60497700

CONTROL DATA CORPORATION

PASCAL VERSION 1 REFERENCE MANUAL

CDC® OPERATING SYSTEM:
NOS 2

REVISION RECORD

Revision	Description
01 (12/01/82)	Preliminary release at PSR level 580.

REVISION LETTERS I, O, Q, AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Front Cover	–
Title Page	–
ii	01
iii/iv	01
v/vi	01
vii/viii	01
ix	01
1-1	01
2-1 thru 2-4	01
3-1 thru 3-21	01
4-1 thru 4-24	01
5-1 thru 5-10	01
6-1 thru 6-7	01
A-1 thru A-3	01
B-1 thru B-3	01
C-1	01
Index-1	01
Index-2	01
Comment Sheet	01
Mailer	–
Back Cover	–

PREFACE

This manual describes the CONTROL DATA® Pascal Version 1 language. It is intended to be used as a reference, not as a tutorial for users who are unfamiliar with a version of Pascal.

Pascal Version 1 is available under control of the NOS 2 operating system on the CDC® CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series Computer Systems.

This manual is based on the RECAU Pascal Manual (RECAU-80-117-M, revision C 801001, edition October 1980) and has been published with the written consent of RECAU, Jorgen Staunstrup, and Ewald Skov Jensen.

This release of the Pascal compiler does not fully comply with International Standard Organization standard Pascal. You can expect to make substantial changes to your compilation control statement and source code once the compiler is released under the ISO standard.

This manual is organized in the following manner:

Section 1 provides an overview of Pascal Version 1 language concepts.

Section 2 contains the basic elements that describe the Pascal Version 1 language.

Section 3 describes the program heading and the declaration and definition of data using the language elements from section 2.

Section 4 describes the statements that manipulate the declared and defined data from section 3.

Section 5 describes the statements that compile, load, and execute a Pascal program under the NOS 2 operating system.

Section 6 shows some complete Pascal programs.

Appendix A describes available character sets.

Appendix B describes compilation error messages.

Appendix C lists the reserved words.

Related material is contained in the NOS Version 2 Reference Set Volume 3, System Commands, publication number 60459680.

CONTENTS

APPENDIXES

INDEX

TABLES

NOTATIONS

Certain notations are used throughout the manual with consistant meaning. These notations are:

$\longrightarrow$	indicates the permissible direction of traversal.
⬭	contains a reserved symbol in a syntax diagram. Alphabetic characters must appear in uppercase in your source code. A complete list of reserved symbols can be found in appendix C.
▭	contains the general name of a construct that you must define; refer to the description of the named item for definition rules.
UPPERCASE	indicates a reserved symbol in the text. Alphabetic characters must appear in uppercase in your source code. A complete list of reserved words can be found in appendix C.
⋮ or . . .	indicates statements that are not shown and are not relevant to the example.

All program statements in this manual are shown in the internal Pascal character set representation. You can translate special characters into the character set used at your site by referring to appendix A, Character Sets.

[illegible]

LANGUAGE CONCEPTS 1

A Pascal program consists of three parts:

Program heading

Declarations and definitions

Statements

The program heading part names the program and lists the parameters that are used in the program.

The declarations and definitions part describes the data objects that are to be manipulated. This part contains the following sections:

Label declarations

Constant definitions

Type definitions

Variable declarations

Value declarations

Procedure and function declarations

The statements part defines the flow of program execution and manipulates the declared and defined data objects.

PASCAL SYMBOLS 2

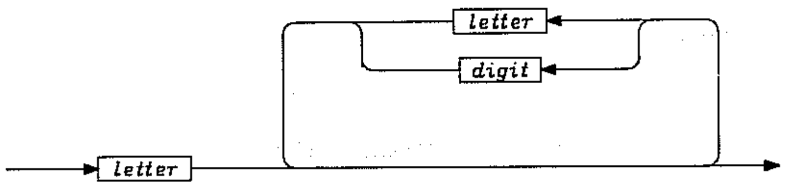
A Pascal program consists of a sequence of the Pascal symbols that are described in this section.

The set of symbols is divided into four categories: identifiers, reserved symbols, literals, and separators.

IDENTIFIERS

Identifiers are names that denote quantities that you have declared, such as constants, types, variables, values, procedures, and functions. An identifier must begin with a letter followed by any combination of letters and digits.

identifier



A letter must be in the set:

{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 digit must be in the set:

{0,1,2,3,4,5,6,7,8,9}

A number of identifiers are reserved symbols that cannot be used in other contexts. You can find a list of reserved words in appendix C.

A number of identifiers have a predefined meaning. Predefined identifiers are not reserved symbols and can be redefined.

Identifiers can be arbitrarily long, however, only the first 10 characters are significant. Identifiers that denote distinct objects must differ in their first 10 characters.

The following are examples of legal, illegal, and predefined identifiers:

Legal

```
SUITS
DAY
NO1
```

Illegal

```
1A
ΔSPACE           (Where Δ is an embedded space.)
SALES TAX
```

Predefined

```
INTEGER
REAL
TEXT
```

RESERVED SYMBOLS

Reserved symbols are symbols that are defined within the Pascal language to have a distinct meaning. This meaning cannot be changed.

You can find a list of reserved symbols in appendix C.

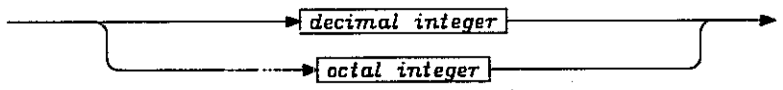
LITERALS

Literals denote values. There are five kinds of literals: integer, real, character, string, and Boolean.

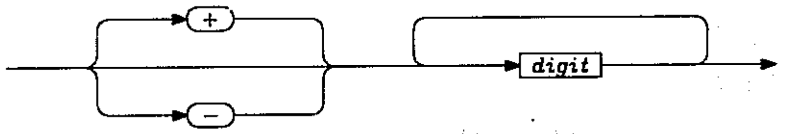
INTEGER LITERAL

An integer literal is a decimal or octal integer.

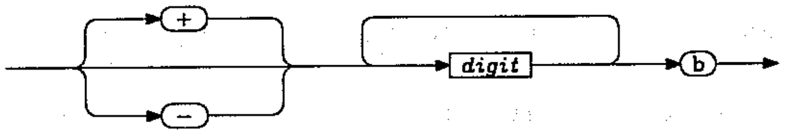
integer literal



decimal integer



octal integer



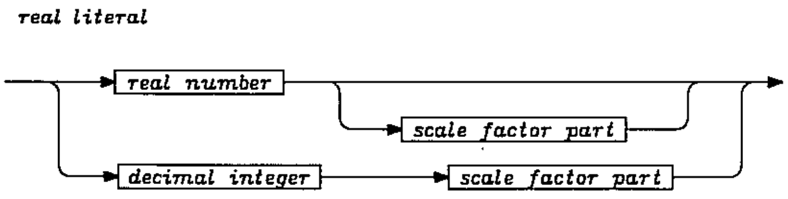
The digits in an octal integer must all be less than 8.

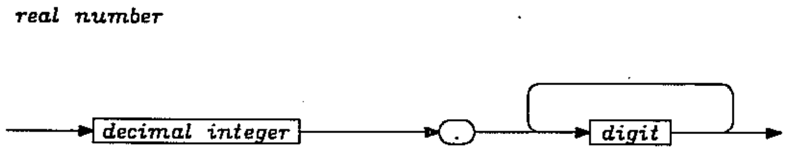
The following are examples of an integer literal:

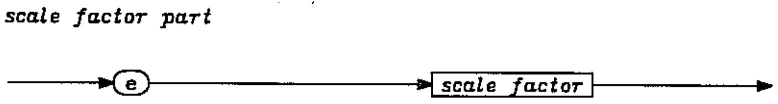
```
-714
777B
```

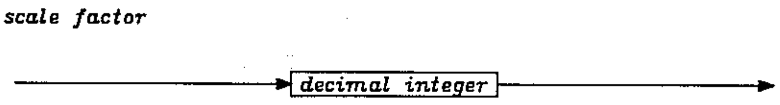
REAL LITERAL

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







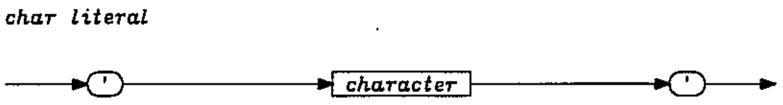


The following are examples of a real literal:

```
3.14
0.314E1
314E-2
```

CHARACTER LITERAL

A character literal is a character enclosed by single quote (') symbols.



The following are examples of a character literal:

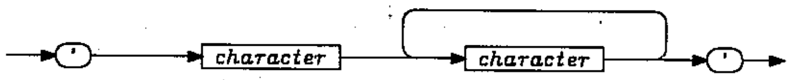
```
'C'
'+'
''''
```

Note that inside a character literal a single quote (') symbol is denoted by two quote ('') symbols.

STRING LITERAL

A string literal is a sequence of characters enclosed by single quote (') symbols.

string literal



The following are examples of a string literal:

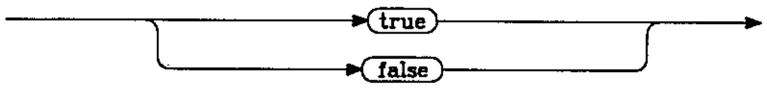
    'EQUALS'
    '''LOOKLIKE'''

Note that inside a string literal a single quote (') symbol is denoted by a two quote ('') symbols.

BOOLEAN LITERAL

A Boolean literal is one of the predefined identifiers TRUE and FALSE.

boolean literal



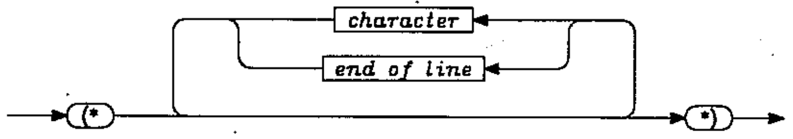
SEPARATORS

A separator is a comment or nonprinting symbol. A separator can occur between any pair of consecutive Pascal symbols. A separator may appear between any pair of consecutive identifiers or literals. A separator cannot occur within a reserved symbol, identifier, or literal.

COMMENT

A comment is a string of explanatory text. You can improve the readability of your program by adding comments, without affecting the results produced by the program.

comment



If the first character after the (* is a dollar sign ($), the comment is interpreted as a list of compiler options. See section 5 for a description of available compiler options.

NONPRINTING SYMBOLS

Nonprinting symbols are the space and the end-of-line.

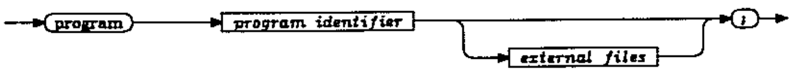
PROGRAM HEADING AND DECLARATION AND DEFINITION PARTS 3

A Pascal program must contain a program heading part, a declaration and definition part, and a statement part. This section describes the program heading part and the declaration and definition of data using the Pascal symbols described in section 2. Section 4 describes the statement part.

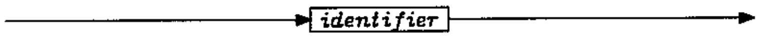
PROGRAM HEADING PART

The program heading part names the program and lists the files that are used in the program.

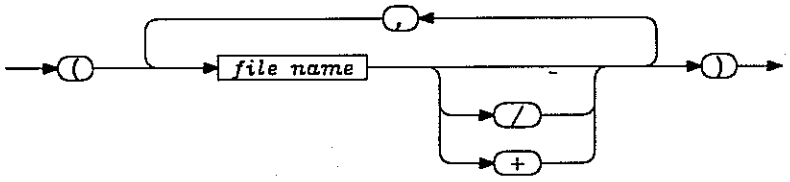
program heading



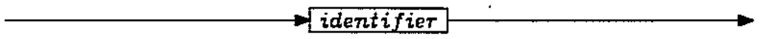
program identifier



external files



file name



The files denoted by file names must be declared as file variables in the statement part of your program, with the exceptions of INPUT and OUTPUT.

If the predefined files INPUT and OUTPUT are to be used as segmented files, a plus sign (+) must follow their names in the program heading.

If a file is to be used interactively, a slash (/) must follow the file name in the program heading.

Kinds of files and how to manipulate them are described under the heading Files in section 4.

DECLARATION AND DEFINITION PART

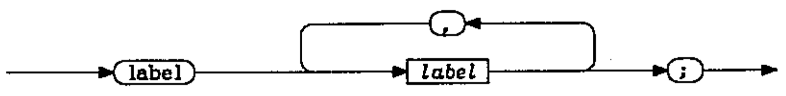
The declaration and definition part describes the data that will be manipulated in the statements part. Seven sections can appear in this part, although any of them may be empty. The section headings are: label, const, type, var, value, procedure, and function. The description of the procedure and function sections is combined under the heading Routines.

LABEL SECTION

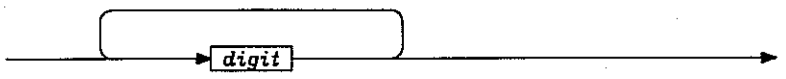
The label declaration section consists of a number of definitions of nonnegative numbers that are used as statement labels.

You must declare all labels in the label declaration part of the routine or program where it is defined.

label declaration part



label



Labels follow the same rules of scope as other quantities, which is that they can only be used in the program or routine in which they are declared.

Two labels that denote the same number are considered identical.

The following is an example of a label declaration section:

```
LABEL
    100, 200;
```

A label is defined in the statement part of your program by prefixing a statement with the label and a colon (:). For example,

```
100 : A := SUCC(THURSDAY);
```

The statement after the colon (:) cannot be a labeled statement.

You can define a label that is referenced by a GOTO only once in the compound statement of the program or routine where it is declared. For example, the statement

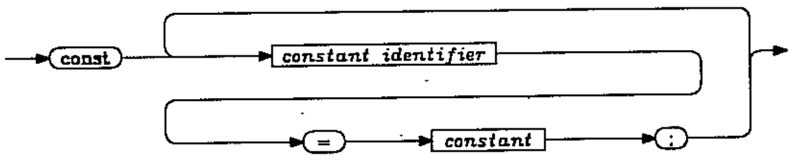
```
GOTO 200
```

can only appear once in the path of execution taken by your program.

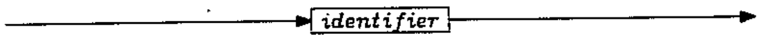
CONST SECTION

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

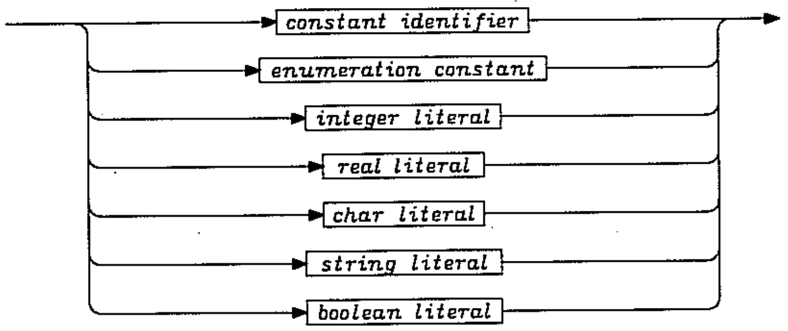
constant definition part



constant identifier



constant



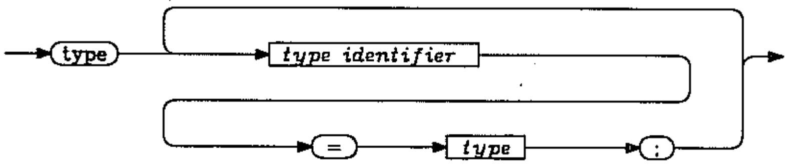
The following is an example of a constant definition part:

```
    CONST
        UPPERLIMIT = 100;
        HEADING = 'TABLE PROGRAM N = 100';
```

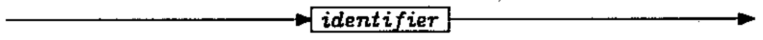
TYPE SECTION

The type declaration section defines sets of values that can be assumed by variables and expressions (operands) of that type. There are three kinds of types: simple, structured, and pointer.

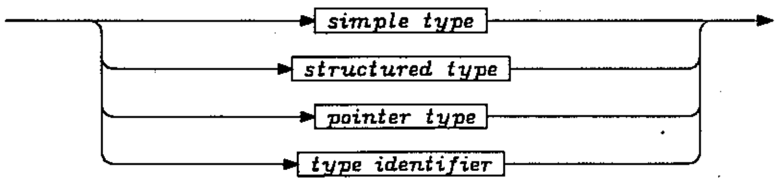
type definition part



type identifier



type



The following is an example of a type definition section:

```
    TYPE
        SUITS = (CLUB,DIAMOND,HEART,SPADE);
        DAYS = (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,
        FRIDAY,SATURDAY,SUNDAY);
        WEEKEND = FRIDAY..SUNDAY;
        MONTHS = (JANUARY,FEBRUARY,MARCH,APRIL,MAY,JUNE,JULY,AUGUST,
        SEPTEMBER,OCTOBER,NOVEMBER,DECEMBER);
        SEASONS = (WINTER,SPRING,SUMMER,AUTUMN);
        COLORS = (BLACK,RED);
```

Given the above type definition section, the following relations are true:

```
    DIAMOND <= HEART
    MONDAY < SUNDAY
    DECEMBER >= APRIL
    WEDNESDAY = SUCC(TUESDAY)
    NOVEMBER = PRED(DECEMBER)
```

The following relations are all false:

```
    CLUB >= DIAMOND
    JANUARY = FEBRUARY
    SUCC(NOVEMBER) = OCTOBER
```

The following expressions have undefined values:

```
    SUCC(SPADE)
    PRED(MONDAY)
    SUCC(DECEMBER)
```

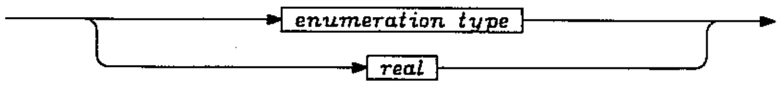
You can also define new data types in the type definition section.

In a type identifier type definition, the new type identifier takes the same type as the old type identifier.

Simple Types

There are six data types that are called simple types: Boolean, char, integer, real, scalar, and subrange.

simple type



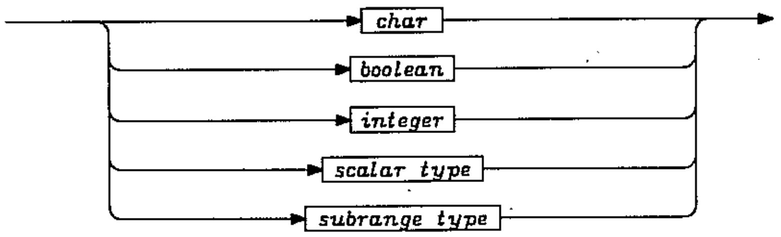
The following operators apply to operands of simple type and yield a Boolean result:

<> The operands are evaluated, then the resulting values are compared. The outcome is true if the resulting values are not equal and false if they are equal.

= The operands are evaluated, then the resulting values are compared. The outcome is true if the resulting values are equal and false if they are not.

Five of the simple types are also called enumeration types because they consist of a finite, totally ordered, set of values.

enumeration type



The following operators apply to operands of enumeration type; they take operands of compatible type and yield a Boolean result:

<	Less than
<=	Less than or equal to
=	Equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to

Boolean

The type Boolean is a predefined enumeration type. Its values are true and false.

The following operators apply to operands of type Boolean and yield a Boolean result:

Operator	Description of Operation
AND	Logical conjunction of the two operands.
OR	Logical disjunction of the two operands.
NOT	Logical negation of the operand.

The following shows the value of some Boolean expressions:

Expression	Result	Expression	Result
TRUE AND TRUE	TRUE	TRUE OR TRUE	TRUE
TRUE AND FALSE	FALSE	TRUE OR FALSE	TRUE
FALSE AND TRUE	FALSE	FALSE OR TRUE	TRUE
FALSE AND FALSE	FALSE	FALSE OR FALSE	FALSE
NOT TRUE	FALSE		
NOT FALSE	TRUE		
TRUE < TRUE	FALSE	TRUE <= TRUE	TRUE
TRUE < FALSE	FALSE	TRUE <= FALSE	FALSE
FALSE < TRUE	TRUE	FALSE <= TRUE	TRUE
FALSE < FALSE	FALSE	FALSE <= FALSE	TRUE
TRUE = TRUE	TRUE	TRUE >= TRUE	TRUE
TRUE = FALSE	FALSE	TRUE >= FALSE	TRUE
FALSE = TRUE	FALSE	FALSE >= TRUE	FALSE
FALSE = FALSE	TRUE	FALSE >= FALSE	TRUE
TRUE <> TRUE	FALSE		
TRUE <> FALSE	TRUE		
FALSE <> TRUE	TRUE		
FALSE <> FALSE	FALSE		

The value of TRUE is greater than the value of FALSE.

Char

The type char is a predefined enumeration type. Its values are the character set used at your site. Appendix A shows the translations between Pascal characters and CDC Scientific and CDC ASCII character sets.

The characters in any character set are numbered; the ordinal number of a character can be obtained from the following Pascal character table by adding the row and column number for the character in question. The ordering of the character is defined by their ordinal values.

	0	1	2	3	4	5	6	7	8	9
0		A	B	C	D	E	F	G	H	I
10	J	K	L	M	N	O	P	Q	R	S
20	T	U	V	W	X	Y	Z	0	1	2
30	3	4	5	6	7	8	9	+	−	*
40	/	(	)	$	=		,	.	#	[
50	]	:	"	_	!	&	'	?	<	>
60	@	\	↑	;						

The identifier COL is a predefined constant of type CHAR:

    ORD(COL)=51

The identifier PER is a predefined constant of type CHAR:

    ORD(PER)=47

You can produce the table with the following program:

```
PROGRAM TABLE(OUTPUT);
VAR
     CHARACTER: CHAR;
BEGIN
     WRITELN('     0  1  2  3  4  5  6  7  8  9');
     WRITE(0:3,' ':3);
     FOR CHARACTER := 'A' TO ';' DO
          BEGIN
               IF ORD(CHARACTER) MOD 10 = 0 THEN
               BEGIN
                    WRITELN;
                    WRITE(ORD(CHARACTER):3)
               END;
               WRITE(CHARACTER:3)
          END;
     WRITELN;
     WRITELN
END.
```

This statement will always be true for any two characters C1 and C2:

    (C1 < C2) = (ORD(C1) < (ORD(C2))

Integer

The type integer is a predefined enumeration type. Its values are the finite set $[-2^{48}+1..2^{48}-1]$. Actually, integers in the range $[-2^{59}+1..2^{59}-1]$ can be stored, but the only operations that are executed correctly in this range are: addition, subtraction, taking the absolute value, comparisons, and multiplication and division by certain constants. These constants must be either a power of two or the sum or difference of two powers of two.

The following operators apply to operands of type integer and yield an integer result:

Operator	Description of Operation
+	Integer addition of the values of the two operands.
+	Monadic plus (redundant).
-	Integer subtraction of the value of the right operand from the value of the left operand.
-	Monadic minus; negation.
*	Integer multiplication of the values of the two operands.
DIV	The value of the left operand is divided by the value of the right operand. The result is the quotient truncated (not rounded) to integer.
MOD	A MOD B is defined as: A - ((A DIV B) * B).

Real

The type real is a predefined enumeration type. Its values consist of a finite subset of real numbers in the range $[-10^{322}..-10^{-294}, 0, 10^{-294}..10^{322}]$. A value of type real is represented in the CDC floating point format: the mantissa has 48 bits and a sign and the exponent has 11 bits. Therefore, there are at least 14 significant decimal digits.

Real data types are ordered according to the natural ordering of real numbers.

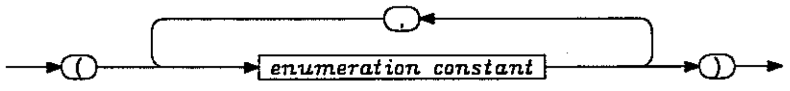
The following operators apply to operands of type real:

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

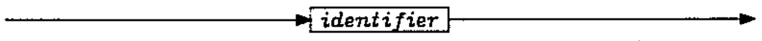
Scalar

The type scalar is defined by listing all the enumeration constants (all the possible values) in increasing order as a list of identifiers.

scalar type



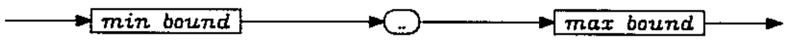
enumeration constant



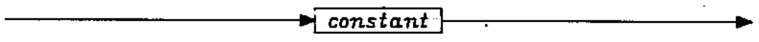
Subrange

An enumeration type can also be defined as a subrange of another enumeration type by specifying its minimum and maximum values separated by a double period (..).

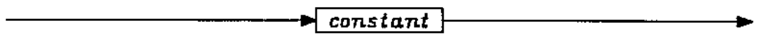
subrange type



min bound



max bound



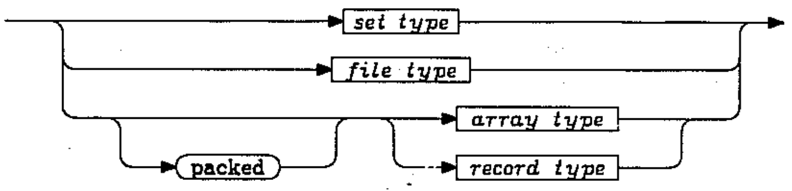
The min bound must not exceed the max bound and they must be of the same enumeration type.

Structured Types

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

There are four kinds of structured types: array, file, record, and set.

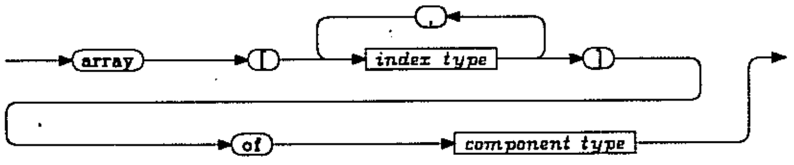
structured type



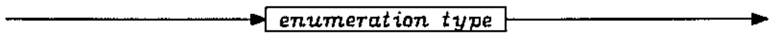
Array

An array consists of a fixed number of components that all have the same type. The number of components is specified by an enumeration type, which is called the index type.

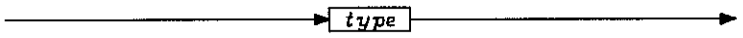
array type



index type



component type



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

A shorthand notation for the type

    ARRAY[T1] OF ARRAY[T2] OF T3

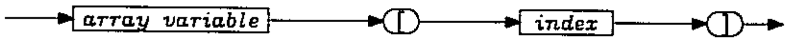
is the notation

    ARRAY[T1,T2] OF T3

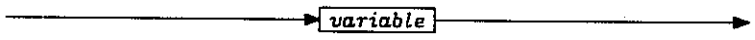
This is called a multi-dimensional array. The number of index types is called the dimension of the array. The array with index type T2 is called the innermost array.

You can use arrays either whole or component wise. A whole array is selected by its array variable. A component of an array is selected by the array variable followed by an index enclosed in brackets. The total number of index expressions must not exceed the dimension of the array. Furthermore, the value of each index expression must be an enumeration type that is compatible with the corresponding index type.

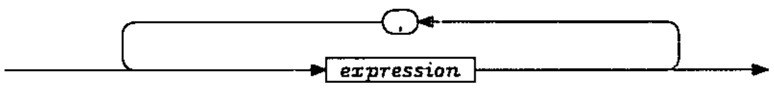
indexed variable



array variable



index



The notations NAME[A1][A2] and NAME[A1,A2] can be used interchangeably.

The following are examples of array declarations:

```
    TYPE
         HOURS = 8..16;
         MATRIX = ARRAY[1..N,1..N] OF REAL; (* N IS AN INTEGER CONSTANT *)
         COUNTER = ARRAY['A'..'Z'] OF INTEGER;
         NAMEOFDAY = ARRAY[DAYS] OF ALFA;
         OCCUPIED = ARRAY[DAYS,HOURS] OF BOOLEAN;
    VAR
         A,B,C : MATRIX;
```

The following statements show array denotations:

```
    A := B; (* THE ENTIRE MATRIX B IS COPIED INTO A *)
    C[I] := A[I]; (* ONE ROW OF A IS COPIED INTO THE CORRESPONDING ROW IN C *)
    C[I,J] := A[K,L]; (* ONE COMPONENT OF A IS COPIED INTO ONE COMPONENT OF C *)
    OCCUPIED[WEDNESDAY,9] := TRUE;
    OCCUPIED[FRIDAY,15] := FALSE;
```

The following statements initialize B to the identity matrix:

```
    FOR I := 1 TO N DO
         BEGIN
              FOR J := 1 TO N DO
                   B[I,J] := 0;
              B[I,I] := 1
         END;
```

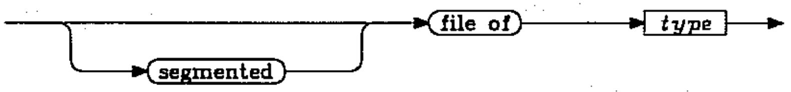
These statements provide an alternate way of initializing B to the identity matrix:

```
    FOR J := 1 TO N DO
         B[1][J] := 0;

    FOR I := 2 TO N DO
         B[I] := B[1];

    FOR I := 1 TO N DO
         B[I,I] := 1;
```

File

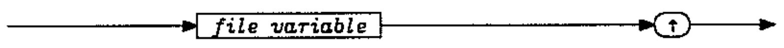
The declaration of a file variable introduces a file buffer to the component type. The file buffer is denoted by the file variable followed by an arrow (↑).

file type

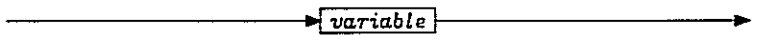


The file buffer can be considered a window through which components can be inspected (read) or new components can be appended (written). A file position is implicitly associated with this window (the file buffer). The window is moved by certain file operations. It is, however, not possible to alternate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

file buffer



file variable



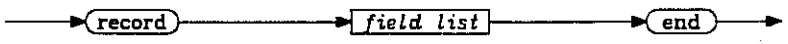
The sequential processing and the existence of a file buffer suggests that files are associated with secondary storage and peripherals. Exactly how the components are allocated varies. Usually only a few components are present in primary storage at a time and only the component denoted by the file buffer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

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

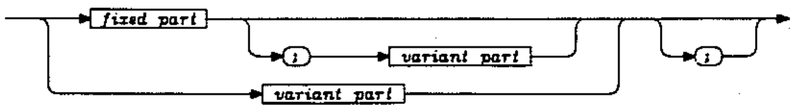
Record

A record consists of a fixed number of components called fields. A field identifier and type must be specified for each field.

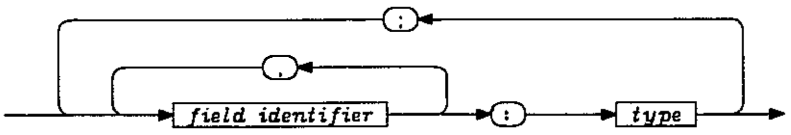
record type



field list

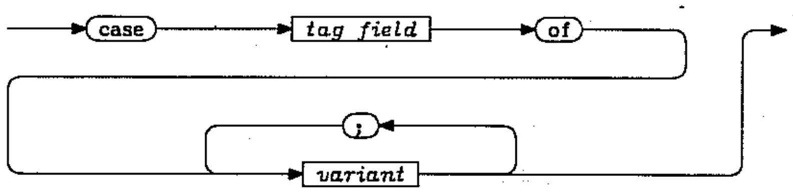


fixed part

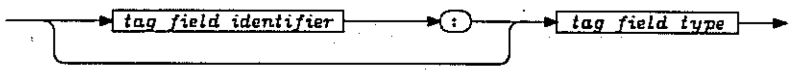


A record can be divided into a fixed part and a variant part, either or both of these parts may be empty.

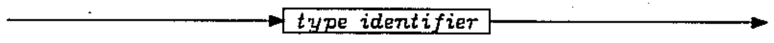
variant part



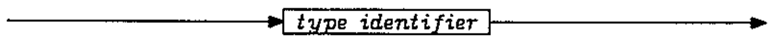
tag field



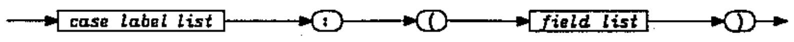
tag field identifier



tag field type



variant



A field list can have a number of variants. In this case, you can designate a tag field that contains a value that indicates which variant is assumed by the field list at a given time. The tag field may be empty.

The tag field type must be an enumeration type. Each variant must be labeled with one or more constants whose type is compatible with the tag field type. All labels must be distinct.

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

The following lines show examples of record types:

```
    TYPE
         CARDTYPE = (NORMAL,WILD);
         COMPLEX = RECORD
                        R,I : REAL
                   END;
         DATE =    RECORD
                        ORDINAL : 1..31;
                        DAY : DAYS;
                        MONTH : MONTHS;
                        YEAR : 1900..2000
                   END;
         PLAYINGCARD =  RECORD
                             CASE T : CARDTYPE OF
                                  NORMAL : (SUIT : SUITS;
                                  RANK : 2..14);
                                  WILD : (FACE : (BLANK,JOKER))
                        END;
```

If you assume the following declarations,

```
    VAR
         S,X,Y : COMPLEX;
         HAND : ARRAY[1..13] OF PLAYINGCARD;
```

then the following are examples of record denotations:

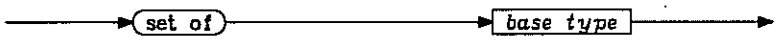
```
    S.R := X.R + Y.R; (* THE REAL COMPONENT OF S BECOMES THE SUM OF THE REAL PARTS OF X AND Y *)
    HAND[1].T := NORMAL;
    HAND[1].SUIT := CLUB;
    HAND[1].RANK := 8;
    HAND[2].T := WILD;
    HAND[2].FACE := JOKER;
```

Note that the tag field is used as any other field is.

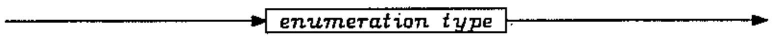
Set

A set type consists of the set of all subsets of some enumeration type.

set type

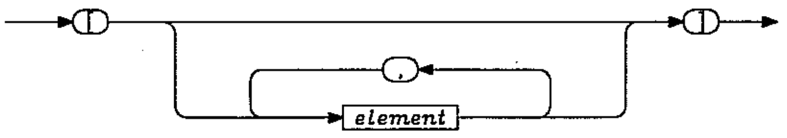


base type

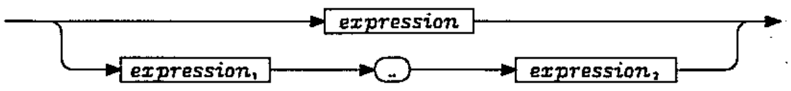


The ordinal number of the largest element must not exceed 58 and the ordinal number of the smallest element must not be negative. It follows that a set type can contain at most 59 elements.

set value



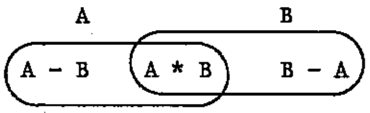
element



A set value denotes a set consisting of the expression values. The form [M..N] denotes the set of all elements i of the base type such that M <= I <= N. If M > N then [M..N] denotes the empty set. The set expressions must all be of compatible enumeration types. The empty set is denoted [ ] and is compatible with any set type.

An example of a set type, assume A and B are of type T and T is a set type, then the following expression is true:

```
    (A - B) + (B - A) = A + B - A * B
```



If you assume the following declarations,

```
    TYPE
         WORKINGDAYS = SET OF DAYS;
         CHARACTERS = SET OF 'A'..'+';
    VAR
         WORKINGDAY : WORKINGDAYS;
         LETTERS, DIGITS, FIRST, FOLLOWING: CHARACTERS;
         LAZY : BOOLEAN;
```

then the following lines are examples of applications of set and set operators:

```
    WORKINGDAY := [MONDAY..FRIDAY];
    LAZY := NOT(SATURDAY IN WORKINGDAY);
    LETTERS := ['A'..'Z'];
    DIGITS := ['0'..'9'];
    FIRST := LETTERS;
    FOLLOWING := FIRST + DIGITS + ['+'];
```

The following relations are all true:

```
    FIRST * DIGITS = [ ]
    FOLLOWING - (DIGITS + ['+']) = LETTERS
    FOLLOWING * FIRST = LETTERS
    ORD([MONDAY, TUESDAY, THURSDAY]) = 1 + 2 + 8
```

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

Operator	Description of Operation
+	The result is the union of the two operand sets.
*	The result is the intersection of the two operand sets.
-	The result is the set difference of the two operand sets (the elements that belong to the left operand, but not to the right operand).

The following operators take two operands of compatible set type and produce a Boolean result.

Operator	Description of Operation
=	The result is true if the left operand is equal to the right operand.
<>	The result is true if the left operand is not equal to the right operand.
<=	The result is true if the left operand is included in the right operand and false if it is not.
>=	The result is true if the right operand is included in the left operand and false if it is not.

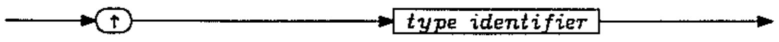
The following operator takes two operands and produces a Boolean result. The right operand is a set type. The left operand must be of an enumeration type compatible with the base type of the right operand.

Operator	Description of Operation
IN	The result is true if the left operand is a member of the set specified as the right operand.

Pointer Types

Pointers are used for constructing dynamic data structures.

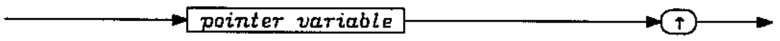
pointer type



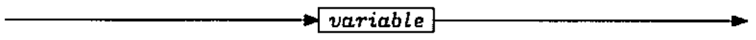
The type identifier cannot denote a type containing a file type. The type identifier may be defined textually after the pointer type.

The value of pointer variable is either nil or a reference to a variable of the specified type. The pointer nil belongs to every pointer type; it points to no variable at all. The variable referenced by a pointer is denoted by the pointer variable followed by an arrow (↑).

referenced variable



pointer variable



For example, a list structure can be declared as follows:

```
TYPE
     PLIST = ↑LIST;
     LIST =    RECORD
                    INF : . . . ;
                    NEXT : PLIST
               END;
VAR
     HEAD : PLIST;
```

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

```
NEW(HEAD);
HEAD↑.INF := . . . ;
NEW(HEAD↑.NEXT);
HEAD↑.NEXT↑.INF := . . . ;
HEAD↑.NEXT↑.NEXT := NIL;
```

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

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

The following operators apply to operands of pointer type and yield a Boolean result:

<> The operands are evaluated, then the resulting values are compared. The outcome is true if the resulting values are not equal and false if they are equal.

= The operands are evaluated, then the resulting values are compared. The outcome is true if the resulting values are equal and false if they are not.

Type Compatibility

Two operands must be compatible in type. In general,

Two types are compatible if they are the same type.

A subrange type is compatible with the type it is a subrange of.

Two subrange types of the same type are compatible.

Two string types are compatible if they have the same length.

Two set types are compatible if their base types are compatible.

The type of the empty set [ ] is compatible with any set type.

The type of the pointer value nil is compatible with any pointer type.

The type INTEGER and any subrange type of INTEGER are compatible with the type REAL except in the following cases:

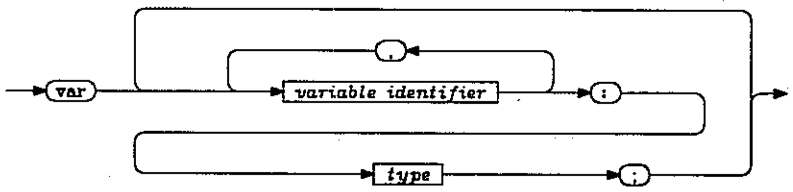
An operand of type REAL cannot be assigned to a variable or function identifier of type INTEGER.

An actual parameter of type REAL cannot be passed to a formal parameter of type INTEGER.

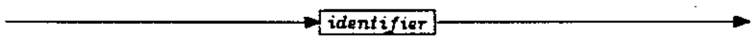
VAR SECTION

The variable declaration section defines the name and type of a variable. Each variable in the statements part must be declared in the variable declaration section prior to its use.

variable declaration part



variable identifier



You can declare several variables of the same type in a single list of identifiers followed by the type.

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

The following is an example of both a type and a variable declaration section:

```
TYPE
    WEEKEND = FRIDAY..SUNDAY;
VAR
    GOODDAY : WEEKEND;
```

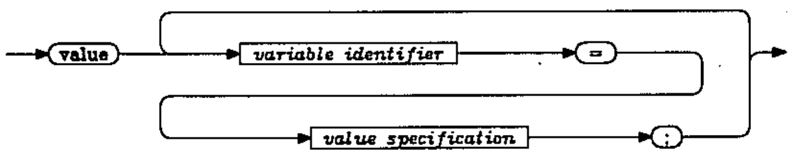
If the variable GOODDAY has the value FRIDAY, then the following relations are true:

```
PRED(GOODDAY) = THURSDAY
SUCC(THURSDAY) = GOODDAY
```

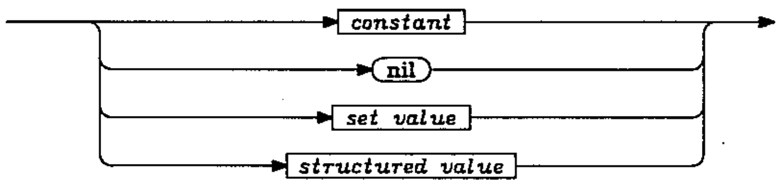
VALUE SECTION

The value declaration section initializes the variables declared in the statements part.

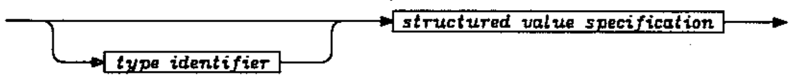
value part



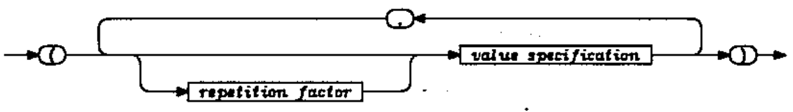
value specification



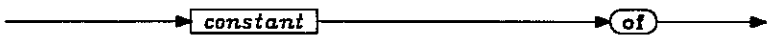
structured value



structured value specification



repetition factor



A variable of a simple type can be initialized with a constant of the same type.

A variable of pointer type can only be initialized with nil.

A variable of set type can be initialized with a set value.

A variable of array or record type can be initialized with a structured value.

A structured value consists of a number of component values, one for each component of the structured type. Each component value must be of the same type as the corresponding component type. If the component type is simple, pointer, or set type, the corresponding component value must follow the rules just given. If a component type is itself an array or record type, the corresponding component value must be a structured value (this rule is used recursively). A multi-dimensional array is considered to be an array of arrays.

A type identifier can be present in a structured value. If it is present, it must denote the same type as the type of the variable being initialized.

The type identifier can be omitted, in which case the rules just given apply.

The following is an example of CONST, TYPE, VAR, and VALUE section declarations:

```
    CONST
         N = 5;
         SIZE = 3;
    TYPE
         VECTOR = ARRAY[1..N] OF INTEGER;
         NAME = PACKED ARRAY[1..8] OF CHAR;
         NODE = RECORD
                   ID : NAME;
                   NEXT = ↑NODE;
                END;
         MATRIX = ARRAY[1..SIZE, 1..SIZE] OF INTEGER;
         DOUBLEVECTOR = ARRAY[1..2] OF VECTOR;
    VAR
         X,Y :VECTOR;
         P,S : NAME;
         N : NODE;
         M1,M2 : MATRIX;
         D : DOUBLEVECTOR;
         I : INTEGER;
    VALUE
         X = VECTOR(1,1,2,2,3);
         Y = (N OF 0);
         P = ('PETER    ');
         S = ('J','O','H','N',4 OF ' ');
         N = NODE('DUMMY   ',NIL);
         M1 = MATRIX((2,3,5),(7,9,13),(17,19,23));
         M2 = ((3 OF 0),(3 OF 1),(1,2,3));
         D = DOUBLEVECTOR(2 OF VECTOR(N OF 0));
         I = 7;
```

Repetition factors can be used to initialize many array elements with the same value. The constant of a repetition factor must be of integer type.

Packed variables can be initialized with a string literal.

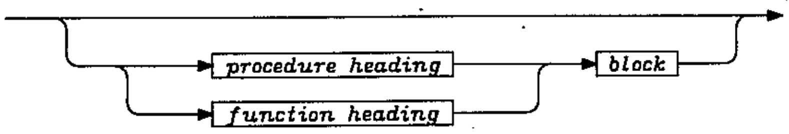
A variable of string type can also be initialized with a string literal.

A record with a variant part can be initialized; the tag field value determines which variant is followed. Even if the tag field has no field identifier the tag field value must be specified to select a variant.

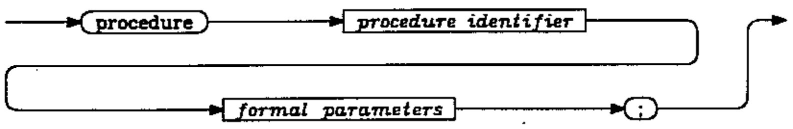
ROUTINES SECTION

The routine declaration section defines a block of statements that can be executed by a procedure of function call within the statements part. The block of statements are bounded by a BEGIN and END.

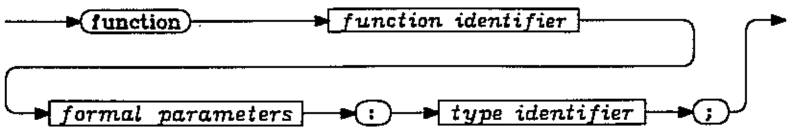
routine declaration



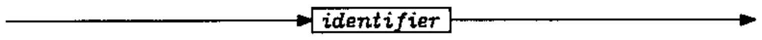
procedure heading



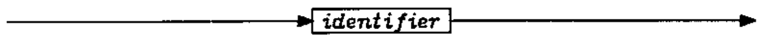
function heading



procedure identifier



function identifier



Routine is a generic term for procedures and functions. The difference between a procedure and function is that while both are subprograms, a function returns a result value and, therefore, can be used in an expression.

A value part is only allowed in the declaration of a program and not within routines.

Routines can be called recursively. At most, ten levels of routines can be declared inside each other. At runtime, however, dynamic routine calls can be nested to any level.

The type in the function heading is the function type, it specifies the range of the function. The function type must be a simple or a pointer type.

The following are examples of routine headings:

```
    FUNCTION MYOWNSQRT(X: REAL): REAL;
    FUNCTION ZERO(LOWER, UPPER: REAL; FUNCTION F(X: REAL): REAL): REAL;
    PROCEDURE INSERT(ELEMENT: COMPONENTTYPE);
    PROCEDURE UPDATE(VAR ELEMENT: COMPONENTTYPE);
```

You can change the current value of a function within the statements that make up the function by writing the function name as the left hand side of an assignment. The value return by the function is the dynamically last value assigned to it.

Formal Parameters

Formal parameters specify the interface between the block and its surroundings. Each formal parameter is given its kind and its formal name. There are four parameter kinds: variable, value, procedure, and function. The kind value is assumed if nothing else is specified.

Formal parameters are denoted by formal names in the block of a routine. A formal parameter of kind variable denotes a variable of the specified name and type. The denoted variable is the actual parameter. A formal parameter of kind value can be used as a local variable of the specified name and type; its initial value is the value of the actual parameter. A formal parameter of kind procedure or function can be used as if it is a locally declared routine.

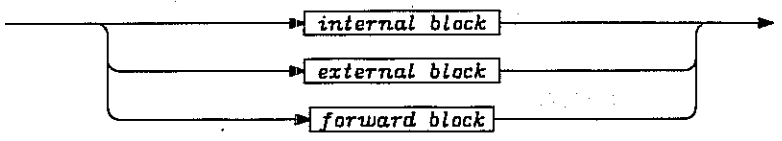
The reserved word DYNAMIC can only precede an array type.

Further differences between the four parameter kinds are given under the heading Actual Parameters in section 4.

Blocks

A block is a group of statements that is bound by a begin and end. There are three classes of blocks: forward, external, and internal.

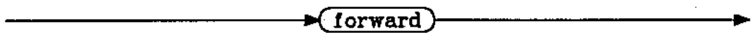
block



Forward Block

The scope rules at the end of this section state that a routine must be declared before it is used. The declaration of a block as a forward block is an announcement of a routine declaration that will be given textually later. When the routine declaration is given, the formal parameters are omitted in the heading.

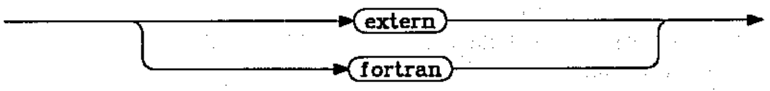
forward block



External Block

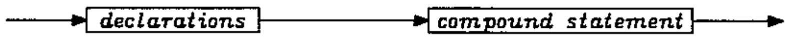
The declaration of a block as an external block is an announcement of a separately compiled routine. The linking of external routines is done by the NOS operating system. Refer to section 5 for a more detailed description of the NOS interface.

external block

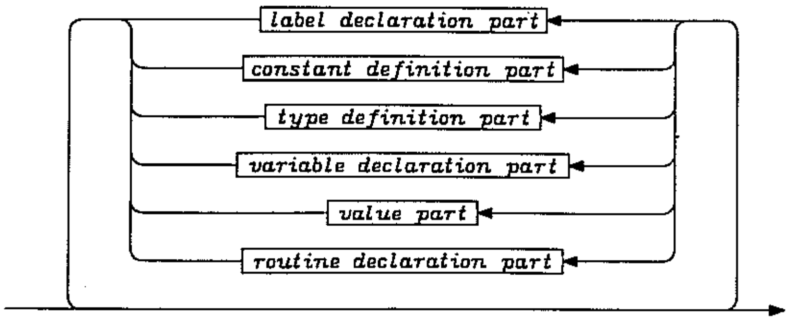


Internal Block

internal block



declarations



Scope Rules

A scope is one of the following:

A field list (excluding inner scopes)

A routine heading

A block (excluding inner scopes)

A name can be declared once in each scope only. All names must be declared before they are used. If a name is declared both in a scope and in an inner scope, it is always the inner declaration that is effective in the inner scope.

Generally, the declaration of a name is effective in the rest of the block where it is declared, however, details for each kind of name are given below.

Constant identifier, type identifier, variable identifier, enumeration constant, label, and routine identifier:

The declaration of these identifiers is effective in the rest of the block.

Pointer type identifier:

In the definition of a pointer type, the type identifier on the right hand side of the arrow (↑) can be defined textually after the pointer type definition.

Field identifier:

The declaration of a field identifier is effective in the rest of the block, but the field identifier can be used in RECORD variables and WITH statements only.

Routine parameter name:

The name is effective in the rest of the block.

Program identifier:

The program identifier has no significance within the program.

STATEMENT PART 4

This section describes the statements that manipulate defined and declared data items.

A collection of statements can be grouped together as a compound statement by enclosing them within BEGIN and END.

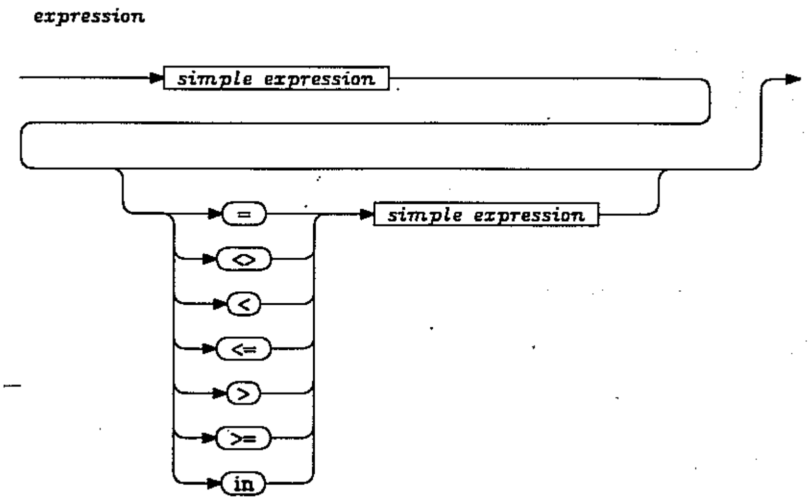
The statement(s) in the statement part are executed sequentially in the same order as they appear.

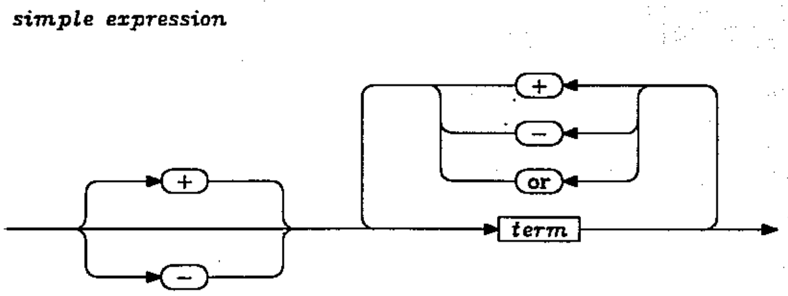
EXPRESSIONS

Expressions are often a fundamental part of a statement. An expression defines a rule of computation for obtaining a value by application of operators to operands. An expression is evaluated from left to right using the following precedence rules (highest to lowest precedence):

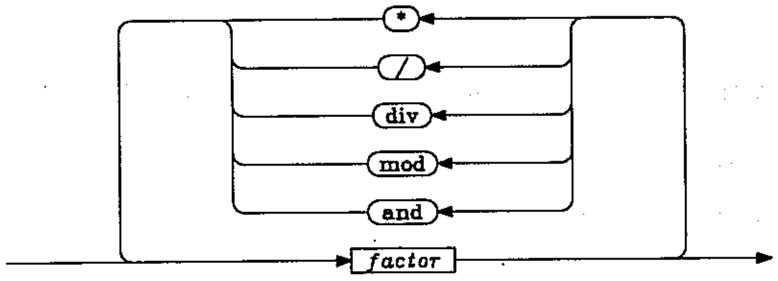
```
NOT
*, /, DIV, MOD, AND
+, -, OR
=, <>, <, <=, >, >=, IN
```

Expressions are written in infix notation.

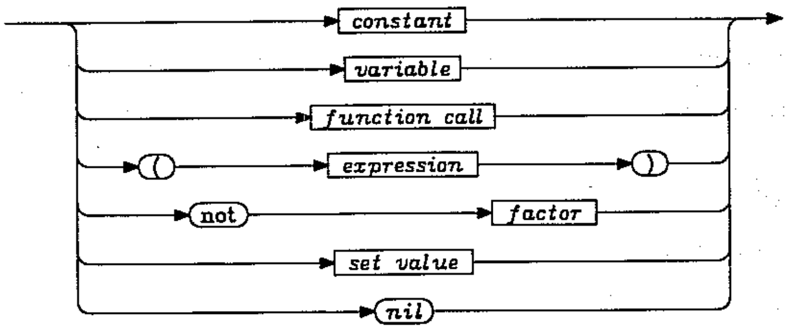




term



factor



The following relations are true:

```
2 * 3 - 4 * 5 = (2 * 3) - (4 * 5)
15 DIV 4 * 4 = (15 DIV 4) * 4
80 / 5 / 3 = (80 / 5) / 3
4 + 2 * 3 = 4 + (2 * 3)
```

For any B1, B2, B3 of type Boolean, the following relation is true:

```
B1 OR NOT B2 AND B3 = B1 OR ((NOT B2) AND B3)
```

The following is an ambiguous expression:

```
0 < X AND X < 10
```

It should be written as

```
(0 < X) AND (X < 10)
```

The following two statements are different:

```
IF (I <= N) AND (TABLE[I] = KEY) THEN S;
IF I <= N THEN IF TABLE[I] = KEY THEN S;
```

All factors in an expression are evaluated and therefore should be defined.

If an expression contains a function whose evaluation causes side effects on other factors in the expression, the left to right evaluation does not always hold; such side effects should be avoided.

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

Operator(s)	Left Operand	Right Operand	Result
+, -, *	INTEGER	INTEGER	INTEGER
	INTEGER	REAL	REAL
	REAL	INTEGER	REAL
	REAL	REAL	REAL
	any type compatible with T	any type compatible with T	set type T
DIV, MOD, /	INTEGER	INTEGER	INTEGER
	INTEGER	INTEGER	REAL
	INTEGER	REAL	REAL
	REAL	INTEGER	REAL
	REAL	REAL	REAL
OR, AND, <, >	BOOLEAN	BOOLEAN	BOOLEAN
	any string type T	any type compatible with T	BOOLEAN
	any simple type T	any type compatible with T	BOOLEAN
<=, >=, =, <>	any string type T	any type compatible with T	BOOLEAN
	any simple type T	any type compatible with T	BOOLEAN
	any set type T	any type compatible with T	BOOLEAN
IN	any enumeration type compatible with T	SET OF T	BOOLEAN

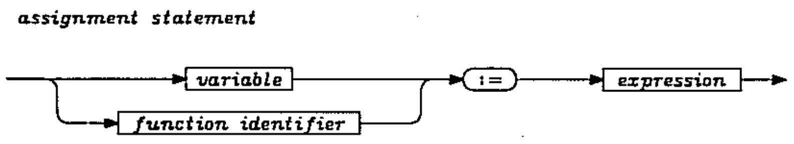
The following table gives all valid combinations of monadic operators and operand types:

Operator(s)	Operand	Result
+, -	INTEGER	INTEGER
	REAL	REAL
NOT	BOOLEAN	BOOLEAN

During evaluation of an expression, intermediate results are kept in a fixed number of registers. If the number of intermediate results exceeds the capacity of the registers, the expression cannot be translated and the compiler issues the error message: expression too complicated. To remedy this, you must either rewrite the expression with a less complicated parenthesis structure or be split into two or more expressions.

ASSIGNMENT STATEMENT

The assignment statement replaces the current value of a variable or function with the value of an expression.



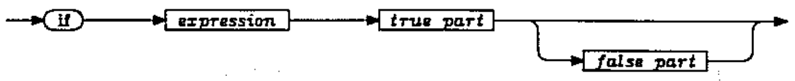
The variable or function and expression must be of compatible types. Assignments can be made to variables of any type except file variables (assignment to the file buffer of a file is legal).

An assignment can be made to a function identifier within its own statement block. The value returned by the function is the dynamically last value that was assigned to it.

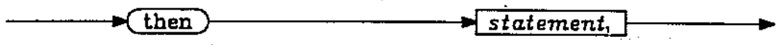
IF STATEMENT

The IF statement defines two paths that can be taken during program execution. The path that is taken depends upon the result of the Boolean expression contained in the statement.

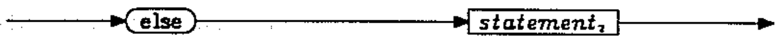
if statement



true part



false part



Statement 1 will only be executed if the value of the expression is true. The statement following statement 1, in this case statement 2, will be executed if the value of the expression is false.

The ambiguity that arises from the construction

```
    IF E1 THEN IF E2 THEN S1 ELSE S2
```

can be resolved by writing it as

```
    IF E1 THEN
        BEGIN
            IF E2 THEN S1
            ELSE S2
        END
```

The following are examples of IF statements:

```
    IF DAY = SUNDAY THEN NEXT := MONDAY
        ELSE NEXT := SUCC(DAY)

    IF X > Y THEN
        BEGIN
            MIN := Y;
            MAX := X
        END
    ELSE
        BEGIN
            MIN := X;
            MAX := Y
        END;
```

The following IF statements are not equivalent:

```
    IF (I <= N) AND (TABLE[I] = KEY) THEN S;

    IF I <= N THEN IF TABLE[I] = KEY THEN S;
```

In the case where I > N, the first statement will evaluate TABLE[I] = KEY and probably cause an index error.

A frequent misuse of the IF statement is the following:

```
IF A = B THEN FOUND := TRUE
ELSE FOUND := FALSE;
```

A much simpler statement is:

```
FOUND := A = B;
```

The following IF statement:

```
IF B THEN S1 ELSE S2;
```

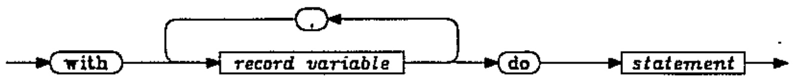
is equivalent to:

```
CASE B OF
    TRUE : S1;
    FALSE : S2
END;
```

WITH STATEMENT

The WITH statement facilitates manipulation of record components.

with statement



The fields of the record variable(s) within the statement can be denoted by writing their field identifiers without preceding them with the denotation of the entire record variable.

The following is an example of a WITH statement:

```
WITH HAND[1] DO
    BEGIN
        T := NORMAL;
        SUIT := CLUB;
        RANK := 8
    END;
```

You can nest WITH statements, as in the example

```
WITH V1 DO S1
    WITH V2 DO S1
        .
        .
        .
            WITH Vn DO S1;
```

A shorter way to write the same nested WITH statement is the following:

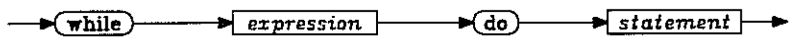
```
WITH V1, V2, V3, ..., Vn DO S1;
```

The record variable selects a record; this selection cannot be changed in the statement. If the record variable has array indexes or pointers, changes to them within the WITH statement will not affect the selection.

WHILE STATEMENT

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

while statement



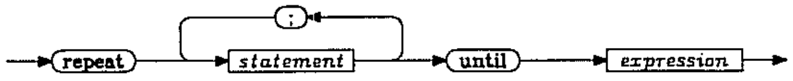
The expression must yield a result of type Boolean. The statement following DO will be executed zero or more times. The expression is evaluated before each execution.

The WHILE statement continues until the evaluation of the expression yields a false result. If the evaluation of the expression is false before execution of the WHILE statement, the statement following DO is not executed.

REPEAT STATEMENT

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

repeat statement

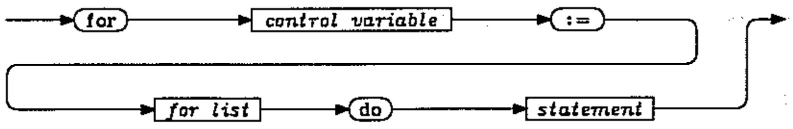


The expression must yield a result of type Boolean. The sequence of statements between the symbols REPEAT and UNTIL are executed one or more times. Every time the sequence is executed, the expression is evaluated. When the resulting value becomes true the REPEAT statement is completed.

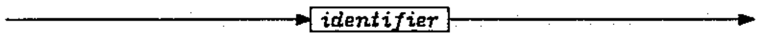
FOR STATEMENT

A value of an enumeration type can be used to execute a statement repeatedly with a consecutive sequence of enumeration values.

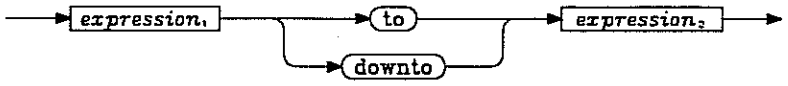
for statement



control variable



for list



The two expressions must be of the same enumeration type and the type of the control variable must be compatible with this type. The control variable must be declared in the same block as the FOR statement. Assignment to the control variable is not allowed within the statement.

The FOR list expresses the size of the interval and the order of progression. The control variable can either be incremented (in steps of 1) from expression 1 TO expression 2, or decremented (in steps by 1) from expression 1 DOWNTO expression 2. The expressions are only evaluated once before the repetition. If expression 1 is greater than expression 2 and increment is specified (TO) then the statement is not executed at all. If expression 1 is less than expression 2 and decrement is specified (DOWNTO), the statement is not executed.

The value of the control variable is undefined after the completion of the FOR statement.

The expression in the FOR list may contain variables. The selection of these variables cannot be changed in the statement. If the variables have array indexes or pointers, changes to them in the FOR statement will not affect the selection.

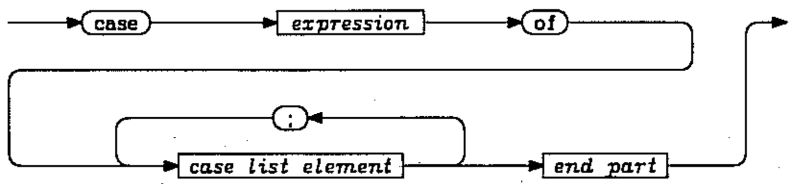
The value of I = J in the following FOR statement is undefined:

```
FOR I := 1 TO N DO
    .
    .
    .
IF I = J THEN
    .
    .
    .
```

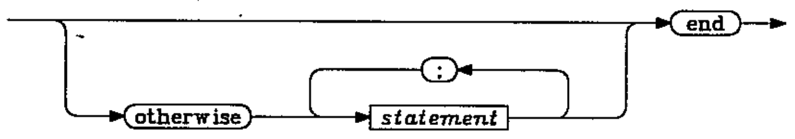
CASE STATEMENT

The CASE statement specifies that the value of an enumeration type is to be used to select one of several statements for execution.

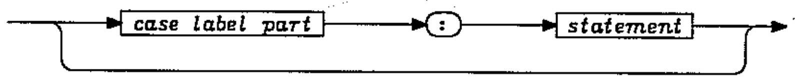
case statement



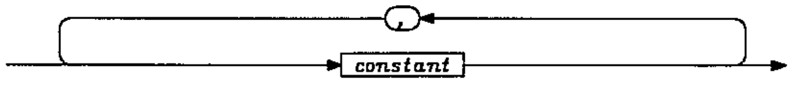
end part



case list element



case label part



A CASE list element is a statement labeled by one or more constants. These constants must all be of the same type as the expression. All labels (constants) in a CASE statement must be distinct. The statement labeled by the current value of the expression is selected for execution. If no such label is present, the statements following OTHERWISE are selected for execution. If OTHERWISE is not included in the CASE statement, the effect of the CASE statement is defined as follows: if the compiler option T+ (see section 5) is specified, the runtime error message INDEX OR CASE EXPR OUT OF RANGE is given and program execution terminated; if the compiler option T- is specified, no statement is selected for execution. Upon completion of the selected statement, the CASE statement is also completed.

If you assume the following declarations:

```
    VAR
        MONTH : MONTHS;
        SUIT : SUITS;
        SEASON : SEASONS;
        COLOR : COLORS;
```

then the following are valid examples of a CASE statement:

```
    CASE MONTH OF
        DECEMBER,JANUARY,FEBRUARY : SEASON := WINTER;
        MARCH,APRIL,MAY : SEASON := SPRING;
        JUNE,JULY,AUGUST : SEASON := SUMMER;
        SEPTEMBER,OCTOBER,NOVEMBER : SEASON := AUTUMN;
    END;

    CASE SUIT OF
        CLUB, SPADE : COLOR := BLACK
    OTHERWISE
        COLOR := RED;
    END;
```

The CASE statement is translated into a jump table. The size of this table is limited; no two labels L1 and L2 can be chosen so that ABS(ORD(L1) - ORD(L2)) > 1000.

JUMPS

Complicated control structures can be constructed using jumps. A jump is a means of transferring control to an arbitary place in a program. It should be noted that any control structure can be constructed using the WHILE and IF statements only (and auxiliary boolean variables). Furthermore, it is not considered good programming style to use jumps.

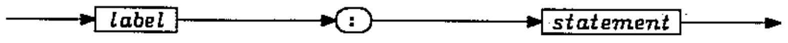
A jump consists of a destination (label) and transfer of control (the GOTO statement).

LABELED STATEMENT

A labeled statement is the destination of a GOTO statement.

A labeled statement is defined by prefixing a statement with a label and a colon (:). The statement after the colon cannot be a labeled statement.

labelled statement

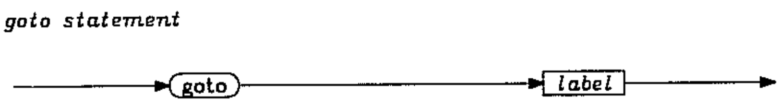


Labels follow the same rules of scope that other declared quantities do: the innermost declared label will be effective in the case of nested routines that use the same label.

All labels must be declared in the LABEL section of the declaraction and definition part of the program or routine in which it is defined. A label that is referenced by a GOTO statement can be used only once in the statements part.

GOTO STATEMENT

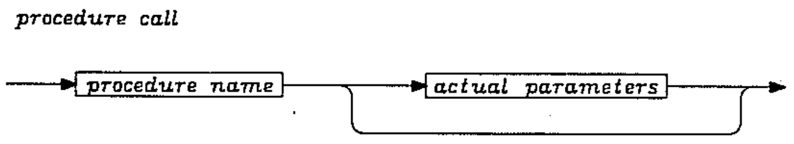
Control is transferred to a labeled statement by a GOTO statement.

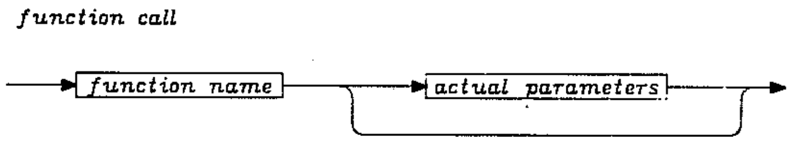


The innermost declared label will be effective in the case where nested routines use the same label. The result of jumping into an inner statement of an IF, WHILE, REPEAT, WITH, FOR, or CASE statement is undefined.

ROUTINES

Routine is a generic term for procedures and functions. The difference between procedures and functions is that while both are subroutines, a function returns a result, therefore, a function name can be a part of an expression.





A routine call binds actual parameters to formal parameters, allocates local variables, and executes the block of statements that make up the routine. When the block is completed, local variables are deallocated and execution is resumed with the statement immediately after the routine call.

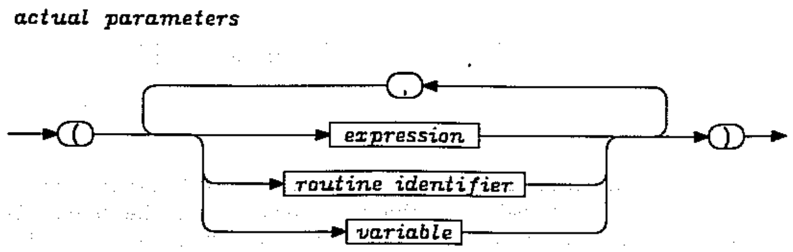
The variables of a routine are associated with a specific call; they exist from the routine call until the block of statements is completed. When a routine is called recursively, several versions of the variables exist simultaneously, one for each uncompleted call.

The scope rules of Pascal lead to a conflict in a situation where two routines call each other. (Which one should be declared first?) The conflict can be removed by substituting the identifier FORWARD for the body of the first routine and postponing the specification of the routine body. For example,

```
    FUNCTION G(X : REAL) : REAL;
    FORWARD;
    FUNCTION F(X : REAL) : REAL;
        .
        .
        .
        BEGIN
            .
            .
            .
            G(X);
            .
            .
            .
        END;
        .
        .
        .
    FUNCTION G;
        BEGIN
            .
            .
            .
            F(X);
            .
            .
            .
        END;
```

ACTUAL PARAMETERS

There must be an actual parameter for each formal parameter.



The binding of an actual parameter to a formal parameter depends on the parameter kind. There are four parameter kinds: value, variable, function, and procedure.

Binding a Value

The type of the actual parameter must be compatible with the type of the formal parameter. The value of the actual parameter is evaluated, then this value becomes the initial value of the formal parameter. Assignments to the formal parameter within the block do not affect the actual parameter (call by value).

Binding a Variable

The type of the actual and formal parameter must be the same. The actual parameter must be a variable. The value of this variable becomes the initial value of the formal parameter. Changes to the value of the formal parameter within the block affects the actual parameter directly. The actual parameter selects a variable; this selection cannot be changed in the block. If the variable contains array indexes or pointers, changes to them do not affect the selection (call by reference).

A component of a packed structure can only be used as an actual VAR parameter if it occupies a whole multiple of 60-bit machine words.

An element or a field of a packed variable cannot be an actual VAR parameter. The whole packed variable can, however, be an actual VAR parameter.

Binding a Procedure or Function

The parameter list of the actual and formal parameters must match. Two parameter lists match if they have the same number of parameters and if the parameters match pairwise. Two parameters match if either of the following conditions are true:

Value and variable parameter types are the same

Routine formal parameters match

Note the following guidelines for choosing between value and variable specification of parameters:

If a parameter is not used to transfer a result from the routine, value specification is generally preferred. But for each value parameter is allocated a storage area for holding the entire value. The value of the actual parameter is transferred to this area. In case of a large structure type, value specification can therefore be very inefficient.

DYNAMIC PARAMETERS

If the formal parameter is specified as DYNAMIC, it must be an array type T. The actual parameter is then required to:

Have the same dimension as T

Have index types that are pairwise compatible with the index types of T

Have the same element type as T

Note that the second condition implies that actual parameters with different index types (for example, size) can be passed as actual parameters to a routine.

Dynamic parameters can only be manipulated componentwise. This means that assignments and comparisons of dynamic parameters must be done componentwise. Furthermore PACK and UNPACK can only be applied to components of a dynamic parameter. Dynamic parameters can be passed as parameters to other routines.

The following is an example of a function with dynamic parameters:

```
    TYPE
        LIST = ARRAY[1..100] OF INTEGER;
    FUNCTION
        MAXIMUM(VAR L : DYNAMIC LIST) : INTEGER;
        (* L IS OF KIND VARIABLE TO SAVE TIME AND SPACE *)
    VAR
        I, MAX : INTEGER;
    BEGIN
        MAX := HIGH(L);
        FOR I := LOW(L) + 1 TO HIGH(L)
            IF MAX < L[I] THEN
                MAX := L[I];
            MAXIMUM := MAX
    END;
```

FILE PARAMETERS

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

If the type of a formal parameter is T or SEGMENTED T, where T is a file type, the actual parameter is allowed to be both of type T or of type SEGMENTED T.

PACKED PARAMETERS

Only the innermost array can be packed when a packed array is passed as a dynamic parameter.

PREDEFINED ROUTINES

The following discussion describes all predefined routines except those that apply to files, which are discussed under the heading Files in this section.

ABS	Takes a single integer argument and returns an integer result that is the absolute value of the argument.
ARCTAN	Takes a single real argument and returns the result of applying the specified mathematical function to the argument.
CARD	Takes a single argument of set type and returns an integer that is the cardinality of the argument (the number of elements in the set).
CHR	Takes a single integer argument and returns a character result that has the ordinal value of the argument. As a consequence, CHR is only defined in the subrange [0..63].
CLOCK	This is a parameterless integer function. It gives the current used CPU-time in milliseconds.
COS	Takes a single real argument and returns the result of applying the specified mathematical function to the argument.
DATE	Takes a single parameter of type alfa and assigns the current date to it in the form: YY/MM/DD. (year/month/day.).

DISPOSE	Releases the variable referenced by P. If the associated type contains variants and NEW(P,C1,. . .,Cn) has been used to allocate the variable, then DISPOSE(P,C1,. . .,Cn) must be used to release the variable.
EOF	Described under Files.
EOLN	Described under Files.
EOS	Described under Files.
EXP	Takes a single real argument and returns the result of applying the specified mathematical function to the argument.
EXPO	Takes a single real argument and returns an integer result that is the exponent of the argument in binary representation.
GET	Described under Files.
GETSEG	Described under Files.
HALT	Takes a single argument of type string and terminates the program (after closing external files) with a CPU abort, places string in the dayfile of the job and produces a dump.
HIGH	Takes an array variable or parameter of index type and returns the max bound of the nth index type of A, $1 \leq N \leq$ dimension of A. HIGH(A) is a shorthand for HIGH (A,1).
LINELIMIT	Described under Files.
LOW	Takes an array variable or parameter of index type and returns the min bound of the nth index type of A, $1 \leq N \leq$ dimension of A. LOW(A) is a shorthand for LOW(A,1).
LN	Takes a single real argument and returns the result of applying the specified mathematical function to the argument.
MESSAGE	Takes a single argument of type string and places it in the dayfile of the job.
NEW	Allocates a new variable of the same type as the argument and assigns a reference to the argument.

In the case where the type associated with P is a record type and the field has variants, the form NEW(P,C1,. . .,Cn) can be used. C1,. . .,Cn is a list of constant selectors used to determine the size of the allocated variable. The size is as if the variable was declared a record type with the field list formed by the following rule of selection: first, the variant corresponding to the selector C1 is selected, then, the field list of this variant is formed by using the selectors C2,. . .,Cn (by a recursive application of this rule), finally, the so-far-formed field list is prefixed by the tag field (if nonempty) and is substituted for the variant part.

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

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

If you assume the following declarations:

```
CONST
    MAXVAL = 50;
TYPE
    PATOM = ↑ATOM;
    ATOM =
RECORD
    NAME : ALFA;
    NUMBER : INTEGER;
    WEIGHT : REAL;
    OCCUPIED : SET OF 1.. MAXVAL;
    BINDINGS : ARRAY[1..MAXVAL] OF PATOM;
    CHARGE : (PLUS,MINUS,NEUTRAL);
    SATURATED : BOOLEAN
    END;
VAR
    A : ATOM;
```

then the following statements give all the names of the atoms to which A is bound:

```
     WITH A DO
          FOR I := 1 TO MAXVAL DO
               IF I IN OCCUPIED THEN
                  WRITELN(I,BINDINDS[I]↑.NAME);
```

If you assume the following declarations:

```
VAR
     P : ↑PLAYINGCARD;
```

then NEW(P,WILD) allocates a variable whose size is as if the variable had been of the type Q defined as

```
TYPE
     Q =  RECORD
               T : CARDTYPE;
               FACE : (BLANK,JOKER)
          END;
```

ODD Takes a single integer argument and returns a Boolean result that is true if the argument is odd and false if the argument is even.

ORD Applies to operands of any enumeration type. Takes a single argument and returns a result that is the number of the argument in the set of values defined by the type of the argument. When applied to a pointer the result is the integer representation of the pointer. When applied to a Boolean value, the result is the following:

```
     ORD(FALSE) = 0

     ORD(TRUE) = 1
```

Ord can also be applied to a subrange type:

```
VAR
     A : INTEGER;
     B : MIN..MAX;
```

In this case, A = B implies ORD(A) = ORD(B).

PACK Packs array values. Assume that A and P are variables of the following types:

```
A: ARRAY [M..N] OF T;
P: PACKED ARRAY [U..V] OF T;
```

When (ORD(N) - ORD(I)) >= (ORD(V) - ORD(U)); M <= I; and the index types of the arrays A and P and the type of I are compatible, then PACK(A,I,P) is equivalent to:

```
K := I;
FOR J := U TO V DO
BEGIN
     P[J] := A[K];
     K := SUCC(K)
END
```

PAGE Described under Files.

PRED Applies to operands of any enumeration type. Takes one argument and returns the predecessor of the argument, which is the same type as the argument. If the argument is the first (smallest) value of the type the result may be undefined.

PUT Described under Files.

PUTSEG Described under Files.

READ Described under Files.

READLN Described under Files.

RESET Described under Files.

REWRITE Described under Files.

ROUND Takes a single real argument and returns a result that is the argument rounded (not truncated) according to standard mathematical conventions.

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

```
TRUNC(1.6) = 1       ROUND(1.6) = 2
TRUNC(-1.6) = -1     ROUND(-1.6) = -2
TRUNC(2.4) = 2       ROUND(2.4) = 2
```

The operators = and <> should be used with great care on real arguments because of round-off errors that often result from the representation of real values, as in the following examples:

```
(1.00000 - 0.00001) = 0.99999      FALSE
SQR(SQRT(2)) = 2                   FALSE
(4.0 * 0.25) = 1                   TRUE
(10000 * 0.0003) = 3               TRUE
(1000000 * 0.000003) = 3           FALSE
```

SIN Takes a single real argument and returns the result of applying the specified mathematical function to the argument.

SQR Takes a single integer argument and returns an integer result that is the square of the argument.

SQRT Takes a single real argument and returns the result of applying the specified mathematical function to the argument.

SUCC Applies to operands of any enumeration type. Takes one argument and returns the successor of the argument, which is the same type as the argument. If the argument is the last (greatest) value of the type the result may be undefined.

TIME Takes a single argument of type alfa and assigns the curent time to it in the form: HH.MM.SS. (hour.minute.seconds.).

TRUNC Takes a single real argument and returns an integer whose sign is the same as the argument and whose absolute value is the greatest among the integers less than or equal to the absolute value of the argument.

Trunc can also be applied to two arguments; the first argument must be of type real, the second argument must be of type integer. TRUNC(X,I) is equal to TRUNC(X * Y) where Y is 2 to the power I.

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

```
TRUNC(1.6) = 1       ROUND(1.6) = 2
TRUNC(-1.6) = -1     ROUND(-1.6) = -2
TRUNC(2.4) = 2       ROUND(2.4) = 2
```

UNDEFINED Takes a single real argument and returns a Boolean result that is true if the argument is out of range or indefinite if a division by 0 was made.

UNPACK Unpacks array values. Assume that A and P are variables of the following types:

```
A: ARRAY [M..N] OF T;
P: PACKED ARRAY [U..V] OF T;
```

When (ORD(N) - ORD(I)) >= (ORD(V) - ORD(U)); M <= I; and the index types of the arrays A and P and the type of I are compatible, then UNPACK(A,I,P) is equivalent to:

```
K := I;
FOR J := U TO V DO
BEGIN
    A[K] := P[J];
    K := SUCC(K)
END;
```

Where J denotes an auxiliary variable that is not used elsewhere in the program.

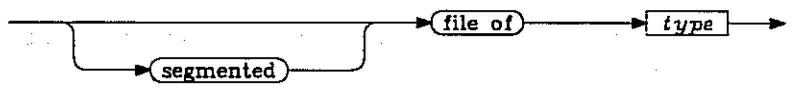
WRITE Described under Files.

WRITELN Described under Files.

FILES

A file is a structure that consists of a sequence of components that are all of the same type.

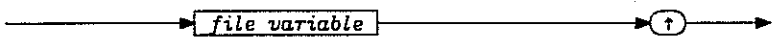
file type



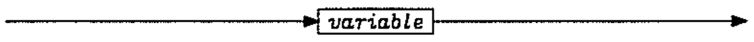
A file type can be defined as the number of components (the length of the file) is not fixed. At any time, only one component of the file is accessible. The other components can be reached by sequencing through the file. A file without any components is said to be empty.

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

file buffer



file variable



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

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

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

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

EOF(F)	Takes a file name as an argument and returns a Boolean true value if the file is positioned at the end-of-file mark and false if it is not.
GET(F)	Advances the position of the file to the next component. The value of the file buffer becomes the content of this component. If no next component exists, EOF(F) becomes true and the value of F↑ is undefined. If EOF(F) is true prior to the execution of GET(F), the call results in the runtime error message: TRIED TO READ PAST EOS/EOF.
PUT(F)	Appends the value of the buffer variable F↑ to the file F. The value of F↑ becomes undefined. If the value of EOF(F) or EOS(F) is false prior to the execution of the PUT(F), the call results in the runtime error message: TRIED TO WRITE WHILE NOT EOS/EOF. Otherwise the value of EOF remains true.

READ(F,X)	A READ statement is exactly equivalent to:
	`X := F↑;`
	`GET(F);`
	X must be of a type compatible with the type of the components in the file F. If F is a textfile, see the description under the heading Textfiles.

RESET(F)	Repositions the file at the start; the file buffer F↑ contains the first component of the file. The file can now be read. If the file is empty, the value of F↑ is undefined and EOF(F) is true.

REWRITE(F) Positions the file at the start for rewriting. The value of F becomes the empty file, F↑ becomes undefined, and EOF(F) becomes true.

WRITE(F,X) A WRITE statement is exactly equivalent to:

```
F↑ := X;
PUT(F);
```

X must be of a type compatible with the type of the components in the file F. If F is a textfile, see the description under the heading textfiles.

By using the B option described in section 5, the size of the main storage area holding part of the file around the current position can be varied. In this way, an exchange of time for space (or vice versa) can be obtained.

TEXTFILES

A file of characters is called a textfile. Accordingly, the predefined type TEXT is defined as:

```
TEXT : FILE OF CHAR;
```

Texts can be subdivided into lines. However, the mark indicating a line boundary is not a character included in the set of char values. The following predefined routines are provided for manipulating this end-of-line mark (EOL). It is assumed that T is a variable of type TEXT.

EOLN(T) The result of this Boolean function is true if T is positioned at an end-of-line mark, and false otherwise. If true, T↑ contains a blank.

READLN(T) Skips to the beginning of the next line of T. Subsequently, T↑ becomes the first character of the next line if any. READLN(T) has the same effect as the following statements:

```
WHILE NOT EOLN(T) DO GET(T);
GET(T);
```

WRITELN(T) Terminates the current line of T; writes an end-of-line mark.

WRITELN may append some extra blanks to the line because of some peculiarities in the representation of end-of-line mark in the NOS operating system.

Two additional predefined routines are provided:

LINELIMIT(T,N) Associates a linecounter with the file T and resets this linecounter to N. The first parameter must be a textfile and the second an integer expression. Each time an end-of-line mark is written onto T the associated linecounter is decremented by 1. The program is terminated if the linecounter reaches zero and the following message is given: LINELIMIT EXCEEDED.

LINELIMIT(OUTPUT,1000) is automatically executed before the program is executed.

PAGE(T) Positions the lineprinter. The argument must be a textfile. PAGE(T) is equivalent to the statements:

```
WRITELN(T);
WRITE(T,'1');
```

The '1' forces the lineprinter to the top of a new page.

A textfile T, subdivided into lines, can be scanned by the following piece of program:

```
RESET(T);
WHILE NOT EOF(T) DO
     BEGIN
          WHILE NOT EOLN(T) DO
               BEGIN
                    READ(T,CH);
                    Q(CH) (* PROCESS SINGLE CHARACTER *)
               END;
          READLN(T);
          R (* PROCESS LINE *)
     END;
```

A textfile T, subdivided into lines with a maximum of N significant characters in each line, can be scanned by the following piece of program:

```
RESET(T);
WHILE NOT EOF(T) DO
     BEGIN
          I := 0;
          WHILE (I < N) > EOLN(T) DO
               BEGIN
                    I := I + 1;
                    READ(T,LINE[I]);
               END;
          READLN(T);
          R (* PROCESS LINE *)
     END;
```

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

Let T denote a textfile and V,V1,. . .,Vn variables of type CHAR, INTEGER, or REAL.

READ(T,V) Reads a sequence of characters from the file T through the file buffer T↑ using GET(T). The first significant character is the character in T↑.

If V is of type char, then READ(T,V) is exactly equivalent to:

```
V := T↑;
GET(T);
```

If V is type integer, a sequence of digits is transformed into a (decimal) value and then assigned to V. Leading blanks and leading end-of-line marks are skipped. The character sequence that follows must be consistent with the syntax for decimal integers given in section 2. If not, execution is terminated and a runtime error message is given. Trailing blanks are skipped (if the file buffer T↑ is left at the first nonblank character after the number or is left at the end-of-line mark).

If V is of type real, a sequence of characters is transformed into a real value and then assigned to V. Leading blanks and leading end-of-line marks are skipped. The character sequence that follows must be consistent with the syntax for real literals given in section 2. If not, execution is terminated and a runtime error message is given. Trailing blanks are skipped (if the file buffer is left at the first nonblank character after the real number or is left at the end-of-line mark).

READ(T,V1,. . .,Vn) is a shorthand notion for:

```
BEGIN
     READ(T,V1);
     READ(T,V2);
     .
     .
     .
     READ(T,Vn)
END;
```

READLN(T,V) is a shorthand notation for:

```
BEGIN
     READ(T,V);
     READLN(T)
END;
```

READLN(T,V1,. . .,Vn) is a shorthand notation for:

```
BEGIN
     READ(T,V1,. . .,Vn);
     READLN(T)
END;
```

The predefined procedure WRITE is extended in a similar way. Let P,P1,. . .,Pn be parameters of the form defined below and T be a textfile.

WRITE(T,P) Transforms the parameter P into a sequence of characters (according to the rules given below). This sequence is written on T.

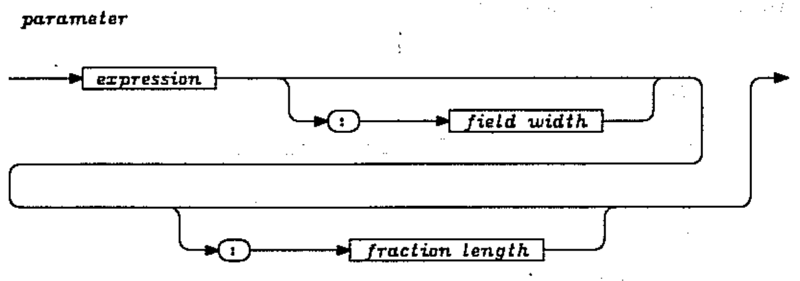
WRITE(T,P1,. . .,Pn) is a shorthand notation for:

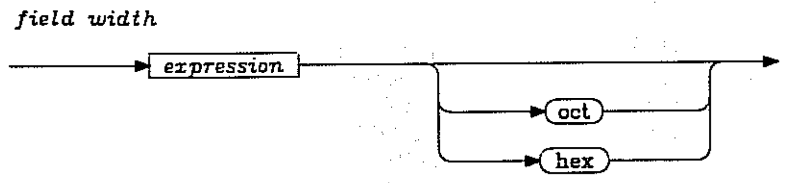
```
BEGIN
     WRITE(T,P1);
     WRITE(T,P2);
     .
     .
     .
     WRITE(T,Pn)
END;
```

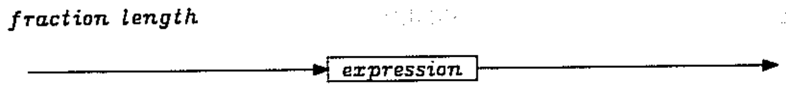
WRITELN(T,P1,. . .,Pn) is a shorthand notation for:

```
BEGIN
     WRITE(T,P1,. . .,Pn);
     WRITELN(T)
END;
```

The parameters in the predefined procedures WRITE and WRITELN must have the following form:







The first expression, which is the value to be written, must be of type: INTEGER, BOOLEAN, CHAR, REAL, or STRING. The fraction length can be given only when the expression is of type REAL. The field width indicates the minimum number of characters to be written. If the expression in the field width is followed by one of the identifiers OCT or HEX, the value to be written must be of type integer; the value is output in octal or hexadecimal form. Integers only can be written in octal form. If the field width is longer than needed, the value is written right justified. The field width must be an integer expression with value greater than or equal to 0. If omitted, a default value is chosen, in accordance with the following table:

Type	Default Field Width	Remarks
integer	10	If the field width is too short, the necessary number of additional character positions is used.
Boolean	10	If the field width is 5 or more either of the strings ' TRUE' or 'FALSE' is written.
		If the field width is 0, 1, 2, 3, or 4 either of the characters 'T' or 'F' is written.
char	1	If the field width is 0, the default field width 1 is used.
real	22	If fraction length is not specified the value will be written with 1 digit before the decimal point; 13 digits after the decimal point; and a scaling exponent written as E±ddd (floating point notation).
		If fraction length is specified, the fraction length must be at least two less than the field width. The fraction length specifies the number of digits to follow the decimal point. If the fraction length is specified no exponent is written (fixed point notation). If the field width is too short the necessary number of additional character positions is used.
string	length of string	If a nonzero field width less than the length of the string is specified, the right part of the string is truncated. If a field width equal to 0 is specified the entire string is written.

Predefined Textfiles INPUT and OUTPUT

Two textfiles named INPUT and OUTPUT are predefined:

```
    VAR
         INPUT,OUTPUT : TEXT;
```

The call LINELIMIT(OUTPUT,1000) is automatically executed before the program is executed.

The first parameter to READ, READLN, WRITE, WRITELN, EOF, EOLN, or EOS can be omitted, in which case INPUT or OUTPUT respectively is used.

Let V denote a variable of type CHAR, INTEGER, or REAL and E denote an expression of type CHAR, INTEGER, REAL, BOOLEAN, or STRING.

```
    WRITE(E)        is equivalent to    WRITE(OUTPUT,E)
    WRITELN(E)      is equivalent to    WRITELN(OUTPUT,E)
    READ(V)         is equivalent to    READ(INPUT,V)
    READLN(V)       is equivalent to    READLN(INPUT,V)
    EOF             is equivalent to    EOF(INPUT)
    EOLN            is equivalent to    EOLN(INPUT)
    EOS             is equivalent to    EOS(INPUT)
```

The predefined textfiles INPUT and OUTPUT correspond to the NOS files INPUT and OUTPUT respectively. Section 6 shows how other external files are declared.

If a file is to be printed, the first character of each line can be used as a carriage control character (depending on the printing device) and the line length can be limited as well.

If a Pascal file is assigned to a terminal, you should be aware that when output is printed (the buffer emptied), the job is swapped out because it is waiting for input data.

To ensure that all output generated up to the present moment is sent to the terminal, you should use the standard procedure WRITELN. Only whole lines will be written because each empty terminal line is treated as an end-of-file mark.

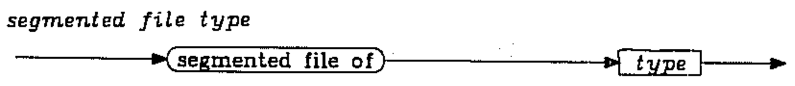
If an external file is used for interactive input, a slash (/) following the file name in the program heading makes it possible to write output to the terminal before any input is read. READLN skips a new line of input and in the case of interactive input ask the user for input by writing a question mark.

The following is an example of interactive use of INPUT and OUTPUT:

```
    PROGRAM IO(INPUT/,OUTPUT);
    VAR
         ID : INTEGER;
         CH : CHAR;
    BEGIN
         WRITELN('PLEASE ENTER YOUR IDENTIFICATION');
         READLN;
         READ(ID);
         .
         .
         .
         WRITELN('NOW GIVE YOUR CONTROL CHARACTER');
         READLN;
         READ(CH);
         .
         .
         .
    END.
```

SEGMENTED FILES

A segmented file makes it possible to manipulate logical records, which are a subdivision of a file into segments of varying length. A segmented file type is defined by prefixing a usual file type definition with the reserved symbol SEGMENTED.



The predefined files INPUT and OUTPUT can be specified as segmented files in the program heading.

A number of routines are provided for manipulating segmented files. Assume that F is of a SEGMENTED file type.

EOS(F) Returns a Boolean true if the file is positioned at an end-of-record mark and a false if it is not. The value of the file buffer F↑ is undefined if EOS(F) is true.

GETSEG(F) Positions the file at the start of the next segment. The file buffer F↑ becomes the first component of the next segment. If no next segment is present execution is terminated and the runtime error message: TRIED TO READ PAST EOS/EOF is given. GETSEG can only be applied to a file that is being read.

PUTSEG(F) Closes the current segment (an end-of-record mark is written onto F). PUTSEG is only allowed if EOF(F) is true.

A segmented file makes it possible to move the file (relatively) quickly to any segment in the file. For the purpose of reading and (re)writing a segmented file, the predefined procedures GETSEG and REWRITE are extended to accept two arguments. Assume that F is of a segmented file type and F an integer expression.

GETSEG(F,N) Positions the file at the start of Nth segment counting from the current position of the file.

The file buffer F↑ becomes the first component of the Nth segment.

N > 0 implies counting segments in the forward direction.

N = 0 means the current segment.

If no Nth segment (N >= 0) is present, EOF(F) becomes true and F↑ becomes undefined.

N < 0 implies counting segments in the backward direction.

If the file is positioned at segment number M, M < -N, then GETSEG(F,N) is equivalent to RESET(F).

REWRITE(F,N) Initiates the file for (re)writing F at the Nth segment counting from the current position. EOS(F) becomes true.

N > 0 implies counting segments in the forward direction.

N = 0 means the current segment.

If no Nth (N >= 0) segment is present the file is initiated for the writing of F after the last segment and EOF(F) becomes true.

N < 0 implies counting segments in the backward direction.

If the file is positioned at segment number M, M < -N, then REWRITE(F,N) is equivalent to REWRITE(F).

Current segment number R, R >= N, counted from the current position is not accessible after the execution of REWRITE(F,N).

Because files are organized for sequential forward processing, GETSEG and REWRITE are not as efficient for N <= 0 as for N > 0.

The following points about segmented files should be noted:

- EOF(F) always implies EOS(F).
- GET(F) is only applicable when EOS(F) is false.
- PUT(F) and PUTSEG(F) are only applicable when EOS(F) is true.
- The routines PUTSEG, GETSEG, and EOS can only be applied to segmented files.

Files denoted by file names must be declared as file variables in the block of the program; an exception to this is INPUT and OUTPUT. The files listed in the program heading are called external files.

COMPILING, LOADING, AND EXECUTING 5

A Pascal job usually passes through the following steps:

1. The source code (program) is compiled. The compiler generates object, or relocatable, code and a listing of the source code if the L compiler option is selected.

2. The object code is loaded and linked with pre-compiled routines (for example, routines for input and output and routines predefined by the user).

3. The loaded code is executed.

You initiate these steps with appropriate control statements to the NOS 2 operating system. The following sequence shows the basic control statements to compile, load, and execute a program:

```
  . . . ,CM50000,S10,P3.
USER, . . ., . . . .
CHARGE, . . ., . . . .
PASCAL.  <------------------------   Step 1
LGO.    <-------------------------   Steps 2 and 3
EOR
PROGRAM SAMPLE
  BEGIN
    .
    .
    .
  END
EOR
  data
EOF
```

ORGANIZATION OF A COMPILED PROGRAM

The object code that is generated by the compiler is relocatable binary code separated into named logical records, or modules. Each module contains the code for a block in the program. The modules occur in the same order as their corresponding compound statements. Global variables are placed in a separate module. The module names depend on the E compiler option. See the description of the E option under the heading Compiling a Program for an explanation of the entry point names in the object code modules.

Here are two examples of source code and the object code they produce:

Source code

```
    (*$E+*)
    PROGRAM A(OUTPUT);
    PROCEDURE B;
        BEGIN
        .
        .
        .
        END;
    PROCEDURE C;
        PROCEDURE D;
            BEGIN
            .
            .
            .
            END;
        PROCEDURE E;
            BEGIN
            .
            .
            .
            END;
        BEGIN
        .
        .
        .
        END;
    BEGIN
    .
    .
    .
    END.
```

Object code

Record:	
1	B
2	D
3	E
4	C
5	A
6	A;

Source code

```
    (*$E+*)
    PROGRAM K(OUTPUT);
    PROCEDURE L;
        BEGIN
        .
        .
        .
        END;
    PROCEDURE M;
    FORWARD;
    PROCEDURE N;
        BEGIN
        .
        .
        .
        END;
    PROCEDURE M;
        BEGIN
        .
        .
        .
        END;
    BEGIN
    .
    .
    .
    END.
```

Object code

Record:	
1	L
2	N
3	M
4	K
5	K;

COMPILING A PROGRAM

To initiate compilation of your program, use the control statement

```
    PASCAL(sfn,lfn,bfn/opts)
```

where

sfn	Program source file name; the default name is INPUT.
lfn	Program listing file name; the default name is OUTPUT.
bfn	Binary object file name; the default name is LGO.
opts	One or more compiler options.

The parameters sfn, lfn, and bfn are order-dependent; two consecutive commas within the parameter list request the compiler to use the default value for the missing parameter. For example,

```
    PASCAL(SS,,BB)
```

This control statement requests the compiler to compile source file SS and to produce both a program listing file named OUTPUT, the default program listing file, and a binary object file named BB.

At least 50000 octal words of common memory are needed to run the compiler.

You can control the compilation mode with compiler directives. For example, you can request the compiler to insert or omit runtime test instructions with compiler directives.

Compiler directives are written as comments, but with a dollar sign ($) as the first character.

Compiler directives can be placed anywhere in a program, which enables you to activate options over specific parts of your program.

Each option consists of an option letter followed by the new value of the option setting. The value may be a + or − which turns some options on and off like switches. Alternately, the value may be a decimal or octal (indicated by a radix B) integer for numeric options, or a literal string for string options (see the E, I, and L options). The rules for these strings are the same as those for character strings appearing in a Pascal program. Finally, for all options except the I option, if the value is an equals sign (=), the option is set to its previous value. However, only one previous value is remembered.

Option scanning terminates when any entry that is not an appropriate option letter or option value is entered. For example, setting a switch option to a numeric value, will cause option scanning to end with no error messages produced (except with the I option). Errors also terminate option scanning.

The following options are available:

B   Determines the size of file buffers. If the value of the B option is less than 64, it is a buffer factor and the actual buffer size (in words) is at least 128 times the buffer factor. If the value is larger than 64, it specifies the actual buffer size.

The compiler adds one to B, then rounds the value to the next multiple of the file element size. Buffer sizes must be adjusted to fit the requirements of peripheral hardware devices. Disk files need at least B1 (or B128). Tape files need at least B4 (or B512).

The buffer size for a file is bound to its type. The type text is predefined at the time the compiler reads the reserved symbol program. Therefore, to change the buffer size for textfiles (including INPUT and OUTPUT), the B option must be set prior to the program heading.

Default is B2.

E Allows you to control the entry-point names generated by the compiler for the main program, main variables block, procedures, functions, and labels. Entry points are required by the operating system loader; the E option is of special interest to you if you want to create a library of compiled, relocatable procedures and functions. The following paragraphs describe the effect of the E option:

a) Procedures and functions declared as EXTERN or FORTRAN get an entry-point name equal to the first seven characters of the procedure or function name. Other routines get an entry-point name depending on the value of the E option at the moment of analyzing the routine name:

E-	Creates a unique name of the form PRCnnnn (where nnnn is an octal number from 0001 to 7777) is generated by the compiler.
E+	Uses the first seven characters of the routine name as the entry-point name.

An extended form of the E option may be used to create an entry-point unrelated to the name of the routine. The following example illustrates this form for procedures and functions:

```
FUNCTION (*$E'P.RND'*) ROUND(X: REAL): REAL;
```

The entry-point for function round is actually P.RND. This gives the ability to define any entry point accepted by the loader, even ones that include special characters (such as a period). This form of the E option applies equally to EXTERN, FORTRAN, and local routines, but E must be specified between the word function or procedure and the routine name.

b) The main program and main variables block get an entry-point name depending on the value of E, as follows:

E-	Uses P.MAIN as the main program and main variables block entry-point name.
E+	Uses the first seven characters of the program name as the main program entry-point name and the first six characters of the program name followed by a semicolon as the main variables block entry-point name.

The extended form of the E option may be used for the main program, but two names should be specified for the main program block and the main variables block. For example,

```
PROGRAM (*$E'P.MAIN'/'P.VARS' *) MYPROG(OUTPUT);
```

c) Labels that are used by goto statements that exit a block are automatically assigned an entry-point name of the form PASCL.X (where X is a letter or digit). The entry-point name of any label may be explicitly assigned with the extended E option. In this case, the E option must immediately precede the declaration of the label. For example,

```
LABEL (*$E'L.1' *) 1,2, (*$E'L.LOOPS' *) 13;
```

It is your responsibility to ensure that duplicate entry-point names are not created when you specify the E- option. You must avoid creating duplicate entry-point names and must ensure that created entry-point names are acceptable to the system loader when you specify the extended form of the E option. The extended form of the E option exists mainly for the Pascal library.

Default is E-.

G Selects the automatic load and go feature, which allows a program to be compiled and executed in a single job step with one control statement. If G+ is selected when the program header is scanned, the binary object file is rewound before compilation. If G+ is selected at the end of compilation and the program is error-free, the binary object file is loaded and executed automatically.

Default is G-.

I Controls the inclusion of external text. The I option includes source code from an external file. This directive has the following two forms:

```
(*$I'PACKAGE'/'FILE'*)

(*$I'PACKAGE'*)
```

The first form attempts to find an entry named PACKAGE on the file named FILE. The second form attempts to find the entry named PACKAGE on the default file, which is PASCLIB for NOS 2. The included text is not restricted to declarations. It can also contain full procedures and functions. Because the text entry is simply inserted into the text of your program, the include facility can be used to create full source libraries.

The included text is written on the program listing if L+ was selected, thereby giving you an accurate record of what was compiled. A complete record is important if you plan to transport the program to another implementation of Pascal. Compiler options embedded within included text will change previous option settings unless they are explicitly restored with an equal sign (=) in the text itself.

L Controls the listing of the program text. The L option turns the listing on and off during compilation, specifies the size of each printed page, and sets page titles and subtitles. L+ turns the listing on and L- turns it off. If the L is followed by a number, the number defines the page size by specifying the last line to be printed on each page. $L \geq 1000$ selects no pagination. If L is followed by a character string, the page title or subtitle is set. The first such specification sets the main title, while subsequent specifications set the subtitle and cause a page eject. To set the title on the first printed page, the L option must appear on the first line.

Default is L+.

P Directs the compiler to generate a Post Mortem Dump (PMD) listing in the event of a runtime error. P+ requests the PMD facility to provide a description of each procedure or function that was active at the time of the error, including the line number of the statement which was currently being executed, and the names and values of all unstructured local variables. Values of pointer variables are printed as 6-digit octal addresses, and values of ALFA variables are printed as 10-character strings. A value of UNDEF means undefined. P+ is recommended until you are sure that your program is correct. P- suppresses most of the PMD information; it includes enough information to list the name of the procedure in which the error occurred. P0 is an option setting designed especially for the Pascal compiler and library. Procedures compiled with P0 are transparent to PMD. Compiling an entire program with P0 deletes the minimal information (3 words per procedure), which includes the name of the procedure and the locations of the entry point and constants. P0 can be used for production programs to delete all unnecessary traceback information.

Default is P+.

R Controls reduce mode. R is used in conjunction with the W option to control execution field length.

Default is R+.

T Directs the compiler to generate extra code that can be used to perform runtime tests to check the following:

a) That the index used for array-indexing operations lies within the specified array bounds.

b) That the value that is assigned to a variable of a subrange type lies within the specified range. This check is also performed when reading such variables.

c) That no divide-by-zero operations were performed.

d) That the absolute value of the result of an automatic real-to-integer conversion is less than MAXINT.

e) That there was no overflow or underflow from a real expression.

f) That the evaluated expression in a CASE statement corresponds to a constant in a case list element (unless OTHERWISE is used).

g) That P is a valid pointer when it is referenced as P or DISPOSE(P). The T+ option must be selected when the pointer type is declared and when the pointer is referenced.

h) That SET elements are within the declared range after assignments to set variables are made.

Also, the control variable in all FOR statements is set to an undefined value upon normal exit from the statement if T+ is selected. T+ is recommended until you are sure that your program is correct.

Default is T+.

U Restricts the number of characters that are scanned by the compiler in every source line. U+ restricts the number of characters to 72. This is convenient when using the default widths under the UPDATE or MODIFY text maintenance programs. U- sets the number of relevant characters to 120. U may be set to any specific numeric value between 10 and 120. The remainder of the line (past the width specified by this option) is treated as a comment. The U option is best used on the first line of the Pascal source program.

Default is U-.

W Controls the workspace size. W is used in conjunction with the R option to control runtime field length.

Default is W0.

X Determines the number of X registers used for passing parameter descriptors. If the value of the X option is in the range ($0 \le N \le 5$), the first N parameter descriptors are passed in the registers X0 to X(N-1) (the first in X0, the second in X1, and so on). Extra parameters are passed through a table in memory.

N > 0 reduces the size of the code produced by the compiler and usually decreases the execution time. However, you must be aware that with the first parameter and with N > 0, the compiler cannot use registers X0 to XJ (where J is the minimum of (N-1) and (I-2)) for its computation. It is possible for the compiler to give the message: EXPRESSION TOO COMPLICATED where N > 0.

Default is X4.

OVERVIEW OF THE RUNTIME SYSTEM

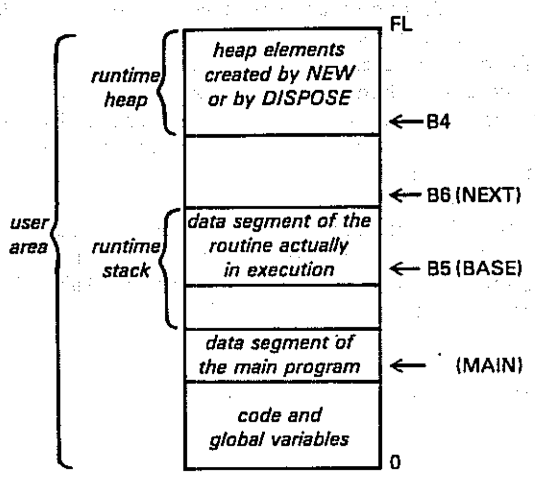
Code and data are separated from each other at runtime. The local data from each executed routine is united in a data segment and is addressed by an offset relative to the segment origin (the so-called BASE address) from this time on. At runtime, a stack containing the data segments of all executed routines is provided. Because the base addresses of the data segments vary during runtime, variable addressing is nontrivial. However, this way of organizing data guarantees maximum storage economy. Every data segment exists only during the routine execution; the data segment is created at routine entry and discarded at routine exit.

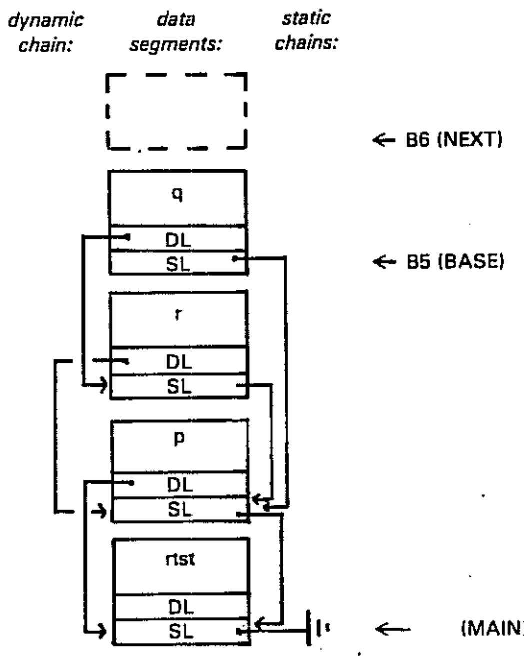
To allow stacking and unstacking of data segments, a link is needed. This link, called the dynamic link (DL), chains every data segment to its immediate predecessor in the stack. Variable addressing is done through a second link, called the static link (SL), which chains only those data segments which are currently accessible. SL and DL are incorporated in the head of every data segment.

For example, refer to the following source code:

```
(*$E+*)
PROGRAM RSTS(OUTPUT);
PROCEDURE P;
    PROCEDURE Q;
         BEGIN
           .
           .
           .
         END;
    PROCEDURE R;
         BEGIN
           .
           .
           .
           Q;
           .
           .
           .
         END;
    BEGIN
      .
      .
      .
      R;
      .
      .
      .
    END;
    BEGIN
      .
      .
      .
      P;
      .
      .
      .
    END.
```

This is the stack of data segments that correspond to the program.


FL
heap elements created by NEW or by DISPOSE
runtime heap
← B4
← B6 (NEXT)
user area
runtime stack
data segment of the routine actually in execution
← B5 (BASE)
data segment of the main program
← (MAIN)
code and global variables
0



The stack, growing upwards, originates from the calling sequence: RTST $\rightarrow$ P $\rightarrow$ R $\rightarrow$ Q. BASE is the base address of the most recently created data segment. It is the head of the chains. NEXT defines the base address of the next data segment to be stacked.

LOADING AND EXECUTING A PROGRAM

To initiate loading and execution of your program, use the control statement:

```
OC(f1,f2, . . . ,fn)
```

where

OC The file that contains the object code, or relocatable binary code.

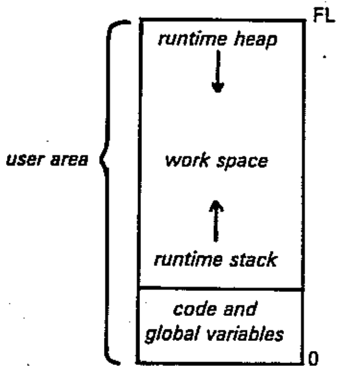
fi The names of files that contain routines that are external to the program, but that are used during execution.

Routines that are referenced, but not included in file OC, are searched for in the PASCLIB system library.

Routines that are referenced, but not included either in the file OC or in the PASCLIB system library, are searched for in the system library.

It is possible to load and initiate execution in several other ways.

After completion of the loading process, a contiguous piece of unused memory remains at the upper end of the user area. This area is called the work space and is used for the runtime stack and runtime heap during execution. The runtime stack grows upward from the lower end while the runtime heap grows downward from the upper end.



The W compiler option controls the calculation of the work space (WS) value.

Wn sets the number of words to be used for the WS (n is a string of digits with an optional post-radix B).

W0 requests the Pascal compiler to calculate an appropriate WS size. Pascal sums the lengths of all nongobal variables declared in the program, then adds a safety factor of 2000 octal (1024 decimal) words. The value that the compiler estimates for the W option is printed at the bottom of the compiler listing.

The R compiler option controls what is done with the WS value. R+ requests that the user program be given the right amount of memory for both the code including global variables (CS) and the WS, even if this is a reduction. R- requests that the memory be increased only if it is necessary to satisfy the sum of the CS and WS. In other words, the memory allocation will never be decreased if R- is set. This option has an effect which is analogous to the REDUCE control statement.

The default settings are W0,R+. This causes Pascal to calculate the WS value and requests that memory allocation be set to reflect this, regardless of whether or not an increase or decrease is required. These option settings will always allocate enough memory for programs that do not use recursion or dynamic allocation, which is the case for most programs. In some cases, however, the defaults may not be appropriate.

When setting the Work Space value explicitly, you should note that there is hidden data (temporary space for anonymous variables) that is used by Pascal program itself. Therefore, you should increase your WS estimate to provide a margin of safety. A good rule of thumb is to add about 10 words per procedure plus an additional several hundred words.

UNDERSTANDING RUNTIME ERROR MESSAGES

When a runtime error occurs, a dayfile message explaining the error is given together with a Post Mortem Dump.

FORTRAN AND PASCAL INCOMPATIBILITIES

Some incompatibilities exist between the Pascal and FORTRAN languages. Two of these are the representation of values and the method of storing multidimensional array values. You may be forced to do some extra programming to get around these incompatibilities.

Table 5-1 shows parameter types in a Pascal routine that correspond to parameter types in a FORTRAN routine. The Pascal compiler does not test for illegal parameter types as in FORTRAN. As in FORTRAN, trailing parameters can be omitted.

Alternate returns from a FORTRAN routine are not allowed.

TABLE 5-1. CORRESPONDING PASCAL AND FORTRAN ROUTINE PARAMETER TYPES

Parameter Type in a FORTRAN Routine	Parameter Type in a Pascal Routine	Remarks
INTEGER	INTEGER	With variable parameters of integer, real, double, and complex types, a negative zero (-0) may be returned by the FORTRAN routine. To eliminate this possibility, you should add a zero to the value upon returning to the Pascal routine.
REAL	REAL	

DOUBLE	RECORD
	P1:REAL;
	P2:REAL
	END	

COMPLEX		
LOGICAL	INTEGER	Return a negative value for true and a positive value for false.
DIMENSION	ARRAY	You must either transpose multidimensional array values before entering a FORTRAN routine or remember that array values are stored rowwise when manipulating them in the FORTRAN routine. Always set the lower array bound to 1.
SUBROUTINE	PROCEDURE	
FUNCTION	FUNCTION	The result returned to the Pascal routine cannot be complex, double, or a negative zero. To eliminate the possibility of a negative zero, you should add a zero to the value upon returning to the Pascal module.

SAMPLE PROGRAMS 6

This section poses some problems and provides a solution.

The first problem deals with placing a class of three steers.

In a judging contest, the official judges the steers on qualities such as height, straightness along the back, and amount of muscle. The steers are numbered 1, 2, and 3, so it is possible for the official to determine the correct placing as: 3, 1, 2.

After the official determines the correct placing, students judge the same class to determine what they feel is the correct placing (the official's placing is unknown to the students).

A student can place the class as one of the following combinations:

```
3   1   2
3   2   1
1   3   2
2   1   3
2   3   1
1   2   3
```

A perfect match between the official's and a student's placing is awarded 50 points. A student whose placing does not match the official's is penalized for each incorrect decision that was made. The penalty is calculated using a number called the degree of difficulty or cut. The cut between a pair of steers is also determined by the official. An example of a cut assignment is:

```
Official placing:  3   1   2

Cuts:                5   1
```

If the official assigns a cut of 5 between steers 3 and 1, then there is a clear difference in quality in the two steers; switching the placing of this pair results in a large penalty. If the official assigns a cut of 1 between steers 1 and 2, then there is a small difference in quality in the two steers; switching the placing of this pair results in a lesser penalty. The following are sample penalty calculations:

```
Official placing:  3   1   2

Cuts:                5   1

Student placing:   1   3   2
```

The score would be calculated as 50 - 5 = 45 because the top pair was switched.

```
Official placing:  1   2   3

Cuts:                1   3

Student placing:   3   2   1
```

The score would be calculated as 50 - (2*cut2 + 2*cut2) = 42 because the top and bottom placing was switched.

The problem is to write a Pascal program that accepts as input the official's placing, cuts, and student's placing, calculates the score, and outputs the score.

This solution uses arrays to hold the data, IF statements to perform the calculations, and labeled statements to control the flow of execution.

```
    PROGRAM JUDGE(INPUT/,OUTPUT);
    TYPE
         PLACINGS = ARRAY[1..3] OF INTEGER;
         CUTS = ARRAY[1..2] OF INTEGER;
    VAR
         O,J : PLACINGS;
         CUT : CUTS;
         I,RESULT : INTEGER;
    LABEL
         50,75;
    BEGIN
    (** INPUT OFFICIAL PLACING. **)
         WRITELN('INPUT OFFICIAL PLACING');
         FOR I := 1 TO 3 DO READ(O[I]);
    (** INPUT CUTS. **)
         WRITELN('INPUT OFFICIAL CUTS');
         FOR I := 1 TO 2 DO READ(CUT[I]);
    (** INPUT JUDGE'S PLACING OR ZERO. **)
    50 : WRITELN('INPUT JUDGES'' PLACING OR FOUR ZEROS');
         FOR I := 1 TO 3 DO READ(J[I]);
         IF (J[1] = 0) THEN GOTO 75;
    (** BEGIN CALCULATION OF SCORE.  PERFECT SCORE. **)
         IF ((O[1]=J[1]) AND (O[2]=J[2]))
              THEN RESULT := 50;
    (** TOP AND BOTTOM PAIR SWITCHES. **)
         IF ((O[1]=J[2]) AND (O[2]=J[1]))
              THEN RESULT := 50 - CUT[1];
         IF ((O[2]=J[3]) AND (O[3]=J[2]))
              THEN RESULT := 50 - CUT[2];
    (** TOP TO BOTTOM. **)
         IF ((O[1]=J[3]) AND (O[2]=J[1]))
              THEN RESULT := 50 - (2*CUT[1] + CUT[2]);
    (** SIMPLE BUST. **)
         IF ((O[1]=J[2]) AND (O[2]=J[3]))
              THEN RESULT := 50 - (CUT[1] + 2*CUT[2]);
    (** MAJOR BUST. **)
    IF ((O[1]=J[3]) AND (O[2]=J[2]))
              THEN RESULT := 50 - (2*CUT[1] + 2*CUT[2]);
    (** OUTPUT SCORE. **)
         WRITELN('SCORE IS ',RESULT:2);
         GOTO 50;
    75 : WRITELN('END OF PROGRAM')
    END.
```

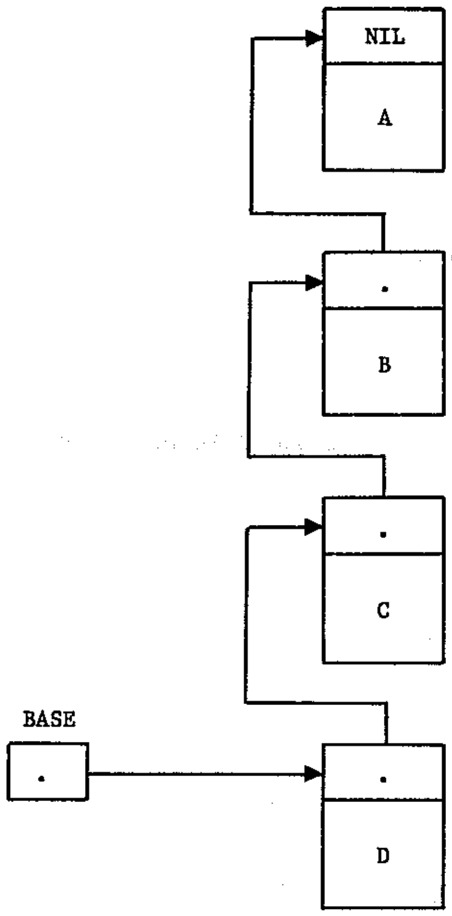
This solution decodes the student's placing to match the placing: 1 2 3 using <u>WHILE</u> and <u>REPEAT</u> statements and then calculates the penalty using a <u>CASE</u> statement and a function. The use of <u>WHILE</u> or <u>REPEAT</u> statements to control execution of a program is preferred over the use of labeled statements because the result is a more structured program.

```
PROGRAM JUDGE(INPUT/,OUTPUT);
TYPE
     PLACINGS = ARRAY[1..3] OF INTEGER;
     CUTS = ARRAY[1..2] OF INTEGER;
VAR
     O,J,R : PLACINGS;
     CUT : CUTS;
     I,M,N,SCORE : INTEGER;
FUNCTION RESULT(X,Y : INTEGER) : INTEGER;
     BEGIN
          RESULT := 50 - (X*CUT[1] + Y*CUT[2])
     END;
BEGIN
(** INPUT OFFICIAL PLACING. **)
     WRITELN('INPUT OFFICIAL PLACING');
     FOR I := 1 TO 3 DO READ(O[I]);
(** INPUT CUTS. **)
     WRITELN('INPUT OFFICIAL CUTS');
     FOR I := 1 TO 2 DO READ(CUT[I]);
(** INPUT JUDGE'S PLACING. **)
     WRITELN('INPUT JUDGES'' PLACING');
     FOR I := 1 TO 3 DO READ(J[I]);
(** CREATE ARRAY R AS IF OFFICIAL PLACING WERE 1 2 3. **)
     FOR N := 1 TO 3 DO
          BEGIN
               M := 0;
               REPEAT
                    M := M + 1;
               UNTIL J[M] = O[N];
               R[M] := N
          END;
(** CALCULATE RESULT. **)
     CASE (100*R[1] + 10*R[2] + R[3]) OF
          123 : SCORE := RESULT(0,0);
          132 : SCORE := RESULT(0,1);
          213 : SCORE := RESULT(1,0);
          231 : SCORE := RESULT(2,1);
          312 : SCORE := RESULT(1,2);
          321 : SCORE := RESULT(2,2)
     END;
     (** OUTPUT SCORE. **);
     WRITELN('SCORE IS ',SCORE:2);
     WRITELN('END OF PROGRAM')
END.
```

The second problem deals with building a linked list. The following program creates a last-in-first-out (LIFO) linked list of four nodes. The data area in each node is assigned a character in the alphabet. After the linked list is constructed, it is traversed from the last entry to the first entry. Traversal is verified by writing the contents of the data area in each node.

```
PROGRAM LNKLIST(INPUT/,OUTPUT);
TYPE
     POINTER = ↑NODE;
     NODE = RECORD
               NEXTPNTR : POINTER;
               DATA : CHAR
            END;
VAR
     BASE,PNTR : POINTER;
     I : INTEGER;
BEGIN
(** CREATE A POINTER THAT POINTS TO NIL. **)
     BASE := NIL;
(** CREATE NODES AND LINK THEM. **)
     FOR I := 1 TO 4 DO
          BEGIN
          (** CREATE A NEW NODE. **)
               NEW(PNTR);
          (** PUT DATA INTO THE NODE DATA AREA. **)
               READLN(PNTR↑.DATA);
          (* PUT THE BASE POINTER VALUE INTO THE NODE POINTER. *)
               PNTR↑.NEXTPNTR := BASE;
          (** POINT THE BASE POINTER TO THE NODE. **)
               BASE := PNTR
          END;
     PNTR := BASE;
     WHILE PNTR <> NIL DO
          BEGIN
               (** VERIFY ORDER OF NODES. **)
               WRITELN(PNTR .DATA);
               (** POINT TO THE NEXT NODE. **)
               PNTR := PNTR↑.NEXTPNTR
          END;
     WRITELN('END OF PROGRAM')
END.
```

If you insert A, B, C, D as data for the nodes, the resulting linked list would appear as follows:

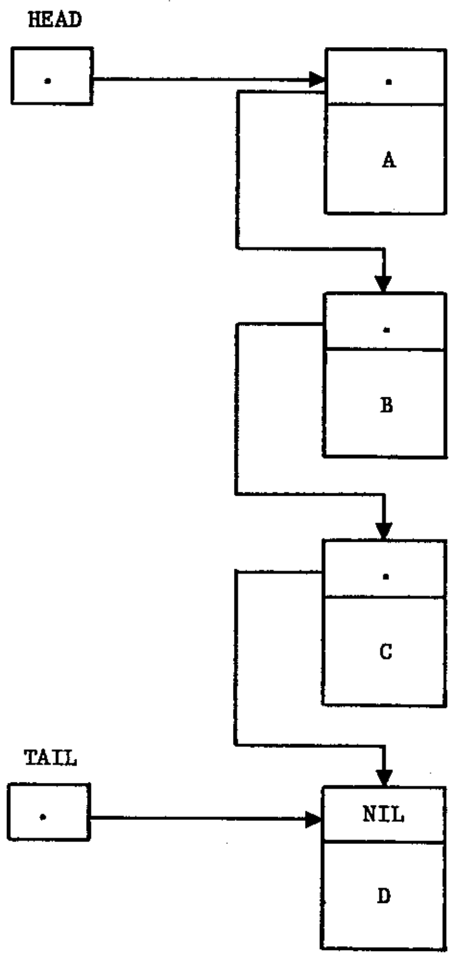


The following program is a variation of the linked list program. A linked list of four nodes is again created, but the first node is pointed to by a pointer named HEAD and the last node by a pointer named TAIL. The advantage of creating the list this way is that modifying the list is much easier.

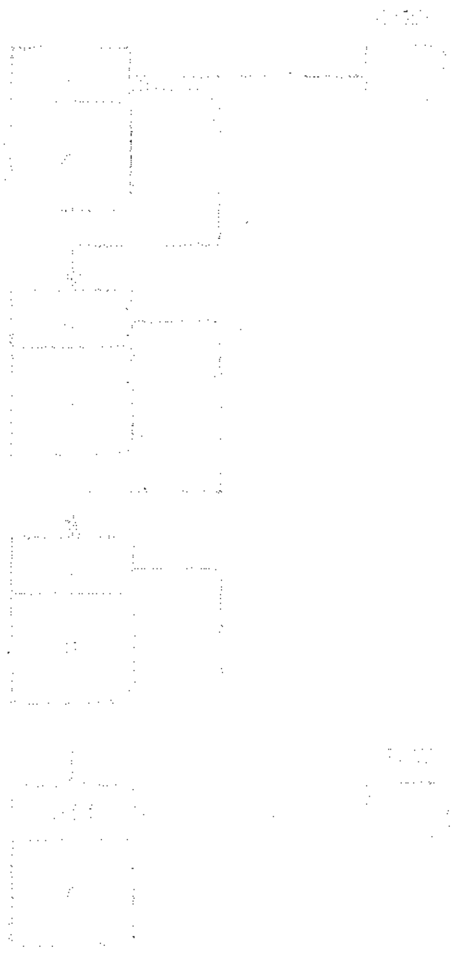
The list must contain at least one node.

```
PROGRAM HEADTAIL(INPUT/,OUTPUT);
TYPE
     POINTER = ↑NODE;
     NODE = RECORD
                NEXTPNTR : POINTER;
                DATA : CHAR
             END;
VAR
     HEAD,TAIL,PNTR : POINTER;
     I : INTEGER;
BEGIN
     (** CREATE FIRST NODE AND POINT HEAD AND TAIL TO IT. **)
     NEW(PNTR);
     READLN;
     READ(PNTR↑.DATA);
     PNTR↑.NEXTPNTR := NIL;
     HEAD := PNTR;
     TAIL := PNTR;
     (** CREATE OTHER THREE NODES. **)
     FOR I := 1 TO 3 DO
          BEGIN
               NEW(PNTR);
               READLN;
               READ(PNTR↑.DATA);
               PNTR↑.NEXTPNTR := TAIL↑.NEXTPNTR;
               TAIL↑.NEXTPNTR := PNTR↑.NEXTPNTR;
               TAIL := PNTR
          END;
     (** VERIFY ORDER OF NODES. **)
     PNTR := HEAD;
     REPEAT
               WRITELN(PNTR↑.DATA);
               PNTR := PNTR↑.NEXTPNTR
     UNTIL PNTR↑.NEXTPNTR = NIL;
     WRITELN(PNTR↑.DATA);
     WRITELN('END OF PROGRAM')
END.
```

If you insert A, B, C, D as data for the nodes, the resulting linked list would appear as follows:



[illegible]



CHARACTER SETS A

This appendix describes character correspondence between the internal Pascal character set and the CDC Scientific and CDC ASCII character sets.

All program statements in this manual are shown in the internal Pascal character representation. You must translate this representation into the character set used at your site.

Ordinal Number	Pascal Character	CDC Scientific Character Set	CDC ASCII Character Set
0	undefined	End of Line in 63 : (colon) in 64	End of Line in 63 : (colon) in 64
1	A	A	A
2	B	B	B
3	C	C	C
4	D	D	D
5	E	E	E
6	F	F	F
7	G	G	G
8	H	H	H
9	I	I	I
10	J	J	J
11	K	K	K
12	L	L	L
13	M	M	M
14	N	N	N
15	O	O	O
16	P	P	P
17	Q	Q	Q
18	R	R	R
19	S	S	S
20	T	T	T

Ordinal Number	Pascal Character	CDC Scientific Character Set	CDC ASCII Character Set
21	U	U	U
22	V	V	V
23	W	W	W
24	X	X	X
25	Y	Y	Y
26	Z	Z	Z
27	0	0	0
28	1	1	1
29	2	2	2
30	3	3	3
31	4	4	4
32	5	5	5
33	6	6	6
34	7	7	7
35	8	8	8
36	9	9	9
37	+	+	+
38	-	-	-
39	*	*	*
40	/	/	/
41	(	(	(
42	)	)	)
43	$	$	$
44	=	=	=
45	(space)	(space)	(space)
46	, (comma)	, (comma)	, (comma)
47	. (period)	. period	. period
48		≡ (equivalence)	# (number sign)
49	[(left bracket)	[(left bracket)	[(left bracket)

Ordinal Number	Pascal Character	CDC Scientific Character Set	CDC ASCII Character Set
50	] (right bracket)	] (right bracket)	] (right bracket)
51	: (colon)	: (colon) in 63 % (percent) in 64	: (colon) in 63 % (percent) in 64
52		≠ (not equal)	" (quote)
53		↱ (right arrow)	_ (underline)
54		∨ (logical OR)	! (exclamation)
55		∧ (logical AND)	& (ampersand)
56	′ (apostrophe)	↑ (up arrow)	′ (apostrophe)
57		↓ (down arrow)	? (question)
58	< (less than)	< (less than)	< (less than)
59	> (greater than)	> (greater than)	> (greater than)
60		≤ (less equal)	@ (commercial at)
61		≥ (greater equal)	\ (back slash)
62	↑ (up arrow)	¬ (logical not)	^ (circumflex)
63	; (semicolon)	; (semicolon)	; (semicolon)

COMPILATION ERROR MESSAGES B

The compiler indicates an error by printing an arrow that points to the place in the text where the error is detected. This is not always the place where the error is made. The arrow is followed by a number, which indicates what kind of error was detected. A list of numbers used in error messages and their corresponding messages is given at the end of the compilation. The list is given on the file containing the compiler listing.

At most 10 errors will be indicated on one line.

MESSAGES

1:	ERROR IN SIMPLE TYPE
2:	IDENTIFIER EXPECTED
3:	'PROGRAM' EXPECTED
4:	')' EXPECTED
5:	':' EXPECTED
6:	UNEXPECTED SYMBOL
7:	ERROR IN PARAMETER LIST
8:	'OF' EXPECTED
9:	'(' EXPECTED
10:	ERROR IN TYPE
11:	'[' EXPECTED
12:	']' EXPECTED
13:	'END' EXPECTED
14:	';' EXPECTED
15:	INTEGER CONSTANT EXPECTED
16:	'=' EXPECTED
17:	'BEGIN' EXPECTED
18:	ERROR IN DECLARATION PART
19:	ERROR IN FIELD-LIST
20:	',' EXPECTED
21:	'..' EXPECTED
40:	VALUE PART ALLOWED ONLY IN MAIN PROGRAM
41:	TOO FEW VALUES SPECIFIED
42:	TOO MANY VALUES SPECIFIED
43:	VARIABLE INITIALIZED TWICE
44:	TYPE IS NEITHER ARRAY NOR RECORD
45:	REPETITION FACTOR MUST BE GREATER THAN ZERRO
50:	ERROR IN CONSTANT
51:	':=' EXPECTED
52:	'THEN' EXPECTED
53:	'UNTIL' EXPECTED
54:	'DO' EXPECTED
55:	'TO' OR 'DOWNTO' EXPECTED
57:	'FILE' EXPECTED
58:	ERROR IN FACTOR
59:	ERROR IN VARIABLE
60:	FILE TYPE IDENTIFIER EXPECTED
101:	IDENTIFIER DECLARED TWICE
102:	LOWBOUND EXCEEDS HIGHBOUND
103:	IDENTIFIER IS NOT OF APPROPRIATE CLASS
104:	IDENTIFIER NOT DECLARED
105:	SIGN NOT ALLOWED
106:	NUMBER EXPECTED
107:	INCOMPATIBLE SUBRANGE TYPES
108:	FILE NOT ALLOWED HERE
109:	TYPE MUST NOT BE REAL

MESSAGES

110:	TAGFIELD TYPE MUST BE AN ENUMERATION TYPE
111:	INCOMPATIBLE WITH TAGFIELD TYPE
112:	INDEX TYPE MUST NOT BE REAL
113:	INDEX TYPE MUST BE AN ENUMERATION TYPE
114:	BASE TYPE MUST NOT BE REAL
115:	BASE TYPE MUST BE AN ENUMERATION TYPE
116:	ERROR IN TYPE OF PREDEFINED PROCEDURE PARAMETER
117:	UNSATISFIED FORWARD REFERENCE
118:	IDENTIFIER USED PRIOR TO DECLARATION
119:	FORWARD DECLARED; REPETITION OF PARAMETER LIST NOT ALLOWED
120:	FUNCTION RESULT TYPE MUST BE A SIMPLE OR POINTER TYPE
121:	FILE VALUE PARAMETER NOT ALLOWED
122:	FORWARD DECLARED FUNCTION; REPETITION OF RESULT TYPE NOT ALLOWED
123:	MISSING RESULT TYPE IN FUNCTION DECLARATION
124:	FIXED-POINT FORMATTING ALLOWED FOR REALS ONLY
125:	ERROR IN TYPE OF PREDEFINED FUNCTION PARAMETER
126:	NUMBER OF PARAMETERS DOES NOT AGREE WITH DECLARATION
127:	INVALID PARAMETER SUBSTITUTION
128:	PARAMETER PROCEDURE/FUNCTION IS NOT COMPATIBLE WITH DECLARATION
129:	TYPE CONFLICT OF OPERANDS
130:	EXPRESSION IS NOT SET TYPE
131:	TESTS ON EQUALITY ALLOWED ONLY
132:	'<' AND '>' NOT ALLOWED FOR SET OPERANDS
133:	FILE COMPARISON NOT ALLOWED
134:	INVALID TYPE OF OPERAND(S)
135:	TYPE OF OPERAND MUST BE BOOLEAN
136:	SET ELEMENT MUST BE AN ENUMERATION TYPE
137:	SET ELEMENT TYPES NOT COMPATIBLE
138:	TYPE OF VARIABLE IS NOT ARRAY
139:	INDEX TYPE IS NOT COMPATIBLE WITH DECLARATION
140:	TYPE OF VARIABLE IS NOT RECORD
141:	TYPE OF VARIABLE MUST BE FILE OR POINTER
142:	INVALID PARAMETER SUBSTITUTION
143:	INVALID TYPE OF LOOP CONTROL VARIABLE
144:	INVALID TYPE OF EXPRESSION
145:	TYPE CONFLICT
146:	ASSIGNMENT OF FILES NOT ALLOWED
147:	LABEL TYPE INCOMPATIBLE WITH SELECTING EXPRESSION
148:	SUBRANGE BOUNDS MUST BE OF AN ENUMERATION TYPE
150:	ASSIGNMENT OF THIS FUNCTION IS NOT ALLOWED
151:	ASSIGNMENT TO FORMAL FUNCTION IS NOT ALLOWED
152:	NO SUCH FIELD IN THIS RECORD
155:	CONTROL VARIABLE MUST NOT BE DECLARED ON AN INTERMEDIATE LEVEL
156:	MULTIDEFINED CASE LABEL
157:	RANGE OF CASE LABELS IS TOO LARGE
158:	MISSING CORRESPONDING VARIANT DECLARATION
159:	REAL OR STRING TAGFIELDS NOT ALLOWED
160:	PREVIOUS DECLARATION WAS NOT FORWARD
161:	MULTIPLE FORWARD DECLARATION
164:	SUBSTITUTION OF PREDEFINED PROCEDURE/FUNCTION NOT ALLOWED
165:	MULTIDEFINED LABEL
166:	MULTIDECLARED LABEL
167:	UNDECLARED LABEL
168:	UNDEFINED LABEL IN THE PREVIOUS BLOCK
169:	ERROR IN BASE SET
170:	VALUE PARAMETER EXPECTED
172:	UNDECLARED EXTERNAL FILE
173:	FORTRAN PROCEDURE OR FUNCTION EXPECTED
174:	PASCAL PROCEDURE OR FUNCTION EXPECTED
175:	MISSING FILE 'INPUT' IN PROGRAM HEADING'
176:	MISSING FILE 'OUTPUT' IN PROGRAM HEADING

177:	ASSIGNMENT TO FUNCTION ALLOWED ONLY IN FUNCTION BODY
178:	MULTIDEFINED RECORD VARIANT
179:	X-OPTION OF ACTUAL PROCEDURE/FUNCTION DOES NOT MATCH FORMAL DECLARATION
180:	CONTROL VARIABLE MUST NOT BE FORMAL
181:	ARRAY SUBSCRIPT CALCULATION TOO COMPLICATED
182:	MAGNITUDE OF CASE LABEL IS TOO LARGE
183:	SUBRANGE OF TYPE REAL IS NOT ALLOWED
184:	ASSIGNMENT TO CONTROL VARIABLE IS NOT ALLOWED
198:	ALTERNATE INPUT NOT FOUND
199:	ONLY ONE ALTERNATE INPUT MAY BE ACTIVE
201:	ERROR IN REAL CONSTANT: DIGIT EXPECTED
202:	STRING CONSTANT MUST BE CONTAINED ON A SINGLE LINE
203:	INTEGER CONSTANT EXCEEDS RANGE
204:	8 OR 9 IN OCTAL NUMBER
205:	STRING OF LENGTH ZERO ARE NOT ALLOWED
206:	INTEGER PART OF REAL CONSTANT EXCEEDS RANGE
207:	REAL CONSTANT EXCEEDS RANGE
250:	TOO MANY NESTED SCOPES OF IDENTIFIERS
251:	TOO MANY NESTED PROCEDURES AND/OR FUNCTIONS
255:	TOO MANY ERRORS ON THIS SOURCE LINE
256:	TOO MANY EXTERNAL REFERENCES
258:	TOO MANY LOCAL FILES
259:	EXPRESSION TOO COMPLICATED
260:	TOO MANY EXIT LABELS
261:	TOO MANY LARGE VARIABLES
262:	NODE TO BE ALLOCATED IS TOO LARGE
263:	TOO MANY PROCEDURE/FUNCTION PARAMETERS
264:	TOO MANY PROCEDURES AND FUNCTIONS
300:	DIVISION BY ZERO
302:	INDEX EXPRESSION OUT OF BOUNDS
303:	VALUE TO BE ASSIGNED IS OUT OF BOUNDS
304:	ELEMENT EXPRESSION OUT OF RANGE
305:	FIRST CHARACTER OF ENTRY POINT MUST BE A-Z, 0-4
350:	ONLY THE LAST DIMENSION MAY BE PACKED
351:	ARRAY TYPE IDENTIFIER EXPECTED
352:	ARRAY VARIABLE EXPECTED
353:	POSITIVE INTEGER CONSTANT EXPECTED
394:	COMPARISON OF DYNAMIC PARAMETERS NOT ALLOWED
395:	ASSIGNMENT TO/FROM DYNAMIC PARAMETER NOT ALLOWED
396:	MULTI-WORD VALUE PARAMETERS ARE NOT IMPLEMENTED FOR FORTRAN ROUTINES
397:	PACK AND UNPACK ARE NOT IMPLEMENTED FOR DYNAMIC ARRAYS
398:	IMPLEMENTATION RESTRICTION

RESERVED SYMBOLS

The following are reserved symbols that have a predefined meaning that cannot be changed. Throughout this manual reserved symbols are depicted in boldface type in the syntax diagrams and in underlined uppercase letters in the text. All alphabetic characters must appear in uppercase in your source program.

+	(	DYNAMIC	PACKED
-	)	ELSE	PROCEDURE
*	[	END	PROGRAM
/	]	FILE	RECORD
:=	(space)	FOR	REPEAT
.	..	FUNCTION	SEGMENTED
,	(*	GOTO	SET
;	*)	IF	THEN
:	AND	IN	TO
'	ARRAY	LABEL	TYPE
=	BEGIN	MOD	UNTIL
<>	CASE	NIL	VALUE
<	CONST	NOT	VAR
<=	DIV	OF	WHILE
>=	DO	OR	WITH
>	DOWNTO	OTHERWISE	

INDEX

COMMENT SHEET

MANUAL TITLE: Pascal Version 1 Reference Manual

PUBLICATION NO.: 60497700 REVISION: 01

NAME:

COMPANY:

STREET ADDRESS:

CITY: STATE: ZIP CODE:

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

___ Please reply ___ No reply necessary

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

TAPE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division

215 Moffett Park Drive
Sunnyvale, California 94086

FOLD

CUT ALONG LINE