SECOND SCHOOL ON ADVANCED TECHNIQUES
IN COMPUTATIONAL PHYSICS
(18 January - 12 February 1988)

SMR.282/ 16

MODULA - 2 AND PASCAL

PART 2: PASCAL

V.B.A.PACK

University of Ghent, Belgium

# TURBO-PASCAL REFERENCE CARD

To start TURBO Pascal, enter the command (MS-DOS) :        turbo
Answer Y to the question "Include error messages (Y/N)?".
(! The file TURBO.MSG, which contains the compiler error messages, has to be in
  your current directory. If not, copy it from the root directory on drive C: )
A menu of TURBO commands and the TURBO prompt (>) appears.
To give a TURBO command, press the associated letter, no <RETURN>.
Press <RETURN> to see the menu again, if it has disappeared from the screen.

## TURBO-COMMANDS

L : change current logged drive
A : change current directory on current drive
W : select a workfile, which will be used to edit, compile, run and save
E : invoke the full-screen editor and edit the workfile
C : compile the workfile
R : run the program
S : save the workfile on disk
D : directory
Q : exit TURBO system, back to MS-DOS

## TURBO-EDITOR

<CTRL>-<K> <D> : exit editor, back to TURBO system
<RETURN> : insert new line at cursor position, if in insert mode

arrow keys : character left or right, line up or down
<PGDN> : page down
<PGUP> : page up
<HOME> : to beginning of line
<END> : to end of line

<INS> : toggle between insert and overwrite mode
<DEL> : delete character under cursor
<   > : delete left character
<CTRL>-<Y> : delete line

<CTRL>-<K> <B> : mark block begin
<CTRL>-<K> <K> : mark block end
<CTRL>-<K> <H> : hide/display block
<CTRL>-<K> <C> : copy marked block to cursor position
<CTRL>-<K> <V> : move marked block to cursor position
<CTRL>-<K> <Y> : delete marked block

<CTRL>-<Q> <A> : find and replace a given string, option G for global search

## COMPILER DIRECTIVES

U+ : user interrupt active (by <CTRL>-<C>)
B- : standard Pascal input/output
R+ : index range check
Put them at the beginning of your program source as pseudo-comment : {$B-,U+,R+}

```pascal
{$B-,U+,R-}
PROGRAM Example1 (input, output);
CONST NMax = 100;
VAR Nr, i, Max, Min : INTEGER;
    Number          : ARRAY [1..NMax] OF INTEGER;
BEGIN
  Read (Number[1]);
  Nr := 2;
  Max := Number[1]; Min := Number[1];
  Read (Number[Nr]);
  WHILE Number[Nr] <> -1 DO
    BEGIN
    IF Number[Nr] > Max THEN Max := Number[Nr]
    ELSE IF Number[Nr] < Min THEN Min := Number[Nr];
    Nr := Nr + 1;
    Read (Number[Nr])
    END { while };
  FOR i := 1 TO Nr - 1 DO
    WriteLn (Number[i]:6, Max-Number[i]:6, Number[i]-Min:6)
END.
```

```
{$B-,U+,R+}
PROGRAM Example2 (input, output);
{ Count the occurrences of the letters 'a'..'z'
  in a line of text, followed by '.'          }
TYPE
   Letters      = 'a'..'z';
   Occurrences = ARRAY [Letters] OF INTEGER;
VAR
   Occ : Occurrences;
   Ch  : CHAR;
BEGIN
   FOR Ch := 'a' TO 'z' DO Occ[Ch] := 0;
   Read (Ch);
   WHILE Ch <> '.' DO
     BEGIN
     IF (Ch >= 'a') AND (Ch <= 'z') THEN
       Occ[Ch] := Occ[Ch] + 1;
     Read (Ch)
     END { while };
   FOR Ch := 'a' TO 'z' DO
     IF Occ[Ch] <> 0 THEN
       WriteLn (Ch, '    ', Occ[Ch]:2)
END.
```

```
{$B-,U+,R+}
PROGRAM Example3 (input, output);
{ Check whether a word is a palindrome }
VAR Word                  : ARRAY [1..80] OF cHAR;
    Ch                    : CHAR;
    Len, first, second : INTEGER;
    palindrome            : BOOLEAN;
BEGIN
   Read (Word[1]);
   WHILE NOT Eof DO
     BEGIN
     Len := 1;
     WHILE NOT Eoln DO
       BEGIN
       Len := Len + 1;
       Read (Word[Len])
       END { while };
     ReadLn;
     FOR first := 1 TO Len DO Write (word[first]);
     first := 1; second := Len; palindrome := TRUE;
     WHILE (first < second) AND palindrome DO
       BEGIN
       IF (Word[first] <> Word[second]) THEN palindrome := FALSE;
       first := first + 1;
       second := second - 1;
       END { while };
     IF palindrome THEN WriteLn (' is a palindrome')
     ELSE WriteLn (' is not a palindrome');
     Read (Word[1])
     END { while }
END.
```

```pascal
{$B-,U+,R+}
PROGRAM Example5 (input, output);
{ Compute product of two matrices
  and input/output of matrices      }

CONST
  NMax = 20;
TYPE
  Matrices = ARRAY [1..Nmax, 1..Nmax] OF REAL;

PROCEDURE ProdMat (dim : INTEGER; VAR ml, m2, res : Matrices);
   (* Computes the product of two square matrices ml and m2
      of dimension dim, result is res                        *)
   VAR
     i, j, k : INTEGER;
     el      : REAL;
   BEGIN
     FOR i := 1 TO dim DO
       FOR j := 1 TO dim DO
         BEGIN
         el := 0.0;
         FOR k := 1 TO dim DO
           el := el + ml[i,k] * m2[k,j];
         res[i,j] := el
         END { for }
   END { ProdMat };

PROCEDURE ReadMat (dim : INTEGER; VAR mat : Matrices);
   VAR
     i, j : INTEGER;
   BEGIN
     FOR i := 1 TO dim DO
       BEGIN
       Read (mat[i,1]);
       FOR j := 2 TO dim DO
         Read (mat[i,j]);
       END { for }
   END { ReadMat };

PROCEDURE WriteMat (dim : INTEGER; VAR mat : Matrices);
   VAR
     i, j : INTEGER;
   BEGIN
     FOR i := 1 TO dim DO
       BEGIN
       Write (mat[i,1]:12:6);
       FOR j := 2 TO dim DO
         BEGIN
         Write ('   '); Write (mat[i,j]:12:6)
         END { for };
       WriteLn
       END { for }
   END { WriteMat };

VAR
  Dimension : INTEGER;
  Mat1, Mat2, Prod : Matrices;

BEGIN
  Write ('Dimension of the matrices ? ');
  Read (Dimension);
  Write ('Give elements of first matrix, in rows :'); WriteLn;
  ReadMat (Dimension, Mat1);
  Write ('Give elements of second matrix, in rows :'); WriteLn;
  ReadMat (Dimension, Mat2);

  ProdMat (Dimension, Mat1, Mat2, Prod);

  Write ('Product of the 2 matrices :'); WriteLn;
  WriteMat (Dimension, Prod)
END.
```

```
{$B-,U+,R+}
PROGRAM Example6 (input, output);
( Compute number of digits in a positive integer number )

FUNCTION NrDigits (num : INTEGER) : INTEGER;
   VAR
     tmp : INTEGER;
   BEGIN
     tmp := 0;
     REPEAT
       tmp := tmp + 1;
       num := num DIV 10
     UNTIL num = 0;
     NrDigits := tmp
   END ( NrDigits );

VAR
   Num : INTEGER;

BEGIN
   Write ('Give a cardinal number : ');
   Read (Num);
   WriteLn ('Number of digits = ', NrDigits (Num):6);
   WriteLn ('Number = ', Num:6)
END.
```

# pascal

## USER MANUAL AND REPORT

### SECOND EDITION

**Kathleen Jensen**
**Niklaus Wirth**

## 1. Introduction

The development of the language _Pascal_ is based on two principa
aims. The first is to make available a language suitable t
teach programming as a systematic discipline based on certai
fundamental concepts clearly and naturally reflected by th
language. The second is to develop implementations of thi
language which are both reliable and efficient on presentl
available computers.

The desire for a new language for the purpose of teachin
programming is due to my dissatisfaction with the presently use
major languages whose features and constructs too often canno
be explained logically and convincingly and which too often def
systematic reasoning. Along with this dissatisfaction goes a
conviction that the language in which the student is taught t
express his ideas profoundly influences his habits of though
and invention, and that the disorder governing these language
directly imposes itself onto the programming style of th
students.

There is of course plenty of reason to be cautious with th
introduction of yet another programming language, and th
objection against teaching programming in a language which i
not widely used and accepted has undoubtedly some
justification, at least based on short term commercia
reasoning. However, the choice of a language for teaching base
on its widespread acceptance and availability, together with th
fact that the language most widely taught is thereafter going t
be the one most widely used, forms the safest recipe for
stagnation in a subject of such profound pedagogical influence.
I consider it therefore well worth-while to make an effort t
break this vicious circle.

Of course a new language should not be developed just for th
sake of novelty; existing languages should be used as a basi
for development wherever they meet the criteria mentioned and d
not impede a systematic structure. In that sense Algol 60 wa
used as a basis for Pascal, since it meets the demands wit
respect to teaching to a much higher degree than any other
standard language. Thus the principles of structuring, and i
fact the form of expressions, are copied from Algol 60. It was,
however not deemed appropriate to adopt Algol 60 as a subset of
Pascal; certain construction principles,particularly those of
declarations, would have been incompatible with those allowing a
natural and convenient representation of the additional feature
of Pascal.

The main extensions relative to Algol 60 lie in the domain o
data structuring facilities, since their lack in Algol 60 wa
considered as the prime cause for its relatively narrow range o
applicability. The introduction of record and file structure
should make it possible to solve commercial type problems wit
Pascal, or at least to employ it successfully to demonstrat
such problems in a programming course.

# Summary of the language
-----------------------

algorithm or computer program consists of two essential
ts, a description of _actions_ which are to be performed, and a
cription of the _data_, which are manipulated by these actions.
ions are described by so-called _statements_, and data are
cribed by so-called _declarations_ and _definitions_.

data are represented by values of _variables_. Every variable
urring in a statement must be introduced by a _variable_
_laration_ which associates an identifier and a data type with
t variable. The _data type_ essentially defines the set of
ues which may be assumed by that variable. A data type may in
cal be either directly described in the variable declaration.
it may be referenced by a type identifier, in which case this
ntifier must be described by an explicit _type definition_.

basic data types are the _scalar_ types. Their definition
icates an ordered set of values, i.e. introduces identifiers
nding for each value in the set. Apart from the definable
lar types, there exist four _standard basic types_: Boolean,
_soer_, _char_, and _real_. Except for the type Boolean, their
ues are not denoted by identifiers, but instead by numbers
quotations respectively. These are syntactically distinct
m identifiers. The set of values of type char is the
racter set available on a particular installation.

type may also be defined as a _subrange_ of a scalar type by
icating the smallest and the largest value of the subrange.

_uctured types_ are defined by describing the types of their
ponents and by indicating a _structuring method_. The various
ucturing methods differ in the selection mechanism serving to
ect the components of a variable of the structured type. In
cal, there are four basic structuring methods available:
ay structure, record structure, set structure, and file
ucture.

an _array structure_, all components are of the same type. A
ponent is selected by an array selector, or _computable index_,
se type is indicated in the array type definition and which
t be scalar. It is usually a programmer-defined scalar type,
a subrange of the type integer. Given a value of the index
s, an array selector yields a value of the component type.
ry array variable can therefore be regarded as a mapping of
index type onto the component type. The time needed for a
ection does not depend on the value of the selector (index).
array structure is therefore called a _random-access_
_ucture_.

a _record structure_, the components (called _fields_) are not
essarily of the same type. In order that the type of a
ected component be evident from the program text (without
cuting the program), a record selector is not a computable
ue, but instead is an identifier uniquely denoting the
ponent to be selected. These component identifiers are

declared in the record type definition. Again, the time needed
to access a selected component does not depend on the selector,
and the record is therefore also a random-access structure.

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

A _set structure_ defines the set of values which is the powerset
of its base type, i.e. the set of all subsets of values of the
base type. The base type must be a scalar type, and will usually
be a programmer-defined scalar type or a subrange of the type
integer.

A _file structure_ is a _sequence_ of components of the same type. A
natural ordering of the components is defined through the
sequence. At any instance, only one component is directly
accessible. The other components are made accessible by
progressing sequentially through the file. A file is generated
by sequentially appending components at its end. Consequently,
the file type definition does not determine the number of
components.

Variables declared in explicit declarations are called _static_.
The declaration associates an identifier with the variable which
is used to refer to the variable. In contrast, variables may be
generated by an executable statement. Such a _dynamic_ generation
yields a so-called _pointer_ (a substitute for an explicit
identifier) which subsequently serves to refer to the variable.
This pointer may be assigned to other variables, namely
variables of type pointer. Every pointer variable may assume
values pointing to variables of the same type T only, and it is
said to be _bound_ to this type T. It may, however, also assume
the value _nil_, which points to no variable. Because pointer
variables may also occur as components of structured variables,
which are themselves dynamically generated, the use of pointers
permits the representation of finite graphs in full generality.

The most fundamental statement is the _assignment statement_. It
specifies that a newly computed value be assigned to a variable
(or components of a variable). The value is obtained by
evaluating an _expression_. Expressions consist of variables,
constants, sets, operators and functions operating on the
denoted quantities and producing new values. Variables,
constants, and functions are either declared in the program or
are standard entities. Pascal defines a fixed set of operators,
each of which can be regarded as describing a mapping from the
operand types into the result type. The set of operators is
subdivided into groups of

1. _arithmetic operators_ of addition, subtraction, sign

inversion, multiplication, division, and computing the remainder.

2. Boolean operators of negation, union (or), and conjunction (and).

3. set operators of union, intersection, and set difference.

4. relational operators of equality, inequality, ordering, set membership and set inclusion. The results of relational operations are of type Boolean.

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

A statement can be given a name (identifier), and be referenced through that identifier. The statement is then called a procedure, and its declaration a procedure declaration. Such a declaration may additionally contain a set of variable declarations, type definitions and further procedure declarations. The variables, types and procedures thus declared can be referenced only within the procedure itself, and are therefore called local to the procedure. Their identifiers have significance only within the program text which constitutes the procedure declaration and which is called the scope of these identifiers. Since procedures may be declared local to other procedures, scopes may be nested. Entities which are declared in the main program, i.e. not local to some procedure, are called global. A procedure has a fixed number of parameters, each of which is denoted within the procedure by an identifier called the formal parameter. Upon an activation of the procedure statement, an actual quantity has to be indicated for each parameter which can be referenced from within the procedure through the formal parameter. This quantity is called the actual parameter. There are four kinds of parameters: value parameters, variable parameters, procedure and function parameters. In the first case, the actual parameter is an expression which is evaluated once. The formal parameter represents a local variable to which the result of this evaluation is assigned before the execution of the procedure (or function). In the case of a variable parameter, the actual parameter is a variable and the formal parameter stands for this variable. Possible indices are evaluated before execution of the procedure (or function). In the case of procedure or function

parameters, the actual parameter is a procedure or function identifier.

functions are declared analogously to procedures. The only difference lies in the fact that a function yields a result which is confined to a scalar or pointer type and must be specified in the function declaration. Functions may therefore be used as constituents of expressions. In order to eliminate side-effects, assignments to non-local variables should be avoided within function declarations.

## 3. Notation, terminology, and vocabulary

According to traditional Backus-Naur form, syntactic constructs are denoted by English words enclosed between the angular brackets < and > . These words also describe the nature or meaning of the construct, and are used in the accompanying description of semantics. Possible repetition of a construct is indicated by enclosing the construct within metabrackets { and }. The symbol <empty> denotes the null sequence of symbols.

The basic vocabulary of Pascal consists of basic symbols classified into letters, digits, and special symbols.

```
<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|
             W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|
             s|t|u|v|w|x|y|z

<digit>  ::= 0|1|2|3|4|5|6|7|8|9
<special symbol> ::=
      + | - | * | / | = | <> | < | > | <= | >= | ( | ) |
      [ | ] | | | | | | | := | . | , | ; | : | ' | ! | div |
      mod | nil | in | or | and | not | if | then | else |
      case | of | repeat | until | while | do | for | to |
      downto | begin | end | with | goto | const | var |
      type | array | record | set | file | function |
      procedure | label | packed | program
```

The construct
    { <any sequence of symbols not containing "}"> }
may be inserted between any two identifiers, numbers (cf. 4), or special symbols. It is called a comment and may be removed from the program text without altering its meaning. The symbols { and } do not occur otherwise in the language, and when appearing in syntactic descriptions they are meta-symbols like | and ::= . The symbol pairs (* and *) are used as synonyms for { and }.

## 4. Identifiers, Numbers, and Strings

Identifiers serve to denote constants, types, variables, procedures and functions. Their association must be unique within their scope of validity, i.e. within the procedure or function in which they are declared (cf. 10 and 11).

```
<identifier> ::= <letter>[<letter or digit>]
<letter or digit> ::= <letter> | <digit>
```

usual decimal notation is used for numbers, which are the
stants of the data types integer and real (see 6.1.2.). The
ter E preceding the scale factor is pronounced as "times 10
the power of".

```
<digit sequence> ::= <digit>[<digit>]
<unsigned integer> ::= <digit sequence>
<unsigned real> ::= <unsigned integer>.<digit sequence> |
      <unsigned integer>.<digit sequence>E<scale factor> |
      <unsigned integer> E <scale factor>
<unsigned number> ::= <unsigned integer> | <unsigned real>
<scale factor> ::= <unsigned integer> |
                    <sign><unsigned integer>
<sign> ::= + | -
```

mples:

|   |     |     |      |         |
|---|-----|-----|------|---------|
| 1 | 100 | 0.1 | 5E-3 | 87.35E+8 |

uences of characters enclosed by quote marks are called
inos. Strings consisting of a single character are the
stants of the standard type char (see 6.1.2). Strings
sisting of n (>1) enclosed characters are the constants of
types (see 6.2.1)

    packed array [1..n] of char

e: If the string is to contain a quote mark, then this quote
    mark is to be written twice.

    <string> ::= '<character>[<character>]'

mples:

```
        'A'         ';'         '...'
        'PASCAL'         'THIS IS A STRING'
```

Constant definitions
--------------------

constant definition introduces an identifier as a synonym to a
stant.

```
<constant identifier> ::= <identifier>
<constant> ::= <unsigned number> | <sign><unsigned number> |
    <constant identifier> | <sign><constant identifier> |
    <string>
<constant definition> ::= <identifier> = <constant>
```

# 6. Data type definitions

A data type determines the set of values which variables of that
type may assume and associates an identifier with the type.

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

## 6.1. Simple types

```
<simple type> ::= <scalar type> | <subrange type> |
                    <type identifier>
<type identifier> ::= <identifier>
```

### 6.1.1. Scalar types

A scalar type defines an ordered set of values by enumeration of
the identifiers which denote these values.

    <scalar type> ::= (<identifier> [,<identifier>] )

Examples:
    (red, orange, yellow, green, blue)
    (club, diamond, heart, spade)
    (Monday, Tuesday, Wednesday, Thursday, Friday,
     Saturday, Sunday)

Functions applying to all scalar types (except real) are :

    succ    the succeeding value (in the enumeration)
    pred    the preceding value (in the enumeration)

### 6.1.2. Standard types

The following types are standard in Pascal:

    integer    The values are a subset of the whole numbers
               defined by individual implementations. Its values
               are the integers (see 4).

    real       Its values are a subset of the real numbers
               depending on the particular implementation. The
               values are denoted by real numbers (see 4).

    Boolean    Its values are the truth values denoted by the
               identifiers true and false.

    char       Its values are a set of characters determined by
               particular implementations. They are denoted by
               the characters themselves enclosed within quotes.

## 6.1.3. Subrange types

A type may be defined as a subrange of another scalar type by indication of the least and the largest value in the subrange. The first constant specifies the lower bound, and must not be greater than the upper bound.

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

Examples:    1..100
             -10 .. +10
             Monday .. Friday

## 6.2. Structured types

A structured type is characterised by the type(s) of its components and by its structuring method. Moreover, a structured type definition may contain an indication of the preferred data representation. If a definition is prefixed with the symbol packed, this has in general no effect on the meaning of a program (for a restriction see 9.1.2.); but it is a hint to the compiler that storage should be economized even at the price of some loss in efficiency of access, and even if this may expand the code necessary for expressing access to components of the structure.

```
<structured type> ::= <unpacked structured type> |
        packed <unpacked structured type>
<unpacked structured type> ::= <array type> |
    <record type> | <set type> | <file type>
```

## 6.2.1. Array types

An array type is a structure consisting of a fixed number of components which are all of the same type, called the component type. The elements of the array are designated by indices, values belonging to the so-called index type. The array type definition specifies the component type as well as the index type.

```
<array type> ::= array [<index type> {,<index type>}] of
                 <component type>
<index type> ::= <simple type>
<component type> ::= <type>
```

If n index types are specified, the array type is called n-dimensional, and a component is designated by n indices.

Examples:    array [1..100] of real
             array [1..10,1..20] of 0..99
             array [Boolean] of color

## 6.2.2. Record types

A record type is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a field, its type and an identifier which denotes it. The scope of these so-called field identifiers is the record definition itself, and they are also accessible within a field designator (cf. 7.2) referring to a record variable of this type.

A record type may have several variants, in which case a certain field may be designated as the tag field, whose value indicates which variant is assumed by the record variable at a given time. Each variant structure is identified by a case label which is a constant of the type of the tag field.

```
<record type> ::= record <field list> end
<field list> ::= <fixed part> | <fixed part>;<variant part> |
                 <variant part>
<fixed part> ::= <record section> {;<record section>}
<record section> ::=
    <field identifier>{,<field identifier>} : <type> | <empty>
<variant part> ::= 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>
```

Examples:    record day: 1..31;
                    month: 1..12;
                    year: integer
             end

             record name, firstname: alfa;
                    age: 0..99;
                    married: Boolean
             end

             record x,y: real;
                    area: real;
                 case s: shape of
                 triangle: (side: real;
                            inclination, angle1, angle2: angle);
                 rectangle: (side1, side2 : real;
                             skew, angle3: angle);
                 circle:    (diameter: real)
             end

## 6.2.3. Set types

A set type defines the range of values which is the powerset of its so-called base type. Base types must not be structured types. Operators applicable to all set types are:

```
+     union
-     set difference
*     intersection
in    membership
```

he set difference x-y is defined as the set of all elements of
which are not members of y.

```
<set type> ::= set of <base type>
<base type> ::= <simple type>
```

## .2.4. File types

file type definition specifies a structure consisting of a
equence of components which are all of the same type. The
umber of components, called the length of the file, is not
ixed by the file type definition. A file with 0 components is
alled empty.

```
<file type> ::= file of <type>
```

iles with component type char are called textfiles, and are a
pecial case insofar as the component range of values must be
onsidered as extended by a marker denoting the end of a line.
his marker allows textfiles to be substructured into lines. The
ype text is a standard type predeclared as

```
type text = file of char
```

## .3. Pointer types

ariables which are declared in a program (see 7.) are
ccessible by their identifier. They exist during the entire
xecution process of the procedure (scope) to which the variable
s local, and these variables are therefore called static (or
tatically allocated). In contrast, variables may also be
enerated dynamically, i.e. without any correlation to the
tructure of the program. These dynamic variables are generated
y the standard procedure new (see 10.1.2.); since they do not
ccur in an explicit variable declaration, they cannot be
eferred to by a name. Instead, access is achieved via a
o-called pointer value which is provided upon generation of the
ynamic variable. A pointer type thus consists of an unbounded
et of values pointing to elements of the same type. No
perations are defined on pointers except the assignment and the
st for equality.
e pointer value nil belongs to every pointer type; it points
 no element at all.

```
<pointer type> ::= ↑<type identifier>
```

Examples of type definitions:

```
color   = (red, yellow, green, blue)
sex     = (male, female)
text    = file of char
shape   = (triangle, rectangle, circle)
card    = array [1..80] of char
alfa    = packed array [1..10] of char
complex = record re,im: real end
person  = record name, firstname: alfa;
                 age: integer;
                 married:Boolean;
                 father, child, sibling: ↑person;
          case s: sex of
            male: (enlisted, bold: Boolean);
            female: (pregnant: Boolean;
                     size: array[1..3] of integer)
          end
```

## 7. Declarations and denotations of variables
-------------------------------------------------

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

```
<variable declaration> ::= <identifier>[,<identifier>] : <type>
```

Every declaration of a file variable f with components of type T
implies the additional declaration of a so-called buffer
variable of type T. This buffer variable is denoted by f↑ and
serves to append components to the file during generation, and
to access the file during inspection (see 7.2.3. and 10.1.1.).

Examples:
```
x,y,z: real
u,v: complex
i,j: integer
k: 0..9
p,q: Boolean
operator: (plus, minus, times)
a: array[0..63] of real
b: array[color,Boolean] of complex
c: color
f: file of char
hue1,hue2: set of color
p1,p2: ↑person
```

Denotations of variables either designate an entire variable, a
component of a variable, or a variable referenced by a pointer
(see 6.3). Variables occurring in examples in subsequent
chapters are assumed to be declared as indicated above.

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

## 7.1. Entire variables

An entire variable is denoted by its identifier.

    <entire variable> ::= <variable identifier>
    <variable identifier> ::= <identifier>

## 7.2. Component variables

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

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

### 7.2.1. Indexed variables

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

    <indexed variable> ::=
        <array variable> [<expression> [,<expression>]]
    <array variable> ::= <variable>

The types of the index expressions must correspond with the
index types declared in the definition of the array type.

Examples:
    a[12]
    a[i+j]
    b[red,true]

### 7.2.2. Field designators

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

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

Examples:
    u .re
    b[red,true] .im
    p2↑ .size

### 7.2.3. File buffers

At any time, only the one component determined by the current
file position (read/write head) is directly accessible. This
component is called the current file component and is
represented by the file's buffer variable.

    <file buffer> ::= <file variable>↑
    <file variable> ::= <variable>

## 7.3. Referenced variables

    <referenced variable> ::= <pointer variable>↑
    <pointer variable> ::= <variable>'

If p is a pointer variable which is bound to a type T , p
denotes that variable and its pointer value, whereas p↑ denotes
the variable of type T referenced by p.

Examples:
    p1↑ .father
    p1↑ .sibling↑ .child

## 8. Expressions
------------------

Expressions are constructs denoting rules of computation for
obtaining values of variables and generating new values by the
application of operators. Expressions consist of operators and
operands, i.e. variables, constants, and functions.

The rules of composition specify operator precedences according
to four classes of operators. The operator not has the highest
precedence, followed by the so-called multiplying operators,
then the so-called adding operators, and finally, with the
lowest precedence, the relational operators. Sequences of
operators of the same precedence are executed from left to
right. The rules of precedence are reflected by the following
syntax:

signed constant> ::= <unsigned number> | <string> |
           <constant identifier> | nil
ctor> ::= <variable> | <unsigned constant> |
     <function designator> | <set> | (<expression>) |
     not <factor>
t> ::= [ <element list> ]
ement list> ::= <element> [,<element>] | <empty>
ement> ::= <expression> | <expression>..<expression>
rm> ::= <factor> | <term><multiplying operator><factor>
mple expression> ::= <term> |
     <simple expression> <adding operator><term> |
     <sign><term>
pression> ::= <simple expression> |
<simple expression><relational operator><simple expression>

ressions which are members of a set must all be of the same
e, which is the base type of the set. [] denotes the empty
. and [x..y] denotes the set of all values in the interval
.y.

mples:

tors:
           x
           15
           (x+y+z)
           sin(x+y)
           [red,c,green]
           [1,5,10..19,23]
           not p

ns:
           x*y
           i/(1-i)
           p or q
           (x<=y) and (y < z)

le expressions:
           x+y
           -x
           hue1 + hue2
           i*j + 1

essions:
           x = 1.5
           p<=q
           (i<j) = (j<k)
           c in hue1

. Operators

both operands of the arithmetic operators of addition,
raction and multiplication are of type integer (or a
ange thereof), then the result is of type integer. If one of
operands is of type real, then the result is also of type
.

1. The operator not

The operator not denotes negation of its Boolean operand.

8.1.2. Multiplying operators

   <multiplying operator> ::= * | / | div | mod | and

| operator | operation | type of operands | type of result |
|---|---|---|---|
| * | multiplication set intersection | real, integer any set type T | real, integer T |
| / | division | real, integer | real |
| div | division with truncation | integer | integer |
| mod | modulus | integer | integer |
| and | logical "and" | Boolean | Boolean |

8.1.3. Adding operators

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

| operator | operation | type of operands | type of result |
|---|---|---|---|
| + | addition set union | integer, real any set type T | integer, real T |
| - | subtraction set difference | integer, real any set type T | integer, real T |
| or | logical "or" | Boolean | Boolean |

When used as operators with one operand only, - denotes sign
inversion, and + denotes the identity operation.

8.1.4. Relational operators

   <relational operator> ::= = | <> | < | <= | >= | > | in

| operator | type of operands | result |
|----------|------------------|--------|
| = <><br>< ><br><= >= | any scalar or subrange type | Boolean |
| in | any scalar or subrange type<br>and its set type respectively | Boolean |

Notice that all scalar types define ordered sets of values.

The operators <>, <=, >= stand for unequal, less or equal,and greater or equal respectively.
The operators <= and >= may also be used for comparing values of set type, and then denote set inclusion.
If p and q are Boolean expressions, p = q denotes their equivalence, and p <= q denotes implication of q by p. (Note that false < true)

The relational operators = <> < <= > >= may also be used to compare (packed) arrays with components of type char (strings), and then denote alphabetical ordering according to the collating sequence of the underlying set of characters.

## 8.2. Function designators

A function designator specifies the activation of a function. It consists of the identifier designating the function and a list of actual parameters. The parameters are variables, expressions, procedures, and functions, and are substituted for the corresponding formal parameters (cf. 9.1.2., 10, and 11).

<function designator> ::= <function identifier> |
  <function identifier>(<actual parameter>{,<actual parameter>})
<function identifier> ::= <identifier>

Examples:      Sum(a,100)
               GCD(147,k)
               sin(x+y)
               eof(f)
               ord(f†)

## 9. Statements

Statements denote algorithmic actions, and are said to be executable. They may be prefixed by a label which can be referenced by goto statements.

---

<statement>::=<unlabelled statement> |
              <label>:<unlabelled statement>
<unlabelled statement> ::= <simple statement> | .
                          <structured statement>
<label> ::= <unsigned integer>

## 9.1. Simple statements

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

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

### 9.1.1. Assignment statements

The assignment statement serves to replace the current value o a variable by a new value specified as an expression.

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

The variable (or the function) and the expression must be o identical type, with the following exceptions being permitted:

1. the type of the variable is real, and the type of th expression is integer or a subrange thereof.
2. the type of the expression is a subrange of the type of th variable, or vice-versa.

Examples:          x := y+z
                   p := (1<=i) and (i<100)
                   i := sqr(k) - (i*j)
                   hue† := [blue,succ(c)]

### 9.1.2. Procedure statements

A procedure statement serves to execute the procedure denoted b the procedure identifier. The procedure statement may contain list of actual parameters which are substituted in place o their corresponding formal parameters defined in the procedur declaration (cf. 10). The correspondence is established by th positions of the parameters in the lists of actual and forma parameters respectively. There exist four kinds of parameters so-called value parameters, variable parameters, procedur parameters (the actual parameter is a procedure identifier), an function parameters (the actual parameter is a functio identifier).

In the case of a value parameter, the actual parameter must b an expression (of which a variable is a simple case). Th

rresponding formal parameter represents a local variable of
a called procedure, and the current value of the expression is
itially assigned to this variable. In the case of a variable
rameter, the actual parameter must be a variable, and the
rresponding formal parameter represents this actual variable
ring the entire execution of the procedure. If this variable
a component of an array, its index is evaluated when the
ocedure is called. A variable parameter must be used whenever
a parameter represents a result of the procedure.

mponents of a packed structure must not appear as actual
riable parameters.

```
<procedure statement> ::= <procedure identifier> |
    <procedure identifier> (<actual parameter>
                            {,<actual parameter>})
<procedure identifier> ::= <identifier>
<actual parameter> ::= <expression> | <variable> |
    <procedure identifier> | <function identifier>
```

mples:      next
            Transpose(a,n,m)
            Bisect(fct,-1.0,+1.0,x)

.3. Goto statements

goto statement serves to indicate that further processing
uld continue at another part of the program text, namely at
place of the label.

```
<goto statement> ::= goto <label>
```

following restrictions hold concerning the applicability of
els:

The scope of a label is the procedure within which it is
defined. It is therefore not possible to jump into a
procedure.

Every label must be specified in a label declaration in the
heading of the procedure in which the label marks a
statement.

. Structured statements

uctured statements are constructs composed of other
tements which have to be executed either in sequence
mpound statement), conditionally (conditional statements), or
eatedly (repetitive statements).

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

9.2.1. Compound statements

The compound statement specifies that its component statements
are to be executed in the same sequence as they are written. The
symbols begin and end act as statement brackets.

```
<compound statement> ::= begin <statement> {;<statement>} end
```

Example:      begin z := x ; x := y; y := z end

9.2.2. Conditional statements

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

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

9.2.2.1. If statements

The if statement specifies that a statement be executed only if
a certain condition (Boolean expression) is true. If it is
false, then either no statement is to be executed, or the
statement following the symbol else is to be executed.

```
<if statement> ::= if <expression> then <statement> |
    if <expression> then <statement> else <statement>
```

The expression between the symbols if and then must be of type
Boolean.

Note:
The syntactic ambiguity arising from the construct

    if <expression-1> then if <expression-2> then <statement-1>
            else <statement-2>

is resolved by interpreting the construct as equivalent to

    if <expression-1> then
    begin if <expression-2> then <statement-1> else <statement-2>
    end

Examples:
            if x < 1.5 then z := x+y else z := 1.5
            if p1 <> nil then p1 := p1↑.father

9.2.2.2. Case statements

The case statement consists of an expression (the selector) and
a list of statements, each being labelled by a constant of the
type of the selector. It specifies that the one statement be
executed whose label is equal to the current value of the

selector.

```
<case statement> ::= case <expression> of
  <case list element> {;<case list element>} end
<case list element> ::= <case label list> : <statement> |
                        <empty>
<case label list> ::= <case label> {,<case label> }
```

Examples :
```
case operator of                    case i of
   plus:   x := x+y;                  1: x := sin(x);
   minus:  x := x-y;                  2: x := cos(x);
   times:  x := x*y                   3: x := exp(x);
end                                   4: x := ln(x)
                                    end
```

### 9.2.3. Repetitive statements

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

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

### 9.2.3.1. While statements

```
<while statement> ::= while <expression> do <statement>
```

The expression controlling repetition must be of type Boolean.
The statement is repeatedly executed until the expression
becomes false. If its value is false at the beginning, the
statement is not executed at all. The while statement

```
while B do S
```

is equivalent to

```
if B then
    begin S;
        while B do S
    end
```

Examples :
```
while a[i] <> x do i := i+1

while i>0 do
begin if odd(i) then z := z*x;
      i := i div 2;
      x := sqr(x)
end

while not eof(f) do
begin P(f↑); get(f)
end
```

### 9.2.3.2. Repeat statements

```
<repeat statement> ::=
     repeat <statement> {;<statement>} until <expression>
```

The expression controlling repetition must be of type Boolean.
The sequence of statements between the symbols repeat and until
is repeatedly executed (and at least once) until the expression
becomes true. The repeat statement

```
repeat S until B
```

is equivalent to

```
begin S;
    if not B then
        repeat S until B
end
```

Examples :
```
repeat k := i mod j;
       i := j ;
       j := k
until j = 0

repeat P(f↑); get(f)
until eof(f)
```

### 9.2.3.3. For statements

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

```
<for statement> ::=
    for <control variable> := <for list> do <statement>
<for list> ::= <initial value> to <final value> |
        <initial value> downto <final value>
<control variable> ::= <identifier>
<initial value> ::= <expression>
<final value> ::= <expression>
```

The control variable, the initial value, and the final value must be of the same scalar type (or subrange thereof), and must not be altered by the repeated statement. They cannot be of type real.

A for statement of the form

```
    for v := e1 to e2 do S
```

is equivalent to the sequence of statements

```
    v := e1; S; v := succ(v); S; ... ; v := e2; S
```

and a for statement of the form

```
for v := e1 downto e2 do S
```

is equivalent to the statement

```
    v := e1; S; v := pred(S); S; ... ; v := e2; S
```

Examples:

```
    for i := 2 to 63 do if a[i] > max then max := a[i]

    for i := 1 to n do
    for j := 1 to n do
    begin x := 0 ;
      for k := 1 to n do x := x+A[i,k]*B[k,j];
      C[i,j] := x
    end

    for c := red to blue do Q(c)
```

## 9.2.4. With statements

```
<with statement> ::= with <record variable list> do <statement>
<record variable list> ::= <record variable>{,<record variable>}
```

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

Example:

```
    with date do
    if month = 12 then
        begin month := 1; year := year + 1
        end
    else month := month+1
```

is equivalent to

```
    if date.month = 12 then
        begin date.month := 1; date.year := date.year+1
        end
    else date.month := date.month+1
```

No assignments may be made in the qualified statement to any elements of the record variable list. However, assignments are possible to the components of these variables.

## 10. Procedure declarations
-------------------------------

Procedure declarations serve to define parts of programs and to associate identifiers with them so that they can be activated by procedure statements.

```
<procedure declaration> ::= <procedure heading> <block>
<block> ::= <label declaration part>
        <constant definition part><type definition part>
        <variable declaration part>
        <procedure and function declaration part>
        <statement part>
```

The procedure heading specifies the identifier naming the procedure and the formal parameter identifiers (if any). The parameters are either value-, variable-, procedure-, or function parameters (cf. also 9.1.2.). Procedures and functions which are used as parameters to other procedures and functions must have value parameters only.

```
<procedure heading> ::= procedure <identifier> ; |
    procedure <identifier> (<formal parameter section>
                    {;<formal parameter section>} ) ;

<formal parameter section> ::=
    <parameter group> |
    var <parameter group>
    function <parameter group> |
    procedure <identifier> {,<identifier>}
<parameter group> ::= <identifier>{,<identifier>}:
    <type identifier>
```

A parameter group without preceding specifier implies that its constituents are value parameters.

The label declaration part specifies all labels which mark a

statement in the statement part.

```
<label declaration part> ::= <empty> |
    label <label> [,<label>] ;
```

The constant definition part contains all constant synonym definitions local to the procedure.

```
<constant definition part> ::= <empty> |
    const <constant definition> [;<constant definition>];
```

The type definition part contains all type definitions which are local to the procedure declaration.

```
<type definition part> ::= <empty> |
    type <type definition> [;<type definition> ];
```

The variable declaration part contains all variable declarations local to the procedure declaration.

```
<variable declaration part> ::= <empty> |
    var <variable declaration> [;<variable declaration>] ;
```

The procedure and function declaration part contains all procedure and function declarations local to the procedure declaration.

```
<procedure and function declaration part> ::=
    [ <procedure or function declaration> ;]
<procedure or function declaration> ::=
        <procedure declaration> | <function declaration>
```

The statement part specifies the algorithmic actions to be executed upon an activation of the procedure by a procedure statement.

```
<statement part> ::= <compound statement>
```

All identifiers introduced in the formal parameter part, the constant definition part, the type definition part, the variable-, procedure or function declaration parts are local to the procedure declaration which is called the scope of these identifiers. They are not known outside their scope. In the case of local variables, their values are undefined at the beginning of the statement part.

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

---

Examples of procedure declarations:

```
procedure readinteger (var f: text; var x: integer) :
var i,j: integer;
begin while f↑ = ' ' do get(f); i := 0;
    while f↑ in ['0'..'9'] do
        begin j := ord(f↑)- ord('0');
            i := 10*i + j;
            get(f)
        end;
    x := i
end

procedure Bisect(function f: real;  a,b: real; var z: real);
var m: real;
begin (assume f(a) < 0 and f(b) > 0 )
    while abs(a-b) > 1E-10*abs(a) do
    begin m := (a+b)/2.0;
        if f(m) < 0 then a := m else b :=m
    end;
    z := m
end

procedure GCD(m,n: integer; var x,y,z: integer);
var a1,a2, b1,b2,c,d,q,r: integer; (m>=0, n>0)
begin (Greatest Common Divisor x of m and n,
    Extended Euclid's Algorithm)
    a1 := 0; a2 := 1; b1 :=1; b2 := 0;
    c := m; d := n;
    while d <> 0 do
    begin (a1*m + b1*n = d, a2*m + b2*n = c,
        gcd(c,d) = gcd(m,n))
        q := c div d; r := c mod d ;
        a2 := a2 - q*a1; b2 := b2 - q*b1;
        c := d; d := r;
        r := a1; a1 := a2; a2 := r;
        r := b1; b1 := b2; b2 := r
    end;
    x := c; y := a2; z := b2
    ( x = gcd(m,n) = y*m + z*n )
end
```

## 10.1. Standard procedures

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

## .1.1. File handling procedures

**:(f)**  appends the value of the buffer variable f↑ to the file f. The effect is defined only if prior to execution the predicate eof(f) is true. eof(f) remains true, and the value of f↑ becomes undefined.

**t(f)**  advances the current file position (read/write head) to the next component, and assigns the value of this component to the buffer variable f↑. If no next component exists, then eof(f) becomes true, and the value of f↑ is not defined. The effect of get(f) is defined only if eof(f) = false prior to its execution. (see 11.1.2)

**set(f)**  resets the current file position to its beginning and assigns to the buffer variable f↑ the value of the first element of f. eof(f) becomes false, if f is not empty; otherwise f↑ is not defined, and eof(f) remains true.

**write(f)**  discards the current value of f such that a new file may be generated. eof(f) becomes true.

ncerning the procedures read, write, readln, writeln, and page a chapter 12.

## .1.2. Dynamic allocation procedures

**#(p)**  allocates a new variable v and assigns the pointer to v to the pointer variable p. If the type of v is a record type with variants, the form

**#(p,t1,....,tn)** can be used to allocate a variable of the variant with tag field values t1,....,tn. The tag field values must be listed contiguously and in the order of their declaration and must not be changed during execution.

**spose(p)**  indicates that storage occupied by the variable p↑ is no longer needed. If the second form of new was used to allocate the variable then

**spose(p,t1,...,tn)** with identical tag field values must be used to indicate that storage occupied by this variant is no longer needed.

## .1.3. Data transfer procedures

t the variables a and z be declared by

    a: array [m..n] of T
    z: packed array [u..v] of T

ere n-m >= v-u. Then the statement pack(a,i,z) means

    for j := u to v do z[j] := a[j-u+i]

d the statement unpack(z,a,i) means

    for j := u to v do a[j-u+i] := z[j]

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

## 11. Function declarations

Function declarations serve to define parts of the program which compute a scalar value or a pointer value. Functions are activated by the evaluation of a function designator (cf. 8.2) which is a constituent of an expression.

    <function declaration> ::= <function heading><block>

The function heading specifies the identifier naming the function, the formal parameters of the function, and the type of the function.

    <function heading> ::= function <identifier>:<result type>; |
        function <identifier> (<formal parameter section>
        {;<formal parameter section>}) : <result type> ;
    <result type> ::= <type identifier>

The type of the function must be a scalar, subrange, or pointer type. Within the function declaration there must be at least one assignment statement assigning a value to the function identifier. This assignment determines the result of the function. Occurrence of the function identifier in a function designator within its declaration implies recursive execution of the function.

Examples:

```
function Sqrt(x: real): real;
var x0,x1: real;
begin x1 := x; {x>1, Newton's method}
    repeat x0 := x1; x1 := (x0+ x/x0)*0.5
    until abs(x1-x0) < eps*x1 ;
    Sqrt := x0
end


function Max(a: vector; n: integer): real;
var x: real; i: integer;
begin x := a[1];
    for i := 2 to n do
    begin {x = max(a[1],....,a[i-1])}
        if x < a[i] then x := a[i]
    end;
    {x = max(a[1],....,a[n])}
    Max := x
end
```

```
function GCD(m,n: integer):integer;
begin if n=0 then GCD := m else GCD := GCD(n,m mod n)
end
```

```
function Power(x: real; y: integer): real ; {y >= 0}
var w,z: real; i: integer;
begin w := x; z := 1; i := y;
    while i > 0 do
    begin {z*(w**i) = x ** y}
        if odd(i) then z := z*w;
        i := i div 2;
        w := sqr(w)
    end;
    {z = x**y}
    Power := z
end
```

## 11.1. Standard functions

Standard functions are supposed to be predeclared in every implementation of Pascal. Any implementation may feature additional predeclared functions (cf. also 10.1).

The standard functions are listed and explained below:

### 11.1.1. Arithmetic functions

abs(x)     computes the absolute value of x. The type of x must be either real or integer, and the type of the result is the type of x.

sqr(x)     computes x**2. The type of x must be either real or integer, and the type of the result is the type of x.

sin(x)
cos(x)
exp(x)     the type of x must be either real or integer, and
ln(x)      the type of the result is real.
sqrt(x)
arctan(x)

### 11.1.2. Predicates

odd(x)     the type of x must be integer, and the result is true, if x is odd, and false otherwise.

eof(f)     eof(f) indicates, whether the file f is in the end-of-file status.

eoln(f)    indicates the end of a line in a textfile (see chapter 12).

### 11.1.3. Transfer functions

trunc(x)   the real value x is truncated to its integral part.

round(x)   the real argument x is rounded to the nearest integer.

ord(x)     x must be of a scalar type (including Boolean and char), and the result (of type integer) is the ordinal number of the value x in the set defined by the type of x.

chr(x)     x must be of type integer, and the result (of type char) is the character whose ordinal number is x (if it exists).

### 11.1.4. Further standard functions

succ(x)    x is of any scalar or subrange type, and the result is the successor value of x (if it exists).

pred(x)    x is of any scalar or subrange type, and the result is the predecessor value of x (if it exists).

## 12. Input and output
------------------------

The basis of legible input and output are textfiles (cf.6.2.4) that are passed as program parameters (cf. 13) to a PASCAL program and in its environment represent some input or output device such as a terminal, a card reader, or a line printer. In order to facilitate the handling of textfiles, the four standard procedures read, write, readln, and writeln are introduced in addition to the procedures get and put. The textfiles these standard procedures apply to must not necessarily represent input/output devices, but can also be local files. The new procedures are used with a non-standard syntax for their parameter lists, allowing, among other things, for a variable number of parameters. Moreover, the parameters must not necessarily be of type char, but may also be of certain other types, in which case the data transfer is accompanied by an implicit data conversion operation. If the first parameter is a file variable, then this is the file to be read or written. Otherwise, the standard files input and output are automatically assumed as default values in the cases of reading and writing respectively. These two files are predeclared as

                    var input, output: text

Textfiles represent a special case among file types insofar as texts are substructured into lines by so-called line markers (cf. 6.2.4.). If, upon reading a textfile f, the file position

advanced to a line marker, that is past the last character of line, then the value of the buffer variable f↑ becomes a nk, and the standard function eoln(f) (and of line) yields value true. Advancing the file position once more assigns to the first character of the next line, and eoln(f) yields se (unless the next line consists of 0 characters). Line kers, not being elements of type char, can only be generated the procedure writeln.

## 1. The procedure read

following rules hold for the procedure read: f denotes a tfile and v1...vn denote variables of the types char, integer subrange of integer), or real.

read(v1,....,vn) is equivalent to read(input,v1,....,vn)

read(f,v1,....,vn)   is   equivalent   to  read(f,v1);  ... ;
read(f,vn)

f v is a variable of type char, then read(f,v) is equivalent
o v := f↑; get(f)

f v is a variable of type integer (or subrange of integer)
r real, then read(f,v) implies the reading from f of a
equence of characters which form a number according to the
yntax of PASCAL (cf. 4.) and the assignment of that number
o v. Preceding blanks and line markers are skipped,
procedure read can also be used to read from a file f which
not a textfile. read(f,x) is in this case equivalent to
f↑: get(f).

## . The procedure readln

readln(v1,....,vn) is equivalent to readln(input,v1,....,vn)

readln(f,v1,....,vn) is equivalent to

        read(f,v1,....,vn); readln(f)

readln(f) is equivalent to

        while not eoln(f) do get(f);
        get(f)

readln is used to read and subsequently skip to the beginning
the next line.

procedure write can also be used to write onto a file f
' is not a textfile. write(f,x) is in this case equivalent
:= x; put(f).

## The procedure write

following rules hold for the procedure write: f denotes a
ile, p1,....,pn denote so-called write-parameters, e denotes
pression, m and n denote expressions of type integer.

ite(p1,....,pn) is equivalent to write(output,p1,....,pn)

---

2. write(f,p1,....,pn) is equivalent to

        write(f,p1); ... : write(f,pn)

3. The write-parameters p have the following forms:

        e:m    e:m:n    e

e represents the value to be "written" on the file f, and m and n are so-called field width parameters. If the value e, which is either a number, a character, a Boolean value, or a string requires less than m characters for its representation, then an adequate number of blanks is issued such that exactly m characters are written. If m is omitted, an implementation-defined default value will be assumed. The form with the width parameter n is applicable only if e is of type real (see rule 6).

4. If e is of type char, then
        write(f,e:m) is equivalent to
        f↑ := ' ' : put(f);       (repeated m-1 times)
        f↑ := e : put(f)
   Note: the default value for m is in this case 1.

5. If e is of type integer (or a subrange of integer), then the decimal representation of the number e will be written on the file f, preceded by an appropriate number of blanks as specified by m.

6. If e is of type real , a decimal representation of the number e is written on the file f, preceded by an appropriate number of blanks as specified by m. If the parameter n is missing (see rule 3), a floating-point representation consisting of a coefficient and a scale factor will be chosen. Otherwise a fixed-point representation with n digits after the decimal point is obtained.

7. If e is of type Boolean, then the words TRUE or FALSE are written on the file f, preceded by an appropriate number of blanks as specified by m.

8. If e is an (packed) array of characters, then the string e is written on the file f, preceded by an appropriate number of blanks as specified by m.

## 12.4. The procedure writeln

1. writeln(p1,....,pn) is equivalent to writeln(output,p1,....,pn)

2. writeln(f,p1,....,pn) is equivalent to write(f,p1,....,pn);
   writeln(f)

3. writeln(f) appends a line marker (cf.6.2.4) to the file f.

## 12.5. Additional procedures

page(f) causes skipping to the top of a new page, when the
       textfile f is printed.

## 13. Programs
   -----------

A Pascal program has the form of a procedure declaration except
for its heading.

    <program> ::= <program heading> <block> .

    <program heading> ::=
                program <identifier> (<program parameters>) ;

    <program parameters> ::= <identifier> { , <identifier> }

The identifier following the symbol program is the program name:
it has no further significance inside the program. The program
parameters denote entities that exist outside the program, and
through which the program communicates with its environment.
These entities (usually files) are called external , and must be
declared in the block which constitutes the program like
ordinary local variables.
The two standard files input and output must not be declared
(cf. 12), but have to be listed as parameters in the program
heading, if they are used. The initialising statements
reset(input) and rewrite(output) are automatically generated and
must not be specified by the programmer.

Examples:

    program copy(f,g);
    var f,g: file of real;
    begin reset(f); rewrite(g);
        while not eof(f) do
            begin gↆ := fↆ; put(g); get(f)
            end
    end.

    program copytext(input,output);
    var ch: char;
    begin
       while not eof(input) do
       begin
          while not eoln(input) do
             begin read(ch); write(ch)
             end;
          readln; writeln
       end
    end.

## 14. A standard for implementation and program interchange
   -----------------------------------------------------------

A primary motivation for the development of PASCAL was the need
for a powerful and flexible language that could be reasonably
efficiently implemented on most computers. Its features were to
be defined without reference to any particular machine in order
to facilitate the interchange of programs. The following set of
proposed restrictions is designed as a guideline for
implementors and for programmers who anticipate that their
programs be used on different computers. The purpose of these
standards is to increase the likelihood that different
implementations will be compatible, and that programs are
transferable from one installation to another.

1. Identifiers denoting distinct objects must differ over their
   first 8 characters.

2. Labels consist of at most 4 digits.

3. The implementor may set a limit to the size of a base type
   over which a set can be defined. (Consequently, a bit pattern
   representation may reasonably be used for sets.)

4. The first character on each line of printfiles may be
   interpreted as a printer control character with the following
   meanings:
        blank  :   single spacing
        '0'    :   double spacing
        '1'    :   print on top of next page
        '+'    :   no line feed (overprinting)
Representations of PASCAL in terms of available character sets
should obey the following rules:

5. Word symbols  - such as begin, end, etc. - are written as a
   sequence of letters (without surrounding escape characters).
   They may not be used as identifiers.

6. Blanks, ends of lines, and comments are considered as
   separators. An arbitrary number of separators may occur
   between any two consecutive PASCAL symbols with the following
   restriction: no separators must occur within identifiers,
   numbers, and word symbols.

7. At least one separator must occur between any pair of
   consecutive identifiers, numbers, or word symbols .