PASCAL MANUAL

by

Wilhelm Franz Burser

July 1973 TR-22

TR 73-22, B & C

Revised December 1973

Revised May 1974

This work was supported in part by the National Science Foundation Grant GJ-36424.

Technical Report No. 22
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Preface

This PASCAL Manual is an introduction to the programming language PASCAL which requires little or no knowledge of programming. In later chapters some features of the PASCAL implementation on the CDC 6600 at the University of Texas at Austin are included. Thus it can also be used as a Reference Manual.

The PASCAL implementation at the University of Texas at Austin originates from the Eidgenoessische Technische Hochschule, Zurich, Switzerland, and corresponds to the Revised Report on the Programming Language PASCAL with the exception of packed arrays and class structures. Packed arrays are not implemented and the class structure has been retained from the previous definition of PASCAL. Some important programming facilities have been added:

Overlays, partial compilation, limited access to external routines, symbolic trace and dump features, and more flexible I/O capabilites.

This PASCAL Manual is also available in machine readable form.

Acknowledgement

The author would like to thank Dr. J. Browne for support, Dr. D. Musser for suggestions and ideas, Dr. R. Bartels and Eileen Josue for reading the draft, and the Computation Center for providing the equipment to produce the final copy.

Table of Contents

Table of Contents (continued)

Introduction

PASCAL is a very easy programming language to learn and yet powerful enough to express sophisticated programs.

When a programmer writes a program he must first consider the data he has to deal with, and then he can formulate algorithms.

PASCAL gives the programmer the possibility of defining his own data structures, so that he can treat his problem in a familiar way. In order to write algorithms logically and convincingly, convenient means for writing the control structure of a program are necessary. These means are provided by PASCAL. They are together with the procedure concept quite natural to the thinking of the programmer.

Input and output are also handled efficiently.

This manual is organized in four parts:

The first part deals with the elements of the language, simple data structures and the basic structure of a program. After reading this part the programmer can already write programs to solve complicated problems.
The second part is dedicated to procedures, functions, and input/output
The third part introduces more data structures.
The fourth part treats some special aspects of the language.

AND NOW GOOD LUCK . . .

PART I

1. Introduction to the Language

1.1 Elements of the Language

On reading this chapter for the first time the sections on identifiers and numbers are of primary interest.

PASCAL programs consist of the following elements:

- Identifiers
- Numbers
- Strings
- Reserved words
- Operators and delimiters

The character set which is available on this machine is in Appendix A.1.

1.12 Identifiers

A sequence of letters and digits which begins with a letter is considered an identifier unless it is a reserved word (Section 1.15).

Example: A123 JOHN

(1AX is not an identifier, as it starts with a digit.)

An identifier can be chosen freely by the programmer for naming purposes. Some identifiers with a predefined meaning are provided by the language. They are listed according to their category in Appendix A.2.

If an identifier is longer than 10 characters, only the first 10 characters are considered, i.e. the identifiers

VERYLONGIDENTIFIER
VERYLONGID

are not distinguished.

1.13 Numbers

Numbers are either of type INTEGER or REAL. Integers may be written in decimal or octal notation. Real numbers are written with a decimal point and/or a scale factor. All numbers can be preceded by a sign + or -.

Integers:

In decimal notation they consist of a sequence of decimal digits. Decimal digits are the digits 0 to 9. The magnitude may not exceed 281474976710655 ($2^{48}-1$).

Example: 123

In octal notation, integers consist of at most 20 octal digits, followed by the letter B. Octal digits are the digits 0 to 7.

Example: 7771200B

Real numbers:

Let x, y be nonempty sequences of decimal digits, and z be a nonempty sequence of at most three decimal digits, which may be preceded by a sign + or -. Then all numbers of type REAL can be expressed in one of these forms:

(a) x.y
(b) x.yEz (represents x.y times 10^{z})
(c) xEz

Examples:

(a) 123.3
0.123

(b) 123.3E+4 (represents 123.3 times 10^{4} = 1233000.0)
0.123E111

(c) 29E-54

(Note that .123 and 123. are not permitted.)

1.14 Strings

A string consists of a nonempty sequence of characters from the character set. The sequence is enclosed in quote symbols.

Example: ΞHELLOΞ

If the quote symbol is part of the string it must be followed by another quote symbol.

Example: Ξ+*-ΞΞ+*-Ξ represents the string +*-Ξ+*-

ΞΞΞΞ represents the string Ξ

1.15 Reserved Words

The language PASCAL contains the following reserved words:

```
IF  DO  TO  OF  IN
END  NIL  FOR  DIV  MOD  VAR  SET
THEN  ELSE  GOTO  CASE  WITH  TYPE  FILE
BEGIN  UNTIL  WHILE  ARRAY  VALUE  CLASS  CONST  LABEL
REPEAT  DOWNTO  RECORD  PACKED
FUNCTION  PROCEDURE
```

They may not be used as identifiers.

1.16 Operators and Delimiters

Most of the punctuation symbols are used as operators or delimiters. The following operators consist of two symbols:

```
:=
..
<>
```

1.17 Comments

A sequence of characters (not containing ↓) which starts with ↱ and finishes with ↓ is a comment.

Example: ↱ THIS IS A COMMENT ↓

A program consists of a proper sequence of the elements of the language. Blanks and comments may be inserted between the elements of the language to improve readability.

1.2 Writing a PASCAL Program

Programs are punched on cards in the first 72 columns. Columns 73-80 are ignored by the compiler. The card boundary is considered to be a blank, thus no elements of the language may cross a card boundary (except comments).

Control Cards will be discussed in Appendix B.

2. Simple Data Structures, Declaration Part of a Program

2.1 Introduction

Data are treated in a program by means of variables. A variable has a name and a type. The type describes the type of the value which this variable may assume.

Before a variable is used it must have been named with an identifier and a type assigned to it. This is done in the declaration part of the program.

Example:
```
VAR  I,K : INTEGER;
       R : REAL;
```

With this declaration the 3 variables with the names I, K, R are introduced, their type is INTEGER, INTEGER, and REAL respectively.

Later we shall see that the declaration part is not only used to name variables, but also to name types, constants and even algorithms.

2.2 Types

A type may be thought of as a set of values, such as the set of integers. If the values of the set are structured, we have structured types, otherwise we talk of simple types.

2.2.1 Simple Types

A simple type is made up by a set of values on which an ordering relation is defined. The set contains a smallest and a largest value.

2.2.11 Predefined Types

INTEGER	This is the set of integers in the range $-(2^{48}-1) \ldots (2^{48}-1)$
REAL	$-10^{+322} \ldots -10^{-294}, 0, 10^{-294} \ldots 10^{+322}$
BOOLEAN	This set contains two values, FALSE and TRUE. The ordering is FALSE < TRUE.

CHAR — The values of this set are the strings with one character and the value EOL (End Of Line). The set is ordered according to the octal representation of the characters. EOL corresponds to 0.

ALFA — The values of this set are all sequences of 10 characters of the character set which includes EOL. The values are ordered according to their octal representation.

NOTE: When a string is used as a constant of type ALFA then it is adjusted to 10 characters either by using the first 10 characters or by filling in blanks (55B) on the right.

2.2.12 Scalar Types

Sometimes it is desirable to order things by attribute rather than by number. The programmer can define a new type by specifying a set of attributes, e.g.

(RED, GREEN, BLUE, WHITE, ORANGE)

The set is ordered according to the order of writing, i.e.

RED < GREEN < BLUE < WHITE < ORANGE

(The values of the set are actually represented during program execution by the integers 0,1,2,3,4.)

We may declare a variable of this type as in the following example:

Example: `VAR PAINT : (RED, GREEN, BLUE, WHITE, ORANGE);`

The variable PAINT will assume values of the type for which it was defined, e.g.

`PAINT := GREEN;`

As we will see later it is often useful to have a name for a programmer defined type:

Example: `TYPE COLOR = (RED, GREEN, BLUE, WHITE, ORANGE);`

A variable can be declared with the type name rather than with the explicit specification of the type:

Example: `VAR PAINT : COLOR;`

2.2.13 Subrange Types

Since the set of a simple type is ordered we can specify a subrange by indicating a lowest and a highest value. This subrange may then be taken as the set for a new type:

Example:
```
VAR SHORTINTEGER : 0..3;
    UT : WHITE..ORANGE;
```

Of course, we can name this programmer defined type:

Example:
```
TYPE SHORT = 0..3;
     FIRST = WHITE..ORANGE;
VAR  SHORTINTEGER : SHORT;
     UT : FIRST;
```

NOTE: When we talk about simple types in the remainder of the manual the predefined types REAL and ALFA are not included. They will be especially mentioned if necessary.

2.2.2 Structured Types

Structured types are introduced by the programmer by specifying the structure and the types of its components. In this section we only treat arrays and records.

2.2.21 Array Type

An array consists of a fixed number of components, all of which have the same type. An ordering of the components is induced by mapping them onto a simple type. Then each component can be indexed throush a value of that simple type. Thus we have to specify the type of the index and the type of the components.

Example:
```
VAR A: ARRAY[1..10] OF INTEGER;
```

The array A has 10 components, each is of type INTEGER. In order to select a component the index must be a value of the set 1..10:

```
A[1] := 100;
A[X+Y] := 5;
```

where we assume that the expression X+Y will yield a value in the range 1..10.

Example:
```
TYPE COLOR = (RED, GREEN, BLUE, WHITE, ORANGE);
VAR X : ARRAY [BLUE..ORANGE] OF BOOLEAN;
```

We can assign a value to a component by:

```
X[WHITE] := TRUE;
```

So far we have discussed only 1 dimensional arrays. A multi-dimensional array is defined by specifying n index types:

Example:
```
VAR THREE : ARRAY [1..10,1..20,BOOLEAN] OF ALFA;
```

When we use a component of this array we need three index expressions as shown in the following assignment:

```
THREE[3,5,F>G]  := ΞABCDEFΞ;
```

NOTE: The number of components of an array cannot be varied dynamically by the program.

2.2.22 RECORD Type

A record differs from an array in that it can have components of different types. For indexing purposes we create a name for each component, the so-called field name. Further we must specify the type of each component.

Example:
```
TYPE DATUM = RECORD DAY:   1..31;
                    MONTH: 1..12;
                    YEAR:  0..2000;
             END;
VAR V: DATUM;
```

The component with name DAY of the variable V is selected as follows:

```
V.DAY := 8;
```

NOTE: The record structure definition is enclosed in

```
RECORD
  .
  .
  .
END;
```

Now we can also create arrays of records:

Example:
```
ABC : ARRAY [-3..20] OF DATUM;
```

An assignment could be of the form

```
ABC[-1].YEAR := 2000;
```

If several components have the same type, then we can use a list of names and indicate the type only once:

Example:
```
VAR X : RECORD  LENGTH,HEIGHT: REAL;
                NUMBER: INTEGER;
        END;
```

Records will be discussed again in Part III.

2.3 Declaration Part of a Program

So far we know how to name variables and types. There is also the possibility of naming constants. The identifier used for the name is a synonym for the constant.

Example: `CONST K = 3; BLANK = ' ';`

This feature is especially helpful when we do not want to think in numbers, as in this example:

```
CONST LIMIT = 5;

VAR   X : ARRAY[1..LIMIT] OF INTEGER;
```

Finally we can name a sequence of statements which then is called a procedure or function. They are treated in Part II.

The "naming" is done in the order

Constants	CONST
Types	TYPE
Variables	VAR
(Procedures and Functions)	

Each section starts with a keyword as indicated above in the right column.

NOTE: The constant declarations are separated by semicolons. Variables which are separated by commas are assigned the same type.

Example of a declaration part of a program:

```
CONST
        MIN=3; MAX=10; LIMIT=100;

TYPE
        MATRIX = ARRAY[1..MAX,1..LIMIT] OF REAL;
        SHORT  = 1..MIN;
        CHOICE = (NO, YES, MAYBE);

VAR
        K,L,M : INTEGER;
            S : REAL;
        F1,F2 : SHORT;
            B : CHOICE;
          A,X : MATRIX;
            R : RECORD   TRY: CHOICE;
                       MONEY: SHORT;
                END;
```

NOTE: All names must be declared before they are used.

3. Basic Structure of a PASCAL Program

3.1 Statements

The program part of a PASCAL program consists of a sequence of statements separated by semicolons. This sequence is enclosed in

```
BEGIN
.
.
END .
```

NOTE: A period follows the END.
A semicolon may be written before the END.

In the following we describe most of the statements which the language offers. Statements which in turn may contain statements are called structured statements; they all start with a keyword. Most of them are named after their keyword and thus easy to remember. The other statements are simple statements. Here we treat the assignment statement and the procedure statement. Statements in a sequence of statements are always separated by semicolons.

3.1.1 Assignment Statement

We have already seen examples of the assignment statement. It is of the form

```
VARIABLE  :=  EXPRESSION
```

The value which is obtained through the evaluation of the expression (which may involve some arithmetic operations) must be in the set for which the variable is defined. (The assignment of a value of type INTEGER to a variable of type REAL is allowed.)

Example:
```
F := (A+B) * 3;
```

3.1.2 Procedure Statement, READ, WRITE

The procedure statement consists of an identifier which designates the procedure. This identifier may be followed by a list of parameters.

Example:
```
DIFFER(A,B,C);
```

In this section we treat only the built-in procedures READ and WRITE.

READ is used to read values of type INTEGER, REAL and CHAR from the input medium.

Example:
```
VAR A,B: INTEGER; R: REAL; CH: CHAR;
BEGIN  READ(A,B,CH)  END.
```

Only variables of type INTEGER, REAL or CHAR are allowed in the parameter list of the procedure READ.

NOTE: The character sequences on the input medium denoting integers and real numbers must be followed by at least one blank. Octal numbers are not permitted.

Expressions are printed with the procedure WRITE. The value of an expression must be of type INTEGER, REAL, BOOLEAN, CHAR, or ALFA.

Example:
```
WRITE(X+3,A,'MILES',EOL);
```

NOTE: The character EOL (End Of Line) finishes a line and starts a new one. The first character written on the new line is interpreted as printer control and thus not printed. For the printer control characters see Appendix A.3.

Strings with more than 10 and less than 70 characters may also be parameters of the procedure WRITE.

Example:
```
WRITE(' PROBLEM 1                JOHN DOE');
```

Now we are in the position to formulate our first complete PASCAL program:

```
VAR X : INTEGER;
BEGIN   ↑A VERY SIMPLE PROGRAM↓
        X := 1;
        X := 3*(X+1);
        WRITE(' ',X,EOL);
END.
```

3.1.3 Compound Statement

A sequence of statements can be made into one statement by enclosing the sequence in

```
BEGIN . . . END
```

3.1.4 Repetitive Statements

It is in the nature of most algorithms to execute certain sequences repeatedly. We can distinguish three kinds of repetitions:

(a) We repeat according to a condition. This condition is tested each time
    (a1) at the beginning of the sequence
    (a2) at the end of the sequence.
(b) We know the exact number of repetitions.

PASCAL provides a statement for each kind of repetition:

The WHILE statement for (a1)
The REPEAT statement for (a2)
The FOR statement for (b)

3.1.41 WHILE Statement

It has the form
WHILE condition DO statement

The condition is an expression which must yield a value of type BOOLEAN.

Example:
```
A := 1;
WHILE A < 3 DO A := A+1;
WRITE(A);
```

The condition is evaluated. If it is false then the statement is not executed, otherwise the statement is executed repeatedly until the condition becomes false.

The example above would print the value 3 .

3.1.42 REPEAT Statement

It has the form
REPEAT statement-sequence UNTIL condition

The condition is an expression which must yield a value of type BOOLEAN.

Example:
```
A := 1;
REPEAT   A:= A+1;
         WRITE(A);
UNTIL A > 3;
```

The statement-sequence is executed repeatedly until the condition becomes true. As the condition is tested at the end, the statement-sequence is executed at least once.

The example above would print the values 2 3 4 .

3.1.43 FOR Statement

It has one of the two forms:

(a) FOR control-variable := initial-expression
TO final-expression DO statement

(b) FOR control-variable := initial-expression
DOWNTO final-expression DO statement

Example:

```
FOR A := 1 TO 3 DO S := S+X[A];

FOR A := 3 DOWNTO 1 DO S := S+X[A];

FOR C := RED TO BLUE DO
    BEGIN WRITE(ARTICLE[C].COST);
          S := S+QUANTITY[C]*ARTICLE[C].COST;
    END;
```

The control-variable must be of simple type. The initial-expression and the final-expression must yield a value of the set for which the control-variable is defined. The initial value and the final value specify a subrange on the set. The values assigned to the control-variable are taken from the subrange starting with the initial value. Remember that the set of a simple type is ordered. In case (a) the values are taken in increasing order, in case (b) in decreasing order. The repeated statement is not executed in case (a) if initial value > final value, in case (b) if initial value < final value.

NOTE: The repeated statement may not change the value of the control-variable nor the values used to evaluate the final-expression. The control-variable should be regarded as having unknown value after completion of a FOR statement.

3.1.5 Conditional Statements

3.1.51 IF Statement

It has one of the two forms:

(a) IF condition THEN statement1 ELSE statement2

(b) IF condition THEN statement1

Example:

```
IF A > 3 THEN B := 5;
IF BOOL THEN B := 1 ELSE B := 5;
```

The condition must be an expression which yields a value of type BOOLEAN. If the condition is true then statement1 is executed, otherwise in case (a) statement2 is executed, in case (b) no statement is executed.

NOTE: No semicolon is allowed before ELSE.
The construction

```
IF X THEN IF Y THEN A := 3 ELSE B := 3;
```

is always interpreted as:

```
IF X THEN
BEGIN IF Y THEN A := 3 ELSE B := 3 END;
```

3.1.52 CASE Statement

It is of the form:

```
CASE  selector-expression  OF
      constant1 :  statement1;
      constant2 :  statement2;
      . . .
END;
```

The constants constant1, constant2, ... are used to label the statements in the CASE statement. The constants are all of the same (simple) type. No constant may be used twice for a label in a case statement. If the value from the selector-expression is equal to a constant with which a statement is labeled then this statement is executed, otherwise no statement is executed.

Example:

```
I := 2;
CASE I OF
     1: X := 1;
     2: X := 9;
     3: X := 4;
END;
WRITE(X);
```

The value printed would be 9 .

A statement of the CASE statement can also be labeled with a list of constants.

Example:

```
CASE COLOR OF
     RED,GREEN:  X := X+1;
           BLUE:  X := 0;
END;

CASE I OF
     1,2,3: ;
       4,5: X := Y;
END;
```

The last example also shows the use of an empty statement.

3.2 Expressions

An expression is formed with operands and operators. Operands are variables, constants, function calls, and expressions in round brackets. Most of the operators require two operands (dyadic operators), some only one operand (monadic operators).

Example: `-F(X)+3*(A+5)`

3.2.1 Operators

Operators are applied in the order which corresponds to their precedence. Operators with higher precedence are applied first. Operators which have the same precedence are applied from left to right.

Precedence	Operator
4	¬
3	* / DIV MOD ∧ <>
2	+ - ∨
1	= ≠ < ≤ ≥ > IN

The type of the value of an expression depends on the operands and the operator involved, e.g. comparison of two numbers results in a value of type BOOLEAN, whereas the addition of two integers will result in a value of type INTEGER.

The following table describes the operation, the type of the operands and the type of the result. (The type of the operands may also be a subrange of the indicated types.)

Operator	Operation	Type of Operands		Type of Results
¬	Logical Negation	Boolean		Boolean
*	Multiplication	Integer	Integer	Integer
		Real	Real	Real
		Integer	Real	Real
/	Division	Integer	Integer	Real
		Real	Real	Real
		Integer	Real	Real
DIV	Integer Division	Integer	Integer	Integer
MOD	Remainder of Int. Division	Integer	Integer	Integer
+	Addition	Integer	Integer	Integer
		Real	Real	Real
		Integer	Real	Real
-	Subtraction	Integer	Integer	Integer
		Real	Real	Real
		Integer	Real	Real
∧	Logical AND	Boolean	Boolean	Boolean
∨	Logical OR	Boolean	Boolean	Boolean
= ≠	Comparison	Any Type		Boolean
< > ≤ ≥	Comparison	Simple Type		Boolean

For comparisons both operands must be of the same type (or subrange).

Example:

```
X := 14 DIV 3;      ↵RESULT IS 4↓
Y := 14 MOD 3;      ↵RESULT IS 2↓
Z := (A∨B)∧¬(A∨B); ↵RESULT IS FALSE↓
B := 5 > 3;         ↵RESULT IS TRUE↓
```

The operators +,- may be used as monadic operators. They cannot be preceded by another operator.

Some of the operators above and the operator IN are also used for another type. This is treated in Part III.

3.2.2 Function Calls to Standard Functions

Using the name of a function in an expression results in a function call. The name of the function may be followed by a list of parameters as required by the function. Functions always return values; the type of the value is defined by the function.

The PASCAL system has several built-in functions. All these standard functions have one parameter.

Example: `A := COS(X) + (SIN(X) +1);`

The programmer can define his own functions. This is treated in Part II.

In the following we describe the built-in functions, the type of their parameter and the type of the value they return. (The type of the parameter may also be a subrange of the indicated type.)

3.2.21 Arithmetic Functions

SIN(X), COS(X), EXP(X), LN(X), SQRT(X), ARCTAN(X)

X is of type INTEGER or REAL. The type of the result is always REAL.

RANF(X)

X is of type INTEGER or REAL. The result is a random number in the interval (0,1) and of type REAL. It is influenced by the value of X:

1. For X=0 the next random number is generated and returned.
2. For X<0 the last previously generated random number is returned.
3. For X>0 a new seed (start of a random number sequence) is created from X and returned.

ABS(X)

The result is the absolute value of X. The type is the same as the type of X. X is of type INTEGER or REAL.

SQR(X)

The result is X*X. The type of X is INTEGER or REAL. The type of the result is the type of X.

3.2.22 Predicates

ODD(X)

The type of X is INTEGER. The result is BOOLEAN and indicates if X is odd.

3.2.23 Transfer Functions

TRUNC(X)

X is of type REAL. The result is the integer part of X and of type INTEGER.

ORD(X) or INT(X)

X is a simple type (including the type ALFA). The result is the octal representation of X expressed as an integer.

CHR(X)

X is of type INTEGER. The result is of type CHAR. It is the character which corresponds to the octal representation of the integer. The value of the integer should be greater than or equal to 0 and less than or equal to 77B.

ALF(X)

X is of type INTEGER. The result is of type ALFA. The octal representation of X is interpreted as a character sequence.

3.2.24 Other Standard Functions

SUCC(X)

X is of simple type. The result is of the same type. Its value is the value following X in the ordered set of the simple type.

PRED(X)

X is of simple type. The result is of the same type. Its value is the value previous to X in the ordered set of the simple type.

WARNING: No check is made if the limits of the set of the simple type are reached.

3.2.25 Examples

```
TYPE COLOR = (RED,GREEN,BLUE);
VAR X: COLOR; C: CHAR; I,K: INTEGER;
    A: ALFA;  B: BOOLEAN;

BEGIN  X := RED;
       I := ORD(X);          ↱RESULT IS 0↓
       X := SUCC(X);         ↱RESULT IS GREEN↓
       K := ORD(ΞEΞ);        ↱RESULT IS 5↓
       B := ODD(K);          ↱RESULT IS TRUE↓
       K := PRED(K);         ↱RESULT IS 4↓
       K := SQR(K);          ↱RESULT IS 16↓
       I := TRUNC(3.027);    ↱RESULT IS 3↓
       C := CHR(5);          ↱RESULT IS ΞEΞ↓
       A := ALF(01020355555555555555);
                             ↱RESULT IS ΞABC        Ξ↓
END.
```

4. Program Examples

4.1 Simple Sort of Values

```
CONST LIMIT=10;
VAR   V: ARRAY[1..LIMIT] OF INTEGER;
      I,K,L,LL,M,Q: INTEGER;

BEGIN FOR I := 1 TO LIMIT DO READ(V[I]);
      FOR M := 1 TO LIMIT-1 DO
      BEGIN K := V[M]; LL := 0;
            FOR L := M+1 TO LIMIT DO
                IF V[L] < K THEN
                BEGIN K := V[L]; LL := L END;
            IF LL≠0 THEN
            BEGIN Q := V[LL]; V[LL] := V[M]; V[M] := Q END;
      END;
      FOR I := 1 TO LIMIT DO WRITE(V[I]);
END.
```

4.2 Computation of the First 100 Prime Numbers

```
VAR P: ARRAY[1..100] OF INTEGER;
    I,J,K,L: INTEGER;
    B: BOOLEAN;

BEGIN J := 3; P[1] := 2; P[2] := 3;
      FOR K := 3 TO 100 DO
      BEGIN
      REPEAT B := TRUE;
             J := J+2;    ↱GET NEXT NUMBER↓
             I := 1;
             ↱TRY ALL PRIME NUMBERS P[I] SO FAR FOUND
             UNTIL J DIV P[I] ≤ P[I]↓
             REPEAT I := I+1;
                    L := J DIV P[I];
                    B := J ≠ L*P[I];
             UNTIL ¬B ∨ (L ≤ P[I]);
      UNTIL B;
      P[K] := J;
      END;
      FOR K := 1 TO 100 DO WRITE(Ξ Ξ,P[K],EOL);
END.
```

PART II

5. Functions and Procedures

5.1 Function Declaration and Function Call

Functions are algorithms which yield a value. A function can be called from different parts of the program. This makes it necessary to have an algorithm in which certain names and values can be adapted to the various situations. When the function is declared so-called formal parameters keep the place for these names and values. Formal parameters are identifiers which are listed in a parameter list. When the function is called the corresponding actual names and values are listed in a parameter list. These parameters then are called actual parameters.

A function declaration consists of the function head, the declaration part and the program part.

The function head consists of the word FUNCTION, the name of the function, a parameter list with at least one parameter and the type of the result. The type must be a simple type or a pointer. (Pointers are treated in Part III). When the parameters are specified the kind of the parameter must be known, its name and, if applicable, its type. There are four kinds of parameters:

Values	
Variables	VAR
Functions	FUNCTION
Procedures	PROCEDURE

The kind of a parameter is defined by preceding it with the appropriate keyword indicated in the right column above. (Parameters of kind value are not preceded by a keyword).

Example:
```
FUNCTION ABC(X: INTEGER;
             VAR F: REAL;
             FUNCTION G: REAL;
             PROCEDURE P) : INTEGER;
```

Names of the same kind and type can be separated by commas.

Example:
```
FUNCTION XYZ(K,L:INTEGER; VAR P,Q: REAL):REAL;
```

The program part of a function declaration is a compound statement, i.e. a sequence of statements enclosed in
```
BEGIN . . . END;
```

Within the program part there must be at least one assignment statement which assigns a value to the function identifier.

The identifiers introduced in the function head are local to the function declaration and not known outside. Other local identifiers may be introduced with the declaration part of the function.

The declaration part has the same form as described in 2.3. (Note that function and procedure declarations may appear in the declaration part of the function.)

As a function is part of a declaration part, all the identifiers introduced before in the declaration part can be used (with their associated meaning) in the function declaration. They are considered to be global identifiers with respect to this function declaration. A global identifier is redefined when declared again as local identifier. The scope of this "redefinition" is the scope of local identifiers, i.e. the function declaration to which the identifier is local.

A function is called when the name of the function is used in an expression. The name is followed by the actual parameter list. According to the kind of formal parameter the corresponding actual parameter must be as follows:

Formal	Actual
value	expression
variable	variable
function	function identifier (with no parameter list)
procedure	procedure identifier (with no parameter list)

The type of the actual parameter and of the formal parameter must be the same (or eventually a subrange thereof).

Parameters of kind value are treated like local variables. The local variable is initialized with the value of the expression when the procedure is called. Remember, an expression may also be a constant, a variable or a function call, i.e. a function identifier followed by an actual parameter list. If the formal parameter is of type array or record the initialization consists of copying the array or record which is actual parameter to the local array or record. The values of the local variables which were introduced by the declaration part of the function are undefined when the program part of the function is entered.

There must be the same number of actual parameters in a function call as there are formal parameters in the function declaration.

NOTE: The index of an indexed variable on actual parameter position is evaluated when the function is called.

An important difference exists between the assignment to a formal parameter of kind variable and a formal parameter of kind value within the program part of the function. In the first case the value is assigned to the variable which is actual parameter of the function call. In the second case the value is assigned to the local variable of the function, and the variable which was used as actual parameter is not afflicted.

The following is an example for a function declaration. It computes the powers of a real number:

Example:

```
↱FUNCTION HEAD↓
FUNCTION POWER(W:REAL; I:INTEGER): REAL;

↱DECLARATION PART↓
   VAR Z: REAL;

↱PROGRAM PART↓
BEGIN Z := 1;
      ↱I MUST BE ≥ 0↓
      WHILE I ≠ 0 DO
      BEGIN IF ODD(I) THEN Z := Z*W;
            I := I DIV 2;
            W := SQR(W);
      END;
      POWER := Z;
      ↱ASSIGNMENT TO THE FUNCTION IDENTIFIER↓
END;
```

The function identifier can appear in an expression of the program part of the function declaration. This causes a recursive call to the function when the function is executed.

The following is an example for the recursive appearance of the function name FAC in the function declaration of the function FAC. It computes the factorial of an integer greater than or equal to 0.

Example:

```
FUNCTION FAC(N: INTEGER): INTEGER;
BEGIN
  IF N = 0 THEN FAC := 1 ELSE FAC := N*FAC(N-1)
END;
```

NOTE: Functions are intended to return only one value, the value which was assigned to the function identifier. However, values can be returned from a function in an indirect way by assigning values to global variables and parameters of kind variable. As functions are used in expressions, this "side effect" should be avoided.

5.2 Procedure Declaration and Procedure Call

Procedures are algorithms which do not explicitly return a value. Procedures are called from procedure statements.

A procedure declaration consists of the procedure head, the declaration part and the program part.

The procedure head consists of the word PROCEDURE, the name of the procedure and eventually of a parameter list. Everything mentioned about parameters for functions also holds for procedures. (As no value is returned, we do not have to specify a type, and there is no assignment to the procedure identifier.)

Example:

```
PROCEDURE XXX (K,L: INTEGER; VAR X,Y: REAL);
```

The program part is a compound statement, i.e. a sequence of statements enclosed in BEGIN . . . END;

The declaration part has the same form as described in 2.3. As procedures are part of a declaration part everything mentioned about global and local identifiers also applies to procedures.

A procedure returns values only by "side effect", i.e. by assigning values to global variables and parameters of kind variable in the program part of the procedure.

The procedure identifier can be used in a procedure statement in the program part of the procedure declaration. This causes a recursive call to the procedure when the procedure is executed.

NOTE: When the program part of the procedure is entered, the values of local variables introduced by the declaration part of the procedure are undefined.

The following is an example for a procedure declaration. It prints the values of a matrix, where the type MATRIX is defined by:

```
CONST N=5;
TYPE MATRIX = ARRAY[1..N,1..N] OF REAL;
```

Example:

```
PROCEDURE PRINT(VAR A: MATRIX; N: INTEGER);
   VAR I,K: INTEGER;

BEGIN FOR I := 1 TO N DO
      BEGIN FOR K := 1 TO N DO WRITE(A[I,K]);
            WRITE(EOL);
      END;
END;
```

5.3 Forward Declaration of Functions and Procedures

Calls to functions or procedures may be necessary before they can be declared, e.g. two procedures call each other. Then only the procedure head (function head) of these procedures (functions) must be declared followed by the word FORWARD. Later, when it is possible to declare the rest of the procedure (function) only a shortened procedure head (function head) is necessary. It consists of the word PROCEDURE (FUNCTION) and the identifier used to name the procedure (function).

Example:

```
FUNCTION F(X: REAL):REAL; FORWARD;

{OTHER PROCEDURES OR FUNCTIONS MAY FOLLOW
HERE CALLING THE FUNCTION F}

FUNCTION F;
{DECLARATION OF FUNCTION F FOLLOWS}
BEGIN F := X END;
```

5.4 Standard Procedures: PACK, UNPACK

The procedures PACK and UNPACK are built-in procedures. They are used to convert values of type ALFA into arrays of type CHAR and vice versa. Let

A be an array of type CHAR,
Z be a variable of type ALFA,
I be an expression of type INTEGER.

PACK(A, I, Z) takes 10 consecutive characters of array A, beginning with the component A[I] and packs them into the variable Z

UNPACK(Z, A, I) unpacks the value Z of type ALFA into 10 consecutive components of array A beginning with A[I].

6. Input/Output

6.1 FILE Type

Files consist of a sequence of components which are all of the same type. The length of the sequence is unknown (and only a part of the file is kept in core). A file is read or written sequentially and only that component is accessible which is under the "file head".

For the declaration of a file variable we must specify the type of its components:

Example:
```
ABC: FILE OF CHAR;
XYZ: FILE OF INTEGER;
```

The type TEXT is another predefined type in PASCAL. It is defined by

```
TYPE TEXT = FILE OF CHAR;
```

Example:
```
ABC: TEXT;
```

The two files INPUT and OUTPUT are also predefined. They are of type TEXT. (A component of a file of type TEXT is of type CHAR).

The component under the "file head" is reached by the so-called buffer variable. It consists of the file name followed by ↑ .

Example:
```
ABC↑ := 'A';
```

More on file type declaration is to be found in Part III.

6.2 Procedures and Functions for Input and Output

EOF(ABC)	is a function which returns the value TRUE if the file ABC is in end-of-file status.
GET(ABC)	moves the "file head" to the next component of the file and assigns the value of the component to ABC↑ . If there is no next component EOF(ABC) becomes TRUE, and the value of ABC↑ is not defined.
PUT(ABC)	writes the component under the "file head" to file ABC and the value of ABC↑ becomes undefined. (Now a new value can be assigned to the "file head".)
WEOR(ABC)	writes an end-of-record on file ABC.
RESET(ABC)	rewinds the file ABC.

The following program copies the file AAA to file ZZZ, element by element

Example

```
VAR AAA: FILE OF INTEGER;
    ZZZ: FILE OF INTEGER;

BEGIN RESET(AAA); RESET(ZZZ);
      GET(AAA);  ↱READ FIRST COMPONENT↳
      WHILE ¬EOF(AAA) DO
      BEGIN ZZZ↑ := AAA↑;
            PUT(ZZZ); ↱WRITE AN ELEMENT↳
            GET(AAA); ↱GET NEXT COMPONENT↳
      END;
      WEOR(ZZZ);
END.
```

7. Labels and GOTO Statements

A label is defined by labeling a statement with an integer consisting of at most 5 decimal digits.

Example: `10 : A := 3;`

A label is used with a GOTO statement to indicate that the execution of the program will continue with the labeled statement. The GOTO statement is a simple statement.

Example: `GOTO 10;`

The scope of a label is the program part of the procedure (function, program) in which the label is defined. This scope can be extended by declaring the label in the corresponding declaration part. Then this label can be used together with a modified GOTO statement in the procedures and functions which are declared in this declaration part. The modified GOTO statement makes explicit that the procedure is not left in the normal way. It has the form:

Example: `GOTO EXIT 12;`

The label declaration precedes all other sections of the declaration part. It starts with the keyword LABEL which is followed by a list of labels:

Example: `LABEL 10,20,30;`

NOTE: All labels which are declared in the label declaration must be defined in the corresponding program part.

It is a sign of good PASCAL programming to avoid labels. The only need for labels arises when a procedure must be left with an EXIT jump. Otherwise the programmer is urged to use the FOR, WHILE, REPEAT and CASE statements. The program in the next section shows the use of a label.

WARNING: Jumps into structured statements may be hazardous to the programmer's health.

8. Program Examples

8.1 Matrix Inversion (GAUSS-JORDAN, With Max. Pivot)

```
LABEL
      10;
CONST
      N=4;N1=5;
TYPE
      MATRIX = ARRAY[1..N,1..N1] OF REAL;
      DIM    = 1..N;
VAR
      I,K: INTEGER;
        A: MATRIX;

PROCEDURE INVERT(VAR A: MATRIX; N: DIM);
      VAR I,K,L,J: INTEGER;
            S,H: REAL;

BEGIN ↱PRESET ROW COUNTER↓
      FOR L := 1 TO N DO A[L,N+1] := L;

      FOR L := 1 TO N DO
      BEGIN ↱FIND LARGEST ELEMENT IN THIS COLUMN↓
            H := ABS(A[L,L]); J := 0;
            FOR K := L+1 TO N DO
                IF ABS(A[K,L])>H THEN
                BEGIN H := ABS(A[K,L]); J := K END;

            IF H < 1E-13 THEN
            BEGIN WRITE(≡ MATRIX IS SINGULAR≡); GOTO EXIT 10 END;

            IF J≠0 THEN  ↱EXCHANGE ROWS↓
            FOR K := 1 TO N+1 DO
            BEGIN H := A[L,K]; A[L,K] := A[J,K]; A[J,K] := H END;

            ↱NORMALIZE PIVOT ROW↓
             H := 1/A[L,L];
             FOR I := 1 TO N DO A[L,I] := A[L,I]*H;
            ↱ELIMINATE AND DEVELOP INVERSE↓
             A[L,L] := H;
             FOR I := 1 TO N DO
                 IF I≠L THEN
                 BEGIN S := A[I,L];
                       A[I,L] := -S*H;
                       FOR K := 1 TO N DO
                       IF K≠L THEN A[I,K] := A[I,K]-S*A[L,K];
                 END;
      END;
```

```
      FOR L := 1 TO N-1 DO
          IF A[L,N+1]≠L THEN  ↑EXCHANGE THE COLUMNS↓
          BEGIN K := L;
                REPEAT K := K+1 UNTIL A[K,N+1]=L;
                A[K,N+1] := A[L,N+1];
                FOR I := 1 TO N DO
                BEGIN H := A[I,K]; A[I,K] := A[I,L]; A[I,L] := H END;
          END;

END;

PROCEDURE PRINT;  ↑PRINT THE MATRIX↓
      VAR I,K: INTEGER;
BEGIN FOR I := 1 TO N DO
      BEGIN FOR K := 1 TO N DO WRITE(A[I,K]);
            WRITE(EOL)
      END;
      WRITE(EOL);
END;

↑MAIN PROGRAM↓
BEGIN ↑READ A MATRIX↓
      FOR I := 1 TO N DO
      FOR K := 1 TO N DO READ(A[I,K]);

      WRITE(Ξ INPUTΞ,EOL);
      PRINT;

      INVERT(A,N);
      WRITE(Ξ INVERTED MATRIXΞ,EOL);
      PRINT;

      INVERT(A,N);
      WRITE(Ξ CONTROL: MATRIX RE-INVERTEDΞ,EOL);
      PRINT;
10:
END .
```

Part III

9. Structured Types: RECORD, CLASS, SET, FILE

9.1 Record Type

9.1.1 Record Declaration

Some problems make it desirable to have data structures which differ only in some components, but are otherwise alike, e.g. records of people where some additional entries are required if they are married.

In PASCAL the type RECORD has a variant part which is useful for these purposes. The variant part follows the fixed part which was described in 2.2.22.

The variant part starts with a component called "tag field". Its value describes the instance of the variant part. The type of the tag field is a simple type. Its name and type are defined by the programmer. The instances of the variant part follow next in the record declaration. Each instance is labeled with a constant of the type used for the tag field. An instance is the continuation of the fixed part of the record and thus has the form of a record. However, it is enclosed in parenthesis. (The different instances are separated by semicolons.) The variant part begins with

```
CASE  tagfieldname : type  OF
```

Example:

```
TYPE C = (YES,NO,MAYBE);
     R = RECORD X,Y: INTEGER;

           CASE KIND: C OF
           YES:   (NAME: ALFA;
                     AGE: 1..100);
           NO:    (NEXT: INTEGER);
           MAYBE: (COND: BOOLEAN)

          END;

VAR P,Q: R;
```

All the components of a record are accessed in the usual way:

Example:

```
P.KIND := NO;
P.NEXT := 3;
P.X    := 27;

Q.KIND := YES;
Q.NAME := 'SMITH';
Q.AGE  := 5;
```

The structure of a value of type RECORD is represented by only one of the instances of the record declaration. (A variable of type RECORD may assume values which are structured in all the different ways which the record declaration allows.)

We can assign a value to the tag field so that we know what instance we used to structure the variant part. The value for the tag field is the constant which labels the appropriate instance in the record declaration.

WARNING: Before assigning or retrieving a component of the variant part of a record the programmer should make sure he knows the structure of the variant part. This is done by means of the tag field. It is the responsibility of the programmer to assign the correct value to the tag field and to test it when necessary. The use of the wrong structure will yield unexpected results.

An instance of a variant part is considered to be a record, thus it may consist of a fixed part and a variant part.

Example:
```
TYPE RV = RECORD
            CASE B1 : BOOLEAN OF
            TRUE:     (A: INTEGER);
            FALSE:    (CASE B2: BOOLEAN OF
                        TRUE:    (B,C: INTEGER);
                        FALSE:   (D:   REAL)
                       );
            END;
```

Sufficient memory is allocated to a variable of type RECORD to take care of all components for the worst combination of the variant parts.

The values of many simple types do not require a full memory word. A record structure which is preceded by the word PACKED takes advantage of that and packs several consecutive components together in one word, if this is possible.

Example:
```
TYPE RR = PACKED RECORD
                   A,B: 1..3;
                     C: CHAR;
                   D,E: BOOLEAN;
          END;
```

A variable of this type occupies only one word, whereas normally 5 words would be needed.

NOTE: A component of a packed record is denoted as usual. However, it cannot be used as actual parameter for a parameter of kind variable.

9.1.2 WITH Statement

It has the form

WITH record-variable DO statement

The WITH statement simplifies the notation for components of the record-variable mentioned in the WITH statement. The field names of this record-variable can be used directly as variables.

NOTE: The scope of the field names (in their use as variable identifiers) is the statement part. The field names can be considered to be declared locally to the statement part, thus other variable identifiers with the same name appear to be global and are not accessible.

The record variable V in the following example is defined as in 2.2.22:

Example:

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

This program part is equivalent to:

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

NOTE: The construction

```
WITH X DO
WITH Y DO
WITH Z DO
.
.
```

can be shortened to: WITH X,Y,Z DO . . .

9.2 CLASS Type

9.2.1 Class Declaration, Pointer Declaration

A class is a structure which consists of a variable number of components. All components have the same type. The components are created during execution of the program. When the class is declared we specify the type of the components and the maximum number of components:

Example:
```
VAR STRUCT: CLASS 100 OF RECORD X: ALFA;
                                Y: INTEGER;
                            END;
```

As we do not have names for the components of the class we must access them in a different way. A class has a POINTER type always associated with it. This is a set of values which are pointers to the components of the class. The set includes the constant NIL, a pointer which points to no element. A name for the POINTER type is defined in the following way:

Example:
```
TYPE POINTER = ↑STRUCT;
```

(This is the only exception where we can use an identifier, i.e. the name of the class, before it is declared.)

Variables of a POINTER type are declared in the usual way:

Example:
```
VAR P1,P2: POINTER;
```

NOTE: Each variable of type CLASS has its own pointer type associated with it. The programmer must define a name for each type. Further, variables of these types are necessary.

A component of a class is reached with a pointer variable followed by ↑. The value of the pointer variable must be the pointer to the component.

Example:
```
P1↑.X := ΞSMITHΞ;
```

NOTE: A run-time error occurs, if the value of P1 in the example above is the NIL pointer.

The usefulness of the class structure becomes obvious when we have components of type RECORD which have fields with POINTER types.

Example:
```
TYPE PTR = ↑LIST;
VAR LIST : CLASS 100 OF
           RECORD NEXT: PTR;
                  VAL: INTEGER;
           END;
    PT1,PT2,PT3 : PTR;
```

With this class as declared above we can create a linked list. Each element of the list consists of two components. The first component contains a pointer which points to the next element of the list, or the value NIL to indicate it is the last element of the list. The second component contains the value of the list element. The construction of the list is described in the next section.

With the appropriate definition of the class component using pointers we can create other structured objects, like trees and graphs.

9.2.2 Allocation of Class Components

9.2.21 The Procedure NEW

Space for the component of a class is allocated with the procedure NEW. The class is thereby treated as stack. An internal pointer points to the stack top. Space is allocated for a component by setting the internal pointer to a new stack top. Its previous value is taken as the pointer to the component and returned as value of the pointer variable with which the procedure NEW was called.

NOTE: The value NIL is assigned to the pointer variable when the space which was reserved by the class declaration is exhausted.

The following program part uses the class definition of the previous example. It creates a linked list of 5 elements.

Example:

```
NEW(PT1);
PT3 := PT1; ↑SAVE THE POINTER TO THE FIRST ELEMENT↓
FOR K := 1 TO 4 DO
BEGIN NEW(PT2);
      PT1↑.NEXT := PT2;
      PT1 := PT2;
END;
PT1↑.NEXT := NIL; ↑STORE NIL IN THE LAST ELEMENT↓
```

9.2.22 The Procedure NEW for Classes of Records with Variant Parts

The various instances of records with variant part have different space requirements. If they are components of a class then the procedure NEW used in the above way will allocate space for the worst possible case.

If we want to be more economical then we allocate only that much space which a certain instance of the record requires. The parameters for the procedure NEW are, besides the pointer variable, the values with which the instances of the variant parts used for this component are labeled.

Example:

```
TYPE CLVP = ↑CLV;
     CR   = (RED,GREEN);
VAR  CLV : CLASS 10 OF
           RECORD X: INTEGER;
                  CASE VR: CR OF
                  RED: (Y: INTEGER;
                        Y1:ALFA;
                        Y2:CLVP);
                  GREEN: (Z: INTEGER);
           END;
     CP :  CLVP;
     . . . . . .
NEW(CP,RED);
WITH CP↑ DO
BEGIN VR := RED;
      Y  := 1;
      Y1 := ΞALFAΞ;
END;
```

WARNING: The procedure NEW only allocates the space and does not assign a value to the tag field. Once space is assigned for a component with a certain variant structure only components with the same space requirement can be stored at that place. This must be observed by the programmer.

9.2.23 The Procedure RESET Used with Pointers

Components of a class are de-allocated when the internal stack pointer is set to a previous stack top. This is done by

Example:

```
RESET(P);
```

P is a pointer variable whose value defines the new stack top.

WARNING: It is the programmer's responsibility not to use pointers to components which are already released.

9.2.3 Operations with Pointers

Pointers of the same type can be compared with the operators = ≠ < > ≤ ≥. Pointers with smaller values point to components closer to the stack bottom. All pointers are smaller than the NIL pointer.

The internal value of a pointer is obtained when a pointer variable is used as parameter of the function ORD. Pointers are absolute addresses. The value of the NIL pointer is 200000B (65536).

9.3 SET Type

9.3.1 Set Declaration, Set Values

The SET type is a structured type whose values are subsets of simple types. The simple type must be a subrange (in what follows called base type), so that in the decimal representation the lower bound is greater than or equal to 0 and the upper bound less than 59. A SET type value is represented by one word. The corresponding bit is set in this word when the value is present in the set. (The sign bit is not used.)

Example:
```
TYPE C = (RED,GREEN,BLUE);
VAR  SET1 : SET OF 0..58;
     SET2 : SET OF 'A'..'Z';
     SET3 : SET OF C;
```

Values of variables of SET type are sets whose elements are values of the base type of the SET type. A set can be empty. SET type values are created by a list of expressions enclosed in brackets. The expressions must evaluate to a value of the base type.

Example:
```
SET1 := [];      (*EMPTY SET*)
SET2 := ['A','B','X','Y'];
SET3 := [GREEN];
```

9.3.2 Operations with Values of Type SET

The following operations are defined for values of type SET:

Operator	Operation	Type of operands		Type of result
∨	Union	SET	SET	SET
∧	Intersection	SET	SET	SET
-	Set difference	SET	SET	SET
+	(Exclusive or)	SET	SET	SET
≤ ≥	Set inclusion	SET	SET	BOOLEAN
IN	Membership	Base type	SET	BOOLEAN

The sets involved in the operations must be of the same type.

The set difference a-b is defined as:
{X: X element of A and X not element of B}

The operation A+B is defined as (A∨B)-(A∧B).

Example:

```
VAR SET11,SET12,SET13: SET OF 0..5;
    B: BOOLEAN;
    . . . . .
SET12 := [0,1,2];
SET13 := SET12 ∨ [0,1,4];      ↱0,1,2,4↓
SET11 := SET12 ∧ SET13;        ↱0,1,2↓
B := 5 IN SET11;               ↱FALSE↓
B := SET11 ≤ SET13;            ↱TRUE↓
B := SET11 ≥ SET13;            ↱FALSE↓
SET11 := SET13-SET11;          ↱4↓
SET11 := SET13+[1,2];          ↱0,4↓
```

9.4 FILE Type

In addition to the type of the file components we can specify the kind of file and the length of the buffer which keeps part of the file in core. There are three kinds of files: Input files, Output files and Scratch files.

Example:

```
AAA[IN] : FILE OF INTEGER;  ↱KIND INPUT↓
ZZZ[OUT]: FILE OF INTEGER;  ↱KIND OUTPUT↓
SSS     : FILE OF INTEGER;  ↱KIND SCRATCH↓
```

The predefined file INPUT is of kind Input, the file OUTPUT of kind Output.

Files defined in the declaration part of procedures or functions must be of kind Scratch.

Files are "opened" when the program part in which they are defined is entered, they are "closed" when this program part is left. A file of kind Input is rewound when opened. A file of kind Output is closed by emptying the file buffer and writing an end-of-record. The action taken for a Scratch file when closed depends on the last operation performed with this file. If it was a write operation then it is treated like a file of kind Output. Files are also closed before they are rewound with the standard procedure RESET.

The size of the file buffer may be specified by the so-called blocking factor. It indicates the number of 64-word blocks to be held in the file buffer. The value 8 is taken if no blocking factor is specified.

Example:

```
SSS: FILE 8 OF CHAR;
```

NOTE: Only the first 7 characters of the file name are used to identify the local file which is attached to the job. The names INPUT and OUTPUT may be considered as formal parameters of the PASCAL program. The actual names are specified by the D and R parameter of the PASCAL Control Card (see Appendix B).

10. Input/Output, Value Section

10.1 The Procedure READ

The procedure READ(CH) is defined for CH of type CHAR by

```
CH := INPUT↑; GET(INPUT);
```

and accordingly for parameters of the other possible types. The character immediately following the item read is then under the "file head".

In order to obtain a defined value for the first call of READ a call to the procedure GET is necessary. However, in the case that the first call to READ never was preceded by a call to GET, READ(CH) is defined by

```
GET(INPUT); CH := INPUT↑; GET(INPUT);
```

The function EOF becomes true when an end-of-file is encountered. The value returned is Indefinite when a number was read, and EOL when a character was read.

The following is an example how to read integers until an end-of-file is encountered.

Example:

```
VAR I: INTEGER;
BEGIN READ(I);
      WHILE ¬EOF(INPUT) DO
      BEGIN . . . . . ; READ(I) END;
END.
```

10.2 The Procedure WRITE

A parameter of the standard procedure WRITE may be followed by a "format specification". A format consists of one or two parts. The first part is always an expression of type INTEGER. It specifies the number of characters, the so-called fieldwidth, with which the parameter value is written. The second part is used for special cases and discussed later.

Example:

```
WRITE(A:5, X:22, Z:3*K);
```

Parameter values are converted to character strings according to their type.

First we treat values of type ALFA, CHAR, INTEGER and BOOLEAN. The character string length of each of these types depends upon the value involved. The character string is adjusted to the length of the fieldwidth. If the character string is smaller, it is right adjusted and preceded with blanks. If it is larger, the character string is for

values of type

BOOLEAN	T or F right adjusted, if the fieldwidth is < 6,
ALFA	the leftmost characters,
INTEGER	asterisks.

If no format is specified the fieldwidth used is for type

BOOLEAN	10,
ALFA	10,
INTEGER	10,
CHAR	1.

(Note that the character string of an integer may consist of up to 15 digits and possibly a sign).

```
Example:        B := TRUE; BB := FALSE;
                I := 123; C := 'X'; A := 'HELLO';
                WRITE(B, B:2, BB:8, I, -I:4, A:2, C:4, A:12);
```

The character sequence written is as follows, colons represent blanks:

```
::::::TRUE:T:::FALSE:::::::123-123HE:::X::HELLO:::::
```

Values of type REAL can be written as floating point numbers or fixed point numbers. The maximum accuracy is 14 digits. If no format or only a format with fieldwidth specification is used the number is written as a floating point number, i.e. with decimal point and scale factor, 1 digit is before and at least 1 digit after the decimal point.

The length of the character string is computed as follows:

```
   2  for blanks or blank and sign
 + 1  for first digit
 + 1  for decimal point
 + 1  for each following digit (at most 13)
 + 4  for the exponentiation part.
```

The character string is adjusted to the fieldwidth either with leading blanks or by omitting the least significant digits of the mantissa. If the fieldwidth is < 9 asterisks are written. If no format is specified, a fieldwidth of 20 is used.

```
Example:        R := 1239E10; S := 1E-100;
                WRITE(R, -S:22, R:10);
```

The character sequence written is as follows, colons represent blanks:

```
::1.239000000000E+12::-1.0000000000000-100::1.24E+12
```

A value of type REAL is written as fixed point number when the second part of the format specification is an expression of type INTEGER. Its value, the so-called fraction width, specifies the number of digits which follow the decimal point. The fraction width must be greater than or equal to 0. The length of the character string is computed as follows:

```
   2  for blanks or blank and sign
 + 1  for each leading digit
 + 1  for decimal point if the fraction width is not equal to 0
 + fraction width
```

If the length is smaller than the fieldwidth asterisks are written. They are also written if the value is larger than can be expressed with 14 leading digits. The character string is adjusted to the fieldwidth with leading blanks or, as at most 14 digits can be written, with trailing blanks if the fraction width is larger.

Example:
```
X := 123.346;
WRITE(X:10:3, X:10:0, -X*1E-5:10:5);
```

The character sequence written is as follows, colons represent blanks:

```
:::123.346:::::::123::-0.00123
```

The octal representation of values of type BOOLEAN, ALFA, CHAR, INTEGER, and REAL can be written if the fieldwidth specification is followed by the word OCT. The character string consists of 20 octal digits, it is adjusted to the fieldwidth with leading blanks or by discarding the most significant digits.

Example:
```
WRITE(123B:2 OCT, ΞABCΞ: 22 OCT);
```

The character sequence written is as follows, colons represent blanks:

```
23::01020355555555555555
```

NOTE: Formats can be variable since the fieldwidth is an expression. The fieldwidth value must be greater than or equal to 0 and less than or equal to 136. If the fieldwidth is 0 no character is printed.

A string with a length greater than 10 and less than or equal to 70 (a so-called long string) may be also a parameter of the procedure WRITE. It may be followed by a format specification. The length of a long string is used as fieldwidth. If the fieldwidth is specified in the format specification the resulting value must be the same or larger as the length of the long string. The string then is preceded by the appropriate number of blanks.

Example:
```
WRITE(Ξ THIS IS A VERY LONG STRINGΞ: 35);
```

The character sequence written is as follows, colons represent blanks:

```
:::::::::THIS:IS:A:VERY:LONG:STRING
```

10.3 The Procedures INFILE and OUTFILE

The procedure READ uses the standard file INPUT, the procedure WRITE the standard file OUTPUT. If we want to read from a file XXX of type TEXT with the standard procedure READ we switch over to this file by:

Example: `INFILE(XXX);`

Henceforth READ reads from file XXX until another call to INFILE is encountered. The file INPUT is used again after INFILE(INPUT).

The procedure OUTFILE is used in a similar way for the procedure WRITE. OUTFILE(YYY) makes the file YYY (of type TEXT) to the file which is used by WRITE until another call to OUTFILE is encountered. The file OUTPUT is used again after OUTFILE(OUTPUT).

10.4 Variable Initialization

Variables of the main program can be initialized. The initialization is done in a section which starts with the keyword VALUE. This section follows immediately after the variable declaration section.

Variables can be initialized with constants of simple types (this includes real numbers and ALFA values). Variables must be initialized in the same sequence as they are declared. An initialization has the form

```
                Variable name = Initial value;
or              Variable name = (Initial value list);
```

The second form is used to initialize arrays. A value in the initial value list can be preceded by a repetition factor which must be an integer, e.g. 3*123 . This initializes 3 consecutive components of the array with the value 123 .

Example:

```
VAR   X: REAL
      A: ARRAY[1..5] OF CHAR;
      Y: (RED,GREEN);

VALUE X = 3.21E-27;
      A = ('X',2*'A',2*'Z');
      Y = RED;
```

WARNING: No check is made if the type of the value and of the variable are the same nor if the number of values in the initialization list corresponds to the array size.

NOTE: Variables which get initialized should appear last in the variable declaration part.

11. Compiler Options, Debugging Aids

11.1 Compiler Options

Compiler options may be selected by the Control Card (see Appendix B) or within a program by comments of a special form:

Example: ↱$A+,R+,X-↓

A comment starting with a $ sign is considered to be an instruction to the compiler. A letter designates the option. The option is activated when the letter is followed by a + sign, it is de-activated when the letter is followed by a - sign. Most of the instructions to the compiler result in additional code generation, e.g. code for run-time checks.

The options are:

A check Assignments.

Additional code is generated to check if a value lies within the subrange of the variable to which it should be assigned. (This applies also to the standard procedure READ.)

C Print generated code in COMPASS form

After each procedure the instructions generated by the compiler are printed in the form of COMPASS assembler code.

D check Division by zero

Additional code is generated for divisions to check if the divisor is zero.

L List source program.

O check for stack Overflow

Additional code is generated to check at procedure entry if enough space is available on the stack for the local variables of the procedure.

R generate code for true Rounding

Additional code is generated for real arithmetic operations to perform true rounding instead of CDC-rounding.

X check for indeX values and values of CASE statements

Additional code is generated to check if the index value lies within the specified array bounds, or if the value of the selector expression has a correspondig label in the CASE statement.

The expansion of code and the degradation in execution speed may be considerable for the options A and X; they are small for the options D, O, and R.

If no options are selected with the Control Card the default conventions are:

↱$A-,C-,D-,L+,O-,R-,X-↓

NOTE: The options A and X should be used for debugging purposes.

11.2 Debugging Aids

11.21 PASCAL Dump

A dump is a snapshot of the program. It can be requested by the programmer with the standard procedure DUMP. If the appropriate option is used on the Control Card a PASCAL dump is also given automatically after a run-time error occured.

The PASCAL dump shows the contents of the variables for each active program part at the time when the dump was taken. In the case of procedures and functions the values of the parameters come first. Then follow the values of the variables in the same sequence as the variables were declared in the corresponding declaration part. The PASCAL dump further shows what procedures are active and from where they were called.

Two forms of the dump are possible: symbolic and octal. A symbolic dump can only be given if the names of the variables are saved during compilation. This is done when the option S is used on the Control Card or a ↱$S+↓ precedes the program part whose variable names of the corresponding declaration part should be saved. The option is in effect until a ↱$S-↓ is encountered. Only the variable names of simple types and pointers are saved.

The dump given by the procedure DUMP is in symbolic form (provided the necessary information was saved during compilation). The octal form is obtained when the option T is used on the Control Card.

The dump given by the PASCAL system in case of a run-time error is symbolic if option P is used on the Control Card and octal if option T is used.

The values of variables defined in the declaration part of procedures or functions are placed on a stack, whereas the values of variables of the main program are assigned to fixed places. The programmer can request a dump of stack or main program or both with the procedure DUMP:

```
DUMP(0)      dump stack and main program
DUMP(1)      dump stack only
DUMP(2)      dump main program only
```

Stack and main program are dumped when the dump is caused by the PASCAL system. The dump is always written to the standard file OUTPUT.

11.22 TRACE

Procedure and function calls can be traced. The program part of the procedure or function which should be traced must be preceded by ↱$T+↓ . The option is in effect until ↱$T-↓ is encountered.

Tracing is enabled or disabled by a call to the standard procedure TRACE:

TRACE(0)	end tracing
TRACE(1)	start tracing The name of the procedure or function called is printed. If the symbolic dump information is available the values of the actual parameters are also printed.
TRACE(2)	start tracing Only the name of the procedure or function called is printed.