HISOFT PASCAL

Programmer's Manual

Version 4T

VERSION 4T

© HISOFT 1982

HISOFT PASCAL 4T IMPLEMENTATION NOTE
48K ZX SPECTRUM

Loading HP4 From Tape.

Unpack the cassette tape from its case and load it into your cassette recorder with Side A (the side with the label) uppermost. On your SPECTRUM make sure that you are in Keyword Entry Mode and then enter:

LOAD "" (press K and then " twice)

Now press PLAY on the tape recorder: first the HP4S loader will be loaded, this will execute automatically and proceed to load the HP4S code. If a tape error is detected then press the SPACE bar on the SPECTRUM, stop the tape, rewind to the start, press NEW (on the 'A' key) on the SPECTRUM and then enter LOAD "" again. If you still get a tape loading error then try adjusting the volume on your tape recorder; if errors persist please return the tape to Hisoft and we will replace it.

Once the HP4S code has been loaded it will execute automatically and the message 'Top of RAM?' will be displayed - now consult Section 0.0 of the Hisoft Pascal 4T Programmer's Manual for details of how to proceed.

Implementation on the SPECTRUM.

The ZX SPECTRUM is a rather unusual computer and, to a certain extent, the implementation of HP4T reflects this. The various control codes discussed in the Programmer's Manual are reached as follows on the SPECTRUM:

RETURN	via the 'ENTER' key. KONEC VRSTICE
CC	via CAPS SHIFT and 1. NAZAJ V EDITOR
CH	DELETE i.e. CAPS SHIFT and 0. IZBRIŠE ZADNJI NATIPKANI ZNAK.
CI	via CAPS SHIFT and 8. PREMAKNE NA NASLEDNJO TAB POZICIJO
CP	not available yet.
CX	via CAPS SHIFT and 5 IZBRIŠE PRAVKAR NATIPKANO VRSTICO
CS	via CAPS SHIFT and SPACE. IZ

The ZX SPECTRUM keyword entry scheme is not supported (we see this as a positive advantage), instead all text must be inserted using the normal alphanumeric keys. Using SYMBOL SHIFT and any key (except I) will always reach the ASCII symbol associated with that key and not the keyword e.g. SYMBOL SHIFT T gives '>' and SYMBOL SHIFT 0 gives '>'. You must not use the single symbols <=, <> and >= ; instead these should be entered as a combination of the symbols <, > and =.

The editor comes up in upper case mode, this may be toggled in the normal way using CAPS SHIFT and 2.

You have control over the temporary attributes of the various character positions on the screen through the use of the standard control codes (e.g. WRITE(CHR(17),CHR(4)) will make the 'paper' green) but you cannot change the permanent attributes. If, while using these control codes, an invalid sequence is detected then the message 'System Call error' will be displayed and the execution aborted. When you dump out text or object code to tape the message 'Start tape, then press any key' will be displayed twice - you must respond to it each time.

There is no need to save the loader since an automatic loader is always dumped with the object program - if you have used the 'T'ranslate command to save the object code and runtimes on tape then to load and run the program simply enter 'LOAD ""' from within BASIC. After the execution of the object code has finished you can run it again, assuming it has not corrupted anything, by entering 'GOTO 3' from within BASIC.

If you use the 'B' command from within the HP4T editor to return to the ZX SPECTRUM BASIC then, assuming you do not change the BASIC program, you can re-enter the HP4T editor in one of two ways: enter 'GOTO 9' to perform a warm start i.e. preserving the Pascal program or 'GOTO 12' to do a cold start, re-initialising the Pascal and clearing any existing Pascal text.

Please do not hesitate to contact us if you experience any difficulty with Hisoft Pascal 4 - we can only solve the problems if we know what they are!

Note: The start address of the package is £6016 (hexaducimal) = 24598.

CONTENTS

SECTION 0 PRELIMINARIES.

0.0 Getting Started.

Hisoft Pascal 4T (HP4T) is a fast, easy-to-use and powerful version of the Pascal language as specified in the Pascal User Manual and Report (Jensen/Wirth Second Edition). Omissions from this specification are as follows:

FILEs are not implemented although variables may be stored on tape.
A RECORD type may not have a VARIANT part.
PROCEDUREs and FUNCTIONs are not valid as parameters.

Many extra functions and procedures are included to reflect the changing environment in which compilers are used; among these are POKE, PEEK, TIN, TOUT and ADDR.

The compiler occupies approximately 12K of storage while the runtimes take up roughly 4K. Both are supplied on cassette tape in the tape format used by the runtimes. All interfacing between HP4T and the host machine takes place through vectors conveniently placed at the start of the runtimes (see HP4T Alteration Guide) - this makes it easy for the user to write his own customised I/O routines if necessary.

Hisoft Pascal 4T uses various control codes, mostly within the editor. Of course, different systems can have very different keyboard designs and thus wil have different ways of reaching control codes. In this manual the control characters used will be referred to as RETURN, CC, CH, CI, CP, CS and CX. The attached Implementation Note will tell you the corresponding keys for your system.

Whenever HP4T is waiting for a line of input, the control characters can be used as follows:

RETURN	is used to terminate the line.
CC	returns to the editor.
CH	deletes the last character typed.
CI	move to the next TAB position.
CP	directs output to the printer (if available) or if output was going to the printer then it returns to the screen.
CX	deletes the whole line typed so far.

A simple loader is also supplied in the package so that the user can load, from tape, data which has been recorded in HP4T format.

Thus, to load the compiler and runtimes from the master tape supplied by Hisoft, the user must first load the loader - when applicable this is supplied in a form suitable for loading by the user's operating system. If the user is unable to access the computer's operating system then the boot loader must be entered into the computer's memory directly either through the use of an assembler or a high level language such as BASIC - details of how to do this and a skeletal loader appear in the HP4T Installation Guide.

Once the loader has been executed it will proceed to search for a file recorded in HP4T tape format. When a file of this format has been found the loader will load the file into memory. If at any stage an error is detected while reading the tape a message will be displayed - you must then rewind the tape to the beginning of the file and attempt to load it again. If you experience repeated errors then adjust the volume level on your tape recorder - if this is not successful then please return the tape to Hisoft and we will send you a replacement tape.

Thus the loader will automatically load the compiler and runtimes into memory for you.

When the compiler has been successfully loaded it will execute automatically and produce the message:

Top of RAM?

You should respond to this by either entering a positive decimal number up to 65536 (followed by RETURN) or by hitting RETURN (See Implementation Note).

If you enter a number then this is taken to represent the highest RAM location + 1 otherwise the first non-RAM location is automatically computed. The compiler's stack is set to this value and thus you can reserve high memory locations (perhaps for extensions to the compiler) by deliberately giving a value less than the true top of RAM. In the ZX Spectrum version the true top of RAM is taken to be the start of the user-defined graphics area (UDG in the Sinclair manual).

You will then be prompted with:

Top of RAM for 'T'

Here you can enter a decimal number or default to the 'Top of RAM' value previously specified. What you enter will be taken as the stack when the resultant object code is executed after using the editor 'T' command (See Section 4 for details). You will need to define a runtime stack different from the top of RAM if, for example, you have written extensions to the runtimes and wish to place them safely in high memory locations.

Finally you will be asked:

Table size?

What you enter here specifies the amount of memory to be allocated to the compiler's symbol table.

Again, either enter a positive decimal number followed by RETURN or simply RETURN by itself in which case a default value of (available RAM divided by 16) will be taken as the symbol table size. In nearly all cases the default value provides more than enough space for symbols. The symbol table may not extend above machine address £8000* (32768 decimal). If you specify so large a value that that this happens then you will be prompted again for 'Top of RAM' etc.

You may, optionally, include an 'E' before the number after this prompt - if you do so then the internal line editor will not be retained for use by the compiler. So do this if you wish to use your own editor with the compiler (see the HP4T Installation Guide for details on how to do this).

At this point the compiler and integral editor (if retained) will be relocated at the end of the symbol table and execution transferred to the supported editor.

*Note: throughout this Manual, the pound sign "£" is replaced by the number sign or hash (decimal 35, hexadecimal 23, shift '3') on all systems which do not use U.K. ASCII. Numbees that are preceded by this symbol are in hexadecimal.

0.1 Scope of this manual.

This manual is not intended to teach you Pascal; you are referred to the excellent books given in the Bibliography if you are a newcomer to programming in Pascal.

This manual is a reference document, detailing the particular features of Hisoft Pascal 4.

Section 1 gives the syntax and the semantics expected by the compiler.

Section 2 details the various predefined identifiers that are available within Hisoft Pascal 4, from CONSTants to FUNCTIONs.

Section 3 contains information on the various compiler options available and also on the format of comments.

Section 4 shows how to use the line editor which is an integral part of HP4T; if you do not wish to use this editor but want to interface your own editor, then you should consult the HP4T Alteration Guide.

The above Sections should be read carefully by all users.

Appendix 1 details the error messages generated both by the compiler and the runtimes.

Appendix 2 lists the predefined identifiers and reserved words.

Appendix 3 gives details on the internal representation of data within Hisoft Pascal 4 - useful for programmers who wish to get their hands dirty.

Appendix 4 gives some example Pascal programs - study this if you experience any problems in writing Hisoft Pascal 4 programs.

0.2 Compiling and Running.

For details of how to create, amend, compile and run an HP4T program using the integral line editor see Section 4 of this manual. For information on what to do if you are using your own editor see the HP4T Alteration Guide.

Once it has been invoked the compiler generates a listing of the form:

```
xxxx nnnn text of source line
```

where: xxxx is the address where the code generated by this line begins.
          nnnn is the line number with leading zeroes suppressed.

If a line contains more than 80 characters then the compiler inserts new-line characters so that the length of a line is never more than 80 characters.

The listing may be directed to a printer, if required, by the use of option P if supported (see Section 3).

You may pause the listing at any stage by pressing CS; subsequently use CC to return to the editor or any other key to restart the listing.

If an error is detected during the compilation then the message *ERROR* will be displayed, followed by an up-arrow ('^'), which points after the symbol which generated the error, and an error number (see Appendix 1). The listing will pause; hit 'E' to return to the editor to edit the line displayed, 'P' to return to the editor and edit the previous line (if it exists), or any other key to continue the compilation.

If the program terminates incorrectly (e.g. without 'END.') then the message 'No more text' will be displayed and control returned to the editor.

If the compiler runs out of table space then the message 'No Table Space' will be displayed and control returned to the editor. Normally the programmer will then save the program on tape, re-load the compiler and specify a larger 'Table size' (see Section 0.0).

If the compilation terminates correctly but contained errors then the number of errors detected will be displayed and the object code deleted. If the compilation is successful then the message 'Run?' will be displayed; if you desire an immediate run of the program then respond with 'Y', otherwise control will be returned to the editor.

During a run of the object code various runtime error messages may be generated (see Appendix 1). You may suspend a run by using CS; subsequently use CC to abort the run or any other key to resume the run.

0.3 Strong TYPEing.

Different languages have different ways of ensuring that the user does not use an element of data in a manner which is inconsistent with its definition.

At one end of the scale there is machine code where no checks whatever are made on the TYPE of variable being referenced. Next we have a language like the Byte 'Tiny Pascal' in which character, integer and Boolean data may be freely mixed without generating errors. Further up the scale comes BASIC which distinguishes between numbers and strings and, sometimes, between integers and reals (perhaps using the '%' sign to denote integers). Then comes Pascal which goes as far as allowing distinct user-enumerated types. At the top of the scale (at present) is a language like ADA in which one can define different, incompatible numeric types.

There are basically two approaches used by Pascal implementations to strength of typing; structural equivalence or name equivalence. Hisoft Pascal 4 uses name equivalence for RECORDs and ARRAYs. The consequences of this are clarified in Section 1 - let it suffice to give an example here; say two variables are defined as follows:

```
VAR A : ARRAY['A'..'C'] OF INTEGER;
    B : ARRAY['A'..'C'] OF INTEGER;
```

then one might be tempted to think that one could write A:=B; but this would generate an error (*ERROR* 10) under Hisoft Pascal 4 since two separate 'TYPE records' have been created by the above definitions. In other words, the user has not taken the decision that A and B should represent the same type of data. She/He could do this by:

```
VAR A,B : ARRAY['A'..'C'] OF INTEGER;
```

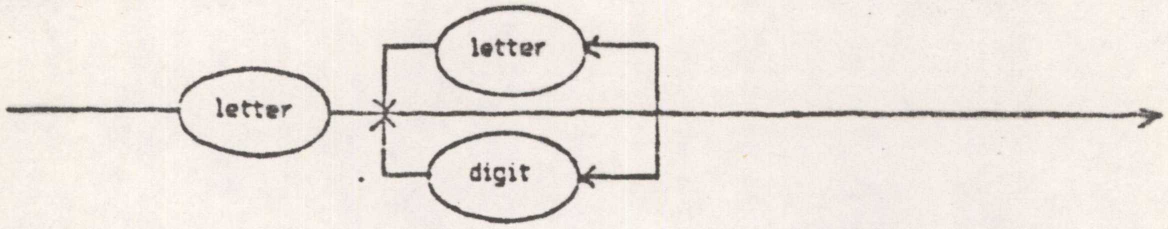
and now the user can freely assign A to B and vice versa since only one 'TYPE record' has been created.

Although on the surface this name equivalence approach may seem a little complicated, in general it leads to fewer programming errors since it requires more initial thought from the programmer.

SECTION 1 SYNTAX AND SEMANTICS.

This section details the syntax and the semantics of Hisoft Pascal 4 - unless otherwise stated the implementation is as specified in the Pascal User Manual and Report Second Edition (Jensen/Wirth).

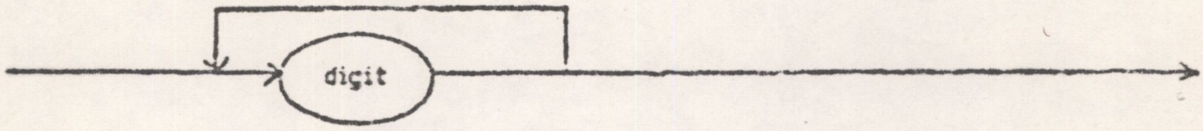
1.1 IDENTIFIER.



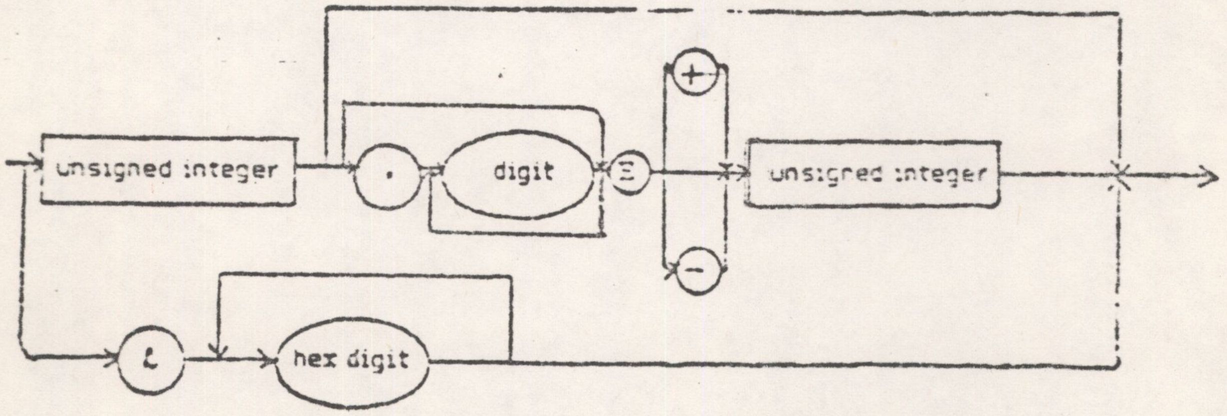
Only the first 10 characters of an identifier are treated as significant.

Identifiers may contain lower or upper case letters. Lower case is not converted to upper case so that the identifiers HELLO, HELlo and hello are all different. Reserved words and predefined identifiers may only be entered in upper case.

1.2 UNSIGNED INTEGER.



1.3 UNSIGNED NUMBER.



Integers have an absolute value less than or equal to 32767 in Hisoft Pascal 4. Larger whole numbers are treated as reals.

The mantissa of reals is 23 bits in length. The accuracy attained using reals is therefore about 7 significant figures. Note that accuracy is lost if the result of a calculation is much less than the absolute values of its arguments e.g. 2.00002 - 2 does not yield 0.00002. This is due to the inaccuracy involved in representing decimal fractions as binary fractions. It does not occur when integers of moderate size are represented as reals e.g. 200002 - 200000 = 2 exactly.

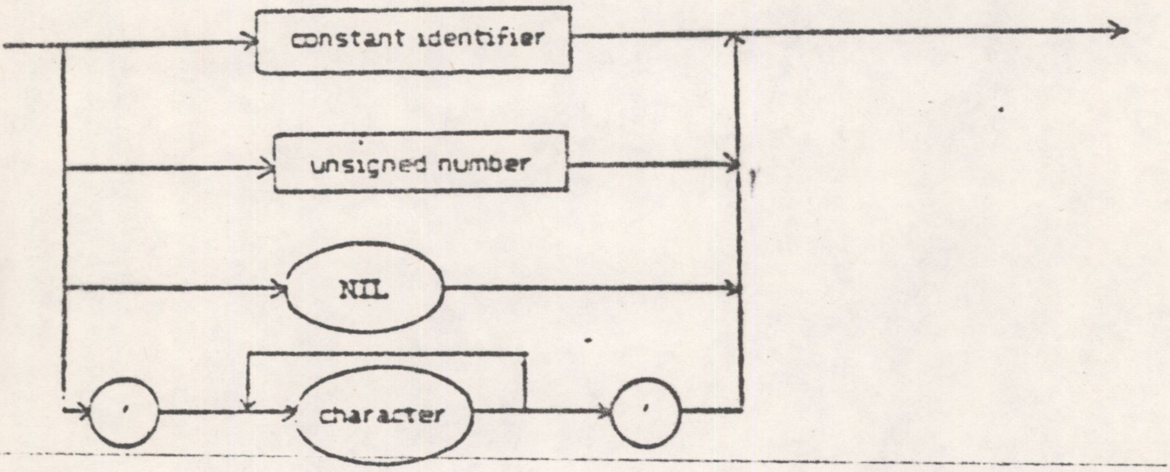
The largest real available is 3.4E38 while the smallest is 5.9E-39.

There is no point in using more than 7 digits in the mantissa when specifying reals since extra digits are ignored except for their place value.

When accuracy is important avoid leading zeroes since these count as one of the digits. Thus 0.000123456 is represented less accurately than 1.23456E-4.

Hexadecimal numbers are available for programmers to specify memory addresses for assembly language linkage inter alia. Note that there must be at least one hexadecimal digit present after the '&', otherwise an error (*ERROR* 51) will be generated.

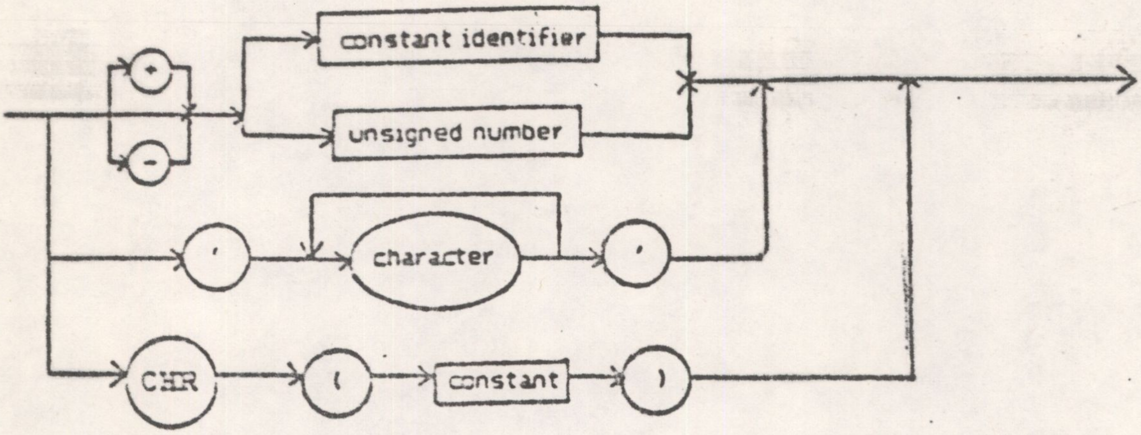
1.4 UNSIGNED CONSTANT.



Note that strings may not contain more than 255 characters. String types are ARRAY [1..N] OF CHAR where N is an integer between 1 and 255 inclusive. Literal strings should not contain end-of-line characters (CHR(13)) - if they do then an '*ERROR* 68' is generated.

The characters available are the full expanded set of ASCII values with 256 elements. To maintain compatibility with Standard Pascal the null character is <u>not</u> represented as '' ; instead CHR(0) should be used.

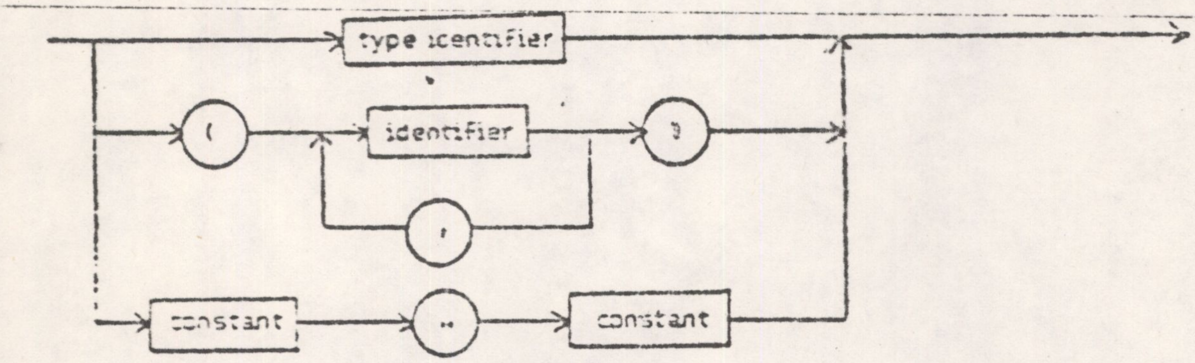
1.5 CONSTANT.



The non-standard CHR construct is provided here so that constants may be used for control characters. In this case the constant in parentheses must be of type integer.

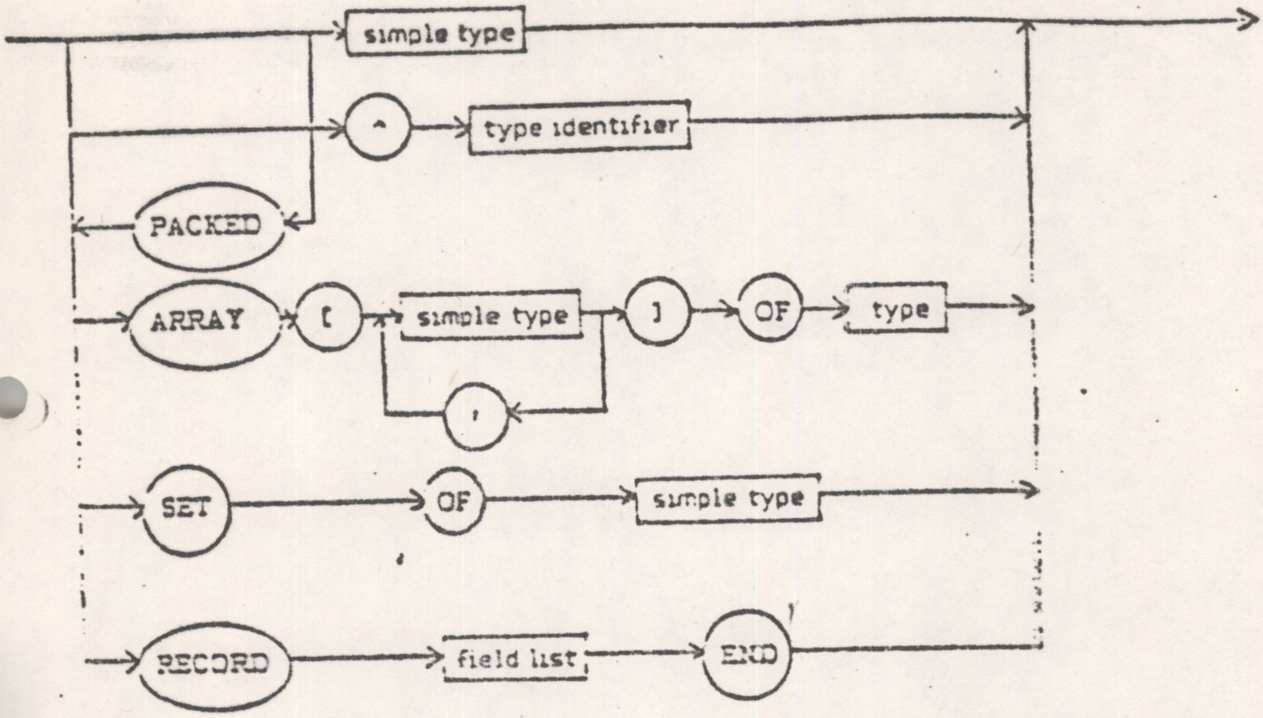
```
E.g.  CONST bs=CHR(10);
            cr=CHR(13);
```

1.6 SIMPLE TYPE.



Scalar enumerated types (identifier, identifier, ...) may not have more than 256 elements.

1.7 TYPE.



The reserved word PACKED is accepted but ignored since packing already takes place for arrays of characters etc. The only case in which the packing of arrays would be advantageous is with an array of Booleans - but this is more naturally expressed as a set when packing is required.

1.7.1 ARRAYs and SETs.

The base type of a set may have up to 256 elements. This enables SETs of CHAR to be declared together with SETs of any user enumerated type. Note, however, that only subranges of integers can be used as base types. All subsets of integers are treated as sets of 0..255.

Full arrays of arrays, arrays of sets, records of sets etc. are supported.

Two ARRAY types are only treated as equivalent if their definition stems from the same use of the reserved word ARRAY. Thus the following types are not equivalent:

```
TYPE
        tablea = ARRAY[1..100] OF INTEGER;
        tableb = ARRAY[1..100] OF INTEGER;
```

So a variable of type tablea may not be assigned to a variable of type tableb. This enables mistakes to be detected such as assigning two tables representing different data. The above restriction does not hold for the special case of arrays of a string type, since arrays of this type are always used to represent similar data.

1.7.2 Pointers.

Hisoft Pascal 4 allows the creation of dynamic variables through the use of the Standard Procedure NEW (see Section 2). A dynamic variable, unlike a static variable which has memory space allocated for it throughout the block in which it is declared, cannot be referenced directly through an identifier since it does not have an identifier; instead a pointer variable is used. This pointer variable, which is a static variable, contains the address of the dynamic variable and the dynamic variable itself is accessed by including a '^' after the pointer variable. Examples of the use of pointer types can be studied in Appendix 4.

There are some restrictions on the use of pointers within Hisoft Pascal 4. These are as follows:

Pointers to types that have not been declared are not allowed. This does not prevent the construction of linked list structures since type definitions may contain pointers to themselves e.g.

```
TYPE
    item = RECORD
              value : INTEGER;
              next  : ^item
             END;

     link = ^item;
```

Pointers to pointers are <u>not</u> allowed.

Pointers to the same type are regarded as equivalent e.g.

```
VAR
    first : link;
    current : ^item;
```

The variables first and current are equivalent (i.e. structural equivalence is used) and may be assigned to each other or compared.

The predefined constant NIL is supported and when this is assigned to a pointer variable then the pointer variable is deemed to contain no address.

1.7.4 RECORDs.

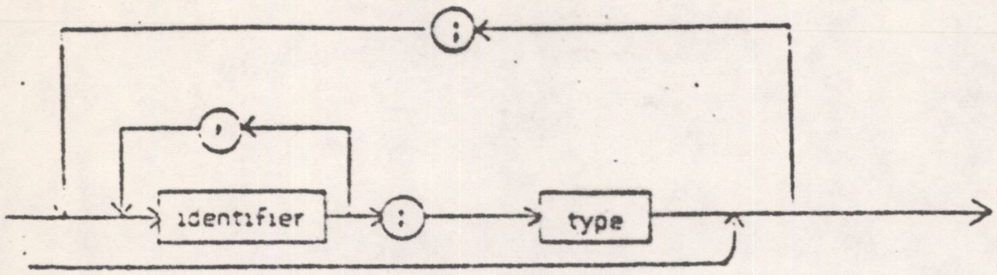
The implementation of RECORDs, structured variables composed of a fixed number of constituents called fields, within Hisoft Pascal 4 is as Standard Pascal except that the variant part of the field list is <u>not</u> supported.

Two RECORD types are only treated as equivalent if their declaration stems from the same occurrence of the reserved word RECORD see Section 1.7.1 above.

The WITH statement may be used to access the different fields within a record in a more compact form.

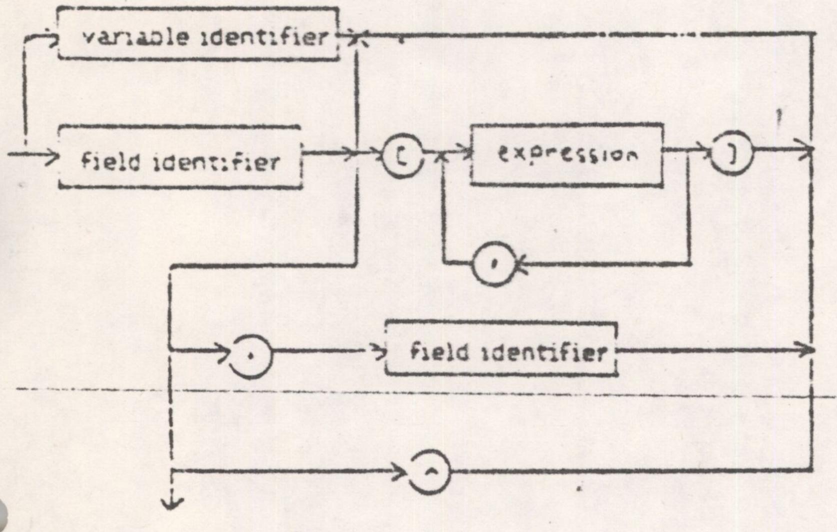
See Appendix 4 for an example of the use of WITH and RECORDs in general.

1.8 FIELD LIST.



Used in conjunction with RECORDs see Section 1.7.4 above and Appendix 4 for an example.

VARIABLE.



Two kinds of variables are supported within Hisoft Pascal 4; static and dynamic variables. Static variables are explicitly declared through VAR and memory is allocated for them during the entire execution of the block in which they were declared.

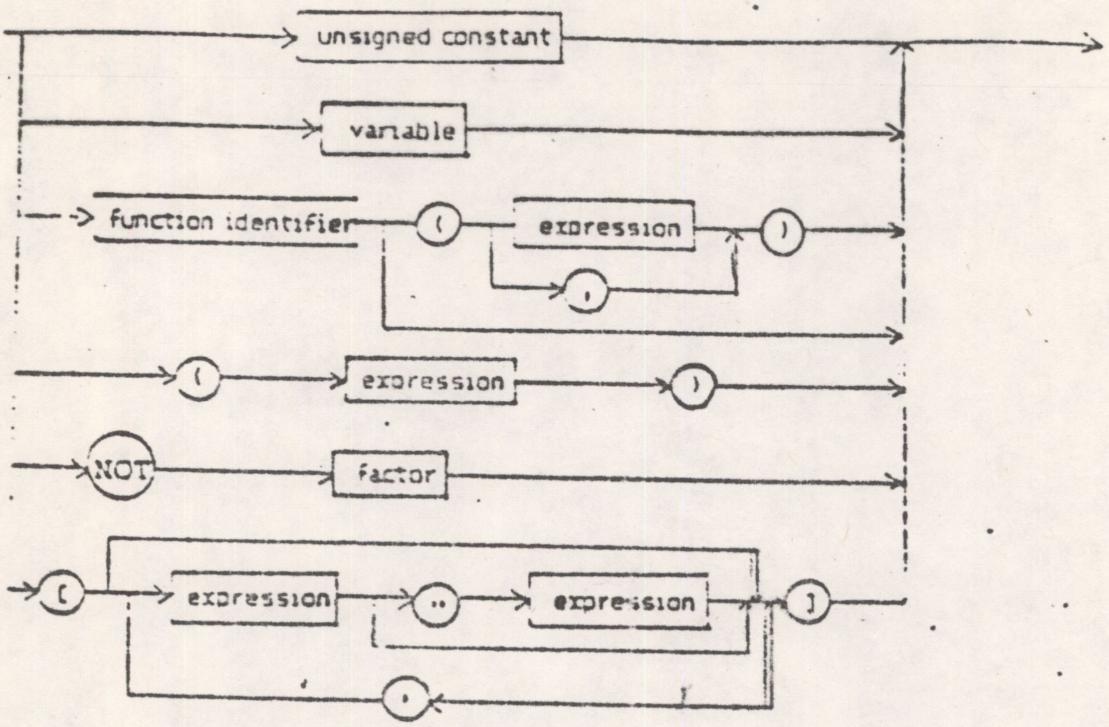
Dynamic variables, however, are created dynamically during program execution by the procedure NEW. They are not declared explicitly and cannot be referenced by an identifier. They are referenced indirectly by a static variable of type pointer, which contains the address of the dynamic variable.

See Section 1.7.2 and Section 2 for more details of the use of dynamic variables and Appendix 4 for an example.

When specifying elements of multi-dimensional arrays the programmer is <u>not</u> forced to use the same form of index specification in the reference as was used in the declaration - this is a change from Hisoft Pascal 3.

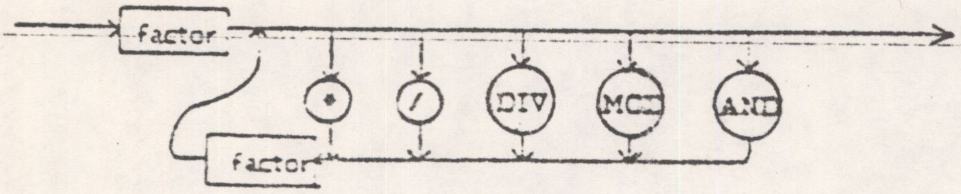
e.g. if variable a is declared as ARRAY[1..10] OF ARRAY[1..10] OF INTEGER then either a[1][1] or a[1,1] may be used to access element (1,1) of the array.

FACTOR.



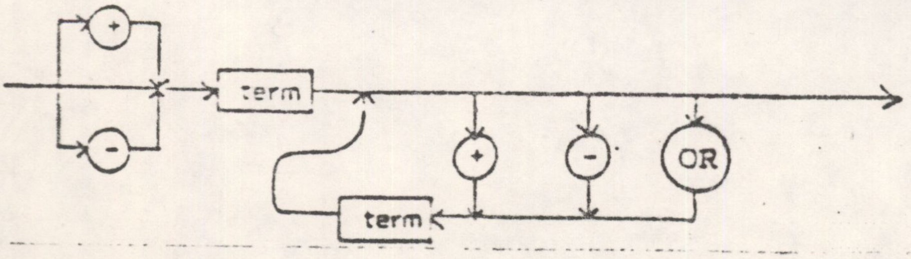
See EXPRESSION in Section 1.13 and FUNCTIONs in Section 3 for more details.

1.11 TERM.



The lowerbound of a set is always zero and the set size is always the maximum of the base type of the set. Thus a SET OF CHAR always occupies 32 bytes (a possible 256 elements - one bit for each element). Similarly a SET OF 0..10 is equivalent to SET OF 0..255.

1.12 SIMPLE EXPRESSION.



H. Javornik

HISOFT
60 Hallam Moor
Liden SWINDON
SN3 6LS

Tel. (0793) 26616

Date:

Your ref:
Our ref:

HISOFT PASCAL 4 - VERSION 1·4.

The release number of Hisoft Pascal 4D and Hisoft Pascal 4T is now 1·4, effective from 31 October 1982.

The differences between Version 1·1 and Version 1·4 are given below:

1. A bug in the evaluation of expressions (which led to expressions like 1+SQR(2) being incorrectly evaluated) has been corrected.
2. A bug in the evaluation of expressions (which led to expressions like I+(I+1) being incorrectly evaluated) has been corrected.
3. A bug that caused the incorrect evaluation of the result of an integer to real comparison has been corrected.
4. Hisoft Pascal 4T only: the documented bug in the editor 'S' sub-command (see page 38 of the Programmer's Manual) has been corrected - 'S' can now be used at any time.
5. Hisoft Pascal 4T only: using the 'F' command from within the editor to find a character string now positions the editing cursor at the beginning of any found occurrence of the string cf. pages 37 and 38 of the Programmer's Manual.
6. Hisoft Pascal 4T only: a new editor command has been incorporated. The 'X' command displays, in hexadecimal, the current end address of the compiler. This allows the user to make a working copy of the compiled package and thus minimises the danger of corrupting the master tape. To do this, find the start address of the package from your Implementation Note and the end address of the package using the 'X' command. Now use the relevant operating system command to dump this block of memory to tape. Note that you will not be able to use the HP4T loader to load the package when saved in this way - instead you should use the relevant operating system load command and then execute a cold start into the editor as given in your Implementation Note.

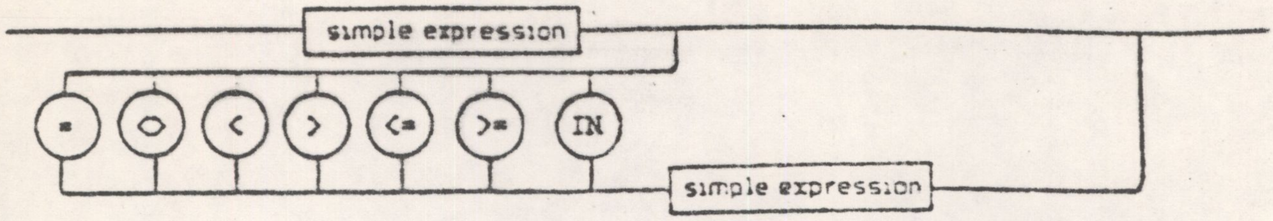
SPECTRUM owners should consult the enclosed sheet 'Making working copies'.

For technical reasons owners of HP4T on a NASCOM (under NAS-SYS) or on a ZX SPECTRUM will be sent Version 1.3 which is identical to Version 1.4 except that bug 3 above is not corrected. To avoid this bug you should always ensure that the real expression in a real-to-integer comparison always appears on the left of the comparison.

Hisoft Pascal 4T owners should note that the Installation Guide and the Alteration Guide are identical documents. This will be available towards the end of December 1982.

The same comments made in Section 1.11 concerning sets apply to simple expressions.

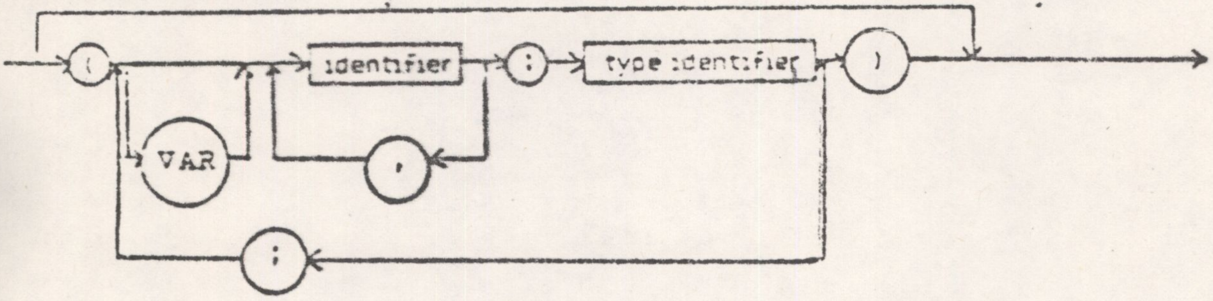
1.13 EXPRESSION.



When using IN, the set attributes are the full range of the type of the simple expression with the exception of integer arguments for which the attributes are taken as if [0..255] had been encountered.

The above syntax applies when comparing strings of the same length, pointers and all scalar types. Sets may be compared using >=, <=, <> or =. Pointers may only be compared using = and <>.

1.14 PARAMETER LIST.



A type identifier must be used following the colon - otherwise *ERROR* 44 will result.

Variable parameters as well as value parameters are fully supported.

Procedures and functions are not valid as parameters.

1.15 STATEMENT.

Refer to the syntax diagram on page 14.

Assignment statements:

See Section 1.7 for information on which assignment statements are illegal.

CASE statements:

An entirely null case list is not allowed i.e. CASE OF END; will generate an error (*ERROR* 13).

The ELSE clause, which is an alternative to END, is executed if the selector ('expression' overleaf) is not found in one of the case lists ('constant' overleaf).

If the END terminator is used and the selector is not found then control is passed to the statement following the END.

FOR statements:

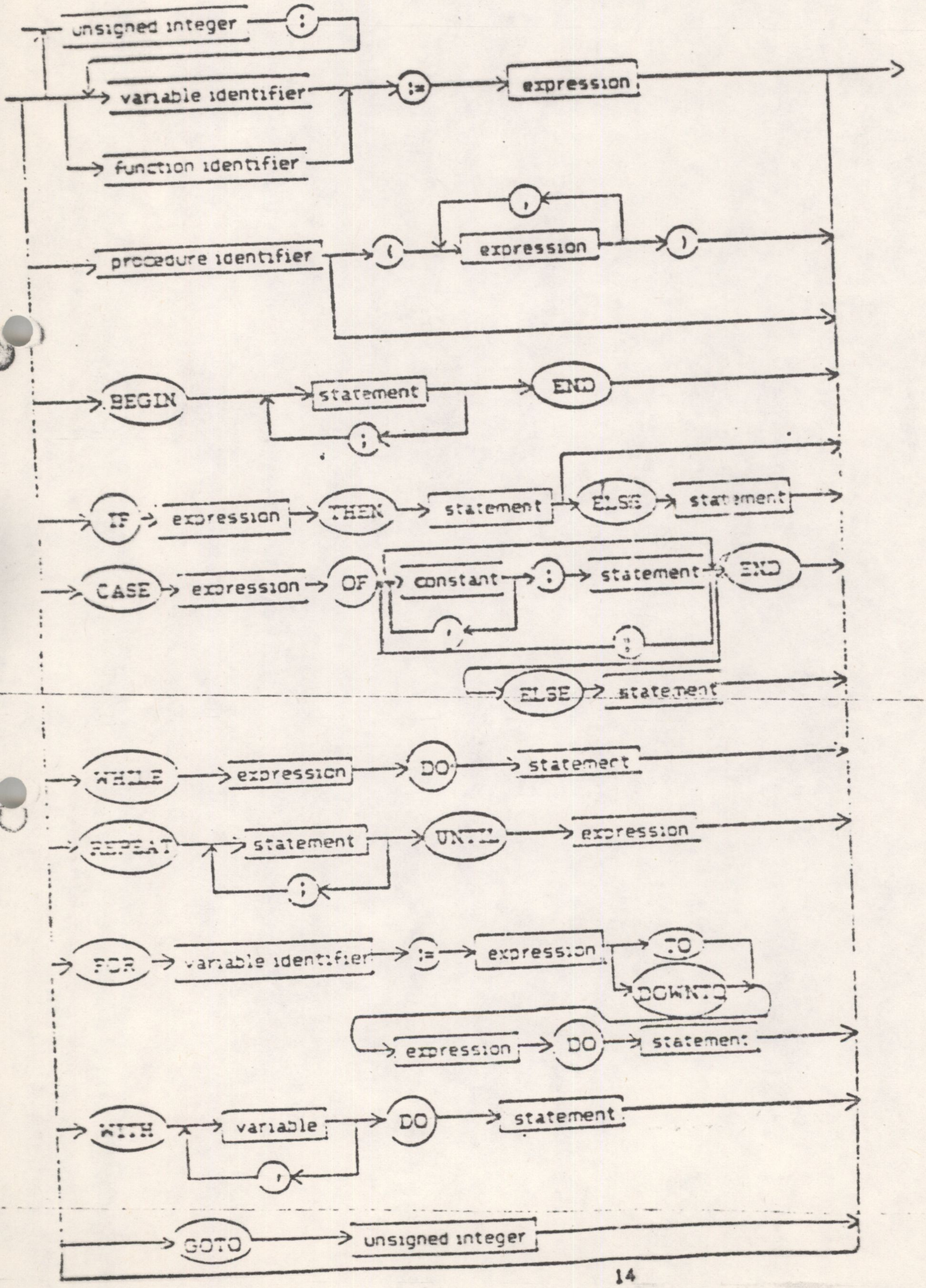
The control variable of a FOR statement may only be an unstructured variable, not a parameter. This is half way between the Jensen/Wirth and draft ISO standard definitions.

GOTO statements:

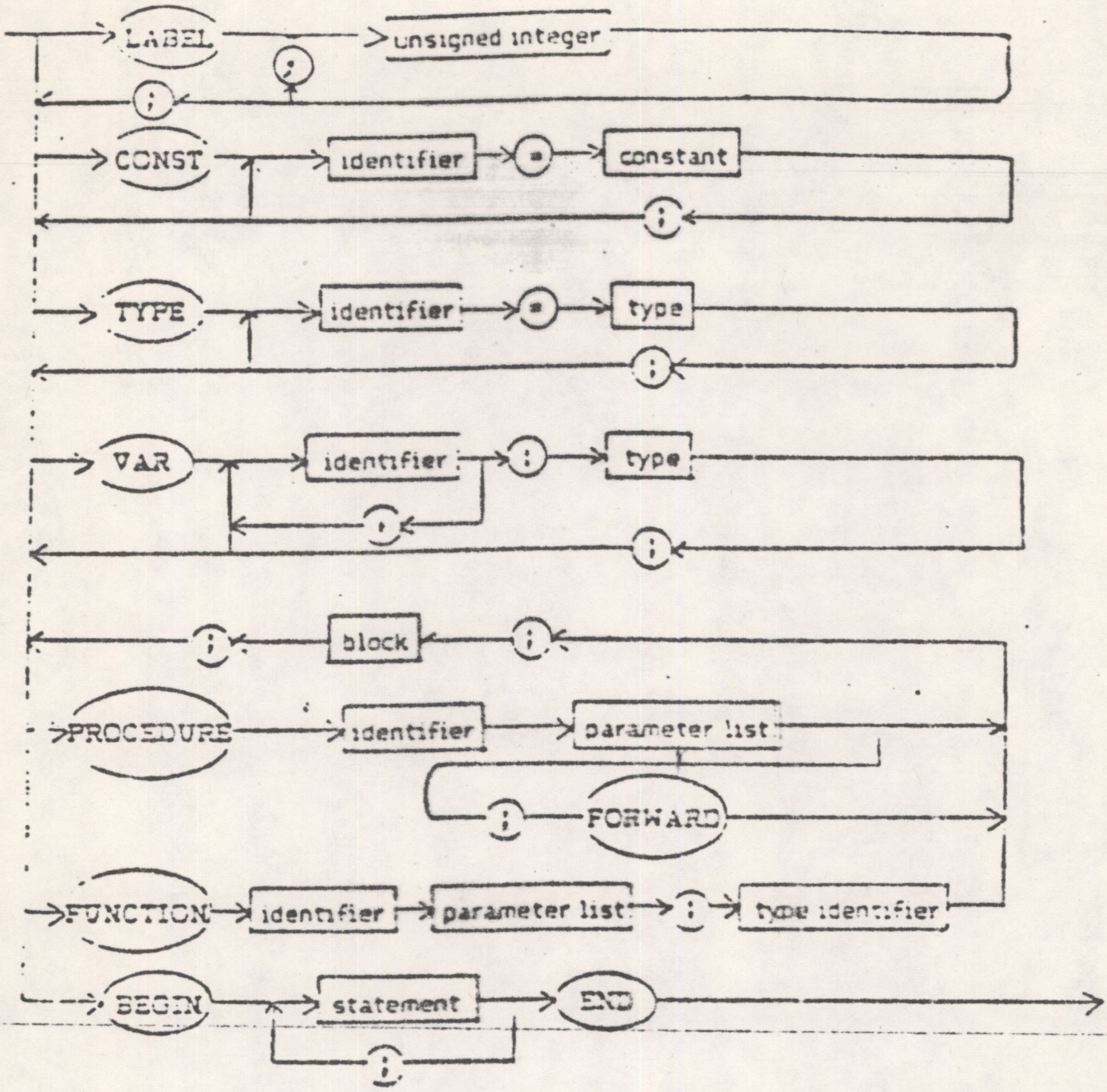
It is only possible to GOTO a label which is present in the same block as the GOTO statement and at the same level.

Labels must be declared (using the Reserved Word LABEL) in the block in which they are used; a label consists of at least one and up to four digits. When a label is used to mark a statement it must appear at the beginning of the statement and be followed by a colon - ':'.

STATEMENT.


unsigned integer
:
variable identifier
:=
expression
function identifier
,
procedure identifier
(
expression
)
BEGIN
statement
END
;
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
:
statement
END
,
;
ELSE
statement
WHILE
expression
DO
statement
REPEAT
statement
UNTIL
expression
;
FOR
variable identifier
:=
expression
TO
DOWNTO
expression
DO
statement
WITH
variable
DO
statement
,
GOTO
unsigned integer

1.16 BLOCK.



LABEL
unsigned integer
;
;
CONST
identifier
=
constant
;
TYPE
identifier
=
type
;
VAR
identifier
:
type
,
;
;
block
;
PROCEDURE
identifier
parameter list
;
FORWARD
FUNCTION
identifier
parameter list
:
type identifier
BEGIN
statement
END
;

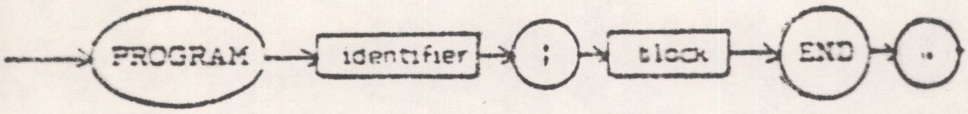
Forward References.

As in the Pascal User Manual and Report (Section 11.C.1) procedures and functions may be referenced before they declared through use of the Reserved Word FORWARD e.g.

```
PROCEDURE a(y:t) ; FORWARD;             (procedure a declared to be)
PROCEDURE b(x:t);                       (forward of this statement)
 BEGIN
 ....
 a(p);                                  (procedure a referenced.)
 ....
 END;
PROCEDURE a;                            (actual declaration of procedure a.)
 BEGIN
 ....
 b(q);
 ....
 END;
```

Note that the parameters and result type of the procedure a are declared along with FORWARD and are not repeated in the main declaration of the procedure. Remember, FORWARD is a Reserved Word.

1.17 PROGRAM.



Since files are not implemented there are no formal parameters of the program.

SECTION 2 PREDEFINED IDENTIFIERS.

2.1 CONSTANTS.

MAXINT   The largest integer available i.e. 32767.

TRUE, FALSE   The constants of type Boolean.

2.2 TYPES.

INTEGER   See Section 1.3.

REAL   See Section 1.3.

CHAR   The full extended ASCII character set of 256 elements.

BOOLEAN   (TRUE,FALSE). This type is used in logical operations including the results of comparisons.

2.3 PROCEDURES AND FUNCTIONS.

2.3.1 Input and Output Procedures.

2.3.1.1 WRITE

The procedure WRITE is used to output data to the screen or printer. When the expression to be written is simply of type character then WRITE(e) passes the 8 bit value represented by the value of the expression e to the screen or printer as appropriate.

Note:

CHR(8)   (CTRL H) gives a destructive backspace on the screen.
CHR(12) (CTRL L) clears the screen or gives a new page on the printer.
CHR(13) (CTRL M) performs a carriage return and line feed.
CHR(16) (CTRL P) will normally direct output to the printer if the screen is in use or vice versa.

Generally though:

```
WRITE(P1,P2,.......Pn); is equivalent to:

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

The write parameters P1,P2,.......Pn can have one of the following forms:

<e> or <e:m> or <e:m:n> or <e:m:H>

where e, m and n are expressions and H is a literal constant.

We have 5 cases to examine:

1] e is of type integer: and either <e> or <e:m> is used.

The value of the integer expression e is converted to a character string with a trailing space. The length of the string can be increased (with leading spaces) by the use of m which specifies the total number of characters to be output. If m is not sufficient for e to be written or m is not present then e is written out in full, with a trailing space, and m is ignored. Note that, if m is specified to be the length of e <u>without</u> the trailing space then <u>no</u> trailing space will be output.

2] e is of type integer and the form <e:m:H> is used.

In this case e is output in hexadecimal. If m=1 or m=2 then the value (e MOD 16^m) is output in a width of exactly m characters. If m=3 or m=4 then the full value of e is output in hexadecimal in a width of 4 characters. If m>4 then leading spaces are inserted before the full hexadecimal value of e as necessary. Leading zeroes will be inserted where applicable. Examples:

```
WRITE(1025:m:H);

m=1   outputs: 1
m=2   outputs: 01
m=3   outputs: 0401
m=4   outputs: 0401
m=5   outputs: _0401
```

3] e is of type real. The forms <e>, <e:m> or <e:m:n> may be used.

The value of e is converted to a character string representing a real number. The format of the representation is determined by n.

If n is not present then the number is output in scientific notation, with a mantissa and an exponent. If the number is negative then a minus sign is output prior to the mantissa, otherwise a space is output. The number is always output to at least one decimal place up to a maximum of 5 decimal places and the exponent is always signed (either with a plus or minus sign). This means that the minimum width of the scientific representation is 8 characters; if the field width m is less than 8 then the full width of 12 characters will always be output. If m>=8 then one or more decimal places will be output up to a maximum of 5 decimal places (m=12). For m>12 leading spaces are inserted before the number. Examples:

```
WRITE(-1.23E 10:m);

m=7   gives: -1.23000E+10
m=8   gives: -1.2E+10
m=9   gives: -1.23E+10
m=10  gives: -1.230E+10
m=11  gives: -1.2300E+10
m=12  gives: -1.23000E+10
m=13  gives: _-1.23000E+10
```

If the form <e:m:n> is used then a fixed-point representation of the number e will be written with n specifying the number of decimal places to be output. No leading spaces will be output unless the field width m is sufficiently large. If n is zero then e is output as an integer. If e is too large to be output in the specified field width then it is output in scientific format with a field width of m (see above). Examples:

```
WRITE(1E2:6:2)    gives:  100.00
WRITE(1E2:8:2)    gives:  _100.00
WRITE(23.455:6:1) gives:  _23.5
WRITE(23.455:4:2) gives:  _2.34550E+01
WRITE(23.455:4:0) gives:  _23
```

4] e is of type character or type string.

Either <e> or <e:m> may be used and the character or string of characters will be output in a minimum field width of 1 (for characters) or the length of the string (for string types). Leading spaces are inserted if m is sufficiently large.

5] e is of type Boolean.

Either <e> or <e:m> may be used and 'TRUE' or 'FALSE' will be output depending on the Boolean value of e , using a minimum field width of 4 or 5 respectively.

2.3.1.2 WRITELN

WRITELN outputs gives a newline. This is equivalent to WRITE(CHR(13)). Note that a linefeed is included.

WRITELN(P1,P2,........P3); is equivalent to:

BEGIN WRITE(P1,P2,.......P3); WRITELN END;

2.3.1.3 PAGE

The procedure PAGE is equivalent to WRITE(CHR(12)); and causes the video screen to be cleared or the printer to advance to the top of a new page.

2.3.1.4 READ

The procedure READ is used to access data from the keyboard. It does this through a buffer held within the runtimes - this buffer is initially empty (except for an end-of-line marker). We can consider that any accesses to this buffer take place through a text window over the buffer through which we can see one character at a time. If this text window is positioned over an end-of-line marker then before the read operation is terminated a new line of text will be read into the buffer from the keyboard. While reading in this line all the various control codes detailed in Section 0.0 will be recognised. Now:

READ(V1,V2,........Vn); is equivalent to:

BEGIN READ(V1); READ(V2); ....; READ(Vn) END;

where V1, V2 etc. may be of type character, string, integer or real.

The statement READ(V); has different effects depending on the type of V. There are 4 cases to consider:

1] V is of type character.

In this case READ(V) simply reads a character from the input buffer and assigns it to V. If the text window on the buffer is positioned on a line marker (a CHR(13) character) then the function EOLN will return the value TRUE and a new line of text is read in from the keyboard. When a read operation is subsequently performed then the text window will be positioned at the start of the new line.

<u>Important note:</u> Note that EOLN is TRUE at the start of the program. This

means that if the first READ is of type character then a CHR(13) value will be returned followed by the reading in of a new line from the keyboard. a subsequent read of type character will return the first character from this new line, assuming it is not blank. See also the procedure READLN below.

2] V is of type string.

A string of characters may be read using READ and in this case a series of characters will be read until the number of characters defined by the string has been read or EOLN = TRUE. If the string is not filled by the read (i.e. if end-of-line is reached before the whole string has been assigned) then the end of the string is filled with null (CHR(0)) characters - this enables the programmer to evaluate the length of the string that was read.

The note concerning in 1] above also applies here.

3] V is of type integer.

In this case a series of characters which represent an integer as defined in Section 1.3 is read. All preceding blanks and end-of-line markers are skipped (this means that integers may be read immediately cf. the note in 1] above).

If the integer read has an absolute value greater than MAXINT (32767) then the runtime error 'Number too large' will be issued and execution terminated.

If the first character read, after spaces and end-of-line characters have been skipped, is not a digit or a sign ('+' or '-') then the runtime error 'Number expected' will be reported and the program aborted.

4] V is of type real.

Here, a series of characters representing a real number according to the syntax of Section 1.3 will be read.

All leading spaces and end-of-line markers are skipped and as for integers above, the first character afterwards must be a digit or a sign. If the number read is too large or too small (see Section 1.3) then an 'Over-flow' error will be reported, if 'E' is present without a following sign or digit then 'Exponent expected' error will be generated and if a decimal point is present without a subsequent digit then a 'Number expected' error will be given.

Reals, like integers, may be read immediately; see 1] and 3] above.

2.3.1.5 READLN

READLN(V1,V2,.......Vn); is equivalent to: BEGIN READ(V1,V2,.......Vn); READLN END;

READLN simply reads in a new buffer from the keyboard; while typing in the buffer you may use the various control functions detailed in Section 0.0. Thus EOLN becomes FALSE after the execution of READLN unless the next line is blank.

READLN may be used to skip the blank line which is present at the beginning of the execution of the object code i.e. it has the effect of reading in a new buffer. This will be useful if you wish to read a component of type character at the beginning of a program but it is not necessary if you are reading an integer or a real (since end-of-line markers are skipped) or if you are reading characters from subsequent lines.

2.3.2 Input Functions.

2.3.2.1 EOLN

The function EOLN is a Boolean function which returns the value TRUE if the next char to be read would be an end-of-line character (CHR(13)). Otherwise the function returns the value FALSE.

2.3.2.2 INCH

The function INCH causes the keyboard of the computer to be scanned and, if a key has been pressed, returns the character represented by the key pressed. If no key has been pressed then CHR(0) is returned. The function therefore returns a result of type character.

2.3.3 Transfer Functions.

2.3.3.1 TRUNC(X)

The parameter X must be of type real or integer and the value returned by TRUNC is the greatest integer less than or equal to X if X is positive or the least integer greater than or equal to X if X is negative. Examples:

TRUNC(-1.5) returns -1   TRUNC(1.9) returns 1

2.3.3.2 ROUND(X)

X must be of type real or integer and the function returns the 'nearest' integer to X (according to standard rounding rules). Examples:

ROUND(-6.5) returns -6 ROUND(11.7) returns 12
ROUND(-6.51) returns -7 ROUND(23.5) returns 24

2.3.3.3 ENTIER(X)

X must be of type real or integer - ENTIER returns the greatest integer less than or equal to X, for all X. Examples:

ENTIER(-6.5) returns -7 ENTIER(11.7) returns 11

Note: ENTIER is not a Standard Pascal function but is the equivalent of BASIC's INT. It is useful when writing fast routines for many mathematical applications.

2.3.3.4 ORD(X)

X may be of any scalar type except real. The value returned is an integer representing the ordinal number of the value of X within the set defining the type of X.

If X is of type integer then ORD(X) = X ; this should normally be avoided

Examples:

ORD('a') returns 97 ORD('@') returns 64

2.3.3.5 CHR(X)

X must be of type integer. CHR returns a character value corresponding to the ASCII value of X. Examples:

CHR(49) returns '1' CHR(91) returns '['

2.3.4 Arithmetic Functions.

In all the functions within this sub-section the parameter X must be of type real or integer.

2.3.4.1 ABS(X)

Returns the absolute value of X (e.g. ABS(-4.5) gives 4.5). The result is of the same type as X.

2.3.4.2 SQR(X)

Returns the value X*X i.e. the square of X. The result is of the same type as X.

2.3.4.3 SQRT(X)

Returns the square root of X - the returned value is always of type real. A 'Maths Call Error' is generated if the argument X is negative.

2.3.4.4 FRAC(X)

Returns the fractional part of X: FRAC(X) = X - ENTIER(X).

As with ENTIER this function is useful for writing many fast mathematical routines. Examples:

FRAC(1.5) returns 0.5 FRAC(-12.56) returns 0.44

2.3.4.5 SIN(X)

Returns the sine of X where X is in radians. The result is always of type real.

2.3.4.6 COS(X)

Returns the cosine of X where X is in radians. The result is of type real.

2.3.4.7 TAN(X)

Returns the tangent of X where X is in radians. The result is always of type real.

2.3.4.8 ARCTAN(X)

Returns the angle, in radians, whose tangent is equal to the number X. The result is of type real.

2.3.4.9 EXP(X)

Returns the value e^X where e = 2.71828. The result is always of type real.

2.3.4.10 LN(X)

Returns the natural logarithm (i.e. to the base e) of X. The result is of type real. If X <= 0 then a 'Maths Call Error' will be generated.

2.3.5 Further Predefined Procedures.

2.3.5.1 NEW(p)

The procedure NEW(p) allocates space for a dynamic variable. The variable p is a pointer variable and after NEW(p) has been executed p contains the address of the newly allocated dynamic variable. The type of the dynamic variable is the same as the type of the pointer variable p and this can be of any type.

To access the dynamic variable p^ is used - see Appendix 4 for an example of the use of pointers to reference dynamic variables.

To re-allocate space used for dynamic variables use the procedures MARK and RELEASE (see below).

2.3.5.2 MARK(v1)

This procedure saves the state of the dynamic variable heap to be saved in the pointer variable v1. The state of the heap may be restored to that when the procedure MARK was executed by using the procedure RELEASE (see below).

The type of variable to which v1 points is irrelevant, since v1 should only be used with MARK and RELEASE never NEW.

For an example program using MARK and RELEASE see Appendix 4.

2.3.5.3. RELEASE(v1)

This procedure frees space on the heap for use of dynamic variables. The state of the heap is restored to its state when MARK(V1) was executed - thus effectively destroying all dynamic variables created since the execution of the MARK procedure. As such it should be used with great care.

See above and Appendix 4 for more details.

2.3.5.4 INLINE(C1,C2,C3,.........)

This procedure allows Z80 machine code to be inserted within the Pascal program; the values (C1 MOD 256, C2 MOD 256, C3 MOD 256, ...) are inserted in the object program at the current location counter address held by the compiler. C1, C2, C3 etc. are integer constants of which there can be any number. Refer to Appendix 4 for an example of the use of INLINE.

2.3.5.5 USER(V)

USER is a procedure with one integer argument V. The procedure causes a call to be made to the memory address given by V. Since Hisoft Pascal 4 holds integers in two's complement form (see Appendix 3) then in order to refer to addresses greater than £7FFF (32767) negative values of V must be used. For example £C000 is -16384 and so USER(-16384); would invoke a a call to the memory address £C000. However, when using a constant to refer to a memory address, it is more convenient to use hexadecimal.

The routine called should finish with a Z80 RET instruction (£C9) and must preserve the IX register.

2.3.5.6 HALT

This procedure causes program execution to stop with the message 'Halt at PC=XXXX' where XXXX is the hexadecimal memory address of the location where the HALT was issued. Together with a compilation listing, HALT may be used to determine which of two or more paths through a program are taken. This will normally be used during de-bugging.

2.3.5.7 POKE(X,V)

POKE stores the expression V in the computer's memory starting from the memory address X. X is of type integer and V can be of any type except SET. See Section 2.3.5.5 above for a discussion of the use of integers to represent memory addresses. Examples:

POKE(£6000,'A') places £41 at location £6000.

POKE(-16384,3.6E3) places 00 0B 80 70 (in hexadecimal) at location £C000.

0

dec BC

add a, b

ld (hl), b

2.3.5.8 TOUT (NAME,START,SIZE)

TOUT is the procedure which is used to save variables on tape. The first parameter is of type ARRAY[1..8] OF CHAR and is the name of the file to be saved. SIZE bytes of memory are dumped starting at the address START. Both these parameters are of type INTEGER.

E.g. to save the variable V to tape under the name 'VAR   ' use:

```
TOUT('VAR     ',ADDR(V),SIZE(V))
```

The use of actual memory addresses gives the user far more flexibility than just the ability to save arrays. For example if a system has a memory mapped screen, entire screenfuls may be saved directly. See Appendix 4 for an example of the use of TOUT.

2.3.5.9 TIN (NAME,START)

This procedure is used to load, from tape, variables etc. that have been saved using TOUT. NAME is of type ARRAY[1..8] of CHAR and START is of type INTEGER. The tape is searched for a file called NAME which is then loaded at memory address START. The number of bytes to load is taken from the tape (saved on the tape by TOUT).

E.g. to load the variable saved in the example in Section 2.3.5.8 above use:

```
TIN('VAR     ',ADDR(V))
```

Because source files are recorded by the editor using the same format as that used by TIN and TOUT, TIN may be used to load text files into ARRAYs of CHAR for processing (see the HP4T Alteration Guide).

See Appendix 4 for an example of the use of TIN.

2.3.5.10 OUT(P,C)

This procedure is used to directly access the Z80's output ports without using the procedure INLINE. The value of the integer parameter P is loaded in to the BC register, the character parameter C is loaded in to the A register and the assembly instruction OUT (C),A is executed.

E.g. OUT(1,'A') outputs the character 'A' to the Z80 port 1.

2.3.6 Further Predefined Functions.

2.3.6.1 RANDOM

This returns a pseudo-random number between 0 and 255 inclusive. Although this routine is very fast it gives poor results when used repeatedly within loops that do not contain I/O operations.

If the user requires better results than this function yields then he/she should write a routine (either in Pascal or machine code) tailored to the particular application.

2.3.6.2 SUCC(X)

X may be of any scalar type except real and SUCC(X) returns the successor of X. Examples:

SUCC('A') returns 'B' SUCC('5') returns '6'

2.3.6.3 PRED(X)

X may be of any scalar type except real; the result of the function is the predecessor of X. Examples:

PRED('j') returns 'i' PRED(TRUE) returns FALSE

2.3.6.4 ODD(X)

X must be of type integer. ODD returns a Boolean result which is TRUE if X is odd and FALSE if X is even.

2.3.6.6 ADDR(V)

This function takes a variable identifier of any type as a parameter and returns an integer result which is the memory address of the variable identifier V. For information on how variables are held, at runtime, within Hisoft Pascal 4 see Appendix 3. For an example of the use of ADDR see Appendix 4.

2.3.6.7 PEEK(X,T)

The first parameter of this function is of type integer and is used to specify a memory address (see Section 2.3.5.5). The second argument is a type; this is the result type of the function.

PEEK is used to retrieve data from the memory of the computer.and the result may be of any type.

In all PEEK and POKE (the opposite of PEEK) operations data is moved in Hisoft Pascal 4's own internal representation detailed in Appendix 3. For example: if the memory from £5000 onwards contains the values: 50 61 73 63 61 6C (in hexadecimal) then:

```
WRITE(PEEK(£5000,ARRAY[1..6] OF CHAR)) gives 'Pascal'
WRITE(PEEK(£5000,CHAR)) gives 'P'
WRITE(PEEK(£5000,INTEGER)) gives 24912
WRITE(PEEK(£5000,REAL)) gives 2.46227E+29
```

see Appendix 3 for more details on the representation of types within Hisoft Pascal 4.

2.3.6.7 SIZE(V)

The parameter of this function is a variable. The integer result is the amount of storage taken up by that variable, in bytes.

2.3.6.8 INP(P)

INP is used to access the Z80's ports directly without using the procedure INLINE. The value of the integer parameter P is loaded into be BC register and the character result of the function is obtained by executing the assembly language instruction IN A,(C).

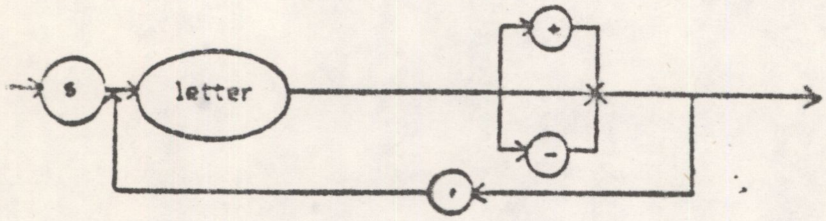
SECTION 3 COMMENTS AND COMPILER OPTIONS.

3.1 Comments.

A comment may occur between any two reserved words, numbers, identifiers or special symbols - see Appendix 2. A comment starts with a '{' character or the '(* ' character pair. Unless the next character is a '$' all characters are ignored until the next '}' character or '*)' character pair. If a '$' was found then the compiler looks for a series of compiler options (see below) after which characters are skipped until a '}' or '*)' is found.

3.2 Compiler Options.

The syntax for specifying compiler options is:



The following options are available:

Option L:

Controls the listing of the program text and object code address by the compiler.

If L+ then a full listing is given.
If L- then lines are only listed when an error is detected.

DEFAULT: L+

Option O:

Controls whether certain overflow checks are made. Integer multiply and divide and all real arithmetic operations are <u>always</u> checked for overflow.

If O+ then checks are made on integer addition and subtraction.

If O- then the above checks are not made.

DEFAULT: O+

Option C:

Controls whether or not keyboard checks are made during object code program execution. If C+ then if CC is pressed then execution will return to with a HALT message - see Section 2.3.5.6.

This check is made at the beginning of all loops, procedures and functions. Thus the user may use this facility to detect which loop etc. is not terminating correctly during the debugging process. It should certainly be disabled if you wish the object program to run quickly.

If C- then the above check is not made.

DEFAULT: C+

Option S:

Controls whether or not stack shecks are made.

If S+ then, at the beginning of each procedure and function call, a check is made to see if the stack will probably overflow in this block. If the runtime stack overflows the dynamic variable heap or the program then the message 'Out of RAM at PC=XXXX' is displayed and execution aborted. Naturally this is not foolproof; if a procedure has a large amount of stack usage within itself then the program may 'crash'. Alternatively, if a function contains very little stack usage while utilising recursion then it is possible for the function to be halted unnecessarily.

If S- then no stack checks are performed.

DEFAULT: S+

Option A:

Controls whether checks are made to ensure that array indices are within the bounds specified in the array's declaration.

If A+ and an array index is too high or too low then the message 'Index too high' or 'Index too low' will be displayed and the program execution halted.

If A- then no such checks are made.

DEFAULT: A+

Option I:

When using 16 bit 2's complement integer arithmetic, overflow occurs when performing a >, <, >=, or <= operation if the arguments differ by more than MAXINT (32767). If this occurs then the result of the comparison will be incorrect. This will not normally present any difficulties; however, should the user wish to compare such numbers, the use of I+ ensures that the results of the comparison will be correct. The equivalent situation may arise with real arithmetic in which case an overflow error will be issued if the arguments differ by more than approximately 3.4E38 ; this cannot be avoided.

If I- then no check for the result of the above comparisons is made.

SECTION 4 THE INTEGRAL EDITOR

4.1 Introduction to the Editor.

The editor supplied with all versions of Hisoft Pascal 4T is a simple, line-based editor designed to work with all Z80 operating systems while maintaining ease of use and the ability to edit programs quickly and efficiently.
Text is held in memory in a compacted form; the number of leading spaces in a line is held as one character at the beginning of the line and all HP4T Reserved Words are tokenised into one character. This leads to a typical reduction in text size of 25%.

NOTE: throughout this section the DELETE key is referred to instead of the control code CH. It appears more natural to do this.

The editor is entered automatically when HP4T is loaded from tape and displays the message:

Copyright Hisoft 1982
All rights reserved

followed by the editor prompt '>'.

In response to the prompt you may enter a command line of the following format:

C N1, N2, S1, S2

followed by a RETURN where:

C is the command to be executed (see Section 4.2 below).
N1 is a number in the range 1 - 32767 inclusive.
N2 is a number in the range 1 - 32767 inclusive.
S1 is a string of characters with a maximum length of 20.
S2 is a string of characters with a maximum length of 20.

The comma is used to separate the various arguments (although this can be changed - see the 'S' command) and spaces are ignored, except within the strings. None of the arguments are mandatory although some of the commands (e.g. the 'D'elete command) will not proceed without N1 and N2 being specified. The editor remembers the previous numbers and strings that you entered and uses these former values, where applicable, if you do not specify a particular argument within the command line. The values of N1 and N2 are initially set to 10 and the strings are initially empty. If you enter an illegal command line such as F-1,100,HELLO then the line will be ignored and the message 'Pardon?' displayed - you should then retype the line correctly e.g. F1,100,HELLO. This error message will also be displayed if the length of S2 exceeds 20; if the length of S1 is greater than 20 then any excess characters are ignored.

Commands may be entered in upper or lower case.

While entering a command line, all the relevant control functions described in Section 0.0 may be used e.g. CX to delete to the beginning of the line.

The following sub-section details the various commands available within the editor - note that wherever an argument is enclosed by the symbols '< >' then that argument must be present for the command to proceed.

DEFAULT: I-

Option P:

If the P option is used the device to which the compilation listing is sent is changed i.e. if the video screen was being used the printer is used and vice versa. Note that this option is not followed by a '+' or '-'.

DEFAULT: The video screen is used.

Option F:

This option letter must be followed by a space and then an eight character filename. If the filename has less than eight characters it should be padded with spaces.

The presence of this option causes inclusion of Pascal source text from the specified file from the end of the current line - useful if the programmer wishes to build up a 'library' of much-used procedures and functions on tape and then include them in particular programs.

The program should be saved using the built-in editor's 'P' command. On most systems the list option L- should be used - otherwise the compiler will not compile fast enough.

Example: ($L-,F MATRIX include the text from a tape file MATRIX);

When writing very large programs there may not be enough room in the computer's memory for the source and object code to be present at the same time. It is however possible to compile such programs by saving them to tape and using the 'F' option - then only 128 bytes of the source are in RAM at any one time, leaving much more room for the object code.

This option may not be nested and is not implemented in the ZX Spectrum version.

The compiler options may be used selectively. Thus debugged sections of code may be speeded up and compacted by turning the relevant checks off whilst retaining checks on untested items of code.

4.2 The Editor Commands.

CS^5 CS^S

4.2.1 Text Insertion.

Text may be inserted into the textfile either by typing a line number, a space and then the required text or by use of the 'I' command. Note that if you type a line number followed by RETURN (i.e. without any text) then that line will be deleted from the text if it exists. Whenever text is being entered then the control functions CX (delete to the beginning of the line), CI (go to the next tab position), CC (return to the command loop) and CP (toggle the printer) may be employed. The DELETE (or BACKSPACE) key will produce a destructive backspace (but not beyond the beginning of the text line). Text is entered into an internal buffer within HP4T and if this buffer should become full then you will be prevented from entering any more text - you must then use DELETE or CX to free space in the buffer.

CS^A

Command: I n,m

Use of this command gains entry to the automatic insert mode: you are prompted with line numbers starting at n and incrementing in steps of m. You enter the required text after the displayed line number, using the various control codes if desired and terminating the text line with RETURN. To exit from this mode use the control function CC (see Section 0.0 and the relevant Implementation Note).

If you enter a line with a line number that already exists in the text then the existing line will be deleted and replaced with the new line, after you have pressed RETURN. If the automatic incrementing of the line number produces a line number greater than 32767 then the Insert mode will exit automatically.

If, when typing in text, you get to the end of a screen line without having entered 128 characters (the buffer size) then the screen will be scrolled up and you may continue typing on the next line - an automatic indentation will be given to the text so that the line numbers are effectively separated from the text.

4.2.2 Text Listing.

Text may be inspected by use of the 'L' command: the number of lines displayed at any one time during the execution of this command is fixed initially (see your Implementation Note) but may be changed through use of the 'K' command.

Command: L n,m

This lists the current text to the display device from line number n to line number m inclusive. The default value for n is always 1 and the default value for m is always 32767 i.e. default values are not taken from previously entered arguments. To list the entire textfile simply use 'L' without any arguments. Screen lines are formatted with a left hand margin so that the line numbers is clearly displayed. The number of screen lines listed on the display device may be controlled through use of the 'K' command - after listing a certain number of lines the list will pause (if not yet at line number m), hit control function CC to return to the main editor loop or any other key to continue the listing.

Command: K n

'K' sets the number of screen lines to be listed to the display device before the display is paused as described in 'L' above. The value (n MOD 256) is computed and stored. For example use K5 if you wish a subsequent 'L'ist to produce five screen lines at a time.

4.2.3 Text Editing.

Once some text has been created there will inevitably be a need to edit some lines. Various commands are provided to enable lines to be amended, deleted, moved and renumbered:

Command: D <n,m>

All lines from n to m inclusive are deleted from the textfile. If m<n or less than two arguments are specified then no action will be taken; this is to help prevent careless mistakes. A single line may be deleted by making m=n ; this can also be accomplished by simply typing the line number followed by RETURN.

Command: M n,m

This causes the text at line n to be entered at line m deleting any text that already exists there. Note that line n is left alone. So this command allows you to 'M'ove a line of text to another position within the textfile. If line number n does not exist then no action is taken.

Command: N <n,m>

Use of the 'N' command causes the textfile to be renumbered with a first line number of n and in line number steps of m. Both n and m must be present and if the renumbering would cause any line number to exceed 32767 then the original numbering is retained.

Command: F n,m,f,s

The text existing within the line range n < x < m is searched for an occurrence of the string f - the 'find' string. If such an occurrence is found then the relevant text line is displayed and the Edit mode is entered - see below. You may then use commands within the Edit mode to search for subsequent occurrences of the string f within the defined line range or to substitute the string s (the 'substitute' string) for the current occurrence of f and then search for the next occurrence of f; see below for more details.

Note that the line range and the two strings may have been set up previously by any other command so that it may only be necessary to enter 'F' to initiate the search - see the example in Section 4. for clarification.

Command: E n

Edit the line with line number n. If n does not exist then no action is taken; otherwise the line is copied into a buffer and displayed on the screen (with the line number), the line number is displayed again underneath the line and the Edit mode is entered. All subsequent editing takes place within the buffer and not in the text itself; thus the original line can be recovered at any time.

In this mode a pointer is imagined moving through the line (starting at the first character) and various sub-commands are supported which allow you to edit the line. The sub-commands are:

' ' (space) - increment the text pointer by one i.e. point to the next character in the line. You cannot step beyond the end of the line.

DELETE (or BACKSPACE) - decrement the text pointer by one to point at the previous character in the line. You cannot step backwards beyond the first character in the line.

CI (control function) - step the text pointer forwards to the next tab position but not beyond the end of the line.

RETURN - end the edit of this line keeping all the changes made.

Q - quit the edit of this line i.e. leave the edit ignoring all the changes made and leaving the line as it was before the edit was initiated.

R - reload the edit buffer from the text i.e. forget all changes made on this line and restore the line as it was originally.

L - list the rest of the line being edited i.e. the remainder of the line beyond the current pointer position. You remain in the Edit mode with the pointer re-positioned at the start of the line.

K - kill (delete) the character at the current pointer position.

Z - delete all the characters from (and including) the current pointer position to the end of the line.

F - find the next occurrence of the 'find' string previously defined within a command line (see the 'F' command above). This sub-command will automatically exit the edit on the current line (keeping the changes) if it does not find another occurrence of the 'find' string in the current line. If an occurrence of the 'find' string is detected in a subsequent line (within the previously specified line range) then the Edit mode will be entered for the line in which the string is found. Note that the text pointer is always positioned at the start of the line after a successful search.

S - substitute the previously defined 'substitute' string for the currently found occurrence of the 'find' string and then perform the sub-command 'F' i.e. search for the next occurrence of the 'find' string. This, together with the above 'F' sub-command, is used to step through the textfile optionally replacing occurrences of the 'find' string with the substitute string - see Section 4.3 for an example.

++Important Note++ In the current version of HP4T there is a known bug in the operation of the sub-command 'S' - this sub-command should only be used immediately after an 'F' command, an 'F' sub-command or an 'S' sub-command. In practice this should pose no problems.

I - insert characters at the current pointer position. You will remain in this sub-mode until you press RETURN - this will return you to the main Edit mode with the pointer positioned after the las character that you inserted. Using DELETE (or BACKSPACE) within this sub-mode will cause the character to the left of the pointer to be deleted from the buffer while the use of CI (control function) will advance the pointer to the next tab position, inserting spaces.

X - this advances the pointer to the end of the line and automatically enters the Insert sub-mode detailed above.

C - change sub-mode. This allows you to overwrite the character at the current pointer position and then advances the pointer by one. You remain in the change sub-mode until you press RETURN whence you are taken back to the Edit mode with the pointer positioned after the last character you changed. DELETE (or BACKSPACE) within this sub-mode simply decrements the pointer by one i.e. moves it left while CI has no effect.

4.2.4 Tape Commands.

Text may be saved to tape or loaded from tape using the commands 'P' and 'G'.

Command: P n,m,s

The line range defined by n < x < m is saved to tape in HP4T format under the filename specified by the string s. Remember that these arguments may have been set by a previous command. Before entering this command make sure that your tape recorder is switched on and in RECORD mode. While the text is being saved the message 'Busy..' is displayed.

Command: G,,s

The tape is searched for a file in HP4T tape format and with a filename of s. While the search is taking place the message 'Busy..' will be displayed. If a valid HP4T tape file is found but has the wrong filename then the message 'Found' followed by the filename that was found on the tape is displayed and the search continued. Once the correct filename is found then 'Found' will appear on the list device and the file will be loaded into memory. If an error is detected during the load then 'Checksum error' is shown and the load aborted. If this happens you must rewind the tape, press PLAY and type 'G' again.

If the string s is empty then the first HP4T file on the tape will be loaded, regardless of its filename.

While searching of the tape is going on you may abort the load by holding CC down; this will interrupt the load and return to the main editor loop.

Note that if any textfile is already present them the textfile that is loaded from tape will be appended to the existing file and the whole file will be renumbered starting with line 1 in steps of 1.

4.2.5 Compiling and Running from the Editor

Command: C n

This causes the text starting at line number n to be compiled. If you do not specify a line number then the text will be compiled from the first existing line. For further details see Section 0.2.

Command: R

The previously compiled object code will be executed, but only if the source has not been expanded in the meantime - see Section 0.2 for more detail.

Command: T n

This is the 'T'ranslate command. The current source is compiled from line n (or from the start if n is omitted) and, if the compilation is successful, you will be prompted with 'Ok?': if you answer 'Y' to this prompt then the object code produced by the compilation will be moved to the end of the runtimes (destroying the compiler) and then the runtimes and the object code will be dumped out to tape with a filename equal to that previously defined for the 'find' string. You may then, at a later stage, load this file into memory, using the HP4T loader, whereupon it will automatically execute the object program. Note that the object code is located at and moved to the end of the runtimes so that, after a 'T'ranslate you will need to reload the compiler - however this should present no problems since you are only likely to 'T'ranslate a program when it is fully working.

If you decide not to continue with the dump to tape then simply type any character other than 'Y' to the 'Ok?' prompt; control is returned to the editor which will still function perfectly since the object code was not moved.

4.2.4 Other Commands.

Command: B

This simply returns control to the operating system. For details of how to re-enter the compiler refer to the HP4T Alteration Guide and your Implementation Note.

Command: O n,m

Remember that text is held in memory in a tokenised form with leading spaces shortened into a one character count and all HP4T Reserved Words reduced to a one character token. However if you have somehow got some text in memory, perhaps from another editor, which is not tokenised then you can use the 'O' command to tokenise it for you. Text is read into a buffer in an expanded form and then put back into the file in a tokenised form - this may of course take a little time to perform. A line range must be specified, or the previously entered values will be assumed.

Command: S,,d

This command allows you to change the delimiter which is taken as separating the arguments in the command line. On entry to the editor the comma ',' is taken as the delimiter; this may be changed by the use of the 'S' command to the first character of the specified string d. Remember that once you have defined a new delimiter it must be used (even within the 'S' command) until another one is specified.

Note that the separator may not be a space.

4.7 An Example of the use of the Editor.

Let us assume that you have typed in the following program (using I10,10):

```
 10 PROGRAM BUBBLESORT
 20 CONST
 30   Size = 2000;
 40 VAR
 50   Numbers : ARRAY [1..Size] OF INTEGER;
 60   I, Temp : INTEGER;
 70 BEGIN
 80   FOR I:=1 TO Size DO Number[I] := RANDOM;
 90   REPEAT
100     FOR I:=1 TO Size DO
110     Noswaps := TRUE;
120     IF Number[I] > Number[I+1] THEN
130       BEGIN
140         Temp := Number[I];
150         Number[I] := Number[I+1];
160         Number[I+1] := Temp;
170         Noswaps := FALSE
180       END
190   UNTIL Noswapss
200 END.
```

This program has a number of errors which are as follows:

Line 10   Missing semi-colon.
Line 30   Not really an error but say we want a size of 100.
Line 100 Size should be Size-1.
Line 110 This should be at line 95 instead.
Line 190 Noswapss should be Noswaps.

Also the variable Numbers has been declared but all references are to Number. Finally the BOOLEAN variable Noswaps has not been declared.

To put all this right we can proceed as follows:

F60,200,Number,Numbers	and then use the sub-command 'S' repeatedly.
E10	then the sequence X ; RETURN RETURN
E30	then _________ K C 1 RETURN RETURN
F100,100,Size,Size-1	followed by the sub-command 'S'.
M110,95	.
110	followed by RETURN.
E190	then X DELETE RETURN RETURN
65 Noswaps : BOOLEAN;	.
N10,10	to renumber in steps of 10.

You are strongly recommended to work through the above example actually using the editor - you may find it a little cumbersome at first if you have been used to screen editors but it should not take long to adapt.

APPENDIX 1 ERRORS.

A.1.1 Error numbers generated by the compiler.

1. Number too large.
2. Semi-colon expected.
3. Undeclared identifier.
4. Identifier expected.
5. Use '=' not ':=' in a constant declaration.
6. '=' expected.
7. This identifier cannot begin a statement.
8. ':=' expected.
9. ')' expected.
10. Wrong type.
11. '.' expected.
12. Factor expected.
13. Constant expected.
14. This identifier is not a constant.
15. 'THEN' expected.
16. 'DO' expected.
17. 'TO' or 'DOWNTO' expected.
18. '(' expected.
19. Cannot write this type of expression.
20. 'OF' expected.
21. ',' expected.
22. ':' expected.
23. 'PROGRAM' expected.
24. Variable expected since parameter is a variable parameter.
25. 'BEGIN' expected.
26. Variable expected in call to READ.
27. Cannot compare expressions of this type.
28. Should be either type INTEGER or type REAL.
29. Cannot read this type of variable.
30. This identifier is not a type.
31. Exponent expected in real number.
32. Scalar expression (not numeric) expected.
33. Null strings not allowed (use CHR(0)).
34. '[' expected.
35. ']' expected.
36. Array index type must be scalar.
37. '..' expected.
38. ']' or ',' expected in ARRAY declaration.
39. Lowerbound greater than upperbound.
40. Set too large (more than 256 possible elements).
41. Function result must be type identifier.
42. ',' or ']' expected in set.
43. '..' or ',' or ']' expected in set.
44. Type of parameter must be a type identifier.
45. Null set cannot be the first factor in a non-assignment statement.
46. Scalar (including real) expected.
47. Scalar (not including real) expected.
48. Sets incompatible.
49. '<' and '>' cannot be used to compare sets.
50. 'FORWARD', 'LABEL', 'CONST', 'VAR', 'TYPE' or 'BEGIN' expected.

51. Hexadecimal digit expected.
52. Cannot POKE sets.
53. Array too large (> 64K).
54. 'END' or ';' expected in RECORD definition.
55. Field identifier expected.
56. Variable expected after 'WITH'.
57. Variable in WITH must be of RECORD type.
58. Field identifier has not had asociated WITH statement.
59. Unsigned integer expected after 'LABEL'.
60. Unsigned integer expected after 'GOTO'.
61. This label is at the wrong level.
62. Undeclared label.
63. The parameter of SIZE should be a variable.
64. Can only use equality tests for pointers.
67. The only write parameter for integers with two ':'s is e:m:H.
68. Strings may not contain end of line characters.
69. The parameter of NEW, MARK or RELEASE should be a variable of pointer type.
70. The parameter of ADDR should be a variable.

A.1.2 Runtime Error Messages.

When a runtime error is detected then one of the following messages will be displayed, followed by ' at PC=XXXX' where XXXX is the memory location at which the error occurred. Often the source of the error will be obvious; if not, consult the compilation listing to see where in the program the error occurred, using XXXX to cross reference. Occasionally this does not give the correct result.

1. Halt
2. Overflow
3. Out of RAM
4. / by zero also generated by DIV.
5. Index too low
6. Index too high
7. Maths Call Error
8. Number too large
9. Number expected
10. Line too long
11. Exponent expected

Runtime errors result in the program execution being halted.

APPENDIX 2 RESERVED WORDS AND PREDEFINED IDENTIFIERS.

A 2.1 Reserved Words.

AND	ARRAY	BEGIN	CÅSE	CONST	DIV	DC
DOWNTO	ELSE	END	FORWARD	FUNCTION	GOTO	IF
IN	LABEL	MOD	NIL	NOT	OF	O
PACKED	PROCEDURE	PROGRAM		RECORD	REPEAT	SE
	THEN					
TO	TYPE	UNTIL	VAR	WHILE	WITH	

A 2.2 Special Symbols.

The following symbols are used by Hisoft Pascal 4 and have a reserved meaning:

+	-	*	/		
=	<>	<	<=	>=	>
(	)	[	]		
{	}	(*	*)		
^	:=	.	,	;	:
'	..				

A 2.3 Predefined Identifiers.

The following entities may be thought of a declared in a block surrounding the whole program and they are therefore available througnout the program unless re-defined by the programmer within a inner block.
For further information see Section 2.

CONST MAXINT = 32767;

TYPE BOOLEAN = (FÀLSE, TRUE);
CHAR (The expanded ASCII character set);
INTEGER = -MAXINT..MAXINT;
REAL (A subset of the real numbers. See Section 1.3.)

PROCEDURE WRITE; WRITELN; READ; READLN; PAGE; HALT; USER; POKE; INLINE;
OUT; NEW; MARK; RELEASE; TIN; TOUT;

FUNCTION ABS; SQR; ODD; RANDOM; ORD; SUCC; PRED; INCH; EOLN;
PEEK; CHR; SQRT; ENTIER; ROUND; TRUNC; FRAC; SIN; COS;
TAN; ARCTAN; EXP; LN; ADDR; SIZE; INP;

APPENDIX 3 DATA REPRESENTATION AND STORAGE.

A 3.1 Data Representation.

The following discussion details how data is represented internally by Hisoft Pascal 4.

The information on the amount of storage required in each case should be of use to most programmers (the SIZE function may be used see Section 2.3.6.7); other details may be needed by those attempting to merge Pascal and machine code programs.

A 3.1.1 Integers.

Integers occupy 2 bytes of storage each, in 2's complement form.
Examples:

1	≡	£0001
256	≡	£0100
-256	≡	£FF00

The standard Z80 register used by the compiler to hold integers is HL.

A 3.1.2 Characters, Booleans and other Scalars.

These occupy 1 byte of storage each, in pure, unsigned binary.

Characters: 8 bit, extended ASCII is used.

'E'	≡	£45
'['	≡	£5B

Booleans:

ORD(TRUE) = 1 so TRUE is represented by 1.
ORD(FALSE) = 0 so FALSE is representd by 0.

The standard Z80 register used by the compiler for the above is A.

A 3.1.3 Reals.

The (mantissa, exponent) form is used similar to that used in standard scientific notation - only using binary instead of denary. Examples:

$$2 \equiv 2*10^0 \quad \text{or} \quad 1.0_2 * 2^1$$

$$1 \equiv 1*10^0 \quad \text{or} \quad 1.0_2 * 2^0$$

$$-12.5 \equiv -1.25\times10^{1} \quad \text{or} \quad -25\times2^{-1} \equiv -11001_2\times2^{-1} \equiv -1.1001_2\times2^{3}$$ when normalised.

$$0.1 \equiv 1.0\times10^{-1} \quad \text{or} \quad \frac{1}{10} \equiv \frac{1}{1010_2} \equiv \frac{0.1_2}{101_2}$$

so now we need to do some binary long division...

```
          0.0001100
    101 / 0.1000000000000000
              101
               110
               101
                 1000
                  101    at this point
                        we see that the
                        fraction recurs
```

$$= \frac{0.1_2}{101_2} = 0.000\ddot{1}10\dot{0}_2$$

$$\underline{1.100\dot{1}10\dot{0} \times 2^{-4}}$$ <u>answer.</u>

So how do we use the above results to represent these numbers in the computer? Well, firstly we reserve 4 bytes of storage for each real in the following format:

```
| sign |     | normalised mantissa |     | exponent |     | data |
  23           22                 0       7        0       bit
 \_____________/\________________/ \______________/
        H              L           E          D           register
```

sign: the sign of the mantissa: 1 = negative, 0 = positive.

normalised mantissa: the mantissa normalised to the form 1.xxxxxx with the top bit (bit 22) always 1 except when representing zero (HL=0, DE=0).

exponent: the exponent in binary 2's complement form.

Thus:

2	≡	0	1000000	00000000	00000000	00000001	(£40,£00,£00,£01)
1	≡	0	1000000	00000000	00000000	00000000	(£40,£00,£00,£00)
-12.5	≡	1	1100100	00000000	00000000	00000011	(£E4,£00,£00,£03)
0.1	≡	0	1100110	01100110	01100110	11111100	(£66,£66,£66,£FC)

So, remembering that HL and DE are used to hold real numbers, then we would have to load the registers in the following way to represent each of the above numbers:

```
      2   ≡    LD     HL,£4000
               LD     DE,£0100

      1   ≡    LD     HL,£4000
               LD     DE,£0000

  -12.5   ≡    LD     HL,£E400
               LD     DE,£0300

    0.1   ≡    LD     HL,£6666
               LD     DE,£FC66
```

The last example shows why calculations involving binary fractions can be inaccurate: 0.1 cannot be accurately represented as a binary fraction, to a finite number of decimal places.

N.B. Reals are stored in memory in the order ED LH.

A 3.1.4 Records and Arrays.

Records use the same amount of storage as the total of their components.

Arrays: if n=number of elements in the array and
s=size of each element then

the number of bytes occupied by the array is n*s.

e.g. an ARRAY[1..10] OF INTEGER requires 10*2 = 20 bytes
an ARRAY[2..12,1..10] OF CHAR has 11*10=110 elements and so requires 110 bytes.

A 3.1.5 Sets.

Sets are stored as bit strings and so if the base type has n elements then the number of bytes used is: (n-1) DIV 8 + 1. Examples:

a SET OF CHAR requires (256-1) DIV 8 + 1 = 32 bytes.
a SET OF (blue, green, yellow) requires (3-1) DIV 8 + 1 = 1 byte.

A 3.1.6 Pointers.

Pointers occupy 2 bytes which contain the address (in Intel format i.e. low byte first) of the variable to which they point.

A 3.2 Variable Storage at Runtime.

There are 3 cases where the user needs information on how variables are stored at runtime:

a. Global variables — declared in the main program block.
b. Local variables — declared in an inner block.
c. Parameters and returned values. — passed to and from procedures and functions.

These individual cases are discussed below and an example of how to use this information may be found in Appendix 4.

Global variables

Global variables are allocated from the top of the runtime stack downwards e.g. if the runtime stack is at £B000 and the main program variables are:

```
VAR     i : INTEGER;
        ch : CHAR;
        x  : REAL;
```

then:

i (which occupies 2 bytes - see the previous section) will be stored at locations £B000-2 and £B000-1 i.e. at £AFFE and £AFFF.

ch (1 byte) will be stored at location £AFFE-1 i.e. at £AFFD.

x (4 bytes) will be placed at £AFF9, £AFFA, £AFFB and £AFFC.

Local variables

Local variables cannot be accessed via the stack very easily (so, instead, the IX register is set up at the beginning of each inner block so that (IX-4) points to the start of the block's local variables e.g.

```
PROCEDURE      test;
VAR            i,j : INTEGER;
```

then:

i (integer - so 2 bytes) will be placed at IX-4-2 and IX-4-1 i.e. IX-6 and IX-5.
j will be placed at IX-8 and IX-7.

Parameters and returned values

Value parameters are treated like local variables and, like these variables, the earlier a parameter is declared the higher address it has in memory. However, unlike variables, the lowest (not the highest) address is fixed and this is fixed at (IX+2) e.g.

```
PROCEDURE test(i : REAL; j : INTEGER);
```

then:

j (allocated first) is at IX+2 and IX+3.
i is at IX+4, IX+5, IX+6, and IX+7.

Variable parameters are treated just like value parameters except that they are always allocated 2 bytes and these 2 bytes contain the address of the variable e.g.

```
PROCEDURE test(i : INTEGER; VAR x : REAL);
```

then:

the reference to x is placed at IX+2 and IX+3; these locations contain the address where x is stored. The value of i is at IX+4 and IX+5.

Returned values of functions are placed above the first parameter in memory e.g.

```
FUNCTION test(i : INTEGER) : REAL;
```

then i is at IX+2 and IX+3 and space is reserved for the returned value at IX+4, IX-5, IX+6 and IX-7.

APPENDIX 4 SOME EXAMPLE HP4T PROGRAMS.

The following programs should be studied carefully if you are in any doubt as to how to program in Hisoft Pascal 4T.

```
{Program to illustrate the use of TIN and TOUT.
 The program constructs a very simple telephone
 directory on tape and then reads it back. You
 should write any searching required.}

PROGRAM TAPE;

CONST
  Size=10;

TYPE
  Entry = RECORD
            Name : ARRAY [1..10] OF CHAR;
            Number : ARRAY [1..10] OF CHAR
          END;

VAR
  Directory : ARRAY [1..Size] OF Entry;
  I : INTEGER;

BEGIN

{Set up the directory..}

  FOR I:= 1 TO Size DO
  BEGIN
    WITH Directory[I] DO
    BEGIN
      WRITE('Name please');
      READLN;
      READ(Name);
      WRITELN;
      WRITE('Number please');
      READLN;
      READ(Number);
      WRITELN
    END
  END;

{To dump the directory to tape use..}

  TOUT('Director',ADDR(Directory),SIZE(Directory))

{Now to read the array back do the following..}

  TIN('Director',ADDR(Directory))

{And now you can process the directory as you wish.....}

END.
```

```
 10 {Program to list lines of a file in reverse order.
 20  Shows use of pointers, records, MARK and RELEASE.}
 30
 40 PROGRAM ReverseLine;
 50
 60 TYPE elem=RECORD                              {Create linked-list structure
 70            next: ^elem;
 80            ch: CHAR
 90           END;
100      link=^elem;
110
120 VAR prev,cur,heap: link;                       {all pointers to 'elem'}
130
140 BEGIN
150  REPEAT                                        {do this many times}
160    MARK(heap);                                 {assign top of heap to 'heap'.}
170    prev:=NIL;                                  {points to no varaible yet.}
180    WHILE NOT EOLN DO
190       BEGIN
200         NEW(cur);                              {create a new dynamic record}
210         READ(cur^.ch);                         {and assign its field to one
220                                                 character from file.}
230         cur^.next:=prev;                       {this field's pointer adresses}
240         prev:=cur                              {previous record.}
250       END;
260
270 {Write out the line backwards by scanning the records
280  set up backwards.}
290
300    cur:=prev;
310    WHILE cur <> NIL DO                         {NIL is first}
320      BEGIN
330         WRITE(cur^.ch);                        {WRITE this field i.e. charac
340         cur:=cur^.next                         {Address previous field.}
350      END;
360    WRITELN;
370    RELEASE(heap);                              {Release dynamic variable space
380    READLN                                      {Wait for another line}
390  UNTIL FALSE                                   {Use CC to exit}
400 END.
{Program to show the use of recursion}

PROGRAM FACTOR;

{This program calculates the factorial of a number input from the
 keyboard  1) using recursion  and  2) using an iterative method.}

TYPE
  POSINT = 0..MAXINT;

VAR
  METHOD : CHAR;
  NUMBER : POSINT;

{Recursive algorithm.}

FUNCTION RFAC(N : POSINT) : INTEGER;

 VAR F : POSINT;

 BEGIN
   IF N>1 THEN F:= N * RFAC(N-1)                         {RFAC invoked N times
          ELSE F:= 1;
   RFAC := F
 END;

{Iterative solution}

FUNCTION IFAC(N : POSINT) : INTEGER;

  VAR I,F: POSINT;
  BEGIN
   F := 1;
   FOR I := 2 TO N DO F := F*I;                          {Simple Loop}
   IFAC:=F
  END;

BEGIN
  REPEAT
    WRITE('Give method (I or R) and number    ');
    READLN;
    READ(METHOD,NUMBER);
    IF METHOD = 'R'
         THEN WRITELN(NUMBER,'! = ',RFAC(NUMBER))
         ELSE WRITELN(NUMBER,'! = ',IFAC(NUMBER))
  UNTIL NUMBER=0
END.
```

```
 10 {Program to show how to 'get your hands dirty'!
 20  i.e. how to modify Pascal variables using machine code.
 30  Demonstrates PEEK, POKE, ADDR and INLINE.}
 40
 50 PROGRAM divmult2;
 60
 70 VAR r:REAL;
 80
 90 FUNCTION divby2(x:REAL):REAL;                  {Function to divide by 2 ..
100                                                 .. quickly}
110 VAR i:INTEGER;
120 BEGIN
130  i:=ADDR(x)+1;                                 {Point to the exponent of x}
140  POKE(i,PRED(PEEK(i,CHAR)));                   {Decrement the exponent of x.
150                                                 see Appendix 3.1.3.}
160  divby2:=x
170 END;
180
190 FUNCTION multby2(x:REAL):REAL;                 {Function to multiply by 2..
200                                                 .. quickly}
210 BEGIN
220  INLINE(£DD,£34,3);                            {INC (IX+3) - the exponent of x
230                                                 - see Appendix 3.2.}
240  multby2:=x
250 END;
260
270 BEGIN
280  REPEAT
290   WRITE('Enter the number r ');
300   READ(r);                                     {No need for READLN - see
310                                                 Section 2.3.1.4 }
320
330   WRITELN('r divided by two is',divby2(r):7:2);
340   WRITELN('r multiplied by two is',multby2(r):7:2)
350  UNTIL r=0
360 END.
```

BIBLIOGRAPHY.

K. Jensen and N. Wirth	PASCAL USER MANUAL AND REPORT. Springer-Verlag 1975.
W. Findlay and D. A. Watt	PASCAL. AN INTRODUCTION TO SYSTEMATIC PROGRAMMING. Pitman Publishing 1978.
J. Tiberghien	THE PASCAL HANDBOOK. SYBEX 1981.
J. Welsh and J. Elder.	INTRODUCTION TO PASCAL.

The first and third books above are useful for reference purposes whereas the second and fourth books are introductions to the language and aimed towards beginners.