PASCAL FOR ENGINEERS:
A COURSE INCLUDING OMEGASOFT PASCAL
FOR MICROCOMPUTERS

Introduction and Training

Lothar A. Tausch
Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

© Copyright CERN, Genève, 1985

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use.

If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.

CERN — Service d'information scientifique – RD/681 – 3000 – septembre 1985
ABSTRACT

These are the notes of a PASCAL course for controls engineers at CERN. The course starts with 'Standard Pascal' and includes OMEGASOFT\textsuperscript{\textcopyright} Pascal, a powerful extension of Pascal towards real-time and systems applications. It demonstrates how a language such as Pascal, with adequate extensions for systems programming and embedded microprocessor-driven systems, can substantially increase the productivity of programmers and the reliability of their products. Also enhanced will be the legibility of the programs and their maintainability, since programming in Pascal automatically leads to autodocumentation. Simple examples show how OMEGASOFT-PASCAL can be used for efficient programming of embedded systems for real-time data acquisition and control using the MC6809 microprocessor.

\textsuperscript{*} OMEGASOFT is a trademark of the CERTIFIED SOFTWARE CORPORATION.
Foreword

During the last years Pascal—a structured high-level programming language—has reached a broad field of applications in engineering. It has been successfully introduced into industrial and research applications. CERN, being no exception, is using Pascal increasingly for real-time data acquisition and controls in the fields of particle accelerator controls and of data reduction and controls applications for high-energy experiments.

To meet the increasing demand for technical training made by the engineers and technicians responsible for these projects, the present course of Pascal, with the particular aim to process control and real-time data acquisition and monitoring, has been given twice to more than 70 people. This was arranged within the PRIAM project of CERN for support in the microprocessor area. After reproducing the course notes for another 50 interested people not participating in the course, and because of many suggestions for a wider publication, the notes are now published in this report.

The present document represents only the backbone of the course, including examples and mementos. It should therefore not be mistaken for a book. A choice of good Pascal books can be found in the Bibliography.

Programming in the above-mentioned areas has now become an engineering discipline like others. It has been demonstrated repeatedly at CERN and elsewhere that modern controls software for process control can be designed even entirely in a modern high-level compiled language.

This enables efficient engineering of reliable and well-readable software, an important factor for future control systems and similar applications, since cost and manpower are shifting more and more from hardware efforts to software construction and software maintenance.
CONTENTS

Foreword V

1. INTRODUCTION 1

2. ALGORITHMS + DATA STRUCTURES = PROGRAM 2
   2.1 Examples of algorithms 2
   2.2 Data structures = description of data 4
   2.3 Scope nesting (block nesting) 5
   2.4 Basic vocabulary 6
   2.5 Rules, syntax 7

3. THE CONCEPT OF DATA 8
   3.1 Scalar data types 8
   3.2 Standard (predifined) scalar types 8
       3.2.1 The type BOOLEAN 8
       3.2.2 The type INTEGER 10
       3.2.3 The type REAL 11
       3.2.4 The type CHAR 12

4. PROGRAM HEADING AND DECLARATION PART 13
   4.1 Label declaration part 13
   4.2 Constant definition part 13
   4.3 Type definition part 14
   4.4 Variable declaration part 14
   4.5 Subprogram declarations 15

5. INPUT AND OUTPUT OF DATA 16
   5.1 Input and output on standard devices 16
   5.2 Style of output 18

6. THE CONCEPT OF ACTION (STATEMENTS) 19
   6.1 The Assignment statement and expressions 19
   6.2 The Compound statement 21
   6.3 Repetitive statements
       6.3.1 The REPEAT statement 21
       6.3.2 The WHILE statement 21
       6.3.3 The FOR statement 22
6.4 Conditional statements
   6.4.1 The IF statement
   6.4.2 The CASE statement
6.5 The GOTO statement
6.6 Procedure statements

7. ADDITIONAL DATA TYPES
7.1 User-defined scalar types (enumerated types)
7.2 Subrange types

8. STRUCTURED DATA TYPES
8.1 The type ARRAY
8.2 The type STRING (implementation-dependent)
8.3 The FILE types
   8.3.1 Text files
   8.3.2 Standard text files (INPUT, OUTPUT)
8.4 Sets
   8.4.1 Definitions
   8.4.2 Basic set operations
8.5 Sets in PASCAL
   8.5.1 The Set types
   8.5.2 Set data
   8.5.3 Set operators and operands
8.6 Record types
8.7 Pointer types (dynamic variables)
   8.7.1 Pointer declaration
   8.7.2 Creation of a linked list
   8.7.3 Moving through a list (Traversing a list)
   8.7.4 Insertion of a new segment into the middle of a list
   8.7.5 Other applications of pointers

9. SUBPROGRAMS (PROGRAM PARTS)
9.1 Procedures and Functions
   9.1.1 Introduction
   9.1.2 Procedures
   9.1.3 Functions
9.2 Parameter types
9.3 Recursive calls of procedures and functions
9.4 Side effects
9.5 Standard procedures and standard functions

BIBLIOGRAPHY

APPENDIX A: SUMMARY
A.1 Table of operators
A.2 Operator precedence
A.3 Reserved words (delimiters)
1. INTRODUCTION

Pascal was created by Niklaus Wirth of the ETH (Eidgenössische Technische Hochschule Zürich) in 1970–1971.

The language's name is a tribute to the French mathematician Blaise Pascal, who invented the first mechanical calculating machine.

The Pascal introduced by N. Wirth is today called 'Standard Pascal'. It is an excellent tool for the teaching of programming, the construction of algorithms, and for the writing of general-purpose programs.

Before Pascal could become more popular and spread to applications in industry and research, some additional features had to be added. In this way several different versions of Pascal have been developed [e.g. UCSD, OMEGASOFT, EYRING (PDOS), and P*, just to mention a few]. These Pascal versions are all based on 'Standard Pascal' and vary in their extensions. Such extra features usually permit the following:

- Low-level access (to computer and environment),
- accessibility of object addresses,
- ability to override the strong type-checking so important otherwise,
- access to part-words (bit fields),
- data-word manipulation,
- separate (modular) compilation,
- string handling,
- explicit type conversion.

Another extended Pascal with some basic syntactical deviations has been developed by N. Wirth and is known as Modula-2. Since the differences between both languages are small, the student will have no major problems in switching to Modula-2 after having studied Pascal.

Since the Pascal versions have 'Standard Pascal' in common, it is recommended to learn 'Standard Pascal' first and then the appropriate extension(s).

Consequently, the course has been organized in such a way that Standard Pascal is presented first. Thereafter, the extensions of OMEGASOFT-Pascal are given. The OMEGASOFT-Pascal has been adopted by CERN as a standard language for the efficient programming of embedded systems for real-time data acquisition and control using the MC6809 microprocessor. It is, however, now also available for the MC68000 microprocessor.

All programming examples are given in OMEGASOFT-Pascal.

To make the use of Boolean and SET-expressions more attractive, some fundamental knowledge of Boolean Algebra and of the Theory of Sets has been added; this should be skipped by the more advanced.

The course starts with the introduction of the concept 'Algorithm(s) + Data Structure(s) = Program' (N. Wirth) and examples of simple algorithms in the form of text and flow charts.
2. **ALGORITHM(S) + DATA STRUCTURE(S) = PROGRAM** (see N. Wirth)

Computer programming has been recognized as a discipline whose mastery is fundamental and crucial to the success of many engineering projects. (See N. Wirth).

*Programs are concrete formulations of abstract algorithms* based on particular representations and structures of *data*. (See N. Wirth).

Decisions about structuring data cannot be made without knowledge of the algorithms applied to the data. On the other hand, the structure and choice of algorithms often strongly depend on the structure of the underlaying data,

i.e. program composition and data structure are inseparably interwoven. (See N. Wirth, C. Hoare).

Algorithm = description of actions = statements.

Data structure = logical representation of the information inherent to the problem.

Any computer program can be divided into a data description part and an action part (concept of data and concept of action). The data description part defines the organization of the data (data structures). The action part is composed of performable instructions, called statements, which represent an algorithm. An algorithm is a method for a computation that processes the arranged data in a certain manner.

2.1 **Examples of algorithms**

**SOLUTION OF QUADRATIC EQUATIONS**

\[ x^2 + p_1 x + q = 0 \]

A. Read and store \( p_1, q \), \( i=1, ..., N \)

\[ l = 1 \]

Read \( p_i, q \)

End of file? Yes No

COUNT \[ l = l + 1 \]

No \( \rightarrow N_{max} \)

Yes \( \rightarrow N_{eq} \)

\[ N_{eq} = \text{No. of equations} \]

Note: It is assumed that at least one couple of coefficients is available on input.

\[ S \rightarrow \text{Go, solve} \]
Flow-chart

B. Solve

\[ x^2 + px + q = 0 \]

\[ x_{1,2} = \frac{-p \pm \sqrt{(p/2)^2 - q}}{2} \]

1. \( \rightarrow 1 \)

PHALF \( \leftarrow \frac{p}{2} \)
RACIAND \( \leftarrow (\text{PHALF})^2 - q \)
ROOT \( \leftarrow \sqrt{|\text{RACIAND}|} \)

Yes \( \rightarrow \)
No \( \rightarrow \text{RACIAND < 0} \)

Real

X1REAL \( \leftarrow -\text{PHALF} + \text{ROOT} \)
X2REAL \( \leftarrow -\text{PHALF} - \text{ROOT} \)
X1IMAG \( \leftarrow 0 \)
X2IMAG \( \leftarrow 0 \)

Complex

X1REAL \( \leftarrow -\text{PHALF} \)
X2REAL \( \leftarrow -\text{PHALF} \)
X1IMAG \( \leftarrow \text{ROOT} \)
X2IMAG \( \leftarrow -\text{ROOT} \)

Print \( p, q, x_1, x_1i, x_2, x_2i \)

I \( \leftarrow I + 1 \)

Count

Nreq \( \leftarrow \sqrt{\text{RACIAND}} \)

Yes \( \rightarrow \)
No \( \rightarrow \text{Repeat} \)

SQUARE ROOT ALGORITHM -- NEWTON'S METHOD \( x = \sqrt{y} \)

Consider the interval \( a_0b_0 \) whose product is \( y \). The interval must then contain \( \sqrt{y} \) as \( a_0 \cdot b_0 = \sqrt{y} \cdot \sqrt{y} = y \). If \( a_0 \) is increased to \( a_1 \) then \( b_0 \) must decrease to \( b_1 \) if \( a_1 \cdot b_1 = y \), i.e. the smaller interval contains \( \sqrt{y} \) as well.

The methods consist of finding a succession of smaller and smaller intervals, each containing \( \sqrt{y} \) and approaching zero:

If \( g \) (the first guess) is the starting point and if we choose \( y/g \) to be the initial end-point, the interval \( g(y/g) \) will contain \( \sqrt{y} \) as \( g \cdot (y/g) = \sqrt{y} \cdot \sqrt{y} = y \). If we choose \( (g + y/g)/2 \) to be the next guess \( g' \), then also the smaller interval \( g' \cdot (y/g') \) will contain \( \sqrt{y} \). Otherwise, if we write \( y = x^2 \) as \( y/x = x \), or

\[
\frac{y}{x} + x = 2x
\]

\[
\frac{[y/x]}{2} = x
\]
Block MAIN contains two subblocks (subprograms) S1 and S2.
S1 contains another subblock S11 which itself includes the subblock S111.
S2 comprises the two parallel subblocks S21 and S22.

<table>
<thead>
<tr>
<th>Block</th>
<th>Nesting level L</th>
<th>Objects declared in Block are accessible from blocks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>0</td>
<td>MAIN, S1, S11, S111, S2, S21, S22</td>
</tr>
<tr>
<td>S1</td>
<td>1</td>
<td>S1, S11, S111</td>
</tr>
<tr>
<td>S11</td>
<td>2</td>
<td>S11, S111</td>
</tr>
<tr>
<td>S111</td>
<td>3</td>
<td>S111</td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
<td>S2, S21, S22</td>
</tr>
<tr>
<td>S21</td>
<td>2</td>
<td>S21</td>
</tr>
<tr>
<td>S22</td>
<td>2</td>
<td>S22</td>
</tr>
</tbody>
</table>

2.4 Basic vocabulary

The basic vocabulary consists of basic symbols which can be letters, digits, and special symbols. Special symbols are operators and delimiters.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Delimiters (Reserved words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>:</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>*</td>
<td>=</td>
</tr>
<tr>
<td>/</td>
<td>&lt; &gt;</td>
</tr>
<tr>
<td>:</td>
<td>&lt; =</td>
</tr>
<tr>
<td>,</td>
<td>&lt; =</td>
</tr>
<tr>
<td>;</td>
<td>&gt;</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>&gt;</td>
<td>=</td>
</tr>
<tr>
<td>:</td>
<td>=</td>
</tr>
</tbody>
</table>

or: and end nil set
array etc.

Note: where { and } are not available (* and *) may be used instead.

Delimiters are reserved words. They may therefore not be used as object names by the programmer.

IDENTIFIERS are names denoting objects. An identifier must start with a letter followed by any combination of letters and digits.

legal: SUM WORD2 H4X
illegal: 3SUM INTEGER

Some identifiers are predefined standard identifiers: Example: SIN, COS, SQRT.

SEPARATORS: Blanks, ends-of-lines, and comments are assumed to be separators. Any number of separators may be specified between two consecutive symbols except within identifiers, numbers, or special symbols. However, consecutive pairs of identifiers, numbers, or word symbols must be separated.
Example: not PROGRAMVOLTS but PROGRAM VOLTS.

COMMENTS: Comments may be any text enclosed in { }. They have no meaning in relation to the program. However, they are useful for human readers as they enhance the readability of a program.

Example: { READ again if voltage too low }

Note: Where { and } are not available (* and * ) may be used instead.
Example: (* branch if negative * ).

STRINGS: Strings are sequences of characters enclosed by single quotes, To represent a single quote in a string, it must be specified twice.

Examples: ‘y’, ‘START’, ‘’
↑
to specify ‘

2.5 Rules, Syntax
For a more precise description of the rules, syntax diagrams or textual syntax formulation may be used.

SYNTAX DIAGRAM
In such diagrams a rounded box is used to represent predefined words and standard identifiers such as PROGRAM. For the representation of reserved symbols (e.g. : , ; ) a circle is used. Rectangular boxes are used to reference syntax elements that are defined by their own diagram elsewhere. The correct flow (path) is shown by lines with arrows.

Example: program heading syntax

![Syntax Diagram]

TEXTUAL SYNTAX FORMULATION (Backus–Naur form)
This form denotes syntactic constructs by English words enclosed in angular brackets ⟨ ⟩. Repetition of sequences of constructs is described by enclosing a sequence between { and }.

Example: program heading syntax.

⟨program⟩ ::= ⟨program heading⟩ ⟨block⟩
⟨program heading⟩ ::= program
⟨identifier⟩ ⟨(⟨file identifier⟩
   , ⟨file identifier⟩)⟩;
⟨file identifier⟩ ::= ⟨identifier⟩.

Example of a real program heading:

PROGRAM ALFRED (INPUT, OUTPUT);
3. THE CONCEPT OF DATA

The data to be processed by the program algorithm can be organized in single units—called scalar data types—or in ordered sets of multiple elements—called structured data.

The scalar data types will be our first concern while the structured data types will be introduced later in a special chapter.

3.1 Scalar data types

Scalar data types are the basic types of data that can be associated with a variable. The declaration of a scalar type is simply a list of the values that may be assumed by a variable of that type. For instance, the declaration

\[
\text{TYPE INTEGER} = (-\text{MAXINT}, \text{MAXINT})
\]

introduces the data type INTEGER for the representation of the whole numbers

\[-\text{MAXINT} .. -3, -2, -1, 0, 1, 2, 3, 4 .. \text{MAXINT},
\]

where the limit MAXINT depends on the particular computer.

Four standard scalar data types are predefined. They permit the representation of whole and decimal numbers, logical conditions, and textual characters.

3.2 Standard (predefined) scalar types

These standard scalar types, of which there are four, are listed below:

\[
\begin{align*}
\text{BOOLEAN} \\
\text{INTEGER} \\
\text{REAL} \\
\text{CHAR}
\end{align*}
\]

3.2.1 The type Boolean

Definition: \text{TYPE BOOLEAN (FALSE, TRUE)};

This is a scalar type whose elements are the predefined constants TRUE and FALSE. Ordering is such that TRUE > FALSE.

Boolean operators are:

\[
\begin{align*}
\text{AND} & \quad \text{logical conjunction} \\
\text{OR} & \quad \text{logical disjunction} \\
\text{NOT} & \quad \text{logical negation}
\end{align*}
\]

The relational operators \((=, <>, <, >, \leq, \geq)\) result in a Boolean value. Since TRUE > FALSE, all other common logical operations can be expressed using the relational operators.

Example:

\[
\begin{align*}
\text{Equivalence} & \quad x \approx y \quad \text{as} \quad x = y \\
\text{Exclusive OR} & \quad x \oplus y \quad \text{as} \quad x < > y \\
\text{Negative implication} & \quad x \not\subset y \quad \text{as} \quad x > y
\end{align*}
\]
STANDARD BOOLEAN FUNCTIONS

- ODD(x) yields true if the integer x is odd, else false;
- EOLN(F) true if end-of-line found, else false;
- EOF(F) true if end-of-file found, else false.

BOOLEAN ALGEBRA

Basic operators:  
\( \land \) AND logical conjunction  
\( \lor \) OR logical disjunction  
\( \neg \) NOT logical negation

Values:
- \( T = \text{TRUE} \)
- \( F = \text{FALSE} \)

Truth tables to represent logical functions

<table>
<thead>
<tr>
<th>x AND y</th>
<th>x</th>
<th>y</th>
<th>x&amp;y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x OR y</th>
<th>x</th>
<th>y</th>
<th>x\lor y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x \oplus y</th>
<th>x</th>
<th>y</th>
<th>x\oplus y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x \equiv y</th>
<th>x</th>
<th>y</th>
<th>x\equiv y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x \not\equiv y</th>
<th>x</th>
<th>y</th>
<th>x\not\equiv y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Exclusive OR  
Equivalence  
Neg. implication

Some laws:
- \( x \& F = F \)
- \( x \& T = x \)
- \( x \lor F = x \)
- \( x \lor T = T \)
- \( x \& x = x \)
- \( x \lor x = x \)

Redundancy laws
- \( x \& \overline{x} = F \)
- \( x \lor \overline{x} = T \)

Commutative laws
- \( x \& y = y \& x \)
- \( x \lor y = y \lor x \)
Associative laws
(x&y)&z = x&y&z = x&(y&z)
(x∨y)∨z = x∨y∨z = x∨(y∨z)

Distributive laws
(x&y)∨(x&z) = x&(y∨z)
(x∨y)&(x∨z) = x∨(y&z)

Involution law
\bar{\bar{x}} = x \quad \text{(NOT(NOT } x) = x \text{ )}
further:
\bar{x} \lor z = \bar{x} \land \bar{z}
\bar{x} \land z = \bar{x} \lor \bar{z}

Examples
7 = 8 \rightarrow \text{FALSE} \quad \text{NOT}(Y > Z) = Z > = Y
11 = 11 \rightarrow \text{TRUE} \quad \text{NOT}(U = V) = U < > V
13 > 12 \rightarrow \text{TRUE} \quad \text{NOT}(X < = Y) = X > Y

3.2.2 The type Integer
Definition: TYPE INTEGER (-MAXINT,MAXINT);

This is a scalar type whose elements are the elements of the machine-dependent set of whole
numbers (integers), described by the enumeration -MAXINT, MAXINT, where MAXINT is the
maximally representable integer determined by the computer architecture.

The arithmetic operators resulting in an integer value when applied to integer operands are:

* Multiply
DIV Divide and truncate (value not rounded)
MOD A mod B = A − ((A DIV B) * B)
+ Add
− Subtract

The relational operators (<, >, < =, <, >, > =) result in a Boolean value when used with
integer operands.

Standard functions returning an integer value:

ABS(x) Returns the absolute value of x.
SQR(x) Returns the square of x.
TRUNC(x) Returns the integer part of the real number x. \text{TRUNC}(4.9) = 4 \quad \text{TRUNC}(-4.9) = -4.

ROUND(x) Returns the rounded value of the real number x as integer.
\text{ROUND}(x) \text{ means for } x \geq 0 \quad \text{TRUNC}(x + 0.5)
\text{ and for } x < 0 \quad \text{TRUNC}(x - 0.5)

The function
\text{SUCC}(K) \text{ returns the next integer } K + 1

and
\text{PRED}(K) \text{ returns the preceding integer } K - 1, \text{ where } K = \text{ integer.}
Note: For integer handling, the notation of $K + 1$ and $K - 1$ is clearer and therefore preferable.

Example of valid integer values:

$$
\begin{align*}
&5 \\
&15 \\
&0 \\
&-80 \\
&1 \\
&-7
\end{align*}
$$

3.2.3 The type Real

Real numbers are the elements of the machine-dependent set of decimal numbers. If at least one of the operands is of type Real (the other may be of type Integer) the following operators result in a real value:

- Multiply
- / divide (both dividend and divisor may be of type Integer, yet the result will be real!)
- + add
- - subtract

Standard functions resulting in a real value:

1. Real argument

$$
\begin{align*}
\text{ABS}(x) & \quad \text{Absolute value of } x \\
\text{SQR}(x) & \quad \text{Square of } x
\end{align*}
$$

2. With real or integer argument

$$
\begin{align*}
\text{SIN}(x) & \quad \text{trigonometric function sine} \\
\text{COS}(x) & \quad \text{trigonometric function cosine} \\
\text{ARCTAN}(x) & \quad \text{inverse trigonometric function arctangent} \\
\text{LN}(x) & \quad \text{natural logarithm} \\
\text{EXP}(x) & \quad \text{exponential function} \\
\text{SQRT}(x) & \quad \text{square root}
\end{align*}
$$

Note: Although real belongs to scalar type there are some restrictions where discrete numbers are required:

A real cannot be specified as argument to SUCC and PRED.
A real cannot be used as an array index.
A real cannot be used to control a FOR statement.
A real cannot be the base type of a set.

The reason for this is that in the above mentioned cases only discrete values e.g. $-1,0,1,2,3$ etc. are acceptable.
Examples of real numbers:
0.107
-36.0
+0.1333
-7.5E-8 (-0.00000075)
+3.16E+3
3E2 (300)
17

Note: Real constants do not need a decimal point. If, however, the point is specified, it must be preceded and succeeded by at least one digit.
Example: .8 7. 6.1E are wrong.

3.2.4 The type Char
The elements of the character type Char are all the characters which are represented in the character set of a computer implementation. The character set is a finite ordered set of the capital latin letters, A, ..., Z in alphabetical order, the numerically ordered and contiguous set of the decimal digits 0, ..., 9, the blank character, and special characters.
Internally the characters are represented by the non-negative integers 0,1,2,3, ...,n-1, where n is the number of distinct characters available. The mapping of the characters onto their internal number representation is not standardized and may vary from computer to computer.
Char: elements are indicated by enclosing the character in apostrophes (single quotes).

Examples of character constants:
‘+’ ‘A’ ‘W’ ‘Y’ ‘’

(single quote is represented by writing it twice).

There are four standard functions that are especially useful for character processing:
Two functions to allow the mapping of the given character set onto a set of positive numbers and in the other direction. These numbers are called the character ordinal number.
Transfer functions ORD and CHR:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORD(C)</td>
<td>This is the ordinal function returning the ordinal of character C.</td>
</tr>
<tr>
<td>CHR(I)</td>
<td>This is the character function. It is the converse of ORD. Returns the character corresponding to ordinal I in the character set.</td>
</tr>
</tbody>
</table>

Therefore:
CHR(ORD(C)) = C and ORD(CHR(I)) = I

Predecessor and Successor functions:
PRED(C) = CHR(ORD(C) - 1) preceding character
SUCC(C) = CHR(ORD(C) + 1) succeeding character

Note: PRED(C) and SUCC(C) may yield an undefined value if such a character does not exist.
4. PROGRAM HEADING AND DECLARATION PART

A PASCAL program consists of a program heading and a block (or body). The block is composed of a data description part and an action part. The data description part contains the object declarations and definitions for the scope of the program. The action part is the dynamic portion of the program. It comprises the statements, i.e. the executable instructions of the program.

\[
\langle \text{program} \rangle \ ::= \langle \text{program heading} \rangle \langle \text{block} \rangle \\
\langle \text{block} \rangle \ ::= \langle \text{label declaration part} \rangle \\
\quad \langle \text{constant definition part} \rangle \\
\quad \langle \text{type definition part} \rangle \\
\quad \langle \text{variable declaration part} \rangle \\
\quad \langle \text{procedure and function declaration part} \rangle \\
\quad \langle \text{statement part} \rangle
\]

PROGRAM HEADING

The heading identifies the program and specifies the parameters through which the program communicates with the environment (see program heading and external files):

\[
\langle \text{program heading} \rangle \ ::= \text{program } \langle \text{identifier} \rangle \\
\quad \langle \text{file identifier} \rangle \\
\quad [\langle \text{file identifier} \rangle]
\]

Example:

PROGRAM XCYS(INPUT,OUTPUT);

4.1 Label declaration part

Any statement may be labelled by writing a decimal number followed by colon (the label) in front of it. The purpose of labelling is to identify a particular statement which than can become the destination of a GOTO statement. The label must be defined in a label declaration. Its general form is:

\[
\text{label} \langle \text{label} \rangle \{, \langle \text{label} \rangle \};
\]

i.e. several labels may be associated with the same statement. A label is an unsigned integer consisting of up to four digits.

Example:

LABEL 1, 100, 999;

4.2 Constant definition part

A constant definition permits the identification of a constant by a more convenient name. The general form is:

\[
\text{CONST} \langle \text{identifier} \rangle = \langle \text{constant} \rangle; \\
\quad [\langle \text{identifier} \rangle = \langle \text{constant} \rangle;]
\]

A constant may be a number, a constant identifier (signed or unsigned), or a string.

By means of constant identifiers a program becomes more legible and a certain autodocumentation is assured. Constants are then centralized at the beginning of the program and may be easily inspected or changed.
Example:

```
PROGRAM ANY (INPUT,OUTPUT);
CONST PI = 3.1416;
VAR Y,X,Z,D,F,OMEGA : REAL;
BEGIN
  ...
  Y := SIN(2*PI*OMEGA);
  ...
  Z := PI/2 + F;
  ...
  X := D*PI;
  ...
END.
```

4.3 Type definition part

The purpose of the type definition is to introduce to a program a data type that is not yet defined. The general form is:

```
TYPE <identifier> = <type>; [<identifier> = <type>;]
```

Predefined types and identifiers (standard identifiers):

All standard types and identifiers may be redefined within a program.

Example:

```
TYPE VECTOR = ARRAY [1..5] OF INTEGER;
DIRECTION = (FORWARD, BACKWARD);
```

4.4 Variable declaration part

Before a variable can be used in a program, it must be declared.

In a variable declaration a variable identifier and a data type are connected with a new variable. The identifier must be followed by its type.

General form:

```
VAR <identifier>[, <identifier>]::<type>; 
  [<identifier>[, <identifier>]::<type>;]
```

Examples:

```
VAR I,J,K,L : INTEGER;
GOT : BOOLEAN;
KEYCHAR : CHAR;
X,Y,Z,A : REAL;
```

The variable declaration that binds an identifier to a type remains valid for the whole block in which it occurs unless the same identifier is redefined locally inside a subordinate block (subblock). If it is redefined in an inner block it will be valid only for this subordinate block, and it will be independent of the item of the same name declared in the outer block, which will then not be accessible from inside the inner block.

*Note:* Declaring an identifier more than once in the same block is not allowed.
4.5 *Subprogram declarations*

Procedure and Function declaration part.

Any subprogram must be declared prior to its use. Procedure subprograms are invoked by means of procedure statements. Function subroutines yield a result value upon return from the call. They may therefore be used as components of expressions. See subprograms.
5. INPUT AND OUTPUT OF DATA

5.1 Input and output on standard devices

INPUT:

Procedure READ
READ(X,Y,Z); will read three variables from the standard input device (terminal, card reader).

Procedure READLN
READLN(X,Y,); will read two variables from the standard input device and skip the rest of the line. (The values of X and Y may, however, stretch over several lines.)

Boolean function EOLN
EOLN yields a Boolean value:
TRUE if end-of-line is encountered,
FALSE if end-of-line is not detected.

Example: IF EOLN THEN ... .

Boolean function EOF
EOF yields a Boolean value:
TRUE if end-of-file is encountered (input exhausted),
FALSE if no end-of-file is found.

OUTPUT:

Procedure WRITE
WRITE(X,Y,Z); will write the three variables onto the current output device and will assume further output to continue on the present line.

Procedure WRITELN
WRITELN; will simply skip to the next line.
WRITELN(X,Y,Z) = WRITE(X,Y,Z);
WRITELN;
i.e. the values of the three variables are output and a skip to the next line is performed.

Examples:

WRITE('VOLTAGE = ', VOLTS); [VOLTAGE = 180, stay on the line]
WRITELN('SQUARE OF BASE ', BASE, ' IS: ', SQR(BASE));

Example:
SQUARE OF BASE 6 IS 36 ↓ skip to next line.

WRITE and WRITELN may be used with additional parameters (natural numbers) denoting the desired maximal field width (No. of characters).

1. WRITE (X:4); [or WRITELN]
   This would present the value of X in a field of four characters.
2. WRITE (Y:10:5); [Y must be of type real!] or
   WRITELN (Y:10:5);
   This would present the value of Y in a field of maximal ten characters with five digits after the decimal point.
PROGRAM CHARCODE (INPUT,OUTPUT);
[Read a datum that contains a single character. Print the character and its numerical representation (its character code). Type the character and RETURN.]

VAR
    CHARACTER  :  CHAR;
    CODE        :  INTEGER;
BEGIN     [EXECUTABLE PROGRAM STARTS HERE]
    READ       (CHARACTER);
    CODE :=    ORD(CHARACTER);
    Writeln   (CHARACTER,' HAS THE CODE:', CODE);
END.

Assuming you typed A, the result might be:
CHARACTER = A  HAS THE CODE:  65

PROGRAM COMPARE (INPUT,OUTPUT);
[Read two data of integers. Compare the first number with the second and indicate whether each of the tests LESS THAN, EQUALS, and GREATER THAN is true or false.]

VAR    A,B : INTEGER;
    LESS, EQUALS, GREATER : BOOLEAN;
BEGIN   [INPUT AND ECHO PRINT THE DATA]

    READ(A,B);
    Writeln('A = ', A, ' AND B = ', B);
    [Perform the three relational comparisons]
    LESS := A < B;  EQUALS := A = B;
    GREATER := A > B;
    [Write the results of the comparisons]
    Writeln('A < B IS', LESS);
    Writeln('A = B IS', EQUALS);
    Writeln('A > B IS', GREATER);
END.

If this program was executed with the following data pair
3    7
what would the output be?

PROGRAM WRSQROOT (INPUT,OUTPUT);
[Read a datum that contains a non-negative real number. Compute and write (print) the square root of that value]

VAR
    INPUTVALUE, SQROOT : REAL;
BEGIN
READ (INPUTVALUE);
SQROOT := SQRT (INPUTVALUE);
[Note: Prior to invoking the square root function we ought to check INPUTVALUE to be sure the argument is positive. This can be done with a conditional statement to be discussed later.]
WRITELN ('X = ', INPUTVALUE,
'SQUARE ROOT OF X = ', SQROOT);
END.

Note: This program does not only print the result of the computation but also the input data. Moreover, some identifying text is added. It is recommended to use both these techniques wherever possible as they enhance the readability of the output.

5.2 Style of output

Very often the user of a program is not identical with its author. In these cases the user will not be concerned with the quality of the underlying algorithm. Rather he wants precise answers to his problems in a well readable presentation. If it is not easy to understand a graphic output on a display or the output of text and numbers on a printed listing, then the program will be used rarely if at all. Moreover, the programmer himself might not understand the results of his program anymore when he glances at the commentless results of his product say six months or later after completion of the program.

Therefore, although the main point is to produce an errorfree program, the second concern should be to achieve a well readable clear output presentation.

As an example, output of the following format is insufficient:

881861 .15739E02 .13900E04

Rather, the output should be annotated and explained:

OCTANT 8  SECTOR 81  GAUGE: VG861
INPUT VOLTAGE = 15.7 V
FREQUENCY = 1390 Hz
6. THE CONCEPT OF ACTION (STATEMENTS)

The performable part of a program consists of instructions that provide for action.

The instructions written in a high-level language to govern the flow of the execution of a computer program and translated by the compiler into a machine language (object program), are called statements. A statement may be an instruction to store the result of a computation or a known value into a memory cell (assignment statement), an instruction to execute a pre-defined sub-process (procedure call), an order to transfer unconditionally or conditionally the execution of a program from the instantaneous point to some other destination in the program (GOTO statement, Conditional statement) or an instruction for the repeated execution of other statements (FOR-, WHILE-, or REPEAT-statement).

The classification of statements is given below.

1) The assignment statement and expressions.
2) The compound statement.
3) Repetitive statements:
   The WHILE statement.
   The REPEAT statement.
   The FOR statement.
4) Conditional statements:
   The IF statement.
   The CASE statement.
5) Unconditional statements:
   The GOTO statement.
6) Procedure statements

Statements are usually executed in the order in which they are written, except that a GOTO statement may interrupt this sequence without resumption and a conditional statement may cause a group of following statements to be omitted. A statement may be preceded by a label to allow references by a GOTO statement.

Statements that do not contain other statement constructs are called simple statements (Assignment, GOTO and Procedure statement). A special case of the simple statement is the empty statement which does not contain any symbol.

6.1 The Assignment statement and expressions

The most essential statement for the modification of data is the assignment statement. It is used to evaluate a new value from an expression given on the right-hand side of the assignment symbol := and to store it to the reference on the left-hand side. This reference may be a variable of any type excluding file. As a result of an assignment statement the quantity on the left side will take the assigned value until it is either changed by another assignment or it goes out of scope. Left and right side must be of the same type with the only exception that an integer value may be assigned to a real variable with implicit type conversion.

The form is: <variable> := <expression>.

Formal syntax:
Example:

\[ Y := 7; \]

This means: Replace the actual contents of \( Y \) by the constant 7.

One can also say \( Y \) 'becomes' 7.

An expression is any combination of basic operators and function calls connected by operation symbols to yield a certain computational process. Evaluation of an expression is carried out from left to right according to operator precedence. If an expression is enclosed in parentheses it will be evaluated independently of adjacent operators.

Examples of expressions:

\[
\begin{align*}
2*3-4*5 &= (2*3) - (4*5) = -14 \\
15 \text{ DIV } 4*4 &= (15 \text{ DIV } 4)*4 = 12 \\
80/5/3 &= (80/5)/3 = 5.333 \\
4/2*3 &= (4/2)*3 = 6.000 \\
\text{SQRT(SQR}(3)+11*5)) &= 8.000 \\
K < 3 \\
(L > 4) \text{ AND } (T = 5) \\
\text{NOT (FOUND)}
\end{align*}
\]

Operator precedence table

Highest precedence
(treated first)

Expressions in parentheses
\( \text{NOT, } \neg \) (unary negation)
\( */\text{DIV MOD AND} \)
\( + - \text{ OR} \)
\( = < > < = > = > \text{ IN} \)

Lowest precedence
(treated last)

Assignment examples

\[
\begin{align*}
z &= 5; \\
K &= K + 1; \\
\text{COUNT} &= \text{COUNT-1}; \\
Y &= \text{SIN (OMEGA*T)}; \\
\text{GOT} &= C > D; \\
\text{MARK} &= A = B; \\
\text{ROOT} &= \text{SQRT (RADICAND)}; \\
z &= (E^3)/1.91; \\
\text{GOT} &= \text{FALSE}; \\
\text{MARK} &= (A = B) \text{ AND } (C = 3); \\
\text{INDIC} &= \text{NOT}((Y < 4) \text{ AND } (Z = T)); \\
\text{INDEX} &= (Y = 4) \text{ OR } (Z < T);
\end{align*}
\]
6.2 The Compound statement

A sequence of statements of any kind may be grouped to form a single statement. This is for instance convenient to avoid GOTO statements in conditional statements.

A compound statement consists of the sequence of statements enclosed between BEGIN and END. The body of a PASCAL program can be considered as a compound statement.

Example

BEGIN
  K := 40;
  J := 105;
END;

A compound statement may be used wherever a statement is allowed.

6.3 Repetitive statements

6.3.1 The REPEAT statement

The REPEAT statement can be used to repeat a group of statements. The informal syntax is:

REPEAT statement(s) UNTIL (condition is true)

Formal syntax:

The statement sequence enclosed in REPEAT and UNTIL is executed at least once. After every execution the value of the Boolean expression is computed and repetition stops as soon as this value is false.

6.3.2 The WHILE statement

The WHILE statement can be used for the repeated execution of a statement. The informal syntax is:

WHILE (Boolean expression is true) DO statement.
Formal syntax:

```
WHILE Boolean expression DO Statement
```

The iteration of the statement following the symbol DO is controlled by the Boolean expression. Any execution of the statement will take place only if the value of this expression is true.

**Repetition with predetermined number of iterations**

### 6.3.3 The FOR statement

The FOR statement permits the iterative execution of a given statement (the controlled statement) for a sequence of values of a chosen variable (the control variable).

Its informal syntax is:

```
FOR control variable := initial value TO DOWNTO final value DO statement.
```

Formal syntax:

```
FOR Variable := Expression TO DOWNTO
Expression DO Statement
```

The type of the control variable as well as initial and final value must be a scalar except real. Prior to executing the FOR loop initial and final value are computed and the control variable is set to the initial value. The controlled statement will be executed in the case of TO (DOWNTO) only if the control variable is less or equal (greater or equal) than the final value otherwise the FOR loop will be exhausted. After every execution the control variable is increased (decreased) by 1.

Note: After exhaustion of the FOR loop the value of the control variable is undefined!
PROGRAM PROD10R(INPUT,OUTPUT); [PRODUCT OF THE FIRST 10 INTEGERS $\prod_{i=1}^{10} i$ USING REPEAT ... UNTIL]

VAR
    PROD, NUMBER : INTEGER;

BEGIN
    PROD := 1; [INITIALIZE PROD TO ONE]
    NUMBER := 1; [INITIALIZE NUMBER TO 1]

    REPEAT
        PROD := PROD*NUMBER; [multiply]
        NUMBER := NUMBER + 1; [increase number]
    UNTIL NUMBER > 10;

    WRITELN(‘THE PRODUCT OF THE FIRST 10 INTEGERS IS’, PROD);

END.

----------------------------------------------------------------------------------

PROGRAM PROD10W(INPUT,OUTPUT); [PRODUCT OF THE FIRST 10 INTEGERS $\prod_{i=1}^{10} i$ USING WHILE AS REPETITION STATEMENT]

VAR PROD, NUMBER : INTEGER;

BEGIN
    PROD := 1; [INITIALIZE PROD TO ONE]
    NUMBER := 1; [INITIALIZE NUMBER TO 1]
Example: A program using IF

PROGRAM WRSQROOT (INPUT,OUTPUT);
[READ A DATUM THAT CONTAINS A REAL NUMBER. COMPUTE AND WRITE (PRINT) THE SQUARE ROOT OF THAT VALUE]

VAR INPUTVALUE, SQROOT, VALUE: REAL; POSITIVE: BOOLEAN;
BEGIN
  READ(INPUTVALUE);
  VALUE := INPUTVALUE;

  POSITIVE := TRUE;  [initialize marker to positive]
      IF VALUE = 0 THEN SQROOT := 0 ELSE
      BEGIN
        IF INPUTVALUE < 0 THEN
            BEGIN
              POSITIVE := FALSE; [set marker to not positive]
              VALUE := –VALUE; [GET MAGNITUDE]
            END;  [IF NEGATIVE]
            SQROOT := SQRT(VALUE);
        END; [IF NON ZERO]

        WRITE(‘X=’, INPUTVALUE, ‘SQUARE ROOT OF X=’, SQROOT);
      IF POSITIVE = TRUE THEN WRITELN
        ELSE WRITELN (‘I’);
      END.

When either or both the consequence statement and the alternative statement are compound statements it is important to position the parts of the IF-THEN-ELSE construct so that the meaning of the whole can be easily understood. Also, as it is possible to nest IF statements to any depth, a well-legible layout is mandatory for the understanding of an algorithm by humans. It is therefore suggested to use some kind of indentation scheme:

IF condition THEN
    BEGIN
      ;
      END
ELSE
    BEGIN
      ;
      END;

IF condition
    THEN BEGIN
      ;
      END
ELSE BEGIN
    ;
    END;
IF condition THEN
BEGIN
  statements
  
  statements
END
ELSE
BEGIN
  statements
  
  statements
END;

IF c_1 THEN s_1
ELSE IF c_2 THEN s_2
ELSE IF

  .

ELSE IF c_{k-1} THEN s_{k-1}
ELSE s_k,

where c_{1,2..k} = conditions
s_{1,2..k} = statements

Instead of the above layout for nested IF statements, which is quite clear but has the disadvantage of reaching the right limit of a display, card, etc. fairly quickly, an alternative layout would be:

IF c_1 THEN s_1
ELSE IF c_2 THEN s_2
ELSE IF

  .

ELSE IF c_{k-1} THEN s_{k-1}
ELSE s_k

Note: Multiple IF statements of the form

IF c_1 THEN IF c_2 THEN s_1 ELSE s_2

which look ambiguous are interpreted as if the innermost IF would have been enclosed between BEGIN and END:

IF c_1 THEN
  BEGIN IF c_2 THEN s_1
  ELSE s_2
END

i.e. an ELSE is always associated with the most recent IF!

To enhance the readability of a program it is however recommended to use appropriate BEGIN/END brackets.
Note further:

**Warning 1**
There is never a semicolon before an ELSE!
Consequently the following IF-statement

\[
\text{IF } A < B \text{ THEN}
\begin{align*}
Z & := V[I]; \\
W & := Z \cdot 3;
\end{align*}
\text{END; ELSE } Z := U[I]; \quad \text{is wrong!}
\]

**Warning 2**
An even more misleading example is the following:

\[
\text{IF } A < B \text{ THEN ;}
\begin{align*}
& \quad \quad ; \\
& \quad \quad \end{align*}
\text{END}
\]

Here, the statement **controlled** by the IF is the **empty statement** between THEN and the semicolon; consequently the **compound** statement after the IF will **always** be executed!

**Recommendation**
Do not use the IF in the case where you want to assign alternative values to a Boolean variable.

**Example:**

\[
\text{IF } A = B \text{ THEN } \text{GOT : = TRUE ELSE GOT : = FALSE;}
\quad \text{is a correct statement but}
\text{GOT : = } A = B; \\
\quad \text{is much easier and usually faster.}
\]

**Examples**

\[
\begin{align*}
\text{IF } X < Y \text{ THEN } Z := Z + 1; \\
\text{IF } (X < Y) \text{ AND } (X < Z) \text{ THEN } W := 0; \\
\text{IF NOT } ((A < C) \text{ OR } (C = D)) \text{ THEN } W := \text{WEEK};
\end{align*}
\]

\[
\begin{align*}
\text{IF } C = E \text{ THEN} \\
\quad \text{BEGIN} \\
\quad \quad F := \sin(E); \\
\quad \quad Z := E + F \cdot \omega; \\
\quad \text{END} \quad \text{ELSE } F := \cos(E);
\end{align*}
\]

IF FREQUENCY > 400 THEN
    IF FREQUENCY > 4000 THEN
        IF FREQUENCY > 20000 THEN
            ERRMESSAGE('FREQUENCY TOO HIGH,F = ',FREQUENCY,' Hz')
        ELSE SETBAND(UPPER)
    ELSE SETBAND(MEDIUM)
ELSE ERRMESSAGE('FREQUENCY TOO LOW,F = ',FREQUENCY,' Hz');
This nested IF represents a Binary Tree whose flow chart equivalent is the following:

An alternative to the preceding binary tree checking would be the following:

```
IF FREQUENCY <= 400 THEN ERRMESSAGE('FREQUENCY TOO LOW,F = ',FREQUENCY,'Hz')
ELSE IF FREQUENCY > 20000 THEN ERRMESSAGE('FREQUENCY TOO HIGH,F = ',FREQUENCY,'Hz')
ELSE IF FREQUENCY > 4000 THEN SETBAND(UPPER)
ELSE RANGE := SETBAND (MEDIUM);
```

P.S. Compare the two solutions.
6.4.2 The CASE statement

The CASE statement permits a program code to select one statement for execution out of a set of statements. On entry to a CASE statement only one of the possible statements contained in the CASE will be executed, the others being neglected. The selection of the particular statement to be executed depends on the value of the selector expression in conjunction with a set of constants.

**Definition:**
The CASE statement combines a selector expression with a list of statements tagged with a label of the same type as the selector. Since statement selection depends on discrete values the type of the selector must be a scalar except REAL. Program control is transferred to the statement the label of which is equal to the actual value of the expression. Warning: If no corresponding label is specified the place of program continuation is undetermined. After execution of the chosen statement: in the list the program part following the END of the CASE statement is resumed. The END of the CASE is a special END that is an exception from the usual BEGIN-END formalism.

The formal syntax of CASE is:

![CASE Syntax Diagram]

**Note:** The labels in a CASE statement are not accessible by a GOTO statement. They are special labels suitable only for statement selection within the CASE statement. They may be stated in any order but each label must appear only once.

**Conclusion:** The CASE statement represents a multiway branch according to the value of a selector where all branches end at the common continuation of flow denoted by the END of the CASE statement.

**FLOW CHART (example)**

![Flow Chart Example]
CASE EXAMPLES

PROGRAM CASE1 (INPUT, OUTPUT);
VAR MONTH : INTEGER;
BEGIN

WRITELN ("+++ TYPE THE NUMBER OF A MONTH (1, ..., 12): ")
READLN (MONTH);
IF (MONTH > 12) OR (MONTH < 1) THEN
  WRITELN ("** ERROR: NO SUCH MONTH!**")
ELSE
  CASE MONTH OF
  1: WRITELN ("JANUARY");
  2: BEGIN
       WRITELN ("FEBRUARY");
       WRITELN ("IN FRENCH: FEVRIER");
       WRITELN ("IN GERMAN: FEBRUAR");
    END;
  3: WRITELN ("MARCH");
  4: WRITELN ("APRIL");
  5: WRITELN ("MAY");
  6: WRITELN ("JUNE");
  7: WRITELN ("JULY");
  8: WRITELN ("AUGUST");
  9: WRITELN ("SEPTEMBER");
 10: WRITELN ("OCTOBER");
 11: WRITELN ("NOVEMBER");
 12: WRITELN ("DECEMBER");
    END [CASE]
END.

Assume VAR I: INTEGER; CH: CHAR;

CASE I OF
  0: X := 0;
  1: X := SIN (Z); [SINE OF Z]
  2: X := COS (Z); [COSINE OF Z]
  3, 4: X := EXP (Z); [EXPONENTIAL FUNCTION OF Z]
  5: X := LN(Z); [NATURAL LOGARITHM OF Z]
END [CASE I]

CASE CH OF
  'A', 'B', 'C' : CH := SUCC (CH); [TAKE NEXT]
  'D', 'E' : CH := PRED (CH); [TAKE PRECEDING]
  'F', 'G' : [NULL CASE I.E. NO ACTION]
END;

Note: 1. Several labels may be specified for the same action.
  2. A Null CASE may be specified that results in no particular action at all, i.e. merely the program after END of CASE will be resumed.
6.5 The GOTO statement

A GOTO statement is used for unconditional transfer of control from one part of the program to another. The next statement to be executed is referenced by a label:

\[ \text{GOTO} \langle \text{label} \rangle \]

Each label (an unsigned integer that is at most four digits) must appear in a label declaration prior to its occurrence in the program body. The label declaration must be the first declaration in a program.

**Example:**

```
PROGRAM PROG (INPUT, OUTPUT);
LABEL 999;
VAR K, L, M, N : INTEGER;

BEGIN
  ...
  IF N<0 THEN
  BEGIN
    WRITELN ('**ERROR : NUMBER OF ITEMS NEGATIVE**');
    GOTO 999; (*TERMINATE THE PROGRAM*)
  END;

999:
```

**SCOPE OF THE GOTO STATEMENT**

- **Single blocks**
  - \[ \text{begin} \]
  - \[ \text{level} K \]
  - \[ \text{7} ; \text{statement} \]
  - \[ \text{GOTO} 7 \]
  - \[ \text{end} ; \]
  - \[ \text{allowed, but bad programming style} \]

- **Parallel blocks**
  - \[ \text{begin} \]
  - \[ \text{level} K \]
  - \[ \text{5} ; \text{statement} \]
  - \[ \text{end} ; \]
  - \[ \text{allowed} \]

- **Nested blocks**
  - \[ \text{begin} \]
  - \[ \text{level} K \]
  - \[ \text{GOTO} 3 ; \]
  - \[ \text{end} ; \]
  - \[ \text{allowed} \]

- **Not allowed**
  - \[ \text{begin} \]
  - \[ \text{level} K \]
  - \[ \text{GOTO} 5 ; \]
  - \[ \text{end} ; \]
The GOTO statement is a simple and powerful statement that unconditionally transfers control from one part of the program to another. Since the use of GOTOs causes a loss of block structuring (tendency to spaghetti programming) and therefore makes it difficult to follow the program flow, GOTO statements are not recommended in PASCAL. Instead the structured constructs REPEAT, WHILE, compound statements and EXIT (OMEGASOFT–PASCAL only) should be used. Especially in cases where a GOTO branches backward these constructs can easily replace the GOTO. If a GOTO is used the programmer should use comments to explain the situation.

Theoretically a GOTO statement is not needed at all in a Pascal program. Yet in certain circumstances it can be awkward to achieve a certain effect without a GOTO statement. The only reasonable application of GOTO statements is in exceptional conditions where an unsolvable situation is to be broken (see example above). This can be meaningful in fairly long programs where one wishes to branch to the end of the program to terminate it.

Note, however, that termination of a program could be achieved more elegantly by calling, say, a procedure FINISH that does not return but finishes execution.

**Examples**

1. Assume we have in a program the following declarations:

   ```pascal
   VAR DAY : INTEGER;
   DAYFOUND : BOOLEAN;
   DAYNAME : STRING [10];
   DAYNAMES : ARRAY [1:7] OF STRING [10];
   ```

2. Assume further that array DAYNAMES is preset to contain the names of the weekdays,

   i.e. DAYNAMES [1] := 'MONDAY';
   i.e. DAYNAMES [7] := 'SUNDAY';

3. Suppose we would like to perform a table look-up to see whether string DAYNAME contains a valid weekday name.

   i) **One solution could be:**

   ```pascal
   FOR DAY := 1 to 7 DO
   IF DAYNAMES [DAY] = DAYNAME THEN GOTO 100;
   WRITELN ('**ERROR: NO SUCH DAY!**');
   GOTO 999; [FINISH]
   ```

   ```pascal
   100:
   [DAY FOUND]
   ```

   ```pascal
   999:
   END.
   ```

   *Note: This is bad programming style.*

   ii) **A solution without GOTO statements would be:**

   ```pascal
   DAY := 1;
   DAYFOUND := FALSE;
   ```
WHILE (NOT DAYFOUND) AND (DAY <= 7) DO
BEGIN
  IF DAYNAMES [DAY] = DAYNAME
    THEN DAYFOUND := TRUE
    ELSE DAY := DAY + 1; ![NEXT DAY]
END; ![WHILE]
IF NOT (DAYFOUND) THEN
BEGIN
  WRITELN ("**ERROR: NO SUCH DAY!**");
END
BEGIN ![REST OF THE PROGRAM IN ELSE CLAUSE—]

END;
END.

Note: Even the GOTO 999 to the end of the program has been avoided by placing the remainder of the program into the ELSE clause of the error test.

As this technique is not always easily applicable, example (iii) may be a good compromise for complex problems.

iii) A solution using one GOTO for emergency break
DAY := 1;
DAYFOUND := FALSE;
WHILE (NOT DAYFOUND) AND (DAY <= 7) DO
BEGIN
  IF DAYNAMES [DAY] = DAYNAME
    THEN DAYFOUND := TRUE
    ELSE DAY := DAY + 1; ![NEXT DAY]
END; ![WHILE]
IF NOT (DAYFOUND) THEN
BEGIN
  WRITELN ("**ERROR: NO SUCH DAY!**");
  GOTO 999; ![GOTO END OF PROGRAM, EXIT]
END;
999: END.

Note: Except in some important cases GOTO statements should be avoided. A program that contains many GOTO statements will be difficult to understand and shows that the programmer has not yet learnt to think really in 'Pascal'.

6.6 Procedure statements
Procedure statements are used to invoke a subprogram. They consist of a procedure name possibly followed by a list of actual parameters enclosed in parantheses.
See chapter 7. Procedures and Functions.
Examples:

MOTORCONTROL (COMMAND, DIRECTION);
Y := SIN (C*X);
WRITELN;
7. ADDITIONAL DATA TYPES

7.1 User-defined Scalar Types (Enumerated types)

The basic data types in Pascal are the scalar types. These are composed of an ordered set of values. The order is simply established by enumerating the constant elements denoted by identifiers, i.e. the order is determined by the sequence in which the constants are listed.

Thus the ordinal values are discrete whole numbers: 7, 8, 9, 10, etc. The ordinal of the first constant in the list is always 0, the ordinal of the second constant is 1, etc.

Four standard scalar types are already provided: INTEGER, BOOLEAN, CHAR, and REAL.

Additional scalar types may be introduced to a program using the type definition

\[
\text{TYPE} \langle \text{type identifier} \rangle = (\langle \text{identifier} \rangle \mid, \langle \text{identifier} \rangle));
\]

Example:

One might wish in a program to deal with the months of a year

Then the following type definition may be used:

\[
\text{TYPE}
\]
\[
\text{MONTHS} = (\text{JAN}, \text{FEB}, \text{MAR}, \text{APR}, \text{MAY}, \text{JUN}, \text{JUL}, \text{AUG}, \text{SEP}, \text{OCT},
\]
\[
\text{NOV}, \text{DEC});
\]

This would associate the constant JAN with ordinal number 0, FEB with ordinal 1, DEC with ordinal 11, etc.

Now, with a variable declaration

\[
\text{VAR MONTH : MONTHS} ;
\]

one can use the above sequence:

\[
\text{MONTH} := \text{DEC};
\]
\[
\text{IF MONTH} = \text{FEB} \text{ THEN CHECKLEAPYEAR};
\]

Other examples:

\[
\text{TYPE WEEKDAYS} = (\text{MON}, \text{TUE}, \text{WED}, \text{THUR}, \text{FRI}, \text{SAT}, \text{SUN});
\]
\[
\text{TYPE COLOURS} = (\text{RED}, \text{GREEN}, \text{YELLOW}, \text{BLUE}, \text{WHITE},
\]
\[
\text{BROWN}, \text{BLACK});
\]
\[
\text{TYPE SEX} = (\text{MALE}, \text{FEMALE});
\]
\[
\text{TYPE FRUIT} = (\text{ORANGE}, \text{GUAVA}, \text{LEMON}, \text{POMELO}, \text{MANGO},
\]
\[
\text{PAPAYA}, \text{ANANAS}, \text{AVOCADO}, \text{BREADFRUIT});
\]

Note: The standard scalar type BOOLEAN is predefined in the same way:

\[
\text{TYPE BOOLEAN} = (\text{FALSE}, \text{TRUE});
\]

The sequence of FALSE and TRUE in the enumeration implies that TRUE \succ \text{FALSE} because \text{ORD (FALSE)} = 0 and \text{ORD (TRUE)} = 1.
Operators

The relational operators =, >, <, <=, >=, and > may be applied to all scalar types unless the operands are of different types.

Example:

MONTH > JAN is correct; MONTH = ORANGE is illegal.

Arithmetic operators are not applicable to enumerated types,

i.e. MONTH := NOV + 1 is not allowed.

Instead, the standard functions SUCC, PRED, ORD must be used:

e.g. MONTH := SUCC(NOV); [move to DEC]

would be the correct form.

SUCC(X) : successor of X
PRED(X) : predecessor of X
ORD(X) : ordinal number of X

\[ \text{ORD}(X) = \text{ORD}(\text{PRED}(X)) + 1 \]

Example: \[ \text{ORD}(\text{MAY}) = \text{ORD}(\text{APR}) + 1 = 3 + 1 = 4 \]

Drawback of enumerated types: Input/output of variables of such types are not possible directly but only through some mapping scheme.

7.2 Subrange types

A type may be declared as a subrange of another scalar type that is already defined. This type is called the associated scalar type.

The subrange declaration defines the lower and the upper range bounds: subrange of type real is not permitted.

Form: \[
\text{TYPE} \langle \text{type identifier} \rangle = (\langle \text{lower bound} \rangle .. \langle \text{upper bound} \rangle)
\]

Example:

\[
\begin{align*}
\text{TYPE} \text{YEARS} & := 0..9999; \quad \text{[subrange of integer]} \\
\text{DAYS} & := 1..31; \quad \{ \ldots \} \\
\text{LETTERS} & := 'A'..'Z'; \quad \text{[subrange of char]} \\
\text{NOWORK} & := \text{SAT..SUN}; \quad \text{[subrange of WEEKDAYS]} \\
\text{VAR} \text{YEAR : YEARS} & ; \\
\text{DAY : DAYS} & ; \\
\text{LETTER : LETTERS} & ;
\end{align*}
\]

A subrange type is an appropriate substitution for the associated scalar type in all definitions. Therefore, variables may be declared directly to subrange type.
Examples

VAR
NDAYS_PER_MONTH : 28..31; [subrange of integer]
DIGIT : '0'..'9'; [subrange of CHAR]
WEEKEND : SAT..SUN; [subrange of WEEKDAYS]

Subranges enable range checking at execution time.

For instance NDAYS_PER_MONTH := 32; would cause a run-time error!

Moreover, the use of subranges highly enhances the clarity and readability of a program as it elucidates the range of data associated with a variable. Good Pascal programs will therefore make use of subranges wherever possible and use the full integer type only rarely.

Example (Enumerated types and subranges)

— Get tomorrow's date from today's date —

BEGIN
IF DAY < NDAYS_PER_MONTH THEN DAY := DAY + 1 ELSE
BEGIN
DAY := 1; [update DAY and MONTH]
IF MONTH < DEC THEN MONTH := SUCC (MONTH); [next month]
ELSE
BEGIN
MONTH := JAN;
YEAR := YEAR + 1; [update YEAR]
END;
END;

An example may elucidate how the terminology of an application can be used inside and outside a program by the use of enumerated types and strings.

Assume a controls application where some commands like 'RUNUP', 'STOP', etc., are to be issued to a stepper motor.

TYPE
COURSE = (FORWARD, BACKWARD);
MODES = (RUNUP, RUNDOWN, BREAK, STOP, ACCELERATE);

VAR
MODE_COMMANDS : ARRAY [RUNUP..ACCELERATE] OF INTEGER;
MODENAME : ARRAY [RUNUP..ACCELERATE] OF STRING [12];
MODE : STRING [12];
MODEFOUND : BOOLEAN;
I : MODES;

MODENAME [RUNUP] := 'RUNUP'; ('INITIALIZATION')
MODENAME [RUNDOWN] := 'RUNDOWN';
MODENAME [BREAK] := 'BREAK';
MODENAME [STOP] := 'STOP';
MODENAME [ACCELERATE] := 'ACCELERATE';
MODEFOUND := FALSE;
FOR I := RUNUP TO ACCELERATE DO

BEGIN
  IF MODENAMES [I] = MODE THEN
    BEGIN
      MODEFOUND := TRUE; (*string match*)
      EXIT; (* exit from loop*)
    END;
  END; (*for I*)

IF NOT (MODEFOUND) THEN ERROREXIT ('WRONG MODE')
ELSE MOTORCONTROL (MODECOMMANDS [I], FORWARD);
BEGIN
    MAX := KAR[I]; \[mark new maximum]\n    IMAX := I \; \[mark its index too]\nEND ELSE
    IF KAR[I] < MIN THEN
    BEGIN
        MIN := KAR[I]; \[Mark new minimum]\n        IMIN := I \; \[and its index]\n    END;
END; \[FOR I]\n
The results should be: \[MAX = 94 \quad IMAX = 3\]
\[MIN = -5 \quad IMIN = 2\]

MULTIDIMENSIONAL ARRAYS

Since the base type TBASE of the elements may be of any type, the array elements may be structured and could be as well of type ARRAY. Such arrays are multidimensional arrays of the form
VAR
    MA : ARRAY [m...n] OF ARRAY [k...l] OF TBASE, where the depth of nesting, i.e. the number of dimensions is unlimited.

Referencing an element is possible by MA[index 1][index 2].

A better known alternative way (abbreviation) of declaring and referencing a multidimensional array is:

VAR
    MA : ARRAY [m..n, k..l] OF TBASE;
Element referencing: \[MA [index 1, index 2]\]

Examples:

Consider the system of three equations,

\[3x_1 + 4x_2 + 5x_3 = 12,\]
\[x_1 - 2x_2 - x_3 = -6,\]
\[0.5x_1 + 5x_2 + 6x_3 = 18,\]

which could be described in matrix notation by \(A \cdot x = y\), where

\[
A = \begin{pmatrix}
3 & 4 & 5 \\
1 & -2 & -1 \\
0.5 & 5 & 6
\end{pmatrix}
\]

is the matrix of the system and \(y\) is the vector of right-hand sides.

This could be conveniently expressed by:

VAR
    A : ARRAY [1..3, 1..3] OF REAL;
    Y : ARRAY [1..3] OF REAL;
and values could be input or assigned to the arrays:

\[
\begin{align*}
A[1,1] &:= 3; & A[1,2] &:= 4; & A[1,3] &:= 5; \\
\end{align*}
\]

Allocation of memory for A and Y would be in ascending order of distinct cells:

\[
\begin{array}{cccccccc}
\hline
\end{array}
\]

As follows from the notion of an array of an array, the right-most indexes are stepped first (row-wise memory allocation).

\[
\begin{align*}
\text{VAR} \\
\text{FLUX} : \text{ARRAY} [1..20, 1..20, 1..10] \text{ OF REAL} ;
\end{align*}
\]

This three-dimensional array could be referenced

\[
\text{FLUX}[3 + 1, K + 1, 8]
\]

PACKED ARRAYS

Packed arrays are ordered sets like arrays, with the simple difference that several elements are packed into one unit of memory. Although the base type is general, packed arrays are mainly used to store small integers or characters, especially in implementations where no type STRING exists.

The declaration form is

\[
\text{PACKED ARRAY} [m..n] \text{ OF TBASE}
\]

Since packing and unpacking is time-consuming, and usually referencing a packed element requires more instructions than accessing a non-packed element, packing should be used with care. It should be used only if a memory cell may contain many packed elements, if the total number of elements is large, and when element references occur infrequently.
Syntax diagram for arrays

a) Declaration

```
identifier : Packed

```

b) Element reference

```
```

STRINGS

Strings are sequences of characters enclosed by single quotes (apostrophe). The apostrophe itself can be represented by writing it twice. Strings consisting of a single character are the constants of type CHAR. Strings of several characters are represented by a PACKED ARRAY [1..n] OF CHAR (where n = No. of characters in the string) or by the type STRING if it is implemented.

Examples of strings:

'THIS IS A STRING' 'B' '**ERROR**' 'CAN'T'

8.2 The type STRING (implementation-dependent)

If the predefined type STRING exists, strings can be defined by a String declaration:

Syntax:

```
Syntax :

```

The integer specified between [ and ] denotes the maximal size of the string. If omitted, in most cases a size of 80 is assumed as default. The absolute maximum length is often 126 characters.

Since the standard definition of strings is based on PACKED ARRAY [1..n] OF CHAR, the string type is compatible with this notation. This means that single elements of a string may be referenced in the form [i], where i = element index expression.

Examples:

```
VAR
C1,C2,C3 : CHAR;
AS,BS,CS : STRING [10];
STRVECTOR : ARRAY [1..7] OF STRING [20];
```
Then the following statements were possible:

```
AS := 'CHAIN';
BS := '*ERROR:';
CS := '*';

STRVECTOR[1] := 'MONDAY';
STRVECTOR[7] := 'SUNDAY';
WRITELN(BS, 'INTEGER OUT OF RANGE*');
```

The above statement would result in the following output:

```
*ERROR: INTEGER OUT OF RANGE*
C1 := AS[1];  [would assign the character 'C' to C1]
C2 := AS[5];  [would assign the character 'N' to C2]
C3 := STRVECTOR[7,2];  [would assign the character 'U' to C3]
```

Usually the relational operators =, <>, <, <=, >=, > may be used to compare strings.
Sometimes strings may be concatenated by means of the operator + or by a standard concatenation procedure,

```
e.g. CS := 'AU' + 'TO';
```

could result in the string 'AUTO' to be assigned to CS.

Standard procedures to extract a substring from a string may also be available. The actual length of a string, i.e. the number of elements occupied in a string, may be provided by a standard procedure LENGTH,

```
e.g. LENGTH(CS) would result in 4.
```

8.3 The File types

Files are ordered (structured) sets of data on an external storage medium, e.g. a disk storage. File types are understood as sequential files, i.e. the order of grouping components on a file is defined by their sequence. This means that a certain file element can only be processed if its predecessors have already been processed. The number of components in a file is called the length of a file. A file that contains no element is an empty file.

Definition: TYPE <identifier) = FILE OF <type> ;

The declaration of a File variable f automatically introduces a file buffer variable (memory variable) f↑ of the same type as the elements; f↑ represents a window through which elements read into memory may be processed in the computer or new elements may be appended to the file in the case of Write. This window is automatically moved by certain file-handling procedures.

If the window is moved beyond the End-of-File (EOF) of file f, the standard function EOF (f) returns TRUE else FALSE.
Sequential file of n elements

<table>
<thead>
<tr>
<th>elements</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>n</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>f↑</td>
<td></td>
<td>--↑--</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>--↑--</td>
</tr>
</tbody>
</table>

empty file

| EOF |   |

STANDARD FILE-HANDLING PROCEDURES

RESET(f) resets the file window to the start for reading, i.e. f↑ = first element of file f. EOF(f) is false if file f is not empty else EOF(f) = true and f↑ is undefined.

REWRITE(f) to start rewriting a file f from the beginning, i.e. existing information would be overwritten; f will be set to empty, i.e. EOF(f) will be true.

GET(f) the file window will be moved to the next element, i.e. f↑ becomes the value of this element. If the file is empty, i.e. the file contains no element at all, then EOF(f) = true and f↑ = undefined.

PUT(f) this appends the contents of the memory buffer variable f↑ to the sequential file f. To do so, EOF(f) must be true before the call.

PAGE(f) issue form feed prior to printing on text file f

For type CHAR the following equivalences are defined:

READ(f, x)  \[x := f↑; \ \ \ \ \ \ \ \ GET(f); \ [get \ element \ x \ from \ buffer; \ move \ window]\]

WRITE(f, x) \[f↑ := x; \ \ \ \ \ \ \ \ PUT(f); \ [assign \ element \ x \ to \ memory \ buffer \ variable \ f↑; \ append \ buffer \ variable \ f↑ \ to \ file \ f]\]

RESET(f) or REWRITE(f) would 'rewind' (notion of a magnetic tape) the file f, i.e. f↑ would point to the first element in the file, if any.

8.3.1 Text files

There are files, the elements of which are characters. Such files are called text files. The standard type is defined as follows:

\[\text{TYPE TEXT} = \text{FILE OF CHAR};\]
Texts are divided into lines of characters. Separation of two consecutive lines is usually indicated by control characters (CR = carriage return and LF = line feed). The End-of-Line (EOLN) can be checked or generated by the following standard procedures:

- **EOLN(f):** a Boolean function that returns TRUE if the End-of-Line on text file f has been encountered;
- **READLN(f):** skip to the start of the next line of the text file f and read the first character of the new line;
- **WRITELN(f):** finish the current line on text file f.

**Note:** In order to test for EOLN or EOF, at least one character has to be input!

If f denotes a text file and CH a variable of type CHAR, then the following equivalences hold:

- **WRITE(f,CH)** is an abbreviation for the general form: \( f\uparrow := \text{CH}; \) PUT(f);
- **READ(f,CH)** is an abbreviation for the general form: \( \text{CH} := f\uparrow ; \) GET(f);

**8.3.2 Standard text files (INPUT and OUTPUT)**

The standard text files INPUT and OUTPUT are text files that provide communication with the environment. They usually consist of terminal keyboard or card reader for INPUT and terminal screen or line-printer for OUTPUT.

The two files are default files for the following input/output routines if no file f is specified:

- **READ(X)** = **READ(INPUT,X)**
- **READLN(X,Y)** = **READLN(INPUT,X,Y)**
- **WRITE(X)** = **WRITE(OUTPUT,X)**
- **WRITELN(A,C,'OHM')** = **WRITELN(OUTPUT,A,C,'OHM')**
- **EOF** = **EOF(INPUT)**
- **EOLN** = **EOLN(INPUT)**

**Note:** The standard procedures RESET (REWRITE) must not be used for files INPUT (OUTPUT).

**Layout of a text file (a Pascal program source file)**

```
PROGRAM MAX (INPUT);  
BEGIN  
;  
END.  
```

↑ EOLN = End-of-Line

↑ EOLN  
End-of-Line and End-of-File
8.4 Sets

Sets represent defined collections or classes of objects, where the objects in sets may be anything. For example, lakes, rivers, numbers, characters, colours may be objects of a set. Such objects are called members or elements of a set.

8.4.1 Definitions

Sets are denoted by capital letters A, B, S, T, X, etc.

Elements of sets are denoted by lower case letters: a, b, s, t, x, etc.

A set that consists of the numbers 2, 4, 6, and 13 can be written in tabular form

$$S = \{2, 4, 6, 13\}$$

If a particular set is defined by stating properties that its elements must satisfy, for instance the set T containing all even numbers, then usually x is used to represent an arbitrary element, and one can write

$$T = \{x | x \text{ is even}\}$$

which reads: T is the set of numbers such that x is even; $\mid = \text{ such that}$:

Example:

$$C = \{x | x \text{ is a lake and } x \text{ is in Finland}\}$$

If an object x is a member of a set S, i.e. S contains x as one of its elements, we write

$$x \in S \quad \text{read as: } x = \text{ element of } S$$

$$x \quad \text{belongs to } S$$

$$x \quad \text{is in } S$$

On the contrary, if x is not a member of S: $x \notin S$

FINITE AND INFINITE SETS

$$S = \{2, 5, 7, 9\} \text{ is a finite set}$$
$$T = \{2, 4, 6, 8, 10\ldots\} \text{ is an infinite set}$$

EQUALITY OF SETS

S = T? If S and T have the same members, i.e. if every element that belongs to S also belongs to T and vice versa, the sets S and T are equal.

NULL SET (EMPTY SET)

This is the set that contains no elements. It is defined for convenience. If S = set of people in the world of age > 200 years, then $S = \text{Null Set. } S = \emptyset$. 

48
SUBSETS

If every element in set \( S \) is also a member of a set \( T \), then \( S \) is called a subset of \( T \), i.e. \( S \) is a subset of \( T \) if \( x \in S \) implies \( x \in T \).

Notation: \( S \subseteq T \)
Read: \( S \) is contained in \( T \) or \( S \) is included in \( T \).

Example:

\[ X = \{1, 5, 7\} \text{ is a subset of } Y = \{1, 2, 3, 4, 5, 6, 7\} \]

Definition: Two sets \( S \) and \( T \) are equal \( S = T \)
if and only if \( S \subseteq T \) and \( T \subseteq S \).

If \( S \) is a subset of \( T \) \( \rightarrow \) \( T \supseteq S \); \( T \) is a superset of \( S \) or \( T \) contains \( S \). \( S \nsubseteq T \) or \( T \nsubseteq S \) if \( S \) is not a subset of \( T \).

Remarks: The Null Set \( \emptyset \) is considered to be a subset of every set.
If \( S \) is not a subset of \( T \), i.e. if \( S \nsubseteq T \), then there is at least one element in \( S \) which is not a member of \( T \).

PROPER SUBSET

Since every set \( S \) is a subset of itself, \( T \) is called a proper subset of \( S \) if
1) \( T \) is a subset of \( S \) and
2) if \( T \) is not equal to \( S \).

Briefly: \( T \) is a proper subset of \( S \) if \( T \subseteq S \) and \( T \neq S \).

In some literature, \( T \subseteq S \) denotes \( T \) is a subset of \( S \) and \( T \subset S \) denotes \( T \) is a proper subset of \( S \).

COMPARABILITY

Two sets \( S \) and \( T \) are comparable if \( S \subseteq T \) or \( T \subseteq S \), i.e. if one of the sets is a subset of the other set.
Sets \( S \) and \( T \) are not comparable if \( S \nsubseteq T \) and \( T \nsubseteq S \). If \( S \) is not comparable to \( T \), then there is an element in \( S \) which is not in \( T \) and also there is an element in \( T \) which is not in \( S \).

Example:

1) \( S = \{x, y\} \) and \( T = \{x, y, z\} \)
\( \rightarrow S \) is comparable to \( T \) as \( S \) is a subset of \( T \)

2) \( S = \{x, y\} \) and \( T = \{u, v, y\} \)
\( \rightarrow S \) and \( T \) are not comparable since \( x \in S \) and \( x \notin T \) and \( v \in T \) and \( v \notin S \).
If \( S \) is a subset of \( T \) and \( T \) is a subset of \( R \), then \( S \) is a subset of \( R \), i.e. \( S \subseteq T \) and \( T \subseteq R \) implies \( S \subseteq R \).
SETS OF SETS

The objects of a set may be sets themselves, e.g. the set of all subsets of S. Usually this is denoted by the notion 'family of sets' or 'class of sets' using script letters ℘, ℂ.

Example:

A family of circles, where circles are themselves sets of points.

UNIVERSAL SET

The universal set ℰ contains all sets to be considered as subsets.

Example:

The universal set for lakes consists of all the lakes on earth.

POWER SET

The family of all the subsets of any set S is called the power set of S: 2^S

Example:

N = {c,d} \rightarrow 2^N = \{[c,d], [c], [d], \phi\}
P = [3,6,9] \rightarrow 2^P = \{P, [3,6], [3,9], [6,9], [3], [6], [9], \phi\}

In the case of a set S being finite and having n elements, the power set has 2^n elements.

DISJOINT SETS

Two sets S and T are disjoint if they have no element in common.

Example:

M = \{1,2,5\}, N = \{2,4,6\} disjoint
P = \{1,3,6\}, Q = \{0,2,6\} not disjoint

VENN DIAGRAMS

To elucidate set relations in a simple way, Venn diagrams are often used, where a set is represented by a circle.

Example:

Suppose S \subset T and S \neq T:
Suppose $S \subset T$ and $T \subset S$, i.e. $S$ and $T$ being not comparable:

- **not disjoint**
  
- **disjoint**

Suppose $Z = \{r,s,t,u\}$ $\quad Y = \{t,u,v,w\}$

**Basic Set Operations**

**UNION**

The union of two sets $S$ and $T$ is a set that contains the elements which belong to $S$ or to $T$ or to both.

Union operator: $\cup$ or also $+$

$S \cup T = \{x | x \in S \text{ or } x \in T\}$

**Example:**

$R = S \cup T$ or $R = S + T$

$R$ = shaded area

**Example:**

$Z = \{z,x,y\}$, $W = \{x,v\}$

$Z \cup W = \{v,x,y,z\}$

$S \cup T = T \cup S$

$S \subset (S \cup T)$ and $T \subset (S \cup T)$

**INTERSECTION**

The intersection of sets $S$ and $T$ is the set of elements being common to $S$ and $T$. Intersection operator $\cap$ or $\cdot$

$S \cap T = \{x | x \in S, x \in T\}$
Example:

\[ R = S \cap T \quad \text{or} \quad R = S \cdot T \]

Example:

\[
\begin{align*}
Z &= \{v,x,y,z\} \quad W = \{x,v\} \\
\Rightarrow Z \cap W &= \{x,v\} \\
S \cap T &= T \cap S \\
(S \cap T) &\subset S \quad \text{and} \quad (S \cap T) \subset T
\end{align*}
\]

If \( S \) and \( T \) are disjoint, then the intersection of \( S \) and \( T \) is the null set \( \emptyset \).

Example:

\[
\begin{align*}
M &= \{7,9,11\} \quad N = \{8,10,12\} \\
\Rightarrow M \cap N &= \emptyset
\end{align*}
\]

**Differenc**

The difference of sets \( S \) and \( T \) is defined to be the set of elements that belong to \( S \) but not to \( T \). Difference operator: \( - \)

Example:

\[
R = S - T \quad S - T = \{x | x \in S, x \notin T\}
\]

Example:

\[
\begin{align*}
Z &= \{v,x,y,z\}, \quad W = \{x,v,u\} \\
Z - W &= \{y,z\}
\end{align*}
\]

**Complement**

The complement of a set \( S \) is the set whose elements do not belong to \( S \), i.e. the difference of the universal set \( U \) and \( S \).
\[
S' = U - S \quad S' = \{x | x \notin S\}
\]

\[S' = \text{shaded area}, \quad S \cup S' = U \text{ universal set} \]

\[S \cap S' = \emptyset \text{ disjoint} \]

\[U' = \emptyset, \quad \emptyset' = U \text{ the complement of the universal set is the Null Set and vice versa} \]

\[(S')' = S \]

\[S - T = S \cap T' \]

proof: \[S - T = \{x | x \in S, x \notin T\} = \{x | x \in S, x \in T'\} = S \cap T' \]

8.5 Sets in Pascal

8.5.1 The Set types

A set type in Pascal enables the encompassing of a variable number of components that must be of the same type, called the base type. Each set of values of the base type may be a possible value of a variable of a type SET. Hence, the set of the values described by the set type is equal to the power set of its base type, i.e. the set of all subsets.

The set type is a structured data type which—unlike the other structured types ARRAY and RECORD—allows processing of groups of elements.

The base type must be an ordinal scalar type or subrange type, i.e. sets of real numbers are not allowed.

Declaration of a set type:

\[
\text{TYPE } <\text{identifier}> = \text{SET OF } <\text{base type}> \]

8.5.2 Set data

Set data are built up from set elements by means of set constructors. The latter consist of expressions of the base type (enumerated set elements) enclosed in square brackets [ and ]. The empty set is denoted by specifying no elements at all, i.e. [].

Syntax of set constructors:

\[
<\text{set constructor}> ::= [<\text{element list}>]
\]

i.e. the mathematical notation [] is replaced by []

\[
<\text{element list}> ::= <\text{element}>,<\text{element}>,..<\text{element}>
\]

\[
<\text{element}>::= <\text{expression}>,<\text{expression}>,..<\text{expression}>
\]

If elements are specified in the form [m..n], all contiguous elements i between m and n (m \leq i \leq n) will be taken. This may result in the empty set if m > n.
Examples of set constructors:

- \([1,3,5]\) elements: 1,3,5
- \([7]\) element: 7
- \([K]\) element: K
- \([1..6]\) elements: 1,2,3,4,5,6
- \(['0'..'9', 'A'..'Z', +, -]\) elements: characters 0 to 9, A to Z, + and –
- \([1,2,3,4,5,6]\) elements: 1,2,3,4,5,6
- \((= [1..6])\)

8.5.3 Set operators and operands

**BASIC OPERATORS**

<table>
<thead>
<tr>
<th>Pascal notation</th>
<th>Mathematical notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Union</td>
<td>∪</td>
<td>Union of set S and T is a set that contains the elements which belong to S or T or to both.</td>
</tr>
<tr>
<td>* Intersection</td>
<td>∩</td>
<td>Intersection of sets S and T is the set of elements being common to S and T.</td>
</tr>
<tr>
<td>− Difference</td>
<td>− or ~</td>
<td>The difference of sets S and T is defined to be the set of elements that belong to S but not to T.</td>
</tr>
</tbody>
</table>

**RELATIONAL OPERATORS**

- = test equality
- < > test inequality
- <= test for inclusion (subset)
- >= test for inclusion (superset)
- IN test for set membership

- Value: true if the first operand of IN is a member of the Set denoted by the second operand

**SET OPERANDS**

- \([k,m..n]\) set \([k,m..n]\)
- \([\ ]\) empty set \(\emptyset\)
- null set

54
Examples:

Set notation may be used indirectly to shorten clumsy tests such as the following, where a character variable CH is to be tested to see if it contains an arithmetic operator character:

\[
\text{OPFOUND} := \text{FALSE}; \\
\text{IF} (\text{CH} = '+' \text{OR} (\text{CH} = '-' \text{OR} (\text{CH} = '*' \text{OR} (\text{CH} = '/'))) \text{THEN} \text{OPFOUND} := \text{TRUE};
\]

A much better approach (easier and faster) could be:

\[
\text{OPFOUND} := \text{FALSE}; \\
\text{IF CH IN ['+', '-', '*', '/']} \text{THEN} \text{OPFOUND} := \text{TRUE};
\]

Suppose we have the following type declarations:

\[
\text{TYPE NUMSET = SET OF 1..20; \quad [integers 1 to 20]} \\
\text{TYPE CHARSET = SET OF 'l'..'z'; \quad [the most important characters (implementation dependent)]} \\
\text{VAR} \\
Z,Y,X,W,V,U :\text{NUMSET}; \\
S,R,Q :\text{SET OF 1..20}; \\
ZC,YC,XC,WC :\text{CHARSET}; \\
QC,RC :\text{SET OF 'A'..'Z'}; \\
CH :\text{CHAR};
\]

[Then the following statements were possible:]

\[
Y := [1..20]; \quad \text{[put elements 1 to 20 into set Y]} \\
X := [1..09]; \quad \text{[put elements 1 to 9 into set X]} \\
W := [1,3,5,7,9]; \quad \text{[put the first five odd numbers into set W]} \\
V := [1,3,5,7,9]; \quad \text{[put the first five odd numbers into set V]} \\
U := []; \quad \text{[Assign the empty set to set U]} \\
Z := [1..5, 21..25]; \quad \text{[would result in EMPTY SET assigned to Z because the enumeration range is exceeded]} \\
YC := ['A'..'Z']; \quad \text{[put capital letters A to Z into YC]} \\
XC := ['A'..'I']; \quad \text{[characters A to I]} \\
WC := ['A', 'C', 'E', 'Q', 'I']; \quad \text{[character elements A,C,E,G,I]} \\
VC := ['A', 'C', 'E', 'G', 'I']; \quad \text{[character elements A,C,E,G,I]}
\]

If we would now assign

\[
S := X + W; \quad \text{[union of sets X and W]}
\]

then S would contain: 1,2,3,4,5,6,7,8,9.

If thereafter we did

\[
S := S - W; \quad \text{[difference of S and W to S]}
\]

then S would contain: 2,4,6,8.
If we asked for

$$R := X \cap W; \quad \text{[intersection of } X \text{ and } W]$$

then \( R \) would consist of 1, 3, 5, 7, 9.

Likewise: if one sets:

$$SC := XC \cup WC; \quad \text{[union of sets } XC \text{ and } WC]$$

then \( SC \) would contain: A, B, C, D, E, F, G, H, I.

If thereafter we did

$$SC := SC - WC; \quad \text{[difference of } SC \text{ and } WC]$$

then \( SC \) would consist of: B, D, F, H.

$$RC := XC \cap WC; \quad \text{would result in } A, C, E, G, I.$$

The statement

$$\text{IF } XC \leq YC \quad \text{would result in } \text{TRUE since } XC \text{ is a subset of } YC,$$

i.e. \( XC \subseteq YC \).

The statement

$$\text{IF } WC = VC \quad \text{would result in } \text{TRUE since the sets } WC \text{ and } VC \text{ are equal.}$$

The statement

$$UC := UC + [\text{`A', `B', `C'}] + [\text{`D' .. `H'}];$$

would assign the character elements

$$A, B, C, D, E, F, G, H \quad \text{to set } UC.$$ 

If we set

\[
\begin{align*}
UC := & \; []; \quad \text{[initialize } UC \text{ to Null Set]} \\
UC := & \; UC + [\text{`A' .. `J'}]; \quad \text{[FILL } UC \text{ TO } A..J]
\end{align*}
\]

then

$$CH := \text{`A'};$$

REPEAT

$$\begin{align*}
UC := & \; UC - [CH]; \quad \text{[difference, remove element } CH]\hphantom{; } \\
CH := & \; \text{SUCC}(CH); \quad \text{[next character]}
\end{align*}$$

UNTIL $$UC = [];$$

would empty the set \( UC \) by removing element by element.
Note: Sets are implemented as bit arrays (bit strings). The layout in memory of

\[ W = [1,3,5,7,9] \]

will be typically \[ 10101010 \]

\[ 109876543210 \]

most significant bit least significant bit

The capacity for sets is implementation-dependent. Some compilers may offer rather limited set sizes that could make the use of sets unattractive.

It is most likely that at many installations the range over the full character set is too large, i.e. that declarations SET OF CHAR are not accepted. Nevertheless, subsets containing letters, digits, and most special characters are nearly always feasible (e.g. \[ 'A'..'Z', '0'..'9', ['1'..'z'] \]).

Example: Status bit handling with Sets

\begin{verbatim}
CONST
READY = 01;  [Define status bits]
CLOSED = 02;
BUSY = 03;
RECEIVED = 04;
SENT = 5;
TIMEOUT = 06;
FAULT = 10;

TYPE
STATBITS = SET OF READY..FAULT;

VAR
STATUS : STATBITS;
OVERDUE : BOOLEAN;

BEGIN
STATUS := \[\];  [initialize to empty set]

STATUS := STATUS + [RECEIVED];
 [set status “message received”]

IF OVERDUE THEN STATUS := STATUS + [TIMEOUT, FAULT];

END
\end{verbatim}

8.6 Record types

The record type is probably the most flexible type of data declaration constructs. It is a structure consisting of a fixed number of components forming a table. The components are called fields. Although
in an array the elements must be of the same type, the fields of a record may be of a different type but they cannot be indexed directly. Access is possible only by named direct referencing. For each component a type of definition and a field identifier are necessary. Records may contain fixed and variants parts.

Syntax

\[
\begin{align*}
\langle \text{record type} \rangle & \quad ::= \ \text{RECORD} \ \langle \text{field list} \rangle \ \text{END} \\
\langle \text{field list} \rangle & \quad ::= \ \langle \text{fixed part} \rangle \ | \ \langle \text{fixed part} \rangle ; \\
& \quad \quad \langle \text{variant part} \rangle \ | \ \langle \text{variant part} \rangle \\
\langle \text{fixed part} \rangle & \quad ::= \ \langle \text{record section} \rangle \ [; \ \langle \text{record section} \rangle ] \\
\langle \text{record section} \rangle & \quad ::= \ \langle \text{field identifier} \rangle \ [; \ \langle \text{field identifier} \rangle ] ; \\
& \quad \quad \langle \text{type} \rangle \ | \ \langle \text{empty} \rangle
\end{align*}
\]

To reference a record field (component) the field identifier must be preceded by the record name followed by a point, e.g. recname.reccomponent.

Example:

A record type for complex numbers:

\[
\begin{align*}
\text{TYPE COMPLEX} & = \ \text{RECORD} \\
& \quad \quad \text{RE,IM} : \ \text{REAL} ; \\
& \quad \quad \text{END} ; \\
\text{VAR Z} : \ \text{COMPLEX} ;
\end{align*}
\]

The above record has two fixed fields (components), namely RE and IM.

\[
\begin{align*}
\text{PROGRAM COMPLX} \ (\text{OUTPUT}); \\
\quad [\text{compute and output the sum and the product of two complex numbers}] \\
\text{TYPE COMPLEX} & = \ \text{RECORD} \ \text{RE,IM} : \ \text{REAL} ; \\
& \quad \quad \text{END} \\
\text{VAR A,B} : \ \text{COMPLEX} ; \\
\text{BEGIN} \\
\quad \text{A.RE} := 6 ; \quad \text{A.IM} := 3 ; \\
\quad \text{B.RE} := 6 ; \quad \text{B.IM} := 3 ; \\
\quad \text{WRITELN} (\text{'}A\text{'}, \ \text{A.RE}, \ \text{A.IM}, \ \text{'}B\text{'}, \ \text{B.RE}, \ \text{B.IM}) ; \\
\quad [\text{get the sum of A and B}] \\
\quad \text{WRITELN} (\text{'}\text{SUM-AB} = \text{'} , \ \text{A.RE} + \ \text{B.RE} , \ \text{A.IM} + \ \text{B.IM}) ; \\
\quad [\text{get the product of A and B}] \\
\quad \text{WRITELN} (\text{'}\text{PRODUCT-AB} = \text{'} , \ \text{A.RE}\*\ \text{B.RE} - \ \text{A.IM}\*\ \text{B.IM} , \\
\quad \quad \text{A.RE}\*\ \text{B.IM} + \ \text{A.IM}\*\ \text{B.RE}) ; \\
\quad \text{WRITELN} ;
\end{align*}
\]

END.

Note: There is no concept of complex arithmetic in PASCAL. The above example of a record for complex numbers can only facilitate the grouping of the two components.

A record may also contain variant parts. Record variable structures may therefore differ although
they are of the same type. The difference may consist of a different number and different types of fields.  
A variant is composed of a list within parentheses and of declarations of its components. Each list is labelled by one or several labels, where the set of the lists is preceded by CASE specifying the data type of the labels.  
The syntax for the variant part is:

```plaintext
<variant par:> ::= CASE <tag field> <type identifier>  
                 OF <variant> [; <variant>]
<variant> ::= <case label list>: ( <field list> ) | <empty>
<case label list> ::= <case label>, <case label>
<case label> ::= <constant>
<tag field> ::= <identifier> : | <empty>
```

**Example**

*Description of aircrafts (airliners) of an airline*

<table>
<thead>
<tr>
<th>Type of aircraft</th>
<th>B747</th>
<th>DC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registration No.</td>
<td>N307PXZ</td>
<td>N212PXZ</td>
</tr>
<tr>
<td>Named</td>
<td>OKLAHOMA</td>
<td>KANSAS CITY</td>
</tr>
<tr>
<td>Internal number</td>
<td>188</td>
<td>99</td>
</tr>
<tr>
<td>Engine type</td>
<td>GEC</td>
<td>PRATT &amp; WHITNEY</td>
</tr>
<tr>
<td>Date of first flight</td>
<td>JULY 20 1973</td>
<td>MARCH 9 1963</td>
</tr>
<tr>
<td>No. of seats</td>
<td>350</td>
<td>160</td>
</tr>
<tr>
<td>Damaged?</td>
<td>UNDAMAGED</td>
<td>DAMAGED</td>
</tr>
<tr>
<td>Never damaged?</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

A record defining ‘AIRLINER’ could now be described as following:

```plaintext
TYPE NAMING = STRING [12];  
STATUS = (UNDAMAGED, DAMAGED);  
DATE = RECORD  
   MONTH: (JAN,FEB,MAR,APR,MAY,JUNE,JULY,AUG,SEPT,OCT,NOV,DEC);  
```
DAY: 1..31;
YEAR: INTEGER;
END;  [End of RECORD DATE]

AIRLINER = RECORD
  CNAME: RECORD
    CRAFTTYPE, CRAFTREG,
    CRAFTNAME: NAMING;
  END;
  NUMBER: INTEGER;
  ENGINETYPE: STRING [20];
  START: DATE;
  NSEATS: INTEGER;
  CASE OPSTATUS: STATUS OF
    DAMAGED: (DAMDATE: DATE;
    LOST: BOOLEAN);
    UNDAMAGED: (NEVERDAMAGED: BOOLEAN)
END;  [end of RECORD AIRLINER].

Note: The record AIRLINER has a variant part described in CASE. OPSTATUS is the tag field. DAMAGED and UNDAMAGED are case labels.

Assuming a variable declaration:
VAR AL: AIRLINER; referencing record components could then be done as follows:

sub-record
  CNAME  
    AL.CNAME.CRAFTTYPE  :=  'B747';
    AL.CNAME.CRAFTREG  :=  'N307PXZ';
    AL.CNAME.CRAFTNAME  :=  'OKLAHOMA';

AL.NUMBER  :=  188;
AL.ENGINETYPE  :=  'GEC';

sub-record
  DATE  
    AL.START.MONTH  :=  JULY;
    AL.START.DAY  :=  20;
    AL.START.YEAR  :=  1973;

AL.OPSTATUS  :=  UNDAMAGED;
AL.NEVERDAMAGED  :=  TRUE;

The WITH statement
Since the notation shown above can be irksome, abbreviation may be introduced using the WITH statement that opens the scope of the record so that field identifiers may be used as if they were variable identifiers.

General form:
WITH <record variable> [ , <record variable>]  
DO <statement>
Let us now repeat the assignment series shown above using the WITH statement:

```plaintext
WITH AL, CNAME, START DO
BEGIN
  CRAFTTYPE    := 'B747';
  CRAFTREG     := 'N307PXZ';
  CRAFTNAME    := 'OKLAHOMA';
  NUMBER       := 188;
  ENGINETYPE   := 'GEC';
  MONTH        := JULY;
  DAY          := 20;
  YEAR         := 1973;
  OPSTATUS     := UNDAMAGED;
  NEVERDAMAGED := TRUE;
END; [end WITH]
```

Further rules for variant parts:
- All field names must be different.
- If a label field list is void, the label should be followed by: ( ).
- There may be only one variant part in a field list and it must be specified after the fixed part.
- Nevertheless a variant part may contain other variants, i.e. they can be nested.

### 8.7 Pointer types (dynamic variables)

Pointers allow references to dynamic variables. A dynamic variable is not declared explicitly and it is not known to a program by an identifier. Unlike a static variable to which memory is allocated throughout the entire execution of the block in which it is declared, a dynamic variable is brought into existence or abandoned during program execution. Since a dynamic variable is not declared it cannot be referenced directly through an identifier. Instead, whenever a dynamic variable is generated, a pointer is set up which is the memory address of the dynamic variable. A pointer type, e.g. P, consists of a set of addresses which point to the elements of a dynamic variable of a type, e.g. V, which usually is type RECORD; P is then bound to V. A pointer can also have the value NIL, which means that it points to no element at all in V.

#### 8.7.1 Pointer declaration

```
TYPE [identifier] = ↑<type identifier>
```

**Example**

```
TYPE T = INTEGER;
TYPE P = ↑T; [declare a pointer type P that is bound to type integer]

VAR PA : P; [Declare implicitly a variable of type P.
Note: This does not allocate memory to a variable to which PA may point]
```

The pointer P is a reference to a dynamic variable of type T, and P↑ denotes that variable. Such a dynamic variable is brought into existence by a call to the standard procedure NEW. It ceases to exist when the procedure DISPOSE is invoked.
Example continued:

The call NEW(PA); would allocate memory to the dynamic variable PA↑. This variable could then be given a value, e.g.

\[
\text{PA}↑ := 13;
\]

<table>
<thead>
<tr>
<th>Memory cell</th>
<th>Cell address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k</td>
</tr>
<tr>
<td></td>
<td>k + 1</td>
</tr>
<tr>
<td>\text{PA}↑</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>k + 2 \rightarrow \text{PA}</td>
</tr>
<tr>
<td></td>
<td>k + 3</td>
</tr>
</tbody>
</table>

According to the figure, PA would have the value \(k + 2\) (a memory address) that points to the variable PA↑ containing 13.

A call DISPOSE(PA) would abandon the dynamic variable PA↑. Any further reference to PA↑ would result in a run-time error.

The main application of pointer variables is to Linked Lists. These consist of several data segments which are linked to form a chain. This chaining is achieved if at least one element of the segment contains a pointer that points to the next segment in the chain.

The ideal data structure for linked segments is the RECORD type. The record contains a component of pointer type which is in turn bound to the RECORD type.

Example

```plaintext
TYPE POINT = ↑SEGMENT;
SEGMENT = RECORD
    NUMBERS : integer;
    NEXT : POINT;
END;
VAR Y,START,PI,PNEW : POINT; [pointers]
I,N,NUM : INTEGER
```

Note:

As an exception to the general rule, the type used in a pointer declaration may be defined after it is used. This is the case for SEGMENT in the above example. The exception is a forward reference which is otherwise only possible for procedures and functions by means of a FORWARD declaration.

Using the above declaration it is possible to dynamically create and chain memory segments to store integer numbers:

By calling NEW(Y) the dynamic variable Y↑ comes into existence, where Y is bound to SEGMENT. Y↑ contains Y↑. NEXT as a component. Since Y↑.NEXT is also of the pointer type SEGMENT, the
variable Y↑.NEXT is a pointer variable too. Calling NEW(Y↑.NEXT) would then generate the pointer to the next segment. The pointer to the third element could be set by calling

\[ \text{NEW(Y↑.NEXT↑.NEXT)} \]

Using this technique the chain can be constructed. All NEXT-components (except the last one) will contain pointer data, although the number components are still untouched:

```
+-------+       +-------+       +-------+
|       |       |       |
\| Y↑    \| Y↑.NEXT  \| Y↑.NEXT↑.NEXT |
\|       \|       \|       |
```

The technique shown above is plausible but not practical because each new pointer variable has to be given another name.

A better technique for list generation is given below:

### 8.7.2 Creation of a linked listed

```
+-------+       +-------+       +-------+  
|       |       |       |      |    
\| number  \| number  \| number  \| number |
\| u       \| x       \| w       \| z      |
\| segment \| n       \| n-1     \| n-2    |
```

Assuming the above declarations, we can write:

```
START := NIL;  \[\text{initialize so that start pointer START points to nothing}\]
FOR I := 1 to N DO
BEGIN
  READ(NUM);  \[\text{read number value}\]
  NEW(Y);    \[\text{get new pointer}\]
  Y↑.NEXT := START; \[\text{update next}\]
  Y↑.NUMBERS := NUM; \[\text{store new value dynamically}\]
  START := Y;  \[\text{update start pointer}\]
END; [FOR I].
```

Adding a new segment to the list is done merely by updating the two pointers Y↑.NEXT and START.

Y is used as the running pointer which changes for each segment. With this method, the numbers read are in the list in reversed order, i.e. the list starts with the last element to which points START.

### 8.7.3 Moving through a list (Traversing a list)

```
Y := START;
WHILE Y <> NIL DO
BEGIN
  WRITE(Y↑.NUMBERS); \[\text{output value}\]
  Y := Y↑.NEXT; \[\text{move Y freely through the list}\]
END;
```

The above code would traverse the constructed integer list and output each element.
8.7.4 Insertion of a new segment into the middle of a list

Assume we want to insert a segment with a new number after the segment that contains the value which is equal to x. The first step is then to find the pointer to the segment containing x:

\[
\text{NEW(PNEW); [allocate a new dynamic variable of type SEGMENT]} \\
Y := \text{START}; \\
\text{WHILE (Y < > NIL) AND (Y\uparrow.NUMBERS < > X) DO} \\
\quad Y := Y\uparrow.NEXT; \quad \text{[move pointer, traverse]} \\
\quad PI := Y; \quad \text{[running pointer to segment containing x]}
\]

[inserting the new segment is now a simple matter of changing the pointers]

\[
P\uparrow.NEXT := PI\uparrow.NEXT; \\
PI\uparrow.NEXT := \text{PNEW};
\]

Deleting the segment following the pointer PI would simply be accomplished by modifying the pointer of the preceding segment:

\[
Y := PI\uparrow.NEXT; \quad \text{[keep pointer to segment to be deleted]} \\
PI\uparrow.NEXT := PI\uparrow.NEXT\uparrow.NEXT; \\
\text{DISPOSE(Y); [abandon dynamic storage for deleted element]}
\]

Note: The above examples for insertion and deletion are valid only for the middle of a list. If the first and the last segment is to be treated as well, more elaborate testing is necessary.

8.7.5 Other applications of pointers
1) Lists which are only accessed through their first element, i.e. lists to which an element is added to or removed from the first position are stacks or ‘last in first out’ (LIFO) lists.

2) Lists to which elements are added to one end and removed from the other are called queues or ‘first in first out’ (FIFO) lists.

Both structures may be implemented by linked lists using pointers.

3) Binary trees and general trees.

64
Note: Linked lists offer an efficient solution for the insertion and deletion of elements because only some pointers have to be changed, but not the bulk of information. The alternative solution, i.e. the representation of the elements by arrays would require adjustment of elements by shifting the array elements.

Advice to the wise following N. Wirth’s recommendation:

“‘It is finally left to the programmer to decide which of the variety of data structures should be chosen. Dynamic allocation is especially efficient and nice for insertions and deletions. If, however, such operations occur only rarely, and instead efficient access is obligatory, then the representation of the data as arrays or records is usually more appropriate.’”
9. SUBPROGRAMS = PROGRAM PARTS

9.1 Procedures and Functions

9.1.1 Introduction

Procedures and functions in Pascal are similar to subroutines and functions in other languages, and they exist for the same reasons:

1) to localize the code which performs a certain computation;
2) to state only once the code that is to be executed more than once;
3) to permit clear, explicit substitution of variable arguments in an invariable computation.

9.1.2 Procedures

A procedure is written outside the main body of the program code by which it is called, in this case in a declaration. The procedure declaration defines a program part as the program declaration defines the program. A procedure is introduced to a program by a procedure heading that corresponds to a program heading.

A procedure is entered and executed whenever the running program encounters a call to it, and it normally, but not necessarily, returns to the calling program just after the point of call.

COMMUNICATION BETWEEN A PROCEDURE AND ITS ENVIRONMENT

1) The procedure may use global identifiers.

2) Formal parameters, the types of which are specified in the heading of the procedure declaration, may be used for the exchange of information. They are represented inside the procedure by local identifiers. When the procedure is invoked, the corresponding actual parameters of the calling program are substituted.

Procedure heading:

```
PROCEDURE <identifier> ((<formal parameter section>)
   | ; (<formal parameter section>));
```

or

```
PROCEDURE <identifier>; in the case of no parameters.
```

The formal parameter section lists the name of the formal parameter followed by its type. The parameter section is followed by the declaration part which, similar to that of the main program, introduces the objects local to the procedure. Local objects belong only to the scope of the procedure and are not known outside their scope. Local variables are activated only during the execution of the procedure and will be undefined on entry to the procedure.

As for a program, the declaration part is followed by the statement part enclosed by BEGIN and END. This is called the procedure block or procedure body. A procedure may contain or invoke other procedures.

A formal parameter may be declared as a value parameter or as a variable parameter. In the latter case the parameter(s) must be preceded by VAR, and any operation inside the procedure will be performed directly upon the actual parameter, i.e. the variables of the calling program. Whenever a result is to be returned through a parameter, the latter must be a variable parameter.

In the case of a value parameter, the actual parameter will be copied to a local variable, and only this local variable will be accessible. A value parameter can therefore be only an expression and can never change the value of the actual parameter, i.e. it cannot be used to return a result.
Examples:

a) A procedure for multiplying two complex numbers

PROCEDURE MULTCOMPLEX (VAR RESRE, RESIM : REAL;
    RE1, IM1, RE2, IM2 : REAL);
BEGIN
    [RE1, IM1 = real and imaginary parts of factor 1]
    [RE2, IM2 = "" "" "" "" factor 2]
    RESRE := RE1*RE2 – IM1*IM2; [resulting real part]
    RESIM := IM1*RE2 + RE1*IM2; [resulting imaginary part]
END; [MULTCOMPLEX]

Note that the result parameters are of variable type, i.e. MULTCOMPLEX will work
directly on the substituted actual parameters, although the factors are value
parameters.

Assuming the declarations in the calling program:
VAR AR, AI, BR, BI, CR, CI, DR, DI, XR, XI, YR, YI : REAL;

possible calls would be:

MULTCOMPLEX (XR, XI, AR, AI, BR, BI);

which would multiply (AR + iAI)-(BR + iBI) and store the product p + iq to XR and XI.

The call MULTCOMPLEX (YR, YI, CR, CI, DR, DI);

would multiply (CR + iCI)-(DR + iDI) and store the result p + iq to YR and YI.

b) A procedure for swapping two integer variables

PROCEDURE EXCH (VAR ITEM1, ITEM2 : integer);
VAR K: integer;

BEGIN
    K := ITEM1; [to swap ITEM1 and ITEM2]
    ITEM1 := ITEM2; [keep a copy of ITEM1 on a local variable]
    ITEM2 := K; [ITEM1 ← ITEM2]
    [ITEM2 ← original ITEM1]
END; [END OF EXCH]

Assuming the following declarations and assignments in the calling program part:

VAR A, B, K, L : INTEGER;
A := 5; K := 7;
the call

EXCH (A, K); would result in A = 7, K = 5.
Note also that the above K and the local variable K in EXCH are distinct and not related. The local K in EXCH is used inside the procedure, whilst the outer K will be used in the calling environment.

9.1.3 Functions

Functions are invokable program parts like procedures, the only difference being that they return a single value (scalar or pointer) that can be used in an expression. The function heading corresponds to the procedure heading, but it is followed by the type of the result of the function:

FUNCTION <identifier> (<formal parameter section>)
   (;<formal parameter section>);<result type>

or

FUNCTION <identifier> : <result type> in the case of no parameters.

The statement part must contain an assignment of the result to the function identifier.

Examples:

FUNCTION PYTHAG (A,B:REAL) : REAL;
BEGIN  [compute \((A^2 + B^2)^{1/2}\) and return the result]
   PYTHAG := SQRT (SQR(A) + SQR(B));
END ; [END OF PYTHAG]
possible calls could be:
C := PYTHAG (AA,B);
D := PYTHAG ((AA + 3)/SIN(ALFA), 40.5);

TS := R[J]*SIN(Q/PYTHAG (Q,P));

Note that function PYTHAG calls other (standard) functions.

A Boolean function that returns true if its parameter, a year, is a leap year:

FUNCTION LEAPYEAR(YEAR : INTEGER) : BOOLEAN;
BEGIN
   LEAPYEAR := ((YEAR MOD 4 = 0) AND (YEAR MOD 100 <> 0))
   OR (YEAR MOD 400 = 0);
   [any year which is a multiple of 4, but years which are multiples of 100 but not of 400 are not leap years]
END; [END OF LEAPYEAR]

An alternative code for the above algorithm would be:

FUNCTION LEAPYEAR(YEAR : INTEGER) : BOOLEAN;
BEGIN
   LEAPYEAR := ((YEAR MOD 400 = 0) OR (YEAR MOD 100 <> 0))
   AND (YEAR MOD 4 = 0);
END;
9.2 Parameter types

A variable parameter should be used only if a result is to be returned or if the parameter is of type ARRAY. Otherwise, value parameters provide faster access and enhanced safety.

Besides variable and value parameters, procedures and functions may also be parameters. They have to be listed in the formal parameter section by PROCEDURE ⟨name⟩ or FUNCTION ⟨name⟩ : ⟨type⟩.

Exceptions:

Procedures and functions which are to be passed as actual parameters must have value parameters only.

File parameters must be variable parameters. Elements of packed arrays are not allowed as parameters.

Example of a FUNCTION parameter

FUNCTION ZFZ (Z: REAL; FUNCTION F: REAL) : REAL;
BEGIN
ZFZ := Z*F(Z);
END ; [END ZFZ]

Possible calls could be:

Y1 := ZFZ (Z, COS); [compute Z\cdot\cos(Z)]
Y2 := ZFZ (Z, SIN); [compute Z\cdot\sin(Z)]
Y3 := ZFZ (Z, EXP); [compute Z\cdot\exp(Z)]

FORWARD DECLARATION

It is possible to invoke procedures or functions prior to their declaration if a forward reference is given. In this case an advanced procedure heading followed by FORWARD; must precede the procedure heading of the first procedure (function) that wishes to reference the forwarded procedure (function). The proper procedure heading at the beginning of the procedure (function) is then reduced to PROCEDURE ⟨identifier⟩
(FUNCTION ⟨identifier⟩ : ⟨type⟩).

Example:

procedure CMULT (VAR Z; X,Y: real); forward;

procedure CADD (VAR T: real; R, S : real);
    |
    begin
    |
    CMULT (V,Q,K + 0.5);
    |
    end; [CADD]
procedure CMULT ; [parameter specification already given in advanced heading!]

begin

CADD (B, A, W);

end; [CMULT]

begin

CADD (CA, CC, CD));
CMULT (CB, CI, CA);

end.

9.3 Recursive calls of procedures and functions

Procedures and functions may call themselves. Recursive functions or procedures offer a solution to problems that are already formulated recursively.

Example:

By means of partial integration the following recursive formula for the calculation of
\[ \int x^n e^x \, dx \]
can be found:

\[ \int x^n e^x \, dx = x^n e^x - n \int x^{n-1} e^x \, dx. \]

This shows that the solution for any \( n \) can be found if the solution for \( n = 0 \) is known, which is trivial since
\[ \int e^x \, dx = e^x. \]

The above algorithm is a recursive algorithm as it is defined in terms of itself. The solution for any next higher rank can be found if the solution for its predecessor is known, which will be trivial if the problem is reduced to its most elementary form,

\[ e.g. \quad \int x^2 e^x \, dx = x^2 e^x - 2 \int x e^x \, dx = \]
\[ x^2 e^x - 2 [xe^x - \int e^x \, dx] = e^x (x^2 - 2x + 2) + C. \]

In some cases recursive algorithms (see above) are the only possible way to get a solution. In many other cases a problem may be formulated recursively, although other iterative solutions exist that are more efficient but which are less transparent.

A problem more apt to numeric computation than the above is the calculation of the power of a number \( Z \), where the recursive algorithm \( Z^n = Z \cdot Z^{n-1} \) is enticing.

The recursive algorithm could be expressed in Pascal by:

function POWER (Z: real; N: integer) : real;
begin

if N = 0 then POWER := 1.0
else POWER := Z * POWER (Z, N - 1);

end; [POWER, valid only for \( Z \geq 0, N \geq 0 \]
The example shows that for \( N > 0 \) the function \( \text{POWER} \) would have to call itself repeatedly until \( N = 0 \) is reached. At every recursive call the value of \( Z \) must be kept (remembered) until the step \( \text{POWER} := 1.0 \) is reached. Thereafter all the values kept must be multiplied and the temporarily used storage released, i.e. a considerable amount of administrative book keeping is required.

**Example:**

\[ Z = 4 \text{, } N = 3. \]

Computing \( 4^3 \) would result in \( \text{POWER}(4,1)*4)*4 \) and the temporary stack would be

\[
\begin{array}{ccc}
N = 3 & N = 2 & N = 1 \\
4 & 4 & 4
\end{array}
\]

**Note:** If an algorithm is recursive, the whole algorithm is to be re-executed, whilst in iterative algorithms only a part is to be reiterated.

An alternative iterative solution for \( Z^n \) could be:

```plaintext
FUNCTION \text{POWER}(Z:REAL; N:INTEGER): REAL;
VAR J : INTEGER;
    PROD : REAL;
BEGIN
    PROD := 1.0;
    FOR J := 1 TO N DO PROD := PROD*Z;
    \text{POWER} := PROD;
END; \text{[POWER, valid only for } Z \geq 0, N \geq 0]\n```

In this case no administrative overhead is required because the product is updated immediately at every reiteration.

In many cases the iterative approach is more efficient but less transparent. There are, however, cases where a recursive algorithm is the best and easiest way to solve a problem.

Similar to functions, also procedures may be used recursively either directly or indirectly.

### 9.4 Side effects

If a function alters the value of a global variable, this is called a side effect. It may lead to program faults that are difficult to find. This is in principle also true for procedures. Therefore, it is recommended to avoid altering global variables from inside subprograms.

### 9.5 Standard procedures and standard functions

Certain procedures and functions are predefined.

**Examples:**

WRITE, NEW, EOF, SIN, COS, etc.
Acknowledgements

The author is indebted to P.G. Innocenti and D.O. Williams who suggested that the course in extended form be held for a second time and before a larger audience within CERN's interdivisional PRIAM microprocessor support project.
BIBLIOGRAPHY

Baldwin, J.N.W., Microprocessors for industry (Butterworth, London, 1982).
Bennett, D.A. and D.A. Linkens (Eds.), Computer control of industrial processes (IEE Control Eng. Series 21, Peter Peregrinus, IEE London and New York, 1982).
Foulger, R.J., Programming embedded microprocessors, a high-level language solution (NCC Publ., The National Computing Centre, Ltd, Manchester, UK, 1982).
Hall, M.L., and R.D. Reimiller, OMEGASOFT Version 2, PASCAL languages handbook (Certified Software Corporation, P.O. Box 842, 5787 Brandywine Ct. Camarillo CA 93010 USA).
Hartel, P.H. Comparing PASCAL and MODULA-2 as systems programming languages in Programming languages and system design, J. Borman (Ed.) (Elsevier Science Publ., Amsterdam, 1983).
Lewis, T.G., PASCAL for the IBM PC (Addison-Wesley, Reading, MA, 1983). See also PC Magazine No. 443, Nov. 1983.
Magnenat-Thalmann, N., D. Thalmann and J. Vaucher, Le language PASCAL ISO avec PASCAL 6000 et PASCAL UCSD (Gaëtan Morin, Chicoutimi, PQ).
Persch G. et al., Early experience with the programming language ADA, in Programming languages and system design, J. Borman (Ed.) (Elsevier Science Publ., Amsterdam, 1983).

*) PDOS is a trade mark of Eyring, Research Institute Inc., Provo, Utah 84601 USA 1455 West 820 North

Tausch, L.A., The suitability of some high-level languages for the programming of embedded computer systems and the similarity of these languages, CERN LEP Controls Note No. 25 (1982).


Wirth, N., MODULA-2 (ETH, Zurich, 1980).

Wirth, N., Programming in MODULA-2 (Springer Verlag, Berlin, 1983).

Zaks, R., Introduction to PASCAL, including UCSD PASCAL (Sybex, USA, 1981).
### A.1 TABLE OF OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operand type</th>
<th>type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARITHMETICAL</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- (unary)</td>
<td>sign inversion</td>
<td>real or integer</td>
<td>as operand</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>real or integer</td>
<td>real or integer</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>real or integer</td>
<td>real or integer</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>real or integer</td>
<td>real or integer</td>
</tr>
<tr>
<td>/</td>
<td>real division</td>
<td>real or integer</td>
<td>real</td>
</tr>
<tr>
<td>DIV</td>
<td>integer division</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>MOD</td>
<td>modulus</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td><strong>LOGICAL</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>negation</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>AND</td>
<td>conjunction</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>OR</td>
<td>disjunction</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>(inclusive OR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RELATIONAL</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>compare for</td>
<td>scalar data, set,</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>equality</td>
<td>pointer or string</td>
<td></td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>inequality</td>
<td></td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>scalar data, string</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less or equal</td>
<td>scalar data, string set</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>set inclusion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>scalar data, string set</td>
<td>Boolean</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater or equal</td>
<td>scalar data, string set</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
IN membership in a set
left operand: scalar
right operand: set base type

SETS
+
set union
set type ANY
set type ANY

-
set difference
set type ANY
set type ANY

*
set intersection
set type ANY
set type ANY

:=
assignment
all types
but not FILE

A.2 OPERATOR PRECEDENCE
NOT -(unary) highest precedence
* / DIV MOD AND
+ - OR
= < > < <= >= > IN lowest precedence

A.3 RESERVED WORDS (DELMITERS)
AND DOWNTO IF OR THEN
ARRAY ELSE IN PACKED TO
BEGIN END LABEL PROCEDURE TYPE
CASE FILE MOD PROGRAM UNTIL
CONST FOR NIL RECORD VAR
DIV FUNCTION NOT REPEAT WHILE
DO GOTO OF SET WITH

A.4 STANDARD IDENTIFIERS
Constants: MAXINT, FALSE, TRUE
Types: BOOLEAN, INTEGER, REAL, CHAR, TEXT
Text files: INPUT, OUTPUT
(Program parameters)
Procedures: DISPOSE, GET, NEW, PACK, PAGE, PUT, READ,
READLN, RESET, REWRITE, UNPACK, WRITE,
WRITELN
Functions: ABS, ARCTAN, CHR, COS, EOF, EOLN, EXP, LN, ODD,
ORD, PRED, ROUND, SIN, SQR, SQRT, SUCC, TRUNC

76
A.5 TRANSFER FUNCTIONS

ROUND (X) Returns the rounded value of the real number X as integer:
X ≥ 0 : ROUND (X) = TRUNC (X + 0.5)
X < 0 : ROUND (X) = TRUNC (X − 0.5)

TRUNC (X) Returns the integer part of the real number X, as integer, e.g.:
TRUNC (4.9) = 4
TRUNC (−4.9) = −4

CHR(I) This is the character function. It returns the character that
corresponds to ordinal I in the character set (converse of ORD);
type of I: integer.

ORD(Z) Returns the ordinal number of the argument Z in the set of
enumerated values.

CHR(ORD(C)) = C and
ORD(CHR(I)) = I,

Where C = character of type CHAR

A.6 ORDINAL COUNTING FUNCTIONS

SUCC(X) Returns the successor value of X, where X is of any scalar type,
except real.

PRED(X) Returns the predecessor value of X, where X is of any scalar type
except real.

A.7 ARITHMETIC AND ELEMENTARY MATHEMATICAL FUNCTIONS

ABS (X) Absolute value (magnitude) of X
Type of X: Real or integer.

SQR (X) Square of X. Type of X: Real or integer.

Functions of type real

SIN (X) Trigonometric function sine sin(x)
COS (X) " " " cosine cos(x)
ARCTAN (X) Inverse trigonometric function arctangent atan(x)
LN (X) Natural logarithm ln(x)
EXP (X) Exponential function exp(x) = e^x
SQRT (X) Square root √x

A.8 BOOLEAN FUNCTIONS (PREDICATES)

ODD(X) returns TRUE if the integer X is odd, otherwise the result will be
FALSE.
EOLN (f) returns TRUE if during reading from file f End-of-Line is encountered, else FALSE.

EOF (f) returns TRUE if during reading from file f End-of-File is encountered, else FALSE.

A9 STANDARD FILE-HANDLING PROCEDURES

Note: \( f \) = memory buffer variable of file \( f \)

RESET (f) = rewind file \( f \) for reading
Resets the file window to start for reading, i.e. \( f \) becomes first element of file \( f \). EOF(\( f \)) is false if file \( f \) is not empty, else EOF(\( f \)) = true and \( f \) is undefined.

REWRITE (f) = rewind file \( f \) for writing
To start rewriting a file \( f \) from the beginning, i.e. existing information would be overwritten; \( f \) will be set to empty, i.e. EOF (\( f \)) will be true.

GET (f)
The file window will be moved to the next element, i.e. \( f \) becomes the value of this element. If the file is empty, i.e. the file contains no element at all, then EOF (\( f \)) = true and \( f \) is undefined.

PUT (f)
This appends the contents of the memory buffer variable \( f \) to the sequential file \( f \). To do so, EOF (\( f \)) must be true before the call.

PAGE (f)
Issue formfeed prior to printing on text file \( f \).

READ
READLN
WRITE
WRITELN

See section on input and output

A.10 PROCEDURES FOR DYNAMIC MEMORY ALLOCATION

NEW (P)
Allocates a new dynamic variable \( P \) for the new pointer \( P \).

DISPOSE (P)
Abandons memory for the existing dynamic variable \( P \); \( P \) will cease to exist.
See section on Pointer types.

A.11 SPECIAL CHARACTER SYMBOLS

( and ) to bracket expressions;
to declare and call procedures and functions with parameters

[ and ] to declare arrays
to reference array elements
to construct set data

78
{ and }
or
(* *)

.. to indicate the range in subrange types, arrays and sets
APPENDIX B

ADDITIONAL FEATURES OF
OMEGASOFT PASCAL

B.1 ADDITIONAL DATA TYPES
    LONGINTEGER, LONGHEX, HEX, STRING, BYTE, and user-defined DEVICE.

B.1.1 LONGINTEGER
    Used where an integer does not have enough range and reals are not desirable either because of speed
    or of roundoff problems. For instance, in business applications where it is desirable to carry large money
    amounts and still have accurate cent amounts, reals are not suitable. The same is true for the handling of
    large frequency values in radio-frequency technology:

    representation: 4 bytes,
    range: \((-2147483648...2147483647\)).

B.1.2 BYTE
    Defined as char, synonym for type char. To be used when data other than characters are to be
    represented as a byte. Very useful for interfacing to byte-wide I/O ports by using absolute addressing:

    representation: 1 byte.

Byte numbers are formed by prefixing the sign # in front of an unsigned integer or hex number.

Example:

    VAR B1, B2 : BYTE;
    |
    B1 := #15; B2 := #$2E;

B.1.3 HEX
    Hex numbers are formed by using the decimal digits 0 through 9 and the letters A through F preceded
    by a $ (dollar sign):

    representation: 2 bytes
    range: $0...$FFFF.

Example:

    VAR A1 : HEX;
    |
    A1 := A1 + $D031;
B.1.4 LONGHEX
A Longhex number is a hex number that is out of the HEX range, or a hex number followed by an ‘L’ with no intervening space:

representation: 4 bytes
range: $0..$FFFFFFF.

Example:
VAR LH1, LH2: LONGHEX;
    LH1 := $AFFFF;
    LH2 := $D1EL;

B.1.5 STRING: implemented according to the option presented under ‘Strings’. See heading ‘Strings’.

B.2. ADDITIONAL ARITHMETIC OPERATORS
AND a and b bitwise ANDing of a and b
OR a or b bitwise ORing of a and b
EOR a eor b bitwise EORing of a and b
    (exclusive or, difference)
NOT not a bitwise complement of a

for types: boolean, character, byte, integer, longinteger, hex, longhex

example: STAT := STAT AND 4; (* mask out 2 bits *)

** a**b raise a to the power of b
    a > = 0!
(On reals only. Automatic conversion if either operand is integer or longinteger)

<< a << b shift a left b places
    (zero filled)

>> a >> b shift a right b places
    (zero filled)

    if b < 0 → operation reversed, i.e. a << b = a >> − b

div, mod for character, byte, integer, longinteger, hex, longhex.

Except where otherwise noted, the arithmetic operators allow mixing of integer, longinteger, and real operands. In the case of a mis-match, the small types are converted automatically to the larger type.

+, − work on character, byte, integer, longinteger, hex, longhex, and real operands.
B.3. RELATIONAL EXPRESSIONS

Operators = , <, >, <=, >=, > = work on boolean, character, byte, integer, longinteger, hex, longhex, real, string, array, and record.

String comparisons are done on a character-by-character basis using the ASCII ordering. If two strings have identical characters until one of the strings runs out of characters, the shorter string is considered as the smaller.

Operator precedence

<table>
<thead>
<tr>
<th>Syntax section</th>
<th>Operators</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>factor</td>
<td>( )</td>
<td>1</td>
</tr>
<tr>
<td>term</td>
<td><em>/</em>* div mod and &lt;&lt; &gt;&gt;</td>
<td>2</td>
</tr>
<tr>
<td>simple expression</td>
<td>+ - eor or</td>
<td>3</td>
</tr>
<tr>
<td>expression</td>
<td>&lt; &lt;= = &lt;&gt; &gt;= &gt; in</td>
<td>4</td>
</tr>
</tbody>
</table>

B.4. CASE STATEMENT with

ELSE

(OTHERWISE) CLAUSE

If none of the constants in the list matches the selector expression, then none of the statements will be executed. If it is desired that an alternative statement be executed when there is no match, then an optional 'else' or 'otherwise' clause may be inserted at the end of the CASE statement.

Example:

CASE CH OF
  'A', 'B', 'C': BEG := TRUE;
  'X', 'Y', 'Z': AEND := TRUE;
  '0', '1', '2',: NUM3 := TRUE
else ILLEGALCH := TRUE;
END;

P.S. The constants may be of subrange type.

B.5. EXIT STATEMENT

Transfer statements (EXIT and GOTO) are used to modify the flow of control in a program where one of the standard conditional or repetitive statements is not suitable.

The EXIT statement is used to 'jump out of' loops. It can only be used to leave a for, repeat, or while loop.

If executed inside a loop it will jump to the point at which the loop would terminate normally.
Example:

FOUND := FALSE;
FOR Q := 1 TO N DO
BEGIN
  IF B[Q] = LOOKFOR THEN
  BEGIN
    FOUND := TRUE; (* mark found *)
    EXIT;
  END;
END; (* FOR Q *)

IF FOUND THEN s1 ELSE s2;

if EXIT (expression) is stated, nested loops may be left, where expression = nesting level:

EXIT ≡ structured GOTO

B.6. INLINE STATEMENT
Allows assembly language source code to be passed through the compiler unchanged.
To be avoided!
A better way is to call a procedure written in assembly language.

Example:

begin
  !LDA #4
  !LOOP TST $A00C
  !BMI LOOP
  !STA $A00D;
  q := true
end;

B.7. LABEL
CONST declarations
VAR

may be given in any order and repeated.

Example:

CONST N = 50;
VAR X,Y : REAL; LABEL 10,11;
CONST M = 10;
VAR IK : INTEGER;

B.8. MODULAR COMPILATION
Large Pascal programs should be broken into several modules whenever possible. The result will be easier maintainability as recompilation time will be reduced.

Recommendation: Divide larger programs into one main program and several modules containing subprograms.
Typically:

```
PROGRAM MAIN (INPUT, OUTPUT);
VAR
  ;
PROCEDURE READPRE(P1, P2 : INTEGER); EXTERNAL;
FUNCTION FUNC1 (X1, X2 : REAL) : REAL; EXTERNAL;
  ;
```

```
BEGIN
  ;
statements
  ;
END.
```

```
MODULE MOD1;
  XDEF READAR
VAR
  READPR EQU *
  ;
PROCEDURE READPRE (P1, P2 : INTEGER); ENTRY;
  
BEGIN
  ;
END; (*READPRE*)

FUNCTION FUNC1 (X1, X2 : REAL) : REAL; ENTRY;

BEGIN
  ;
END; (*FUNC1*)
MODEND.
```

Declaration Options

EXTERNAL

If a procedure is declared as external then its name is used, truncated to six characters and upshifted. In the declaration

```
procedure setports (b : byte) ; external ;
```

the following code is emitted:

```
XREF SETPOR
```

and if the procedure is called then the following code will be used:

```
LBSR SETPOR
```
This is used to access procedures defined in other Pascal modules or to access assembly language procedures.

ENTRY

If a procedure has the word ‘entry’ placed between the declaration and its block then the procedure name truncated to six characters and upshifted will be made available as an entry point. In the declaration

```pascal
procedure setports (b: byte) ; entry;
  begin
    ;
  end;
```

the following code is emitted at the start of the procedure:

```pascal
XDEF SETPOR
SETPOR EQU *
```

Using this declaration you can reference this procedure in another Pascal module by declaring it external, or you can call the procedure from assembly language by using a

```pascal
XREF SETPOR
;
LBSR SETPOR
```

B.9. IDENTIFIERS

Sequence of letters, digits and UNDERSCORE — started by letter. Maximal length: 80 characters

*Example:*

```pascal
Control__Status
POWER__SUPPLY
MagnetTemp
```

B.10. POINTERS

A pointer is essentially a HEX value.

B.11. SETS

Maximal set size: 1008 elements

B.12. ABSOLUTE ADDRESSING

Objects at absolute addresses may be used.

*Examples:*

```pascal
procedure KICK (C: integer) at $D0F0;

VAR
  ISTATUS : INTEGER AT $E0CF;
  CTRL1 : BYTE AT $E0D1;
  Settings: array [0...7] at $D200;
```
This is very important as it allows the access to fixed parts of the environment (special memory, display interfaces, etc.)

B.13. ADDITIONAL ARITHMETIC FUNCTIONS
- ARCCOS
- ARCSIN
- RANDOM
- TAN

B.14. TYPE CONVERSION FUNCTIONS
- BOOLEAN
  converts its parameter into Boolean value.
- CHAR
- CHR
- ENUM
- FLOOR
- HEX
- INTEGER
- LONGHEX
- LONGINTEGER
- ODD
- ORD
- REAL
- ROUND
- STRING
- TRUNC

B.15 STRING FUNCTIONS
LENGTH
Returns an integer in the range of 0 to 126 that represents the current dynamic length of a string parameter. This function is similar to

\[
\text{ORD}(\text{parameter}[0])
\]

The difference is that the length function will accept an expression and that it runs much slower. If you are trying to save time and space and need the length of a string parameter, you can use the above equivalent. If you just need a byte representation of the length, you should use

\[
\text{string-variable}[0]
\]

rather than

\[
\text{CHR}(\text{length(string-variable)})
\]

\[
\text{length-function} = \text{length(string-variable)}
\]
SUBSTR

Returns a string that is a subrange of the string parameter. The subrange is defined by a starting expression indicating which is the first character to include, and by a count expression indicating which is the number of characters to include. If the sum of the starting and count expressions would exceed 126 or if the starting expression is 0 then an error occurs. If the starting expression is past the end of the string then a Null string will be returned. If the sum of the starting and count expressions is past the end of the string, no error will be generated; the resulting string will just be shorter than the count expression specifies:

```
substr-function = substr (string-param, start-expression, count-expression)
```

```
string-param = (character-expression | string-expression)
start-expression = count-expression = integer-expression
```

UPSHIFT

Returns a character or string with each character that lies in the range ‘a’, … ‘z’ converted to lie in the range ‘A’ … , ‘Z’. This function will return the same type as its parameter:

```
upshift-function = upshift ((string-expression | character-expression))
```

CLINE

Returns a string with the contents of the command line (running under operating systems only). If no parameter is provided, or if the parameter has a value of zero then the entire command line will be returned. If the parameter is a positive number “n” then the “nth” command line argument will be returned. If there is no argument “n” then a Null string will be returned. Command line arguments are separated by spaces or commas.

```
cline-function = cline [('(integer-expression | character-expression)')
```

CONCAT

Returns a string that is the concatenation of its parameters. The parameters will be concatenated in the order in which they are listed. If the resulting string would exceed 126 characters the concatenation is not performed and a dynamic length error will be generated.

```
concat-function = concat(param [, param])
```

```
param = (character-expression | string-expression)
```

INDEX

Returns an integer that corresponds to the location that one string is contained within another. The result is the location (starting character number) in the first parameter where the second parameter occurs. If there is no occurrence then 0 will be returned.
index-function = index(param, param)
param = (character-expression | string-expression)

**B.16 MISCELLNEOUS FUNCTIONS**

**ADDR**

Returns the HEX absolute address of a variable. This is useful for many pointer operations.

**SIZEOF**

Returns the integer size (in bytes) of its type or variable parameter. If the type contains a variant record, the size returned will be the size of the largest variant.

**B.16.1 PREDECLARED CONSTANTS**

These are predeclared identifiers having specific constant values, they include:

- false – Boolean with an ordinal value of 0
- true – Boolean with an ordinal value of 1
- maxint – integer with a value of 32767
- minint – integer with a value of –32768
- nil – pointer or hex with a value of $0$
- maxlint – longinteger with a value of 2147483647
- minlint – longinteger with a value of –2147483648
- e – real with a value of 2.718282
- pi – real with a value of 3.141593

**B.16.2 REAL NUMBERS FOR 6809**

representation: 4 bytes = 32 bits

<table>
<thead>
<tr>
<th>M</th>
<th>E</th>
<th>exponent</th>
<th>mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_M</td>
<td>S_E</td>
<td>EEEEE</td>
<td>MMMMMMMM.~MMM</td>
</tr>
</tbody>
</table>

- 6 bits
- 24 bits
- 0

binary point (normalization)
APPENDIX C

SIMPLE PROGRAMMING EXAMPLES IN
OMEGASOFT-PASCAL FOR THE
M6809 MICROCOMPUTER
PROGRAM GRAPH; INPUT, OUTPUT, AUXOUT; (* Wirth's example *)

\( \text{Graph of } \sin(cx) \cdot e^{-x} \)

\begin{align*}
1:0 & \text{VAR} \\
11:2 & \text{I, J, K, N : INTEGER ; X, Y : REAL ;} \\
12:2 & \text{A : ARRAY[1..H2] OF CHAR ;} \\
13:2 & \text{OUTFILE : TEXT ;} \\
14:2 & \text{OUTKEY : CHAR ;} \\
15:2 & \text{IPRINT : BOOLEAN ;} \\
16:1 & \text{BEGIN} \\
17:2 & \text{WRITE('++TYPE P FOR PRINTER, ELSE CRT OUTPUT ASSUMED: ') ;} \\
18:2 & \text{READLN( OUTKEY ) ;} \\
19:2 & \text{IF IPRINT THEN OPEN(OUTFILE, AUXOUT) } \\
20:2 & \text{ELSE OPEN(OUTFILE, OUTPUT) ;} \\
21:2 & \text{FOR J:=1 TO H2 DO AIJ := ' ' ; (* BLANKS *)} \\
22:2 & \text{FOR I:=0 TO LIMIT DO} \\
23:2 & \text{BEGIN} \\
24:3 & \text{X := D*I ;} \\
25:4 & \text{Y := EXP(-X) \times SIN(C*X) ;} \\
26:4 & \text{AIH1J := ' ' ;} \\
27:4 & \text{N := ROUND(S*Y)+H1 ;} \\
28:4 & \text{AIJ := '*' ;} \\
29:4 & \text{IF N < H1 THEN K := H1 ELSE K := N ;} \\
30:4 & \text{FOR J:=1 TO K DO WRITE(OUTFILE, AIJ) ;} \\
31:2 & \text{WRITE(OUTFILE) ;} \\
32:4 & \text{WRITEN(' - GRAPH3 TERMINATED - ') ;} \\
33:4 & \text{END} \\
34:3 & \text{END ;} \\
35:2 & \text{END.} \\
\end{align*}

No Compilation Errors.
PROGRAM CHARAC( INPUT, OUTPUT, AUXOUT ) ;
CONST
SC1 = 'CHAIN' ;
SC2 = 'KETTE' ;
SC3 = 'UNION' ;
CH1 = 'Y' ;
CH2 = 'K' ;
CH3 = 'U' ;
VAR
S1, S2, S3 : STRING[10] ;
C1, C2, C3 : CHAR ;
I, J, K, L, M, N : INTEGER ;
OUTKEY : CHAR ;
IPRINT : BOOLEAN ;
OUTFILE : TEXT ;
BEGIN
WRITE( '++TYPE P FOR PRINTER, ELSE CRT OUTPUT ASSUMED: ' ) ;
READLN( OUTKEY ) ;
IPRINT := OUTKEY = 'P' ;
IF IPRINT = TRUE THEN OPEN( OUTFILE, AUXOUT ) ;
ELSE OPEN( OUTFILE, OUTPUT ) ;
S1 := SC1 ;
S2 := SC2 ;
S3 := SC3 ;
C1 := CH1 ;
C2 := CH2 ;
C3 := CH3 ;
N := LENGTH( S1 ) ;
FOR I := 1 TO N DO WRITELN( OUTFILE, S1[I] ) ;
WRITELN( OUTFILE ) ;
WRITELN( OUTFILE, S1[I3] ) ;
WRITELN( OUTFILE, S1[2] ) ;
WRITELN( OUTFILE, S1[3] ) ;
WRITELN( OUTFILE, S1[4] ) ;
WRITELN( OUTFILE, S1[5] ) ;
WRITELN( OUTFILE, S1[6] ) ;
WRITELN( OUTFILE, S1[7] ) ;
WRITELN( OUTFILE, '...' ) ;
WRITELN( OUTFILE ) ;
IF S1[I] = 'C' THEN WRITELN( OUTFILE, 'S1[I] =C' ) ;
ELSE WRITELN( OUTFILE, 'S1[I]<C' ) ;
WRITELN( OUTFILE, 'STVECTOR[23] := ' ) ;
WRITELN( OUTFILE, 'STVECTOR[33] := ' ) ;
WRITELN( OUTFILE, 'STVECTOR[43] := ' ) ;
FOR I := 1 TO 4 DO WRITELN( OUTFILE, STVECTOR[I] ) ;
FOR I := 4 DOWNTO 1 DO
64:3  BEGIN
65:4   N := LENGTH( STRVECTORIIJ );
66:4   FOR J := 1 TO N DO WRITE(OUTFILE, STRVECTOR[I,J]);
67:4   WRITELN( 'OUTFILE' );
68:3   END;  (* FOR I *)
69:3
70:2   WRITELN(' - CHARAC TERMINATED -');
71:1 END.
No Compilation Errors

C
HAIN

...
S1[1] =C
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
STRVECTOR[3,2]=E
STRVECTOR[1,4]=STRVECTOR[3,3]=D
THURSDAY
WEDNESDAY
TUESDAY
MONDAY
PROGRAM ALFR(AUXOUT); 
TYPE
VECTOR17 = ARRAY[1..7J OF INTEGER;
VECTORM15= ARRAY[-1..5]OF INTEGER;
VAR
I,J,K,L,M,N : INTEGER;
A : ARRAY[1..7J OF INTEGER;
B : ARRAY[1..7J OF INTEGER;
C : ARRAY[-1..53]OF INTEGER;
D : ARRAY[1..7J OF INTEGER;
E : VECTOR17;
PRINTER : TEXT;
PROCEDURE WIARR( VAR IARRAY : VECTOR17;
NB,NE : INTEGER );
VAR I:INTEGER;
BEGIN
FOR I := NB TO NE DO WRITE(PRINTER, IARRAY[IJ]);
END; (*WIARR*)
PROCEDURE WIARRM1( VAR IARRAY : VECTORM15;
NB,NE : INTEGER );
VAR I:INTEGER;
BEGIN
FOR I := NB TO NE DO WRITE(PRINTER, IARRAY[IJ]);
END; (*WIARRM1*)
BEGIN
OPEN( PRINTER, AUXOUT ); (* INITIALIZE PRINTER OUTPUT *)
FOR I := 1 TO 7 DO
BEGIN
AIJ := I;
BIJ := AIJ;
CII-2] := I-2;
DIJ := I-1;
WRITE(PRINTER, 'A=');
WIARR( A,1,7 );
WRITE(PRINTER, 'B='); 
WIARR( B,1,7 );
WRITE(PRINTER, 'C=');
WIARRM1( C,-1,5 );
WRITE(PRINTER, 'D=');
WIARR( D,1,7 );
WRITE(PRINTER, 'E=');
WIARR( E,1,7 );
E := A;
WRITE(PRINTER, 'E=');
WIARR( E,1,7 );
END.
No Compilation Errors

<table>
<thead>
<tr>
<th>A=</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>B=</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>C=</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D=</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>E=</td>
<td>16729</td>
<td>8224</td>
<td>8224</td>
<td>8224</td>
<td>8224</td>
<td>8224</td>
<td>8224</td>
</tr>
<tr>
<td>E=</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Program IFS(INPUT, OUTPUT); 
VAR 
FREQUENCY: REAL;
RANGE: STRING[12]; 
BEGIN 
WRITE('TYPE FREQUENCY: '); 
READLN(FREQUENCY);
RANGE := 'OUT OF RANGE'; 
IF FREQUENCY <= 400 THEN 
WRITELN('FREQUENCY TOO LOW'); 
ELSE IF FREQUENCY > 20000 THEN 
WRITELN('FREQUENCY TOO HIGH'); 
ELSE IF FREQUENCY > 4000 THEN 
RANGE := 'HIGH RANGE'; 
ELSE RANGE := 'MEDIUM RANGE'; 
WRITELN('FREQUENCY =', FREQUENCY:6:1, ' RANGE = ', RANGE);
RANGE := 'OUT OF RANGE'; 
IF FREQUENCY > 400 THEN 
IF FREQUENCY > 4000 THEN 
WRITELN('FREQUENCY TOO HIGH'); 
ELSE RANGE := 'HIGH RANGE'; 
ELSE RANGE := 'MEDIUM RANGE'; 
ELSE WRITELN('FREQUENCY TOO LOW'); 
END.
No Compilation Errors
PROGRAM MIMX1( INPUT, OUTPUT, AUXOUT );
CONST
N = 20; (* NUMBER OF ITEMS IN VECTOR A *)
NPERLINE = 5; (* NUMBER OF ITEMS PER OUTPUT LINE *)
VAR
MAX, MIN, IMAX, IMIN, I, J, U, V : INTEGER;
A: ARRAY[1..N] OF INTEGER;
OUTKEY: CHAR;
PRINT: BOOLEAN;
OUTFILE: TEXT;
PROCEDURE WRITEI(V, I, NPERLINE: INTEGER);
VAR
J : INTEGER;
BEGIN
J := I-1;
IF J >= 0 THEN
BEGIN
IF J0 THEN
IF J MOD NPERLINE = 0 THEN WRITELN(OUTFILE); (*NEWLINE*)
END;
WRITE(OUTFILE, V);
(* WRITE THE NUMBER*)
END (* WRITEI *);
BEGIN
(* -- PRESET ARRAY A -- *)
WRITE(‘**TYPE P FOR PRINTER, ELSE CRT OUTPUT ASSUMED: ’);
READLN(OUTKEY);
IF PRINT THEN OPEN (OUTFILE, AUXOUT);
ELSE OPEN (OUTFILE, OUTPUT);
WRITELN(OUTFILE, ‘IPRINT= ’, IPRINT);
WRITELN(OUTFILE, ‘-- ARRAY A[1..N] = ’);
WRITELN(OUTFILE);
FOR J := 1 TO N DO WRITEI(A[J], J, NPERLINE);
WRITELN(OUTFILE);
IMIN := 1;
IMAX := IMIN;
MIN := A[1];
MAX := MIN;
(* SET FIRST ELEMENT AS MIN AND MAX *)
FOR I := 2 TO N DO
IF A[I] > MAX THEN
MAX := A[I];
END IF
IF A[I] < MIN THEN
MIN := A[I];
END IF
WRITELN(OUTFILE, ‘RESULTS!’);
WRITELN(OUTFILE, ‘MAX = ’, MAX);
WRITELN(OUTFILE, ‘MIN = ’, MIN);
66#2    WRITELN( '- MIMX1 TERMINATED -' );
67#1 END.
No Compilation Errors

IPRINT= TRUE
-- ARRAY A[1..N] =

<table>
<thead>
<tr>
<th>35</th>
<th>68</th>
<th>94</th>
<th>7</th>
<th>88</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>-3</td>
<td>12</td>
<td>35</td>
<td>9</td>
</tr>
<tr>
<td>-6</td>
<td>3</td>
<td>0</td>
<td>-2</td>
<td>74</td>
</tr>
<tr>
<td>88</td>
<td>52</td>
<td>43</td>
<td>5</td>
<td>41</td>
</tr>
</tbody>
</table>

RESULTS:
MAX= 94
MIN= -6
PROGRAM SORTS1( INPUT, OUTPUT, AUXOUT );
CONST
N = 30 ; (* NUMBER OF ITEMS IN VECTOR A *)
nperline = 5 ; (* NUMBER OF ITEMS PER OUTPUT LINE *)
TYPE
VECTOR = ARRAY[1..N] OF INTEGER ;
VAR
I, J, K, L, JMIN, MIN, NUMB : INTEGER ;
A : ARRAY[1..N] OF INTEGER ;
OUTKEY : CHAR ;
PRINT : BOOLEAN ;
OUTFILE : TEXT ;
PROCEDURE IEXCH ( VAR WORD1,WORD2 : INTEGER ) ;
VAR K:INTEGER ;
BEGIN (* -- TO SWAP ELEMENTS WORD1 & WORD2 -- *)
K := WORD1 ;
WORD1 := WORD2 ;
WORD2 := K ;
END ; (* IEXCH *)
PROCEDURE PRINTARRAY( VAR IARRAY : VECTOR ;
n,nperline : INTEGER;
TITLESTR : STRING );
VAR I:INTEGER ;
BEGIN
(* -- CHECK IF STRING = EMPTY STRING -- *)
IF TITLESTR = '' THEN
BEGIN
WRITELN( OUTFILE, TITLESTR, ':
') ;
WRITELN( OUTFILE ) ; (* SIMPLY NEWLINE *)
END ;
FOR I := 0 TO N-1 DO
BEGIN
IF ( I MOD NPERLINE =0 ) AND ( I<>0 ) THEN WRITELN(OUTFILE);
WRITE( OUTFILE, IARRAY[I+1] ) ;
END ; (* FOR I *)
WRITELN( OUTFILE ) ; (* CLOSING NEWLINE *)
END ; (** PRINTARRAY **) -- -- -- -- -- -- -- --
BEGIN
(* -- PRESET ARRAY A -- *)
AC01I := 1000 ; AC02I := 0 ; AC03I := -12 ;
AC04I := 12 ; AC05I := -11 ; AC06I := 11 ;
AC07I := -10 ; AC08I := 10 ; AC09I := -1000 ;
AC10I := 2 ; AC11I := 3 ; AC12I := -6 ;
AC13I := 6 ; AC14I := 32000 ; AC15I := 5 ;
AC16I := 4 ; AC17I := -8 ; AC18I := 9 ;
AC19I := -9 ; AC20I := 8 ; AC21I := 7 ;
AC22I := 1 ; AC23I := -7 ; AC24I := -2 ;
AC25I := -1 ; AC26I := -4 ; AC27I := -3 ;
AC28I := -32000 ; AC29I := -5 ; AC30I := 31007 ;
WRITE( ''++TYPE P FOR PRINTER, ELSE CRT OUTPUT ASSUMED: '' );
READLN( OUTKEY ) ;
IF OUTKEY = 'P' Then
IF PRINT THEN OPEN ( OUTFILE, AUXOUT )
ELSE OPEN ( OUTFILE, OUTPUT ) ;
WRITELN( OUTFILE, 'PRINT=', ', IPRINT = ', IPRINT ) ;
PRINTARRAY( A, N, NPERLINE, 'ARRAY A BEFORE SORTING' ) ;
WRITELN(OUTFILE) ; (* NEWLINE *)
FOR L := 1 TO N DO
6613 rBEGIN
674 | MIN := A[1] ; (* TAKE 1ST ELEMENT AS MINIMUM *)
684 | JMIN := -1 ; (* MARK NO MINIMUM FOUND *)
694 | FOR J := L TO N DO
705 | rBEGIN
716 | NUMB := A[J] ;
726 | IF NUMB < MIN THEN
738 | rBEGIN
749 | MIN := NUMB ; (* SMALLER NUMBER FOUND *)
759 | JMIN := J ; (* MARK ITS INDEX *)
768 | rEND ; (* FOR J *)
778 | rEND ; (* FOR L *)
806 | IF JMIN = D THEN
816 | BEGIN
826 | (* -- EXCHANGE ELEMENTS -- *)
836 | IF JMIN < L THEN IEXCH( A[L], A[JMIN] ) ;
846 | END ;
853 PRINTARRAY( A, N, NPERLINE, 'ARRAY A AFTER SORTING' ) ;
858 WRITE( OUTFILE ) ; (* NEWLINE *)
861 WRITE( ' - SORTS1 TERMINATED - ' ) ;
881 rEND.

No Compilation Errors

IPRINT= TRUE
ARRAY A BEFORE SORTING:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>-12</td>
<td>12</td>
<td>-11</td>
</tr>
<tr>
<td>11</td>
<td>-10</td>
<td>10</td>
<td>-1000</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-6</td>
<td>6</td>
<td>32000</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>-8</td>
<td>9</td>
<td>-9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>-7</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>-4</td>
<td>-3</td>
<td>-32000</td>
<td>-5</td>
<td>31007</td>
</tr>
</tbody>
</table>

ARRAY A AFTER SORTING:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-32000</td>
<td>-1000</td>
<td>-12</td>
<td>-11</td>
<td>-10</td>
</tr>
<tr>
<td>-9</td>
<td>-8</td>
<td>-7</td>
<td>-6</td>
<td>-5</td>
</tr>
<tr>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>1000</td>
<td>31007</td>
<td>32000</td>
</tr>
</tbody>
</table>
PROGRAM ESCALAR ( INPUT, OUTPUT, AUXOUT ) ;

TYPE
MONTHS = ( JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC ) ;
YEARS = 0..9999 ;
DAYS = 1..31 ;

VAR
MONTH, IXMONTH : MONTHS ;
YEAR : YEARS ;
DAY : DAYS ;
NDAYS_PER_MONTH : 28..31 ; (* SUBRANGE *)
MONTHNAME : STRING[10] ;
OUTKEY : CHAR ;
PRINT : BOOLEAN ;
MONTHFOUND : BOOLEAN ;
OUTFILE : TEXT ;

FUNCTION LEAPYEAR ( YEAR: YEARS ) : BOOLEAN ;
BEGIN
LEAPYEAR := ((YEAR MOD 4 = 0 ) AND ( YEAR MOD 100 <> 0 ) )
OR ( YEAR MOD 400 = 0 ) ;
END ; (* LEAPYEAR *)

BEGIN
(* - INITIALIZE TABLE OF MONTH NAME STRINGS - *)
MONTHNAMES[0] := 'JANUARY' ;
MONTHNAMES[1] := 'FEBRUARY' ;
MONTHNAMES[2] := 'MARCH' ;
MONTHNAMES[3] := 'APRIL' ;
MONTHNAMES[7] := 'AUGUST' ;
MONTHNAMES[8] := 'SEPTEMBER' ;
MONTHNAMES[9] := 'OCTOBER' ;
MONTHNAMES[10] := 'NOVEMBER' ;

(* INPUT *)
WRITE( '*** TYPE P FOR PRINTER, ELSE CRT OUTPUT ASSUMED: ' ) ;
READ( OUTKEY ) ;
IF IPRINT = 'P' THEN OPEN( OUTFILE, OUTPUT ) ;
ELSE OPEN( OUTFILE, AUXOUT ) ;

(* FIND MONTH FROM TABLE *)
MONTH := JAN ;
MONTHFOUND := FALSE ;
WHILE ( NOT MONTHFOUND ) AND ( MONTH <= DEC ) DO
rBEGIN
  IF  MONTNAMES(MONTH) = MONTHNAME
  THEN MONTHFOUND := TRUE
  ELSE MONTH := SUCC(MONTH); (* NEXT MONTH *)
  END; (* WHILE *)
  IF NOT(MONTHFOUND) THEN
    rBEGIN
    WRITEFLN( **ERROR: NO SUCH MONTH!** );
    END ELSE
  END;

rBEGIN (* -- REST OF THE PROGRAM IN ELSE CLAUSE -- *)
  IF MONTH = FEB THEN
    IF LEAPYEAR(YEAR) THEN NDAYSPERMONTH := 29
    ELSE NDAYSPERMONTH := 28
  ELSE
    IF (MONTH=A PR) OR (MONTH = JUN) OR
    (MONTH=SEP) OR (MONTH = NOV) THEN
      NDAYSPERMONTH := 30
    ELSE NDAYSPERMONTH := 31;
  END;
  (* OUTPUT NUMBER OF DAYS PER MONTH AND YEAR *)
  WRITEFLN( OUTFILE, ',ORD(MONTH)=',ORD(MONTH):5,
    ',ORD(SUCC(MONTH))=',ORD(SUCC(MONTH)):5,
    ',ORD(PRED(MONTH))=',ORD(PRED(MONTH)):5 );
  WRITEFLN( OUTFILE, ',DAY=',DAY:5, ', NO. OF DAYS FOR ');
  MONTHNAME(MONTH), YEAR:6, '*,NDAYSPERMONTH:5);
  WRITEFLN( OUTFILE );
  (* UPDATE DAY, MONTH AND YEAR TO TOMORROW'S DATE *)
  IF DAY (NDAYSPERMONTH THEN DAY := DAY+1 ELSE
  rBEGIN
  DAY := 1;
  IF MONTH < DEC THEN MONTH := SUCC(MONTH); (* NEXT *)
  ELSE
  rBEGIN
  MONTH := JAN;
  YEAR := YEAR+1;
  END;
  END;

  (* OUTPUT TOMORROW'S DATE *)
  WRITEFLN( OUTFILE, 'NEXT DAY:', DAY:5, 'TH ');
  MONTHNAME(MONTH), YEAR:6 );
  WRITEFLN( OUTFILE ); (* SIMPLY NEWLINE *)
  END; (* END OF ELSE CLAUSE OF TABLE LOOK-UP *)
  WRITEFLN ( ' - ESCALAR TERMINATED - ');
116*1 END.

No Compilation Errors
ORD(MONTH) = 1 ORD(SUCC(MONTH)) = 2 ORD(PRED(MONTH)) = 0
DAY = 28 NO. OF DAYS FOR FEBRUARY 1984 29

NEXT DAY: 29TH FEBRUARY 1984
ORD(MONTH) = 1 ORD(SUCC(MONTH)) = 2 ORD(PRED(MONTH)) = 0
DAY = 28 NO. OF DAYS FOR FEBRUARY 1985 28

NEXT DAY: 1TH MARCH 1985
ORD(MONTH) = 11 ORD(SUCC(MONTH)) = 12 ORD(PRED(MONTH)) = 10
DAY = 31 NO. OF DAYS FOR DECEMBER 1984 31

NEXT DAY: 1TH JANUARY 1985

100
PROGRAM EQU2 (INPUT, OUTPUT, AUXOUT) ;  (**I**)

(* -- QUADRATIC EQUATIONS -- *)

CONST
NMAX = 50 ;  (* MAXIMAL NUMBER OF DATA PAIRS P,Q *)

VAR
I, NL : INTEGER ;
X1REAL, X1IMAG, X2REAL, X2IMAG : REAL ;
PHALF, RADICAND, ROOT : REAL ;
OUTKEY : CHAR ;
IMPRINT : BOOLEAN ;
INFILE : TEXT ;
OUTFILE : TEXT ;
ENDSTRING : STRING[10] ;
INFILESTRING : STRING[20] ;
P, Q : ARRAY[1..NMAX] OF REAL ;

FUNCTION TESTEND : BOOLEAN ;
BEGIN
TESTEND := EOF( INFILE ) OR ( ENDSTRING = ' END' ) ;
END ;  (* TESTEND *)

BEGIN
WRITE( '"**TYPE P FOR PRINTER, ELSE CRT OUTPUT ASSUMED: "');
READLN( OUTKEY ) ;
IMPRINT := OUTKEY = 'P' ;
IF IMPRINT THEN OPEN ( OUTFILE, AUXOUT ) ;
ELSE OPEN ( OUTFILE, OUTPUT ) ;
WRITE( '"**TYPE FILENAME FOR FILE INPUT, ELSE CRT INPUT: "');
READLN( INFILESTRING ) ;
IF INFILESTRING = '"' THEN OPEN( INFILE, INPUT ) ;
ELSE OPEN( INFILE, INFILESTRING, INPUT) ;
(* -- INPUT FILE FOR INPUT OF DATA PAIRS P,Q NOW OPENED -- *)

WRITELN( OUTFILE,
'** SOLUTION OF QUADRATIC EQUATIONS X**2 + PX + Q = 0 -- ');
WRITELN( OUTFILE, 'P' ) ;
WRITELN( OUTFILE, 'Q' ) ;
(* -- READ AND WRITE INPUT DATA -- *)

NL := 0 ;
REPEAT
NL := NL+1 ;  (* DATA COUNT *)
READLN( INFILE, P[NL], Q[NL], ENDSTRING ) ;
WRITELN( OUTFILE, P[NL], Q[NL] ) ;
UNTIL TESTEND OR ( NL )= NMAX ) ;
WRITELN( OUTFILE ) ;
IF ENDSTRING = '" END' THEN WRITE( OUTFILE, '" END MARKER ENCOUNTERED' ) ;
ELSE WRITE( OUTFILE, '" EOF ENCOUNTERED "' ) ;
WRITELN( OUTFILE, ' NDATA=', NL ) ;
(* -- NORMAL FORM OF EQUATION: X**2 + PX + Q = 0 -- *)
(* -- SOLUTION: X1,2 = -P/2 +/- SQRT( (P/2)**2 - Q ) -- *)
FOR I := 1 TO NL DO
Phase 2

```
683  r BEGIN
694  |  PHALF := P[i]/2.0;
704  |  RADICAND := SQRT(PHALF) - G[i];
714  |  ROOT := SQRT(ABS(RADICAND));
724  |  IF RADICAND > 0 THEN
734  |    (* --- REAL SOLUTIONS --- *)
744  |    X1REAL := -PHALF + ROOT;
754  |    X2REAL := -PHALF - ROOT;
764  |    X1IMAG := 0;
774  |    X2IMAG := 0;
784  |  END ELSE
794  |    (* --- COMPLEX SOLUTIONS --- *)
804  |    X1REAL := -PHALF;
814  |    X2REAL := -PHALF;
824  |    X1IMAG := ROOT;
834  |    X2IMAG := -ROOT;
844  |  END; (* COMPLEX SOLUTIONS *)
854  |  WRITELN('OUTFILE, X1REAL, X1IMAG, X2REAL, X2IMAG);
864  |  END; (* FOR I *)
874  |  CLOSE('INFILE');
884  |  WRITELN('OUTFILE ');
894  |  WRITELN('OUTFILE.', ' --- ALL EQUATIONS SOLVED --- ');
904  |  WRITELN(' EQU2 TERMINATED - ');
944  |  END.
```

No Compilation Errors

--- SOLUTION OF QUADRATIC EQUATIONS X**2 + PX + Q = 0 ---

```
<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.000000E+00</td>
<td>-3.000000E+00</td>
</tr>
<tr>
<td>-2.500000E-01</td>
<td>-2.000000E+00</td>
</tr>
<tr>
<td>-2.500000E-01</td>
<td>-1.000000E+00</td>
</tr>
</tbody>
</table>
```

```
- END MARKER ENCOUNTERED NDATA= 3
```

--- ALL EQUATIONS SOLVED ---

--- SOLUTION OF QUADRATIC EQUATIONS X**2 + PX + Q = 0 ---

```
P          Q
```

```
| -1.000000E+00 | -3.000000E+00 |
```

```
- EOF ENCOUNTERED - NDATA=
```

```
<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.302776E+00</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>1.544728E+00</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>1.327033E+00</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>-1.302776E+00</td>
<td>0.000000E+00</td>
</tr>
</tbody>
</table>
```

--- ALL EQUATIONS SOLVED ---

--- SOLUTION OF QUADRATIC EQUATIONS X**2 + PX + Q = 0 ---

```
P          Q
```

```
| -1.000000E+00 | -3.000000E+00 |
| -5.000000E-01 | -2.000000E+00 |
| -2.500000E-01 | -1.000000E+00 |
| 0.000000E+00 | 0.000000E+00 |
```

102
\[ \begin{array}{rrrr}
2.500000E-01 & 1.000000E+00 \\
5.000000E-01 & 2.000000E+00 \\
1.000000E+00 & 3.000000E+00 \\
2.000000E+00 & 0.000000E+00 \\
3.000000E+00 & 2.250000E+00 \\
-1.000000E+00 & 0.000000E+00 \\
-5.000000E-01 & 5.000000E-01 \\
\end{array} \]

- END MARKER ENCLOSED NDATA= 11

<table>
<thead>
<tr>
<th>X1REAL</th>
<th>X1IMAG</th>
<th>X2REAL</th>
<th>X2IMAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.302776E+00</td>
<td>0.000000E+00</td>
<td>-1.302776E+00</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>1.686142E+00</td>
<td>0.000000E+00</td>
<td>-1.186141E+00</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>1.132783E+00</td>
<td>0.000000E+00</td>
<td>-8.827826E-01</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>0.000000E+00</td>
<td>0.000000E+00</td>
<td>0.000000E+00</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>-2.500000E-01</td>
<td>1.391941E+00</td>
<td>-2.500000E-01</td>
<td>-1.391941E+00</td>
</tr>
<tr>
<td>-5.000000E-01</td>
<td>1.658313E+00</td>
<td>-5.000000E-01</td>
<td>-1.658313E+00</td>
</tr>
<tr>
<td>3.576280E-07</td>
<td>0.000000E+00</td>
<td>-2.000000E+00</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>-1.500000E+00</td>
<td>0.000000E+00</td>
<td>-1.500000E+00</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>1.000000E+00</td>
<td>0.000000E+00</td>
<td>-1.788140E-07</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>2.500000E-01</td>
<td>6.614380E-01</td>
<td>2.500000E-01</td>
<td>-6.614380E-01</td>
</tr>
</tbody>
</table>

-- ALL EQUATIONS SOLVED --
PROGRAM SETS( AUXOUT ) ;

TYPE NUMSET = SET OF 1..20 ;

VAR
  Z, Y, X, W, V, U, T : SET OF 1..20 ;
  S, R, Q : NUMSET ;
  I, J, K : INTEGER ;
  PRINTER : TEXT ;

PROCEDURE NEWLINE ;
BEGIN
  WRITELN( PRINTER ) ;
END ; (* NEWLINE *)

PROCEDURE OUTNUMSET( NAMES: STRING ;
                      S: NUMSET ; N: INTEGER ) ;
VAR
  I: INTEGER ;
  MARKIN: BOOLEAN ;
BEGIN
  WRITELN( PRINTER, 'THE SET ', NAMES, ' CONTAINS:' ) ;
  MARKIN := FALSE ;
  FOR I := 1 TO N DO
    IF I IN S THEN
      MARKIN := TRUE ;
      WRITE( PRINTER, I:4 ) ;
  END ; (* FOR I *)
  IF MARKIN THEN NEWLINE
  ELSE WRITELN( PRINTER, '*NOTHING* (EMPTY SET) ' ) ;
  NEWLINE ;
END ; (* OUTNUMSET *)

PROCEDURE TEXTLINE( TXSTRING : STRING ) ;
BEGIN
  WRITELN( PRINTER, TXSTRING ) ;
END ; (* TEXTLINE *)

PROCEDURE CHECKMEMBER( NAMES: STRING ;
                        S: NUMSET ; ELEMENT: INTEGER ) ;
BEGIN
  WRITE( PRINTER, ELEMENT:4, ' IS ' ) ;
  IF ELEMENT IN S THEN WRITE( PRINTER, 'A' )
  ELSE WRITE( PRINTER, 'NOT A' ) ;
  WRITELN( PRINTER, ' MEMBER OF ', NAMES ) ;
  NEWLINE ;
END ; (* END OF CHECKMEMBER *)

PROCEDURE CHECKSETEQ( NAMES: STRING ; S : SET OF 1..20 ;
                       NAME: STRING ; T : SET OF 1..20 ) ;
BEGIN
  WRITE( PRINTER, 'SETS ', NAMES, ' AND ', NAME, ' ARE ' ) ;
  IF S = T THEN WRITELN( PRINTER, 'EQUAL' )
  ELSE WRITELN( PRINTER, 'NOT EQUAL' ) ;
  NEWLINE ;
END ; (* END OF CHECKSETEQ *)
PROCEDURE CHECKSUBSET ( NAMES: STRING; S, T: NUMSET;
                     NAME: STRING; U: NUMSET);
BEGIN
WRITE( PRINTER, 'SET ', NAMES, ' IS ');
IF S = T THEN WRITE( PRINTER, 'A' )
ELSE WRITE( PRINTER, 'NOT A' )
END;
(* END OF CHECKSUBSET *)
(* -- START OF MAIN PROGRAM -- *)
OPEN ( PRINTER, AUXOUT );
(* INITIALIZE PRINTER OUTPUT *)
Y := [1..20];
X := [1..59];
W := [1,3,5,7,9];
V := [1,3,5,7,9];
U := [];
Z := [1..5,11..15,21..25];
T := [2..6,12..16,19,20];
Q := [20,21,3,6];
OUTNUMSET( 'W', W, 20 );
OUTNUMSET( 'V', V, 20 );
CHECKSET0( 'W', W, 'V', V );
OUTNUMSET( 'X', X, 20 );
OUTNUMSET( 'Y', Y, 20 );
CHECKSUBSET( 'X', X, 'Y', Y );
CHECKSUBSET( 'Y', Y, 'X', X );
OUTNUMSET( 'U', U, 20 );
S := X + W;
OUTNUMSET( 'S', S, 20 );
S := S - W;
OUTNUMSET( 'S', S, 20 );
R := X * W;
OUTNUMSET( 'R', R, 20 );
CHECKMEMBER( 'R', R, 5 );
CHECKMEMBER( 'R', R, 4 );
OUTNUMSET( 'Z', Z, 20 );
OUTNUMSET( 'T', T, 20 );
OUTNUMSET( 'Q', Q, 20 );
TEXTLINE( '--- FILL SET U TO ELEMENTS 2,4,6,8..20 ---');
FOR I := 1 TO 10 DO  U := U + [2*I] ; (* UNION OF EVEN *)
135:2
136:2 OUTNUMSET( 'U', U, 20 ) ;
137:2 TEXTLINE(' -- TAKE ELEMENTS 2,4,6,8,10..20 OUT OF  U --- ');
138:2
139:2 I := 2 ;
140:2 REPEAT
141:3 U := U - [I] ;  (* DIFFERENCE UNTIL EMPTY *)
142:3 I := I+2 ;  (* STEP *)
143:2 UNTIL U = [] ;
144:2 OUTNUMSET( 'U', U, 20 ) ;
145:2
146:1 END.

No Compilation Errors

THE SET W CONTAINS:
1 3 5 7 9

THE SET V CONTAINS:
1 3 5 7 9

SETS W AND V ARE EQUAL

THE SET X CONTAINS:
1 2 3 4 5 6 7 8 9

THE SET Y CONTAINS:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

SET X IS A SUBSET OF Y

SET Y IS NOT A SUBSET OF X

THE SET U CONTAINS:
*NOTHING* (EMPTY SET)

-- S := X + W ( UNION OF X AND W ) --

THE SET S CONTAINS:
1 2 3 4 5 6 7 8 9

-- S := S - W ( DIFFERENCE OF S AND W ) --

THE SET S CONTAINS:
2 4 6 8

-- R := X * W ( INTERSECTION OF X AND W ) --

THE SET R CONTAINS:
1 3 5 7 9

5 IS A MEMBER OF R

4 IS NOT A MEMBER OF R

THE SET Z CONTAINS:
*NOTHING* (EMPTY SET)

THE SET T CONTAINS:
2 3 4 5 6 12 13 14 15 16 19 20

THE SET Q CONTAINS:
20

-- FILL SET U TO ELEMENTS 2,4,6,8..20 --

THE SET U CONTAINS:
2 4 6 8 10 12 14 16 18 20
-- TAKE ELEMENTS 2, 4, 6, 8, 10, .. 20 OUT OF U --
THE SET U CONTAINS:
*NOTHING* (EMPTY SET)
PROGRAM CSETS( AUXOUT ); (**R**) 

TYPE CHARSET = SET OF '!',..,'z';
NUMSET = SET OF 1..10;

VAR
  Y,X,W,V,U,T : CHARSET;
  S,R,O,P : CHARSET;
  Z : SET OF 'A'..'z';
  CH : CHAR;
  OPFOUND : BOOLEAN;
  I,J,K : INTEGER;
  PRINTER : TEXT;

PROCEDURE NEWLINE;
  BEGIN
  WRITELN( PRINTER );
  END (* NEWLINE *)

(*---------------------------------------------------------------*)

PROCEDURE OUTCHARSET( NAMES: STRING ;
                      S: CHARSET );

CONST
  NPERLINE = 20;

VAR
  I,COUNT:INTEGER;
  MARKIN: BOOLEAN;

BEGIN
  WRITELN( PRINTER, 'THE CHAR.SET ', NAMES, ' CONTAINS:' );
  MARKIN := FALSE;
  COUNT := 0;
  FOR I := 33 TO 126 DO IF CHR(I) IN S THEN
    BEGIN
      MARKIN := TRUE;
      IF COUNT > 0 THEN NEWLINE;
      WRITE( PRINTER, ' ', CHR(I), ' ' ) ;
      END (* FOR I *);
  IF MARKIN THEN NEWLINE
  ELSE WRITELN( PRINTER, '*NOTHING* (EMPTY SET)' );

NEWLINE ;
END (* OUTCHARSET *)

PROCEDURE OUTLINE ( TXSTR : STRING );
BEGIN
  WRITELN( PRINTER, TXSTR );
END (* OUTLINE *)

PROCEDURE CHECKMEMBER( NAMES: STRING ;
                        S: CHARSET ; ELEMENT: CHAR );
BEGIN
  WRITE( PRINTER, ' ', ELEMENT, ' IS ' ) ;
  IF ELEMENT IN S THEN WRITE( PRINTER, '"A"' ) ;
  ELSE WRITE( PRINTER, '"NOT A"' ) ;
  WRITELN( PRINTER, '"MEMBER OF ', NAMES );
  NEWLINE ;
END (* END OF CHECKMEMBER *)

PROCEDURE CHECKSETEQ( NAMES: STRING ; S: CHARSET ;
                        NAMET: STRING ; T: CHARSET );
BEGIN
WRITE( PRINTER, 'SETS ' ); NAMES, ' AND ', NAMEIT, ' ARE ' 
IF S = I THEN WRITELN( PRINTER, 'EQUAL' ) 
ELSE WRITELN( PRINTER, 'NOT EQUAL' );
NEWWLINE ; 
END ; (* END OF CHECKSETEQ *)

PROCEDURE CHECKSUBSET ( NAMES: STRING ; S: CHARSET ; 
NAMEIT: STRING ; T: CHARSET );
BEGIN
WRITE( PRINTER, 'SET ' ); NAMES, ' IS ' ;
IF S (= I THEN WRITE( PRINTER, 'A' ) 
ELSE WRITE( PRINTER, 'NOT A' ) ;
WRITELN( PRINTER, ' SUBSET OF ' );
NAMEIT ;
NEWWLINE ;
END ; (* END OF CHECKSUBSET *)

(* -- START OF MAIN PROGRAM -- *)

OPEN ( PRINTER, AUXOUT ) ; (* INITIALIZE PRINTER OUTPUT *)

Y := [ 'A'..'Z'] ; (* PUT ELEMENTS INTO SETS *)
X := [ 'A'..'I' ] ;
W := [ 'A'..'C'..'E'..'G'..'I' ] ;
V := [ 'A'..'C'..'E'..'G'..'I' ] ;
U := [ ] ; (* EMPTY SET *)
Z := [ 'A'..'E'..'K'..'O'..'O'..'Z' ] ;
T := [ 'B'..'F'..'L'..'P'..'Y'..'Z' ] ;
Q := [ 'Z'..'Z' ] ;

(* --- NOTE THAT THE FIRST CHECK FOR AN ARITHMETIC 
OPERATOR CHARACTER USING SET NOTATION IS 
MUCH EASIER AND FASTER THAN THE FOLLOWING 
CONVENTIONAL CHECK --- *)

OPFOUND := FALSE ;
IF CH IN ['+', '-', '*', '/'] THEN OPFOUND := TRUE ;

OPFOUND := FALSE ;
IF ( CH='+' ) OR ( CH='-' ) OR ( CH='*' ) OR ( CH='/' ) 
THEN OPFOUND := TRUE ;

OUTCHARSET( 'S', S ) ;
OUTCHARSET( 'W', W ) ;
OUTCHARSET( 'V', V ) ;
CHECKSETEQ( 'W', W, 'V', V ) ;
OUTCHARSET( 'X', X ) ;
OUTCHARSET( 'Y', Y ) ;
CHECKSUBSET( 'X', X, 'Y', Y ) ;
CHECKSUBSET( 'Y', Y, 'X', X ) ;
OUTCHARSET( 'U', U ) ;
S := X + W ; (* UNION OF X AND W *)
OUTLINE( '--- S := X + W ( UNION OF X AND W ) ---' );

(* END OF MAIN PROGRAM *)
OUTCHARSET( 'S', S );
S := S - W ;  (* DIFFERENCE OF X AND W *)
OUTLINE( '"-- S := S - W ( DIFFERENCE OF S AND W ) --"');
OUTCHARSET( 'S', S );
R := X * W ;  (* INTERSECTION OF X AND W *)
OUTLINE( '"-- R := X * W ( INTERSECTION OF X AND W ) --"');
OUTCHARSET( 'R', R );
CHECKMEMBER( 'R', R, 'B' );
CHECKMEMBER( 'R', R, 'C' );
OUTCHARSET( 'Z', Z );
OUTCHARSET( 'T', T );
OUTCHARSET( 'Q', Q );
(* OUTCHARSET ( 'P', P ) .. P NOT USED, --> ERROR 150 *)
OUTLINE( '"-- FILL SET U TO SOME ELEMENTS --"');
U := U + ['A', 'B', 'C'] + ['D'..'H'] ;
OUTCHARSET( 'U', U );
OUTLINE( '"-- TAKE ELEMENTS OUT OF SET U --"');
I := ORD('A');  (* ORDINAL OF CHARACTER 'A' *)
REPEAT
  U := U - [CHR(I)];  (* DIFFERENCE UN TIL EMPTY *)
  I := I+1;  (* STEP *)
UNTIL  U = [] ;
OUTCHARSET( 'U', U );
OUTLINE( '"-- FILL SET U TO SOME ELEMENTS --"');
U := U + ['A'..'J'];
OUTCHARSET( 'U', U );
OUTLINE( '"-- TAKE ELEMENTS OUT OF SET U --"');
CH := 'A';
REPEAT
  U := U - [CH];
  CH := SUCC(CH);  (* NEXT CHARACTER *)
UNTIL  U = [] ;
OUTCHARSET( 'U', U );
REPE ET

THE CHAR. SET S CONTAINS:
*NOTHING* (EMPTY SET)

THE CHAR. SET W CONTAINS:
'A' 'C' 'E' 'G' 'I'

THE CHAR. SET V CONTAINS:
'A' 'C' 'E' 'G' 'I'

SETS W AND V ARE EQUAL
THE CHARSET X CONTAINS:
'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I'

THE CHARSET Y CONTAINS:
'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J' 'K' 'L' 'M' 'N' 'O' 'P' 'Q' 'R' 'S' 'T'
'U' 'V' 'W' 'X' 'Y' 'Z'

SET X IS A SUBSET OF Y

SET Y IS NOT A SUBSET OF X

THE CHARSET U CONTAINS:
*NOTHING* (EMPTY SET)

-- S := X + W ( UNION OF X AND W ) --

THE CHARSET S CONTAINS:
'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I'

-- S := S - W ( DIFFERENCE OF S AND W ) --

THE CHARSET S CONTAINS:
'B' 'D' 'F' 'H'

-- R := X * W ( INTERSECTION OF X AND W ) --

THE CHARSET R CONTAINS:
'A' 'C' 'E' 'G' 'I'

'B' IS NOT A MEMBER OF R

'C' IS A MEMBER OF R

THE CHARSET Z CONTAINS:
'0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'A' 'B' 'C' 'D' 'E' 'F' 'K' 'L' 'M' 'N' 'O'

THE CHARSET T CONTAINS:
'B' 'C' 'D' 'E' 'F' 'L' 'M' 'N' 'O' 'P' 'Y' 'Z'

THE CHARSET Q CONTAINS:
'*' 'Z'

-- FILL SET U TO SOME ELEMENTS --

THE CHARSET U CONTAINS:
'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H'

-- TAKE ELEMENTS OUT OF SET U --

THE CHARSET U CONTAINS:
*NOTHING* (EMPTY SET)

-- FILL SET U TO SOME ELEMENTS --

THE CHARSET U CONTAINS:
'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J'

-- TAKE ELEMENTS OUT OF SET U --

THE CHARSET U CONTAINS:
*NOTHING* (EMPTY SET)
program primes(input, output, auxout);
(* 1. Put all the numbers between 2 and N into the sieve. *)
(* 2. Select and remove the smallest number remaining in the sieve. *)
(* 3. Include this number in the primes. *)
(* 4. Step through the sieve, removing all multiples of this number. *)
(* 5. If the sieve is not empty, repeat steps 2 to 5. *)

const
nmax = 1000; (* Machine limitation to 1007 *)
nperline = 5; (* Number of items per output line *)

var sieve, primes : set of 2..nmax;
next, j, c, nprimes : integer;
m, n, new, np : integer;
i, k, l : integer;
icollect : boolean;
collectkey : char;
outkey : char;
iprint : boolean;
outfile : text;

procedure writeln(v, i, nperline : integer);
var j : integer;
begin
  j := i - 1;
  if j = 0 then
    if j mod nperline = 0 then writeln(outfile); (*newline*)
  write(outfile, v);
  writeln(outfile); (*write the number*)
end (*writei*)

begin
  repeat
    write('++type range (2..2000) ');
    readln(m);
    n := m div 2;
    until (n>=2) and (n<=2000);
    write('++type p for printer, else crt output assumed: ');
    readln(outkey);
    write('++type c to collect primes: ');
    readln(collectkey);
    icollect := collectkey = 'c';
    if icollect then
      begin
        if iprint then open(outfile, auxout)
        else open(outfile, output);
        writeln(outfile, 'iprint= ', iprint);
        writeln(outfile);
        writeln(outfile, '--- prime numbers between 2 and ',
        2*n, '---');
        writeln(outfile);
      end;
    (* -- initialize prime number search -- *)
    sieve := [2..n];
    primes := [ ]; (* initialize to empty set *)
    next := 2;
    nprimes := 0;
    repeat
      (* find next prime number *)
      while not(next in sieve) do next := succ(next);
      writeln(next);
    until
end.
IF ICOLLECT THEN
PRIMES := PRIMES + [NEXT]; (* SET UNION, ADD ELEMENT NEXT *)
NPRIMES := NPRIMES + 1;
C := 2*NEXT - 1; (* C = NEW PRIME NUMBER *)
WRITEI( C, NPRIMES, NPRIMES, NPERLINE );
J := NEXT;
WHILE J <= N DO (* ELIMINATE *)
BEGIN
SIEVE := SIEVE - J; (* SET DIFFERENCE, ELIMINATE ELEMENT J *)
J := J + C;
END;
UNTIL SIEVE = []; (* UNTIL THE SIEVE BECOMES EMPTY *)
BEGIN
WRITELN( 'OUTFILE' ); (* NEWLINE *)
WRITELN( 'N=', N, ', NEXT=', NEXT, ', NPRIMES=', NPRIMES );
WRITELN( 'OUTFILE' ); (* NEWLINE *)
(* -- WRITE COLLECTION IN SET PRIMES -- *)
IF ICOLLECT THEN
BEGIN
WRITELN( 'OUTFILE', '; (-- NUMBERS IN SET PRIMES --' );
WRITELN( 'OUTFILE' );
NP := 0;
FOR I := 2 TO N DO
BEGIN
IF I IN PRIMES THEN
BEGIN
NP := NP + 1;
WRITEI( I, NP, NPERLINE );
END; (* IF I IN PRIMES *)
END; (* IF ICOLLECT *)
WRITELN( '-- PRIMES TERMINATED --' );
IF IPRINT THEN
BEGIN
WRITELN( 'OUTFILE' );
WRITELN( 'OUTFILE', '; -- PRIMES FINISHED --' );
END;
END;
END.
No Compilation Errors

IPRINT := TRUE

-- PRIME NUMBERS BETWEEN 2 AND 2000 --

<table>
<thead>
<tr>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
<td>31</td>
</tr>
<tr>
<td>37</td>
<td>41</td>
<td>43</td>
<td>47</td>
<td>53</td>
</tr>
<tr>
<td>59</td>
<td>61</td>
<td>67</td>
<td>71</td>
<td>73</td>
</tr>
<tr>
<td>79</td>
<td>83</td>
<td>89</td>
<td>97</td>
<td>101</td>
</tr>
<tr>
<td>103</td>
<td>107</td>
<td>109</td>
<td>113</td>
<td>127</td>
</tr>
<tr>
<td>131</td>
<td>137</td>
<td>139</td>
<td>149</td>
<td>151</td>
</tr>
<tr>
<td>157</td>
<td>163</td>
<td>167</td>
<td>173</td>
<td>179</td>
</tr>
<tr>
<td>181</td>
<td>191</td>
<td>193</td>
<td>197</td>
<td>199</td>
</tr>
<tr>
<td>211</td>
<td>223</td>
<td>227</td>
<td>229</td>
<td>233</td>
</tr>
<tr>
<td>239</td>
<td>241</td>
<td>251</td>
<td>257</td>
<td>263</td>
</tr>
<tr>
<td>269</td>
<td>271</td>
<td>277</td>
<td>281</td>
<td>283</td>
</tr>
<tr>
<td>293</td>
<td>307</td>
<td>311</td>
<td>313</td>
<td>317</td>
</tr>
<tr>
<td>331</td>
<td>337</td>
<td>347</td>
<td>349</td>
<td>353</td>
</tr>
<tr>
<td>359</td>
<td>367</td>
<td>373</td>
<td>379</td>
<td>383</td>
</tr>
<tr>
<td>389</td>
<td>397</td>
<td>401</td>
<td>409</td>
<td>419</td>
</tr>
<tr>
<td>421</td>
<td>431</td>
<td>433</td>
<td>439</td>
<td>443</td>
</tr>
<tr>
<td>449</td>
<td>457</td>
<td>461</td>
<td>463</td>
<td>467</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>479</td>
<td>487</td>
<td>491</td>
<td>499</td>
<td>503</td>
</tr>
<tr>
<td>509</td>
<td>521</td>
<td>523</td>
<td>541</td>
<td>547</td>
</tr>
<tr>
<td>557</td>
<td>563</td>
<td>569</td>
<td>571</td>
<td>577</td>
</tr>
<tr>
<td>587</td>
<td>593</td>
<td>599</td>
<td>601</td>
<td>607</td>
</tr>
<tr>
<td>613</td>
<td>617</td>
<td>619</td>
<td>631</td>
<td>641</td>
</tr>
<tr>
<td>643</td>
<td>647</td>
<td>653</td>
<td>659</td>
<td>661</td>
</tr>
<tr>
<td>673</td>
<td>677</td>
<td>683</td>
<td>691</td>
<td>701</td>
</tr>
<tr>
<td>709</td>
<td>719</td>
<td>727</td>
<td>733</td>
<td>739</td>
</tr>
<tr>
<td>743</td>
<td>751</td>
<td>757</td>
<td>761</td>
<td>769</td>
</tr>
<tr>
<td>773</td>
<td>787</td>
<td>797</td>
<td>809</td>
<td>811</td>
</tr>
<tr>
<td>821</td>
<td>823</td>
<td>827</td>
<td>829</td>
<td>839</td>
</tr>
<tr>
<td>853</td>
<td>857</td>
<td>859</td>
<td>863</td>
<td>877</td>
</tr>
<tr>
<td>881</td>
<td>883</td>
<td>887</td>
<td>907</td>
<td>911</td>
</tr>
<tr>
<td>919</td>
<td>929</td>
<td>937</td>
<td>941</td>
<td>947</td>
</tr>
<tr>
<td>953</td>
<td>967</td>
<td>971</td>
<td>977</td>
<td>983</td>
</tr>
<tr>
<td>991</td>
<td>997</td>
<td>1009</td>
<td>1013</td>
<td>1019</td>
</tr>
<tr>
<td>1021</td>
<td>1031</td>
<td>1033</td>
<td>1039</td>
<td>1049</td>
</tr>
<tr>
<td>1051</td>
<td>1061</td>
<td>1063</td>
<td>1069</td>
<td>1087</td>
</tr>
<tr>
<td>1091</td>
<td>1093</td>
<td>1097</td>
<td>1103</td>
<td>1109</td>
</tr>
<tr>
<td>1117</td>
<td>1123</td>
<td>1129</td>
<td>1151</td>
<td>1153</td>
</tr>
<tr>
<td>1163</td>
<td>1171</td>
<td>1181</td>
<td>1187</td>
<td>1193</td>
</tr>
<tr>
<td>1201</td>
<td>1213</td>
<td>1217</td>
<td>1223</td>
<td>1229</td>
</tr>
<tr>
<td>1231</td>
<td>1237</td>
<td>1249</td>
<td>1259</td>
<td>1277</td>
</tr>
<tr>
<td>1279</td>
<td>1283</td>
<td>1289</td>
<td>1291</td>
<td>1297</td>
</tr>
<tr>
<td>1301</td>
<td>1303</td>
<td>1307</td>
<td>1319</td>
<td>1321</td>
</tr>
<tr>
<td>1327</td>
<td>1361</td>
<td>1367</td>
<td>1373</td>
<td>1381</td>
</tr>
<tr>
<td>1399</td>
<td>1409</td>
<td>1423</td>
<td>1427</td>
<td>1429</td>
</tr>
<tr>
<td>1433</td>
<td>1439</td>
<td>1447</td>
<td>1451</td>
<td>1453</td>
</tr>
<tr>
<td>1459</td>
<td>1471</td>
<td>1481</td>
<td>1483</td>
<td>1487</td>
</tr>
<tr>
<td>1489</td>
<td>1493</td>
<td>1499</td>
<td>1511</td>
<td>1523</td>
</tr>
<tr>
<td>1531</td>
<td>1543</td>
<td>1549</td>
<td>1553</td>
<td>1559</td>
</tr>
<tr>
<td>1567</td>
<td>1571</td>
<td>1579</td>
<td>1583</td>
<td>1597</td>
</tr>
<tr>
<td>1601</td>
<td>1607</td>
<td>1609</td>
<td>1613</td>
<td>1619</td>
</tr>
<tr>
<td>1621</td>
<td>1627</td>
<td>1637</td>
<td>1657</td>
<td>1663</td>
</tr>
<tr>
<td>1667</td>
<td>1669</td>
<td>1693</td>
<td>1697</td>
<td>1699</td>
</tr>
<tr>
<td>1709</td>
<td>1721</td>
<td>1723</td>
<td>1733</td>
<td>1741</td>
</tr>
<tr>
<td>1747</td>
<td>1753</td>
<td>1759</td>
<td>1777</td>
<td>1783</td>
</tr>
<tr>
<td>1781</td>
<td>1789</td>
<td>1801</td>
<td>1811</td>
<td>1823</td>
</tr>
<tr>
<td>1831</td>
<td>1847</td>
<td>1861</td>
<td>1867</td>
<td>1871</td>
</tr>
<tr>
<td>1873</td>
<td>1877</td>
<td>1879</td>
<td>1889</td>
<td>1901</td>
</tr>
<tr>
<td>1907</td>
<td>1913</td>
<td>1931</td>
<td>1933</td>
<td>1949</td>
</tr>
<tr>
<td>1997</td>
<td>1999</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ N = 1000 \quad \text{NEXT}\_\text{NUM} = 1000 \quad \text{NPRIMES} = 302 \]

**IPRINT = TRUE**

--- **PRIME NUMBERS BETWEEN 2 AND 50** ---

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>41</td>
<td>43</td>
<td>47</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ N = 25 \quad \text{NEXT}\_\text{NUM} = 24 \quad \text{NPRIMES} = 14 \]

--- **PRIMES FINISHED** ---
IPRINT= TRUE

-- PRIME NUMBERS BETWEEN 2 AND 100 --

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
<td>31</td>
</tr>
<tr>
<td>37</td>
<td>41</td>
<td>43</td>
<td>47</td>
<td>53</td>
</tr>
<tr>
<td>59</td>
<td>61</td>
<td>67</td>
<td>71</td>
<td>73</td>
</tr>
<tr>
<td>79</td>
<td>83</td>
<td>89</td>
<td>97</td>
<td></td>
</tr>
</tbody>
</table>

N=  50  NEXT=  49  NPRIMES=  24

- PRIMES FINISHED -

IPRINT= TRUE

-- PRIME NUMBERS BETWEEN 2 AND 300 --

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
<td>31</td>
</tr>
<tr>
<td>37</td>
<td>41</td>
<td>43</td>
<td>47</td>
<td>53</td>
</tr>
<tr>
<td>59</td>
<td>61</td>
<td>67</td>
<td>71</td>
<td>73</td>
</tr>
<tr>
<td>79</td>
<td>83</td>
<td>89</td>
<td>97</td>
<td>101</td>
</tr>
<tr>
<td>103</td>
<td>107</td>
<td>109</td>
<td>113</td>
<td>127</td>
</tr>
<tr>
<td>131</td>
<td>137</td>
<td>139</td>
<td>149</td>
<td>151</td>
</tr>
<tr>
<td>157</td>
<td>163</td>
<td>167</td>
<td>173</td>
<td>179</td>
</tr>
<tr>
<td>181</td>
<td>191</td>
<td>193</td>
<td>197</td>
<td>199</td>
</tr>
<tr>
<td>211</td>
<td>223</td>
<td>227</td>
<td>229</td>
<td>233</td>
</tr>
<tr>
<td>239</td>
<td>241</td>
<td>251</td>
<td>257</td>
<td>263</td>
</tr>
<tr>
<td>269</td>
<td>271</td>
<td>277</td>
<td>281</td>
<td>283</td>
</tr>
<tr>
<td>293</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N= 150  NEXT= 147  NPRIMES=  61

- PRIMES FINISHED -
```
1:0  PROGRAM POWER ( INPUT, OUTPUT, AUXOUT ) ;
2:0
3:1  VAR N: INTEGER ;
4:2  Z, RESULT: REAL ;
5:1  VAR OUTKEY : CHAR ;
6:2  IPRINT : BOOLEAN ;
7:2  OUTFILE: TEXT ;
8:2
9:2  (* -- GET POWER ITERATIVELY -- *)
10:2
11:1  FUNCTION IPOWER( Z:REAL; N:INTEGER ) :REAL ;
12:2  VAR I : INTEGER ;
13:3  PROD : REAL ;
14:2  BEGIN
15:3  IF Z=0 THEN IPOWER:=0 ELSE
16:5  BEGIN
17:6  PROD := 1.0 ;
18:6  FOR I := 1 TO N DO PROD:=PROD*Z ;
19:6  IPOWER := PROD ;
20:5  END ;
21:2  END ;  (* IPOWER *)
22:2
23:2
24:1  BEGIN
25:2  WRITE( '++TYPE P FOR PRINTER, ELSE CRT OUTPUT ASSUMED: ' ) ;
26:2  READLN( OUTKEY ) ;
27:2  IPRINT := OUTKEY = 'P' ;
28:2  IF IPRINT = TRUE THEN OPEN( OUTFILE, AUXOUT )
29:3   ELSE OPEN( OUTFILE, OUTPUT ) ;
30:2  READ( Z,N ) ;
31:2  IF N<0 THEN WRITELN( '**ERROR: NUMBER NEGATIVE!' )
32:3   ELSE
33:4   BEGIN
34:5   RESULT := IPOWER( Z,N ) ;
35:5   |
36:5   WRITELN( OUTFILE, Z:6:3, ' TO THE POWER OF', N:4,' =',RESULT )
37:4   END ;
38:1  END .
No Compilation Errors
2.000 TO THE POWER OF 3 = 8.000000E+00
5.000 TO THE POWER OF 3 = 1.250000E+02
2.000 TO THE POWER OF 10 = 1.024000E+03
2.000 TO THE POWER OF 16 = 6.553600E+04
```
Recursive Function

PROGRAM RPOWER ( INPUT, OUTPUT, AUXOUT ) ;

VAR
   N : INTEGER ;
   Z, RESULT : REAL ;
   OUTKEY : CHAR ;
   IPRINT : BOOLEAN ;
   OUTFILE : TEXT ;

(* -- EXAMPLE OF A RECURSIVE FUNCTION -- *)

FUNCTION RPOWER( Z:REAL; N:INTEGER ):REAL ;
BEGIN
   IF Z <= 0 THEN RESULT := 0 ELSE
   IF N = 0 THEN RESULT := 1.0
   ELSE RESULT := Z*RPOWER( Z,N-1 ) ;
END ; (* RPOWER *)

BEGIN
   WRITE( '++TYPE P FOR PRINTER. ELSE CRT OUTPUT ASSUMED: ' );
   READLN( OUTKEY ) ;
   IPRINT := OUTKEY = 'P' ;
   IF IPRINT = TRUE THEN OPEN( OUTFILE, AUXOUT )
   ELSE OPEN( OUTFILE, OUTPUT ) ;
   READ( Z;N ) ;
   IF N<0 THEN WRITELN( '***ERROR: NUMBER NEGATIVE!' )
   ELSE
      BEGIN
         RESULT := RPOWER( Z,N ) ;
      END ;
   WRITELN( OUTFILE, Z:6:3, ' TO THE POWER OF', N:4,' =',RESULT );
END ;

No Compilation Errors

2.000 TO THE POWER OF  3  =  8.000000E+00
5.000 TO THE POWER OF  3  =  1.250000E+02
2.000 TO THE POWER OF 10  =  1.024000E+03
2.000 TO THE POWER OF 16  =  6.553600E+04

117
PROGRAM FAN ( INPUT, OUTPUT );
VAR
I, J, K, L, M, N : INTEGER;
BEGIN
M := 3;
N := 3;
FOR I := 1 TO N DO
  FOR J := 1 TO N DO
    M := M + 1;
  WRITELN( M );
END.

No Compilation Errors

result: 30 ✓
Program PFAN

1:0 PROGRAM PFAN (INPUT, OUTPUT) ;
2:1 VAR
3:2 I, J, K, N : INTEGER ;
4:1 BEGIN
5:2 N := 3 ;
6:2 FOR I := 1 TO N DO
7:3 FOR J := 1 TO N DO
8:4 FOR K := 1 TO N DO
9:5 N := N + 1 ;
10:2 WRITELN( N ) ;
11:1 END.

Self-explanatory

No Compilation Errors

result: -76364 ! / please note
FLEX FILE SYSTEM

Minimal file size: 1 sector
Maximal file size: all sectors of the diskette

File identification:
⟨filename⟩.⟨extension⟩

file name: letter + 7 additional characters (letters, numbers —or—)

file name extension:
Letter + 2 additional characters (as above)
TXT for text (editor) files (source program, data, etc.)
BIN for binary program files (object programs)
CMD for utility command files
BAK for Editor Backup files
CO for Pascal result files

e.g. 0. COPY.CMD
  0. PC. CMD
  1. JEAN. BIN

SOME UTILITY COMMANDS
++ DIR, ⟨unit no.⟩
→ list of all files stored on the diskette

example: DIR, 0 DIR, 1

++ LIST, ⟨unit no.⟩
→ list of all files with extension TXT on the diskette

++ COPY, 0.PROG1. TXT, 1.PROG2. TXT
→ copy file PROG1.TXT from unit 0
to file PROG2.TXT on unit 1

++ COPY, 0,1
→ copy all files on unit 0 to unit 1.
**Important for BACKUP!**

++ P, ⟨utility command name and parameters⟩
→ route command output to the printer.

example: P, DIR, 1

→ Output file list of diskette in unit 1 on printer
+ + + RENAME, ⟨file spec.1⟩, ⟨file spec. 2⟩
to rename a file

example: RENAME, 1.FRED.TXT, 1.JEAN.TXT

+ + + DELETE, ⟨file spec.⟩
to delete a file
example: DELETE 1.PROG2.BIN

+ + + EDIT, ⟨source file⟩, ⟨destination file⟩
  if omitted
  → destination = source

basic syntax:
⟨line no.⟩ = ⟨text in the line⟩ or # CR
# ⟨line no.⟩ ⟨command⟩ ⟨optional parameters⟩

Directives:
SAVE (or S) to save the edited file in memory on the destination file

EDITor commands (continued)

Some EDIT commands:
  P  to print                      # 8P15
      # 1P!
  I  to insert text after the line no. specified    # 1251
  D  to delete a line or several lines               # 8D
      # 129 D6
  C  to replace (change)
      # ⟨line no.⟩C/⟨old text⟩/⟨new text⟩/    # 17C/NEW/NEW/

For more details please consult the manual

PASCAL COMPILER
+ + + PC, ⟨source file⟩, ⟨destination file⟩, >> options

examples

+ + + PC, ⟨MOTEURS,⟩ MOTEURS, >> TERM L D

  compile program MOTEURS. TXT
  into code MOTEURS. CO
  for use with the Symbolic Debugger
  and list the program on the terminal.
+ + + PC, ⟨MOTEURS,⟩ MOTEURS, >> P L O P = 66

as above, but produce object code for Assembly and Linkage Loading and print the program
with 66 lines per page.

SYMBOLIC DEBUGGER FOR PASCAL PROGRAMS

+ + + DB, filename
example: DB, MOTEURS
or DB, 1.MOTEURS

→ assemble and load program
MOTEURS compiled by PASCAL

commands:

H help
G to start the program
Q to leave the program
B107 to implant a break point at line 107
D NOM-MOTEUR to display the actual setting of the variable NOM-MOTEUR
P to proceed from a break point
E ON E OFF turn on/off procedure entry/exit flag
C NOM-MOTEUR STOPPER change setting of a variable
R107 to remove breakpoint at line 107
R to remove all breakpoints

GENERATION OF A BINARY OBJECT PROGRAM (especially in view to store it in PROM)

+ + + PC, ⟨MOTEURS,⟩ MOTEURS 0 PASCAL Compiler

+ + + LC

Pascal compiler output file name : MOTEURS
Pascal program name : MOTEURS
Auto stop ? Y
System stack size : 200
Starting load location : 100
Library drive number : 0
Additional files to load :
Load options :
Map options :

+ + + CHAIN, MOTEURS Assembler + Chain & Linking Loader

+ + + MAP, 1.MOTEURS.BIN
to get the field covered
0100-10F6
by the binary program
0100
+ + + 1.MOTEURS.BIN

to execute the object program MOTEURS

P.S. Example only. More details to be found from manual.

+ + + CHAIN, MOTEURS

(Assembly Linking & Loading)

will automatically execute the following steps:

Example

assembly of program
RA, (MOTEURS.CO,) MOTEURS.CA O

assembly of pre-setup source

RA, (MOTEURS.PS,) MOTEURS.PA O
LL
?STRP = $0100
?LOAD = MOTEURS.PA MOTEURS.CA
?LIB = 0. RL
?OBJA = MOTEURS.BIN
?STRP = $0100
?LOAD = MOTEURS.PA MOTEURS.CA
?LIB = 0. RL
?OBJA = MOTEURS.BIN
?MAPC
PSCT SIZE = 45E8 START = 0100 END = 46E7
SYMBOL TABLE USAGE : USED = 79
MODULE TABLE USAGE : USED = 28 OUT OF = 58
? EXIT
End chain

+ + +

Example of a simple program

PROGRAM SQUARE (INPUT, OUTPUT);
VAR BASE : INTEGER;
BEGIN
READ (BASE);
WRITELN ('HELLO, THE SQUARE OF',
BASE, 'IS:', SQR(BASE));
END.

or

program square (input, output);
var
base : integer;
begin
    read (base);
    writeln ('HELLO, the square of',
        base, 'is', sqr (base));
end.