a number of definitions and declarations and a compound statement.

Within the block of the routine the routine name itself may be used to denote a recursive call of the routine. However the occurrence of a function identifier as a left hand side of an assignment statement denotes changes in the current value of the function. Such occurrences are only allowed within the compound statement of the block of the function.

The type of the values which can be returned by a function must be specified in the function head. The value of a function is determined by the dynamically last value assigned to the function identifier within the block of the function. The type of a function is restricted to be a simple type or a pointer type.

The following are all examples of function declarations.

```

FUNCTION zero(FUNCTION test: boolean; lower,upper: real; FUNCTION f:
  real):real;
VAR centre,y:real;s:boolean;
BEGIN(* compute solution to f(x)=0 by bisection *)
  s:=f(lower)<0;
  REPEAT
    centre:=(lower+upper)/2;
    y:=f(centre);
    IF(y<0)=s THEN lower:=centre
    ELSE upper:=centre;
  UNTIL test(lower,upper);
  zero:=centre
END(* zero *);

```

test(lower,upper) is true if and only if the difference between lower and upper is small enough.

The following machine-independent function can be used unless the solution is 0.0.

```

FUNCTION test(i,u: real): boolean;
BEGIN
  test:=((u+i)/2=u) OR ((u+i)/2=i)
END

```

```

FUNCTION sign(x: real):integer;
BEGIN
  if x<0 THEN sign:=-1 ELSE sign:=ord(x>0)
END;

```

```

FUNCTION bincoef(p,q:integer):integer;
(* Calculates binomial coefficient  $p!/(q!(p-q)!)$ . The function
is computationally inefficient but may be useful when only single
values are desired *)
BEGIN
  IF p-q<q THEN q:=p-q;
  IF q<0 THEN bincoef:=0
  ELSE
    IF q=0 THEN bincoef:=1
    ELSE
      bincoef:=bincoef(p-1,q-1)+bincoef(p-1,q)
  END;

```

The names introduced by a definition or by a declaration in a routine, (a local definition or declaration) are only valid in the rest of the block of the routine. On the other hand local definitions and declarations take precedence over definitions and declarations in the surroundings (global definitions and declarations). As routine declarations can be nested, the same routine name can be introduced at several levels. In this case a use of the name will always refer to the innermost declaration.

Routine Pseudo-declaration

The scope rules of Pascal (see chapter 4) require that the declaration of a routine must appear in text before use.

A routine may be pseudo-declared by substituting the block of the routine declaration with the identifier FORWARD.

routine pseudo-declaration:

→routine heading →FORWARD →

A routine declaration where the block is substituted by the identifier FORWARD serves as an announcement of the full block which is given in text later. The block itself is then just headed by a routine head the formal parameters are not needed, but it is allowed to specify them again.

Example:

The scope rules of Pascal lead to a conflict in the situation where two routines call each other. (Which one should be declared first?). The conflict can be avoided by substituting the reserved word FORWARD for the body of the first routine and postponing the specification of the routine body. The following is an example of this.

```

FUNCTION g(x:real):real;FORWARD;
FUNCTION f(x): real;
.....
.....
BEGIN.....g(x).....END;
.....
FUNCTION g(x:real):real;
BEGIN.....f(x).....END;

```

3.4 The Statement Part

3.4

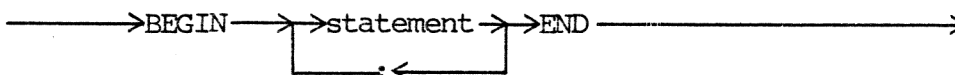
This section contains subsections describing the syntax and the use of the different statements which are included in the language.

3.4.1 Statements

.3.4.1

The statements of a program describe the manipulations performed on data when the program is executed. These statements are collected in a compound statement.

compound statement:



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

3.4.2.1 Expressions

3.4.2.1

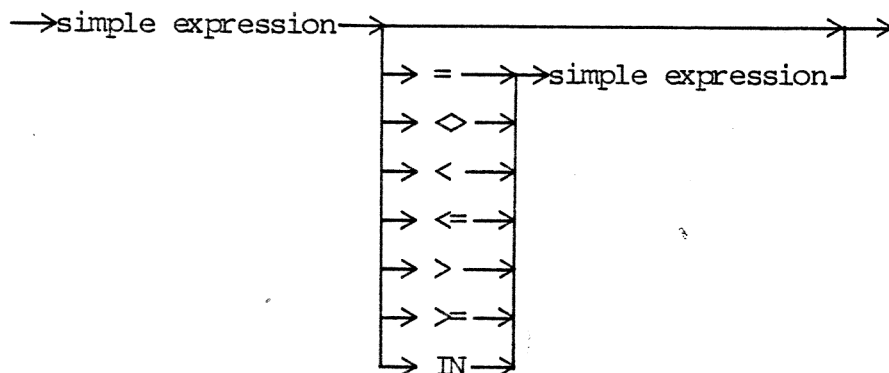
An expression defines a rule of computation for obtaining a value by application of operators to operands. An expression is evaluated using the following precedence rules.

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

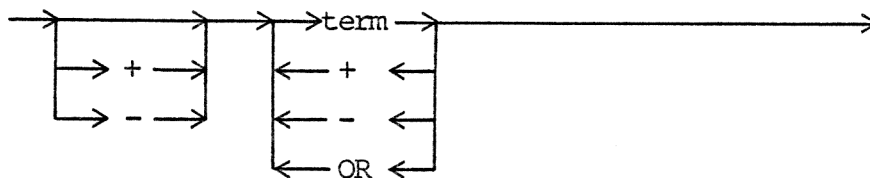
Expressions are written in infix notation.

Note: All factors in an expression may be evaluated and hence should all be defined.

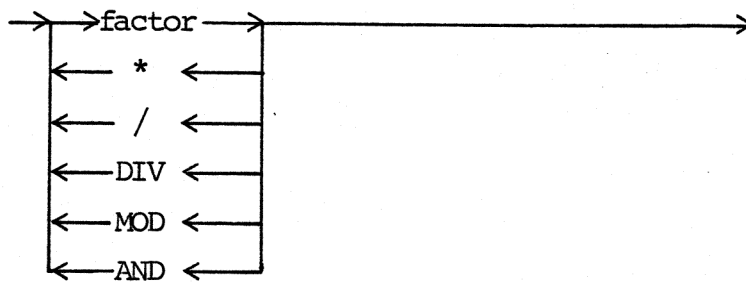
expression:



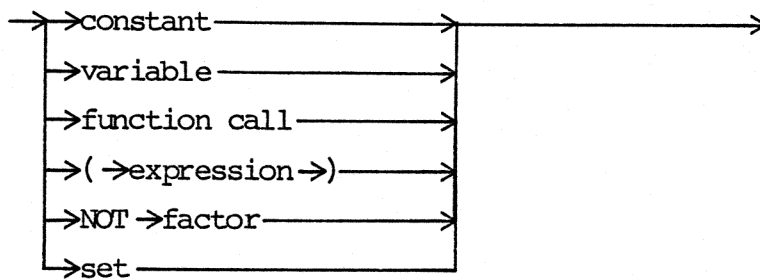
simple expression:



term:



factor:



The following two statements are different.

```
IF (1<=n) AND (table[1]=key) THEN s;
IF 1<=n THEN IF table[1]=key THEN s;
```

The following table gives all valid combinations of dyadic operators and operand types:

operator(s)	left operand	right operand	result
+,-,*	integer	integer	integer
	integer	real	real
	real	integer	real
	real	real	real
	any set type T	T	T
DIV,MOD /	integer	integer	integer
	integer	integer	real
	integer	real	real
	real	integer	real
	real	real	real
OR,AND	boolean	boolean	boolean
=,◇	any type T (see Note)	T	boolean
<=,>=,<,>,◇	any string type T	T	boolean
	any enum.type T	T	boolean
IN	any enum.type T	SET OF T	boolean

Note: Files cannot be compared.

The corresponding table for monadic operators is as follows:

operator(s)	operand	result
+,-	integer	integer
	real	real
NOT	boolean	boolean

Note: During evaluation of an expression, intermediate results are kept in registers and in some reserved locations. If the number of intermediate results exceeds the capacity of reserved

space, the expression cannot be translated and the compiler issues the error message 311: Not enough room for temporaries. To remedy this, the expression must either be rewritten with a less complicated paranthesis structure or split into two or more expressions.

3.4.3 Goto Statement

3.4.3

goto statement:

→GOTO →label →

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

The statement defining the label must be within the same or a surrounding block of the block where the goto is given, i.e. it is not possible to jump into an inner routine by a goto statement. Furthermore the result of jumping into an inner statement of an if, while, repeat, with, for or case statement is undefined.

labelled statement:

→label →: →statement →

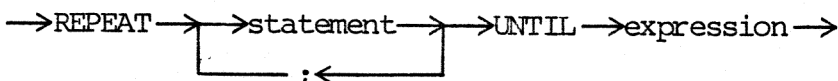
3.4.4 Repetitive Statements

3.4.4

Repeat Statement

The repeat statement specifies that a sequence of statements is to be executed repeatedly.

repeat statement:

→REPEAT → statement → UNTIL → expression →


The result of the expression must be of type boolean.

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

While Statement

The while statement specifies that a statement is to be executed a number of times.

while statement:

→WHILE →expression →DO →statement →

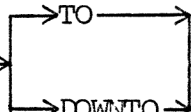
The expression must yield a result of type boolean. The statement following DO will be executed a number of times (possibly 0) and the expression will be evaluated before each execution. This will be repeated until the evaluation of the expression yields a result which is false. Thus, for example if the value of the expression is false prior to the execution of the while statement, the statement following DO will not be executed at all.

For Statement

for statement:

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

for list:

→expression₁ →  →expression₂ →

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

The repeated statement must not change the value of the control variable.

The control variable must be simple (i.e. not of array type, not of record type, not of pointer type and not function identifier).

The statement is executed with consecutive values of the variable. The ordinal value of the variable can either be incremented (in steps by 1 (succ)) from expression₁ TO expression₂, or decremented (in steps by 1 (pred)) from expression₁ DOWNTO expression₂. The two expressions are evaluated once, before the repetition. If the value of expression₁ is greater than the value of expression₂ and TO is specified, the statement is not executed.

Similarly, if the value of expression₁ is less than the value of expression₂ and DOWNTO is specified, the statement is not executed.

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

The value of $i=j$ in the following example depends on the value of n . If n is less than 1 i is unchanged, else i is equal to n .

```
FOR i:=1 TO n DO...;
IF i=j THEN...
```

The assignment statement $i:=n+1$ in the following example is not allowed.

```
FOR i:=1 TO n DO
BEGIN
  ...
  i:=n+1;
  ...
END;
```

3.4.5 Conditional Statements

3.4.5

A conditional statement, an if or case statement, selects a single statement of its component statements for execution.

If Statement

if statement:

→IF →expression →THEN →statement₁ →false part →

false part:

→ELSE →statement₂ →

The expression must yield a result of type boolean. Statement₁ will only be executed if the value of the expression is true. If it is false, the statement (if any) following ELSE (statement₂) will be executed.

The ambiguity arising from the construction:

IF e₁ THEN IF e₂ THEN s₁ ELSE s₂

is resolved by interpreting the construction as equivalent to:

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


The following are examples of if statements.

```

IF day=sunday THEN next:=monday
    ELSE next:=succ(day)
IF x>y THEN
BEGIN
    min:=y; max:=x
END
ELSE
BEGIN
    min:=x; max:=y
END;

```

Note: The following two statements are different.

```

IF (l<=n) AND (table[l]=key) THEN s;
IF l<=n THEN IF table [l]=key THEN s;

```

In the case where $l > n$ the former may evaluate $\text{table}[l]=\text{key}$ and probably cause an index error.

If the expression is constant no testing code is generated, and code is only produced for the chosen part of the if statement.

Example.

The constant 'test' may be true in the debugging phase, and set to false in the resulting program, i.e. code for test output is only generated while the program is tested.

```

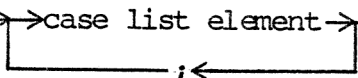
CONST
    test=false;
    .
    .
    .
if test then writeln ('Kilroy was here');
    .
    .
    .

```

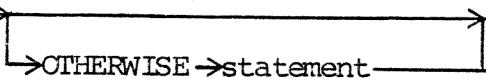
Case Statement

A value of an enumeration type can be used to select one of several statements for execution.

case statement:

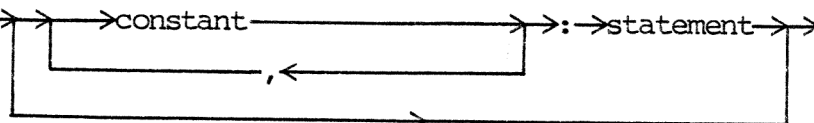
→CASE→expression→OF→

 →end part→

end part:

→END→

 →OTHERWISE→statement→

A case list element is a statement labelled by one or more constants. These constants must all be of a type compatible with that of the expression. All labels (constants) in a case statement must be unique. The statement labelled by the current value of the expression is selected for execution. Upon completion of the selected statement the case statement is also completed.

case list element:

→

 →:→statement→

Notes: The case statement is translated into a jump table. The size of this table is limited. Hence no two labels l_1 and l_2 of one case statement may be chosen so that

$$\text{abs}(\text{ord}(l_1) - \text{ord}(l_2)) > 4000.$$

"Case labels" are not ordinary labels and cannot be referred by a goto statement. Their ordering is arbitrary; however, labels must be unique within a given case statement.

The value $(-\text{maxint}-1)$ is not allowed as case label.

Assume the declarations

```
VAR
  suit: suits;
  colour: colours;
```

Then the following is an example of a case statement.

```
CASE suit OF
  club, spade: colour:=black
END
OTHERWISE colour:=red;
```

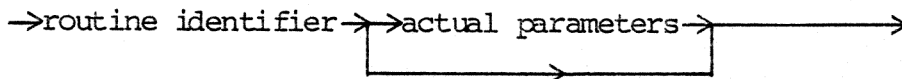
3.4.6 Procedure Call

3.4.6

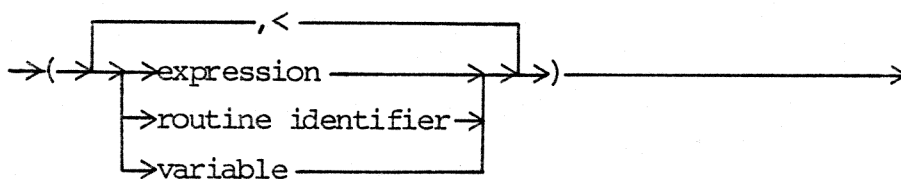
A routine call serves to establish a binding between actual and formal parameters and to allocate locally declared variables and invoke execution of the compound statement of the routine block in its proper surroundings. A routine call consists of the routine identifier followed by a list of actual parameters.

When the compound statement is completed, locally declared variables are deallocated, and execution is resumed at the point immediately after the routine call.

routine call:



actual parameters:



If the routine is declared without formal parameters, the routine call consists of the routine identifier only.

If the routine is declared with a list of formal parameters, this list will be replaced by the list of actual parameters prior to the execution of the routine. The number of actual parameters must be identical to the number of formal parameters. An actual and its corresponding formal parameter have the same position in their respective lists.

There exists the following four kinds of bindings between an actual and its corresponding formal parameter:

value The actual parameter must be an expression or a variable of a type compatible with that of the formal parameter. The value of the expression or variable will be evaluated and substituted in place of the formal parameter. Changes within the block of the routine to the formal parameter will not affect the actual parameter. (The usual term for this parameter binding is call by value).

variable The actual parameter must be a variable of a type compatible with that of the formal parameter. All changes within the block of the routine to the formal parameter will affect the actual parameter directly. The formal parameter denotes throughout the routine body a specific variable of the specified type. The actual parameter specifies which actual variable the formal parameter must denote, if the actual parameter denotes a component of a structured type or a referenced variable the computation of which variable is to be denoted is only performed once at the routine call. (The usual term for this parameter binding is call by reference).

A component of a packed structure cannot be given as an actual variable parameter.

All the actual variable parameters of a given call should denote distinct variables, or else the effect of the routine call will be difficult to comprehend.

procedure The actual parameter must be the name of a procedure. This procedure must either be declared with all formal parameters specified as kind value or it must itself denote a formal parameter of kind procedure.

function The actual parameter must be the name of a function. This function must either be declared with all formal parameters specified as kind value or it must itself denote a formal parameter of kind function.

The type of the actual parameter function must be compatible with the type of the formal parameter.

Note: If the routine call is a call of a formal parameter of kind procedure or function the correspondence between the lists of actual and formal parameters cannot be checked by the compiler.

Note: A predefined routine must not be used as an actual parameter of kind procedure or function.

Note: A parameter of file type must be passed as a variable parameter.

As a guide to the choice between value and variable specification the following should be noted:

If a parameter is not used to transfer a result of the procedure a value parameter is generally preferred.

The referencing is then quicker and one is protected against mistakenly altering the data. However in the case where a parameter is of a structured type one should be cautious because the value specification may lead to unacceptable inefficiency compared to a variable specification. The explanation is as follows: A procedure allocates a new storage area for each value parameter which the formal parameters denoted.

The value of the actual parameter is assigned to this storage area. The assignment operation may be time consuming and the amount of storage allocated to the formal parameter may be large.

The set of local variables of a routine can be regarded as associated with a specific call of the routine; they exist from the moment the execution of the routine starts and until it is completed. Thus, in case of recursive calls of a routine, several incarnations of the local variables and formal parameters may exist simultaneously, namely one incarnation for each uncompleted call. By execution of a routine is meant the execution of the compound statement of its body. The execution is completed, either when the compound statement is completed, or when a jump to a label in a surrounding routine is performed. The only difference between a procedure and a function call is that a procedure call is a statement, and a function call is a factor which may be used in an expression.

Example:

```

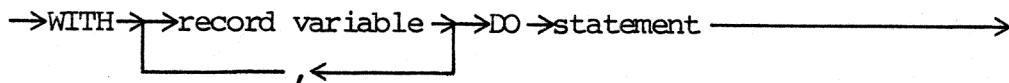
TYPE
  list=ARRAY[1..100] OF integer;
FUNCTION maximum (VAR l: list): integer;
(* l is of kind variable to save time and space *)
VAR
  i, max: integer;
BEGIN
  max:=l[1];
  FOR i:=2 TO 100 DO
    IF max<l[i] THEN max:=l[i];
  maximum:=max
END;
```

3.4.7 With Statement

3.4.7

A with statement can be used to facilitate the manipulation of record components.

with statement:



Within the statement the fields of the record variable(s) can be denoted by giving their field identifiers only (without preceding them with the denotation of the entire record variable).

For a nested with statement in the form

```

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

```

you may use the following shorthand notation

```

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

```

If a set of variables (of enumeration type) is used for selecting the record variable (e.g. the variable *i* in the statement `WITH a [i] DO`) then the values of these variables must not be changed in the statement. However, a violation of this rule cannot be checked. The only effect of such a violation will be the change of the values of these variables.

Examples:

```

WITH hand[1] DO
BEGIN
    t:=normal;
    suit:=club;
    rank:=8
END;

```

```
WITH date DO
  IF month = 12 THEN
    BEGIN
      month:= 1; year:=year+1
    END
  ELSE month:=month+1
```

is equivalent to

```
IF date.month=12 THEN
  BEGIN
    date.month:=1; date.year:=date.year+1
  END
ELSE date.month:=date.month+1
```


This chapter contains the detailed scope rules.

The scope of a name is the declarations and statements in which the declaration of the name is valid. All names must be declared textually before they are used.

The scope depends on the kind of the object denoted by the name.

Label-, Constant-, Type-, Variable- or Routine-names.

The scope of the name is the rest of the program or routine in which it is declared.

Parameter-names.

The scope of a formal parameter is the body of the routine.

Field-names.

The scope of a field name in a record is only that record.

Enumeration-values.

The scope of an identifier introduced as a value of an enumeration type is the rest of the program or routine in which it is declared.

Program-name.

The program name has no significance within the program.

The same identifier must be introduced at most once in each body or record. If the scopes of an identifier overlap, it is always the innermost scope which is valid.

5. PREDEFINED ROUTINES

5.

Standard routines are predeclared in the implementation of Pascal. Since they are, as all standard quantities, assumed as declared in a scope surrounding the program, no conflict arises from a declaration redefining the same identifier within the program. The standard procedures are listed and explained below.

5.1 Standard Procedures

5.1

Standard procedures are not allowed as actual procedural parameters.

5.1.1 File Handling Procedures

5.1.1

put(f)	see under FILE types in subsection 3.3.3.3
get(f)	-
read	-
readln	-
write	-
writeln	-
page(f)	-
reset(f)	-
rewrite(f)	-
open(f, file name)	-
close(f)	-

5.1.2 Dynamic Allocation Procedures

5.1.2

new(p) see under Pointer Types in subsection 3.3.3.3

dispose -

5.1.3 Transfer Procedures

5.1.3

pack see under Packed Representation in subsection 3.3.3.3

unpack

5.1.4 Date and Time

5.1.4

date(a): assigns the current date to the alfa variable a, in the form: 'yy.mm.dd. '.

time(a): assigns the current time to the alfa variable a, in the form: 'hh.mm. '.

5.1.5 Program Control Procedure

5.1.5

replace (program name) The procedure replace terminates the current program and invokes the program denoted by program name. Program name must be the name of a Pascal object file, i.e. a compiled Pascal program.

The program which calls replace must prior to the call close all files, (except input and output, if they are not connected to external files).

The procedure is restricted only to be called from the main program.

The procedure returns to the invoked program. If an error occurs during the replacement the execution is terminated and an error message is given.

2 Standard Functions

5.2

Standard functions are not allowed as actual functional parameters.

.2.1 Arithmetic Functions

5.2.1

For the following arithmetic functions, the type of the expression x is either real or integer. For the functions `abs` and `sqr`, the type of the result is the same as the type of the parameter, x . For the remaining arithmetic functions, the type of the result is always real.

<code>abs(x)</code>	computes the absolute value of x .
<code>sqr(x)</code>	computes the square of x .
<code>sin(x)</code>	computes the sine of x , where x is in radians.
<code>cos(x)</code>	computes the cosine of x , where x is in radians.
<code>exp(x)</code>	computes the value of the base of natural logarithms raised to the power x .
<code>ln(x)</code>	computes the natural logarithm of x , if x is greater than zero. If x is not greater than zero an error occurs.
<code>arcsin(x)</code>	computes the principal value, in radians, of the arcsine of x .
<code>sinh(x)</code>	computes the hyperbolic sine of x .
<code>sqrt(x)</code>	computes the positive square root of x , if x is not negative. If x is negative an error occurs.
<code>arctan(x)</code>	computes the principal value, in radians, of the arctangent of x .

5.2.2 Transfer Functions

5.2.2

`trunc(x)` From the real parameter `x`, this function returns an integer result which is the integral part of `x`. The absolute value of the result is not greater than the absolute value of the parameter.

For example:

`trunc(3.7)` yields 3

`trunc(-3.7)` yields -3

`round(x)` From the real parameter `x`, this function returns an integer result which is the value of `x` rounded to the nearest integer. If `x` is positive or zero then `round(x)` is equivalent to `trunc(x+0.5)`, otherwise `round(x)` is equivalent to `trunc(x-0.5)`.

For example:

`round(3.7)` yields 4

`round(-3.7)` yields -4

5.2.3 Ordinal Functions

5.2.3

`ord(x)` The parameter `x` is an expression of ordinal-type. The result is of type integer. If the parameter is of type integer then the value of the parameter is yielded as the result. If the parameter is of any other ordinal-type, the result is the ordinal number determined by mapping the values of the type on to consecutive non-negative integers starting at zero.

For example:

`ord(false)` yields 0

`ord(true)` yields 1

`chr(x)` Yields the character value whose ordinal number is equal to the value of the integer expression `x`, if such a character value exists.

For any character value, *ch*, the following is true:

`chr(ord(ch))=ch`

`succ(x)` The parameter *x* is an expression of ordinal-type. The result is of a type identical to that of the expression. The function yields a value whose ordinal number is one greater than that of the expression, if such a value exists. If such a value does not exist, the result is undefined.

`pred(x)` The parameter *x* is an expression of ordinal-type. The result is of a type identical to that of the expression. The function yields a value whose ordinal number is one less than that of the expression *x*, if such a value exists. If such a value does not exist, the result is undefined.

5.2.4 Predicates

5.2.4

`odd(x)` Yields true if the integer expression *x* is odd, otherwise it yields false.

`eof(f)` Indicates whether the associated buffer variable *ft* is positioned at the end of the file *f*.

`eoln(f)` Indicates whether the associated buffer variable *ft* is positioned at the end of a line in the textfile *f*.

5.2.5 Processing Time Function

5.2.5

`clock` Clock is a parameterless real function, the result of which is the current processing time in seconds with an accuracy given by the length of a time slice (usually 25.6 milliseconds).

The integer function 'monitor' is the Pascal equivalent of the RC8000 monitor procedures. For the time being only the following calls are implemented:

'create entry' (40)

'lookup entry' (42)

'change entry' (44)

The call is:

```
result := monitor(wanted_function, <file>, tail);
```

where

wanted_function is one of the allowed numbers (40,42,44),

<file> is a file identifier,

tail is declared as tail: ARRAY [1..10] OF integer, and corresponds to the tail of the file catalog entry.

For further information see ref. [4] and ref. [5].

Example:

If the result of "lookup pascal" is

```
pascal      =set 224 disc d.810113.1045 0 0 2.0 68 ; system
              ; 159 139 3 -8388607 8388605
```

then the Pascal statements

```
file_name := 'pascal';
```

```
result := monitor(42, file_name, tail);
```

will return with

```
result = 0
```

and the following contents in tail:

index	word	char			half words		text
1	224	0	0	224	0	224	''
2	6580595	100	105	115	1606	-1677	'dis'
3	6488064	99	0	0	1584	0	'c'
4	0	0	0	0	0	0	''
5	0	0	0	0	0	0	''
6	7846624	119	186	224	1915	-1312	'w:``'
7	0	0	0	0	0	0	''
8	0	0	0	0	0	0	''
9	8192	0	32	0	2	0	' '
10	68	0	0	68	0	68	'D'

where: tail [1] = length of file
tail [2..5] = document name
tail [6] = date and time
tail [9] = contents key * 4096

The function call is successful if and only if the result is zero.
For further information about the unsuccessful results see
chapter 2 of ref. [5].

5.2.7 Access to File Processor Parameters

5.2.7

The function 'system' of type integer gives access to the parameters from the FP-command stack, i.e. the call of the program.

With the declarations

```
VAR
  result, paramno, int: integer;
  alf: alfa;
```


The result of a call:

```
result := system (paramno,int,alf);
```

is:

```
IF (paramno >= 0) AND (paramno <= number of parameters in FP-
    stack) THEN
```

```
result := separator_length
```

```
ELSE result := 0;
```

where separator_length is built as: separator *4096 + length,

int and alf are set according to the following scheme:

```
IF length = 4 THEN
```

```
  BEGIN (* the stack parameter is a number *)
```

```
    int := the parameter;
```

```
    alf := undefined
```

```
  END
```

```
ELSE
```

```
  IF length = 10 THEN
```

```
    BEGIN (* the stack parameter is a word *)
```

```
      int := undefined;
```

```
      alf := the parameter;
```

```
    END;
```

The separator values are

0: end of parameter list

2: new line (start of list)

4: space

6: equality sign

8: point

Example of the numbering of the parameter stack items:

```
p1 = pascal p heap.1000
```

```
0   1   2   3   4
```

Result of calls of system:

paramno	result	int	alf	
0	2*4096+10	-	'pl	'
1	6*4096+10	-	'pascal	'
2	4*4096+10	-	'p	'
3	4*4096+10	-	'heap	'
4	8*4096+4	1000	-	
otherwise	0	-	-	

For further information about separator see ref. [6].

5.3 Complete List of Predefined Routines

5.3

Name:	Subsection:	
abs	3.3.3.2	5.2.1
arcsin	3.3.3.2	5.2.1
arctan	3.3.3.2	5.2.1
chr	3.3.3.2	5.2.3
clock	5.2.5	
close	3.3.3.3	
cos	3.3.3.2	5.2.1
date	5.1.4	
dispose	3.3.3.3	
eof	5.2.4	
eoln	5.2.4	
exp	3.3.3.2	5.2.1
get	3.3.3.3	
ln	3.3.3.2	5.2.1
monitor	5.2.6	
new	3.3.3.3	
odd	3.3.3.2	5.2.4
open	3.3.3.3	
ord	3.3.3.2	5.2.3
pack	3.3.3.3	
page	3.3.3.3	
pred	3.3.3.2	5.2.3
put	3.3.3.3	

Name:	Subsection:	
read	3.3.3.3	
readln	3.3.3.3	
replace	5.1.5	
reset	3.3.3.3	
rewrite	3.3.3.3	
round	3.3.3.2	5.2.2
sin	3.3.3.2	5.2.1
sinh	3.3.3.2	5.2.1
sqr	3.3.3.2	5.2.1
sqrt	3.3.3.2	5.2.1
succ	3.3.3.2	5.2.3
system	5.2.7	
time	5.1.4	
trunc	3.3.3.2	5.2.2
unpack	3.3.3.3	
write	3.3.3.3	
writeln	3.3.3.3	

6. COMPILER DIRECTIVES

6.

The compiler has some optional features. In particular, it may be requested to insert or omit run-time test instructions. Compiler directives are written as comments and are designated by an `$`-character as the first character of the comment:

```
(*$<option sequence> <any comments> *)
```

The option sequence is a sequence of instructions separated by commas. Each instruction consists of a letter, designating the option, followed either by a plus (+) if the option is to be activated or by a minus (-) if the option is to be deactivated.

The following options are available on RC8000:

- l Lists the program text between (`*$l+*`) and (`*$l-*`). Default is (`*$l-*`).
This option may be used for partial listing of a program in the contrary to the list directive of the call (see chapter 7).
- r The code of the procedures between (`*$r+*`) and (`*$r-*`) will during initialization be transferred to core and remain resident during the run (see chapter 8).
- c Lists the generated code for the procedures/functions between (`*$c+*`) and (`*$c-*`), default is `c-`.
The listing may be used for calculations of execution times for the different parts of a program.
- t Includes run-time tests that check
 - all (non constant) array indexing operations, to ensure that the index is within the specified array bounds,
 - all (non constant) assignments to variables of subrange types, to make certain that the assigned value is within the specified range,

- all case statements, to ensure that the case selector corresponds to one of the specified case labels, if no otherwise part is present an empty otherwise part is assumed.

The standard mode is:

include tests for:

- array indexing operations unless the test ought to be superfluous according to the type of the index expression and the index type.

Example:

assume the declarations

TYPE

index_range = 1..6;

super_range = 0..7;

VAR

index : index_range;

super_index: super_range

table : ARRAY [index_range] OF 1..2;

then no code for index check is generated in the following statement

table [index]:= 1;

Indexing with a constant expression is tested at compile time.

- Assignments to variables of subrange types unless the test ought to be superfluous according to the type of the expression and the type of the variable.

Example:

Assume the above declarations, then no code for range test is generated in the statement:

super_index:= index;

- Case statements. If no explicit OTHERWISE part is specified, an empty one is assumed.

The super check mode (t+) is mainly introduced as a debugging tool. The difference between standard mode and t+ mode is the tests for legal values of subrange variables, i.e. uninitialized variables are easily found in t+ mode. The following example may emphasize the usefulness of t+.

```
PROGRAM index_check (output);
TYPE
    index_range = 1..6;

VAR
    table: ARRAY [index_range] OF 1..2;
    index_1: index_range;
    index_2: 2..5;

VALUE
    table=(1,2,1,2,1,2);

BEGIN
    index_1:= index_2;
    CASE table [index_1] OF
    1: write(' odd ');
    2: write(' even ');
    end; (* of case statement *)
    write (' index ' );
END.
```

In standard mode the result may be " index ", because index_2 is uninitialized. In t+ mode this would have resulted in an error message, detected at the line index_1:= index_2; unless the contents of the memory location allocated for index_2 accidentally are a value inside the bounds of index_range.

7. CALL OF THE PASCAL COMPILER

7.

7.1 How to Compile a Pascal Program

7.1

The compiler works in the job process and the compilation is started by means of an FP-command specifying the source text, the compiler options and the file where the resulting object program should end. The result of the compilation is, in case no error is detected, a binary file with code for the procedures/functions and the main program, value segments for value initialization, and procedure table and an information segment; with each of these items occupying an integral number of bs-segments. The object code may be loaded and executed by means of the Pascal Runtime system - see below.

Syntax of the FP-call:

$$\langle \text{object} \rangle =) \underset{0}{1} \text{ pascal}(\langle \text{source} \rangle) \underset{0}{1} (\langle \text{option} \rangle) \underset{0}{\infty}$$

$\langle \text{object} \rangle ::= \langle \text{bs-file name for the generated object code} \rangle$

$\langle \text{source} \rangle ::= \langle \text{text file} \rangle$, if not specified then primary input is assumed.

$\langle \text{option} \rangle ::= \text{list.} \langle \text{on or off} \rangle$
 | heap.<integer>
 | codesize.<integer>
 | survey.<on or off>

$\langle \text{on or off} \rangle ::= \text{yes} \mid \text{no}$

Semantics of the options:

list.yes

produce a program listing on current output, with line numbers added.

Default is list.no

`survey.yes`

Produce a table of the compiled routines and some information about them, for example start line number, size of code, required stack size and some other information necessary for the Pascal system.

Default is `survey.no`

`heap.<int>`

`<int>` is the size of a core area initially assigned to the use of the heap (default is `heap.0`). If a program uses the heap it may be convenient to set the `heapsize` because it may save some execution time.

`codesize.<int>`

`<int>` is the maximum number of instructions which may be generated for the statement part of a "main program", procedure or function. (Default is `codesize.1500`). `<int>` is rounded up to the nearest multiple of 500; the maximum size is 6000.

Storage Requirements

The compiler requires a job with a core area of at least 50000 halfwords. A too small size may cause the compilation to terminate with the alarm 'pascal runtime error: process too small'. A greater core area may remedy the problem.

The compiler uses the following files: 'pascalpif' and 'pascalenv', in addition to current input and output.

How to Run a Compiled Pascal Program

The object code produced by the Pascal compiler may be loaded and executed by the FP-command:

$$(\langle \text{param} \rangle =) \begin{matrix} 1 \\ 0 \end{matrix} \langle \text{object} \rangle \quad (\langle \text{parameters} \rangle) \begin{matrix} n \\ 0 \end{matrix}$$

During execution two area processes are used, one to the Pascal library placed in the file 'pascallib' and one to the object file.

8. RUNTIME ENVIRONMENT

8.

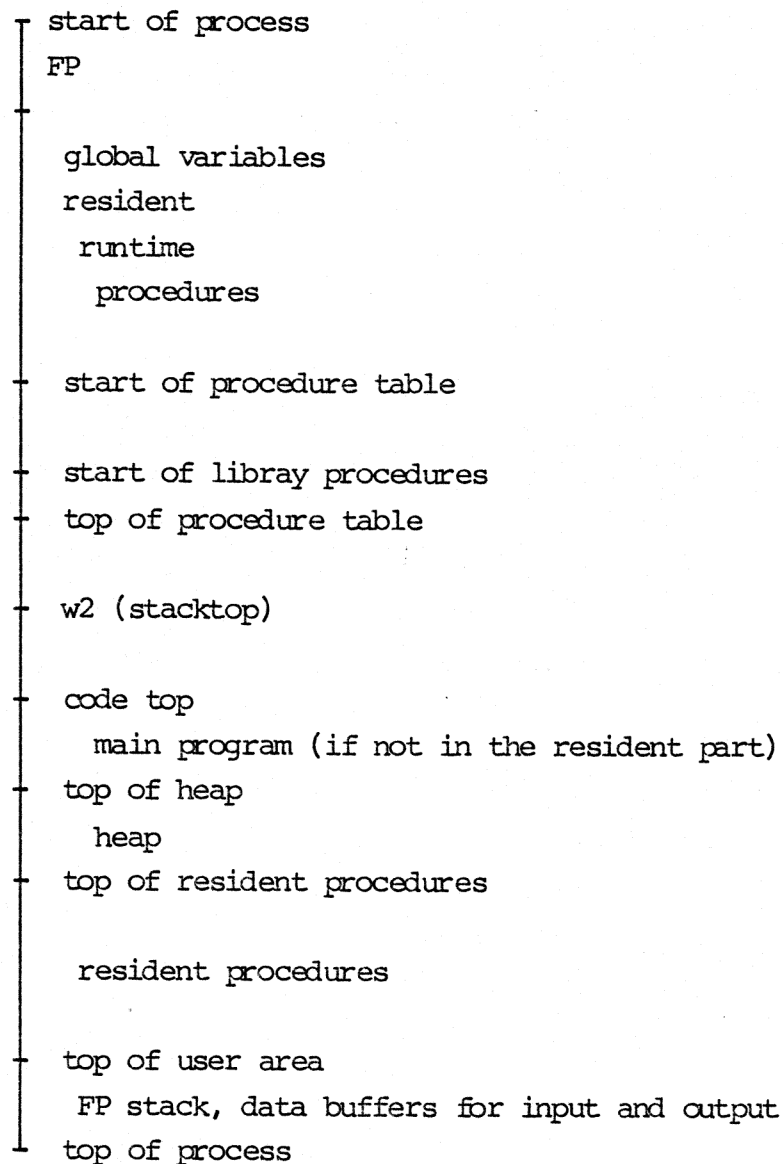
Unsophisticated users should not read all the details of this chapter.

8.1 The Pascal Process at Runtime

8.1

A Pascal object code file contains instructions for initialization of the core area, i.e. loading of the run time system (PASC RUN) and the main program, after loading control is given to the main program.

After the initialization the core image looks like:



A call of a procedure which is not already in core implies a transfer of code from backing storage to core, and the code will be placed inside the area between the code top and the stack top.

The code area is managed by means of a logical segmentation algorithm, handling each routine as one segment.

If enough space is available, the procedure is allocated space from the code top towards the stack top. Else the runtime system will decide which routine(s) to declare 'not in core'. The code area is managed according to a modified round robin strategy.

8.1.1 Resident Procedures

8.1.1

During initialization the procedures declared to be resident are read into core and reside there during the whole run. The resident procedures are placed outside the normal used area for code, therefore these procedures do not influence the former mentioned algorithms.

9. ERROR MESSAGES

9.

Errors in a program are indicated depending of the categori of the error. Compile time error messages are separated into two categories. Errors discovered during the first pass are indicated by an extra line in the program listing, with an uparrow pointing at/after the erroneous item, and a number between 0 and 300 according to the messages given as appendix C.

Errors discovered during the second pass are indicated by: 'error no <int> in line no <int>'.

Examples:

Pass1 error: mis-spelling, i.e. identifier not declared:

```

81.09.14.      15.51.      pascal      version 1981.01.08

  1          PROGRAM show_error( output );
  2          VAR
  3              result : integer;
  4              int     : integer;
  5              alf     : alfaa;
  *****
  6
  7          0      BEGIN
  8          1      result1 := system( 1, int, alf );
  *****
  9          END.
number of errors   :   2
number of warnings:   0

error description
 101: identifier not declared

end
blocksread = 88

```

81.09.14. 15.51. pascal version 1981.01.

```

1        PROGRAM pass2_error( output );
2        VAR
3            int : integer;
4        0 BEGIN
5        1     int := 9000000;
6        END.

```

error no 301 in line no 5

Code: OK + 20 Halfwords
 Error(s) found in pass2
 number of errors : 1

error description
 301: decimal integer constant too large

end
 blocksread = 61

In case of a run time error a message indicating the error is written on current output, and the program is terminated. The line number where the error occurred is written followed by a trace of the active routines.

Example of a procedure for program exit with a trace of the active routines. This may be used as a debugging tool.

81.09.14. 15.52. pascal version 1981.01.08

```

1        PROGRAM runtime_error( output );
2        PROCEDURE stop;
3        0 BEGIN
4        1     writeln(output, ' intentionally stop ... ');
5        2     readln( output );
6        END; (* procedure stop *)
7
8        0 BEGIN
9        1     stop;
10        END.

```

Code: OK + 74 Halfwords

end
 blocksread = 67
 intentionally stop ...

illegal zonestate
 occurred in 5 = line 2 of stop
 called from 10 = line 2 of runtime erro
 blocksread = 8

- 1) There is no check for overflow on integers.
- 2) It is not checked if the tag field of a record with variant part has the correct value when a component of the variant part is referenced.
- 3) Unrestrained use of packed records and arrays may slow down the program execution, because of the many slow shift operations which are required, i.e. the saved space for the variables is paid for in execution time.
- 4) If the variable requirement is so extensive that it is desirable to use packed data types it may be helpful to observe the following advices:
 1. For sequential referencing of the items of a packed array of char (e.g. in a for-statement): operate on an unpacked array and use the standard procedures 'unpack' and 'pack', before and after the referencing.
 2. It is cheaper to use data types occupying an integral number of halfwords, even if it is not necessary, instead of data types of different sizes with only one or two items in each word.
 3. For packed records with items with different storage requirements, the number of shifts may be minimized if the items are declared with descending size.

Example:

<pre>a) t1 : PACKED RECORD bool : boolean; pos_int : 0..maxint; END;</pre>	<pre>b) t2 : PACKED RECORD pos_int : 0..maxint; bool : boolean END;</pre>
--	---

in case a) an assignment to bool will require between 14.2 and 18.6 microseconds just for the shift operations, in case b) the interval is 5.4 to 9.8 microseconds.

An assignment 'boolean_variable := t1.bool' (with boolean_variable and t1 declared on the current level) requires 3 instructions, if t1 was unpacked the same assignment would require 2 instructions.

- 5) The value-statement may be a very convenient construction, but it has some disadvantages:

Initialization of a structure with the same value to almost all the items may be fast but it will require about three words of code per item, and if the number of elements exceed 255 words it will involve a transfer of data from backing storage, which takes about 30 milliseconds, plus 1 msec. for each segment to transfer to core, in that case it will be much faster and less space consuming to use a for-statement.

- 6) Sets are convenient to use for many purposes, but it is a rather expensive construction. The expression 'ch IN ["a", "b", "q"]' will cause 18 words of code. This is due to the fact that a set is always constructed as a 6-word bit map, with each bit indicating if the set element with the corresponding number is in the set (elements are counted from 0).

In the case of 'colour IN [red,blue]' three words of code is saved (the range check) if 'colour' is declared as a subrange of elements, the ordinal values of which lies between 0 and 143. E.g. variables of type ISO or char fulfil this condition.

If the cardinality of 'colour' in the example above is less than 24, then the test for IN only requires 7 words of code.

- 7) The heap is implemented as a stack. This means that each time the standard procedure 'new' is called, a piece of core is allocated on top of the heap. This piece of core is able to contain a variable referenced by the pointer variable used as argument to new.

Dispose(p) work as unstacking. The core inclusive the part referenced by p is released and may now be reused.

- 8) Compile time if-statement (conditional code).

In the statement:

```
IF <const bool expr> THEN st1 ELSE st2
```

code will only be produced for either st1 (if the value of <const bool expr> is true) or st2 (if the value is false).

In the case of:

```
IF <const bool expr> THEN st
```

no code at all will be produced if <const bool expr> is false.

- 9) It is not allowed to 'close' input and output, if they are not connected to external files.

A. REFERENCES

A.

- [1] The Programming Language PASCAL,
Acta Informatica, 1, 35-63, 1971
- [2] ISO Draft International Standard ISO/DIS 7185:
Specification for Computer Programming Language Pascal
- [3] RCSL No 42-i1235:
RC8000 Computer Family, Reference Manual
June 1979, Einar Mossin
- [4] RCSL No 31-D476:
RC8000 Monitor, Part 1, System Design
November 1979, Henrik Sierslev and Pierce C. Hazelton
- [5] RCSL No 31-D477:
RC8000 Monitor, Part 2, Reference Manual
January 1978, Tove Ann Aris and Bo Tveden-Jørgensen
- [6] RCSL No 31-D364:
System 3, Utility Programs, Part one
H. Rischel

B. RC8000 PASCAL SYNTAX DIAGRAMS

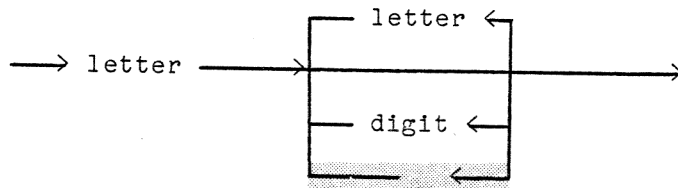
B.

The shaded areas denote differences/extensions in proportion to the report [1].

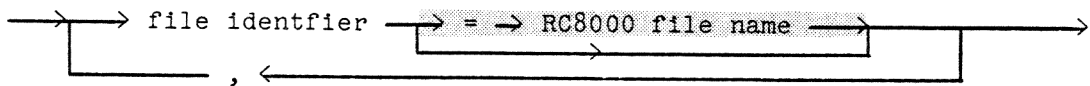
program

→ PROGRAM → identifier → (→ file list →) → ; → block → . →

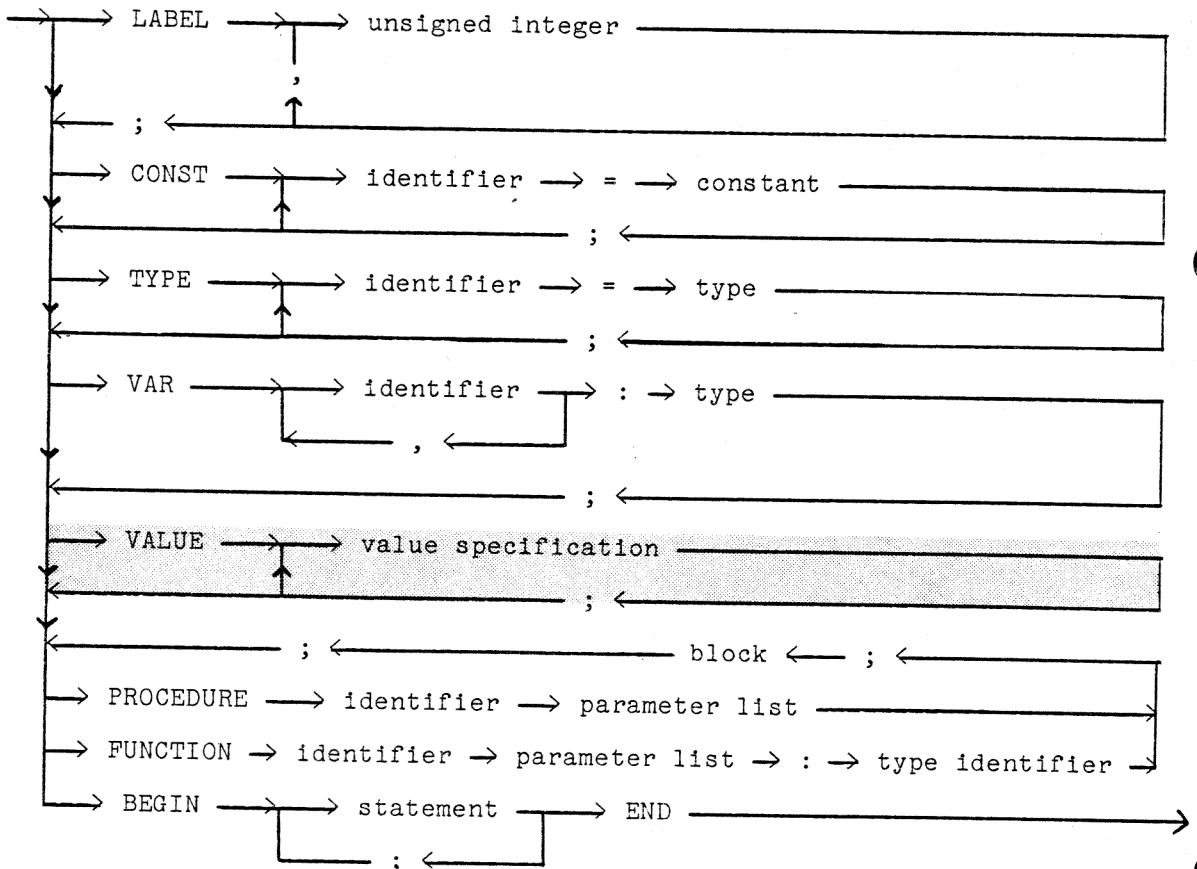
identifier



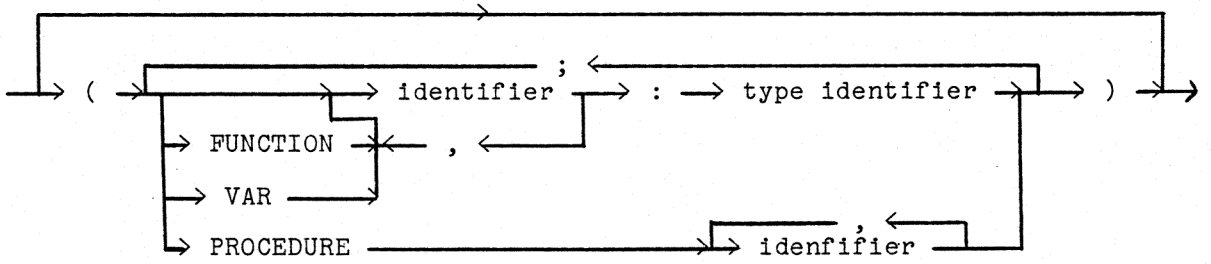
file list



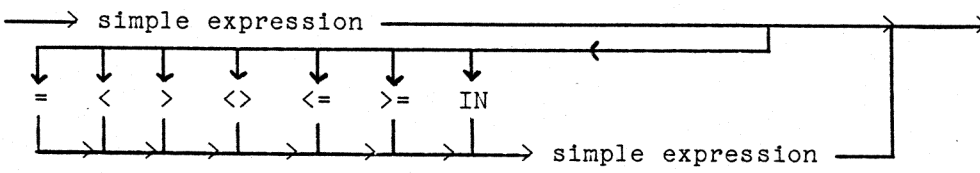
block



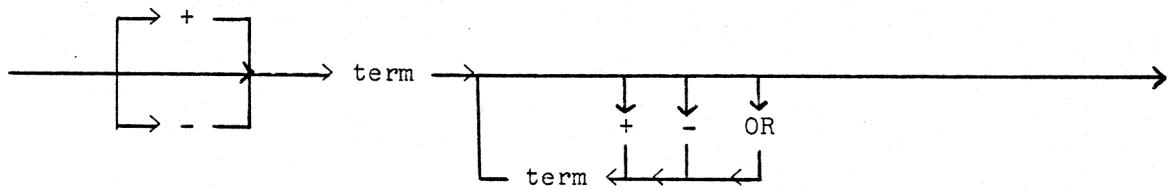
parameter list



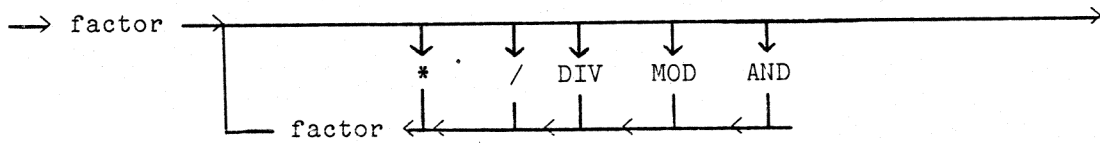
expression



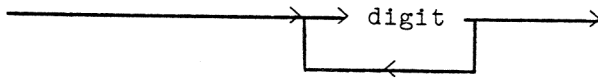
simple expression



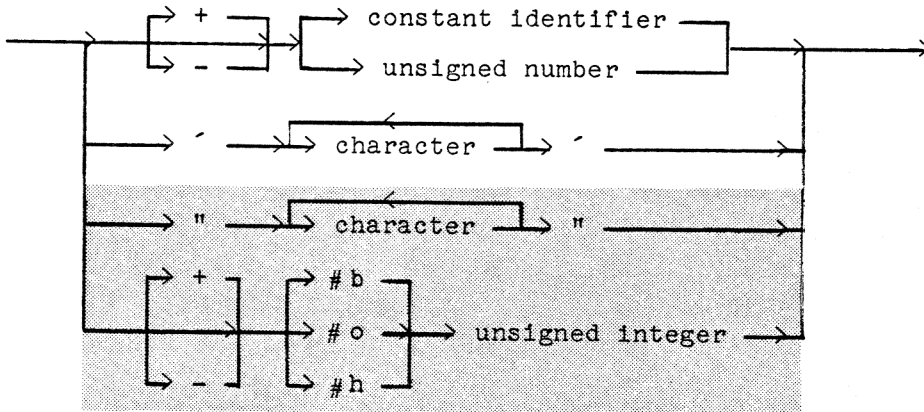
term



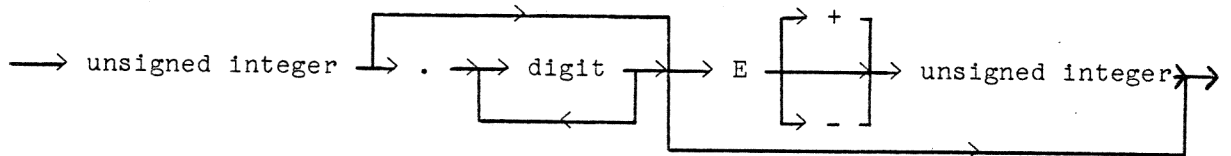
unsigned integer



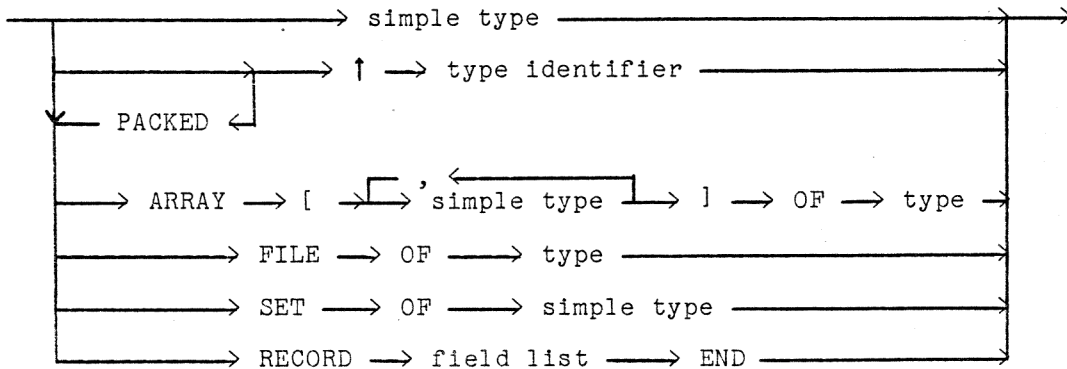
constant



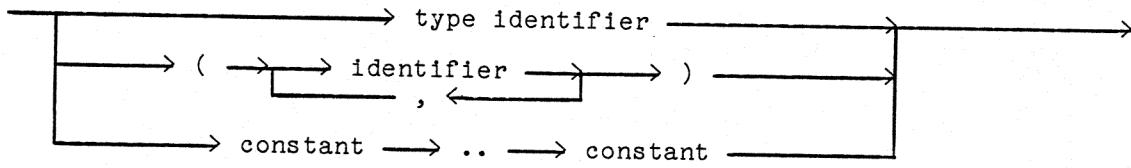
unsigned number



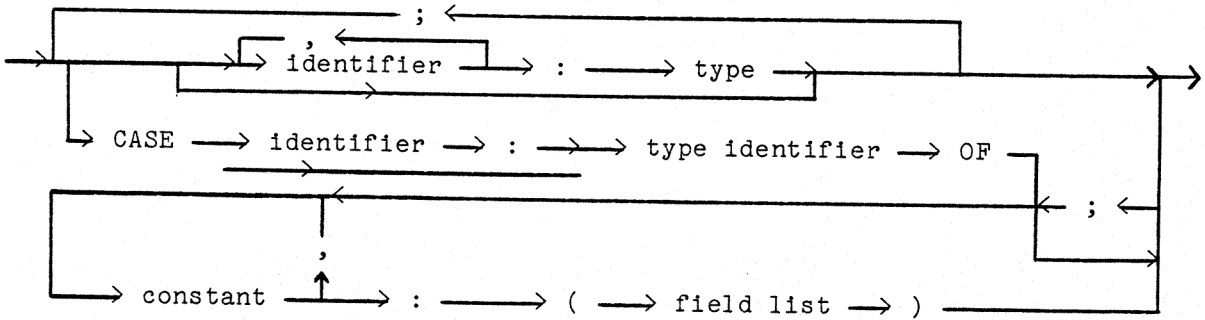
type



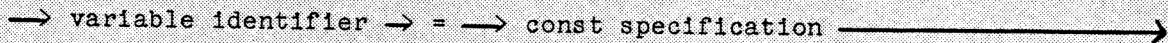
simple type



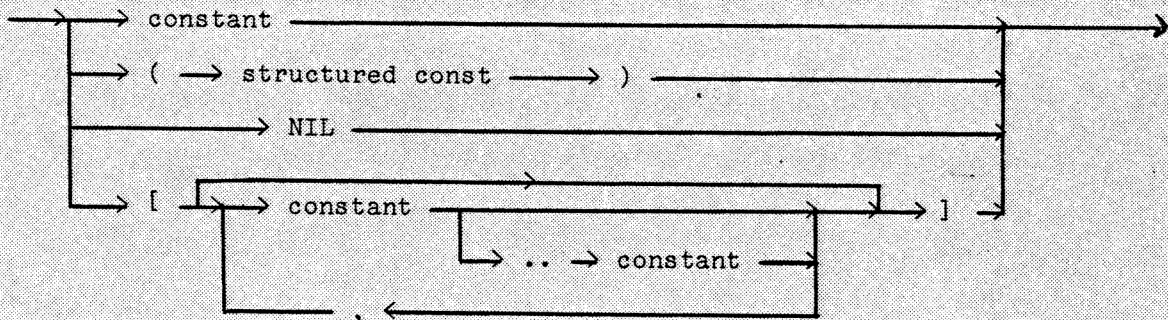
field list



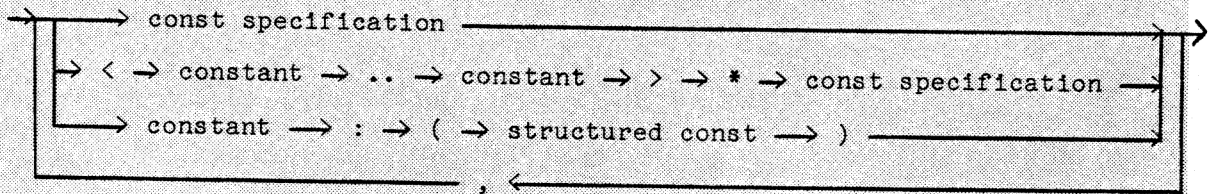
value specification



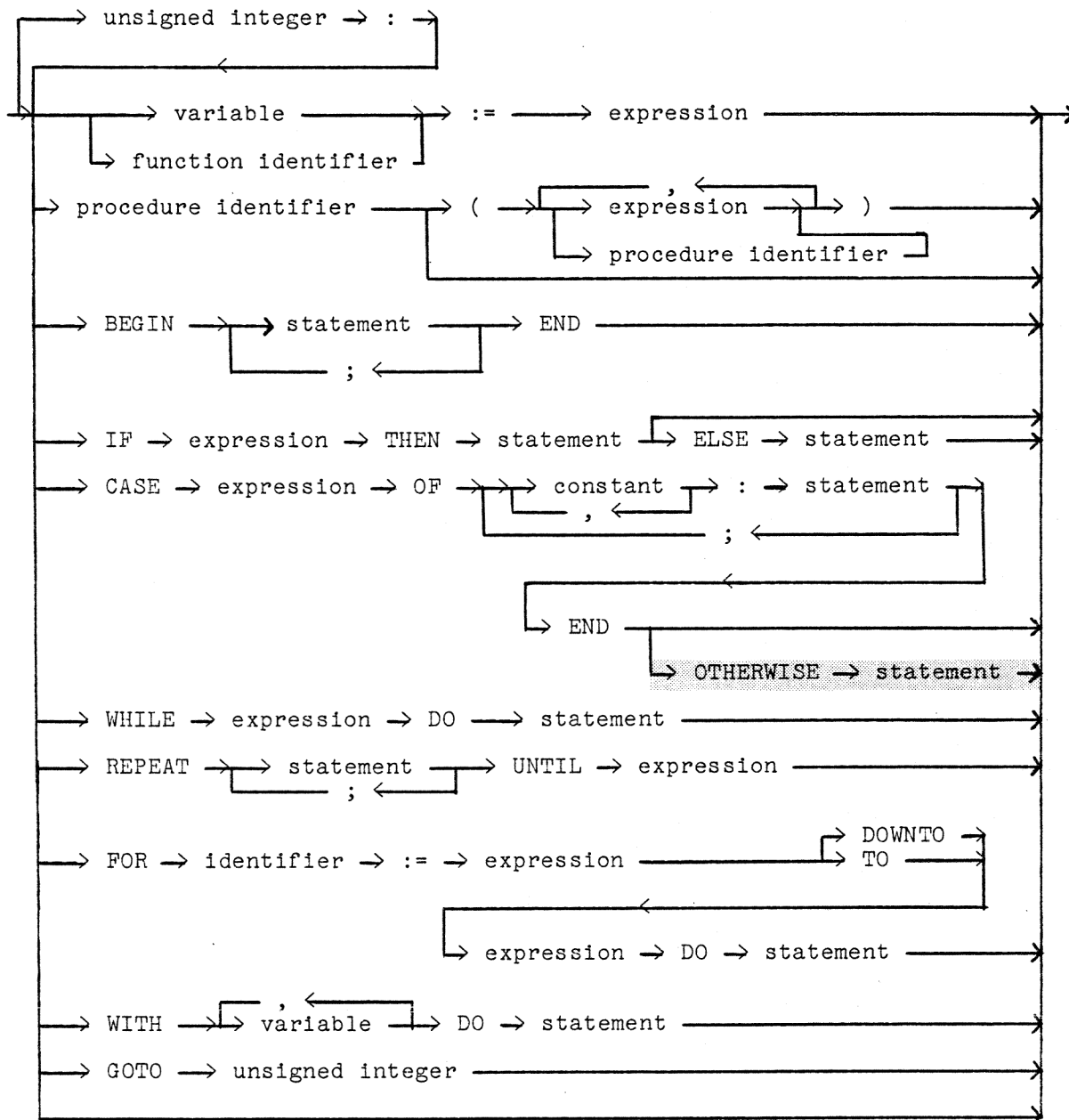
const specification



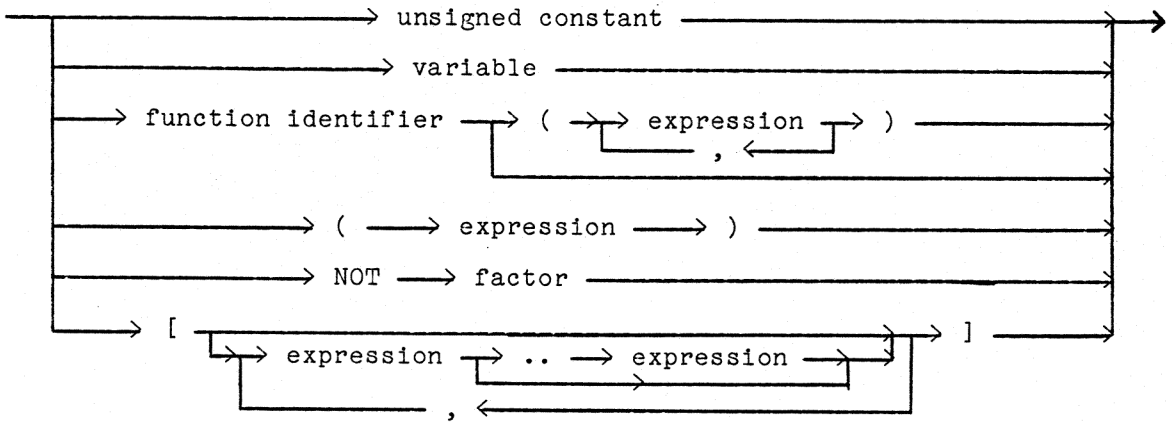
structured const



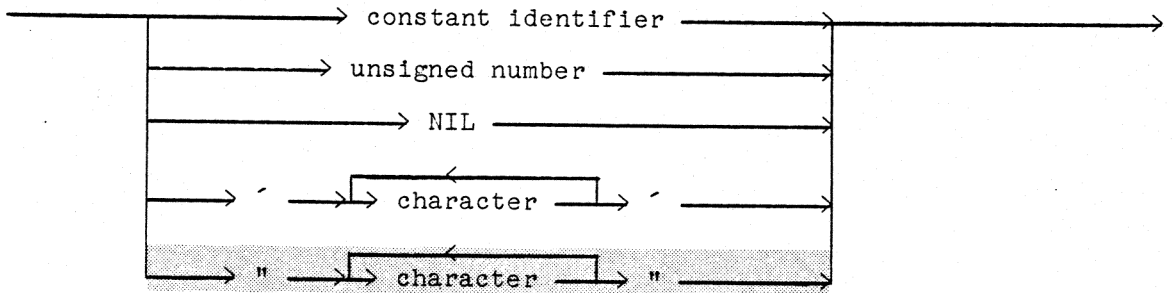
statement



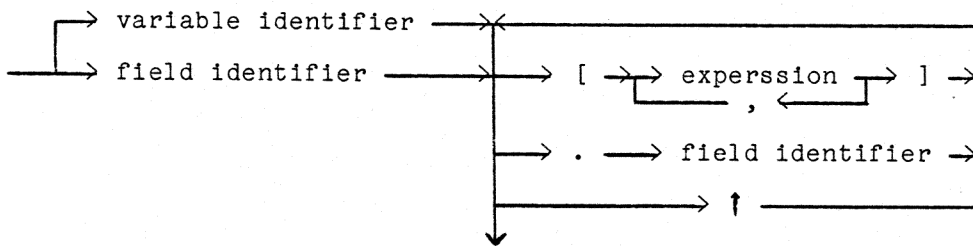
factor



unsigned constant



variable



C. UTILITY PROGRAMS

C.

C.1 Indent (Text Formatting Program)

C.1

The program performs indentation of source programs depending on the options specified in the call and on the keywords (reserved words) of Pascal/PASCAL80.

call:

$$(\langle\text{outputfile}\rangle) \begin{matrix} 1 \\ 0 \end{matrix} \text{ indent } \langle\text{input file}\rangle (\langle\text{option}\rangle) \begin{matrix} \infty \\ 0 \end{matrix}$$

$\langle\text{option}\rangle ::=$	lines	line numbers are added
	mark	the blockstructure is made clear by means of ! between matching begin-ends
	list	the same as: lines mark
	noind	the output will be left justified
	myind	the output indentation is the same as the in- put indentation
	lc	lists keywords in capital letters and ident- ifiers in small (lower case) letters
	uc	both key words and identifiers are listed in upper case letters
	help	produces a list of legal options

Storage requirements:

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

Error messages:

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


Call:

<output file> = cross <input file> (<option>)¹₀

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

<yes or no> ::= yes bosslines are added to the listing.
 (Default).

no only Pascal/PASCAL80 line numbers are
 generated.

Storage Requirements:

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

Error Messages:

- ??? illegal output-filename
 left hand side of the call must be a name
- ??? illegal input-filename
 input file must be specified
- ??? yes or no expected
 option 'bossline' must be 'bossline.yes' or 'bossline.no'
- ??? error in bracket structure, detected at line: xx
 missing ")" ('s)
- ??? error in blockstructure, detected at line: xx
 unmatched END

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

C.3 Use of Indent and Cross

C.3

Indent and cross are two independent programs but a sequence of calls similar to the following will produce a nice listing of a Pascal program with line numbers according to those of the compiler listing, i.e. the numbers used in case of errors.

Example of program calls:

```
10 job jaba 600 time 4 0 size 50000
20 udlist= set 0
30 sourcelist= indent source mark lc
40 udlist= cross sourcelist bossline.no
50 convert udlist
60 finis
```

The contents of source and output from the job are shown on the following pages.

Contents of source.

```

program test_listing(output);
label
7913;
const
first = 1; last = 25;
type
structure = record
field1, field2 : real;
random_field : integer;
name_field : alfa;
case_cheat : boolean of
true : ( name_conv : alfa );
false : ( int1, int2, int3 : integer );
end;
var
random_help, help : integer;
very_long_identifier_name : alfa;
table : array # first .. last # of structure;
value
table = (<first .. last> * (0.0, 1.0, 13, 'abcdef',
true : ( ' ' ) ) );

function random_number : integer;
(* generate a pseudo random number sequence *)
begin
random_number := (random_help * 1023) mod last + 1;
end;

begin
random_help := 13;
(* .
.
.
*)
for help := first to last do
with table # help # do
begin
random_field := random_number;
end;
(* .
.
.
*)
7913:
end.

```

source1ist 81.09.15. 10.49.

```

1 PROGRAM test_listing(output);
2 LABEL
3 7913;
4 CONST
5 first = 1; last = 25;
6 TYPE
7 structure = RECORD
8 ! field1, field2 : real;
9 ! random_field : integer;
10 ! name_field : alfa;
11 ! CASE_cheat : boolean OF
12 ! true : ( name_conv : alfa );
13 ! false : ( int1, int2, int3 : integer );
14 ! END;
15
16 VAR
17 random_help, help : integer;
18 very_long_identifier_name : alfa;
19 table : ARRAY #first..last A OF structure;
20 VALUE
21 table = (<first..last> * (0.0, 1.0, 13, 'abcdef',
22 true : (
23 ) ) );
24
25 FUNCTION random_number : integer;
26 (* generate a pseudo random number sequence *)
27 BEGIN
28 ! random_number := (random_help * 1023) MOD last + 1;
29 END;
30
31 BEGIN
32 ! random_help := 13;
33 ! (*
34 ! .
35 ! *)
36 ! FOR help := first TO last DO
37 ! WITH table # help A DO
38 ! BEGIN
39 ! ! random_field := random_number;
40 ! ! random_help := 13;
41 ! ! (*
42 ! ! .
43 ! ! *)
44 ! ! 7913;
45 ! ! END.

```


(Contents of udlister contd.)

page 3

sourcealist 81.09.15. 10.49.

ARRAY	1
BEGIN	3
CASE	1
CONST	1
DO	2
END	4
FOR	1
FUNCTION	1
LABEL	1
MOD	1
OF	2
PROGRAM	1
RECORD	1
TO	1
TYPE	1
VALUE	1
VAR	1
WITH	1

Because of the software managed program segmentation on routine level it is possible to gather statistical information for a Pascal program during execution, without extra statements in the program and without special compilation. The running system is provided with two sets of code for call and exit management. The standard action is without gathering information for statistics. The statistical version is chosen if the FP mode bit 'listing' is set (mode listing.yes). At program end the measurement is tabulated as shown below.

Each table entry contains information as:

routine name (first entry is for the main program), begin-line, number of times the routine has been called and some time consumption informations. It should be noted that the time information is real time (not CPU-time). This means that swap out and backing storage transfer time is accounted and hence may disturb the result. The reason why real time is measured instead of CPU-time is based upon experience showing that input and output operations very often constitute the greater part of the program execution time, and this would not be seen from CPU-time measurements.

Performance measurement summary for PASCAL program :

Name	Line	Called	% of calls	Average (sec)	Total (sec)	% of time
pascal cross	623	1	0.058	1.0094	1.0094	18.716
insert id	325	123	7.197	0.0017	0.2178	4.038
init	189	1	0.058	0.8836	0.8836	16.384
error	180	0	0.000	-----	0.0000	0.000
newpage	247	3	0.175	0.0312	0.0936	1.735
checkbracket	260	30	1.755	0.0009	0.0290	0.537
nextsymbol	273	1056	61.790	0.0017	1.8075	33.515
add to id	464	459	26.857	0.0013	0.6146	11.396
sort table	500	1	0.058	0.1280	0.1280	2.373
print table	559	1	0.058	0.3777	0.3777	7.003
writealfa	550	34	1.989	0.0068	0.2321	4.303
Totals:		1709		0.0031	5.3933	

D. ERROR MESSAGES

D.

D.1 Error Messages from First Pass

D.1

number	meaning
000	illegal character
001	'program' or 'module' expected
002	identifier expected
003	error in parameter list
004	identifier expected
005	':' or ',' expected
006)' or ';' expected
007	;' expected
008	digit expected
009	;' or ',' expected
010	digit expected
011	'=' expected
012	constant expected
013	unsigned constant expected
014	error in declaration
015	'file' expected
016	'E' expected
017	<type> expected
018	..' expected
019)' or ',' expected
020	'A' expected
021	'of' expected
022	,' or 'A' expected
023	unpacked structured type expected
024	'(' expected
025)' expected
026	'end' expected
027	<const specification> expected
028	<set const element> expected
029	'A' or ',' expected
030	<str const element> expected
031	'>' expected
032	'*' expected
033	'module' expected
034	'pascal' or 'fortran' expected
035	'end' or ';' expected
036	'begin' expected
037	':=' expected
038	<simple expression> expected
039	expression expected
040	expression expected
041	expression expected
042	'to' or 'downto' expected
043	'do' or ',' expected
044	'do' expected
045	'then' expected

046 `:` expected
047 `else` expected
048 `until` or `;` expected
049 `.` expected
050 string expected
051 end of file expected

100 error in real constant: digit expected
101 identifier not declared
102 identifier declared twice
103 illegal integer constant
104 incompatible subrange types
105 subrange bounds must be scalar
106 index type must be scalar or subrange
107 not a type
108 illegal type
109 only tests on equality allowed
110 illegal pointer type
111 type of variable is not record
112 no such field in this record
113 previous declaration was not `forward`
114 too many digits in label
115 multideclared label
116 illegal value name
117 not a variable
118 type of variable must be file or pointer
119 type of variable is not array
120 index type is not compatible with declaration
121 type of variable must be boolean
122 incompatible set element types
123 illegal set element type
124 type conflict of operands
125 illegal type of operand(s)
126 file comparison not allowed
127 strict inclusion not allowed
128 not a function
129 undeclared label
130 illegal type of expression
131 number of parameters does not agree with declaration
132 illegal parameter substitution
133 actual parameter must be a variable
134 not a procedure
135 incompatible with tagfield type
136 label type incompatible with selecting expression
137 type of expression must be boolean
138 unsatisfied forward pointer reference
139 function type does not correspond to the forward declaration
140 parameter list does not correspond to the forward declaration
141 undeclared external file

- 142 already forward declared
- 143 error in option
- 144 missing file 'output' in program heading
- 145 unsatisfied forward function/procedure declaration
- 146 undefined label(s)
- 147 multidefined label(s)
- 148 array elements out of sequence
- 149 no variant part in this record
- 150 erroneous number of fields in this record
- 151 valuespecification incompatible with recorddeclaration
- 152 number of array elements does not agree with declaration
- 153 multiple occurrence of variable in value part
- 154 illegal formatting
- 155 module name(s) must be unique
- 156 assignment to function not allowed at this level
- 157 illegal procedure call
- 158 only 'value' parameter(s) allowed in formal function/procedure
- 159 control variable must be a variable or a parameter
- 160 multidefined external file
- 161 'input' not in program heading
- 162 'input' has illegal type
- 163 readln and writeln only allowed on text files
- 164 not a constant
- 165 not an external declared file
- 166 assignment to function identifier must occur in function itself
- 167 textstring not terminated within the same line
- 168 file parameter must be VAR-parameter
- 169 comment did not terminate

number	meaning
301	decimal integer constant too large
302	non-decimal integer constant too large
303	exponent in real constant too large
304	index type too large
305	basetype of set too large
306	too many nested function/procedure declarations and/or too many parameters/labels in this procedure
307	first element in subrange specification less than second
308	multideclared label
309	the lowest integer is not allowed as case-label
310	the range of case-labels is too large
311	not enough room for temporaries
312	constant out of subrange bounds
313	comparison and assignment of strings with different length not fully implemented yet
314	not enough room for parameters, structure too complicated
315	range of set-elements only with constant bounds
316	tag field values must be scalar
317	no such tag field in this record
318	too many tag fields specified
319	standard routine argument too complicated in this context
320	erroneous arguments to pack or unpack
321	***** warning: label may lead to erroneous code
322	standard procedure 'replace' may only be called from main program
323	packed fields not allowed as var-parameters
324	division by zero not allowed
401	compiler constant 'maxident' too small
402	compiler constant 'stringmax' too small
403	compiler error (should be reported to maintenance staff)
404	compiler constant 'maxnest' too small
405	too much code: use option 'codesize'
406	random files not implemented
407	read and write of user defined scalars not implemented
408	pack and unpack only implemented on array of char

D.3 Runtime Error Messages

D.3

D.3.1 Start Up Errors

D.3.1

During the start up (initialization) of the running program some error messages may appear.

The error message consists of two lines:

```
*** pascal init trouble  
W0 = <status> <message>
```

<status> is the result delivered by some monitor calls causing the error.

<message> may be one of the following:

```
'cannot create area process'  
the job is run with too few area processes.
```

```
'error in program call'  
the call to get a compiled program executed is wrong.
```

```
'wrong answer'  
the object file is not ok. It cannot be loaded or it is not  
possible to read from it.
```

```
'process too small'
```

D.3.2 Errors During Program Execution

D.3.2

During the execution of a Pascal program the program may be terminated by a runtime error. Runtime error messages consist of a message and a trace of the active routines (see the example in chapter 9).

The messages are:

'b, o or h expected'

during the reading of a number with base 2, 8 or 16 a wrong base has been encountered.

giveup, blocklength = <integer>

possibly because of too few bs-resources.

'digit expected'

during the reading of a number an erroneous character has been encountered.

'dispose outside used area'

the reference used as argument to dispose is outside the used area of the heap.

'file cannot be connected for I/O: <file name>'

an external file cannot be used, maybe because the job is run with too few area processes.

'file does not exist: <name>'

'illegal argument to arcsin'

the argument to arcsin has an absolute value greater than or equal to 1.0.

'illegal argument to exp or sinh'

exp or sinh has been called with a too big argument.

'index or subrange out of bounds, value is: <value>'

'integer overflow'

during input an integer greater than maxint (8388607) has been read.

'negative argument to ln or sqrt'

'negative field width'

it is tried to write a number with a negative number of significant digits.

'process too small'

the program cannot be executed in a process with the size used.

'illegal zonestate'

illegal use of a file:

read before reset or write before rewrite.

'illegal pointer value'

'try to read past eof'

during input EM has been encountered.

'wrong answer on input request'

a procedure cannot be transferred from backing storage to core (if no hardware problems it should be reported to the maintenance staff).

'wrong no of halfwords transferred'

a procedure cannot be transferred from backing storage to core (if no hardware problems it should be reported to the maintenance staff).

Uncontrolled runtime error.

Use of an undefined pointer variable (uninitialized) may cause a

*** break 0 <address>

RETURN LETTER

Title: RC8000 PASCAL, User's Guide

RCSL No.: 42-i1786

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

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

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

How can this manual be improved?

Other comments?

Name: _____ Title: _____

Company: _____

Address: _____


Date: _____

Thank you

..... Fold here

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

Affix
postage
here

 **REGNECENTRALEN**
af 1979

Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark