

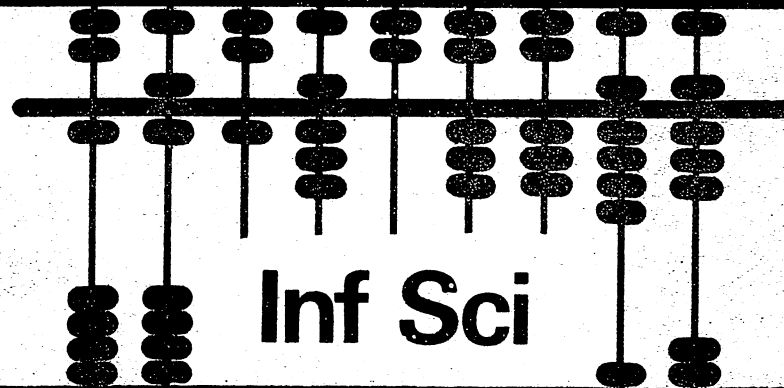
R80-4

**B6700/7700 PASCAL
Reference Manual**

Compiler Version III-0-001

1980 April

Department of Information Science
The University of Tasmania
G.P.O. Box 252C Hobart
Tasmania 7001



c Copyright 1977, by A.H.J. Sale

All rights reserved.

No part of this document may be reproduced by any means, nor transmitted, nor translated into a machine-readable form without the written permission of the author.

Professor A.H.J. Sale
Department of Information Science
University of Tasmania
Box 252C, G.P.O.,
Hobart, Tasmania 7001

CONTENTS

CONTENTS

1. INTRODUCTION

COMPLIANCE STATEMENT
INTRODUCTION TO THE MANUAL

2. LEXICAL TOKENS

LEXICAL TOKENS
CHAR CONSTANT
COMMENT
DOUBLET SYMBOLS
INTEGER CONSTANT
ONE-CHARACTER SYMBOLS
REAL CONSTANT
RESERVED WORDS
STRING CONSTANT
NAMES

3. SUBCOMPONENTS

SUBCOMPONENTS
ASSIGNMENT COMPATIBILITY
EXPRESSION
LABELS
NAME LIST
OPERATORS (ARITHMETIC)
OPERATORS (BOOLEAN)
OPERATORS (SET AND RELATIONAL)
PARAMETER LIST
SCALAR RANGE
SCOPE
SET CONSTRUCTOR
SIGNED INTEGER
SUBRANGE
TYPE COMPATIBILITY
TYPE IDENTITY
VARIABLE

4. DECLARATIONS

ARRAY TYPE
ATTRIBUTES
BOOLEAN TYPE
CHAR TYPE
CONST DECLARATION
FIELD LIST
FILE TYPE
FORMAT DECLARATION
INTEGER TYPE
LABEL DECLARATION
PACKED
POINTER TYPE
REAL TYPE
RECORD TYPE

SCALAR TYPE
SET TYPE
SUBRANGE TYPE
TEXT TYPE
TYPE DECLARATION

5. STATEMENTS

STATEMENTS
ASSIGNMENT
BODY
CASE STATEMENT
COMPOUND STATEMENT
EMPTY STATEMENT
FOR STATEMENT
GOTO STATEMENT
IF STATEMENT
PROCEDURE INVOCATION
REPEAT STATEMENT
WHILE STATEMENT
WITH STATEMENT

6. PROGRAM UNIT

PROGRAM UNIT
EXTERNAL DECLARATIONS
FORWARD DECLARATIONS
FUNCTION
PROCEDURE
PROGRAM

7. PRE-DEFINED PROCEDURES

ARITHMETIC FUNCTIONS
MARK AND RELEASE
MIN AND MAX
MIXED-TYPE FUNCTIONS
NEW
OPERATING SYSTEM PROCEDURES
PACK AND UNPACK
PASCAL GENERIC FUNCTIONS
RANDOM
TIME PROCEDURES

8. INPUT AND OUTPUT

INPUT AND OUTPUT
CLOSE
EOLN, EOF AND ENDOFFILE
GET AND PUT
PAGE
PRE-DEFINED FILES
READREC
READ
RESET AND REWRITE
SEEK

CONTENTS

SPACE
WRITEREC
WRITE

9. COMPILER OPTIONS

COMPILER OPTIONS

\$
ASCII
AUTOBIND
BIND
BINDER
BINDINFO
BOUNSCHECK
CHECK
CODE
ERRLIST
ERRORLIMIT
HEAP
HEXCODE
INCLNEW
INCLUDE
LINEINFO
LIST
LISTINCL
MERGE
NAMES
NEW
OMIT
PAGE
SEQ
SETSIZE
STANDARD
STATISTICS
STRIPBLANKS
TRUSTWORTHY
WARNINGS
USER-OPTIONS

10. COMPILER FILES

COMPILER FILES
FILE DEFINITIONS
FILE EQUATION

11. ERRORS

ERRORS
COMPILE-TIME ERRORS
INV-OPERATOR
PASCAL READ ERRORS
PASCAL WRITE ERRORS
RUN-TIME PASCAL ERRORS
RUN-TIME SYSTEM ERRORS
STACK HISTORY

12. SAMPLE PROGRAMS

13. GENERAL

CHARACTER SETS

WRAP UP INFO

COMPILER NOTES

INTRODUCTION

1. INTRODUCTION

The B6700/B7700 Pascal language is a dialect of the programming language Pascal, designed by Niklaus Wirth (see References) and first implemented for CDC 6000 computer systems. The implementation for the B6700 and B7700 computer systems was undertaken by the Department of Information Science at the University of Tasmania, and has a number of extensions from standard Pascal to adapt it to the new environment. Nevertheless, it is capable of handling programs written in Pascal and compiled on other machines, though its searching tests for 'undefined features' may cause the rejection of programs that compile successfully elsewhere.

The Pascal language is primarily intended for teaching programming, and in this aim it is unexcelled. An Algol-like language, it has a few clean executable statement kinds built on the Algol model and incorporating the improvements of knowledge of the 1970s. Its major advantage is its good facilities for data-typing and data-structuring, which are far superior to any other language on the B6700 or B7700 systems.

Pascal has also been touted as the long-awaited replacement for FORTRAN, as it has very similar capabilities and would permit FORTRAN-like constructs to be embedded in a Pascal program by binding. There are however two major problems with this suggestion which must be solved if the prediction is to come true. The first relates to the deviance of the Pascal i/o system from the record-oriented system most programmers are used to; to minimize the relearning process B6700/B7700 Pascal incorporates the record-oriented i/o system with formats from B6700/B7700 Algol (derived from FORTRAN itself with tidying). The second relates to the lack of adjustable arrays in Pascal: this problem is not tackled in this compiler as it requires some fundamental changes in the language.

The other major function envisaged for Pascal is that of a suitable vehicle for writing system software, for example compilers. With a minor addition of a routine it could be so used at its present level, and could certainly be used for all purposes short of generating code without change. Pascal is relatively successful in this area (though not perfect) mainly due to its good data structuring facilities.

The compiler was written with three major targets: that of providing a standard-compatible Pascal compiler for these Burroughs machines; that of providing an efficient implementation of Pascal; and that of making the compiler as compatible as possible with the rest of Burroughs' standard software. Very few additions or changes were necessary for this last purpose.

INTRODUCTION (COMPLIANCE STATEMENT)

COMPLIANCE STATEMENT

This Statement is made in conformance with the requirements of Section 5.1 of the draft ISO Standard for Pascal 1979 (N462). The compiler described in this manual purports to support Standard Pascal as described in Section 6 of the Standard with the following differences, extensions, observations, and implementation-dependent features.

The following sections are declarations made in accordance with the requirements of the Standard. All section numbers following refer to the Standard, not to this manual.

Implementation-defined features
(See 3 and 5.1.1(b))

The handling of these features may differ from processor to processor. Use of the features is permitted to Standard-conforming programs, but they must not rely on these specific interpretations nor any others.

Value of maxint (6.4.2.2 and 6.7.2.2)

$$549755813887 = 2^{**}39 - 1$$

Real values (6.1.5)

See manual for details of precision, range, etc.

Char values (6.1.5 and 6.6.6.4)

The char values are represented according to the EBCDIC code or the ASCII code, depending on the setting of the compiler option ASCII.

Component type of a set (6.4.3.4, 6.7.1 and 6.7.2.5)

The number of elements in a set must be less than 65536.

Div operator (6.7.2.2)

The following axiom is obeyed:

$$\text{abs}(a \text{ div } b) = \text{abs}(a) \text{ div } \text{abs}(b)$$

INTRODUCTION (COMPLIANCE STATEMENT)

Implementation-dependent features (See 3, 5.1 and 5.2)

These features are similar to implementation-defined but need not have an interpretation at all (in other words, be prohibited) on a particular processor. Standard-conforming programs should not use them according to 5.2.

Directives (6.6.1 and 6.6.2)

Only the directives forward and external are permitted. (Note: it is thought that forward should be standard, and only other directives are implementation dependent.)

Put procedure (6.6.5.2)

The put procedure will fail in execution if applied to a file in readstate. An error will be reported.

Standard procedures (6.6.5 and 6.6.6)

Some standard procedures and functions are permitted as procedural or functional parameters. See the manual for details.

Evaluation order of operands (6.7.2)

The operands of binary operators are always evaluated in left-to-right order.

Boolean expressions (6.7.2.3)

All components of a boolean expression are always evaluated.

Binding of parameters (6.7.3 and 6.8.2.3)

Binding (the identification of the object involved) takes place in strict left-to-right order. Scalar, real, pointer, and set expressions corresponding to value parameters are copied immediately after they are bound. Array and record parameter copying corresponding to value parameters are deferred, and the copying takes place after the call is initiated in left-to-right order of the deferred values.

Assignment statements (6.8.2.2)

Binding of the variable on the left-hand-side of an assignment always precedes evaluation of the right-hand-side expression.

Reset and rewrite (6.10)

Reset and rewrite are permitted on the standard files input and output.

INTRODUCTION (COMPLIANCE STATEMENT)

Error handling (See 5.1.1(c))

Access to variant with wrong real or virtual tagfield (6.4.3.3)
Not detected.

Subrange errors in assignment compatibility (6.4.6)
Detected during compilation if a constant, otherwise detected during execution.

Dereferencing nil pointer (6.5.4)
Detected in execution by INVALID INDEX interrupt.

Dereferencing undefined pointer (6.5.4)
Detected in execution if pointer has tag six value by INVALID OPERAND interrupt.

Using put while eof false (6.6.5.2)
Detected in execution.

Using get while eof true (6.6.5.2)
Detected in execution.

Aliasing with file and file-buffer (6.6.5.2)
Aliasing errors arising from binding of the file buffer are not detected.

Dispose with nil parameter (6.6.5.3)
Dispose implemented but always returns a nil pointer.

Dispose with bound pointer (6.6.5.3)
Dispose implemented but always returns a nil pointer.

Assignment of dynamic variable created with tags (6.6.5.3)
Not detected except in unusual circumstances.

Error in ln(x) (6.6.6.2)
Detected in execution by Burroughs intrinsic procedure.

Error in sqrt(x) (6.6.6.2)
Detected in execution by Burroughs intrinsic procedure.

Trunc and round with non-integer result (6.6.6.3)
Detected in execution by INTEGER OVERFLOW interrupt.

Error in chr (6.6.6.4)
Detected in execution.

INTRODUCTION (COMPLIANCE STATEMENT)

Error in succ and pred (6.6.6.4)

Detected in execution.

Undefined values (6.7.1)

The attempted use of any undefined value which has acquired the tagsix value is detected by the INVALID OPERAND interrupt. See later for an analysis of undefinition.

Set value outside limits (6.7.1)

Detected during compilation if a constant, otherwise detected during execution.

Divide by zero (6.7.2.2)

Detected in execution by the DIVIDE BY ZERO interrupt.

Integer range trespass (6.7.2.2)

If the result of an integer operation exceeds the integer range, the value automatically becomes real. However, at assignment compatibility points, a check is applied which gives rise to the INTEGER OVERFLOW interrupt if the value is non-integer.

Goto into structured statement (6.8.2.4)

Not detected.

Case expression without label (6.8.3.5)

Detected in execution.

Altering for-index (6.8.3.9)

Blatant attempts detected in compilation and treated as errors. Possible attempts (use as actual variable parameter) cause compile-time warnings. Sneaky attempts will be detected in execution if the loop is optimized.

Syntax of real and integer on input file (6.9.2)

Detected in execution.

Undefined

Many errors are traceable to undefined values: this section explains the treatment of undefined by this compiler.

Local variables (6.2)

Scalar, real, pointer, and set variables are set to a special undefined value (tagsix) at the beginning of the statement part. Records and arrays acquire all-zero binary values. Files acquire a value or not depending on their attributes (extension).

Change of variant (6.4.3.3)

No changes are made when variants are selected. The fields retain their original binary values.

Function values (6.6.2)

The function value is initialized to a special undefined value (tagsix) at the beginning of the statement part. If the value is not overwritten by a function assignment, an interrupt occurs at exit.

File buffer (6.6.5.2)

The file buffer is not altered from its current value under these conditions of undefined.

Dispose (6.6.5.3)

Dispose implemented but always returns a nil pointer.

For index (6.8.3.9)

Always acquires a special undefined value (tagsix) at exit.

INTRODUCTION (COMPLIANCE STATEMENT)

Extensions to Standard Pascal

(See 5.1)

These are more fully described in the manual. The compiler option STANDARD enables a checking which flags use of these extensions in general.

1. The provision of file attribute declarations.
2. The provision of type transfers from integer to scalar type (inverse of ord).
3. The provision of a format declaration and record-oriented read and write statements.
4. The provision of random-access (relative-indexed) reading and writing.
5. The provision of extra pre-defined procedures and functions.
6. Allowing external files to be attached to inner procedures or functions without attachment to the main program.
7. Allowing strings to use the double quote as an alternative to the single quote for Algol compatibility.
8. The lexical alternatives @, (*, *) are permitted for use on devices which do not support {, }.
9. Allowing a % to end-of-line comment form.
10. The permitting of an otherwise clause in case statements.
11. Allowing external procedures or functions to be declared within a program.

INTRODUCTION (COMPLIANCE STATEMENT)

Deviations from Standard Pascal

These are more fully described in the manual, and represent places where the processor does not conform to the requirements of section 6 of the Standard.

1. Files may not be components of any structured type.
2. Program parameters are permitted, but have no effect.

INTRODUCTION

INTRODUCTION TO THE MANUAL

Burroughs Algol programmers should find little difficulty in writing Pascal programs which are almost isomorphic to the Algol ones they presently write; experience will allow transition to better structured code as the concepts of data-structures become more understood. FORTRAN programmers will find more difficulty as the control structures are also less familiar; PL/I programmers will be amazed at the simplicity and power of the Pascal language compared to PL/I.

The rest of this document discusses the components and structures of the B6700/B7700 Pascal language, categorized into categories that seem appropriate. These categories, by section, are:

- Lexical tokens
(the words of Pascal)
- Subcomponents
(bits and pieces otherwise unclassifiable)
- Declarations
(the objects and concepts of Pascal and stating them)
- Statements
(the executable commands of Pascal)
- Program units
(constructing wholly executable programs)
- Pre-defined procedures
(procedures available without declaration)
- I/O
(the input/output system of B6700/B7700 Pascal)
- Options
(how to manipulate the compiler options and their effects)
- Compiler files
(the definitions of the compiler's file attachments)
- Errors
(the interpretation of error situations)
- Sample programs
(to illustrate the language and the listings produced)
- General
(which cannot be classified elsewhere)

This manual was produced using the RUNOFF text editing system and printed on a Diablo 1620 terminal.

References

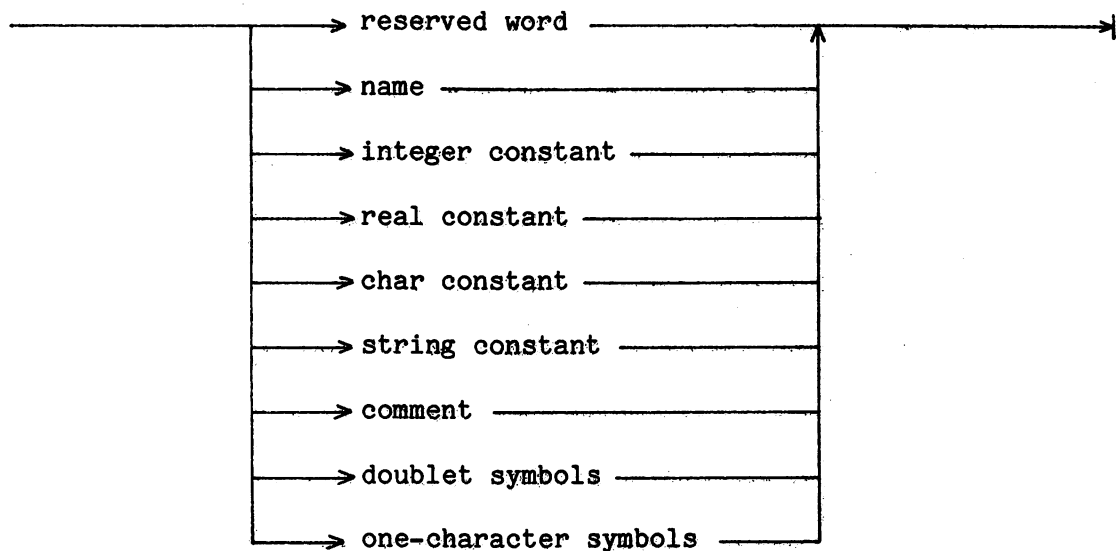
- Addyman, A: The BSI/ISO Working Draft of Standard Pascal by the BSI DPS/13/4 Working Group, Pascal News, Number 14, pp 9-54; see also Draft Proposal ISO/TC97/SC5 DP7185
- Jensen, K and Wirth, N (1974): "PASCAL User Manual and Report", Notes in Computer Science Series, No.18, Springer-Verlag.
- Wirth, N (1973): "Systematic Programming", Prentice-Hall.
- Welsh, J. (1978): "Economic Range Checks in Pascal", Software - Practice and Experience, vol. 8, p 85-97.

2. LEXICAL TOKENS

LEXICAL TOKENS

Syntax

lexical token



Semantics

The formation of the lexical tokens is explained in the succeeding pages. Lexical tokens in B6700/B7700 Pascal are formed from characters in the EBCDIC character set. All lexical tokens must be contained wholly on a single line of the source text and may not contain any embedded space characters. Except within string and char constants, and within comments, the space character serves to delimit adjacent tokens but has no other meaning.

CHAR CONSTANT

CHAR CONSTANT

Syntax

char constant

—————→ ' → character → ' —————→

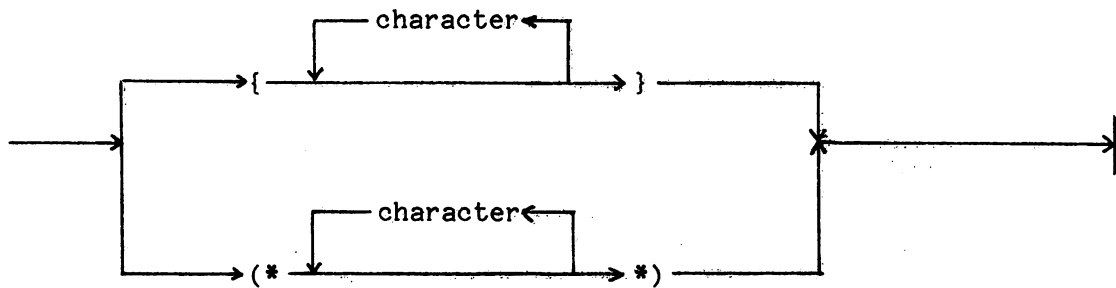
Semantics

A char constant defines a constant of the pre-defined Pascal type char. In each case above, the enclosed character may be any legal character except the quote symbol used to delimit the token.

The internal representation of the graphic used in a char constant is normally an 8-bit EBCDIC value. However, if the ASCII compiler option is set when a char constant is compiled, the value is represented internally in the ASCII code. This will affect the internal collating sequence and the result returned by the ORD and CHR functions. If a string delimiter is to appear as a char constant then that character is written twice. Thus ''' contains the character '.

COMMENTSyntax

comment

Semantics

A comment has no effect on the compilation or execution of a Pascal program except for a role in delimiting other tokens. The two forms of comment are equivalent to a space character. Comments may therefore be used wherever a space may be used, except within string constants or format lists.

The purpose of a comment is to introduce information for human readers of the program; therefore any character may be used in the body of the comment except for the symbol that terminates it.

If a closing marker is omitted by mistake, following text will not be compiled and is treated as commentary until another comment is reached. To detect this situation in a large number of cases, a warning message is issued if a semicolon is encountered in these comment forms. The message may be suppressed by resetting the WARNINGS compiler option.

DOUBLET SYMBOLS

DOUBLET SYMBOLS

Syntax

doublet symbol	token-name
:=	becomes-token
..	subrange-token
<>	not-equal-to
<=	less-or-equal
>=	greater-or-equal

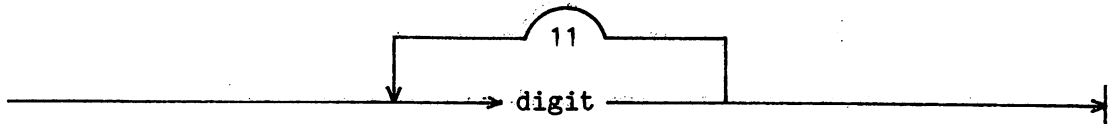
Semantics

Tokens composed of two adjacent characters are used in Pascal to augment the basic character set and to construct extra tokens. The use of these tokens in the language will be described in later sections. Note that the pair of characters must be immediately adjacent to be recognized as a doublet symbol; if a space separates them the characters are recognized as separate tokens.

INTEGER CONSTANT

Syntax

integer constant



Semantics

An integer constant is represented internally in a B6700/B7700 Pascal program by a value of type integer. The external form is written as a sequence of decimal digits (0123456789) and converted according to the usual rules. A valid integer must have no more than 12 digits (including any leading zeros), and must be less than 549755813887 since this is the largest representable integer in the B6700/B7700 computers. The predefined constant, MAXINT, represents the largest representable integer in the B6700/B7700 computers.

ONE-CHARACTER SYMBOLS

ONE-CHARACTER SYMBOLS

Syntax

symbol	token-name	equivalent
+	plus	
-	minus	
*	times	
/	divide	
=	equalto	
<	less-than	
>	greater-than	
(left-parenthesis	
)	right-parenthesis	
[left-bracket	
]	right-bracket	
.	point	
,	comma	
:	colon	
;	semicolon	
↑	at (see note)	@ or ^

Semantics

The use of these tokens will be explained in later sections. If the B6700/B7700 Pascal compiler encounters a character outside the context of the other tokens which is not one of these characters (for example the &-character), a lexical error is reported.

Note:

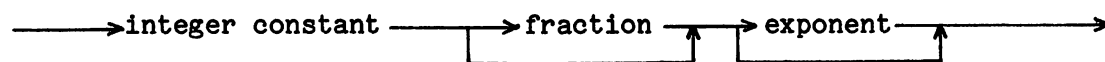
The ↑-character may not be available on all devices on a B6700/B7700 system (as it is not a common graphic) and the use of the @-character is provided as an alternative. On some devices the ↑-character masquerades as the ↗-character, or prints as a ^ .

REAL CONSTANT

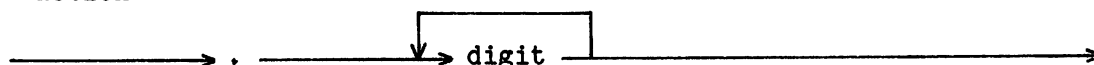
REAL CONSTANT

Syntax

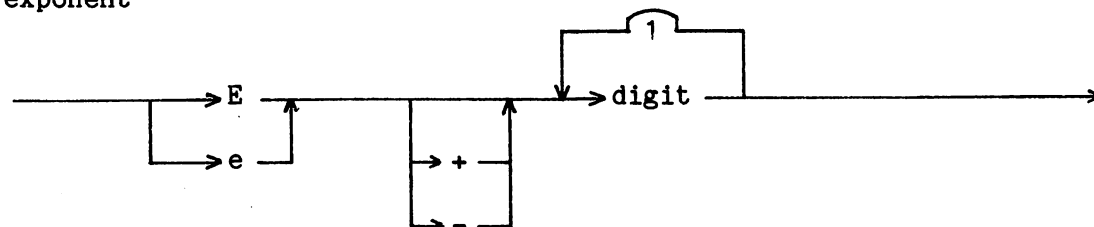
real constant



fraction



exponent



Semantics

A real constant is represented internally in a B6700/B7700 Pascal program as a read-only value of type real. The value of the real constant must lie in the representable range of the B6700/B7700 computers:

between 8.75811540203E-47 (8**-51)
and 4.31359146674E+68 (8**76 - 8**63)

or may be exactly zero. The fraction part may have any number of digits, up to the limit imposed by the line length, but only the first 23 are used in the conversion. A fraction written with a large number of fractional leading zeros may therefore be inaccurately converted. The exponent part is a scale factor expressed as a power of 10, and may have one or two digits.

Note:

An integer constant is a valid real constant. If a real constant is in fact an integer, it may lead to more efficient code if it is written as such without a fraction or exponent.

RESERVED WORDSSyntax

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

RESERVED WORDS

Semantics

The use of reserved words is described in later sections. The reserved words are absolutely reserved: they may not be used as names elsewhere in a B6700/B7700 program since they will always be recognized as reserved words. The reserved words are recognized according to the rules for names: they may appear in the source text in upper-case letters, or lower-case letters, or a mixture of both. BEGIN, Begin and begin are all recognized as the reserved word BEGIN.

The reserved word FORMAT has no counterpart in standard Pascal; the word NEQ is provided as an Algol-compatible equivalent for <> (not-equal-to). The word PROGRAM is treated as fully synonymous with the word PROCEDURE.

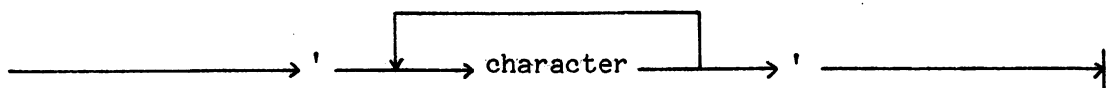
Standards

In B6700/B7700 Pascal, FORWARD and EXTERNAL are a reserved words, and may not be redefined by a programmer. This, however, is not standard Pascal, although forward declarations are permitted.

NIL is included here as a reserved word as specified by the Pascal Standard. However, in B6700/B7700 Pascal, NIL is not a reserved word, but a predefined name (as are TRUE and FALSE). Programmers may redefine these names if they wish, however, this is not recommended.

STRING CONSTANTSyntax

string constant

Semantics

A string constant defines an object which can be used in Pascal as a read-only packed array of char. Any legal characters may appear in the internal part of the string constant except the character used to delimit the token.

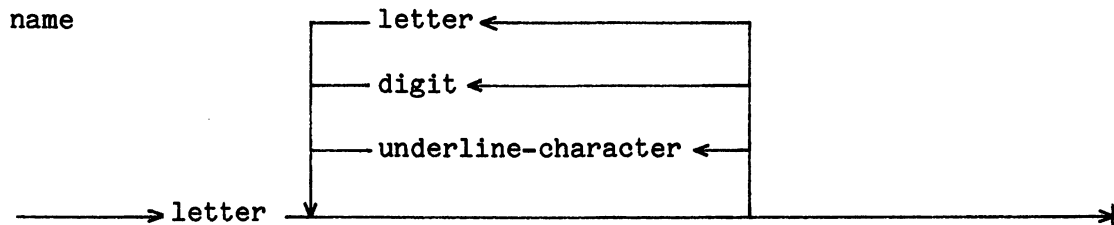
The maximum length of a string constant is 70 characters, and is possible only if the string constant occupies the whole of a source line. The minimum length is 2 characters, as a 1-character string is regarded as a char constant.

The internal representation of the graphics used in a string constant is normally in 8-bit EBCDIC values. However, if the ASCII compiler option is set when a string constant is compiled, the graphics are represented internally in the ASCII code. This will affect the internal collating sequence and the result returned by the ORD and CHR functions. If a string delimiter is to appear within a string with that delimiter then the character is written twice. Thus the string constant 'DON'T' contains the characters DON'T.

NAMES

NAMES

Syntax



Semantics

Names are used to identify Pascal objects, apart from labels. In the above syntax, a letter means any alphabetical character in either uppercase (A to Z) or lower-case (a to z); a digit means a decimal digit (0 to 9); and the underline-character means an underlined space. Any length name is permitted up to the limit imposed by the line length and all characters of names are significant in distinguishing names. However, for the purposes of naming, a lower-case letter and an upper-case letter are regarded as equivalent. Thus the name FRED is the same as the name Fred. Names are held internally in the compiler in upper-case form and any compiler-produced name-tables, etc., use this canonical form for printing. For compatibility with other Pascal compilers, programmers should consistently use either upper-case or lower-case.

A programmer-defined name may not be the same as any reserved word.

Examples

J

THING

temperatureofkiln

PAINT_MIXTURE_FOR_PAINTING_THE_KITCHEN_WITH_ON_SUNDAY

disaster_point

WITH2PARAMETERS

PartNo4536Z

CourseSIS102H

Note

Some other compilers for Pascal only treat the first 8 characters of a name as significant. This should be borne in mind if compatibility with other compilers is important. The use of two cases of letters, and of the underline character, should also be avoided in these circumstances. See the use of the compiler option 'STANDARD'.

3. SUBCOMPONENTSSUBCOMPONENTSExplanation

Some constructs appear in the B6700/B7700 Pascal language in several contexts. Rather than define the constructs in the main part of the manual, they are defined here as subcomponents of the language: comprised of lexical tokens but not major components of the language such as statements or declarations.

The subcomponents described are:

- signed integer
- expression
- name list
- parameter list
- subrange
- scalar range
- set constructor
- labels
- variables

Also discussed are:

- operators
- type identity
- type compatibility
- assignment compatibility
- scope

ASSIGNMENT COMPATIBILITY

ASSIGNMENT COMPATIBILITY

Semantics

Compatibility is not expressed in the B6700/B7700 Pascal language, but is a notion used to test whether an assignment or type association is semantically meaningful.

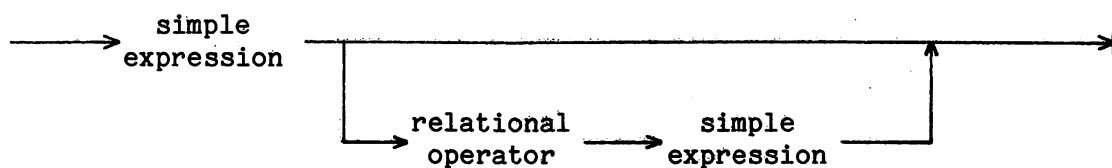
An expression E of type T2 is assignment-compatible with a type T1 if any of the four statements which follow is true.

1. T1 and T2 are identical and neither is a file-type nor a structured-type with a file component.
2. T1 is a real-type and T2 is integer.
3. T1 and T2 are compatible ordinal-types and the value of E is in the closed interval specified by the type T1.
4. T1 and T2 are string types with the same number of components.

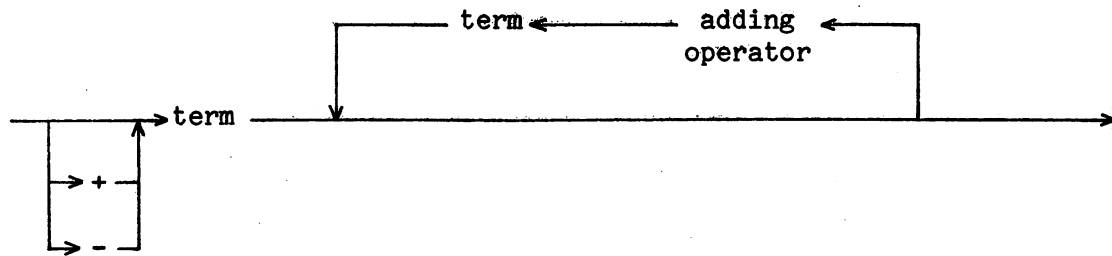
EXPRESSION

Syntax

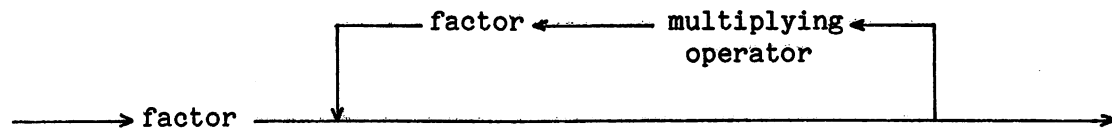
expression



simple expression

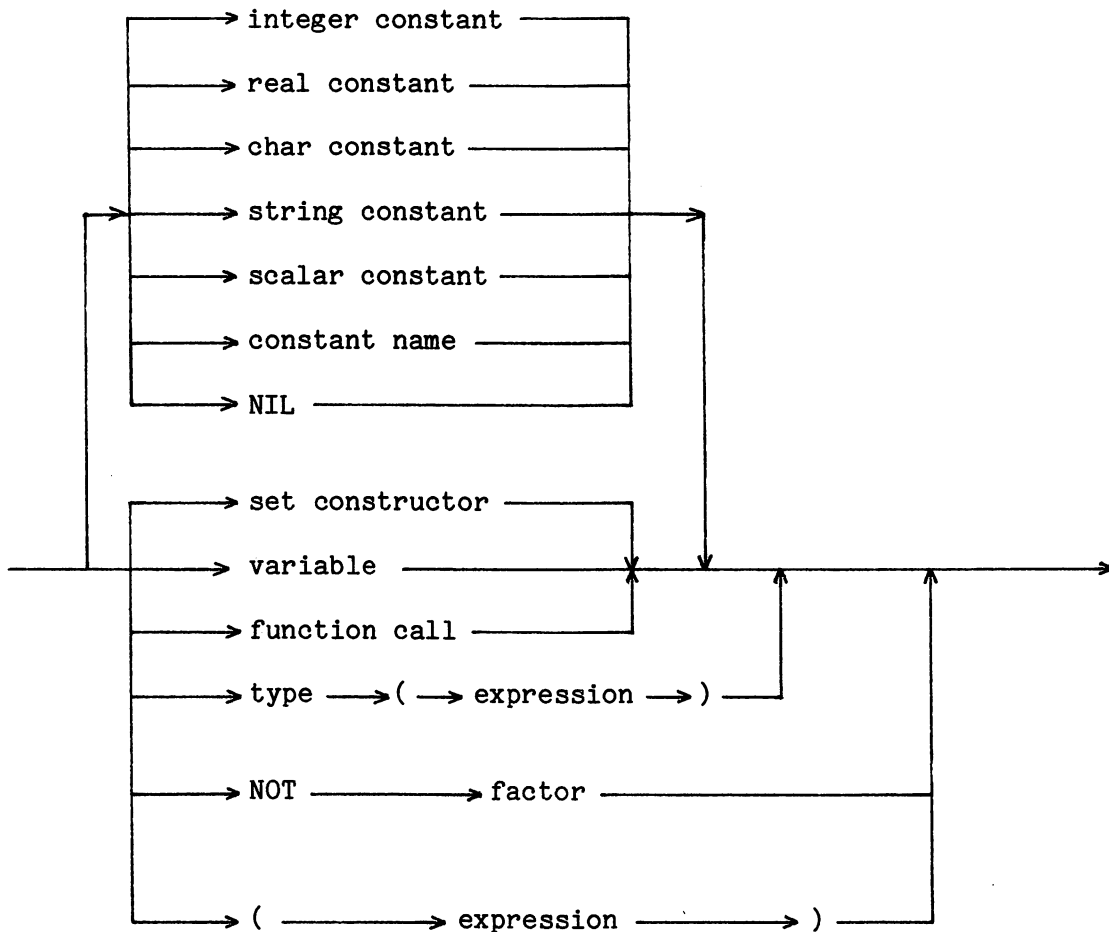


term



EXPRESSION

factor



Semantics

An expression is a construct denoting a computation for deriving a value from variables and constants by the application of operators. Expressions consist of operands (objects having value such as variables and constants), operators (rules for computation), and some structuring tokens (parentheses). An error occurs if any variable, or function used as an operand in an expression has an undefined value at the time of its use.

The operators are applied according to rules of precedence, according to four classes of operators. The operator NOT has the highest precedence, followed by the 'multiplying' operators, then the 'adding' operators and signs, and finally the relational operators.

PRECEDENCE ORDER OF OPERATORS

NOT

* / DIV MOD AND (multiplying operators)

+ - OR (adding operators)

> = < >= <= <> (relational operators)

The higher precedence operators are applied before any of lower precedence. These notions are implicit in the syntax charts given. Sequences of operators of the same precedence are executed from left-to-right. In all expressions, including boolean expressions, all terms and factors are evaluated.

Expressions which are members of a set are of identical type. [] denotes the empty set which belongs to every set type. The set [x..y] denotes the set of all values of the base type in the closed interval x to y. If x is greater than y then [x..y] denotes the empty set.

The type of an expression may be altered by specifying the type name followed by the expression enclosed in parentheses. The bounds of the type are checked and an error results if the bounds are exceeded. The types INTEGER and REAL may not be used in this manner.

A legal expression in B6700/B7700 Pascal must comply with the type and compatibility rules as well as the syntax given. The operators are defined only over certain types and return values of particular types; these are detailed in the sheets on operators. The requirements for further compatibility are given in the sheets under that title.

EXPRESSION

Examples

FACTORS:

X
15
(X + Y + Z)
SIN(X + Y)
[RED,C,GREEN]
[1,5,10..19,23]
NOT P

TERMS:

X*Y
I * J + 2
(X <= Y) AND (Y < Z)

SIMPLE EXPRESSION:

X + Y
-X
P OR Q
HUE1 + HUE2
I * J + 1

EXPRESSIONS:

X = 1.5
P <= Q
P = Q AND R
(I < J) = (J < K)
CR IN [RED,GREEN]

Standards

Some other Pascal compilers implement boolean expressions by selective evaluation (sequential conjunction or disjunction); this may pose some problems for programs imported into a B6700/B7700 environment but will not be likely to affect the portability of exported programs except for rare cases. All programs affected are non-standard.

The precedence rules for boolean expressions give the effect that:

$$a > 0 \text{ and } b < 10$$

is illegal since it is parsed:

$$a > (0 \text{ and } b) < 10$$

the correct expression is of course:

$$(a > 0) \text{ and } (b < 10)$$

Since a few compilers do not conform to the standard Pascal precedence rules, it is recommended that expressions involving boolean operands be fully parenthesized, especially if relational operators are used between booleans; for example:

$$a = b \text{ and } c$$

Changing type by using the type-name in a function-like usage is not standard Pascal. Only the ORD, TRUNC and ROUND functions (see functions) are allowed in standard Pascal.

LABELS

LABEL

Syntax

label

—————→ integer constant —————→

Semantics

Labels are used to mark places in the executable body of a program, procedure or function, so that the goto statement can utilize them. Further references will be found under goto statement, label declaration, and statement.

A valid B6700/B7700 Pascal label has a corresponding numeric value from 0 to 9999 inclusive. The numeric value is not important, except for establishing correspondence between usages of labels.

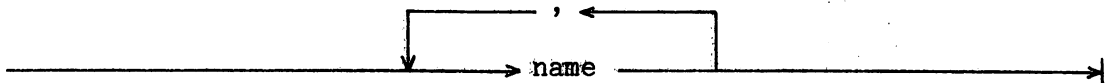
Examples

1

7876

NAME LISTSyntax

name list

Semantics

A name list consists of one or more names, separated by commas. It occurs in several forms of declaration.

Examples

REDCOLOUR

RED,BLUE,YELLOW,GREEN,PURPLE

X,Y,Z

OPERATORS (ARITHMETIC)

ARITHMETIC OPERATORS

Binary

operator	operation	type of operands	type of result
+	addition	integer or real	integer or real
-	subtraction	integer or real	integer or real
*	multiplication	integer or real	integer or real
/	division	integer or real	real
DIV	division with truncation	integer	integer
MOD	modulo	integer	integer

Unary

operator	operation	type of operands	type of result
+	identity	integer or real	integer or real
-	sign-inversion	integer or real	integer or real

Semantics

If both the operands of the addition, subtraction or multiplication operators are of the type integer, then the result is of the type integer otherwise the result is of the type real. If the operand of the identity or sign-inversion operators is of the type integer then the result is of the type integer otherwise the result is of the type real.

The value of $i \text{ div } j$ is such that:

$$\text{abs}(i \text{ div } j) = (\text{abs}(i)) \text{ div } (\text{abs}(j))$$

Clearly, if $j = 0$ then an error occurs.

The value of $i \text{ mod } j$ is such that:

$$i = (j * \text{quotient}) + \text{remainder}$$

$$\text{where } 0 \leq \text{abs}(\text{remainder}) < \text{abs}(j)$$

$$\text{and } \text{sign}(i) = \text{sign}(\text{remainder})$$

{See Standards below}

The predefined constant maxint is of type integer and has an implementation defined value of 549755813887. This value satisfies the following conditions:

1. All integral values in the closed interval from $-\text{maxint}$ to $+\text{maxint}$ are representable in the integer type.
2. Any unary operation performed on an integer value in the above interval is correctly performed according to the mathematical rules for integer arithmetic.
3. Any binary integer operation on two integer values in the above interval is correctly performed according to the mathematical rules for integer arithmetic. The result of an intermediate calculation in an expression may temporarily exceed the interval above (when it is converted to a real value, with consequent loss of exactness). If the final result, however, is outside the interval, the B6700/B7700 will interrupt and terminate the program's execution.
4. Any relational operation on two integer values in the above interval is correctly performed according to the mathematical rules for integers.

OPERATORS (ARITHMETIC)

Standards

The Pascal standard defines the value of $i \text{ div } j$ to be such that

$$i - j < (i \text{ div } j) * j \leq i$$

where $i \geq 0$ and $j > 0$; an error occurs if $j = 0$. Other Pascal compilers may produce different results for $i < 0$ and/or $j < 0$.

Also the value of $i \text{ mod } j$ is defined to be the value of

$$i - (i \text{ div } j) * j$$

The result for negative operands is dependent on the method of implementation of the div operator, and care should be taken if compatibility with other Pascal compilers is required. The B6700/B7700 Pascal system satisfies this constraint.

BOOLEAN OPERATORS

operator	operation	type of operands	type of result
OR	logical 'or'	boolean	boolean
AND	logical 'and'	boolean	boolean
NOT	logical negation	boolean	boolean

Semantics

Boolean expressions are completely evaluated in B6700/B7700 Pascal. The sheets on expressions give a fuller discussion of boolean expressions.

OPERATORS (SET AND RELATIONAL)

SET OPERATORS

operator	operation	type of operands	type of result
+	set union	any set type T	T
-	set difference	any set type T	T
*	set intersection	any set type T	T

RELATIONAL OPERATORS

operator	type of operands	type of result
= <>	any set, simple, pointer or string type	boolean
< >	any simple or string type	boolean
<= >=	any set, simple or string type	boolean
IN	left operand any ordinal type T right operand SET OF T	boolean

Semantics

Except when applied to sets, the operators <> , <= , >= stand for not equal, less than or equal, and greater than or equal respectively.

The operands of = , <> , < , > , >= and <= are either of compatible type or one operand is real and the other integer.

If u and v are set operands, u <= v denotes the inclusion of u in v and u >= v denotes the inclusion of v in u.

Since type Boolean is an ordinal type with false < true, then if p and q are Boolean operands, p = q denotes their equivalence and p <= q denotes the implication of q by p.

When the relational operators + , <> , < , > , <= , >= are used to compare strings, they denote lexicographic ordering according to the ordering of the character set.

OPERATORS (SET AND RELATIONAL)

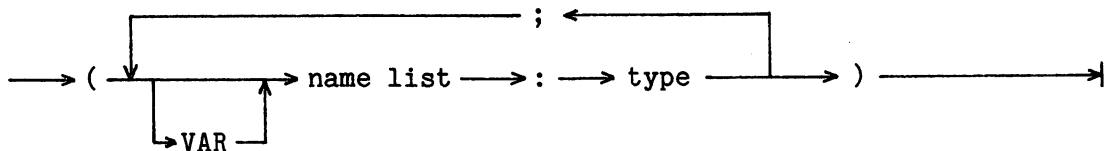
The operator IN yields the value true if the value of the operand of ordinal-type is a member of the set, otherwise it yields the value false. In particular, if the value of the operand of ordinal-type is outside the bounds of the set, the operator IN will yield the value false.

PARAMETER LIST

PARAMETER LIST

Syntax

parameter list



Semantics

A parameterlist defines objects which are accessible within the body of the procedure or function to which they are attached, and which have some attributes which are imported from outside the procedure or function when it is invoked. The form of the parameterlist serves to:

- * identify the names of the objects,
- * their types, and
- * whether they are VAR or 'value' parameters.

If a parameter is not preceded by the reserved word VAR, it is regarded as the default parameter type: a value parameter. The object is then identical to a locally declared object of the same type, except that it is initialized at the time of procedure or function invocation to the value given by the corresponding actual parameter. Any changes to the value of this object will therefore not have any effect on objects declared outside the procedure. A file may not be a 'value' parameter; neither may a procedure or function passed in to another procedure.

If a parameter is preceded by the reserved word VAR, the object accessible within the procedure or function is the actual outside object referenced in the invocation of the procedure or function, viewed through the window of the parameter list specification. The parameter attachment is implemented by a mechanism which is equivalent to a "reference parameter". Any Pascal object may be a VAR parameter, including a file.

Examples

```
(X,Y: REAL)

(VAR I: INTEGER; BFLAG: BOOLEAN)

(VAR FILEX: FILE_OF_SECTOR;
 INDEX: INTEGER;
 VAR MASK,NEWSECTOR: SECTOR)
```

Efficiency

It is marginally more efficient to access a Pascal object which occupies a single B6700/B7700 word by the value mechanism. Since this also protects exterior objects from alteration inadvertently, the default value parameter mechanism should be used for all such objects unless it is explicitly desired to alter the external object passed through the parameter list. This advice applies to all scalar types including integer, char and boolean, to type real, to all pointer types, and to all set types restricted to fewer than 49 members.

Using the value parameter mechanism to pass objects of type record or array will cause the compiler to request the allocation of extra memory to hold a duplicate copy of the entire record or array, and to initialize that array to be such a copy at procedure entry time. This is expensive in space and may be expensive in time if few accesses are made to the parameter within the procedure or function. It is therefore recommended that such parameters normally be passed by the VAR mechanism, except in two cases:

- (1) When a local copy of the record or array is explicitly needed, and no external modification is required. The default value mechanism will provide this.
- (2) When the efficiency of access to the object is critical, and yet external changes are necessary. In this case, make a call by reference (VAR), but insert declarations and code to make a local copy and restore the copy to the external world. In general this will only be faster if the number of references to the object exceeds (4 x the number of words in the object), supposing that single-word references are made.

PARAMETER LIST

Standards

Both parameter mechanisms comply with the requirements of standard Pascal. Programmers writing programs that may be used with other Pascal compilers, or which are derived from installations with other Pascal compilers, should be aware that the VAR parameter passing mechanism is not always implemented by a 'reference' mechanism as in B6700/B7700 Pascal, but sometimes by a mechanism called 'value/recopy'. In this mechanism, a local object is created in the procedure or function just as for the value call, but at the termination of the procedure the copy is recopied back into the external object. The difference between this and the reference mechanism cannot be detected if the external object is not referenced (other than through the parameter) between the invocation of the procedure/function and its termination, and provided no gotos across procedure levels are executed.

Programmers writing code which is intended to be portable across Pascal compilers should therefore avoid accessing global objects which are also referenced through parameters, and avoid referencing the same object in two VAR parameters.

Programmers who receive programs which may contain machine dependencies due to the use of value/recopy as a parameter mechanism can achieve the required effect by explicitly writing in the copying required. The code thereby generated is as efficient (to about 2 instructions) as if it had been implemented by the compiler. This is the same advice as is given for case (2) under the efficiency subheading. An example:

```
PROCEDURE Z(VAR A: ARRAYTYPE);
  VAR LOCALA: ARRAYTYPE;
  ...

BEGIN
  LOCALA:=A;      {the value copy}
  ...

  {the body of the procedure}
  ...

  A:=LOCALA;     {the re-copy}

END;             {of Z}
```

SCALAR RANGESyntax

scalar range

—————→ (—————→ name list —————→) —————→

Semantics

The scalar range construct is used to define the values of a programmer defined scalar type. The values are externally represented by the names listed between the parentheses. The relational operators (less, equal, etc.) are defined between these values assuming them mapped one-for-one with the natural numbers (0, 1, 2, ...) in enumeration order.

The scalar range construct is used in declarations.

Examples

(RED,BLUE,YELLOW,GREEN,PURPLE)

(FALSE,TRUE)

(YES,NO,MAYBE)

SCOPE

SCOPE

Semantics

Scope is not expressed in B6700/B7700 Pascal, but is a notion used to determine the range for which an identifier or label is defined. The concept and meaning of scope is described in the following six paragraphs.

1. Each identifier or label within a Pascal program has a defining occurrence. Associated with each defining occurrence is a scope which is the range in the program text for which that defining occurrence holds. Each identifier or label may have one and only one association in each scope.
2. In the case of identifiers or labels whose defining occurrence is within a block of a Pascal program, or identifiers whose defining occurrence is in a formal-parameter-list associated with a block, the scope extends from the commencement of the formal-parameter-part (if it exists) or the commencement of the block otherwise, to the closing "end" of the block, subject to (3) and (4) below.
3. When an identifier or label which has a defining occurrence for range A has a further defining occurrence for some range B enclosed by A, then range B and all ranges enclosed by B are excluded from the scope of the defining occurrence for range A.
4. An identifier which is a field identifier may be used as a field identifier within a field-designator in any range in which a variable of the corresponding record-type is accessible.
5. The scope of identifiers which are field-identifiers, and whose defining occurrence as variable-identifiers occurs as a result of the execution of a with-statement, extends over the internal statement of the with-statement.
6. The defining occurrence of an identifier or label precedes all corresponding occurrences of that identifier or label in the program text with one exception, namely that a type-identifier T, which specifies the domain of a pointer-type \hat{T} , is permitted to have its defining occurrence anywhere in the type-definition-part in which \hat{T} occurs.

Note

The definition of a constant-identifier takes place at the end of its constant definition; consequently a constant-identifier may not be used in its own definition.

Similarly, the definition of a type-identifier takes place at the end of its type-definition, except for pointer-types (see 6 above); consequently a type-identifier may not be used in its own definition with this exception.

Standards

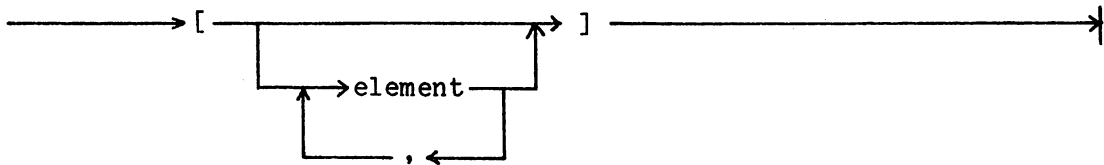
These semantics conform to those of the Pascal Standard.

SET CONSTRUCTOR

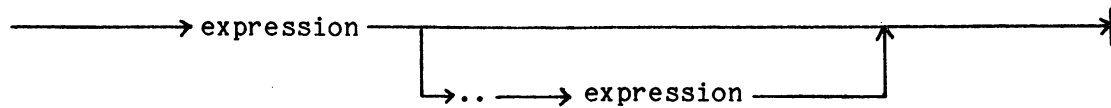
SET CONSTRUCTOR

Syntax

set



element



Semantics

The set constructor contains expressions and subranges of constants and represents a set containing as members the values so expressed. The types of the expressions and bounds of the subranges must be of identical type. The empty set is represented by the construct []. When the expressions are of type integer, the compiler option SETSIZE determines the type of set produced by this construction. If SETSIZE is 48 or less then a one word set is produced otherwise a set with bounds 0 and (SETSIZE-1) is constructed.

If all the values in the set constructor are constants, and the value of SETSIZE is less than 49, then the set functions as a set constant. In all other cases, code is inserted in the program to construct the set.

If a subrange construct is used and the lower bound is greater than the upper bound the subrange functions as the empty set. A warning is produced for this occurrence.

Examples

[RED, GREEN]

[0 .. 6, 9]

[YES, PERHAPS, MAYBE]

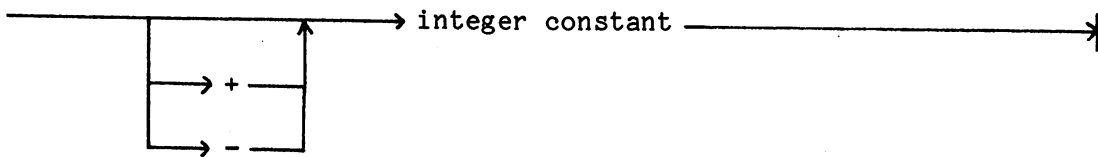
[]

SIGNED INTEGER

SIGNED INTEGER

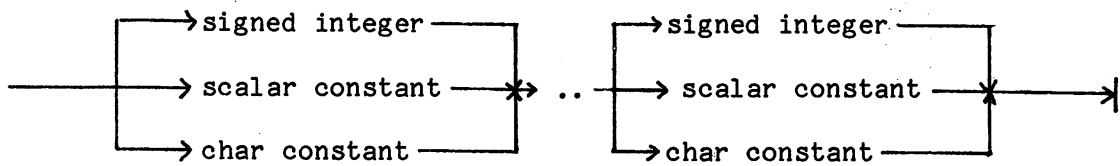
Syntax

signed integer



Semantics

A signed integer value represents a Pascal value of the predefined type integer. It is valid in several contexts, notably in defining constant names in a CONST declaration, and in subranges. Any valid integer constant may be signed: the representable range is symmetrical about zero on the B6700/B7700 computers. Both +0 and -0 are regarded as arithmetically equal.

SUBRANGESyntaxSemantics

The subrange defines a subset of a scalar type from the first value given up to and including the second value given. The two bounds of the subrange must be of identical type. A scalar constant is a name declared as such in a scalar declaration occurring in a TYPE or VAR declaration, or a name equated to such a name in a CONST declaration.

A subrange may occur in a declaration, or in a set-constructor.

Examples

-1 .. 99

RED .. GREEN

5 .. 7

TYPE COMPATIBILITY

TYPE COMPATIBILITY

Semantics

Compatibility is not expressed in B6700/B7700 Pascal, but is a notion which is used to determine whether an operator or parameter linkage is semantically meaningful.

Two types are compatible if they are identical, or if one is a subrange of the other, or if both are subranges of the same type, or if they are string types with the same number of components (or in the case of assignment, if the lefthand side has a larger number of components, in which case the components not assigned a value are blank filled. The assignment is left justified.), or if they are set types of compatible base types.

Standards

Pascal compilers that conform to the standard, will only allow compatibility between string types with the same number of components. Programmers should be aware of this if portability is required.

TYPE IDENTITYSemantics

Types which are designated at two or more different places in the program are identical if the same type identifier is used at these places, or if different identifiers are used which have been defined to be equivalent to each other by type definitions of the form TYPE1 = TYPE2.

Standards

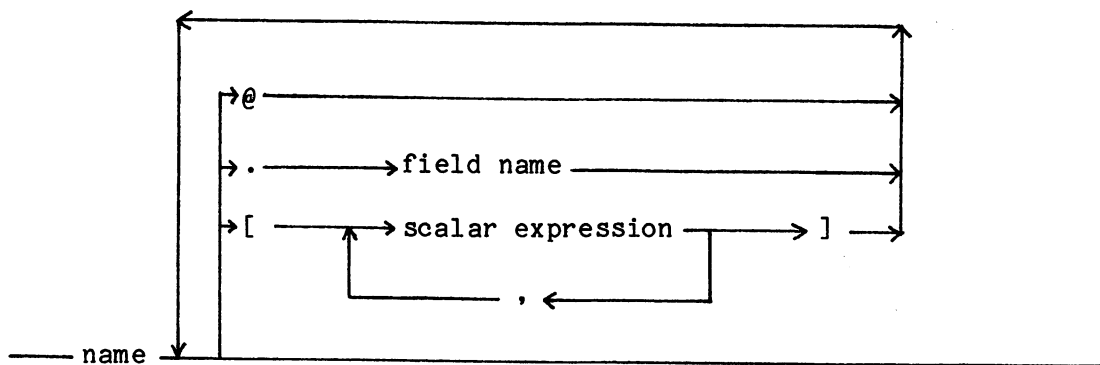
These semantics conform exactly with the Pascal Standard.

VARIABLE

VARIABLE

Syntax

variable



Semantics

A variable is a rule for determining a reference to a Pascal object. It may be used in an expression to determine a value, but it may be used in other contexts where the identification of the variable is the prime purpose, not its value. Examples are procedure invocation parameter lists, assignment statements, and for statements.

The form with @ (or $\hat{\quad}$) is valid only if the preceding variable part has evaluated to filetype (when the new form references the file buffer), or if it has evaluated to a pointer type (when the new form references an object of the pointer's bound type).

The form with . is valid only if the preceding variable part has evaluated to a record type, and the field name is a field name of the record type. It selects the nominated field of the record and gives a corresponding type.

The form with square brackets is valid only if the preceding variable part has evaluated to an array type. The type and number of the scalar expressions within the brackets must correspond to declaration of the array type. The effect is to select one component of the array as determined by the values of the subscript expressions; the type is of course that of the array components.

Examples

X

PERSON.AGE

CLASS[MEMBER].PTR@.AGE

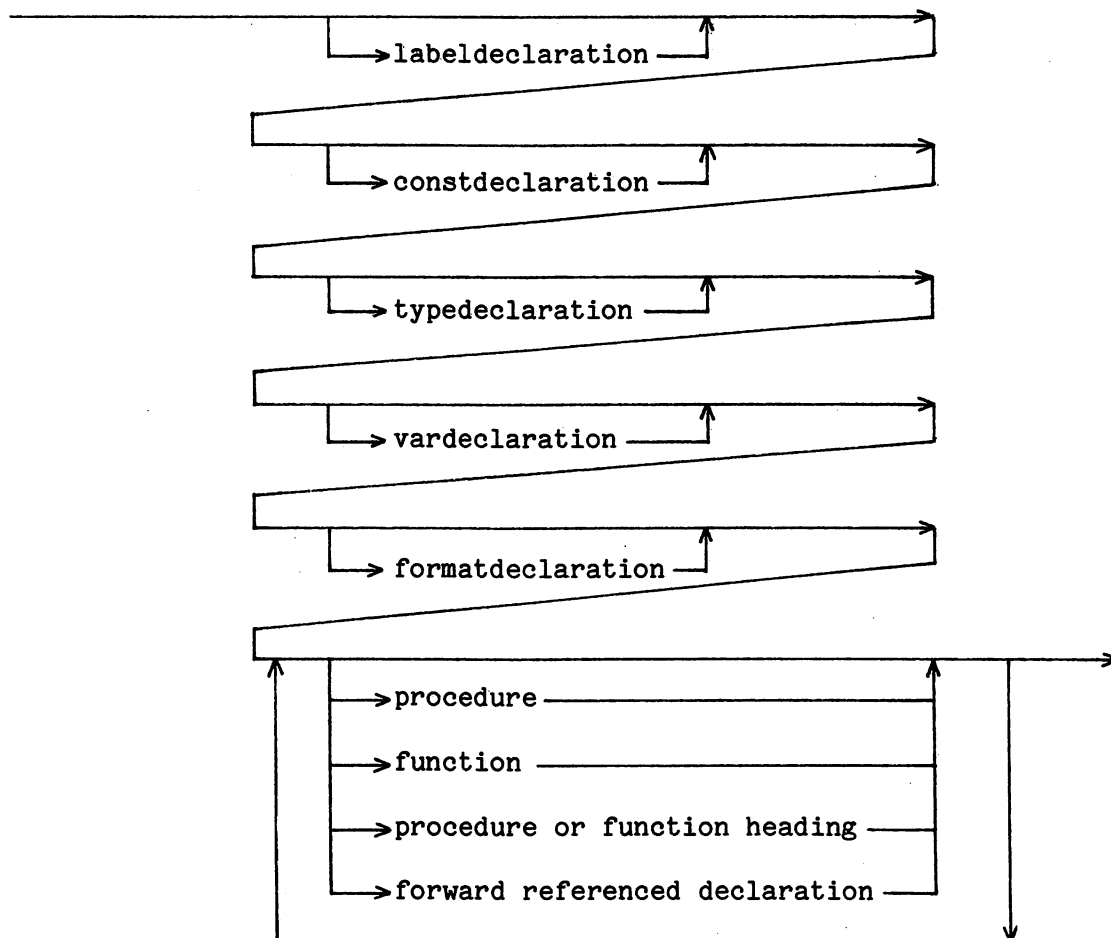
4. DECLARATIONS

Explanation

Declarations serve to specify objects or concepts to be used in the body of a program, procedure or function. The order of the various parts of a declarationpart is required by standard Pascal, or by the extensions.

Syntax

declarationpart



Cross-reference

All components of declarations except procedures, functions, etc, are

DECLARATIONS

described in this section. The exceptions are covered in the section on program units (#6).

ARRAY TYPESemantics

An array type is a collection of objects of the component type, one of which may be selected by nominating values for the array's index.

The component type of an array may be any type other than a file type. The index types of an array may be any scalar or subrange type except integer itself (as this is virtually infinite). There is no limit to the number of index types.

An array is stored in B6700/B7700 Pascal by a single segment of memory, described by a descriptor in the stack. Selecting a component is carried out by computing the displacement of the component from the start of the segment and then indexing the descriptor. An attempt to access outside the bounds of the array will result in the computer detecting an attempted violation and terminating the program's execution. Accessing arrays with several indices, or arrays of multi-word objects, is relatively slow compared to simple scalar variable access.

Storage is allocated for an array at the first time it is accessed after entering the program unit in which it is entered, and deallocated on leaving that program unit. Procedures or functions which are frequently called may therefore incur less operating system overhead if any arrays declared in them are moved to an outer program unit that has a longer lifetime. On creation, an array will be filled with all-zero words. This initialization will not hold for other Pascal compilers.

Examples of array type declarations

TYPE

```

FLOORBUTTON = ARRAY[FLOOR] OF SWITCH;
BIRDDENSITY = ARRAY[GRIDINDEXTYPE,GRIDINDEXTYPE]
              OF INTEGER;
CATALOGUE   = ARRAY[USERCODETYPE] OF
              ARRAY[AREATYPE] OF
              ARRAY[0..30] OF WORD;
HASHTABLE   = ARRAY[0..MAXSIZE] OF
              RECORD
                KEY:KEYTYPE;
                VALUE:VALUETYPE;
                VALUEKIND:VALUEKINDTYPE
              END;

```

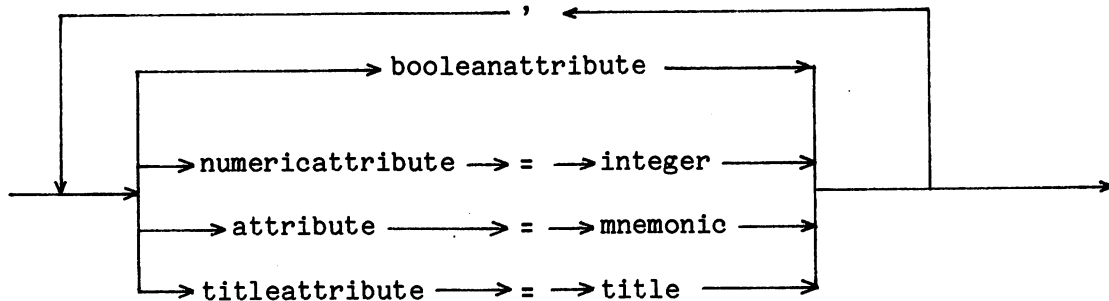
DECLARATIONS

ATTRIBUTES

ATTRIBUTES

Syntax

attributelist



Semantics

The attribute list serves to define some attributes of files declared in a Pascal program, and will determine the nature of the file's attachment to a real file in the Burroughs operating system. The attribute names are always the same as those used elsewhere in the Burroughs operating system and in Burroughs Algol, if the attribute has been incorporated into Pascal. For full details of the use and operation of file attributes, the reader is referred to Burroughs documentation and especially the 'I/O Subsystem Manual'.

In general, an attribute name is followed by a mnemonic which defines its value, or an integer value, as in the example:

KIND=PACK, FLEXIBLE=TRUE, MAXRECSIZE=30

However, for compatibility with other Burroughs usages, it is possible to substitute a numeric value for the word PACK above if you know the appropriate encoding. It is also possible to omit the '=TRUE' for attributes which have boolean values. Neither of these practices are recommended.

In the case of attributes whose values are file-titles (title and security guard), the right-handed side part may be written in the normal way for file-titles, but all possible syntax assumptions are allowed. Thus quotes may be used to delimit the file-title or not, and a stop may terminate it or not. The recommended standard treatment is the same as WFL - with no quotes and no stop:

TITLE=COURSE1/MARKS/DATA

(The options allow Algol, FORTRAN and COBOL programmers to use their familiar file attribute syntax without causing an error). If a reserved word is used in a file title, the title should be enclosed in quotes in the attribute list. Attribute lists are scanned by a modified lexical scanner, and the mnemonics that appear in it bear no relation to any declared Pascal object. It is permissible to declare a Pascal object to have the name kind, or private, for example, and no confusion will result. However, if the parenthesis that opens an attribute list is inadvertently not matched by a closing parenthesis, some curious messages may be evoked.

The following table details the attribute subset at present built into B6700/B7700 Pascal. All attributes may be over-ridden by file equation statements in the Work-Flow program (job control program). It is not possible, at present, to alter file-attributes during execution, as can be done in Burroughs Algol.

ATTRIBUTES	ALLOWED VALUES
AREAS	numeric
AREASIZE	numeric
BLOCKSIZE	numeric
BUFFERS	numeric
DENSITY	LOW, MEDIUM, HIGH, SUPER
EXCLUSIVE	TRUE, FALSE
EXTMODE	SINGLE, HEX, BCL, EBCDIC, ASCII, BINARY
FAMILYNAME	file-title
FILEKIND	numeric
FILETYPE	numeric
FLEXIBLE	TRUE, FALSE
INTMODE	SINGLE, HEX, BCL, EBCDIC, ASCII
KIND	READER, PRINTER, REMOTE, DISK, PACK, TAPE7, TAPE9, PETAPE, TAPE
MAXRECSIZE	numeric
MINRECSIZE	numeric
MYUSE	IN, OUT, IO, CLOSED
PACKNAME	file-title
PROTECTION	TEMPORARY, SAVE, PROTECTED
SAVEFACTOR	numeric
SECURITYGUARD	file-title
SECURITYTYPE	PRIVATE, CLASSA, CLASSB, PUBLIC, GUARDED
SECURITYUSE	IN, OUT, IO, SECURED
TITLE	file-title
UNITS	WORDS, CHARACTERS

ATTRIBUTES

Examples

The following examples show file-attribute lists embedded in realistic declarations:

TYPE

```
  REMOTETERMINAL=  
    FILE(KIND=REMOTE,UNITS=CHARACTERS,MAXRECSIZE=132,  
          MYUSE=IO,FILETYPE=3,EXTMODE=ASCII)  
    OF PACKED ARRAY[0..131] OF CHAR;
```

VAR

```
  INPUT2:  
    FILE(KIND=PACK,TITLE=SECONDARY/STAR/DATA,  
          FILETYPE=7) OF STARRECORD;
```

CODEF:

```
  FILE(KIND=DISK,MAXRECSIZE=30,UNITS=WORDS,  
        BLOCKSIZE=300,FLEXIBLE=TRUE,MYUSE=OUT,  
        SECURITYTYPE=PRIVATE) OF ARRAY[0..29] OF WORDSET;
```

BOOLEAN TYPESemantics

In many ways, type boolean behaves as if it were declared:

```
TYPE
  BOOLEAN = (FALSE,TRUE);
```

particularly in i/o. However, several operators are provided uniquely for the boolean type (AND, OR, NOT) so that its logical calculus can be used to control the flow of the program in execution.

Each boolean variable is stored in a full B6700/B7700 Pascal word. Only the rightmost bit is significant however, and is 0 to represent FALSE and 1 to represent TRUE. The remaining bits (1-47) are always 0.

CHAR TYPE

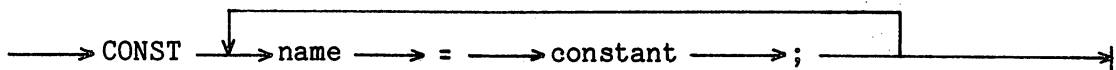
CHAR TYPE

Semantics

The char type is a scalar type which represents character objects. The ordering of the character objects and the range of the scalar depends upon the setting of the compiler option ASCII. The char type is either:

- * a scalar type of 256 values, being the characters of the EBCDIC character set, or
- * a scalar type of 128 values, being the characters of the ASCII character set.

The ordering between characters depends upon the set chosen. Refer to the appendix for details.

CONST DECLARATIONSyntaxSemantics

The CONST declaration is used to declare certain names as constant names. Their employment in the program is exactly as if the corresponding constant appeared in place of the name.

The allowable constants are integer, char, real and string. The names of course have the appropriate types and properties. The predefined constant MAXINT represents the largest integer, 549755813887, available in the B6700/B7700 computers.

Example

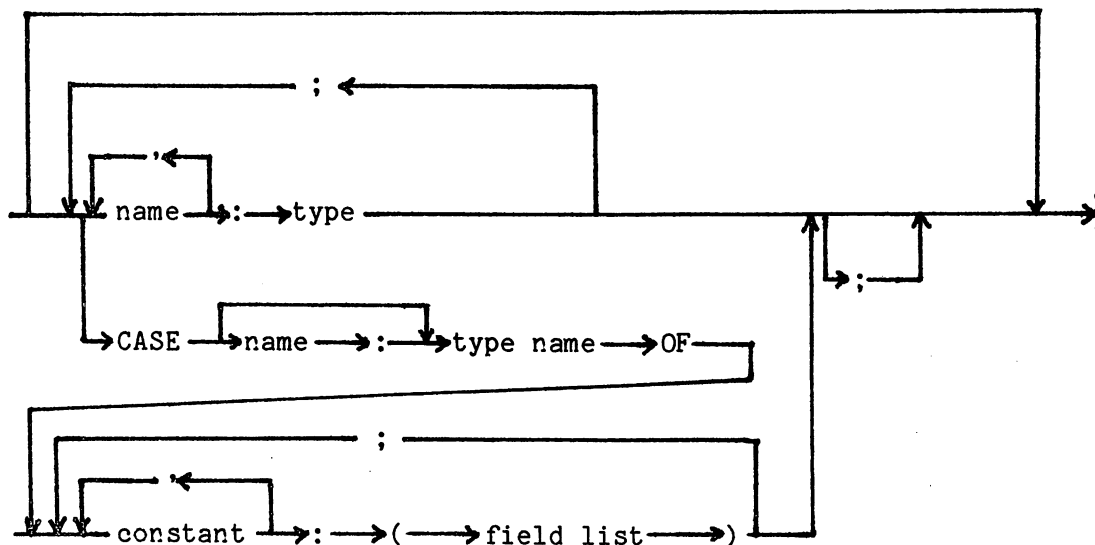
```
CONST
  PI = 3.1415926;
  SPACE = ' ';
  FIFTY = 50;
```

FIELD LIST

FIELD LIST

Syntax

field list



Semantics

A field list serves to define the field names and properties of the components of a record. In the simplest form of field list, the body of the definition consists of a sequence of field names and their types (no CASE part).

In the event of a CASE part following an initial part (if any), the field which corresponds to the selection determines which set of interpretations are to be placed on the remnant of the record. Attempts to access fields of a record in fact share storage in the B6700 implementation, and a declared record is allocated sufficient space for the largest variant.

Field names in a field list are regarded as declared at a lexical level greater than that of the program unit they are enclosed in, so that in the event of a name clash with an already existing object or type (even in the same program unit) the field name is allowed in its appropriate context. The use of a WITH statement in execution is of interest in this connection.

FILE TYPESemantics

An object of file type is a sequence of objects of the file's componenttype, whose length is not necessarily fixed as in an array, and which is conceptually too large to be all accessible at high speed. Associated with every file object is a file buffer: an object of the file's componenttype which either holds a copy of the last component read (if the file is in read status), or is used to contain components to be written to the file (if it is in write status). The file buffer is referenced by giving the filename followed by "@" or "^" thus:

INPUT@ or INPUT^

In B6700/B7700 Pascal, the componenttype of a file may only be a record type, an array type, or a simple type (char, integer, etc.). If a file is accessed through the Pascal stream-oriented READ and WRITE procedures it is set into 'read-status' or 'write-status', and stream-oriented operations of the wrong kind will not be permitted. Thus to carry out input and output on a single remote terminal using the stream oriented procedures, two Pascal files must be declared, with MYUSE=IN and MYUSE=OUT, both attached to the same physical device.

A check is made to ensure that the declared MAXRECSIZE is at least capable of holding the componenttype of the file, and an error message is generated if not. If the Work Flow alters the MAXRECSIZE so as to cause such an error, the effects of accessing the file are undefined when using I/O to files of a structured type.

A file declared in B6700/B7700 Pascal has other attributes besides its componenttype, and these relate to the actual mode of storage or entry of the file and its physical retrieval. See the section on Attributes.

It is also possible to carry out random-access read or write actions on a file in B6700/B7700 Pascal. The section on I/O describes the necessary constructs for this purpose (READ, WRITE, SEEK).

Standards

Attributes are not a part of standard Pascal. The file parameter part which is allowed in CDC Pascal as part of the program heading is similarly permitted in B6700/B7700 Pascal but is not parsed and has no effect on the compilation other than the issuing of a warning note.

In standard Pascal it is possible to declare a file whose componenttype is a scalar, a set, a pointer, a record, or an array. In B6700/B7700 Pascal only records, arrays, or simple types are permitted.

FILE TYPE

Implementation

Since aspects of files are dependent upon the attachment of the file to the Burroughs operating system, the following implementation details are given to simplify the disentangling of unexpected effects.

Internal representation:

Each file variable is represented by three items in the local stack activation area. These items are:

- * a descriptor pointing to a File Information Block (FIB) which is the area used by the operating system to describe the file and its status.
- * a descriptor pointing to a segment which is used as the file-buffer.
- * a descriptor pointing to a small segment which contains information which is specific to Pascal stream i/o (and particularly character i/o).

The FIB is completely determined by the operating system and its descriptor must reside in the stack. Consequently files cannot be subcomponents of other data structuring methods.

Parameter passing:

A file cannot be passed by value (default mechanism), but may be passed to a program unit as a VAR parameter. Three copy-descriptors are passed in the program units calling sequence, corresponding to the items above.

External representation:

This is complex, and in accordance with the B6700 operating system rules.

The records contain an integral number of 48-bit words if (UNITS=WORDS), and an integral number of 8-bit bytes if (UNITS=CHARACTERS). On printing or display devices, the characters corresponding to particular byte values are device-dependent. In B6700/B7700 Pascal, programmers have access to all the types of files that exist on the system, including READER, PRINTER, REMOTE, PACK and TAPE files.

Management:

The FIB and associated descriptors are set up on entry to the program unit in which they are declared. The association of a Pascal file variable with an actual Burroughs file is only attempted when the first access attempt on the file is made. At block-exit (when the program unit returns to its caller) any stream i/o is flushed from the buffer and the file is closed if it is not already in this state. On re-entry to this program unit the file attachment is set up anew, and in some cases a completely new Burroughs file may be involved. Thus printer and temporary files are completely local to the program unit in which they are declared.

The stream i/o procedures are implemented by calls on specially written intrinsic procedures; the formatted i/o procedures are implemented by calls on the ALGOL-FORTRAN intrinsic procedures. The two types of i/o should not be mixed on one file, but can be, provided that formatted i/o is only attempted when a whole input - or output record has been processed by the stream procedures. The stream i/o buffer is flushed before exit from the program unit, and before selected i/o procedures such as RESET, REWRITE, REWIND, CLOSE, and SEEK. It is not flushed if an error causes program termination and in these circumstances an incomplete line will be lost on an output file.

The following description details significant events in the lifetime of a Pascal file.

(i) at the first access attempt

if the reference is to a file that may have a title, or an external existence, the directory is searched for one with the same TITLE (or STATION if REMOTE), and the attachment made if the file title is found;
otherwise if the attribute PROTECTION=PROTECTED then a new permanent file is created (this is rarely used);
otherwise a new temporary file is created.

(ii) at the execution of a CLOSE statement

if LOCK or CRUNCH is specified, and the file is temporary, it is entered in the directory and made permanent;
otherwise if PURGE is specified; the file is destroyed whether or not it is permanent;
otherwise the file is closed, but not necessarily lost.

(iii) at block exit

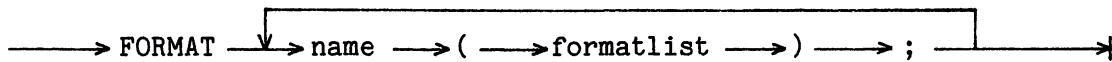
if the file is now permanent, it is simply detached from the program;
otherwise if it is KIND=PRINTER, it is detached as a printer-backup file, and lives on in the system until it is completely printed;
otherwise it is destroyed.

FORMAT DECLARATION

FORMAT DECLARATION

Syntax

format declaration



Semantics

A format declaration serves to associate a name with a format layout, and to define that layout. The syntax of a formatlist is identical to that of the Burroughs B6700/B7700 Algol language, and is described in detail in the manual for that language. The major exception is the use of * in the repeat count part of a format element, which should not be used in formats which will be associated with read statements. Because of the use of tag-six words to detect uninitialized variables in Pascal, the *-facility used with a read statement will act as though zero was substituted for the asterisk.

Standards

Format declarations are not part of standard Pascal.

Warning

The use of the *-facility can cause mysterious errors if the lexical rules are not complied with. Consider the example:

```
FORMAT
  MANYNUMBERS(*I5);
```

This will be parsed as though it contained a Pascal-comment with no closing *), probably swallowing chunks of source text:

```
FORMAT MANYNUMBERS (* I5);
-----> on to the next *)
```

A space should appear between the parenthesis and the asterisk to avoid this problem.

Examples of format declarations

FORMAT

```
HEADING ("CROSS-REFERENCE LISTING - XREF PROGRAM"  
/ "=====");  
TABLELINE (X5,20I5);  
COMPLEXES (" (",F18.8," ",F18.8,"")");
```


INTEGER TYPE

INTEGER TYPE

Semantics

The integer type in B6700/B7700 Pascal is implemented as the one-word integer data-type of the B6700/B7700 computers. All values of integers from -549755813887 to +549755813887 can be represented, and provided no intermediate result of an integer expression exceeds these bounds the arithmetic conforms exactly to the usual arithmetic axioms. If an attempt is made to store an out-of-bounds result into an integer, or use it in an integer context (as a parameter for instance, or an index to an array), the B6700/B7700 will interrupt and terminate the program's execution. Though the machine has both a +0 and a -0, the Pascal programmer should not be able to detect the difference except by printing a value in radix notation: they are arithmetically identical.

An integer variable occupies a full B6700/B7700 word. The maximum integer size corresponds to a 39-bit field: $549755813887 = (2^{39}-1)$. The sign is separately represented.

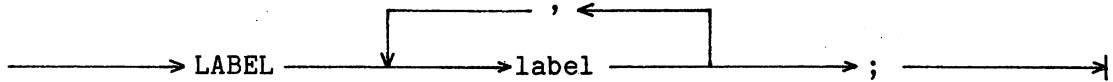
The integer type is one of the numeric types: the operators + - * div mod are defined on it. The usual rules apply for + - * ; div and mod are defined to return quotients or remainders such that:

$$\begin{aligned} \text{abs}(a \text{ div } b) &= (\text{abs}(a)) \text{ div } (\text{abs}(b)) \\ a &= (b * \text{quotient}) + \text{remainder} \\ &\text{where } 0 \leq \text{abs}(\text{remainder}) < \text{abs}(b) \\ &\text{and } \text{sign}(a) = \text{sign}(\text{remainder}) \end{aligned}$$

LABEL DECLARATION

Syntax

label declaration



Semantics

The integer constants in the list which are labels must lie in the range (0..9999), and are used to declare to the compiler that these 'labels' will be used to mark places in the executable text of the program.

The necessity to declare labels arises infrequently, as a consequence of the use of a GOTO statement. Refer to LABELS in SUBCOMPONENTS, and GOTO in STATEMENTS for other information.

Example

```
LABEL
    1, 2, 56, 999;
```

PACKED

PACKED

Explanation

The use of the word PACKED in a type declaration tells the compiler to compact the type structure even if this means adding a code-space or run-time penalty when the structure is accessed.

If PACKED is not specified then each element of a structured type occupies a word of storage.

When PACKED is specified, the elements of a structured type occupy as few bits as necessary to represent the simple type with the following restrictions:

- 1) a structured type occupies an integral number of computer words and always starts on a word boundary
- 2) no elementary item crosses a word boundary.

In addition, the following restrictions apply to individual types:

set type: The structured type SET is always regarded as PACKED, whether PACKED was specified or not and each element occupies one bit of a computer word.

scalar type: A declared scalar type occupies as many bits as are needed to represent the type

eg.

```
TYPE  
    COLOUR = (RED,BLUE,GREEN);    occupies 2 bits
```

subrange type: An element of type subrange is stored as the offset from its base value and occupies as many bits as are necessary to represent the range of values. The base value is added or subtracted each time the element is accessed.

eg.

```
TYPE  
    SUBR = 100 .. 110;
```

represents 11 values and occupies 4 bits of storage. The items are stored as 0 .. 10 and 100 is added or subtracted each time the element is accessed.

array types: An array type consists of elements occupying 1,4,6,8 or 48 bits of storage. Elements which require fewer bits to represent them use the smallest number from this list to represent them adequately. These element sizes were chosen because an array

descriptor on a Burroughs B6700 uses element sizes of 4,6,8 or 48 bits.

eg.

```

TYPE
  COLOUR = (RED,BLUE,GREEN);
  ARR = PACKED ARRAY [1..10] of COLOUR;

```

Each element of the array requires 2 bits of storage to represent it but it will occupy 4 bits because of the B6700 array handling mechanism.

record types: If a record is packed then each component occupies as many bits as the component needs. Word alignment only occurs when a component is a SET, RECORD or ARRAY. No component may be split across a word boundary.

Packing starts at the left hand end of a word.

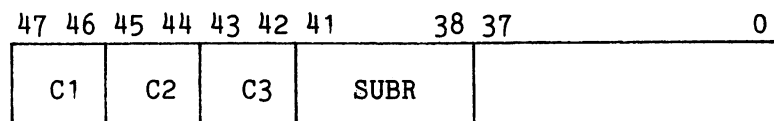
eg.

```

TYPE
  COLOUR = (RED,BLUE,GREEN);
  REC = PACKED RECORD
    C1,C2,C3 : COLOUR;
    SUBR : 100 .. 110;
    ...
  end;

```

The physical representation of this record will be



Standards

The Pascal Standard does not specify a way to implement PACKED and so the storage mechanism may vary amongst implementations.

In B6700 Pascal packing applies to the level at which PACKED is specified only.

for example:

```

A : PACKED RECORD
  B : ARRAY [1..10] OF SUBR;
  ...
END;

```

In this case the record is packed, but the array within the record is

PACKED

unpacked.

For multi-dimensioned arrays, packing applies to each dimension. ie.

PACKED ARRAY [0..5,0..7,0..9] OF THING

is equivalent to

PACKED ARRAY [0..5] OF
PACKED ARRAY [0..7] OF
PACKED ARRAY [0..9] OF
THING

POINTER TYPE

Semantics

An object of pointer type is a reference to an object of the basetype of the pointer type, or has the value nil (which refers to no objects at all). Pointers are therefore bound to objects of a particular type, and since pointer values may only be created by the use of the NEW procedure, they may only refer to objects created in the Pascal run-time heap by that procedure.

Pointer values may be assigned, or may be used in a variable reference to identify a particular object (see VARIABLE in SUBCOMPONENTS). The constant value nil is compatible with all pointer types.

The declaration of a pointer type may precede the declaration of its basetype, or the completion of declaration of its basetype. This permits the declaration of types which contain mutually-referent pointers, and of types that are self referent. The full declaration of such a type is postponed until the close of the TYPE part when any unsatisfied declarations are reported as errors.

Standards

If a forward-declared pointer type is declared in a procedure outside which there is a type with the same name as the forward declaration, then the standard Pascal definition requires binding the pointer to the inner type. Thus the following is legal in B6700/B7700 Pascal, and in Standard Pascal. Some other compilers may be non-standard.

```

PROGRAM P;
  TYPE THING = BOOLEAN;
  PROCEDURE Q;
    TYPE LOOSETHING = @THING;
    THING = REAL;
    VAR THINGPTR : LOOSETHING;
    BEGIN
      NEW(THINGPTR);
      THINGPTR@:=1.0;
    END;
  .....
```

POINTER TYPE

Implementation

The Pascal run-time heap is implemented as a segmented array (segment size = 256 words) based in the outermost stack activation area with a current-top-of-heap value in that same area. The size of the segment is determined at compile-time by the HEAP compiler option. Values of pointer type are represented by B6700/B7700 integer values, the nil value being an out-of-range integer (thus causing invalid index faults when accidentally used for access). However, pointer objects may only acquire values as a result of the NEW procedure, or by association with some other type in a discriminated union (and therefore by error). Such occurrences may cause interrupts such as INTEGER OVERFLOW, or may result in undetected erroneous access.

REAL TYPESemantics

The real type in B6700/B7700 Pascal is implemented as the one-word real data type of the Burroughs B6700/B7700 computers. This allows the approximate representation of values whose magnitude is either zero (exactly represented) or lies between

$$8.75811540203 * (10^{**-47}) \quad (\text{or } 8^{**-51}) \quad \text{and} \\ 4.31359146674 * (10^{**+68}) \quad (\text{or } 8^{**+76} - 8^{**+63}).$$

The precision of the representation is approximately 11 digits. Accurate representation details are given below if they are needed.

A real variable occupies a full B6700/B7700 word.

The real type is one of the numeric types: the operators + - * / are defined for it. For the purposes of expressions, a variable of type integer is regarded as being compatible with a variable of type real. The values of type integer are all EXACTLY representable in type real, and they form a subset of the real values. This may not be true of other computers and other Pascal compilers.

Representation data

Real values are represented in a 48-bit word by

- * a one-bit sign (bit 46),
- * a 13 octal-digit mantissa (bits 0-38),
- * a one-bit exponent sign (bit 45), and
- * a six-bit exponent magnitude giving a power of 8 (bits 39-44)

The mantissa is regarded as an integer, and the value represented exactly by a real word is:

$$\text{signed-mantissa} * (8^{** \text{ signed-exponent}})$$

To preserve as much accuracy as possible, the mantissa is octal-normalized if necessary. The high octal digit is therefore in the range 1 to 7 if the number is normalized. Integer values are often not normalized, and are represented as real numbers with zero exponent. Zero is represented by the all zero word.

Arithmetic is carried out in a double-length calculator, and the result is rounded to a single-length result. Since the arithmetic is basically octal (not binary) the relative precision of the representation varies between 8^{**-12} to 8^{**-13} , depending on the mantissa. This is normally sufficient for most numerical computations. See JUB6700 (Journal for the Users of the Burroughs 6700), No 5, April 1975 for an article entitled "Round-off errors in the elementary arithmetic operations of the B6700" by P. Voss giving details of the arithmetic properties.

RECORD TYPE

RECORD TYPE

Semantics

A record type is a collection of objects of different (or the same) types, one of which is selected by giving the appropriate fieldname. The types of the objects are determined by the fieldlist in the declaration. It is possible to have variant forms of a record type.

The component types of a record type may be any type other than a file type. There is no limit to the number of different types, and effectively no limit to the number of objects other than general system constraints.

A record is stored in B6700/B7700 Pascal by a single segment of memory, described by a descriptor in the stack. Selecting a field is carried out by indexing up the segment by the known fixed displacement. Accessing a field of a record is necessarily slower than accessing a directly declared object of the same type, especially if it is of scalar, real, set, or pointer type.

Storage is allocated for a record at the first time it is accessed after entering the program unit in which it is declared, and deallocated on leaving that unit. Procedures and functions which are frequently called may therefore incur less operating system overhead if any records declared in them (as actual declarations or default value parameters) are moved to an outer program unit that has a longer lifetime. On creation, a record will be filled with all-zero words. This initialization will not hold for other Pascal compilers. A declared record has sufficient space for the longest of its declared variants.

Examples of record type declarations

```
TYPE
  COMPLEX = RECORD REAL,IMAG : REAL END;
  PERSON  = RECORD
    BIRTHDATE : DATE;
    CASE PERSONSEX:SEX OF
      MALE   : (WORKER : BOOLEAN);
      FEMALE : (ARTISTIC : BOOLEAN;
                FIRSTCHILD : PTRTOPERSON)
    END;
END;
```

SCALAR TYPESemantics

A scalar type declared by a programmer is regarded as an infinite collection of values which are represented in the external world (in the program text or in the i/o stream) by upper-cases constant names. Mixed upper and lower case forms are permitted and are treated as identical. The internal representation of the scalar values is the integer B6700 words: 0, 1, 2, 3,...etc. The ORD function applied to any scalar value will yield the appropriate numeric value.

The SUCC and PRED functions, defined on scalar types, are implemented by in-line code. The attempt to take the PRED of the first scalar constant, or SUCC of the last, is always detected and causes the program to be terminated.

Boolean type is treated similarly to scalar types, but with additional operators; the char type is a scalar type with a special syntax for constants of the type, as is the string type. Integer type is a scalar type with a very large number of values and additional operators, and real type is hardly a scalar type except in name.

Examples of scalar type declarations

TYPE

```
FLOOR = (BASEMENT,GROUND,MEZZANINE,FIRSTFLOOR,
          SECONDFLOOR);
REMARK = (ATROCIOUS,BAD,POOR,SATISFACTORY,GOOD,EXCELLENT);
SWITCH = (ON,OFF);
REPLY = (YES,NO,MAYBE);
```

Standards

If, within the scope of a scalar declaration, another scalar is declared using the same constant names, then the two types are considered distinct, and the innermost redefines the outer definition for purposes of parsing. If an apparent redefinition occurs at the same lexical level then this is an error.

SCALAR TYPE

Thus in :

```
TYPE SEX = (MALE,FEMALE);  
VAR X : (MALE,FEMALE);  
PROCEDURE Q;  
  VAR Y : (MALE,FEMALE);  
  BEGIN  
    IF Y = MALE THEN....
```

the declaration of X is in error (twice, once for each matching identifier), while the declaration of Y is legal, but of different type from that denoted by SEX. Within the scope of Y the identifier MALE is a constant of Y's type, as shown in the IF statement. Some compilers may not comply with the requirements of the Standard in this regard.

SET TYPESemantics

The component type of a set type may only be a scalar or subrange, and the set cannot have more than 65536 possible members.

The bounds of a set are not restricted to positive integers. Negative integers may be used if necessary. However, the bounds imposed on set constructors limit the use of negative bounds. If more than 48 members of a set are required the compiler option SETSIZE must be set accordingly. A set of char is permitted by the implementation.

A set with bounds lying between 0 and 47 inclusive is implemented as a single B6700/B7700 word. A set with bounds lying outside these bounds is implemented as an array. Membership of sets is indicated by the corresponding bit of the set being 1.

The set operations on short sets (ie. those with bounds between 0 and 47 inclusive) are implemented by word-wise logical operations. On long sets a call to the intrinsics is required.

SUBRANGE TYPE

SUBRANGE TYPE

Semantics

A subrange type is represented in the same way as a variable which has a basetype (the type in which the subrange is defined). A subrange type is compatible with its basetype (or indeed, other subranges of the basetype) for most purposes. An error is produced if an attempt is made to assign a value which is outside the defined subrange (but which may be inside the basetype range). See the BOUNDSCHECK compiler option.

A subrange of integer has a property not possessed by its basetype: the number of values in it is regarded as infinite, whereas the number of values in integer type is regarded as sensibly infinite (though in practice this is a large finite value). This means that a subrange of integer can be an indextype of an array, or a componenttype of a set, but integer type cannot.

Examples of subrange type declarations

TYPE

```
DIRECTION = -1..+1;  
BOARDSQUARE = 0..7;  
ALPHABET = 'A'..'Z';  
LADIESWEAR = MEZZANINE..FIRSTFLOOR;  
PASSREMARK = SATISFACTORY..EXCELLENT;
```

TEXT TYPESemantics

A variable of type text is represented in the same way as a file of char.

Implementation

The default file attributes are

(KIND=DISK, FILETYPE=7, INTMODE=EBCDIC)

If different attributes are required, they should be overwritten in the Work Flow Language or the file (attributes) of char form of declaration should be used.

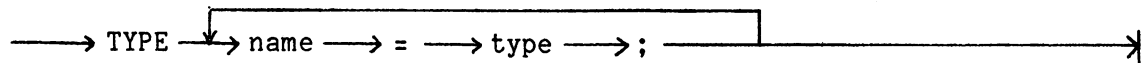
When the file is initially opened, a check is made to see if the internal buffer is large enough to accommodate a record from the file. If it is not, the buffer is resized to make it large enough. The default buffer size is 132 characters for all files except the predefined file INPUT where the buffer size is 80 characters.

TYPE DECLARATION

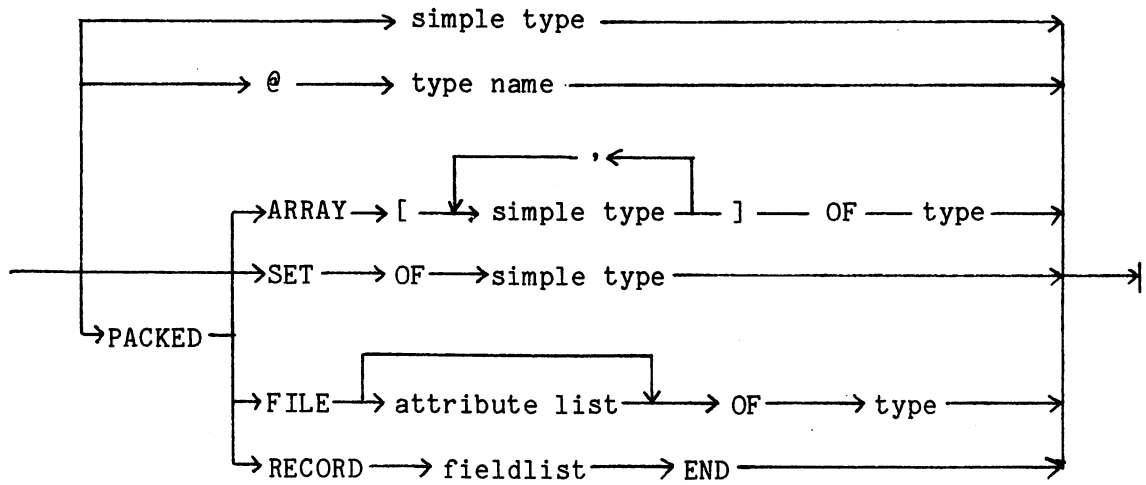
TYPE DECLARATION

Syntax

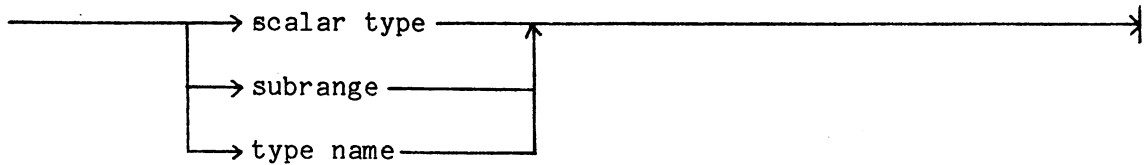
type declaration



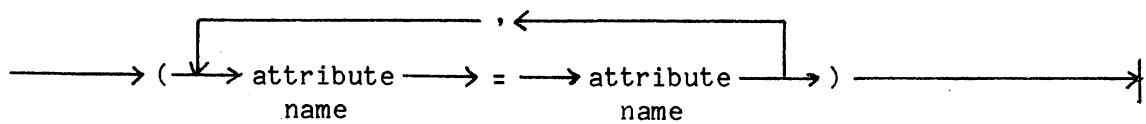
type



simple type



attribute list



Semantics

A type declaration serves to associate a name (that given in the

type declaration) with some properties. The properties of variables declared to be of a given type are used by the compiler to generate the appropriate code.

There are several types which are regarded as predefined in B6700/B7700 Pascal. The names of these types are boolean, char, integer, and real and they are separately discussed.

There are basically six possible kinds of types which can be declared in a type declaration:

- * scalars and subranges of scalars,
- * pointers,
- * arrays,
- * sets,
- * files, and
- * records.

A scalar type is either one of the predefined types boolean, char or integer, or is defined by a list of constant names of the type, or is defined to be a subrange of one of the scalar types. Each of the other types is explained separately.

Example

```

TYPE
  GENDER      = (FEMALE, NEUTER, MALE);
  GRIDINDEX   = 0..99;
  XARRAY      = ARRAY[GRIDINDEX] OF REAL;
  PLOT        = ARRAY[0..59, 0..131] OF CHAR;
  ALLOWABLEGENDERS = SET OF GENDER;
  MOVIEFILM   = FILE OF PLOT;
  RECORDCOMPLEX =
    RECORD
      X, Y : REAL
    END;

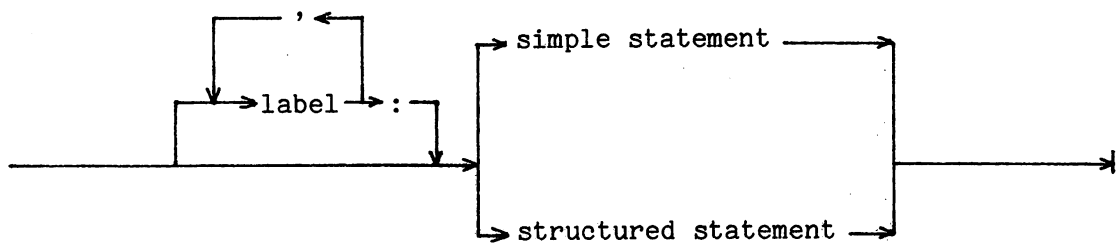
```


5. STATEMENTSSTATEMENTSExplanation

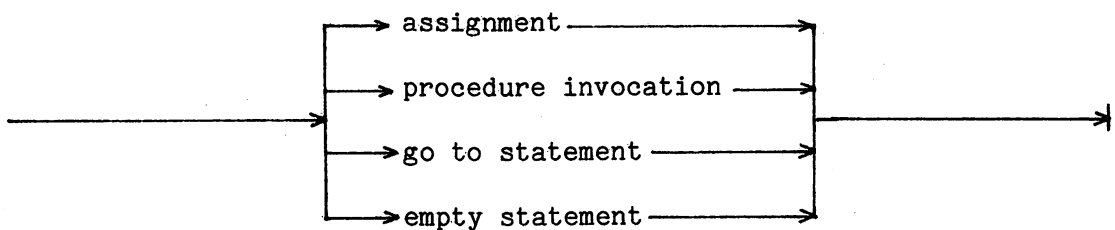
In this section are described those components of the B6700/B7700 Pascal language that make up the body of a program, procedure, or function. These components are executable actions, or rules which determine the interpretation of executable actions, and are called statements.

Syntax

statement

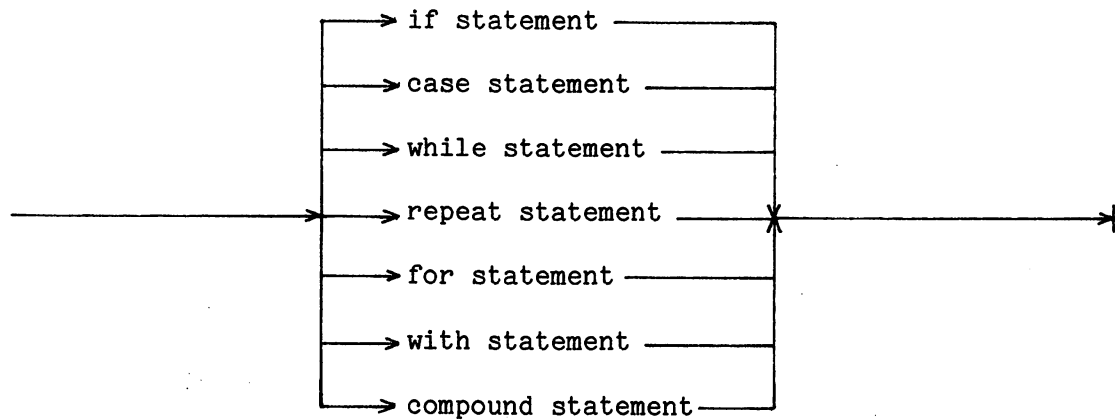


simple statement



STATEMENTS

structured statement



Semantics

A statement defines an action or interpretation as described in this section. Optionally, a statement may be preceded by a label or labels. Such labels consist of integer constants less than 10000 in denoted value and declared in a label declaration. Their purpose is to mark a statement for the purpose of executing a goto statement to that point.

Examples

R:=R-3

73 : Z:=r

ASSIGNMENTSyntax

assignment

→ variable → := → expression →

Semantics

The expression is evaluated, and its value replaces the current value of the variable identified by the left-hand side of the assignment. The types of the variable and the expression must be assignment compatible. A file cannot be assigned, nor be assigned to.

If the expression is a string whose length is shorter than that required by the variable, trailing blanks will be added to the string.

If the variable is a subrange, an error is produced if the value of the expression lies outside the subrange limits. See the BOUNDSCHECK compiler option.

Standards

The expansion of short strings is non-standard and a warning will be given if STANDARD is set. This expansion does not take place in any other circumstances.

Examples

X:=3.1415926

J:=J+1

FLOOR:=SUCC(FLOOR)

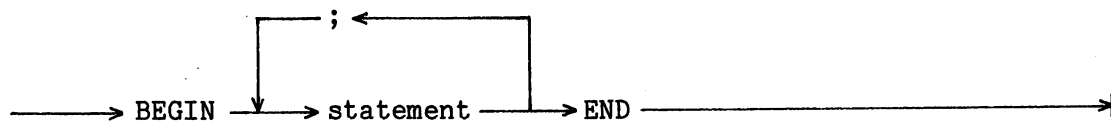
RECORD1:=RECORD2

BODY

BODY

Syntax

body



Semantics

The statements enclosed within the body of a program, procedure, or function are executed in sequence, or as determined by goto statements, until execution arrives at the closing END. At this point, execution is complete, and control returns to the calling program in the case of a procedure or function, or to the B6700/B7700 operating system in the case of a program. The empty statement in Pascal permits a semicolon to immediately precede an END, as used in some program styles.

Examples

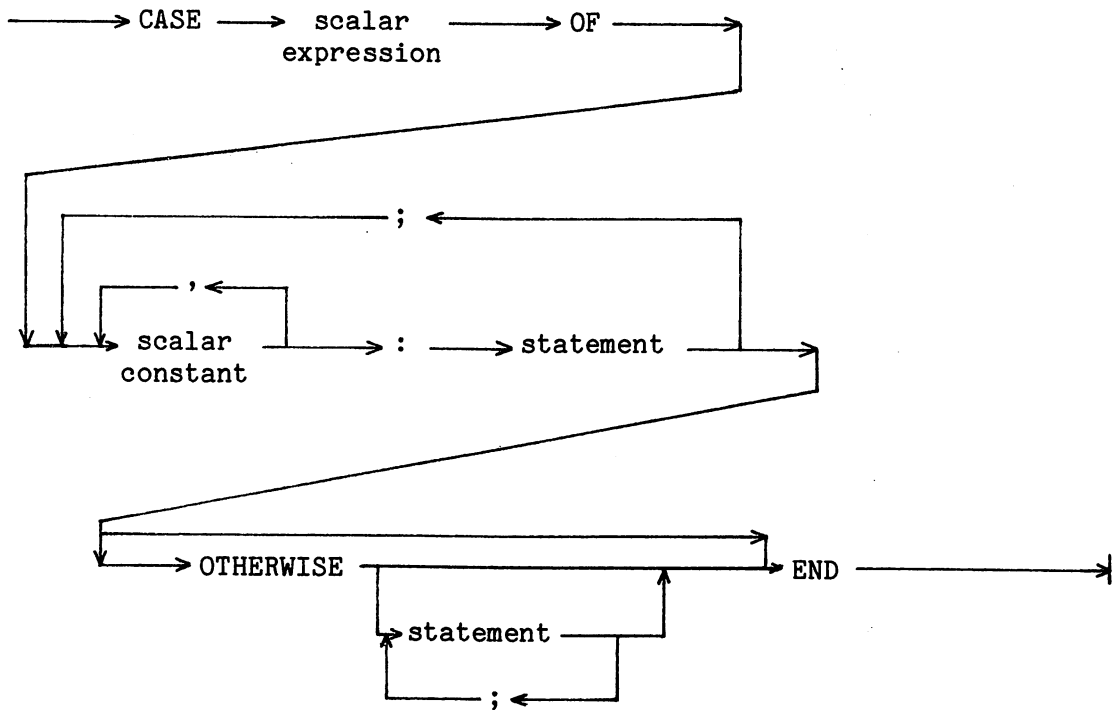
```
BEGIN
  X:=2.0;
  Y:=-Y
END

BEGIN
  PROCESSDATA;
  REPORTERERRORS;
  WRITEFILE;
END
```

CASE STATEMENT

Syntax

case statement



Semantics

The case-statement is used for selecting one of a number of possible computation paths which are mutually exclusive. The scalar expression is evaluated and the statement corresponding to the scalar constant whose value is the same as the expression value is executed. If the expression does not have a value corresponding to any scalar constant in the case-statement, then the action depends upon the appearance or not of OTHERWISE.

- * If OTHERWISE is present, all such values cause the corresponding statements to the OTHERWISE to be executed.
- * If OTHERWISE is not present, all such values are regarded as errors and cause the program to be terminated during

CASE STATEMENT

execution.

The expression must be of scalar type (including integer, but not real) and must be compatible with all the case constants.

Examples

```
CASE PAINT OF
  RED:   EXOTICPROCEDURE;
  GREEN: RESTFULPROCEDURE;
  BLUE:  COOLPROCEDURE;
END
```

```
CASE CH OF
  '0','1','2','3','4','5','6','7','8','9': THING:=DIGIT;
  OTHERWISE THING:=NOTDIGIT;
END
```

Standards

The OTHERWISE part is a new conventionalized extension of standard Pascal; it may not be found in other Pascal compilers. If other compilers provide the facility the syntax and semantics should be the same. It is intended to provide a facility whereby programs can be written that are robust against any possible computation, and should not be misused simply to save writing a long list of case constants.

Some Pascal compilers may object to a semicolon between the last statement of the case construct and its closing END, as permitted by the standard.

In some Pascal compilers, the implementation technique may limit the case-statements that can be successfully compiled either by an explicit compiler limit, or by running out of space. Such an eventuality is very unlikely on B6700/B7700 Pascal, but writers of portable programs should be aware of the problems they may encounter on other systems.

Efficiency

The case statement is implemented in B6700/B7700 Pascal by a sequence of code that evaluates the expression, selects an action based on its value, and jumps to attempt that action. The compiler chooses between two techniques for selection, based on its estimate of the storage required :

- (a) The preferred technique checks that the expression value lies in the range of the least and greatest case-labels, and executes an indexed-jump into a jump-table. This is almost always faster than the other technique, but may demand a large amount of storage for the table. An arbitrary limit of 500 words is imposed. The selection time is independent of the selection width; the space required grows linearly with the case-label range.
- (b) The alternative is a balanced tree of comparisons to classify the expression value into a subrange. If the case-labels are used as sub-ranges, or if the selection is of sparse values, this may use less space. If the number of subranges (including default subranges) is m , then the selection time is proportional to $\log (m)$ and the space required is linearly proportional to m .

COMPOUND STATEMENT

COMPOUND STATEMENT

Syntax

compound statement



Semantics

A compound statement has no action other than to group a sequence of statements and make them syntactically a single statement. The statements referred to in the for-, if-, case-, while-, repeat-, and with-statements are usually compound-statements.

Examples

```
BEGIN X:=X+1; Y:=Y-1 END
```

```
BEGIN  
  ANALYSEDATA;  
  PROCESSANALYSIS;  
  REPORTRESULTS;  
  IF ERRORHAPPENED THEN ERRORREPORT  
END
```


EMPTY STATEMENTSyntax

empty statement

Semantics

The empty statement denotes no action and has zero execution time. Its uses in Pascal include

- (a) permitting redundant semicolons in a statement sequence, for example the last semicolon in :

```
BEGIN
    PROCESSDATA;
    REPORTERRORS;
    WRITEFILE;
END
```

- (b) permitting labels to precede END or UNTIL, for example:

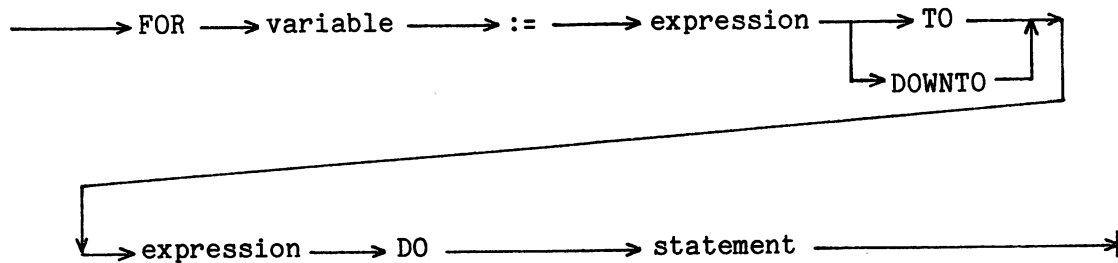
```
BEGIN
    PROCESSDATA;
99:
END
```

FOR STATEMENT

FOR STATEMENT

Syntax

for statement



Semantics

The for-statement sets up a loop which is executed firstly with the variable specified having the value of the first expression, then with the successor or predecessor of this expression (according to whether TO or DOWNTO was used in its construction), and so on repeatedly until the loop is executed with the variable having the value of the last expression. This is the construct commonly known as the 'count loop'.

The variable and the expressions must all be of scalar types (including integer, but not real type), and must be compatible. If the limit expressions evaluate so as to be in the wrong order for counting, the controlled statement is not executed at all.

Examples

```
FOR J:=0 TO 99 DO VEC[J]:=0
```

```
FOR COUNT :=MAXENTRY DOWNTO (MARKER + 1) DO BEGIN
  IF (ARRAYOFENTRY[COUNT] = FORWARDENTRY) THEN BEGIN
    PROCESS(COUNT); FORWARDFLAG :=TRUE;
  END; {of if}
END {of for}
```

Standards

The implementation of the for-statement in B6700/B7700 Pascal forces some definite interpretations on aspects of the Pascal language which may not be handled identically by all compilers for Pascal. These are explained below, but should not be important to the average Pascal programmer.

The expressions in the for-statement are evaluated once only before the loop is ever entered, in the order they appear in the for-statement. The value of the second expression is saved in the local stack for re-use every time the loop test is made.

The assignment to the variable is first made after the computation of both expressions, and before the first test is made as to whether the loop should be entered.

If the variable is a subrange, the values of the two expressions are checked against the subrange bounds before the loop is entered. An error occurs if either bound is exceeded. See the BOUNDSCHECK compiler option.

The value of the controlled variable after the loop is exhausted, is set to be 'uninitialized operand' (tag-six). The value should not be used before redefinition of its value, in accordance with the definition of standard Pascal.

The for-statement is compiled as if it were written as follows, where temp1 and temp2 are conceptually two compiler-generated locations in the local stack activation record.

FOR v := e1 TO e2 DO s;
temp1 := e1;
temp2 := e2;
v := temp1;
check the bounds of v if necessary
<u>while</u> (v <= temp2) <u>do begin</u>
s;
v := succ(v);
<u>end;</u>
undefinevalueof(v);

FOR STATEMENT

For the DOWNTO case, replace \leq by \geq , and succ by pred.

If the following conditions are fulfilled:

- * the first expression (e1) is simply a constant,
- * the second expression (e2) is simply a constant, or is an expression of scalar type having a type limit less than 65535, and
- * the loop is a TO loop,

then the code generated is optimized to use the STBR instruction of the B6700/B7700 computers, and is also checked by the hardware for alterations of the controlled variable from within the loop. Such changes alter the tag of the STEP-INDEX word and the program will be terminated when it next reaches the loop test point since such changes are not permitted in standard Pascal.

Naked attempts to alter the value of the controlled variable from within a for-loop are detected by the compiler and flagged as errors. Examples are assignments to the variable, or a READ into it, in a statement within the loop. Using the controlled variable as an actual parameter matching a formal VAR parameter is flagged as a WARNING. Surreptitious alteration of this variable, which is prohibited in standard Pascal, may therefore only be achieved by ignoring the warning, using a side-effect in a procedure, or by a complex GOTO structure.

Standard Pascal requires that the control variable be local to the block in which the statement appears. This requirement is relaxed in B6700/B7700 Pascal and the control variable may be global to the block in which it appears. A warning is produced if the STANDARD compiler option has been set.

GOTO STATEMENTSyntax

goto statement

—————→ GOTO —————→ label —————→

Semantics

The effect of a goto statement is to force the next statement to be executed to be the one with the label corresponding to the label in the goto statement.

If the goto statement leads out of a construct to a higher level, and does not enter any construct that the goto is not itself in, the action is well-defined. If the goto statement leads into a with statement, or into a for statement, in which the goto statement is not also enclosed, the action of the program may be quite unpredictable as the necessary initialization of these constructs has not been carried out. In other cases the behaviour may be deduced from the description of the constructs entered. Such usages are not defined in standard Pascal.

The goto statement is intended for relatively rare usage when efficiency demands it.

Examples

GOTO 99

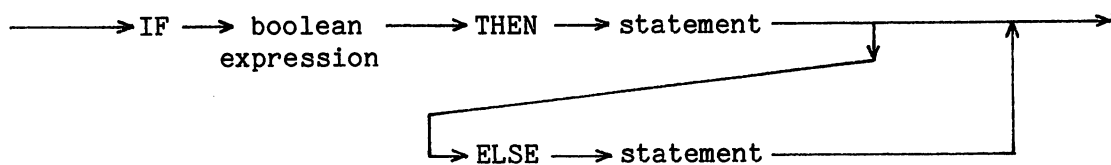
GOTO 1065

IF STATEMENT

IF STATEMENT

Syntax

if statement



Semantics

The boolean expression is evaluated, and if it has the value TRUE, the first statement is executed. If it has the value FALSE, then the second statement is executed if it is present.

If either statement is itself an if-statement containing an ELSE-part, then the ELSE-part is deemed to belong to the immediately preceding IF-THEN part.

```
IF B THEN IF B2 THEN S1 ELSE S2;
```

is equivalent to:

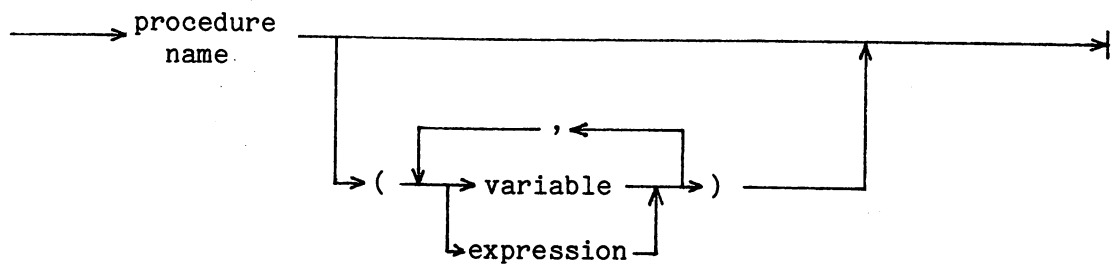
```
IF B THEN BEGIN
    IF B2 THEN S1 ELSE S2;
END;
```

The if-statement is used for selecting one of two alternate computation paths (the IF-THEN-ELSE form), or conditionally executing some code (the IF-THEN form).

PROCEDURE INVOCATION

Syntax

procedure invocation



Semantics

A procedure invocation identifies a procedure by name, and initiates its execution. The parameter list must correspond in length and type (one-for-one) with the declared parameter list of the procedure in its declaration. In addition, for each VAR parameter in the declared parameter list, the corresponding actual parameter must be a variable (not an expression).

A procedure invocation may only occur in the body of a program unit in which the name and attributes of the procedure are known (see declarations).

When the procedure terminates execution, it is resumed at the point immediately following the procedure invocation.

Some procedures are regarded as pre-defined, and are available to all B6700/B7700 Pascal programs unless their names are re-defined by a programmer's own declarations. These are described in a later section.

Other relevant sections to this statement will be found in the section on program units, and particularly on parameter lists.

PROCEDURE INVOCATION

Examples

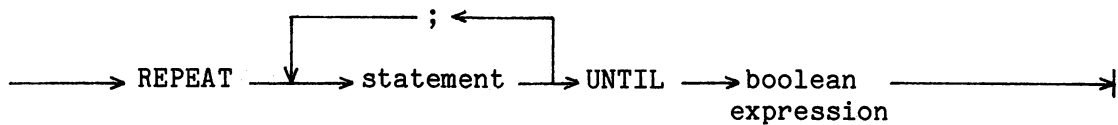
ANALYSE(INPUTCARD, PTR, TYPE, K+2)

PrintARowOfStars

CONJUGATE(X, Y, (SQR(X)+SQR(Y)))

REPEAT STATEMENTSyntax

repeat statement

Semantics

The statement is executed, then the boolean expression is evaluated. If it returns FALSE then the process is repeated, and the statement is re-executed. This continues until the boolean expression evaluates to TRUE (if ever). The statement will always be executed at least once.

Note that the REPEAT-UNTIL statement functions like a BEGIN-END pair of brackets. It is regarded as good B6700/B7700 Pascal language practice to ignore this possibility and place BEGIN-END around the body of a REPEAT statement.

Examples

```
REPEAT X:=SQR(X) UNTIL (X>SIZE)
```

```
REPEAT BEGIN
  WRITELN(J, THTERM);
  J:=J+1;
  NEXTTERM(J, THTERM);
END UNTIL (J=99)
```

WHILE STATEMENT

WHILE STATEMENT

Syntax

while statement

→ WHILE → boolean expression → DO → statement →

Semantics

The boolean expression is evaluated, and if it returns TRUE, the statement is executed. After this is complete, the boolean expression is again evaluated and the process repeated, until the boolean expression becomes FALSE (if ever). The statement will never be executed if the boolean expression returns FALSE on its first evaluation.

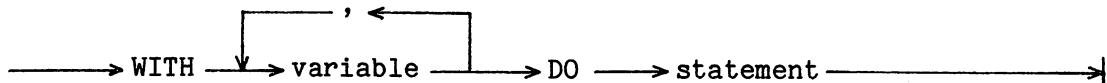
Examples

```
WHILE Z < Y DO BEGIN
    Z:=Z+1;
    Y:=Y DIV 2;
END
```

```
WHILE (PTR <> NIL) DO PTR:=PTR.PTRFIELD
```

WITH STATEMENT

with statement

Semantics

The variable must evaluate to a record. Within the statement, the definition of names is extended to include the field-names of the record, and these may be used without specifying their record prefix.

If several variables occur in the WITH expression part, the effect is the same as if they were written:

```
WITH V1 DO WITH V2 DO....S;
```

Efficiency

If the record variable involves substantial selection, the use of the WITH statement allows this to be carried out once on entry to the WITH statement. Subsequent accessing of the field may then be more efficient than re-evaluation. Use of WITH will never result in inefficient code in B6700/B7700 Pascal.

WITH STATEMENT

Examples

```
WITH PTR@ DO BEGIN
  IF (PTRFIELD <> NIL) AND (VALFIELD <> VAL)
  THEN PTR:=PTRFIELD;
END
```

```
WITH DISPLAY[J].NAMEFIELD@ DO BEGIN
  *****
  *****
END
```

6. PROGRAM UNITPROGRAM UNITExplanation

This section describes the construction of executable program units. There are three types of units:

- * a Pascal program,
- * a procedure, and
- * a function.

The resemblances between these three are very close, and the differences are essentially confined to their headings and usage, not to their internal structure.

The section also contains a description of

- 1) forward declaration of procedures and functions: a facility provided in the Pascal language for separating the declaration of the procedure heading from the declaration of its internal description.
- 2) external declaration of procedures and functions: a facility provided in B6700/B7700 Pascal to allow a Pascal program to use subprograms written in another language.

EXTERNAL DECLARATIONS

EXTERNAL DECLARATIONS

Syntax

external procedure declaration

—————> heading part ———> ; ———> EXTERNAL ———> ; —————>

Semantics

Procedures and functions may be declared to be EXTERNAL, ie. they are not included in the current program compilation but are 'bound in' to the program before execution. These procedures or functions are written in another language and quite often they are part of a scientific subroutine library.

This facility is provided in B6700/B7700 Pascal so that users may access such subroutine libraries. Only subroutines written in another language may be declared external - there is no facility for separately compiling a Pascal subprogram.

The compiler option, BINDINFO, must be SET to use external declarations.

Example

```
PROCEDURE EXTERN(R:REAL);  
EXTERNAL;
```

FORWARD DECLARATIONSSyntax

forward referenced declaration

→ heading part → ; → FORWARD → ; →

procedure or function heading

→ PROCEDURE → name → ; →
 → FUNCTION →
 → declarationpart → body → ; →

Semantics

It is sometimes necessary to be able to call a procedure or function before it can be declared. Typically this arises in program units that are mutually recursive, or at least which call each other. In such cases the procedure or function declaration may be broken into two parts. The first appears before the first use of the procedure and defines its parameterlist, name and type. This includes what has been called the heading part above: all that part of a procedure or function declaration up to the first semicolon. The second part appears later, and here the heading is now abbreviated to simply the reserved word PROCEDURE (or FUNCTION) and the name of the unit, followed by the rest of the procedure.

FORWARD DECLARATIONS

Example

```
PROCEDURE ANALYSE(C:COUNTER);
```

```
    FORWARD;
```

```
    {and here would be included some procedures that  
    call ANALYSE}
```

```
PROCEDURE ANALYSE;
```

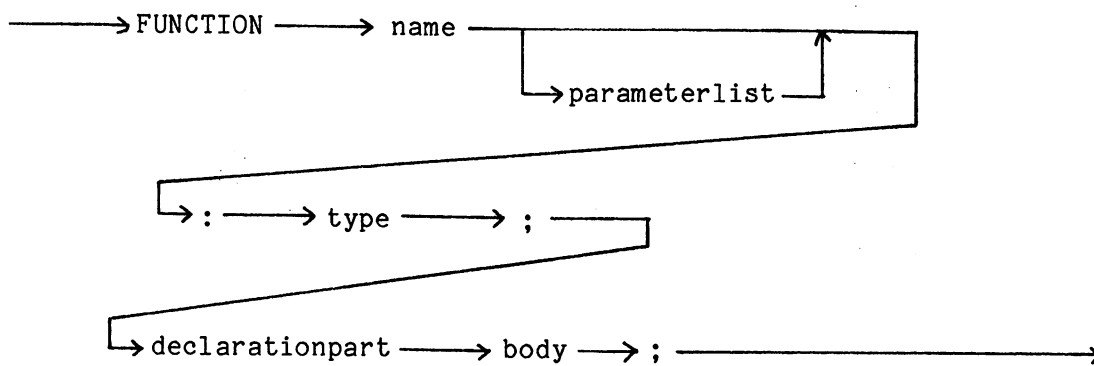
```
    TYPE .....
```

```
    {and here is the body of ANALYSE which makes some calls  
    on the procedures above it}
```

```
END;
```


FUNCTIONSyntax

function

Semantics

The name is the name of the function and is used both to refer to it, and to refer to the value returned by the function. The type of a function may be a scalar type, integer, real, pointer or subrange type. The parameterlist, if present, specifies the types and method of parameter passing of variables imported into the function from the caller. The function is invoked by a function call: an appearance of the function name in an expression context with an actual parameter part.

A B6700/B7700 Pascal function may be recursively used: the appearance of the function name in an expression context within the function itself forces such recursive use. The appearance of the function name in a left-hand side assignment context implies a setting of the returned value.

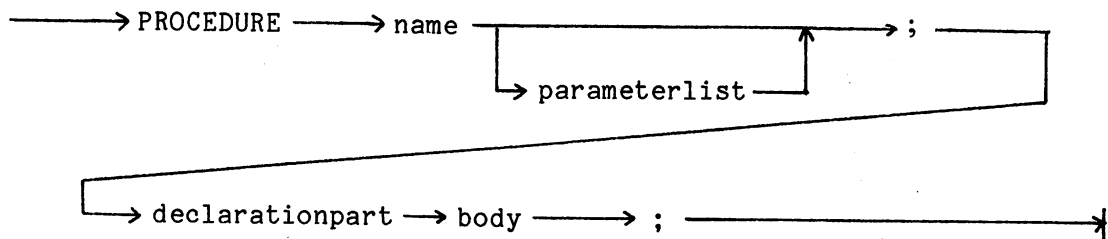
The execution of a function which returns without giving a value to the function is not defined in Standard Pascal. In B6700/B7700 Pascal such an event causes the program to be terminated with the INVALID OPERAND interrupt.

FUNCTION

Examples

```
FUNCTION COSH(X : REAL) : REAL;  
BEGIN  
    COSH := (EXP(X) + EXP(-X))/2;  
END;
```

```
FUNCTION CHOPPED(J,LOWER,UPPER : INTEGER) : INTEGER;  
    VAR RESULT : INTEGER;  
BEGIN  
    IF(J < LOWER) THEN RESULT:=LOWER  
        ELSE IF (J > UPPER) THEN RESULT:=UPPER  
            ELSE RESULT:=J;  
    CHOPPED:=RESULT;  
END;
```

PROCEDURESyntaxSemantics

The name is the name of the procedure and is used to refer to it. The parameterlist, if present, specifies the types and method of parameter passing of variables imported into the procedure from the caller. A procedure returns no one value, but specifies a computation which is to be carried out when the procedure is invoked by a procedure call.

B6700/B7700 Pascal procedures may be recursively used; the only limit is the available stack and memory space.

Examples

```

PROCEDURE PrintARowOfStars;
  VAR j: integer;
BEGIN
  FOR j:=1 TO 132 DO write(output, '*');
  writeln(output);
END;

```

```

PROCEDURE EXCHANGE(VAR A,B : REAL);
  VAR TEMPORARY : REAL;
BEGIN
  TEMPORARY:=A; A:=B; B:=TEMPORARY;
END;

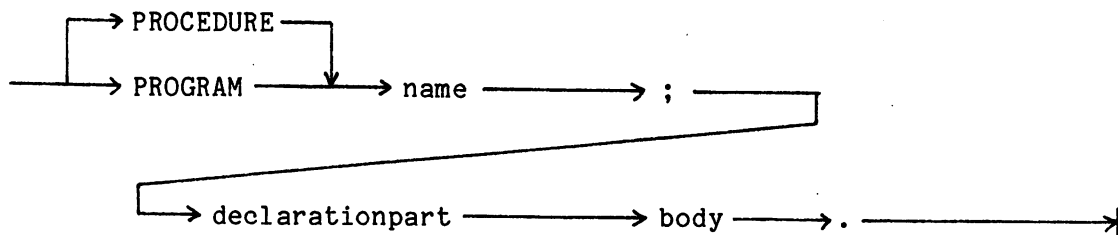
```

PROGRAM

PROGRAM

Syntax

program



Semantics

A program is very similar to a parameterless procedure. It may however begin with the reserved word PROGRAM as in standard Pascal, and it is terminated by a point after the END closing the program body. This must be on the last record of the file.

The program name is not used for any purpose in B6700/B7700 Pascal.

A program is the compilable unit for the B6700/B7700 Pascal compiler.

Example

```
PROGRAM PASCALCROSSREFERENCER;  
  CONST .....;  
  TYPE .....;  
BEGIN  
  {and the whole of a program body}  
END.
```

Standards

The use of 'file parameters' as in Pascal for the CDC Cyber series computers causes a warning 'note' to be printed; this part is not parsed by the B6700/B7700 compiler as attachment to external files are handled otherwise.

7. PRE-DEFINED PROCEDURES

In B6700/B7700 Pascal there exist a number of procedures and functions which are available to the Pascal programmer as though they had been declared in a procedure surrounding the Pascal program. They may all have their name redefined by the Pascal programmer in accordance with the usual scope rules. Pre-defined procedures are placed in Pascal for one or more of the following reasons:

- * the procedure/function is part of standard Pascal,
- * the procedure/function offers an important facility which cannot otherwise be achieved in Pascal,
- * the procedure/function offers very efficient implementation since special B6700 facilities can be used,
- * the function is a common arithmetic function, built into the B6700/B7700 system as an intrinsic,
- * the procedure/function exists in Pascal for the CDC Cyber series and seems worthwhile for compatibility reasons.

In the rest of this section, procedures, or functions which are not included in standard Pascal are marked with the symbol "*" at their first occurrence or definition.

The procedures which are additionally compatible with Pascal for the CDC Cyber series are halt, card, and random.

The pre-defined procedures used for i/o and file control are documented in a separate section.

Some of the pre-defined procedures may be used as actual procedure or function parameters to a procedure/function invocation. Those which can be passed as parameters are all of the arithmetic functions plus the function RANDOM. None of the others may be passed as parameters because they involve in-line code.

ARITHMETIC FUNCTIONS

ARITHMETIC FUNCTIONS

All the following arithmetic functions return a real result, and are implemented by calls on the B6700/B7700 operating system intrinsic for that function. The algorithm used is therefore described in standard Burroughs literature.

FUNCTION DECLARATION	DESCRIPTION
sin(r:real):real cos(r:real):real tan(r:real):real *cotan(r:real):real	standard trigonometric functions
arctan(r:real):real *arctan2(r1,r2:real):real *arcsin(r:real):real *arccos(r:real):real	inverse trigonometric functions (where arctan2(r1,r2)= arctan(r1/r2) but division by zero is avoided)
*sinh(r:real):real *cosh(r:real):real *tanh(r:real):real *atanh(r:real):real	standard and inverse hyperbolic functions
exp(r:real):real ln(r:real):real *log(r:real):real	e to the power r natural log of r log of r to base 10
sqrt(r:real):real	square root of r
*erf(r:real):real *erfc(r:real):real *gamma(r:real):real *lngamma(r:real):real	error function complementary error function gamma function natural log of gamma function

MARK AND RELEASE

*MARK and *RELEASE are a pair of non-standard procedures which permit the Pascal programmer to manage the allocation of space in the heap by operating it as a user-controlled stack. The procedures are implemented by special code, but are effectively declared as:

```
procedure mark (var p:pointer);
```

```
procedure release (p:pointer);
```

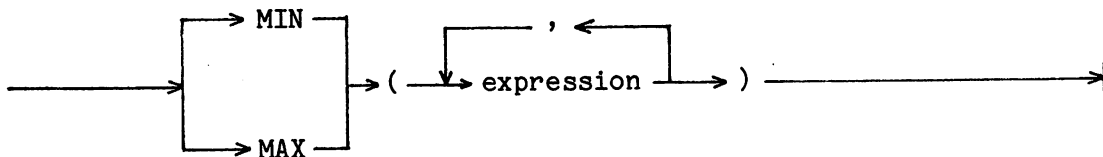
The pointer may be any pointer type (its reference-type is not relevant). A call to MARK sets p to hold a copy of the current top-of-allocated-heap pointer which is otherwise inaccessible to the Pascal programmer. A call on RELEASE with this same value as parameter will cause the current-top-of-heap pointer to be reset to the preserved value, thereby allowing the space higher in the heap address space to be re-allocated.

If any pointers retain values pointing above the new top-of-heap as a result of older NEW calls, they are not thereby invalidated, and if they are used the results will involve accidental remappings. The preserved value of the top-of-heap pointer should not be processed in any way, nor assigned to other variables.

MIN AND MAX

MIN AND MAX

Syntax



Semantics

The expressions in the parameter list must all be compatible (of the same type or subranges thereof) and must be an ordered type (scalar or real). The *MIN function returns the value in the list which is least in the ordering (or numerically smallest) and *MAX returns the value in the list which is largest in the ordering (or numerically greatest).

MAX and MIN are implemented by in-line code which evaluates each expression and compares it with a putative result held in the stack.

The parameter list of MAX and MIN may have any number of parameters, there is no limit.

Example

```
DSPEC:=MIN(12,MAX(DSPEC,0));
```

```
LENGTH:=MAX(WIDTH,SIZE,LEFTINLINE);
```


MIXED-TYPE FUNCTIONS

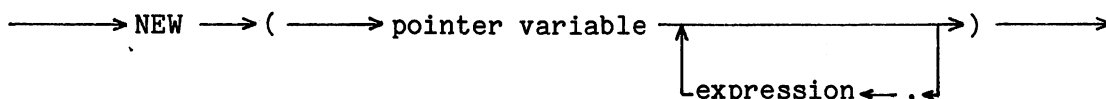
All the functions listed here are (in some sense) transfer functions between types. Some are generic: they will accept as argument any variable of the appropriate type kind.

<code>trunc(r:real):integer</code>	returns the integer value i which satisfies $i \leq r < (i+1)$
<code>round(r:real):integer</code>	returns the integer value i which satisfies $\text{abs}(r-i) \leq 0.5$
<code>odd(i:integer):boolean</code>	equivalent to $((i \bmod 2) = 1)$
<code>chr(i:integer):char</code>	returns the value as a char type whose ordinal number in the enumeration is i .
<code>ord(s:scalar):integer</code>	returns the ordinal number of the scalar s in the enumeration. May be used with any scalar type, including char
<code>*card(s:set):integer</code>	returns the number of elements in the set s .

NEW

NEW

Syntax



Semantics

A call to the NEW procedure allocates sufficient space in the run-time heap to accommodate an object of the reference-type of the pointer variable, and sets the pointer variable up to point at that newly allocated space. In the simple form of the call, sufficient space is allocated for the object (array, real, scalar, or set types), or sufficient space for the largest variant (record type). The form with expressions is applicable only to record-types with variants, and the expressions must be of the same type as the successive variant-case selection fields in that record-type. In this case the NEW procedure will allocate only sufficient space to hold the particular variant selected. It follows therefore that such allocated space must never have its variant-case selection fields altered during its lifetime. There is however no compile-time or run-time check on this error.

The size of the Pascal run-time heap is controlled by the compiler option HEAP, and the deallocation of allocated space can be achieved in a limited way by use of the MARK and RELEASE pre-defined procedures.

Examples

```
NEW(NODEPOINTER);
```

```
NEW(PERSONPOINTER, MALE);
```

OPERATING SYSTEM PROCEDURES

The following procedures communicate directly with the operating system to inform it of a desired action, or enquire of the environment.

<code>*halt</code>	causes the program to cease execution. The job-description sheet carries the annotation "P-DSED" (program discontinued) and a call-history of the point of invocation of halt.
<code>*startjob(var f:file)</code>	the file, which must be closed, and which is presumed to contain WFL (job control) statements in EBCDIC, is put in the operating system's queues for initiation as an independent job.

PACK AND UNPACK

PACK AND UNPACK

These two procedures are provided so that data may be transferred between packed and unpacked arrays.

Explanation

If a is an array variable of type

array [m..n] of T

and z is a variable of type

packed array [u..v] of T

where $(n-m) \geq (v-u)$ then

1) pack(a,i,z) is equivalent to:

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

2) unpack(z,a,i) is equivalent to:

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

In both cases, j denotes an auxiliary variable not occurring elsewhere in the program.

PASCAL GENERIC FUNCTIONS

The Pascal generic functions are all standard Pascal, and are all defined over a range of Pascal types.

<p>abs(n:numeric):numeric</p>	<p>returns the magnitude (absolute value) of the parameter value. The result is of the same type as the parameter, which must be integer or real, or a subrange thereof.</p>
<p>sqr(n:numeric):numeric</p>	<p>returns the square (n*n) of the parameter value. The result is of the same type as the parameter, which must be integer or real, or a subrange thereof.</p>
<p>pred(s:scalar):scalar succ(s:scalar):scalar</p>	<p>return the predecessor or successor value of the parameter value respectively. The result is of the same type as the parameter, which may be any scalar type including char and integer. If the application of pred or succ causes the range of the scalar (as declared) to be exceeded, the program is terminated with the message "SCALAR RANGE OFLO" printed on the job description sheet.</p>

RANDOM

RANDOM

*RANDOM returns a real value which is a pseudo-random number uniformly distributed over (0,1). It acts as though declared:

```
function random(var r:real):real;
```

though it is implemented by a call on an operating system intrinsic.

The var parameter is used to supply the seed for each successive pseudo-random number, and it is changed by the function during each call. The parameter is necessary as random may be generating pseudo-random numbers for several users simultaneously. It is therefore necessary for the user to declare a real variable he will otherwise never use in the outermost program unit, and to initialize it to some arbitrary value. (Using one of the time functions will give a really random start, but is unrepeatable.) This variable is then given to random in every call to it. If the program is re-run with a different initialization, a different subsequence of the pseudo-random cycle will be generated. For example:

```
PROGRAM MONTECARLO;
  VAR
    RANDOMSEED : REAL;
    ....

BEGIN
  ...

  {initialize the seed}

  RANDOMSEED := 1977;
  ...

  FOR J:= ..... DO BEGIN
    X:=RANDOM(RANDOMSEED);
    ...

  END;
  ...

END.
```

TIME PROCEDURES

There are four procedures which deal with time. The first three are similar and are parameterless functions returning the amount of the appropriate time in seconds that has passed since the start of the task (here the execution of a Pascal program). Since these are derived from a clock which increments every 2.4 microseconds, the graininess of the real result is 2.4E-6. These functions are intended for monitoring program performance, and they are:

*elapsedtime:real	returns actual elapsed time since midnight
*processtime:real	returns time spent by processor since start of task
*iotime:real	returns time spent by i/o processor since start of task

The other time procedure serves a very different purpose: it allows a program to enquire of the epoch (date and time) primarily for purposes of documentation, but also for validation purposes. It is a procedure declared as:

```
*procedure timestamp(var a:array[0:5] of integer)
```

The array is returned with a[0] holding the year number (for example 1977), a[1] holding a month number (1..12), a[2] holding a day-of-month number (1..31), and a[3], a[4], a[5] holding the hour, minute and second respectively in the ranges (0..23), (0..59), (0..59). Standard Pascal programs can format this into a convenient timestamp output, and to compute the day-of-the-week from it.

8. INPUT AND OUTPUTINPUT AND OUTPUT

Input and output in B6700/B7700 Pascal is achieved by means of calls on pre-defined procedures. The syntax of some of these procedures is peculiar, and therefore all the pre-defined i/o procedures have been collected into this section.

B6700/B7700 Pascal supports the concept of textfiles as defined by standard Pascal. Textfiles are considered as file of char.

Other forms of i/o supported are:

- * i/o on files with structured components eg. file of array ... GET and PUT perform binary i/o on these files and the procedures READ and WRITE access the components of the files in a stream oriented manner.
- * record-oriented formatted i/o (a FORTRAN-like system), through the procedures READREC and WRITEREC and FORMATS.
- * binary i/o on files of a predefined type (eg. file of integer,...) through the procedures GET and PUT.

While all forms may co-exist in a Pascal program, it is recommended that only one form is used on a particular file. Mixing stream-oriented i/o with other types is particularly likely to give synchronization problems as the stream text is held in buffers internal to the program until empty (full).

In record-oriented i/o, the transactions are always expressed in integral numbers of file-components. In stream-oriented i/o the file is regarded as a continuous stream of characters, with a line-marker separating each file-component (reckoned as a line).

Standards

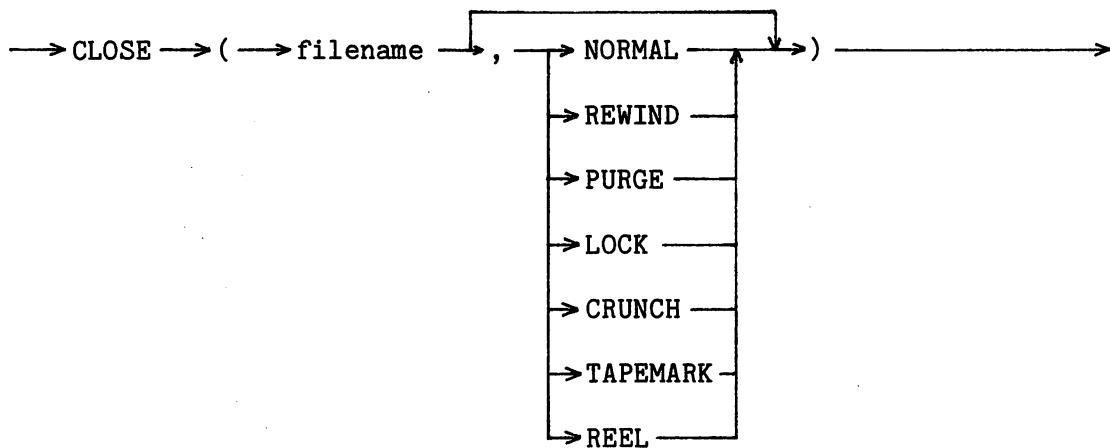
Record-oriented formatted i/o, and all calls which involve a random-access to a file, are not standard Pascal.

CLOSE

CLOSE

Syntax

close



Semantics

The CLOSE call causes the nominated file to be closed.

If the NORMAL option is specified, the action depends on the file KIND. A PUNCH file has a card with an ending label written; a PRINTER file is skipped by SKIP 1 (and possibly an ending label written); a TAPE file has a tapemark written and the file is rewound; a temporary DISK file has its space returned to the system. (This corresponds to the default interpretation of the Burroughs Algol CLOSE procedure.)

If the REWIND option is specified, the file is rewound if it is a TAPE file or the record pointer is reset to the beginning of a DISK file. (This corresponds to the Burroughs Algol REWIND procedure.)

If the PURGE option is specified, the file is closed and purged from the system directories. If the file is a permanent disk file, its space is returned to the system.

The LOCK and CRUNCH options specify that the file is to be closed, and in the case of a DISK file, entered as a permanent file in the disk directories. Additionally in the case of CRUNCH, unused space

in the last AREA of the file is trimmed from it, thereby making it impossible to extend it in situ later.

If the REEL option is specified, the file must be a multi-reel tape file. The current reel is closed and the next reference of the file implicitly opens the next reel.

The TAPEMARK option is used in conjunction with multi-file tapes. The file is closed but the tape remains positioned past the tapemark so that the next file can be read or written. (This corresponds to the Burroughs Algol CLOSE(file,*) use.)

Standards

The CLOSE procedure is not standard Pascal.

Examples

```
CLOSE(DISKFILE);  
CLOSE(INPUT2,REWIND);  
CLOSE(STATISTICS,CRUNCH);
```

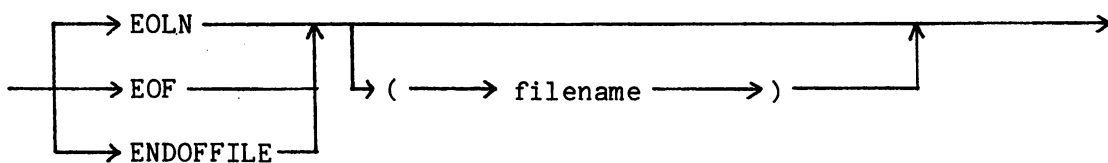
Note

The words NORMAL, REWIND, etc. are not reserved words in the system, and have no meaning in any other context except as defined by a programmer. In the context of a CLOSE call, only these words are permitted in the second parameter place.

EOLN EOF AND ENDOFFILE

EOLN, EOF AND ENDOFFILE

Syntax



Semantics

These are all boolean functions. EOLN returns TRUE if the file has been opened for reading or the last READ on this file was just past the end of the line. (The character returned if it was a character read will be a space under this condition.) Otherwise EOLN is FALSE. The effects of testing EOLN on a written file, or other than with stream-oriented i/o, are undefined. The next READ on the tested file (if EOLN was TRUE) will cause a new line to be read. EOF is valid for all file-status, and returns TRUE if

(i) the file has been positioned past the end-of-file
or (ii) the file is a stream-oriented file in write-status.
ENDOFFILE is similar to EOF, but returns TRUE only under condition (i), regardless of whether the file is being read or written. If the filename and parentheses are omitted, the standard file INPUT is assumed.

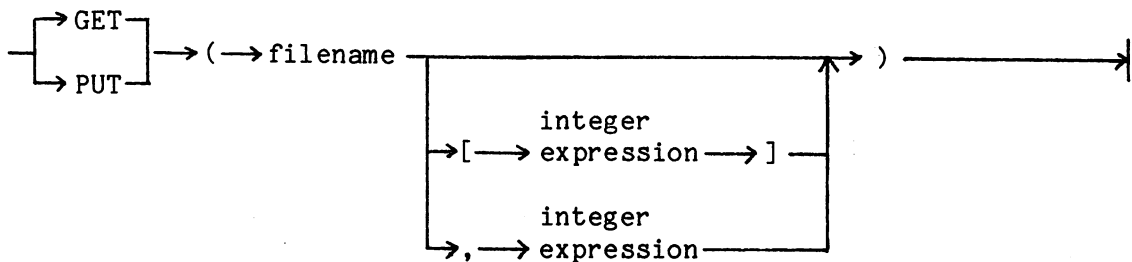
Standards

EOLN and EOF are standard Pascal; ENDOFFILE is provided to remedy the deficiencies of the Pascal EOF function when writing files that may reach end-of-medium, or end-of-allocated-areas, or be denied terminal access.

Examples

```
WHILE NOT EOLN(INPUT) DO READ(INPUT,CH);

STATUS:=EOF(INPUTD);
```

GET AND PUTSyntaxSemantics

For textfiles or files of a predefined type, execution of a GET call will advance the pointer to the next component, or set eof. A PUT call shall cause the file buffer variable to be appended to the file. In these cases the random-access options are not permitted.

For other files, after execution of a GET call, the file-buffer of the nominated file is filled from the file. The normal effect is to transfer the next file-component of the file to the file-buffer; if however the form with square brackets is used the integer expression is evaluated and the file-component referenced by the expression is read.

After execution of a PUT call, the file-buffer of the nominated file is written to the file. The normal effect is to write immediately following the preceding transfer into the file; if however the form with square brackets is used the integer expression is evaluated and the file-component referenced by the expression is written.

If a file has never been read, or has been reset, or rewound, the contents of the file-buffer are undefined. In B6700/B7700 Pascal, the execution of a PUT call does not alter the contents of the file-buffer.

The alternative forms are equivalent, and allow random-access to components of the file. The components are numbered from 0 upwards in steps of 1. (GET(FILEX[0]) will read the first component of the file FILEX.)

GET AND PUT

Examples

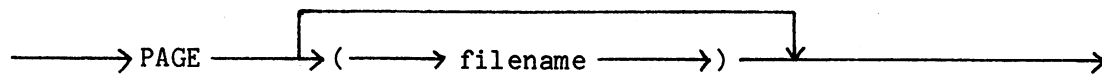
```
GET(INPUT);  
GET(DISKFILE[K+1]);  
PUT(REMOTEFILE);  
PUT(DISKFILE,5);
```

Standards

Random-access is not standard Pascal.

PAGESyntax

page

Semantics

Execution of the PAGE call causes a "SKIP 1" command to be given to the file. If it is a (KIND=PRINTER) file, this causes a skip to the top-of-form, and the next line written will be printed at the top of the page or form. If the filename and parentheses are omitted the standard file OUTPUT is assumed.

Example

PAGE(OUTPUT)

PRE-DEFINED FILES

PRE-DEFINED FILES

Semantics

Two files are always included in compiled B6700/B7700 Pascal programs. Their names are INPUT and OUTPUT, and references to them are presumed when a filename is omitted in a READ or WRITE statement respectively (or the READLN, WRITELN, READREC, WRITEREC forms).

The scope of the names is as if they had been declared in a block surrounding the program. In other words, if INPUT or OUTPUT are used as names for any purpose within a Pascal program, that definition over-rides the pre-defined meaning for these names. (But the default action for READ and WRITE will still reference the same files which are otherwise inaccessible.)

The default declarations for these pre-defined files are:

VAR

```
INPUT  : TEXT;
```

```
OUTPUT : TEXT;
```

The default attributes for these predefined files are:

```
INPUT: (KIND=READER, INTMODE=EBCDIC, MYUSE=IN)
```

```
OUTPUT: (KIND=PRINTER, INTMODE=EBCDIC, MYUSE=OUT)
```

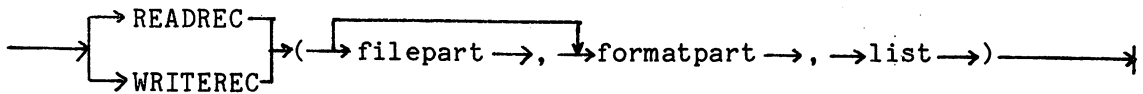
The file attributes may, of course be over-ridden by a Work-Flow statement.

Implementation

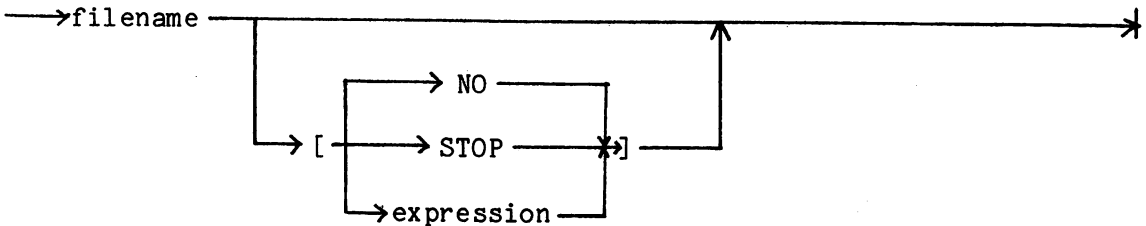
These two files and their associated buffer and state segments are always included in the stack activation record of the outermost (D2) level of a Pascal program. This causes any locally declared variables to be allocated higher in the stack than would otherwise be the case but has no other significant effect.

READREC

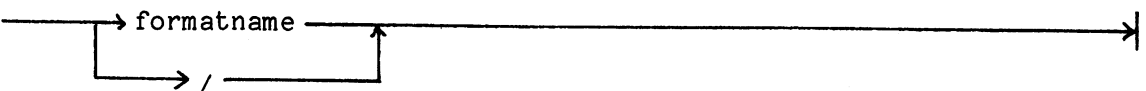
read or write (record-oriented)



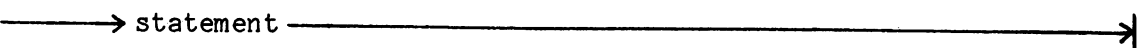
filepart



formatpart



list



READREC

Semantics

The filepart specifies where the record or records are to be written or read, and contains the filename (internal Pascal form), and optionally some specifiers in square brackets. The expression, if present, is interpreted as specifying a random access read or write starting at the record specified by the integer value of the expression. The NO and STOP qualifiers have the same effects as in Burroughs Algol: READ(FILEX[NO],...) causes the read-buffer to be left marked not-empty at the end of the read process so that subsequent READ(FILEX,...) can re-interpret the record. WRITEREC(FILEX[STOP],...) suppresses the newline at the end of an output record after a write to a remote terminal. This is useful for input prompt messages. The filepart (including the separating comma) may be entirely omitted, when the pre-defined file INPUT (for a READREC) or OUTPUT (for a WRITEREC) are assumed.

The formatpart either specifies the name of a format declared earlier in the text which is to be used in the interpretation of the record, or is a slash, meaning that the Burroughs Algol free-form i/o rules are to be applied.

Important: The Algol i/o rules should not be confused with the Pascal stream-oriented i/o system. Algol free-form i/o is still record-oriented: each read begins a new record and each write starts a new line. The format part may not be omitted in a record-oriented i/o statement. In Pascal free-form i/o each data item is separated by a space and not a comma as in Algol.

The listpart specifies the items to participate in the formatting, and since the specification of these may require computation, has the form of a statement. The only differences are that within a read- or write-list the following are forbidden:

- * labels
- * goto statements

and the following are newly permitted:

- * variables
- * expressions

The appearance of a naked variable in a read-list, or of a naked variable or a naked expression in a write-list, is interpreted as a request for that variable or that value to participate in the formatting process. All other statement constructs operate the same as they do outside the list context. The most useful constructs are FOR, WHILE and WITH. This list-structure minimizes the number of rules to be learnt or re-learnt in the B6700/7700 Pascal language

while conferring considerable expressive power. However programmers should resist the temptation to include substantial computational tasks in a read- or write-list, as this will obscure the structure of the program.

It is worthwhile explicitly pointing out some consequences of this simple and attractive list structure as it will not be familiar. Firstly, BEGIN/END brackets are available for lumping variables together, and will often enclose the whole list. Secondly, an indentation structure can be used to show the relationships in the list in the source text. Thirdly, the separators of statements in the list are semicolons, not commas. Since it is possible to have procedure and function calls in a list, it is forbidden to attempt to carry out further i/o on the same file in such activations of procedures or functions external to the list. An attempt to do so will cause the program to be terminated in execution.

Standards

Record-oriented i/o is not part of standard Pascal.

Implementation

The execution of an i/o statement can be explained as follows. The first action is to call the formatting procedure (a B6700 "intrinsic" procedure) passing to it the file-name, a file-buffer, and the format description. The formatting procedure attempts to interpret the format as far as it can without variable values, then calls the list as a procedure. The list executes until it finds a variable or expression, when it calls a procedure nested in the formatting intrinsic either passing or receiving a value, but also passing some typing information. This nested procedure continues the formatting actions as far as possible, and then returns. These two steps are repeated as often as there remains code to be executed in the list. When the list is completed, it returns to the formatting procedure, which tidies up, empties the buffer and releases the file, and returns to the caller.

Since the intrinsic used is the standard Burroughs Algol/FORTRAN formatting intrinsic procedure, any error treatment is that determined by Burroughs. The error numbers are explained in the relevant Burroughs documentation.

READREC

Examples

```
READREC(XFILE,LAYOUT1,J);

READREC(LAYOUT2, BEGIN J; K; X END);

READREC(DFILE[K], FORMATOFDRECORD,
  FOR J:=1 TO 10 DO V[J]);

WRITEREC(DFILE[K+KINCREMENT], FORMATOFDRECORD,
  BEGIN
    Z+SQR(R)/5;
    FOR J:=1 TO 8 DO BEGIN
      V[J+1];
    END;
  END);

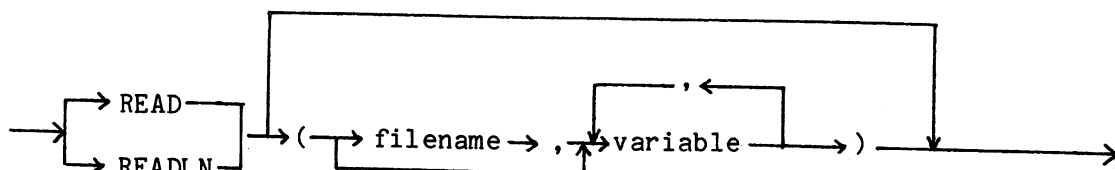
WRITEREC(OUTPUT, FORMATTREE,
  BEGIN
    PTR:=HEADOF TREE;
    WHILE (PTR <> NIL) DO BEGIN
      WITH PTR @ DO BEGIN
        NODEVALUE;
        PTR:=NODELEFTLINK;
      END;
    END;
  END);

WRITEREC(OUTPUT,/,
  BEGIN KMAX/2; FOR K:=1 TO KMAX/2 DO BVEC[J] END);

WRITEREC(REMOTEFILE[STOP], PROMPTFORMAT);
READREC(REMOTEFILE, RESPONSEFORMAT, K);
```

READSyntax

read



(NOTE: a stream-oriented READ has no format.)

Semantics

A READ or READLN call causes the nominated variables (which must be of pre-defined types boolean, integer, char or real, or programmer-defined scalar type) to have their values replaced by a value read and interpreted from the file regarded as a stream of characters.

If the filename is omitted, the pre-defined file INPUT is assumed.

If no variables are listed, no change is made to the file except for the READLN action.

If READ is used, the file character pointer is left pointing in the stream where the procedure leaves off processing the last value. If READLN is used, the file character pointer is moved on from that point to the end of an input line. Except for the action of READLN, and the rules for validity of tokens, the file-component boundaries (lines) have no meaning.

The action of READ is quite different for the case where the variable is of type char, as opposed to any other type. A READ of a character simply returns the next character in the input stream, regardless of what it is. If the pointer moves past the end-of-line, a ' ' character (space) is given as the char value and the function eoln becomes true. The next READ of a character will give the first character of the next line.

All other types of variables correspond to lexical tokens represented

READ

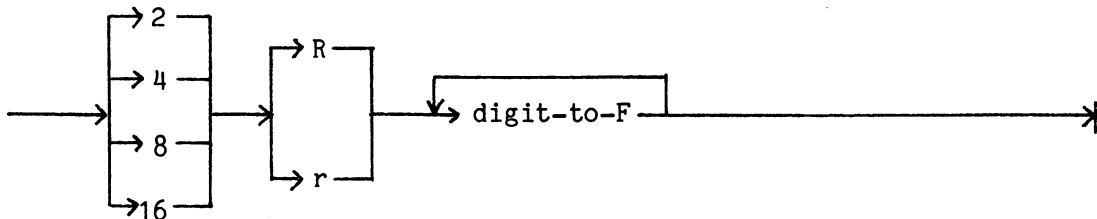
in the input stream. These tokens are subject to exactly the same rules as in the B6700/B7700 Pascal language itself, with the additional feature that real values can be extracted from objects that conform to integer-constant syntax and to real-constant syntax. For files with a structured component, the input stream is scanned for a character which might be part of a token (the alphabets A-Z, a-z; the digits 0-9; the underline; or the number-characters +-.), all other characters being ignored by this scan. For textfiles only spaces are ignored. If necessary, new lines are read until a putative token is found. The character and those following it are then scanned in accordance with the expected token rules until an incompatibility signals the end of the token. The value, if valid, is returned into the variable.

If the token is invalid for this type, a run-time error is caused and the program is terminated. In most circumstances this is a sign that the input stream and the program have irrevocably lost synchronism. In a few cases, this fatality should be suppressed and this can be done (see Pascal READ ERRORS).

It is recommended that tokens on the input stream be separated by the space character.

The radix-based notation for real numbers allows an exact bit pattern to be specified. The syntax is given below. The first integer (to base 10) is the radix to be used, and must be 2,4,8, or 16. The second string is of digits to the radix specified (using A-F for digits 10-15). This is only available for files of a structured component.

radix-based notation



Stream-oriented i/o is available on textfiles and on files with a structured component, eg. file of packed array [0..79] of char; Stream-oriented i/o on files of a structured component was the standard form of i/o before release 3.0.001 of the B6700/B7700 Pascal compiler but with the introduction of textfiles it is no longer standard. Users should note that the two forms of i/o do not produce identical results.

Standards

The radix-based notation has no counterpart in standard Pascal.

Scalars defined by the programmer, and boolean values, may not be read in standard Pascal.

The allowable input token forms may vary slightly between different Pascal compilers due to implementation differences.

Examples

```
VAR
    RA, RB : REAL;
    J      : INTEGER;
    B      : BOOLEAN;
    S      : (YES, NO, MAYBE);
BEGIN
    READ(INPUT, RA, RB);
    READ(INPUT, J);
    READ(INPUT, B, S);
END.
```

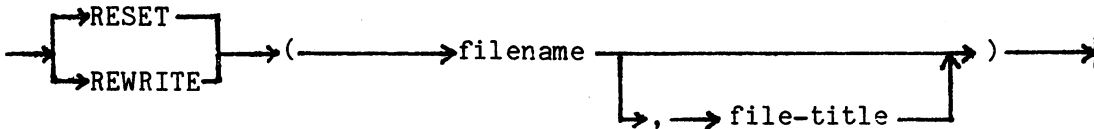
The input file:

```
12.5 -45.7654E-8
10 TRUE
MAYBE
```

RESET AND REWRITE

RESET AND REWRITE

Syntax



Semantics

The RESET and REWRITE calls are both identical in effect to:

```
    CLOSE(filename,REWIND);
```

and then the file is opened.

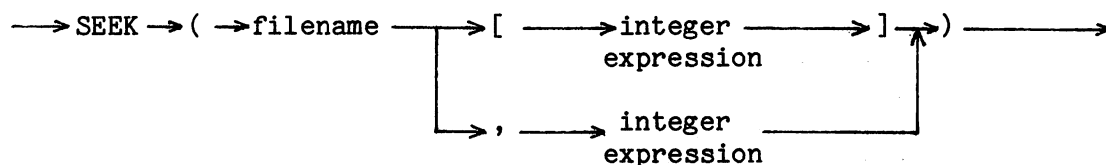
On a RESET the buffer is filled with the first file component. The file-title must be a string constant or a string variable and must be terminated by a "." to satisfy the B6700/B7700 syntax. The title is changed after the file has been closed and before the new file is opened.

Standards

Use of the file-title in a RESET or REWRITE is not standard Pascal.

SEEKSyntax

seek

Semantics

The seek procedure allows programs doing random-access actions on a file to achieve overlap of processing and i/o. The integer expression is evaluated and the file-component referenced by the expression value is read into an internal buffer attached to the file-information block. When subsequently a READ or GET is received for this record, it is made immediately available.

The two forms of the seek call are equivalent, and are provided as syntactic sugar in case the call is written analogously to the GET and PUT procedures, or in accordance with standard Pascal rules.

Overlap of i/o and processing is automatic for sequentially accessed files.

Standards

The seek procedure is not standard Pascal.

Example

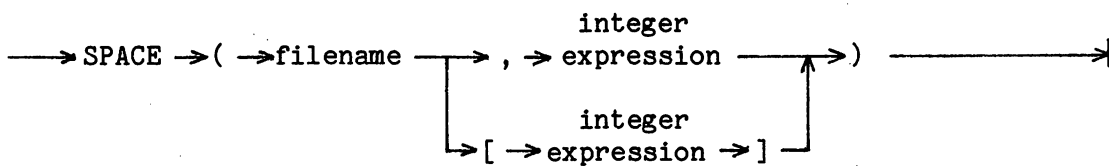
```
SEEK(DISKFILE[K+1]);
```


SPACE

SPACE

Syntax

space



Semantics

The integer expression is evaluated and the appropriate number of file-components are ignored in a sequential READ process. The value must be greater than zero. If the SPACE call is used for writing to a (KIND=PRINTER) file, blank lines are written to the file.

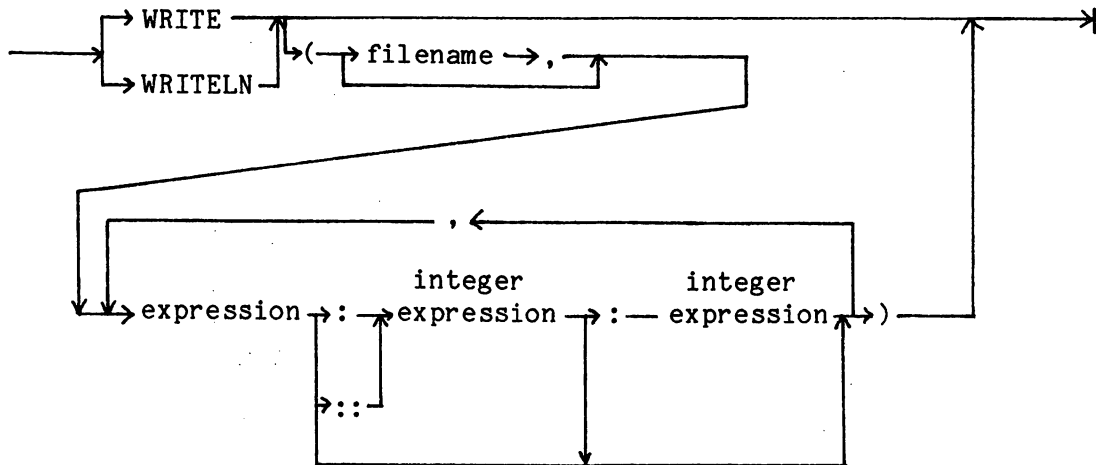
WRITEREC

Record-oriented write statements are completely treated in the section which deals with record-oriented read statements.

WRITE

WRITE

Syntax



Semantics

A WRITE or WRITELN call causes the nominated expressions (which must be of pre-defined types boolean, integer, char or real, or programmer-defined scalar types) to be written out to the nominated file by a character representation. The integer expressions allow some user-control of the layout.

If the filename is omitted, the pre-defined file OUTPUT is assumed. The filename must be explicitly stated if the first expression involves a file buffer.

If no expressions are listed, no writing takes place except for any WRITELN action.

If WRITE is used, the character pointer is left pointing into the character stream where the writing ceased. If WRITELN is used, then the character pointer is forced to flush out the remnant of the currently written line to the file just before the WRITELN returns. The character pointer is left at the beginning of a new line. If a token to be written will not fit on what is left of a line, the line-flushing is automatic and it will be printed on the next line automatically. No error occurs as a result of this action.

The action of WRITE is slightly different for the case where the expression is of char type, as opposed to other types. In this case, all char values are directly inserted into the text stream, except a CR character, which causes the line to be flushed out to the file.

All other values are represented by an external token. These tokens conform to the internal rules for B6700/B7700 Pascal tokens of the same type, except for literal strings which are printed without quotes. The printing may be controlled by the presence or absence of the two integer expressions, denoted here by "width" and "dspec". These are valid only for some conversions given by the table:

type	width allowed	dspec allowed
BOOLEAN	yes	no
SCALAR	yes	no
INTEGER	yes	yes/no
REAL	yes	yes
string	yes	no
radix-based	yes (compulsory)	yes

For all conversions, width is the desired width of the field into which the value should be converted. It should be sufficient to hold the external representation. A width of 0 is equivalent to omitting the width.

For INTEGER and REAL, the dspec value specifies a number of decimal digits. For integer the dspec value is not allowed on textfiles but on files with a structured component and it represents a guaranteed number of digits (thereby allowing non-zero-suppressed integers). For real, it is the number of digits desired after the decimal point. If dspec is present, a real number is converted in a form without an exponent (for example 8.141593). If it is absent or zero, an exponent is printed and (width-8) is taken as the dspec value provided this does not exceed 11 digits.

For radix conversions, dspec must be 0,2,4,8, or 16, and specifies the desired radix conversion base. A value of 0 corresponds to no value, and the default of 16 is used. A radix conversion occurs only if the first colon is doubled. In this case, any one-word object may be printed, including one-word sets and pointers.

Booleans and scalars are externally represented by the appropriate constant names in upper-case. Integers are represented conventionally; a - sign is printed for negative numbers but no sign

WRITE

is needed for positive numbers. Real numbers always have a place reserved for a sign and at least one digit each side of the decimal point. Two exponent digits are printed.

For textfiles, if no width specification is given, the defaults are 1 for type char, 15 for real, 5 for boolean and an integer is printed in the minimum space required.

For files of a structured type if no width is specified, default specifications come into effect. A char is printed in a width of 1 character place (except for CR), and a string in exactly the space it needs. Other types are preceded by one space, and are printed in the minimum space needed for the value (e.g. TRUE requires 4 spaces, -12 requires 3 spaces). Real values are printed in a width of 15, without an exponent if possible, otherwise with one.

Errors that occur during conversion are discussed under Pascal WRITE ERRORS.

Examples

```
WRITELN(OUTPUT, 'THIS VALUE = ', -15);  
WRITE(OUTPUT, 1.5:15:3); WRITELN(OUTPUT, 1.5:15);  
WRITELN(MAYBE);
```

output file:

```
THIS VALUE = -15  
1.500 1.50000000E+00  
MAYBE
```

9. COMPILER OPTIONS

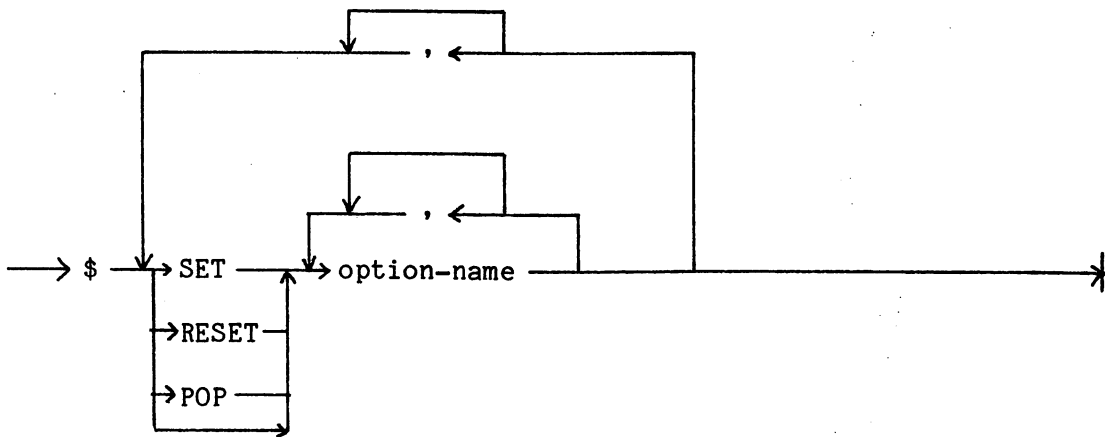
COMPILER OPTIONS

Explanation

Compiler options are specified on special compiler option records, and allow the user of the B6700/B7700 Pascal compiler to control some of the functions of the compilation, such as requesting a listing of the source text, or the inclusion of text, or the merging of two input files.

Syntax

compiler option record



NOTE: this syntax does not handle the HEAP, ERRORLIMIT, PAGE, SETSIZE, BIND, BINDER and INCLUDE options. These are separately described: the above syntax is for the boolean options (with an extension for the SEQ option).

COMPILER OPTIONS

Semantics

A compiler option record is recognised by the compiler since it has a \$ in the first character position of the record, or space followed by \$ in the first two characters of the input record. These two forms are identical in effect except as noted in the sheet on the NEW option.

The options which have a boolean value are referred to as being SET or RESET. Each boolean option has associated with it a stack of values (limited to a maximum of 48); if the option is reset by default the top-of-stack (and all the stack) are RESET, and if it is set by default, the top-of-stack alone is SET. Only the top-of-stack value affects the progress of compilation.

The appearance of an option-name after SET forces the stack to be pushed down by one (the old top-of-stack value becomes the second value) and the new top-of-stack value is SET. RESET functions identically except that the new top-of-stack value is RESET. Thus the following option record will cause LIST and LINEINFO to be SET and CODE to be RESET:

```
$ SET LIST, LINEINFO, RESET CODE
```

The appearance of an option-name after POP forces the top-of-stack value to be discarded, and the stack is 'popped': all elements move up one, the second value becomes the top-of-stack (and was probably an old state saved by a previous use of SET or RESET).

If the first item on an option record is not SET, RESET, or POP, then all the options listed subsequently are SET, and all those not listed are RESET. This is not particularly useful, but conforms to Burroughs Algol.

For an extension of syntax for SET, see the sheet on USER-OPTIONS; this also illustrates a use for the POP facility in conjunction with the OMIT option.

The options may appear in upper or lower case but are translated to upper case by the compiler before printing and analysis. This prevents confusion with user options and is consistent with the compiler's handling of identifiers.

List of options

Boolean-valued	numeric-valued	other
<p>\$ ASCII AUTOBIND BINDINFO BOUNDSCHECK CHECK CODE ERRLIST HEXCODE INCLNEW LINEINFO LIST LISTINCL MERGE NAMES NEW OMIT SEQ STANDARD STATISTICS STRIPBLANKS TRUSTWORTHY WARNINGS</p>	<p>ERRORLIMIT HEAP SETSIZE</p>	<p>BIND BINDER INCLUDE PAGE</p>
<p>user options</p>		

\$

\$ (Default: reset)

If this option is reset, option-records are not listed on the LINE file. If it is set, then option records are listed. Since the effects are immediate, an option record containing SET \$ will be listed, but one containing RESET \$ will not.

The option has no effect if LIST is reset.

ASCII (Default: reset)

If reset (the default) the Pascal program is compiled to hold all values and types involving type char as though they were held in the EBCDIC character set (as they in fact are).

If ASCII is set, then despite the B6700/B7700 standard EBCDIC character set, some variations are made to the compiled program so that it appears to carry out processing in an ASCII environment. In fact, the external files may remain in EBCDIC; translations are made so that internal Pascal values are held in the ASCII code. If ASCII is not set over the whole program, users should be aware that it affects the compilation of char or string constants, any declaration involving char, and stream-oriented read/write procedure calls.

NOTE: The formatted read and write procedures use the standard Burroughs B6700/B7700 formatting intrinsics which do not have provision for ASCII. These should not therefore be used if ASCII is set, or if they are, no char or string should be written with them.

The specific changes make all char and string constants use the ASCII representation internally (thereby affecting the values returned by ORD and the effects of CHR), and define type char as a scalar type which can be put into one-to-one correspondence with 0..127. A bit is set in calls to the read/write intrinsics to force external world translation as necessary.

The ASCII option is provided specifically to increase the usefulness of the B6700/B7700 Pascal compiler in that it can process a wider range of non-portable programs that assume the ASCII collating sequence, and so that the supposed portability of Pascal programs can be tested by switching them from one character set to another.

AUTOBIND

AUTOBIND (default: reset)

The AUTOBIND compiler option combines the processes of compiling and program binding into one job. During compilation the compiler produces a set of instructions to be passed to the binder. In many cases, these binder instructions are self sufficient for binding purposes and the user need not be concerned with binder control cards. In those cases where binder instructions are required, the user can insert binder control cards.

The AUTOBIND compiler option can be set or reset at any point throughout compilation but it is recommended that the option be set or reset only once at the beginning of the compilation.

When the option is set, the compiler option BINDINFO is also set.

BIND

BIND (no default status)

This \$ card is passed directly to the binder when autobinding.

BINDER

BINDER (no default status)

This option allows the passing of compiler options when autobinding. The compiler 'strips off' the word BINDER and passes the rest of the card intact as an option card to the binder.

BINDINFO (Default: reset)

The BINDINFO option instructs the compiler to include binder information in the code file. The option is reset by default but may be set at any point in the compilation. However, it is recommended that it be set or reset once only - at the beginning of the compilation.

It is necessary to use the option when procedures or functions are to be declared EXTERNAL in the program. The binder then needs the binder information in the code file to link in the externally compiled subprograms.

When the option is set, the size of the code file produced is almost twice as big as when the option is reset. For this reason, the option is reset by default.

The option is set when the compiler option AUTOBIND is set.

BOUNDSCHECK

BOUNDSCHECK (default: set)

If set, the Pascal program is compiled with bounds checking code inserted to check for certain run-time bounds error conditions. If reset, the bounds checking code is omitted. It is recommended that this option remain set unless a user is certain no bounds errors will occur and he wants the last ounce of speed from the program.

The bounds conditions checked are:

- (a) reading a subrange value
- (b) assigning an expression to a subrange variable
- (c) passing a subrange type as a value parameter
- (d) indexing a multi-dimensioned array. (Hardware effectively checks a single dimensioned array)
- (e) the parameter to the function CHR
- (f) assignments to short sets

The scheme used is that published by J. Welsh in "Economic Range Checks in Pascal", Software - Practice and Experience, vol. 8, pp85-97, 1978. It ensures that checking code is inserted only if there is doubt whether a value will lie in the required range.

CHECK (Default: reset)

If this option is set, the TAPE and NEWTAPE files are checked for sequencing errors in character positions 73-80 of the record. If any such errors are detected, they are noted by an appropriate message on the LINE file.

CODE

CODE (Default : reset)

If this option is set, the compiler produces a listing of the generated code on the file LINE. This listing is intermixed with the source code listing if the option LIST is also set.

The code listing is in symbolic form: labels and instructions are represented by mnemonic names, and these generally conform to those used in the B6700 or B7700 REFERENCE MANUAL. Instructions that occupy more than one syllable, and are primary mode operators, are printed with the details of the additional data between parentheses. In some cases the value is repeated to the right in hexadecimal format in case this interpretation is more natural. Examples:

```
VALC (04,00027)
BSET (31)      #1F
```

Multiple-syllable edit operators are printed in a special format, one line per syllable, as there are so many possible formats.

The compiler sometimes has to emit padding syllables, and it uses the NVLD instruction for this purpose (#FF). Such padding is not listed by the CODE option as it is not intended to be executed. The purpose of the option is to illustrate the generated code so that precise points of failure can be pinpointed, so that compiler flaws can be quickly diagnosed, so that esoteric behavioural points can be explained in terms of the machine's instruction set, and to show programmers what code corresponds to their constructs.

The NAMES option is also of use with the CODE option.

ERRLIST (Special default)

It will seldom be necessary for user-programmers to set or reset the ERRLIST option explicitly.

If a compilation is initiated by CANDE in response to a COMPILE or RUN command, ERRLIST is SET and LIST is RESET. This suppresses the usual line-printer listing, and diverts error messages in a slightly different form to the terminal requesting the compilation. If ERRLIST and LIST are both set, errors are reported twice, once on the line-printer file LINE, and once on the remote file (ERRORFILE).

For compilations which are not initiated by CANDE, ERRLIST is reset, and nothing is written to the ERRORFILE.

ERRORLIMIT

ERRORLIMIT (Special numeric default)

ERRORLIMIT is not a boolean option; it cannot be set or reset. It is associated with a numeric value which is the maximum number of compiler errors permitted. If the compiler finds more errors than the specified limit, the compilation is aborted.

Since compilations initiated from remote terminals may swamp the user with error messages, this option is set at the low value of 6 for compilations initiated by CANDE COMPILE or RUN commands. For queue compilations, the limit is 150.

The user may select an ERRORLIMIT by following the option name by a numeric value as in the syntax below. A value of 0 implies no errorlimit: as many errors as occur will be reported.

errorlimit option

→ ERRORLIMIT → = → integer constant →

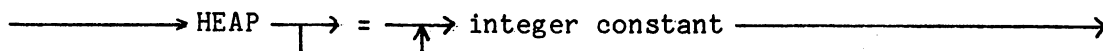
HEAP (Special numeric default)

The HEAP option is not a boolean option; it cannot be set or reset. It is associated with a numeric value which is the maximum number of words in the run-time heap available to the program. If this limit is exceeded during execution the program is terminated with an INVALID INDEX interrupt.

The default limit is set at 1000 (i.e. 1000 words are set aside in the heap for use by the program).

The user may set the HEAP size by following the option name by a numeric value as in the syntax below.

→ HEAP → = → integer constant →

Example

\$HEAP=10000

HEXCODE

HEXCODE (Default: reset)

The HEXCODE option is not intended for user-programmers. It is provided for systems programmers who may have to track down an elusive bug in the compiler-generated code. It turns on listings on the file LINE which display every generated code word in hexadecimal, and a number of other diagnostic print-outs. The data is intermixed with information produced by LIST and CODE if these are set.

It is rarely that this option is used, yet it is left in the production compiler to encourage accurate reporting of flaws, and to aid on-site maintenance. Its function may change without notice in later versions of the compiler.

INCLNEW (Default: reset)

This option only has effect if the NEW option is set. It affects the way in which included files are treated with respect to the NEWTAPE file. Refer to the option NEW for its usage.

INCLUDE

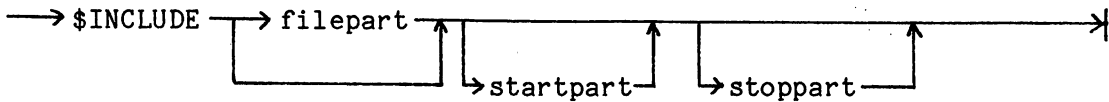
INCLUDE (No default status)

The INCLUDE option has a special syntax, and cannot be set or reset. It must appear on a special compiler option record by itself. The INCLUDE option specifies that a nominated file, or portion thereof, is to be textually included in the compilation at the point at which the INCLUDE record exists. This permits of several useful techniques:

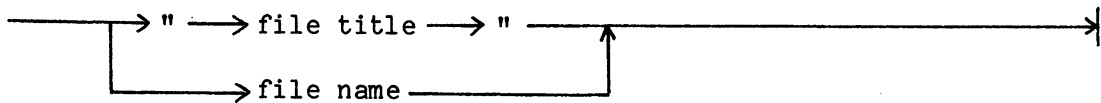
- * A large program may be structured by means of included files to increase clarity of the components.
- * A library of standard procedures or functions may be added to the program being compiled; for example graphics routines.
- * A particular set of declarations may be preserved across several programs by having but one copy, referenced by INCLUDEs in each program.
- * A program may include textual material produced by another program without the necessity of having to actually physically merge the two. For example, an error handling routine may INCLUDE a file of error-messages produced by a special formatter.

Syntax

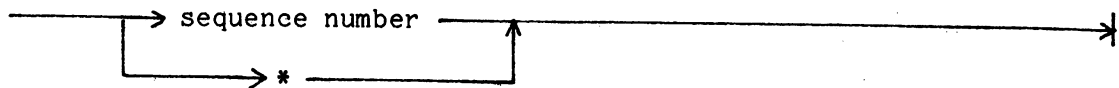
include option



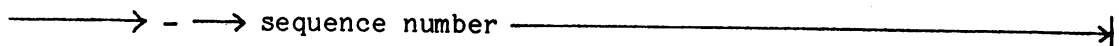
filepart



startpart



stoppart



Semantics

The filepart specifies the file to be included. The file-title form specifies the file explicitly by title; the file-name form specifies an 'internal name' that can be used in a WFL statement to assign an actual file to the name, as in:

```
COMPILER FILE INCLFILE(TITLE=COURSE/S102/MARKS)
```

If neither form is used, the file most recently referenced at this level of nesting of includes is used. Thus the first INCLUDE in any file (at least) must have a non-empty file-part.

If the start and stop parts are omitted, the whole of the nominated file is included.

If the start part is present, it indicates the first sequence number from the file to be included (or the next higher numbered record if that one does not exist). Its absence implies that the start is reckoned to be at the first record of the file. The * form implies that the inclusion is of a file previously referenced at this level of INCLUDE nesting, and the start is to be taken where the previous inclusion left off.

If the stop part is present, it indicates the last sequence number to be included in the file. If absent, this implies the end of the file.

INCLUDEd files may themselves contain INCLUDEs, up to a level of nesting currently set at 5. The included text is by default not listed when LIST is set. See LISTINCL and INCLNEW for a discussion of some of the interactions of INCLUDE with other options. The section on compiler files gives the file attributes built into the compiler for includes files. Suitable files may be created by the NEW option, or as CANDE SEQ files.

INCLUDE

Examples

```
$INCLUDE "GRAPHICS/ROUTINES"
```

```
$INCLUDE VARIANT 1-5050
```

```
$INCLUDE VARIANT *-99999999
```

LINEINFO (Default: set)

If this option is set the compiler writes additional information to the code file containing links between code-addresses and the sequence numbers of the source text. If an error occurs in execution, this permits the operating system to be able to pinpoint the error to a particular source line, to trace the program's call history in source line terms, and to notify the user of this on the job description sheet and possibly a remote terminal.

The option is set by default so as to permit the optimum debugging conditions. If it is reset, the size of the code-file may be halved, and this may be necessary for large code-files which are in production use and thoroughly debugged.

The use of the LINEINFO option has absolutely no effect on the execution speed or memory demands of a compiled Pascal program.

LIST

LIST (Special default)

If this option is set, a listing of the text as compiled is printed on the file LINE. Associated with the listing are three extra items:

- * A heading giving the compiler level number, the date and the time.
- * Associated with each listed line, the value of the compiler location counter for the generated code at the start of compiling that particular line.
- * A summary, giving the names of the source and code files, and various items of information concerning the generated code.

The location counter information is formatted as:

segment-number : word-address : syllable-address

and is principally of use to programmers who are very familiar with the B6700/B7700 computer systems. It may be of use in understanding the generated code, in tracking down elusive bugs, and in understanding dumps.

The summary information gives the total number of code-segments and read-only arrays generated by the compiler and their total size in words. It also lists the compiler's estimates of the required memory requirements of the program. These estimates are needed for the program's first execution attempt, but cannot be accurate as the compiler does not carry out an accurate flow analysis of the program. The values in the code-file will in fact be updated by the operating system based on its experiences with the program in execution.

If the compilation was initiated by CANDE in response to a COMPILE or RUN command, LIST is reset by default so as to suppress the generation of a line-printer listing. It can be explicitly set if desired. In all other cases, LIST is set by default.

LISTINCL (Default: reset)

This option only has effect if the LIST option is set. It affects the listing of included files.

If LISTINCL is reset, included files are not listed, even if LIST is set.

If both LIST and LISTINCL are set, then included files are printed on the file LINE together with primary input. The inclusion level is marked by a one-digit integer just to the left of the location-counter data on the listing.

See INCLUDE for a description of the use of included files, and LIST for a description of the listing format.

MERGE

MERGE (Default: reset)

If MERGE is reset, then the compiler takes its input only from the primary file CARD. This is the usual situation when compiling disk or pack files which have been edited by CANDE.

If MERGE is set, then the compiler takes its input as the merged result of the primary file CARD and the secondary file TAPE, which despite its name, is usually a disk or pack file. The files are merged according to a sequence number in columns 73-80 of the input records; if there is a record from each file with the same sequence number, that from TAPE is ignored.

When set, this option therefore allows patch cards to be merged with the main text held on the tape file. This is often convenient for compilations submitted from cards, or through queues.

NAMES (Default: reset)

If the NAMES option is set, then at the close of compilation of every procedure, function, or program that contains local objects or parameters, a 'name table' is printed giving brief details of each name used and declared in the program unit. The option is principally of use together with the CODE option as it gives address information, but it may also be of use by itself as a documentation aid.

The names are listed in alphabetical order, in canonical upper-cased form. Field-names of a record are printed indented by two spaces at the first occurrence of a type or variable that may use them, in their own alphabetical order. Since a variable may utilize a type declaration outside this particular procedure or function, some field names printed may not have been declared in the procedure/function immediately above.

The table gives details of whether the object is a type, variable, constant, or program unit. If it is a parameter, VAR parameters are marked as references to a variable (REF TO). Variables have the stack address listed; field-names have the word-displacement from the start of the record listed.

NEW

NEW (Default: reset)

This option is provided so that in conjunction with MERGE, an updated version of the source text can be written to disk or pack, or by itself, so that a card deck may be written to disk or pack.

If NEW is set, then the compiler input is written to a new file with the internal name NEWTAPE. Despite its name, this is usually a disk or pack file. There are some exceptions to the general rule that the compiler input is written directly to NEWTAPE, and these affect option records and included files.

Option records are only written to the NEWTAPE file if they contain 'space' and '\$' in columns 1 and 2 of the input record. If the '\$' appears in column 1 of the input record, it is regarded as a temporary option, and not written to NEWTAPE.

Included files are generally not written to the NEWTAPE file; instead the \$INCLUDE record is written, subject to the previous rule concerning the \$-position. However if the INCLNEW option is set (as well as NEW) the \$INCLUDE is not written to the NEWTAPE file, but the included text is.

The NEWTAPE file is locked at the end of compilation, regardless of whether the compilation was successful or not. It is good practice when using MERGE and NEW to update source text to maintain a father-son relationship, or a three-generation cycle. In other words, do not give the TAPE and NEWTAPE files the same title.

OMIT (Default: reset)

If this option is set, succeeding lines of source text are not compiled and are ignored by the compiler except for listing them with the word OMIT in place of the location counter field.

This option is primarily used in conjunction with user-options for selectively compiling pieces of program for a particular purpose. For example, suppose that the compiler is to handle a program which may run on a B6700 which does not have the VECTORMODE option installed, and in this case some of the source text of the program should not be compiled, and an alternative should be in its place. Then by setting a user-option (say VMODE) at the program head, the desired action can take place at many points in the program according to the following schemata:

```
$SET VMODE
    {program text}

$SET OMIT = NOT VMODE
    {code to be used if vectormode is installed}

$POP OMIT

$SET OMIT = VMODE
    {alternative to be used if vectormode is
      not installed}

$POP OMIT
    {more program text}

END.
```


PAGE

PAGE (No default status)

The PAGE option has effect only when the LIST option is set, and it then instructs the compiler to continue the listing at the top of the next printer page. It has no other effect on compilation, and is simply provided so that users can format long compilation listings as they wish, so that logical sections are not unduly broken by page boundaries. The PAGE option should appear on an option record by itself, and it has no value or default status.

Syntax

→ \$ PAGE →

SEQ (Default: reset)

If SEQ is set, the compiler input is resequenced (and therefore anything written to the NEWTAPE file). The first line is sequenced "00000100" and succeeding lines have sequence numbers going up in steps of 100. If SEQ is reset, the sequence number fields of the input lines are not altered.

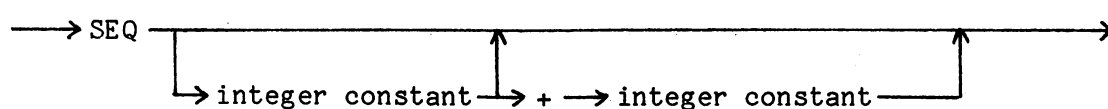
It is possible to set the starting sequence number and the increment to levels chosen by the programmer, and this is achieved by following the SEQ option name by two numeric parameters separated by a + sign. Leading zeros are not necessary in the numeric parameters. Thus to get the compiler input resequenced starting from "00050000" and incrementing by 100 for successive lines, the option record might be written:

Syntax

```
$SET SEQ 50000+100
```

The SEQ option only has effect if LIST or NEW is set.

seq option



SETSIZE

SETSIZE (Default value 48)

The SETSIZE option is not a boolean option; it cannot be set or reset. It specifies the size of the set where the set constructor is used. The bounds of the set are then assumed to be 0 and (SETSIZE - 1). The option is only applicable to sets with a component type of integer or a subrange of integer.

The default value is 48 and the maximum value is 65536.

The user may set the SETSIZE option by using the syntax specified below

→ SETSIZE → = → integer constant →

Example

```
$ SETSIZE = 200
```

STANDARD (Default: reset)

This option is provided so that Pascal programs (and particularly those written in a B6700/B7700 Pascal environment) may be checked for non-standard features which may impair their portability. It will be useful in

- (a) checking B6700/B7700 Pascal programs before export, and
- (b) checking supposedly portable Pascal programs imported into a B6700 or B7700 environment.

If the option is set, the compiler will issue additional WARNING messages for most non-standard features used in the source text. In particular, the use of OTHERWISE in a CASE statement, the use of formats or formatted record-oriented i/o, the use of non-standard pre-defined procedures, and the use of non-standard type changing are flagged. In addition, the compiler will analyse the identifier structure to determine whether there may be any semantic differences in interpretation arising in Pascal compilers which treat such identifiers as having insignificant text after the eighth character. These are also flagged as warnings, and since re-definition according to the scope rules may give rise to errors these are simply marked by a NOTE, though perfectly legal Pascal.

The use of simple lexical alternatives is not flagged as these are usually easy to modify on a target machine. Note that the compilation process is unaffected by the setting of STANDARD; it only enables additional checking and the emission of warnings to the LINE file.

The exact effects of this option may be subject to change from time to time as additional checking facilities are incorporated. It should be noted that the B6700/B7700 Pascal compiler is very strict on checking semantic interpretations which are often omitted in other Pascal systems, whether or not STANDARD is set. The option WARNINGS is set when this option is set.

STATISTICS

STATISTICS (Default: reset)

If set, this option causes the program in execution to collect timing and call-frequency information for each program procedure or function for which it is set. The value of the option is only relevant at the point at which the opening BEGIN of the body appears; the value of the STATISTICS option at that point is preserved for the whole of the procedure/function regardless of SET, RESET or POP actions. If STATISTICS is set over any procedure/function, then at program termination a program profile is printed on the job diagnostic file.

This profile consists of a single line entry for each procedure/function over which STATISTICS was set. The entries are in order of appearance of the procedure or function declaration in the source text, and are identified by (up to) the first six characters of the procedure/function name. For each entry, the line contains:

- (a) the number of entry calls,
- (b) the cpu-time spent in that unit, and
- (c) the average cpu-time in that unit per call (computed as (b)/(a)).

The cpu-time is given to the nearest microsecond and is derived from an MCP call with a 2.4 microsecond precision. Naturally, the collection of statistics itself consumes cpu-time; the increase may be significant for units with very little internal processing. The measured cpu-time does not include time spent in user-written procedure/functions called from the unit in question, but does include time spent in system intrinsics or Pascal standard procedures/functions. (This allows the summed cpu-time to approximately equal the overall cpu-time resource used).

Note: the Pascal STATISTICS option is not identical in effect to the Burroughs Algol or Burroughs FORTRAN options of a similar name.

Note: If a procedure/function is left by an error failure, the time printed for that unit includes that necessary for the profile printer to realize that its data is incomplete. Such occurrences are marked in the profile entry.

STRIPBLANKS (Default: set)

This option is only applicable to textfiles. If set, it causes all trailing blanks from input records to be removed before the record is passed to the user's buffer. Thus the user cannot access past the last non-blank character of an input record.

If reset, all characters on an input record are available to the user's program.

TRUSTWORTHY

TRUSTWORTHY (Default: reset)

If this option is set, the compiler takes it to be an assertion that the program to be compiled is trustworthy, and it omits to insert some checking code which might exact a significant time penalty in execution. Not all run-time checks are omitted; simply those which fall into the above category. In particular, stack variables are not set to 'uninitialized operand' at procedure entry, and instead are set to zero (or false, or the appropriate equivalent for the variable's type).

This option should not be set unless the programmer is (a) certain that the program is indeed trustworthy, and (b) the last ounce of speed is required. It is in any case worthwhile checking performance with and without TRUSTWORTHY set, and leaving it reset if it makes no significant difference.

The exact effects of this option are subject to change from time to time as more information on Pascal performance becomes available.

WARNINGS (Default: set)

This option is provided so that Pascal programmers may suppress warning messages or notes. It is useful when a 'clean' listing of a program is required. The option has no affect on error messages, or serious compiler fault messages. The option is also set as a side-effect of setting the option STANDARD.

USER-OPTIONS

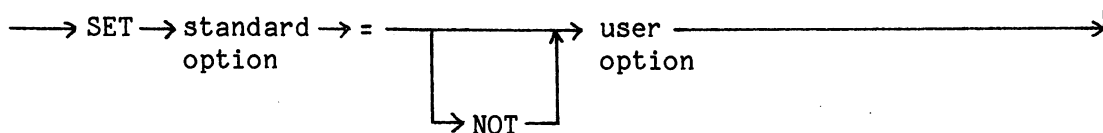
USER-OPTIONS

If any name appears in a compiler option record which is not recognized by the compiler as a standard option name, then the compiler enters that name into its table as a user-option, and creates a new stack for it. User-options are therefore of boolean type and can be SET or RESET. Since, however, they have no effect on the compilation directly, they would appear to be of little use. The purpose of user-options is instead to give a mechanism whereby the user may alter a multiplicity of related options throughout the program by the simple expedient of altering one setting. To achieve this, the syntax of SET is extended to allow the value of a user-option (the top-of-stack value) to be transferred into a standard option.

This facility is mainly used to selectively omit or insert pieces of code into the text as determined by the installed configuration or the installation's management practice. It can however be used to selectively list parts of the program, and for other purposes.

An example of the use of user-options is given in the sheet on the OMIT option.

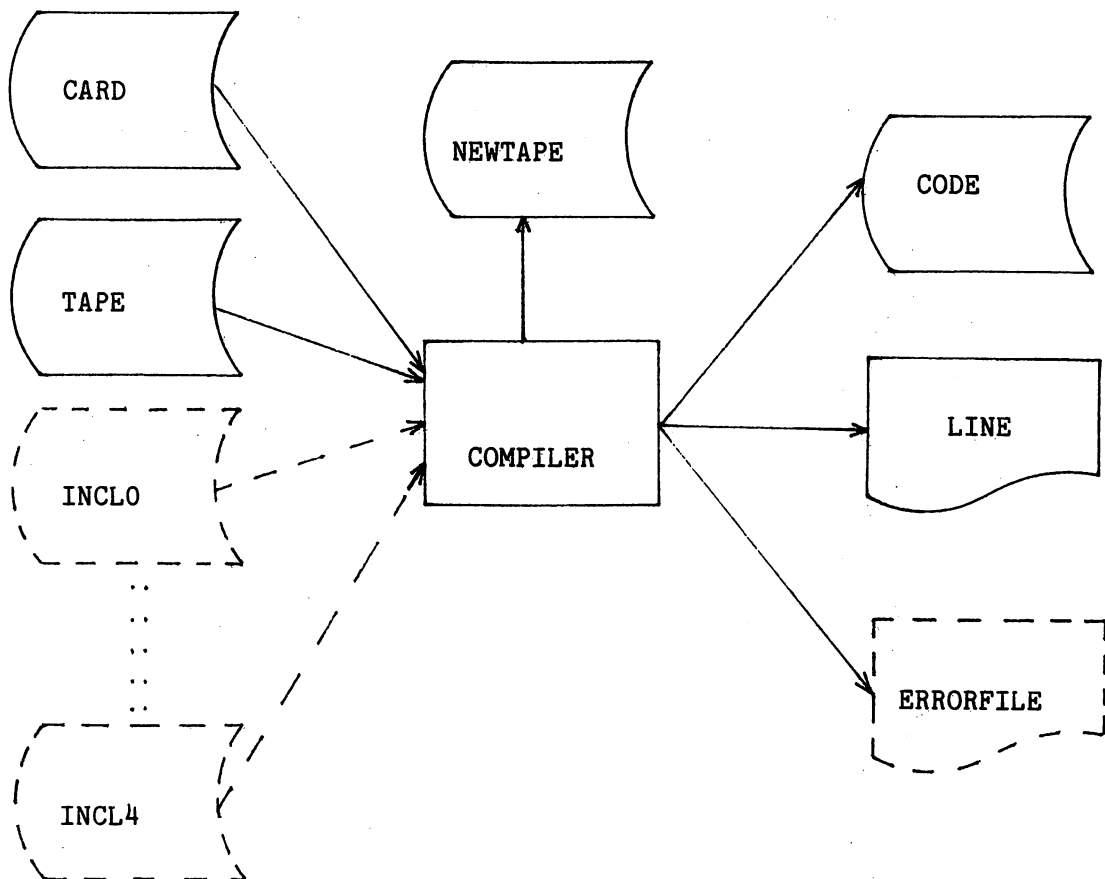
Extended syntax of set



10. COMPILER FILES

COMPILER FILES

The B6700/B7700 Pascal compiler may have many files open during compilation. These include three files which hold source text (CARD and TAPE for merged input, and NEWTAPE for output), up to 5 more source text files which hold included text, the code-file CODE (which holds the generated executable code), and two user-information files (LINE and ERRORFILE, used for line-printer and remote terminal output respectively). Seldom are all these in operation simultaneously. The following diagram illustrates the usage of the files by the compiler.



FILE DEFINITIONS

FILE DEFINITIONS

The following table gives the file attributes as declared in the B6700/B7700 Pascal compiler. These may be useful if it is necessary to alter any attribute by a WFL statement.

NOTE: The attributes of CODE should not be changed, except possibly for its TITLE or areas. The compiler forces AREASIZE and MAXRECSIZE to the values given.

FILE NAME	ATTRIBUTES
CARD	(KIND=READER, FILETYPE=8, BUFFERS=2, INTMODE=EBCDIC)
TAPE	(KIND=DISK, FILETYPE=8, BUFFERS=2, INTMODE=EBCDIC)
NEWTAPE	(KIND=DISK, BUFFERS=2, INTMODE=EBCDIC, AREAS=60, AREASIZE=1000, BLOCKSIZE=450, MAXRECSIZE=15, SAVEFACTOR=999)
an included file	(KIND=DISK, FILETYPE=8, BUFFERS=2, INTMODE=EBCDIC, BLOCKSIZE=150, MAXRECSIZE=14)
LINE	(KIND=PRINTER, BUFFERS=2, INTMODE=EBCDIC, MAXRECSIZE=22)
ERRORFILE	(KIND=REMOTE, BUFFERS=3, MAXRECSIZE=12)
CODE	(KIND=DISK, BUFFERS=2, AREAS=20, AREASIZE=300, BLOCKSIZE=150, MAXRECSIZE=30, SAVEFACTOR=999)

FILE EQUATION

The compiler files are declared so as to require the minimum of WFL file equation statements. The following examples are given to illustrate the necessary items; they are not exhaustive.

CASE 1

Compile a workfile from CANDE. No explicit file equation needed; CANDE will supply titles for CODE and CARD;

COMPILE WITH PASCAL

CASE 2

A queue job which is to compile a pack file MY/PROGRAM.

COMPILE OBJECT/MY/PROGRAM WITH PASCAL;
COMPILER FILE CARD(KIND=PACK,TITLE=MY/PROGRAM);

CASE 3

A queue job which is to merge MY/PROGRAM with a card reader global file MY/MERGE, and write the merged result as MY/PROGRAM/V2, compiling the merged result. There is one included file used with internal name SWITCH which is to be equated to MY/SWITCH/TESTING.

COMPILE OBJECT/MY/PROGRAM/V2 WITH PASCAL LIBRARY;
COMPILER FILE CARD(TITLE=MY/MERGE);
COMPILER FILE TAPE(TITLE=MY/PROGRAM);
COMPILER FILE NEWTAPE(TITLE=MY/PROGRAM/V2);
COMPILER FILE SWITCH(TITLE=MY/SWITCH/TESTING);

CASE 4

A programmer at a CANDE terminal which can print 132 character lines wishes to initiate a compilation with the LINE file diverted to her terminal.

COMPILE WITH PASCAL; COMPILER FILE LINE(KIND=REMOTE)

11. ERRORSERRORSExplanation

This section explains the reaction of the compiler to errors in the source text, and possible errors in Pascal programs in execution. The purpose is not to detail all possible errors, but to explain the approach taken to the detection and notification of errors to aid in diagnosing the flow and understanding subtle interactions that may occur.

Three types of error are treated:

- * compile-time errors,
- * system-handled run-time errors, and
- * Pascal-specific run-time errors.

This last category is itself split into 3: fatal errors, write-formatting errors, and read-formatting errors.

COMPILE-TIME ERRORS

COMPILE-TIME ERRORS

Explanation

The compiler may emit several kinds of messages to the user on the LINE file, or on the ERRORFILE file. These range in severity from a NOTE (not an error, but a usage worth commenting on), a WARNING (an error, but a plausible interpretation has been applied), and an ERROR (what it says, and the program will not be allowed to execute since the code is incorrect).

In most cases, where the error can be directly related to the source text being processed at that point, the first line printed on the LINE file is one containing a star or row of stars under the lexical symbol last accepted by the compiler. This may be the symbol causing the error, or the cause may lie earlier in the text. (It cannot be past this point, for it has not yet been scanned.) If the error is not related to the current symbol as for example a missing forward declaration at the close of the declaration part, the first line makes this point.

The next line contains an error message, in English text. It should be self-explanatory. Programmers should be aware however that the compiler detects an incompatibility, and cannot know what they intended. Some error messages contain the compiler-writer's guess as to the most likely mistake; others cannot go beyond a cryptic "UNEXPECTED SYMBOL IN THIS CONTEXT". In all cases be wary of what the text says; it may be that the syntactic analysis has taken a different twist from the one you intended or the compiler-writer envisaged. It may occur, as a result of an oversight, that the compiler does not contain a message for a particular error situation. Such cases should be immediately reported by a Field Trouble Report; the error number always appears at the extreme-right of the error message line.

In some cases the compiler has to recover from an error by scanning for a syntactically valid continuation. If this scan takes it past the next symbol the compiler will print another line indicating where this scan went to before syntactic analysis recommenced. All text between the point indicated and the immediately previous error has been ignored. This indication is of use in diagnosing later parasitic errors as having been caused by such a skip.

If the compiler cannot or should not continue the compilation, it issues a special type of error message and terminates the compilation. This typically occurs if the program evokes more errors than the ERRORLIMIT for the compilation, or if the compiler runs out of space for one of its tables. This last contingency is only

possible for very large programs.

If the compiler was compiled with the DEBUG option set, it also contains internal consistency checks. If one of these consistency checks fails during compilation, a special "COMPILER FLAW" message is printed. The occurrence of one of these messages and its context should be reported by a Field Trouble Report. The flaw is in all probability not due to any error in the source text of the program being compiled.

Any definite error in compilation will cause the code-file to be purged so that it cannot be executed. Only if no errors are detected is the code-file made available for execution.

The compiler option, WARNINGS, may be used to suppress the printing of warning messages and notes.

INV OPERATOR

INV OPERATOR

Termination of a Pascal program in the debugging and testing phases is frequently caused by the INV OPERATOR interrupt. Though this is a standard Burroughs response, the interpretation of the message is different from that of most other compilers on B6700/B7700 systems.

The INV OPERATOR interrupt is caused by either an illegal instruction or by an instruction that finds an illegal operand. It is this latter interpretation (INVALID OPERAND) that applies to Pascal, and it is caused by the extra run-time checking incorporated into Pascal code. Variables which occupy single words (sets, reals, scalar types) and which are declared as local variables (thereby residing in the run-time stack) are created with a special bit-pattern that is taken by the B6700/B7700 computers to be 'uninitialized operand'. Any attempt to utilize the value of such a variable without first assigning it some legal value therefore evokes the INV OPERATOR interrupt.

The 'uninitialized operand' value is also used to undefine the value of the control variable of a for-loop after the normal exit so that adherence to the Pascal definition can be checked at run-time.

INV OPERATOR is therefore a sign of user programmer error, and will only mean a bug in the compiler in exceptionally rare cases. A similar effect exists with uninitialized character-pointers in Burroughs Algol.

Technical Note

The 'uninitialized operand' bit pattern consists of an all-zero word with a tag of 110 (a tag-six word). It can be overwritten by any store instruction since it is not storage-protected. It is distinguished from a 'Software Control Word' (SCW) by the absence of a set bit in position 47 of the word.

PASCAL READ ERRORSExplanation

If a program is using the Pascal stream-oriented i/o system (not the record-oriented system with formats), then if an incompatibility arises between the type of the object for which a read is requested and the actual characters (tokens) found on the input file, the program is terminated in exactly the same way as for other Pascal run-time errors.

There are two exceptions to this rule, an attempt to read past the end of the input file. This is not fatal since the program may wish to test for end-of-file using the EOF function. However if a second attempt is made to read the file past this point, it is treated as a fatal error.

The second exception is intended for one specific purpose: communication with a user on an interactive terminal. In such a case a mistake by the user should not be immediately fatal to the program so that it can re-prompt him/her to respond again correctly. To provide this facility, the READ procedure in its stream-oriented form may be used as a function of INTEGER. If it is so used, the fatal error is suppressed, and the error number is returned as the function value, zero being used to indicate that no error was detected.

The error numbers and their meanings are:

ERROR NUMBER	MESSAGE
0	No error
1	READFILE=WRITEBUSY
2	END OF READ-FILE
3	NO DIGIT AFTER +-
4	TOO MANY DIGITS
5	NO DIGIT AFTER .
6	EXPONENT TOO LONG
7	NO EXPNT DIGIT
8	NOT SCALAR NAME
9	RADIX <> 2/4/8/16
10	DIGIT >= RADIX
11	NUM TOO BIG/SMALL
12	REAL/INT CLASH
13	SCALAR/NUM CLASH
14	NOT IMPLEMENTED
15	RANGE VALUE ERROR

PASCAL WRITE ERRORS

PASCAL WRITE ERRORS

Explanation

If a program is using the Pascal stream-oriented i/o system on a file with a structured component then if the WRITE procedure is unable to produce the object's external representation in the desired layout it intercalates one or more lines on the output file.

If the error relates to a specified field width, that field is filled with stars before the intercalation begins. The first intercalated line gives the reason for the error in a brief message, followed by one or two numeric values to illustrate the error (for example the necessary size for the object, and the allocated width). If the value has not been printed, it may appear on the next line in a default format. Finally, as the WRITE procedure returns to process the next item, it fills the line with the character ">" until it reaches the point where the last genuine output line had to be terminated.

No write layout errors are fatal: they all appear on the output file. The only two fatal errors that can be caused by this WRITE procedure are an attempt to use a read-file for write without resetting or rewriting, and an attempt to write past the end of the file (fatal on the second attempt).

If a program is using a textfile, the only fatal WRITE error occurs when an attempt is made to write to a read-only file. All flaws in the layout specification are handled according to the specification in the standard.

RUN-TIME PASCAL ERRORS

Explanation

Some checks are compiled into Pascal programs to detect violations of the semantic rules of Pascal during execution. Such checks include the computation of a scalar value outside the declared range, the use of a case expression with no matching case-label, etc. The execution of the program is terminated, a brief message is displayed on the job-description sheet, and the program call-history is printed.

RUN-TIME SYSTEM ERRORS

RUN-TIME SYSTEM ERRORS

Explanation

The B6700/B7700 operating system will respond to run-time errors caused by executing Pascal programs in the same way as for Algol programs. These errors arise either from an interrupt (trap) arising from a violation of the conditions required for the successful completion of an instruction, or as programmatically detected by components of the B6700/B7700 system activated from Pascal.

The first category includes such events as:

INTEGER OVERFLOW

Detected at an attempt to assign, or use as a subscript, etc., an arithmetic expression which is too large for the representation.

REAL OVERFLOW

Detected at an arithmetic operator that causes the result to exceed the representation range.

INVALID INDEX

Caused by an attempt to access an array, record, or heap object outside its declared bounds. The most likely causes are out-of-range subscripts in arrays, or invalid pointers into the heap.

INV OPERATOR

The system's response to many unusual situations. May be caused by a set constructor with an out-of-range variable component, or by an attempt to use the value of an uninitialized operand.

The second category arise primarily from use of the (FORTRAN-like) record-oriented formats, and from use of the pre-defined arithmetic intrinsic functions. The messages are the standard ones produced for such errors, and these are documented in the appropriate manuals.

It is also possible to terminate a program from a terminal by the ?DS command, or by BREAK when receiving output. These occurrences are treated similarly to errors, and cause the system to terminate the

RUN-TIME SYSTEM ERRORS

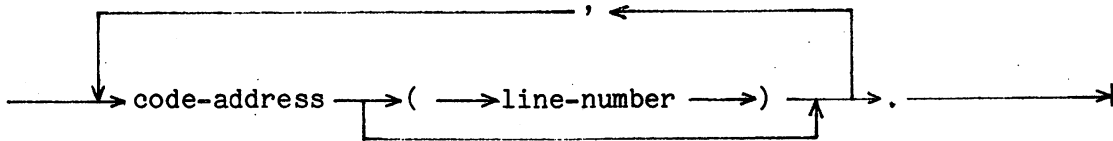
program with the messages "OPERATOR DS" or "BREAK ON OUTPUT".

STACK HISTORY

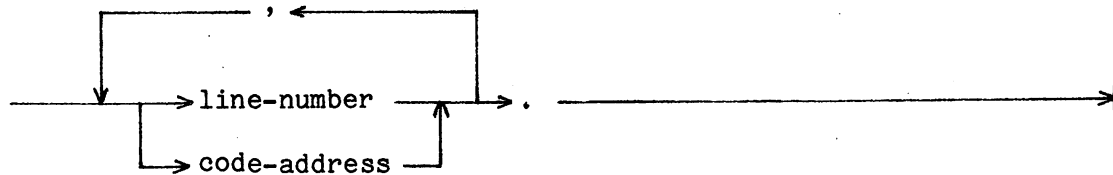
STACK-HISTORY

Syntax

job-description stack-history



terminal stack-history



Explanation

The stack-history would be better named the call-history of the program as it gives the history of currently uncompleted procedure or function calls in the program. It is produced as a by-product of most run-time errors that are fatal, and is announced to an interactive terminal user and printed on the job-description sheet.

The line-number information is only produced if the LINEINFO compiler option was set when the segment (program/procedure/function/intrinsic) was being compiled; otherwise only the code-address appears. The code-address is in the same format as that produced by the compiler on the LINE file by the LIST option, and can be related to that listing if the segment was one produced by the Pascal compiler. The line-number is an 8-digit number: the sequence number of the line which is thought to contain the cause of the termination, or at which termination intervened.

The first address or number is that of the point of termination; the next is of the place from which this procedure/function/intrinsic was called (and this call has not yet returned), followed by successive

addresses of call-points back to the outermost Pascal program. Some errors are associated with calls to operating system or installation intrinsics, and such addresses/numbers are also included in the call-history. If the call-history is very deep (as may occur in highly recursive programs), the operating system may not continue the trace all the way back to the main program, and it will be terminated at some other point.

Knowledge of the stack history may enable difficult errors to be isolated more easily as the program's state can be partially reconstructed. This is particularly true of infinite loop and infinite recursive situations.

12. SAMPLE PROGRAMS

The following small sample programs are given to illustrate the listings produced by the compiler, and the form of programs written in the Pascal language.

They are:

- *Art Squares - a Pascal program that illustrates many of the standard features available, most notably the use of lower case. The program produces a page of reflections and rotations of an initially specified square.

- *Warshall - An implementation of Warshall's algorithm, written in Pascal to use the set facility (rather than arrays of booleans). It illustrates some of the data-structuring facilities available. It also shows the non-standard i/o features introduced for compatibility with FORTRAN and Algol, and the use of the compiler options STANDARD (to flag non-standard features) and STATISTICS (to collect execution-time statistics on the program).

- *A short annotated program to illustrate the code listing produced by the compiler option CODE for some Pascal statements and constructs. It also shows the use of the NAMES option.

- *A short program with syntax errors, showing the type of message produced by the compiler.

B0700 PASCAL COMPILER VERSION 2.9.001 WEDNESDAY, 1979 AUGUST 29, 06:17 PM
 =====

CARD FILE :- (USIS0013)PMANUAL/SECTION12/EXAMPLE1 ON PACK.

```

program artsquares(input,output);
  (*****)
  ( Note: This program shows the use of lower case and
  standard Pascal symbols and constructs. )
const
  side = 5; { side of square pattern }
  charperline = 150; { multiple of side }
  linesperpage = 60;
type
  sidesize = 1 .. side;
  squarekind = array[sidesize] of
    packed array[sidesize] of char;
  linesize = 1 .. charperline;
  pagesize = 1 .. linesperpage;
  rotation = 0 .. 3; { in 90 degree steps }
var
  sq : squarekind;
  strip : array[sidesize] of
    packed array[linesize] of char;
  w : sidesize;
  x : linesize;
  y : pagesize;
  refl : boolean; { reflected about y-axis }
  seed : real; { starts off randomizer }
  probability : real; { used to randomize }

procedure plotsquare
  (square : squarekind; { to be plotted }
  xmove : linesize; { this far along }
  rotated : rotation; { multiple of 90 degrees }
  reflect : boolean; { about y if true })
var
  exch,revx,revy : boolean; { how to scan square }
  xa,ya,yp,yp : linesize; { indexes }
begin
  { are x,y reversed or exchanged? }
  revx := reflect;
  case rotated of
    0: begin exch:=false; revy:=false end;
    1: begin exch:=true; revy:=false; revx:=not revx end;
    2: begin exch:=false; revy:=true; revx:=not revx end;
    3: begin exch:=true; revy:=true end;
  end;
  { plot the square onto the strip }
  for ya := 1 to side do begin
    if revy then yp:=side-ya+1 else yp:=ya;
    for xa := 1 to side do begin
      if revx then xp:=side-xa+1 else xp:=xa;
      if exch then begin
        strip[xp,xmove+yp-1] := square[ya,xa];
      end else begin
        strip[yp,xmove+xp-1] := square[ya,xa];
      end;
    end;
  end;
end;
  
```

```

00059000
00060000
00061000
00062000
00063000
00064000
00065000
00066000
00067000
00068000
00069000
00070000
00071000
00072000
00073000
00074000
00075000
00076000
00077000
00078000
00079000
00080000
00081000
00082000
00083000
00084000
00085000
00086000
00087000
00088000
00089000
00090000
00091000
004:003F:0
004:0042:4
004:0048:2
004:0048:2
003:0000:1
003:0000:1
003:0002:3
003:0002:3
003:0006:3
003:000A:5
003:000F:1
003:0013:3
003:0017:5
003:0017:5
003:001A:5
003:001A:5
003:0020:3
003:0027:3
003:002D:1
003:002D:1
003:0030:1
003:0030:1
003:0033:1
003:0033:1
003:0035:2
003:0039:5
003:003C:5
003:003C:5
003:003F:1
003:0048:2
003:004C:0
003:004F:0
003:004F:3

```

```

end_ { of plot-a-square }
begin { of art-squares }
  { set up randomizer, with the time for starters }
  seed := 12.03;
  { set up a square pattern for example }
  sq[1] := 'X';
  sq[2] := 'X';
  sq[3] := 'X X';
  sq[4] := 'X X';
  sq[5] := 'X X';
  { repeatedly make a picture until end of file input is reached }
  while not eof(input) do begin
    { read in data to affect picture }
    read(input,probability);
    if (probability < 0.0) or (probability > 1.0) then halt;
    page(output); { to top of screen or page }
    { loop to construct and write strips }
    for y := 1 to ((linesperpage div side) do begin
      { construct a strip }
      for x := 1 to (charsperline div side) do begin
        { plot a square }
        refl := (random(seed) < probability);
        plotasquare(sq,((x-1)*side+1),(x mod 4),refl);
      end;
      { write a strip }
      for w := 1 to side do begin
        writeLn(output,strip[w]);
      end;
    end;
  end; { of y-loop to print a page }
end; { of art-squares }

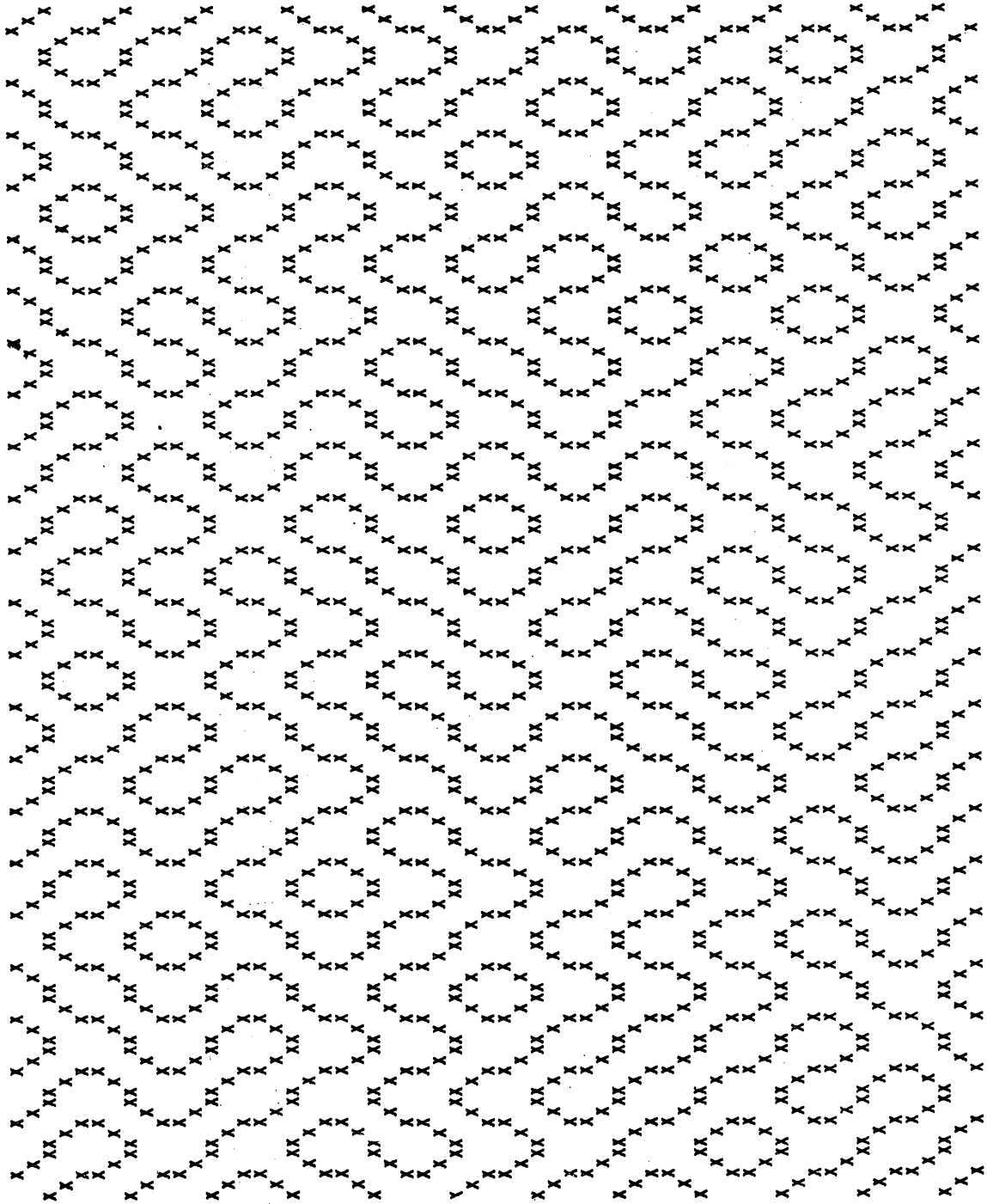
```

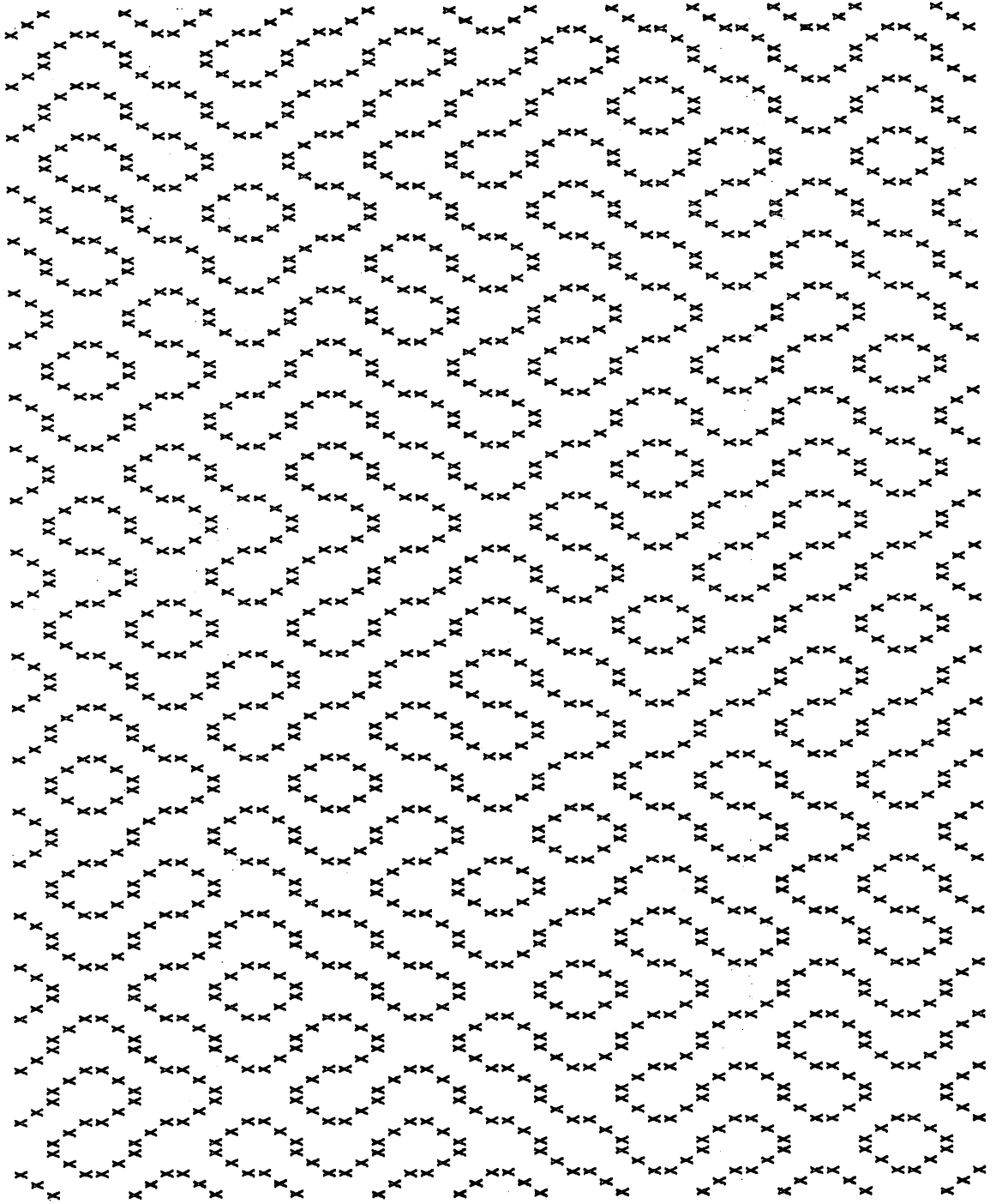
```

=====COMPILATION SUMMARY=====
PROGRAM SOURCE FILE TITLE: (USIS0013)PMANUAL/SECTION12/EXAMPLE1 ON PACK.
PROGRAM CODE FILE TITLE: OBJECT/EXAMPLE/1.
NO ERRORS FOUND BY COMPILER

```

CODE SEGMENTS	VALUE ARRAYS	EST. STACK	EST. MEMORY	DISK SECTORS
2	1	38	1184	18
185	5			





R6700 PASCAL COMPILER VERSION 2.9.001 TUESDAY, 1979 AUGUST 28, 12:38 PM.

CARD FILE :- (US150013)PMANUAL/SECTION12/EXAMPLE2 ON PACK.

```

$SET $ LIST LINEINFO STANDARD STATISTICS
PROGRAM WARSHALL:
(* THIS PROGRAM USES THE NON-STANDARD FORMATTED I/O FEATURES.
THE "STANDARD" COMPILER OPTION HAS BEEN SET TO POINT OUT
THE NON-STANDARD FEATURES USED.
THE STATISTICS OPTION IS ALSO SET TO INDICATE THE KIND OF
INFORMATION PRODUCED *)
CONST
  SIZE=69; (* ARRAY IS DIMENSIONED (SIZE+1) SQUARE *)
  WORDS=1; (* SIZE DIV 48 *)
  BITSPERWORD=48; (* BITS IN 66700 WORD *)
  BITSMINUS1=47; (* BITSPERWORD - 1 *)
TYPE
  BTYPE = ARRAY[0..SIZE] OF ARRAY[0..WORDS] OF SET OF 0..BITSMINUS1;
VAR
  N : INTEGER;
  T1,T2 : REAL;
  OUTPUT : FILE (KIND=PRINTER,MAXRECSIZE=22);
  A,B : BTYPE;
FORMAT
  *****
  RECORD ORIENTED I/O IS NOT STANDARD PASCAL
  F132 (132A1);
  FTIME ("TIME (SECS) =",F20.6);
  FVALUES ("SIZE, INPUT, CLOSURE BITS: ",315);
  FINPUT ("INPUT MATRIX");
  FTRANS ("TRANSITIVE CLOSURE");
FUNCTION GENERATE: INTEGER;
BEGIN
  N:=57*N+1;
  GENERATE:=(N MOD (SIZE+1));
  N:=N MOD 571;
END;
PROCEDURE FILL( VAR A : BTYPE;
THIS IDENTIFIER HAS BEEN DEFINED AT ANOTHER LEVEL
SIZE, P : INTEGER);
  *****
THIS IDENTIFIER HAS BEEN DEFINED AT ANOTHER LEVEL
VAR I,J,K,L : INTEGER;
BEGIN
FOR I:=1 TO WORDS DO A[0][I]:=[];
FOR J:=1 TO SIZE DO
  ALLJ:=A[0];

```

```

00006100 00A:0011:3
00006200 00A:0014:2
00006300 00A:0021:5
00006400 00A:0028:0
00006500 00A:002B:0
00006600 00A:0033:3
00006700 00A:0033:3

N6213
N6213
00006800 00C:0000:1
00006900 00C:0000:1
00007000 00C:0000:1
00007100 00C:0000:1
00007200 00C:0000:1
00007300 00C:0003:0

W1906
00007400 00C:0008:3
00007500 00C:0008:3
00007600 00C:000D:0
00007700 00C:000D:0
00007800 00C:000E:5
00007900 00C:000E:5
00008000 00C:0015:0
00008100 00C:0019:0
00008200 00C:0022:4
00008300 00C:0022:4

N6213
N6213
N6213
00008400 00C:0000:1
00008500 00E:0000:1
00008600 00E:0000:1
00008700 00E:0000:1
00008800 00E:0000:1
00008900 00E:0003:0
00009000 00E:0005:5
00009100 00E:0008:4
00009200 00E:000D:1
00009300 00E:0010:1
00009400 00E:0020:5
00009500 00E:0028:2
00009600 00E:0028:2

N6213
00009700 00C:0000:1
00009800 00F:0000:1
00009900 00F:0006:1
00010000 00F:0000:1
00010100 00F:0001:0
00010200 00F:0004:1
00010300 00F:0007:1

W1978
00010400 00F:0012:1
00010500 00F:0013:1
00010600 00F:001B:5
00010700 00F:001B:5

```

```

FOR I:=1 TO P DO BEGIN
  J:=GENERATE; L:=GENERATE; K:=L DIV BITSPERWORD;
  ALJ[K]:=ALJ[K] + E[L MOD BITSPERWORD];
END;
END; % OF FILL

PROCEDURE PRINT( VAR B : BTYPE; SIZE : INTEGER);
  THIS IDENTIFIER HAS BEEN DEFINED AT ANOTHER LEVEL
  ***
  THIS IDENTIFIER HAS BEEN DEFINED AT ANOTHER LEVEL
  ***
  VAR I,J : INTEGER;
  BEGIN
    FOR I:=0 TO SIZE DO BEGIN
      WRITE (OUTPUT,F132, FOR J:=0 TO SIZE DO
        ***
        RECORD ORIENTED I/O IS NOT STANDARD PASCAL
        BEGIN
          IF (J MOD BITSPERWORD) IN B[I][J DIV BITSPERWORD] THEN
            ELSE
          END);
        END;
      END; % OF PRINT
    END;
  PROCEDURE WARSHALLSALGORITHM( VAR A,B : BTYPE; SIZE : INTEGER);
    THIS IDENTIFIER HAS BEEN DEFINED AT ANOTHER LEVEL
    *
    THIS IDENTIFIER HAS BEEN DEFINED AT ANOTHER LEVEL
    *
    THIS IDENTIFIER HAS BEEN DEFINED AT ANOTHER LEVEL
    ***
    VAR I,J,K : INTEGER;
    BEGIN
      B:=A;
      FOR I:=0 TO SIZE DO
        FOR J:=0 TO SIZE DO
          IF (I MOD BITSPERWORD) IN B[J][I DIV BITSPERWORD] THEN
            FOR K:=0 TO WORDS DO
              B[J][K]:=B[J][K] + B[I][K];
            END; % OF WARSHALL
          END;
        FUNCTION BITS( VAR A : BTYPE) : INTEGER;
          THIS IDENTIFIER HAS BEEN DEFINED AT ANOTHER LEVEL
          *
          VAR SUM,I,J : INTEGER;
          BEGIN
            SUM:=0;
            FOR I:=0 TO SIZE DO
              FOR J:=0 TO WORDS DO
                SUM:=SUM+CARD(A[I][J]);
            END;
            THIS PROCEDURE/FUNCTION IS NOT STANDARD PASCAL
            BITS:=SUM;
            END; % OF BITS
          END;
        BEGIN

```

SAMPLE PROGRAMS

```

00010800 003:0000:1
          W1878
00010900 003:0007:4
00011000 003:0008:3
00011100 003:000F:3
00011200 003:0016:4

          W1878
00011300 003:0019:4

          W1906
00011400 003:0022:0

          W1906
00011500 003:0024:1
00011600 003:0039:0

          W1906
00011700 003:003E:0
00011800 003:0044:3

          W1906
00011900 003:0049:0
00012000 003:004F:3
    
```

```

T1:=PROCESSTIME;
*
* THIS PROCEDURE/FUNCTION IS NOT STANDARD PASCAL
N:=1;
FILL(A, SIZE, 125);
WARSHALLSALGORITHM(A, B, SIZE);
T2:=PROCESSTIME;
*
* THIS PROCEDURE/FUNCTION IS NOT STANDARD PASCAL
WRITE(OUTPUT, FTIME, (T2-T1));
****
* RECORD ORIENTED I/O IS NOT STANDARD PASCAL
WRITE(OUTPUT, FVALUES);
*****
* RECORD ORIENTED I/O IS NOT STANDARD PASCAL
BEGIN (SIZE+1); BITS(A); BITS(B); END);
WRITE(OUTPUT, FINPUT);
****
* RECORD ORIENTED I/O IS NOT STANDARD PASCAL
PRINT(A, SIZE);
WRITE(OUTPUT, FTRANS);
****
* RECORD ORIENTED I/O IS NOT STANDARD PASCAL
PRINT(B, SIZE);
END.
    
```

*****COMPILATION SUMMARY*****
PROGRAM SOURCE FILE TITLE: (USIS0013)PMANUAL/SECTION12/EXAMPLE2 ON PACK.
PROGRAM CODE FILE TITLE: OBJECT/EXAMPLE/2.
NO ERRORS FOUND BY COMPILER

NO OF:	CODE SEGMENTS	VALUE ARRAYS	EST. STACK	EST. MEMORY	DISK SECTORS
SIZE:	6	5	47	1361	32
	328	35			

B0700 PASCAL COMPILER VERSION 2.9.001 TUESDAY, 1979 AUGUST 20, 12:38 PM

CARD FILE :- (USIS0013)PMANUAL/SECTION12/EXAMPLE3 ON PACK.

SSET \$ LIST NAMES LINEINFO
 (* THIS PROGRAM ILLUSTRATES THE INFORMATION PRODUCED BY THE
 "CODE" AND "NAMES" OPTIONS, AND THEREFORE THE INSTRUCTIONS
 PRODUCED BY THE COMPILER CORRESPONDING TO PASCAL
 STATEMENTS AND CONSTRUCTS. *)

PROGRAM EXAMPLE_CODE_LISTING;

VAR

J : INTEGER;
 B : BOOLEAN;

BEGIN
 B:=TRUE;
 SSET CODE

FOR J:=1 TO 100 DO BEGIN

003:0000:5

003:0002:0

003:0002:2

003:0002:4

003:0003:0

003:0003:1

003:0003:1

003:0003:3

003:0003:4

003:0003:5

003:0004:0

003:0004:2

003:0004:4

003:0004:5

003:0005:0

003:0005:1

003:0005:2

003:0005:4

END;

(01,00004) = SEPARATE SEQ.DICT. INTRINSIC

003:0007:2

003:0007:4

003:0008:0

003:0008:1

003:0008:2

003:0008:4

003:0009:0

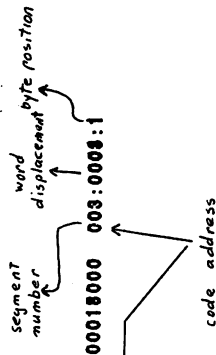
003:0009:2

SPOP CODE

END.

00001000 003:0000:1
 00002000 003:0000:1
 00003000 003:0000:1
 00004000 003:0000:1
 00005000 003:0000:1
 00006000 003:0000:1
 00007000 003:0000:1
 00008000 003:0000:1
 00009000 003:0000:1
 00010000 003:0000:1

00011000 003:0000:1
 00012000 003:0000:1
 00013000 003:0000:1
 00014000 003:0000:1
 00015000 003:0000:1
 00016000 003:0000:1
 00017000 003:0000:1



00019000 003:0000:1

00020000 003:0000:1
 00021000 003:0000:1

input and output file descriptor blocks
 SEGMENT 000 CONTAINS INFO
 SEGMENT 000 LENGTH = 0005 WORDS
 SEGMENT 000 CONTAINS INFO
 SEGMENT 000 LENGTH = 0005 WORDS

code mnemonics

LT48 (#001006400001)
 LT8 (004) #04

STAG

NAMC (02,00011)

STOD

EL0003}

VALC (02,00012)

LN0T

ONE

LAND

VALC (02,00011)

LT8 (003) #03

IDIV

ZERO

EQU

LOR

NAMC (02,00012)

STOD

NAMC (02,00011)

STBR (L0004)

BRTR (L0003)

MKST

NAMC (01,00004)

LT8 (004) #04

ENR

EL0004}

ZERO (006) #06

LT8

STAG

NAMC (02,00011)

STOD

Name Table

NAME	STACK	LOCATION	NAME-TYPE	NAME
	(02,00012)		VAR	B
	(02,00011)		VAR	J

size of this
code segment

SEGMENT 003 LENGTH = 0032 WORDS

PROGRAM SOURCE FILE TITLE: (US1S0013)PHANUAL/SECTION12/EXAMPLE3 ON PACK.
PROGRAM CODE FILE TITLE: OBJECT/EXAMPLE/3.
NO ERRORS FOUND BY COMPILER

NO OF:	CODE SEGMENTS	VALUE ARRAYS	EST. STACK	EST. MEMORY	DISK SECTORS
1	0	0	13	1040	9
32	0	0			

CARD FILE :- (USIS0013)PMANUAL/SECTION12/EXAMPLE4 ON PACK.

```

>>>>>0001>>>>>PROGRAM negative powers;
      *****
>>>>>MISSING SEMICOLON AFTER PROGRAM HEADING (INSERTED)
      (* When free of syntax errors, this program will produce the
         decimal representation of the negative powers of 2 up to
         a predefined value 'n'. *)
      const
        n = 30;
      type
        digit = 0..9;
      var
        i,k,r : integer;
        decimal : array[1..n] of digit;
      begin
        for k:=1 to n do
          begin
            write(' ');
            r:=0;
            k:=k-1;
          end
        end;
        *****
>>>>>0002>>>>>YOU CANNOT ALTER A FOR STATEMENT CONTROL VARIABLE
        for i:=1 to k do
          begin
            r:=10*r+decimal[i];
            decimal[i]:=r div 2;
          end;
          *****
>>>>>0003>>>>>NAME NOT YET DECLARED (MIS-SPELT? OMITTED?)
        r:=r-2*decimal[i];
        write(chr(decimal[i])+ord('0'));
        *****
>>>>>0004>>>>>THIS SYMBOL CANNOT START A VARIABLE (OR SELECTION THEREOF)
      TEXT IGNORED TO-----
>>>>>0005>>>>>ILLEGAL + OR - OPERATOR (INCOMPATIBLE TYPES)
      end;
      k:=k+1;
      *****
>>>>>0006>>>>>YOU CANNOT ALTER A FOR STATEMENT CONTROL VARIABLE
        decimal[k]:=5;
        write(ln('5'));
      end.
      *****
PROGRAM SOURCE FILE TITLE: (USIS0013)PMANUAL/SECTION12/EXAMPLE4 ON PACK.
PROGRAM CODE FILE TITLE: OBJECT/EXAMPLE/4.
PROGRAM CAUSED 6 TRANSLATION ERRORS DUE TO SYNTACTIC/SEMANTIC FLAWS

```

NO OF:	CODE SEGMENTS	VALUE ARRAYS	EST. STACK	EST. MEMORY	DISK SECTORS
SIZE:	71	1	0	0	8

71 1 0 0 8

13. GENERALExplanation

This section contains a number of appendices which could not be classified into other sections. The information is not important for the use of the compiler, but may be interesting or illuminate the reasons for a particular decision.

CHARACTER SETS

CHARACTER SETS

EBCDIC CHARACTER SET

ORD	CHR	ORD	CHR	ORD	CHR	ORD	CHR
0	NUL	32		64	space	96	-
1	SOH	33		65		97	/
2	STX	34		66		98	
3	ETX	35		67		99	
4		36		68		100	
5	HT	37	LF	69		101	
6		38	ETB	70		102	
7	DEL	39	ESC	71		103	
8		40		72		104	
9		41		73		105	
10		42		74	[106	!
11	VT	43		75	.	107	,
12	FF	44		76	<	108	%
13	CR	45	ENQ	77	(109	_
14	SO	46	ACK	78	+	110	>
15	SI	47	BEL	79	!	111	?
16	DLE	48		80	&	112	
17	DC1	49		81		113	
18	DC2	50	SYN	82		114	
19	DC3	51		83		115	
20		52		84		116	
21		53		85		117	
22	BS	54		86		118	
23		55	EOT	87		119	
24	CAN	56		88		120	
25	EM	57		89		121	`
26		58		90]	122	:
27		59		91	\$	123	#
28	FS	60	DC4	92	*	124	@
29	GS	61	NAK	93)	125	'
30	RS	62		94	;	126	=
31	US	63	SUB	95	^	127	"

EBCDIC CHARACTER SET (continued)

ORD CHR	ORD CHR	ORD CHR	ORD CHR
128	160	192 {	224 \
129 a	161 ~	193 A	225
130 b	162 s	194 B	226 S
131 c	163 t	195 C	227 T
132 d	164 u	196 D	228 U
133 e	165 v	197 E	229 V
134 f	166 w	198 F	230 W
135 g	167 x	199 G	231 X
136 h	168 y	200 H	232 Y
137 i	169 z	201 I	233 Z
138	170	202	234
139	171	203	235
140	172	204	236
141	173	205	237
142	174	206	238
143	175	207	239
144	176	208 }	240 0
145 j	177	209 J	241 1
146 k	178	210 K	242 2
147 l	179	211 L	243 3
148 m	180	212 M	244 4
149 n	181	213 N	245 5
150 o	182	214 O	246 6
151 p	183	215 P	247 7
152 q	184	216 Q	248 8
153 r	185	217 R	249 9
154	186	218	250
155	187	219	251
156	188	220	252
157	189	221	253
158	190	222	254
159	191	223	255

NOTE: Some line-printers do not have the facilities to print certain character codes. These will appear as a '?', and may include the control codes from CHR(0) to CHR(63) and the lower-case alphabet. Other line-printers may be able to print the lower-case alphabet as well as some other character codes.

CHARACTER SETS

ASCII CHARACTER SET

ORD CHR	ORD CHR	ORD CHR	ORD CHR
0 NUL	32 space	64 @	96 `
1 SOH	33 !	65 A	97 a
2 STX	34 "	66 B	98 b
3 ETX	35 #	67 C	99 c
4 EOT	36 \$	68 D	100 d
5 ENQ	37 %	69 E	101 e
6 ACK	38 &	70 F	102 f
7 BEL	39 '	71 G	103 g
8 BS	40 (72 H	104 h
9 HT	41)	73 I	105 i
10 LF	42 *	74 J	106 j
11 VT	43 +	75 K	107 k
12 FF	44 ,	76 L	108 l
13 CR	45 -	77 M	109 m
14 SO	46 .	78 N	110 n
15 SI	47 /	79 O	111 o
16 DLE	48 0	80 P	112 p
17 DC1	49 1	81 Q	113 q
18 DC2	50 2	82 R	114 r
19 DC3	51 3	83 S	115 s
20 DC4	52 4	84 T	116 t
21 NAK	53 5	85 U	117 u
22 SYN	54 6	86 V	118 v
23 ETB	55 7	87 W	119 w
24 CAN	56 8	88 X	120 x
25 EM	57 9	89 Y	121 y
26 SUB	58 :	90 Z	122 z
27 ESC	59 ;	91 [123 {
28 FS	60 <	92 \	124
29 GS	61 =	93]	125 }
30 RS	62 >	94 ^	126 ~
31 US	63 ?	95 _	127 DEL

WRAP-UP INFO

At the end of the compilation listing the compiler prints a few statistics. Some are easy to understand: the number of errors detected by the compiler, the name of the source file, and the name of the code file (executable instructions and data). The other information relates to the storage requirements of the program.

Each procedure or function in the current version of Pascal is mapped into a single code-segment. The section describing these gives the number of segments and their total size in words. Only some of the code segments are in memory at any one time as the MCP overlays them. The other objects given are the value arrays (read only arrays used to hold FORMATS, string constants, and the like), and the data storage. The data storage is accumulated from all the off-stack storage, which includes all records, arrays, and the heap. These, too, are overlaid as necessary by the MCP. The stack space, on the other hand, is never overlaid and permanently resides in main memory as the activation record of the task.

The memory estimates provided by the compiler are naive, but are needed to give the scheduler an initial estimate of the requirements of the Pascal program in execution. If the code-file is repeatedly executed, the MCP will update the information in it to reflect its past history of memory demands so as to give a more accurate picture. The estimate is based on a fixed percentage of the code and value-array space, and of the off-stack space accumulated to give the maximum at any level. The estimates of off-stack space and the stack estimate do not involve tracing possible program flows, and do not take into account any possible recursion.

COMPILER NOTES

COMPILER NOTES

The Pascal compiler is a one-pass compiler based on the various versions of Pascal-P, but not bootstrapped from them. The compiler is written in Burroughs Algol, as are all compilers in the B6700/B7700 system. (Parenthetically, it would be possible now to modify the Pascal compiler so that future compilers could be written in Pascal.)

The compiler makes use of three compiler packages written and copyright by the author: PACKAGE 1 to provide a standard Burroughs input interface, PACKAGE 2 to provide a generally useful code-generation interface, and PACKAGE 3 to add LINEINFO data to the codefile. Executable instructions and read-only data are accumulated in the CODE file which, like much of the B6700/B7700 system, is highly structured and segmented. The code file generated is in the same format as all other code files.

The code generated by the compiler is near-optimal because it is very simple to generate near-optimal code for the B6700/B7700 computers (provided the language does not contain messy constructs such as Algol's FOR loop). The execution speed of programs is not very different from those of Algol or FORTRAN programs. In particular, procedure or function calls (with display update) are efficient compared to other computers.

All one-word and double-word objects are allocated storage in the program stack; records, arrays, and other multi-word objects are allocated a memory segment of the required size with an in-stack descriptor pointing to the segment. The operating system will dynamically overlay the segments in execution according to its memory management algorithm so that an approximation to the working-set of segments is retained in main store. Very long segments (greater than 1023 words) are broken into 256 word "pages": the only case of a fixed-size object. Accessing a multi-dimensional array in lexicographic order will impose the least demand on the operating system (and hence in elapsed execution time) since multi-dimensioned arrays are mapped into a linearly addressed segment in lexicographic order.

All procedures and functions are compiled into a single code-segment each, as is the main program. The only exception is an i/o list of a formatted i/o statement or an i/o list using a file with a structured component, which is compiled as a subprocedure within the code-segment that contains it. Similarly all formats and other read-only multi-word objects are allocated single read-only data segments each. The operating system overlays code and read-only segments in much the same way as data segments, so that programs in

execution acquire storage (apart from the stack) as they need it. Pascal is often likely to have a large number of small segments, thereby leading to more presence-interrupts than say Algol or FORTRAN, and correspondingly longer elapsed time in execution. The processor time spent is not significantly different.

The code generated for expressions is straightforward stack code. All terms of boolean expressions are evaluated; the insertion of branches to "optimize" the evaluation in the conventional sense usually has the effect of making the evaluation slower, and certainly longer.

The code generated for control constructs is straightforward, being composed of boolean expression evaluation and branch tests. The case statement is an exception. Assignment of multi-word objects such as records is achieved by the TWS operator (transfer words); comparison of multi-word objects is currently achieved through a call to an intrinsic.

Normal code generation sets up all declared scalar objects to have a special 'undefined' value. A controlled variable of a FOR loop also acquires this value after normal exit from the loop (not via a GOTO) or if the loop is not entered. Any attempt to utilize this value, for example as an expression value, will cause the INVALID OPERATOR interrupt and lead to the abortion of the program. Objects in records, arrays, or the heap, are not so checked.