# PASCAL 8000

## IBM 360/370 Version

## for OS and VS environments

## REFERENCE MANUAL

## VERSION 1.2

1st february, 1978


Original authors - Hitac version

Teruo Hikita — University of Tokyo
Kiyoshi Ishihata — Japan

Rewritten for IBM 360/370 Operation

Gordon Cox — Australian Atomic Energy Commission
Jeffrey Tobias — Australia

TABLE OF CONTENTS

# 1. INTRODUCTION

Pascal is a general purpose programming language proposed and defined by Wirth(1971a). It was later revised and appeared as Standard Pascal (Jensen and Wirth, 1975). Its principal emphases are on teaching programming and on the reliable and efficient implementation of the language. It now seems to have gained considerable popularity among many computing communities.

Pascal may be considered a successor to ALGOL 60, from which it inherits syntactic appearances . The novelties of Pascal lie mainly in its ample data structuring facilities such as record, set and file structures. It also affords more sophisticated control structures suitable to "structured programming" (Dahl, Dijkstra and Hoare, 1972).

An extended version of Standard Pascal, named Pascal 8000, has been designed, and a compiler implemented for the HITAC 8800/8700 computer (which has an IBM/370-like machine instruction set) under the operating system OS7 at the Computer Centre of the University of Tokyo (Hikita, Ishihata, 1976; Ishihata, Hikita, 1976). This compiler is itself written in Pascal, and is supported by a runtime system written in Fortran and Assembler language. This version has now been adapted at the AAEC research establishment for use on IBM 360/370 computers under the OS family of operating systems. The runtime system has been rewritten entirely in Assembler language with some changes and additions, and additional new language features have been incorporated.

Several proposals for extensions to Pascal have been published, (for example Lecarme and Desjardins, 1975). Extensions implemented at the University of Tokyo are concerned with constant definitions for structured types, variable initialisations, two new control structures (forall and loop), specifications of procedure and function parameters, and specifications of types. Extensions implemented at the AAEC include an exponentiation operator, type-change functions, case-tag list syntax extensions, and extensions to read and write capabilities. We believe that, though they may not be entirely new, the extensions do not disturb the original consistency and transparency of both syntax and semantics, and give the user more power for describing algorithms easily and clearly.

This reference manual gives a complete specification of Pascal 8000, including both the above sets of extensions. Additional information on the operational aspects of the compiler is included in the Appendices. Since this manual is intended to be a rigid and concise description of our implementation, we recommend other appropriate references such as Jensen and Wirth (1975) and Yasamura, Hikita and Ishihata (1975) as introductory guides to the Pascal language.

## NOTATION - BACKUS-NAUR FORM

According to traditional Backus-Naur form (bnf), syntactic constructs are denoted by English words enclosed between the angular brackets < and > . These words also describe the nature or meaning of the construct, and are used in the accompanying description of semantics. Zero or more occurrences of a construct is indicated by enclosing the construct within metabrackets ¯ and ò . The square brackets [ and ] indicate optional constructs. The symbol <empty> denotes the null (zero-length) sequence of symbols.

## Summary of the Extensions to Standard Pascal

Following are brief explanations of the new features implemented in the IBM 360/370 version of Pascal 8000.

### 1. Constant Definitions for Structured Types

In Standard Pascal one may define an identifier as a synonym for constant data, but such constant definitions are applicable only to numbers and strings. In Pascal 8000 this facility is extended to structured types such as the array, record (without variants) and set.

The usual set notation denotes a constant set. The syntax of "constant set" is defined along the lines of "set":

```
<constant set> ::= (. <constant set element list> .)
<constant set element list> ::=
    <constant set element> ¯,<constant set element>ò | <empty>
<constant set element> ::= <constant> | <constant> .. <constant>
```

The following is an example of a constant definition for a set:

```
const evennumbers = (. 0, 2, 4, 6, 8 .);
```

For the array or record type, a new convenient notation is introduced in order to simplify the description of the constant value of these types. Constants of these types are denoted by listing the values of the most basic (simple, string or set type) components sequentially between the two symbols (# and #) .

```
<structured constant> ::= (# <constant or constant set>
                              ¯,<constant or constant set>ò #)
<constant or constant set> ::= <constant> | <constant set>
```

When defining an identifier as a synonym for a structured constant, one is required to specify its type. Thus the syntax of the constant definition becomes:

```
<constant definition> ::= <identifier> = <constant or constant set> |
                          <identifier> = <structured constant> : <type>
```

Following is an example of constant definition for structured constants:

```
const v0 = (# 0.0, 0.0, 0.0 #) : array(.1..3.) of real;
      p1 = (# 'tom', 20, male #) : person;
```

## 2. Variable Initialisations

The ability to initialise variables is introduced in Pascal 8000. Values of variables declared in the outermost block, namely the main program, may be initialised at compile time. The variables that can be initialised are of types simple, string, array, record (without variants) and set. No type specification is necessary. The syntax is

```
<variable initialisation part> ::=
    value <variable initialisation> ¯;<variable initialisation>ò;
    | <empty>
<variable initialisation> ::=
    <entire variable> := <constant or constant set> |
    <entire variable> := <structured constant>
```

The variable initialisation part is placed between the variable declaration part and the procedure and function declarations of the outermost block. For example:

```
value set1 := (. .) ;
      m := (# 0, 0, 0, 0 #) ;
```

## 3. Forall Statements

A forall statement is a new type of control structure operating over the components of a set. It specifies that a statement is to be repeatedly executed while a control variable ranges among all the elements of a certain set. The syntax is

```
<forall statement> ::= forall <control variable> in <expression>
                       do <statement>
```

The expression following the symbol in must be of type set, and the control variable following the symbol forall must have the base type of the set. For example the following statement

```
forall x in set1 do if odd(x) then writeln(x)
```

selects and writes out all the odd numbers from the set set1.

## 4. Loop Statements

A new type of control structure, the loop statement, is introduced to give more sophisticated loop exit control. A loop statement specifies that a group of statements is to be repeatedly executed until control

encounters an event. Events are neither boolean variables nor conditions, but signals that indicate escape from the loop. Control can then be transferred to the statement labelled by the event named in the "postlude" part, and that statement is executed before control leaves the loop statement. The syntax of the loop statement is

```
<loop statement> ::= loop <statement> ¯;<statement>ò end |
    loop until <event> ¯,<event>ò : <statement> ¯;<statement>ò
        postlude <event> : <statement> ¯;<event> : <statement>ò
    end
<event> ::= <identifier>
```

Syntactically, events are used just like procedure calls, but only within loop statements. The scope of an event is the loop statement in which it is defined. The predefined event named exit is provided which means that no "postlude" statement is supposed when escaping from the loop statement. An example of the use of the loop statement is

```
    loop until found, nofound:
         i := i+1;
         if table(.i.) = x then found;
         if i = tablesize then nofound
    postlude found: key := i;
             nofound: errorflag := true
    end
```

## 5. Procedure Skeletons

In Standard Pascal, the syntax for the declaration of a procedure or function with procedure or function parameters is as follows

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

This definition, however, causes some difficulties at run time with the possible conflicts of type and number of the parameters of the procedure or function parameter. It is difficult for the compiler to detect this kind of mismatch in the source program.

In Pascal 8000, the notion of the <procedure skeleton> is employed as a solution to this problem. This solution was originally proposed by Lecarme and Desjardins (1975). The idea is to specify the types of the parameters of procedure or function parameters explicitly. The syntax above is then replaced by

```
<formal parameter section> ::= <parameter group> | var <parameter group>
    | procedure <procedure skeleton> ¯,<procedure skeleton>ò |
    function <procedure skeleton> ¯,<procedure skeleton> : <type>
<procedure skeleton> ::= <identifier> | <identifier> (<type> ¯,<type>ò)
```

In the original proposal, only <type identifier> is allowed rather than the less restrictive <type> for <procedure skeleton>. This kind of extension occurs in two more places in Pascal 8000. (See the following section.)

As an example, the declaration of a function to find zeroes of parameter functions (say, by bisection) would be:

```
function bisect (function f(real) : real) : real;
```

instead of as in Standard Pascal:

```
function bisect (function f : real) : real ;
```

## 6. Type specifications for parameter and function results

It frequently occurs that in Standard Pascal, only a type identifier is allowed instead of the general form of type specification. For example,

```
<parameter group> ::= <identifier> ¯,<identifier>ò : <type identifier>
<result type> ::= <type identifier>
```

In Pascal 8000 this restriction is relaxed to allow <type> instead of <type identifier> in the above type specifications. The syntax is extended to:

```
<parameter group> ::= <identifier> ¯,<identifier>ò : <type>
<result type> ::= <type>
```

As a result, the following declaration of a function is now possible:

```
function f(p : 1..10) :(male,female) ; ...
```

## 7. Exponentiation

Exponentiation is supported. The multiple character which signifies exponentiation is ** . Thus a**7 is equivalent to a¬, and a**b**c is equivalent to a**(b**c).

The bnf syntax for exponentiation is

```
<factor> ::= <facbody> | <facbody> ** <factor>
<facbody> ::= <variable> | <unsigned constant> | <expression> |
              <function designator> | <set> | not <factor>
```

The semantics permits <facbody> to contain only variables, constants expressions and function designators for the purpose of exponentiation.

If the power factor is of type real, then a**b is calculated by evaluating exp(b*ln(a)), and the standard routines for exp and ln are invoked. If the type of the power is integer, then a run-time system

call is made.

## 8. Reading Character Strings

An extension has been made so that variables of type

packed array (.n..p.) of char

(i.e. strings) may be read

The syntax specification is:

read ([file,] x:m) or read ([file,] x)

where x is a variable of type packed array of char, and m is an integer expression, variable or constant. The following cases need to be considered, given that the remaining length of the input record is r.

i) if m is not specified, and length(x)>r, then r characters will be read into x, and length(x)-r blanks inserted to pad x.
ii) if m is not specified, and length(x)<=r, then length(x) characters will be read into x and the input pointer moved up by length(x).
iii) if m is specified, then replace "length(x)" by "min(m,length(x))" in i) and ii) above.

## 9. Extension of procedures "read" and "write" to non-text files

The standard procedures read and write may now apply to any file, text or non-text. Operating on a non-text file, the definitions are

read(f,v1) is equivalent to

begin v1:=f@; get(f) end

and

write(f,v1) is equivalent to

begin f@:=v1; put(f) end

Standard procedures writeln and readln, however, may not be applied to non-text files.

## 10. The Type-change Function

A "type-change" function has been introduced (courtesy Kludgeamus (1976)). The mechanism provided by the standard functions "ord" and "chr" has been extended, and now any type-identifier can be used to change the type of an expression, with the expression result remaining constant. The type function has one argument, of any type. For example

```
var name : packed array (.1..4.) of char;
    ebcdic : integer;

begin
    name:='fred';
    ebcdic:=integer(name)
end
```

## 11. Extended Case Statement

Case-tag lists may now range over a number of constants, without explicitly having to list each constant. The extended range is denoted by:

<constant> .. <constant>

Thus,

4,6..10,15,30..45:

is now a valid case-tag list. A default exit is also supplied by:

else:<statement>

i.e. else: is a valid case tag in every case statement. The else tag will be used if none of the other tags match.

## 12. Additional Standard Procedures and Functions

Four additional standard procedures have been implemented. These are

pack, unpack, halt, message.

An additional standard function, card, is also implemented.

## 13. Additional Standard Type

The standard type alfa is defined to mean packed array (.1..8.) of char.

## 1. VOCABULARY

A Pascal program consists of a sequence of the following basic symbols.

```
<basic symbol> ::= <letter> | <digit> | <special symbol>
<letter> ::=  a | b | c | d | e | f | g | h | i | j | k |  l |  m |  n |
              o | p | q | r | s | t | u | v | w | x | y | z | $ |
                                                                  _
<digit> ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<special symbol> ::=  + | - | * | / | = | <> | < | > | <= | >= | ( | ) |
                      (. |  .) |  (* | *) | := | . | , | [ | ] | & | ; |
                      : | ' | @ | .. | (# | #) | | |  ¯  | ò | ^ | ^= | **
                      | <reserved word>
<reserved word> ::= div | mod | nil | in | or | and | not | if  | then |
                    else | case | of |  repeat |  until |  while |  do |
                    for |   to |  downto |  forall |  loop |  postlude |
                    begin | end | with |  goto |  const |  var |  type |
                    value |  array |  record |  set |  file | function |
                    procedure | label | packed | program
```

The construct

(* <any sequence of characters not containing "*)"> *)

or

¯ <any sequence of characters not containing "ò"> ò

is called a comment. Blanks, end-of-lines and comments are considered to be separators. An arbitrary number of separators may be inserted between any two numbers, strings, identifiers or special symbols without affecting the meaning of the program, subject to the following two rules:

i) Separators may not occur within numbers, strings, identifiers, or reserved words. Blanks within strings have the same meaning as other characters.

ii) At least one separator must occur between two consecutive numbers, identifiers or reserved words.

Note also that the symbols [ and ] can be used for array and set operations if these characters are available. If they cannot be easily utilised, then the substitute character pairs are (. and .). The same applies to the comment braces ¯ and ò, which may be interchanged with (* and *).

Note also that the underscore character acts as an alphabetic this_is_an_identifier is a valid identifier.

# 2. NUMBERS,STRINGS AND IDENTIFIERS.

## 2.1. Numbers

There are two kinds of constant numbers, namely integers and reals.

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

Examples: 1024 3.14 5.772E-1 1E-10

## 2.2. Strings

Sequences of characters enclosed by single quote marks are called strings. Strings consisting of a single character are constants of the standard type char. Strings consisting of n (>1) enclosed characters are constants of type:

packed array (. 1..n .) of char

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

```
<string> ::= ' <character> ¯<character>ò '
```

Examples: 'a' 'don''t' 'is this a string?'

## 2.3. Identifiers

Identifiers serve to denote constants, types, variables, fields, events, procedures and functions.

```
<identifier> ::= <letter> ¯<letter or digit>ò
<letter or digit> ::= <letter> | <digit>
```

The length of an identifier is arbitrary, but only the first eight characters are significant, namely, identifiers with the same first eight characters are considered to be the same identifier. Identifiers must be different from reserved words. Certain identifiers, called standard identifiers, are predefined, such as integer, char and boolean. In contrast to the reserved words, one may redefine any standard identifier. Identifiers must be unique within their scope of definition.

Examples: pi person x5

## 3. CONSTANT DEFINITIONS

A constant definition serves to introduce an identifier as a synonym for constant data. One may define a structured constant of array, record (without variant part) or set type. For structured constants of array or record type, the values of the basic (unstructured) components of the data are listed sequentially, surrounded by the two special symbols (# and #), and followed by a type specification. For multi - dimensional arrays, the order is on the basis of varying the last index first.

```
<constant definition> ::= <identifier> = <constant or constant set> |
                          <identifier> =
                             <structured constant> : <type>
<constant or constant set> ::= <constant> | <constant set>
<constant> ::= <unsigned number> | <sign> <unsigned number> |
               <constant identifier> | <sign> <constant identifier>
               | <string>
<constant identifier> ::= <identifier>
<constant set> ::= (. <constant set element list> .)
<constant set element list> ::= <constant set element>
                                ¯,<constant set elementò | <empty>
<constant set element> ::= <constant> | <constant> .. <constant>
<structured constant> ::= (# <constant or constant set>
                             ¯,<constant or constant set>ò #)
```

```
Examples:       pi = 3.14159
                minuspi = -pi
                title   = 'this is a title'
                oddnumbers = (. 1,3,5,7,9 .)
                vector = (# 0.0,0.0 #) : array (.1..2.) of real
```

# 4. TYPE DEFINITIONS

A type definition associates an identifier with a specific data type. A data type defines the set of values a variable may assume. Every variable occurring in a program must be associated with one and only one type.

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

## 4.1. Simple types

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

### 4.1.1. Scalar types

A scalar type defines an ordered set of values by enumeration of the identifiers which denote these values. An integer value that is transparent to the programmer is associated with each of these identifiers.

<scalar type> ::= ( <identifier> ¯,<identifier>ò )

Examples: (club,diamond,heart,spade)
(sunday,monday,tuesday,wednesday,thursday,friday,saturday)

Besides the user-defined scalar types explained above, the following four scalar types are predefined.

1) boolean. A boolean value is one of the logical truth values denoted by the predefined identifiers true and false. The lexical order is false < true.

2) Integer. This incorporates the integers from -2žœ to 2žœ-1 inclusive. Maxint is a predefined constant with a value of 2žœ-1.

3) Real. Real numbers are expressed in double word, floating format, and can take absolute values between 10-¬« and 10¬µ. This gives an accuracy of approximately 14 decimal digits.

4) Char. Each character is represented by the EBCDIC code. Their ordering and associated integer values are defined by their internal representations.

### 4.1.2. Subrange types

A subrange type is defined as a subrange of another scalar type (except for real, of course), by specifying the minimum and maximum

values of the subrange. The lower bound must be less than the upper.

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

Examples: -100..100 'a'..'z' monday..friday

## 4.2 Structured types

A structured type is specified by its component types and structuring methods. There are four kinds of structuring methods: arrays, records, sets, and files. Both arrays and records may be specified as packed structures. This implies that data elements that can be represented by less than one word will be "packed" together in words, thus occupying the minimum amount of storage. This representation is achieved at the expense of some execution speed.

### 4.2.1 Array types

An array type is a structure consisting of a fixed number of components all of which are of the same type. Each component is distinguished by an index. The index must be a simple type of finite elements (thus, the type real is not allowed as an index).

```
<array type> ::= array (. <index type> ¯,<index type>ò .)
                 of <component type>
<index type> ::= <simple type>
<component type> ::= <type>
```

```
Examples:  array (. boolean .) of week
           array (. 1..10, 1..10 .) of real
           array (. 1..80 .) of char
```

### 4.2.2. Record types

A record type consists of a fixed number of components, called fields, of possibly different types. A record type may have several variants, and a tagfield specifies which variant is applicable at a given time. The scope of a field identifier is the smallest record in which it is defined.

```
<record type> ::= record <field list> end
<field list> ::= <fixed part> | <fixed part> ; <variant part> |
                    <variant part>
<fixed part> ::= <record section> ¯;<record section>ò
<record section> ::= <field identifier> ¯,<field identifier> :<type>
                       | <empty>
<field identifier> ::= <identifier>
<variant part> ::= case <tag field> <type identifier> of
                       <variant> ¯;<variant>ò
<tag field> ::= <identifier> : | <empty>
```

<variant> ::= <record label list> : ( <field list> ) | <empty>
<record label list> ::= <case label> ,<case label>ò
<record label> ::= <constant>

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

record
    name : alfa;
    case sex : boolean of
       true : (age : integer);
      false : (height : real)
end
```

### 4.2.3. Set types

A set type defines the range of values which becomes the powerset of its so-called base type. A base type must be a simple type (except for real), and its associated integer values must be between 0 and 63. (For this reason, a set of characters is not allowed due to the representation by the EBCDIC code).

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

<base type> ::= <simple type>

Examples: set of 1..10 ; set of monday..friday

### 4.2.4. File types

A file type is a structure consisting of a sequence of components all of which are of the same type. Components must in turn not be of type file.

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

Examples:
```
file of integer
file of array (. 1..3 .) of real
```

The standard file type "text" specifies the files of component type char, i.e. file of char. Files of type text are considered to be subdivided into lines, each separated by an end-of-line marker. The standard file names "input" and "output" are of type text, and represent the standard input and output files for the user.

### 4.2.5. Pointer types

A variable of pointer type serves to "point" to data and contains the address of the item pointed to. The pointer value nil belongs to every pointer type, and it in turn points to no data at all. The dereference operator "@" can be used to access the data being pointed to by a pointer variable or structure.

<pointer type> ::= @ <type identifier>

Examples: @integer @person

# 5. DECLARATIONS AND DENOTATIONS OF VARIABLES

A variable declaration specifies the variables local to a procedure or function, and associates them to their data types. Every variable occurring in a statement must have been previously declared.

<variable declaration> ::= <identifier> ¯,<identifier>ò : <type>

Examples: x,y,z : real p1,p2 : person

Denotations of variables either designate entire variables, components of variables, or variables referenced by pointer variables.

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

## 5.1. Entire variables

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

## 5.2. Component variables

Component variables are the components of the data of array, record or file type.

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

### 5.2.1. Indexed variables

An indexed variable is a component of a variable of type array. The types of the expressions used as indices must coincide with those that are defined as index types.

<indexed variable> ::= <array variable>
(. <expression> ¯,<expression>ò .)
<array variable> ::= <variable>

Example: matrix (. i+1, 2*j+1 .)

### 5.2.2. Field designators

A field designator is a component of a variable of record type.

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

Example: person.name family.parents.son

### 5.2.3. File buffers

Only one component of a file is directly accessible at any instant. This component is represented by the buffer variable of the component type.

<file buffer> ::= <file variable> @
<file variable> ::= <variable>

Example: file1@

## 5.3. Referenced variables.

A referenced variable is a variable pointed to by a pointer variable.

<referenced variable> ::= <pointer variable> @
<pointer variable> ::= <variable>

Example: p1@ s@.r@

## 6. VARIABLE INITIALIZATIONS

Variables declared in the outermost block may be initialized at compile time. Variables of array and record type (without variant parts) are initialized by structured constants.

```
<variable initialization> ::= <entire variable> :=
                                 <constant or constant set> |
                              <entire variable> :=
                                 <structured constant>
```

```
Examples:     k := 0
              m := (# 1.0,1.0,1.0 #)
```

## 7. EXPRESSIONS

An expression consists of operators and operands, where operands are either constants, variables or function designators. It specifies a rule for evaluation of a value, where the conventional rules of left to right evaluation and operator precedence are observed. The operator not has the highest precedence, next the exponentiation operator, next the multiplying operators, then the adding operators, with the relational operators taking the lowest precedence. An expression enclosed within parentheses is evaluated independently of preceding or following operators.

```
<expression> ::= <simple expression> |
                 <simple expression> <relational operator>
                     <simple expression>
<simple expression> ::= <term> | <sign> <term> |
                        <simple expression> <adding operator> <term>
<term> ::= <factor> | <term> <multiplying operator> <factor>
<factor> ::= <facbody> | <facbody> <exponentiation operator>
                             <factor>
<facbody> ::= <unsigned constant> | <variable> | <set> |
              <function designator> | not <factor> |
              ( <expression> )
<set> ::= (. <element list> .)
<element list> ::= <element> ,<element>ò | <empty>
<element> ::= <expression> | <expression> ..  <expression>
```

```
Examples:
      (set)     --  (. 3,6,9 .)      (. sunday,monday .)
   (factor)     --  12  x   x**y    sin(0)
                    (a+b+c)      3**x**2.0
     (term)     --  x  y       b1 and b2
   (simple
   expression)  --  -pi    i-(j-k)      b or odd(n)
  (expression)  --  2 in set1       x<=y     3*j<>k
```

## 7.1. Operators

### 7.1.1. The operator not

The operator not implies the logical negation of its boolean operand. Note that the symbol ^ may also be used.

### 7.1.2. Exponentiation operator

<exponentiation operator> ::= **

The type of the result of raising a to the power b is defined by the following table.

| a / b | real | integer(>=0) | integer(<0) |
|---|---|---|---|
| integer | real | integer | undefined |
| real | real | real | real |

Note that the exponentiation operator is right associative

Example: a**b**c is equivalent to a**(b**c)

### 7.1.3 Multiplying operators

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

| operator | operation | operands | result |
|---|---|---|---|
| * | multiplication<br>set intersection | real, integer<br>set type T | real, integer<br>set type T |
| / | division | real, integer | real |
| div | division | integer | integer |
| mod | modulus | integer | integer |
| & | logical conjunction | boolean | boolean |
| and | logical conjunction | boolean | boolean |

1) As long as at least one of the operands is of type real, the result is real.

### 7.1.4. Adding operators

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

| operator | operation | operands | result |
|---|---|---|---|
| + | addition<br>set union | real, integer<br>set type T | real, integer<br>set type T |
| - | subtraction<br>set difference | real, integer<br>set type T | real, integer<br>set type T |
| \| | logical disjunction | boolean | boolean |
| or | logical disjunction | boolean | boolean |

1). As long as at least one of the operands is of real type, the result is real.

### 7.1.5. Relational operators

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

| operator | operation | operands | result |
|---|---|---|---|
| = <> ^= | equality,<br>inequality | simple, string, set<br>or pointer types | boolean |
| <= >= | order<br>set inclusion | simple or string types<br>set types | boolean<br>boolean |
| < > | order | simple or string types | boolean |
| in | set membership | base type and set type | boolean |

## 7.2 Function designators

A function designator is used in expressions to invoke a function. Formal parameters of the function are replaced by the actual parameters.

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

Examples: gcd(k,1024) sin(x+y)

# 8. STATEMENTS

A statement is the unit for the execution of a program. Statements may be preceded by labels which designate them as the destination of goto statements. Labels are unsigned integers composed of at most four digits. The scope of a label is the entire text of the block in which it is declared.

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

## 8.1. Simple statements

Simple statements are statements not composed of other statements. Events are used only in loop statements.

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

### 8.1.1. Assignment statements

An assignment statement serves to replace the value of the left-hand variable with the evaluated value of the right-hand expression.

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

Both sides of the statement must be of the same type (file types are not allowed). The exceptions are when

1) The type of a variable is real and the type of an expression is integer. In this case the integer value of the expression is converted to real value.
2) One of them is a subrange type of the other.

```
Example:     found := x <> 0
             add := p1 + p2
```

### 8.1.2. Goto statements

A goto statement indicates that control of execution is to be transferred to another part of the program text, namely to the place of the label.

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

Every label must be declared in the label declaration part of the procedure in which the label is defined. It is not possible to jump into a procedure, but jumping out of a procedure is possible. The effect of a jump from outside a structured statement into that statement is not defined.

Examples: goto 9999 goto 1977

### 8.1.3. Procedure statements

A procedure statement activates the procedure named by the statement. Formal parameters of the procedure are replaced by the actual parameters. There are four kinds of parameters - value, variable, procedure and function.

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

Examples: f(2.0,x1,fun1)
cos(pi)

## 8.2. Structured statements

Structured statements are statements which themselves are composed of several other statements.

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

### 8.2.1. Compound statements

A compound statement specifies that component statements are to be executedin the same sequence as they are textually written.

<compound statement> ::= begin <statement> ¯;<statement>ò end

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

### 8.2.2. Conditional statements

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

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

### 8.2.2.1. If statements

An if statement specifies that a statement is to be executed only if a certain boolean expression is true. If the expression is false, then either no statement, or the statement following the symbol else, is executed.

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

The syntactic ambiguity arising from the construct:

```
if e1 then if e2 then s1 else s2
```

is resolved by interpreting it as equivalent to:

```
if e1 then begin if e2 then s1 else s2 end
```

```
Examples:     if x=y then table(.i.):=true
              if b1 then a:=0 else a:=a+1
```

### 8.2.2.2. Case statements

A case statement consists of an expression called the selector, and a list of statements, each labelled by either a constant, or a range of constants, of the type of the selector, or the default label "else:". There can be only one default label per case statement. The selector type must be simple (except real). The case statement selects for execution that statement the label of which contains the current value of the selector. If no such label exists, and no default label was specified, an execution error results. If a default label is supplied, the default path is taken.

```
<case statement> ::= case <expression> of <case list element>
                     ¯;<case list element>ò end
<case list element> ::= <case label list> : ¯<statement> | <empty>
<case label list> ::= <case statement label> ¯,<case statement label> ò
                      | else
<case statement label> ::= <constant> | <subrange>
```

```
Example:     case i+2*j of
                    3    : z := sin(x);
                -1..1,10 : z := cos(x)
                    7    : z := tan(x);
                    else : z := 0
             end
```

### 8.2.3. Repetitive statements

Repetitive statements specify that certain statements are to be repeatedly executed. Escaping from the loop is controlled by several

methods, depending on the statement used.

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

### 8.2.3.1. While statements

A while statement indicates that a statement is to be repeatedly executed while the value of a certain boolean expression is true. The boolean expression is evaluated before each iteration.

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

```
Example:     while m(.i.)=0 do i:=i+1
```

### 8.2.3.2. Repeat statements

A repeat statement indicates that a group of statements is to be executed repeatedly until the value of a certain boolean expression becomes true. The boolean expression is evaluated after each iteration. Therefore, the group of statements is executed at least once.

```
<repeat statement> ::= repeat <statement> ¯;<statement>ò
                       until <expression>
```

```
Example:     repeat x:=x+m(.i.); i:=i+k until i=c
```

### 8.2.3.3. For statements

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

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

The control variable, the initial value, and the final value must be of the same simple type. They may not be of type real. The value of the control variable must not be altered in the repeated statement. The initial and final values are evaluated only once. If, in the case of to (downto), the initial value is greater (less) than the final value, the controlled statement is not executed. The final value of the control variable is left undefined on normal termination of the for statement.

```
Examples:     for i:=1 to 100 do x:=x+y(.i.)
              for j:=10 downto -c do if j=a then goto 99
```

### 8.2.3.4. Forall statements

A forall statement indicates that a statement is to be repeatedly executed for each element of a certain set.

```
<forall statement> ::= forall <control variable> in <expression>
                       do <statement>
```

The type of the variable must be that of the base type of the expression, which in turn must be of set type. The value of the control variable must not be altered in the repeated statement. The set expression is evaluated only once. The final value of the control variable is left undefined on normal exit from the forall statement.

```
Examples:     forall day in week do wage:=wage+c
              forall x in (.0,1.) + set0 do count:=count+1
```

### 8.2.3.5. Loop statements

A loop statement indicates that a group of statements is to be repeatedly executed until an event name is encountered. Control is then transferred to the statement labelled by that event name in the postlude part. Before termination of the loop statement, that statement is executed once.

```
<loop statement> ::= loop <statement> ¯;<statement>ò end    |
                loop until <event> ¯,<event>ò : <statement>
                                                ¯;<statement>ò
                     postlude <event> : <statement> ¯;<event> :
                                                <statement>ò
                end
<event> ::= <identifier>
```

Syntactically, events may be used like statements, but only within loop statements. The scope of an event is the loop statement in which it is defined. The predefined event name exit means that no "postlude" statement is supposed when escaping the loop statement.

```
Example:     loop until found,nofound:
              if table(.i.)=x then found;
              i:=i+1;
              if i=tableix then nofound
             postlude
              found:   ;
              nofound:  begin table(.tableix.):=x;
                         tableix:=tableix+1   end
             end
```

### 8.2.4. With statements

A with statement opens the scope containing the field identifiers of the specified record variables, so that the field identifiers may occur as variable identifiers. Within it the fields of record variables may be designated only by its field identifiers.

```
<with statement> ::= with <record variable list> do <statement>
<record variable list> ::= <record variable> ¯,<record variable> ò
```

No assignments may be made to any element of the record variable list in the with statement.

```
Example:     with person do begin name:='mari'; age:=20 end
```

## 9. PROCEDURE DECLARATIONS

Procedure declarations serve to define a program part which can be activated (possibly recursively) by procedure statements.

```
<procedure declaration> ::= <procedure heading> <block>
<procedure heading> ::= procedure <identifier> ; |
             procedure <identifier>
                              ( <formal parameter section>
                              ¯;<formal parameter section>ò ) ;
<formal parameter section> ::= <parameter group> | var <parameter group>
    | procedure <procedure skeleton> ¯,<procedure skeleton>ò
    | function <procedure skeleton> ¯,<procedure skeleton>ò : <type>
<parameter group> ::= <identifier> ¯,<identifier>ò : <type>
<procedure skeleton> ::= <identifier> |
                   <identifier> ( <type> ¯,<type>ò )
```

An identifier following the symbol procedure denotes the name of the procedure. The formal parameter section lists the name of each formal parameter followed by its type. Four kinds of parameters are possible: value parameters, variable parameters, procedure parameters and function parameters. For the value parameters, the actual parameters must be expressions, and their values are evaluated and passed when the procedure is called. Parameters preceded by the symbol var are variable parameters, and the corresponding actual parameters must be variables. When the procedure is called, variable formal parameters are replaced by actual parameters. File parameters must be specified as variable parameters. Parameters preceded by the symbols procedure and function are procedure and function parameters respectively. Corresponding actual parameters are procedures and functions, with the following rules:

1) They must have value parameters only.
2) The number and types of their parameters must coincide with those specified in the procedure skeletons.
3) Actual parameters must not be the standard procedures or functions.

```
<block> ::= <label declaration part>
            <constant definition part>
            <type definition part>
            <variable declaration part>
            <procedure and function declaration part>
            <statement part>
<label declaration part> ::= label <label> ¯,<label>ò ; | <empty>
<constant definition part> ::= const <constant definition>
               ¯;<constant definition>ò ; | <empty>
<type definition part> ::= type <type definition>
                     ¯;<type definition>ò ; | <empty>
<variable declaration part> ::= var <variable declaration>
                  ¯;<variable declaration>ò ; | <empty>
<procedure and function declaration part> ::=
              ¯<procedure or function declaration>;ò
<procedure or function declaration> ::= <procedure declaration> |
                                        <function declaration>
```

<statement part> ::= <compound statement>

The label declaration part lists all the labels which mark a statement in the statement part of this block. The constant definition part defines all the synonyms for constants local to the block. The type definition part contains all the type definitions local to the block. The variable declaration part contains all the variable declarations local to this block. The procedure and function declaration part defines subordinate program parts, namely, procedures and functions. Labels, constants, types, variables, procedures and functions have significance only within the block in which they are declared, which is called the scope of these items. If a name is redefined within a block, the scope of the second occurrence of the name is excluded from the scope of the first. All labels and identifiers must be declared before they are referenced. The following two exceptions are however allowed.

1) The type identifier in a pointer type definition.

2) Procedure and function calls when there is a forward reference.

The statement part specifies the actions to be taken when this procedure is activated.

If a procedure is referenced before its declaration appears, then a forward declaration must be made before the reference. The general format is

<procedure heading> forward ;

In this case a parameter list (and result type for the case of functions) is unnecessary at the actual procedure declaration.

## 9.1. Standard procedures

### 9.1.1. File handling procedures

In the following, the parameter f is a variable of file type.

1) reset(f) resets the buffer variable of f to the beginning of the file. eof(f) becomes false if f is non-empty; otherwise, f@ is undefined and eof(f) becomes true. It cannot be applied to the standard file input.

2) rewrite(f) precedes the rewriting of the file f. The current value of f is replaced with the empty file. It cannot be applied to the standard file output.

3) get(f) advances the current file position to the next component. If no next component exists, then eof(f) becomes true and the value of f@ is undefined.

4) put(f) appends the value of the buffer f@ to the file f.

5) page(f) is applicable only to textfiles. Instructs the printer to skip to the top of a new page before printing the next line of the textfile f.

### 9.1.2. Dynamic allocation procedures

In the following, the parameter p is the variable parameter of pointer type.

1) new(p) allocates a new variable and assigns its pointer reference to the pointer variable p. If p is of record type with variants, then the form

2) new(p,t1,...,tn) can be used to allocate a variable of the variant with tag field values t1...tn. The tag field values must be listed contiguously and in the order of their declaration. Note that the use of "new" with tags actually assigns the record tags with the values t1...tn; no further tag assignment is necessary. NOTE. This scheme of automatic tag assignment differs from the description in the manual by Jensen and Wirth. It also differs from the Cyber implementation of Pascal. However, the authors believe that it is the correct approach, and the user may choose to make explicit tag assignments anyway if he/she so desires.

3) mark(p) A pointer variable p is set to point to the current end of the area allocated for the dynamically generated data.

4) release(p) The end of the area currently occupied by dynamically generated data is reset to the place pointed to by p, so that the dynamically allocated area beyond the place pointed to by p is released.

### 9.1.3. Data transfer procedures

pack(a,i,z) means

```
for j:=u to v do
z(.i.) := a(.j-u+i.)
```

unpack(z,a,i) means

```
for j:=u to v do
a(.j-u+i.) := z(.i.)
```

where 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
and i, j, u and v are of type integer.

Note that the bounds on i are:

m <= i <= u-v+n

Run-time bounds checking on variable i is optionally performed by the compiler.

### 9.1.4. Further standard procedures

In the following, the parameter s is the variable parameter of type alfa

1) time(s) gives the current time in the form hh:mm:ss, where hh denotes the hour, mm denotes the minute and ss the second.

2) date(s) gives the current date in the form dd/mm/yy, where yy denotes the last two digits of the year, mm the month and dd the day of the month.

3) message(x) The string x is written into the joblog. x should contain at most 80 characters.

4) halt terminates the execution of the program and issues a post-mortem dump if the program was appropriately compiled(i.e. using the $P+ option).

# 10. FUNCTION DECLARATIONS

Functions are subroutines which yield a single scalar or pointer value

```
<function declaration> ::= <function heading> <block>
<function heading> ::= function <identifier> : <result type> ; |
                       function <identifier>
                        ( <formal parameter section>
                        ,<formal parameter section>ò ) : <result type>;
<result type> ::= <type>
```

At least one assignment to the function must appear in the statement part of the function declaration.

## 10.1. Standard functions.

### 10.1.1. Predicates

In the following, the parameter x is called by value, and the parameter f is called as a variable parameter.

1) odd(x) x is of type integer. The result is true if x is odd, and false if x is even.

2) eof(f) f is of file type. The result is true if f is in an end-of-file state. It cannot be applied to the standard files input and output.

3) eoln(f) f is of file type. The result is true if f is in an end-of-line state. It cannot be applied to the standard files input and output.

### 10.1.2. Arithmetic functions

1) abs(x) Absolute value of the number x.

2) sqr(x) Square of the number x.

3) sqrt(x) Square root of the number x.

4) exp(x) Exponential function e**x.

5) ln(x) Natural logarithm of x.

6) sin(x) Trigonometric function.

7) cos(x) Trigonometric function.

8) arctan(x) Inverse trigonometric function.

The parameter x is called by value. The type of x may be real or integer. The type of the result is the same as that of x for 1) and 2), and real for 3)-8).

### 10.1.3. Transfer functions

In the following, the parameter x is called by value.

1) trunc(x) x is of type real, and the result is the truncated value of x.

2) round(x) x is of type real, and the result is the nearest integer to x.

3) ord(x) x is of any simple type, except real, and the result is the integer value associated with x.

4) chr(x) x is of type integer, and the result is the character whose associated value is x, if it exists.

### 10.1.4. Further standard functions

In the following, the parameter x is called by value.

1) succ(x) x is of any simple type, except real, and the result is the successor to x. It is undefined if one does not exist.

2) pred(x) x is of any simple type, except real, and the result is the predecessor of x. It is undefined if one does not exist.

3) clock yields an integer value equal to the central processor time, expressed in milliseconds, already used by this job.

4) card(x) x is of set type. The returned result is the cardinality of x (i.e. the number of elements contained in the set x).

### 10.1.5. The "type-change" function

A type changing function has been introduced. The mechanism provided by the standard functions ord and chr has been extended, and any <type identifier> can be used to change the type of an expression, with the expression result remaining unchanged. The type-change function has one argument, of any type. Note that the argument may not be a constant.

```
Example: var  name : packed array  (. 1..4 .) of char;
                ebcdic: integer;
         begin
             name := 'fred';
            ebcdic:= integer(name)
         end;
```

NOTE: The type change function can create havoc in a program if used incorrectly. It is recommended for use by advanced programmers only.

# 11. INPUT AND OUTPUT PROCEDURES

Four standard procedures, read, readln, write, and writeln are provided as the usual and convenient facilities for input and output. They are applied to files of type text, besides the standard files input and output, and further, procedures read and write may be applied to files of any type.

## 11.1 The procedure read

The general format is

read ( [f,] v1,v2, ... vn )

where f is an optional filename. If f is omitted, the standard file input is assumed. v1,v2 ... vn are variables of type integer, real, char, or packed array (1..n) of char when f is of type text. When f is not of text type, then v1,v2 ... vn should be of types equivalent to the component types of f.

The above read statement is equivalent to:

begin read([f,]v1); ... read([f,]vn) end

### 11.1.1. f of type text

If v is of type char, then read(f,v) is equivalent to

begin v:=f@; get(f) end

If a parameter v is of type integer or real, a sequence of characters which represents an integer or real number is read into v (that is, free-format input). Consecutive numbers must be separated by blanks or end-of-lines.

If a parameter v is of type packed array (. 1..n .) of char, the specification of the read procedure is extended to

read ( [f,] v [ :m ] )

where m is an integer valued expression. For this form, the following cases need to be considered, given that the remaining length of the input record is r.

1) if n is not specified, and n>r, then r characters will be read into v, and n-r blanks inserted to pad v to the right. The input pointer now points to the end of the input record.

2) if m is not specified, and n<=r, then n characters will be read into v and the input pointer moved up by n characters.

3) if m is specified, then replace n by min(n,m) in 1) and 2) above.

### 11.1.2. f of non-text type

If f is not of type text, then read(f,v) is equivalent to

```
begin  v:=f@; get(f) end
```

## 11.2. The procedure readln

The procedure readln is identical to read except that, after reading the values into the variables, it skips the remainder of the current record, and the pointer f is positioned at the beginning of the next record. readln may only be applied to text files.

## 11.3. The procedure write

The procedure write has the following general format:

```
write ( [f,] p1,p2, ... pn )
```

where f is a file of any type, and p1,p2 ... pn are the parameters of the form defined below. The above is equivalent to

```
begin write([f,]p1); ... ; write([f,]pn) end
```

If f is a textfile, then write appends character strings (one or more characters) to the textfile. In this case, the general format of the parameters p1,p2 ... pn is either one of

```
e       e:m         e:m:n
```

where e is an expression, the value of which is to be written out. The type of the expression e may be one of:

```
boolean     integer      real

char      packed array (.1..p.) of char
```

m and n are integer valued expressions, where m denotes the number of columns for e with preceding blanks, and n specifies the fraction length if e is of type real.

If the value e requires less than m characters for its representation, then an adequate number of blanks is issued so that exactly m characters are written, with the value right justified. If

the number of characters required to represent e exceeds m, then the specified field width is expanded to enable the full value of e to be written. When m is not explicitly specified, the following default values are employed:

| type | default m |
|---|---|
| boolean | 4 or 5 |
| integer | 12 |
| real | 24 |
| char | 1 |
| string | length of string |

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

If the file f is not of type text, then

```
write (f,p)
```

is equivalent to

```
begin f@ := p; put(f) end
```

## 11.4. The procedure writeln

The procedure writeln is entirely the same as write, except that, after writing out the value of the expression, an end-of-line marker will be written. Note that writeln may not be applied to non-text files.

## 11.5. Printer-control characters

If a text file is to be sent to the printer, the first character of each line is interpreted as a control character by the printer, and is not printed. The control characters are interpreted as follows:

| character | action |
|---|---|
| '+' | no line feed (overprinting) |
| Blank | single spacing |
| '0' | double spacing |
| '-' | triple spacing |
| '1' | new page before next line of printing |

## 12. PROGRAMS

A Pascal program consists of a program heading and a block, possibly with a variable initialisation part. A name following the symbol program is a user program name , and it has no further significance in the program. Program parameters are the names of the external files used in the program. The outermost variables may be initialised by the variable initialisation part.

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

 <program heading> ::= program <identifier>
                        [ ( <program parameters> ) ] ;
 <program parameters> ::= <file variable> ¯,<file variable>ò
 <variable initialisation part> ::= value <variable initialisation>
                                    ¯;<variable initialisation>ò | <empty>
```

Example:

```
program writeout(file1, output);
var file1 : file of integer; b : integer;
begin reset (file1);
      while not eof(file1) do
            begin b:=file1@; writeln(b);
                get(file1)
            end
end.
```

ACKNOWLEDGEMENTS

In the original report from the University of Tokyo(Hikita and Ishihata,1976), the following acknowledgement was included:

"We are grateful to the many people who assisted us in various ways during the work. Our compiler is based on Dr. H. H. Naegeli's "trunk" compiler, which Professor T. L. Kunii arranged to be sent to us. Our special gratitude goes to Professor H. Ishida of the Computer Centre for his supervision and support of the project, to Professor E. Goto for his supervision, and to Mr. M. Yasumura, our previous coworker (now at UCLA), for his contribution to the work. The implementation is done as a cooperational research project with the Computer Centre."

The Australian authors are very grateful to Professor Teruo Hikita and his co-workers, for supplying their compiler on which the IBM 360/370 version is based.

We also acknowledge helpful discussions with staff in the Computer Science departments of the University of New South Wales and the University of Sydney. We also thank the Sydney Water Board for generously giving us access to their computer centre to enable testing under the SVS and MVS operating systems to be carried out.

## APPENDIX 1 Compiler features

### 1. Listing Format

Compiled programs are listed in an environment designed to provide useful information to the programmer about program size and structure. Further, several options are available to control the listing produced, as well as to select options that affect compilation.

Heading: Each page is headed by one line, indicating the version of Pascal that is currently executing, the date and time of compilation, as well as a page number. A title is also printed, if one has been defined.

Listing: Consider the following example

```
PASCAL 8000/2    AAEC (01 JULY 77)

0630 --    PROGRAM PUT3(OUTPUT); (*THIS SHOULD PRINT '3' *)
0630 --     VAR I : INTEGER;
0634 -- A  FUNCTION DUMMY : INTEGER;
0000 00 A   BEGIN DUMMY = -1 END;
001C -- A  PROCEDURE P(FUNCTION F:INTEGER);
0048 --     VAR L : INTEGER;
004C -- B  FUNCTION R:INTEGER;
0000 0- B     BEGIN (* R *)
000A --         R:=L;    (* SHOULD PASS VALUE OF L BOUND *)
0012 -0 B     END;       (* WHEN R WAS PASSED AS A PARAMETER*)
0000 0- A  BEGIN (* P *)
000A --      I := I + 1; L := I;
001E --      IF I = 3 THEN P(R)
0036 --        ELSE IF I = 5 THEN WRITELN(' ',F)
008E --          ELSE P(F)
009C -0 A  END; (* P *)
0000 0-    BEGIN
000A --       I := 0;
0010 --       P(DUMMY)
001C -0    END.

*AAEC PASCAL COMPILATION CONCLUDED *

*NO ERRORS DETECTED IN PASCAL PROGRAM *
```

The four hexadecimal digits on the side of the page indicate the relative addresses of variables, data and code, wherever appropriate. While variables are being declared with the var construct, the hex address will reflect the relative offsets from the start of the stack for the procedure being compiled. (A fixed amount of space is required for each procedure before variables can be allocated, and this is 40(hex) bytes.)

For statements, the hexadecimal address indicates the relative offset from the start of the code for that procedure. The value shown is the offset at the start of each line of listing, before code has been generated for that line of Pascal source. These relative addresses are most useful for determining the size of procedures, as well as for relating to post-mortem dump information.

The next two indicators are known as nest level indicators, and reflect the static block structure of a procedure. The left indicator is incremented, and printed, whenever a begin, loop, repeat, or case is encountered. On termination of these structures, with an end or until, the right indicator is printed, and the static level counter decremented. This scheme makes it very convenient to match begin - end pairs, while quickly pointing to missing end terminators. A correctly composed procedure should commence with a zero left indicator and terminate with a zero right indicator.

The character that follows the nest indicator reflects static procedure levels. The character is updated for each nest level ('A' for level 2, 'B' for level 3, etc) and printed next to the heading and the begin and end associated with that procedure. It is therefore possible to see at a glance the static level nesting of each procedure. This indicator is also useful in finding missing end terminators.

The input Pascal source line is printed following these indicators. Each line of source text is printed exactly as read. Blank input lines appear as such.

## 2. Listing control

There are several options available for the programmer to control his output listing. These are indicated to the compiler by a '$' character in column one of the source input record, immediately followed by the option keyword. '$option' cards are not printed on the output listing. The options available are:

$title <title> replaces the title currently printed (if at all) with <title> and then skips to a new page. The title is printed at the top of each page, until a new $title record is encountered, or an $untitle record is found. Only the first 40 characters of the supplied title are relevant.

$eject causes the next line of listing to appear on a new page (unless that line is $untitle)

$space n n blank lines are printed in the program listing.

$untitle compiler-generated page skipping, and titling, is suppressed.

## 3. Compiler options

Several compiler options are provided in order to control the modes of compilation. Compiler options are specified by the first part of any comment. The general format is as follows:

(*$x+,y-, ... <any comment> *)

where x, y, ... are the compiler options described below, and the symbol '+' means the activation of the option, and the symbol '-' means the suppression of the option. The specification of options may be inserted anywhere in the program, so that users can control the code generation selectively over specific parts of a program.

Compiler options.

C — indicates that the object code produced by the compiler should be listed in assembly language format. The default value is '-'.

L — indicates that the source program should be listed. The default is '+'.

T — indicates that code to provide run time checking should be generated. Examples of checking include:

1) assignment of values to variables of type <subrange>.

2) ensuring that array index operations are within the bounds of the array as specified.

3) ensuring that case statement selection falls within the realm of one of the case tags.

The default value is '+'.

U — restricts compilation to the first 72 columns of the input record. The remainder of the record is effectively ignored, but listed by the compiler. The default value is '-', and the first 120 columns are relevant.

P — instructs the compiler to produce the code necessary to generate a traceback and full post-mortem dump of local variables if an execution error were to occur. The default is '+'.

N — instructs the compiler to produce the code necessary to generate a traceback (without dump of variables) if an execution error were to occur. (linkage-editor version only). The default is '-'.

S the compiler will flag with a warning message all constructs that are not 'Standard Pascal'. The default is '-'.

## 4. Compilation error messages

Errors detected by the compiler during compilation of a Pascal program will be flagged both by an error number and an error message. The erroneous line will be marked with an '@' pointing to just past the symbol in error on that line in the listing.

A log of compile-time error messages that may have been generated throughout the program is printed at the conclusion of compilation. The compiler will print the text for each message only once, no matter how many occurrences of that error number appeared.

Programs that have compiled with errors cannot be executed.

## 5. Execution summary

In all cases where the runtime system completes execution normally (with or without Pascal errors) without a system abend, an execution summary of all steps is output to the SYSPRINT dataset. This gives the time taken for each step, plus an indication of the main storage used by the program, the stack and the heap for that step. Use of main storage for the runtime system, system I/O control blocks and other sundry operating system requirements is not included in this summary. Partitioning of storage into stack and heap usage may not be accurate, as it is determined by initialising the stack-heap area before the step, and considering the largest area remaining untouched after the step to be the unused area between the stack and the heap. Step timing includes the overhead of performing this initialising and examination.

## 6. Post mortem dump

If any error condition is detected by the Pascal runtime system, execution terminates, and after printing the execution summary described above, the nature of the error is printed, followed by a post-mortem dump of the segments (procedures or functions) which were active at the time of the error. Only the innermost n1 and the outermost n2 segments are dumped, together with a count of the ones omitted. n1 and n2 both default to a value of 5, but they may be changed separately through the PARM field on the EXEC card. Offsets from the start of each segment to the call of the next or the location of the error are printed in hexadecimal, and may be related to the addresses printed on the left of the compiler listing of the relevant segment.

Within each segment, a local variable dump is provided if the $P+ option was specified (or implied by default) for compiling that

segment. Variables included in this dump are all those of types integer, real, char(EBCDIC character enclosed in quotes printed), alfa (8 characters enclosed in quotes printed), boolean (<TRUE> or <FALSE> printed), user-defined scalar variables (ordinal value followed by (S) printed), and pointer (hexadecimal value or <NIL> printed). Arrays (except alfa), sets and other user-defined structures are not included in the local variable dump.

All stack and heap space is initialised at the start of a Pascal program execution so that all bytes contain x'7f'. If a local variable is found to contain this value, it is printed as <UNDEFINED>. Not all undefined variables will be printed in this way however, as stack initialisation is done at program-start time rather than at segment-start time. Thus if a segment call uses an area of stack which has been previously used by another segment, its local variables will be initialised to seemingly random rather than 'undefined' values.

Error conditions which cause termination are of several types. Errors such as compile-time errors and program loading errors do not give the traceback dump. Errors such as value out of bounds, stack overflow or time overflow give a simple message followed by the traceback. Errors in using the file-system in an incorrect way give a message including the file-name referred to, followed by the traceback. Errors in calling a standard function with a value out of range give a message including the incorrect argument value, followed by the traceback. Program interrupts are trapped by the runtime system, and cause termination of the run, usually with a traceback. An initial printout for the interrupt gives the program-old psw, the general register values and an indication of whether the interrupt occurred in compiled code or in the runtime system. Storage contents surrounding the point of interrupt are printed if the interrupt is from compiled code, and the traceback as for other errors is then given. If the environment of the interrupt cannot be determined, a system abend is given with code 000.

At the start of a step, the CPU time remaining for the job is determined, and a trap is set for about 10 seconds before this time expires. When this trap is sprung, an error message, followed by the post-mortem dump is printed. At this time, any local files which were open are closed and scratched. (At the time of writing, this does not work on SVS and MVS systems, and CPU time expiry results in a S322 abend on these systems.)

Of course the printing af the post-mortem dump relies critically on the integrity of the stack after the error has occurred, and a correct print-out may not be obtained if the stack has been corrupted. Stack corruption will rarely occur if the $T+ option is used. If stack corruption causes a program interrupt while trying to print a local variable dump, the dump for that segment is abandoned, and the next is attempted. If stack corruption is detected while trying to trace segment calls, the whole traceback is abandoned.

The post-mortem dump capability has proved most useful when debugging Pascal programs. It is at its most useful when used with the $P+ and $T+ options (which are the default), although these incur slight storage and execution speed penalties. The $P+ option incurs a time penalty of one instruction execution per segment call and a space penalty of 12 bytes per segment plus 12 bytes per local variable. The $T+ option incurs a time penalty of 5 machine instructions and a space penalty of up to about 26 bytes per bounded variable assignment, per array subscript evaluation, per case statement and per pointer variable reference.

## 7. Miscellaneous

1) The external file name "output" does not have to appear in the "program" parameter list, unless operations on the file "output" are to be performed.

2) External files do not have to be declared at level one in a program, but may be declared in procedures at any level. All the conventions regarding local files are applicable to external files, and further, a DD card designating a dataset is required for each external file referenced in the "program" parameter list.

3) The maximum amount of code that can be generated for each procedure is calculated by the formula:

codemax = 4 * (7-level)k bytes

Thus, the maximum of code that may be generated for the main program is 24k bytes. The mechanism of implementation of this extended addressing feature involves the use of spare 'display registers'. Some of the registers are also used for optimisation of the 'with' construct, as they can be used to store the base address of records currently being referenced.

## APPENDIX 2. File support

1) Files may be external or local. External files are named in the program header; local files are not.

2) Both external and local files may be declared in a procedure at any level. System control blocks are allocated for a file on the stack when the procedure containing the file declaration is invoked. Buffers are allocated on the stack when the first reset or rewrite to the file is issued. These areas are not however accessed directly from the procedure's display register.

3) External files are referred to by providing a DD statement which uses the filename as the ddname, except that the standard files input and output use the ddnames SYSIN and SYSPRINT respectively. Any DD statement supported by the QSAM access method may be used.

4) If local files are used, a DD statement with a ddname of 'LOCAL' must be provided. This statement should refer to a direct-access volume containing sufficient free space to hold all local files required. It will usually be of the form:

```
//LOCAL DD UNIT=SYSDA,DISP=SHR,VOL=SER=volser
```

   Current implementation restrictions for local files are that no secondary extents may be taken, and all local files have an equal primary space allocation quantity. The primary space allocation is taken from the SPACE parameter on the LOCAL DD statement if it exists, otherwise a default value of 5 tracks is used. If DCB parameters are specified on the LOCAL DD statement, these must be suitable for all local files used. DSNAME must not be specified on the LOCAL DD statement.

5) If a long-jump or external jump is made to an outer procedure in a situation where files have been opened in intermediate procedures, then these files are correctly closed, and if local, are scratched at the time of the long-jump.

6) If premature termination of a program for any reason causes the printing of a 'Pascal Termination Log', then all files which were open are closed, and if local are scratched. If a system abend occurs, the files are not closed and any local files in use will remain allocated on the disk. Note that the time-limit condition is intercepted and local files are scratched. However, operator cancel, open failure or getmain failure, for example, will cause system abends, and may leave local files allocated. The possibility of getmain failure is particularly worthy of consideration; if a large number of local files are opened, enough space must be left outside the Pascal stack to build their system I/O blocks and channel programs. Space required for these is usually of the order of 100 bytes per buffer per file (2 buffers per file default).

7) Implementation of local files is achieved through use of the DASD allocation and SCRATCH functions of OS. DASD allocation is done using SVC32 in the same way as it is used by the IEHMOVE utility program. A unique dataset name is constructed for each used local file as follows: the dsname generated for the LOCAL DD statement by the operating system has the form:

SYSyyddd.Thhmmss.xxnnn.jobname.Rnnnnnnn

where yyddd and hhmmss are the date and time of job initiation, xx is variable depending on the environment, nnn is a serial number, jobname is the name of the job, and nnnnnnn is a serial number. Through its use of variable fields, this name is different from all others in the operating system. Each local file is given a unique name built from this name by replacing the jobname field with Pnnnnnnn where nnnnnnn is a local file serial number. These dsnames still appear as temporary datasets to the operating sysem, and thus if any are inadvertently left allocated on the disk by premature termination of a job, they should in due course be scratched by whatever means is used by an installation to remove old temporary datasets.

8) An external file may be declared in more than one procedure or in a procedure that is invoked recursively. This will not normally be useful, but the effect will be that file positioning will be maintained for the outer procedure during its use by an inner one. If the file is used for output, the results are unpredictable.

9) A distinction is made between files of type "text" and files of other types. Both types are supported with the QSAM access method; text files using get-locate and put-locate, and non-text files using get-move and put-move.

10) For non-text files, support is provided for RECFM=F, FB or FBS. For input, the LRECL or BLKSIZE for RECFM=F must match the record size implied by the file declaration. For output, if RECFM is not specified, it is set to FB; LRECL (or BLKSIZE for RECFM=F ) is set to the size implied by the file declaration. For RECFM=FB or FBS, if BLKSIZE is specified it must be a multiple of LRECL; if not specified it is set to the lowest multiple of LRECL greater than or equal to 2000.

11) For text-files, support is provided for all valid combinations of fixed, variable or undefined RECFM with blocked, standard, spanned or ASA control. Valid combinations are represented by F[B][S][A], V[B][S][A] or U[A]. The logical record length (or blocksize if RECFM is U or not B) may be specified and may not exceed 256. If LRECL is not specified, it is set to 132 + (1 if RECFM=A) + (4 if RECFM=V). If RECFM is not specified, it is set to RECFM=VBA. If RECFM=B and blocksize is not specified, it is set as for non-text files to the lowest multiple of LRECL not less than 2000. An exception to this rule is that for SYSIN, the default LRECL is 80.

12) For output to a text file with RECFM=A, the carriage control character position is accessible to the user, and should normally be filled with a blank. The "page" function will cause the character written in the control position to be overwritten at the time the record is written. If a text field being written overruns the end of the record, the excess is placed on the following record.

13) For input from a text file with RECFM=A, the carriage control character will be read by the user as the first character of each record.

## APPENDIX 3. Linking to External Procedures.

The linkage editor version of the Pascal 8000 system produces object code following the specifications of the IBM linkage editor. Separately compiled Pascal modules can therefore be linked together, and can link to object modules produced by other language compilers, such as the FORTRAN compiler, or to modules produced by the IBM Assembler.

### 1. How to specify that a module is external.

If a procedure or function that has been developed independently of the current compilation is to be invoked, it must first be declared by the standard procedure/function heading, followed by the word "pascal" or "extern" (if the external routine is written in Pascal) or "fortran" (if the routine is written in FORTRAN or follows the FORTRAN argument passing conventions).

Examples:

```
procedure plotxy(var x,y:real); fortran;
function rand(x:integer) : real; fortran;
```

Once an external routine has been declared in such a manner, it may be used in the same way as ordinary Pascal procedures and functions, with some restrictions detailed below.

It should be noted that the standard IBM-defined parameter passing mechanism is used when invoking FORTRAN or Assembler routines (detailed in the IBM FORTRAN Programmers Guide). The external reference generated is to a CSECT with the name of the procedure truncated (or padded) to eight characters. Thus, in the above example, the linkage editor would expect to find external modules with the names "rand" and "plotxy", and will resolve all references to them.

### 2. Externally compiled Pascal procedures and functions.

There are no restrictions to the form that an externally compiled Pascal procedure or function may take. The argument lists and function results are treated in exactly the same way as with "internally" compiled routines, and all scope rules apply as regards the use of the procedure name.

There are, however, certain rules that must be followed in order to actually compile an external module. These are:

1. The (*$E+*) option must occur before a procedure declaration. This signifies to the compiler that the first procedure following is to be compiled as an external module, and that there will be no further procedures or "main-body" after this procedure definition.

2. "value" statements are obviously not permitted. However, global type, constant and variable declarations are allowed.

3. Only one "level one" procedure declaration and body is allowed per compilation i.e. that of the procedure which is to be regarded as external. However, the external procedure may have many levels on nested internal procedures within itself, and the usual syntax rules apply to these.

4. No "main program", i.e. a dummy begin ... end. , is to be specified. This would serve little purpose.

It should also be noted that an externally compiled procedure may have references to further externally compiled routines within it; no restrictions are imposed in this regard. Further, the externally compiled routine may be invoked recursively from within itself and from within any nested procedure declared within itself. The external procedure may also be passed as an argument to any nested procedure, with the usual scope rules applying.

Example.

```
 (*$E+*)
 program external(output);
   procedure incr (n:integer);

     procedure addto5(procedure g(integer));
     begin
       g(n+1)
     end;

   begin
    if n <> 5 then addto5(incr) else writeln(n);
   end;
```

This externally compiled procedure may be invoked by the program:

```
program callit;
  procedure incr(n:integer); pascal;
begin
   incr(1)
end.
```

Externally compiled procedures may access the global variables of the calling program, as long as these are declared in exactly the same order in both the external and calling routines. This feature is useful when several separately compiled routines wish to access a common set of variables without passing all of them to

each routine as arguments.

Note that once the (*$E+*) option has been set, it may not be reset, and such attempts will be trapped by the compiler. Further, procedures and functions cannot be passed as arguments to externally compiled routines.

## 3. Externally compiled FORTRAN and Assembler routines.

The linkage-editor version of Pascal 8000 permits linking between Pascal programs and any routine that comforms to the IBM FORTRAN calling sequence and argument passing conventions. Such routines can behave as procedures or functions, with function return types of real, integer and boolean being permissible.

When using arguments passed to external routines, one must be aware of the internal representation of the standard types. This is given in the following table.

| type | bytes | comment |
|---|---|---|
| integer | 4 | |
| boolean | 4 | 1=true,0=false (same as IBM FORTRAN) |
| char | 4 | EBCDIC code in low order byte |
| real | 8 | IBM floating point representation |
| set | 8 | bits representing elements, starting from left |
| scalar | 4 | ordinal value of occurrence, starting at 0 |

For example, if a FORTRAN routine is called with a real argument, the argument should be declared as DOUBLE PRECISION in the called routine. "Packed" structures are represented by single bytes when the value of the packed item is in the range zero to 255.

Examples.

```
    x : integer;                        (* 4 bytes *)
    a : packed array(.1..4.)
        of char                         (* 4 bytes *)
    b : array(.1..4.) of char           (* 16 bytes *)
    c : real                            (* 8 bytes *)
```

# APPENDIX 4. How to Run the System.

## 1. The compile-and-go system.

The load module PASCAL, from file 2 of the tape, works as the Pascal system driver. It has no specific knowledge of the Pascal compiler; it merely provides a running environment for programs compiled by the compiler - the compiler itself being one such program. This program requires a SYSPRINT dataset for its standard output. This dataset should have a usable print width of 133 characters, including carriage control, and it may have a a RECFM of UA, VA, VBA, FA, or FBA. If no DCB parameters are specified, the default of RECFM=VA, BLKSIZE=141, LRECL=137 is used. If local files are to be used a LOCAL dd statement is also required. This should have the form:

```
//LOCAL     DD  UNIT=3330,DISP=SHR,VOL=SER=volser
```

If it is undesirable to mention a specific volume on this card, it could be replaced by:

```
//$LOCAL    DD  UNIT=SYSDA,SPACE=(TRK,0)
//LOCAL     DD  VOL=REF=*.$LOCAL,DISP=SHR
```

Other datasets required are determined by the contents of the PARM field passed to the program. The complete specifications of the PARM field, which may not contain any blanks, are as follows:

```
PARM=[n1] [,n2] [,] step1[=s1] [,step2[=s2]]

                    [,step3[=s3]] .....etc
```

where the square brackets denote optional parameters. The stepnames step1, step2 etc. are symbolic names given to Pascal programs which are to be run under the Pascal system. Each must consist of 1 to 7 characters, and must otherwise obey the rules for constructing ddnames. Three ddnames are associated with each stepname - by appending the symbols 1, 2 and I respectively to the stepname. The Pascal program loader accesses the first two of these, and they must describe the two datasets produced by the compiler when it compiled the program which is to be run. The first of these must have the attributes DCB=(RECFM=F,BLKSIZE=1024) (it may not have RECFM=FB), and the second is a text-file and usually has the attributes DCB=(RECFM=FB,LRECL=8) with any suitable blocksize. The third ddname describes a textfile containing the standard input file for the compiled program, and must be present if 'INPUT' is specified in the program header of the compiled program. If a Pascal program performs a reset or rewrite to any external file name, then a ddname the same as this file name must also be provided to refer to the external file.

For example, the Pascal compiler itself is a Pascal program, and if referenced by the stepname "CMP", needs the following ddnames:

| | |
|---|---|
| CMP1,CMP2 | describe the object code |
| CMPI | describes standard input to the compiler and should be the textfile containing the source program whose compilation is desired. (Text files may have a record format of F, FB, V, VB, or U and a logical record length up to 256). |
| $PASOBJ1, $PASOBJ2 | These are external file names known to the compiler, and on which it produces the object code for the source program given on CMPI. They may describe temporary datasets if a compile and go run is desired; they may describe permanent datasets if the compiled program is to be saved. |
| $PASMSGS | This external file name is a textfile on which the compiler expects to find error message texts if it needs them. |

Thus if a following step is given the name "$PASOBJ", it will load and execute the program just compiled. Standard input, if any, for this program will be accessed via the ddname "$PASOBJI".

The optional parameters s1, s2 etc after each stepname are used to control the main storage usage for each step. Each must be a decimal integer, and describes the amount of storage (in multiples of 2K bytes) which is to be left free during the running of the step. This storage is required for system functions such as OPEN, CLOSE, ABEND etc. The remainder of the storage available within the user's region is allocated for the compiled code, the stack and the heap. If this parameter is not specified, it defaults to a value of 4 (implying 8K), and this has been found to be adequate for most purposes at the authors' installation. It may need to be increased if more temporary storage is needed for any one system function, or if heavy demands on permanent storage are made by the Pascal program - e.g. by opening a large number of local or external files (although file buffers are constructed on the stack, control blocks and channel programs associated with each buffer are constructed from the free storage). The control of main storage in this way means that for most applications the combined stack and heap size is specified by a single parameter - the REGION parameter on the EXEC statement.

The optional parameters n1 and n2 in the PARM field are decimal integers, and control the printing of the post-mortem dump if any. The post-mortem dump routine, if invoked, prints the environment of the innermost n1 and the outermost n2 segments. If n1 and/or n2 is not specified, each defaults to a value of 5.

If any of the defaults for the parameters n1, n2, or for the free-storage parameter s is required to be changed, the change may be effected by changing their values in the member PASDATA in the runtime system source library, reassembling the module PASCAL in this library,

and relinking the load module PASCAL or PASCALO.

2. The linkage-editor version.

The load module PASCALC, from file 2 of the tape, contains both the runtime system and the compiled code of the compiler. This compiler and runtime system is functionally the same as the compile-and-go version, except that the load module is structured differently, and the format of compiled code follows the IBM object-module conventions rather than appearing as the two datasets produced by the compile-and-go version.

Datasets required by this compiler are not determined by the parm field, and are as follows:

SYSPRINT a print output dataset, the same as for the compile-and-go version, normally SYSOUT=A.

SYSGO describes the dataset which will contain the object module produced by the compiler. This should have DCB attributes of RECFM=F or FB, and have LRECL=80, with any suitable blocksize.

$PASMSGS the text file on which error message texts are provided and used by the compiler in the same manner as for the compile-and-go version.

SYSIN the standard input to the compiler, containing the source program to be compiled.

When a compiled program has been linkage-edited, it will need the following file-names:

SYSPRINT describes its standard output.

SYSIN describes its standard input.

LOCAL defines a device which may be used for local file allocation, as for the compile-and-go version.

Any external files declared and used by the program will require matching ddnames.

2.1 Paramters.

The parm field may be used to control the depth of traceback to be printed in a post-mortem dump, as well as main storage usage. The specification is as follows:

PARM = [n1] [, [n2] [, [S]]]

where each of the parameters n1, n2 and s is a decimal integer.

The parameters n1 and n2 control the post-mortem dump traceback printout, if any, in the same way as for the compile-and-go version. The parameter s controls the main storage usage for the jobstep in the same way as the parameters s1, s2, etc do for the compile-and-go version. If an external routine requires a significant amount of main storage, this parameter may need to be specified at a value greater than its default of 4 (implying 8K ). Examples of routines which may require extra main storage are FORTRAN or Assembler programs which perform I/O (storage needed for buffers), or Assembler programs which use the GETMAIN macro directly or indirectly.

When the runtime system of the linkage-edit version initialises itself, it checks for the presence of the FORTRAN runtime system (using a weak reference to the external name IBCOM#). (The FORTRAN runtime system will be present if an external FORTRAN routine uses any FORTRAN input/output.) If present, the runtime system initialises the FORTRAN system as well, to enable the FORTRAN I/O. However, program interrupts will be trapped by the Pascal system, and a Pascal traceback given, even if the interrupt occurs in the FORTRAN code.

FORTRAN initialisation is performed using the following sequence of instructions:

```
            L       R15,=V(IBCOM#)
            BAL     R14,64(0,R15)
```

This works for the G and H versions of FORTRAN under OS Release 21.8; it has not been tested in other environments.

## APPENDIX 5. Operating system dependence

"Local" file usage.

To use local files under the MVS operating system, it is necessary to linkage edit the runtime system as an "authorised program". This is because DASD space allocation is done using SVC 32, which under MVS is a restricted SVC. When used in the foreground under TSO, local files work only if both the runtime system and the terminal monitor program are authorised. Local files work under all other systems without any changes required.

Time limit trap

When the runtime system initialises itself, a trap is set for about 10 seconds before job-step CPU time expiry. When a Pascal program (e.g. in a loop) springs this trap, a traceback with local variable dump is provided. Currently this time-limit feature does not work under SVS or MVS, and the job proceeds to a S322 abend. This will be fixed when the method of accessing the time remaining in the current job step in these systems is learnt. IBM Australia cannot help here - can anyone on the distribution list?

## REFERENCES

Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A.R. (1972) ¿ Structured Programming. Academic Press.

Habermann, A.N. (1973) ¿ Critical comments on the programming language Pascal. Acta Informatica, 3, 47-57.

Hikita, T and Ishihata, K (1976) ¿ An Extended PASCAL and its Implementation Using a Trunk. Report of the Computer Centre University of Tokyo, Vol 5, 23-51.

Hoare, C.A.R. and Wirth, N. (1973) ¿ An axiomatic definition of the programming language Pascal. Acta Informatica, 2, 335-355.

Ishihata, K. and Hikita, T. (1976) ¿ Bootstrapping Pascal using a trunk. Department of Information Science, Faculty of Science, University of Tokyo.

Jensen, K. and Wirth, N. (1974; Second Edition, 1975) ¿ Pascal: User Manual and Report. Springer.

Kludgeamus, S. (1976) ¿ Sydney University, private communication.

Lecarme, O. and Desjardins, P. (1975) ¿ More comments on the programming language Pascal. Acta Informatica, 4, 231-243.

Wirth, N. (1971) ¿ The programming language Pascal. Acta Informatica, 1, 35-63.

Wirth, N. (1971) ¿ The design of a Pascal compiler. Software ¿ Practice and Experience, 1, 309-333.

Wirth, N. (1975) ¿ An assessment of the programming language Pascal. SIGPLAN Notices, 10, 6, 23-30.