COMMODORE

TCL PASCAL FOR CBM

Users' Manual

99b-1181

commodore COMPUTER

Warning
Although programs are tested by Commodore before release, no claim is made concerning the accuracy of this Software. Commodore and its distributors cannot assume liability or responsibility for any loss or damage arising from the use of these programs, and programs are sold only on the basis of this understanding. Individual applications should be thoroughly tested before implementation, as programs are standardized in order to offer low cost Software, and therefore may not necessarily fulfill a particular requirement. You are advised to consult your Business Software Dealer for installation, maintenance, and training costs, should you require these services. Any changes to the Software not recommended by Commodore may lead to the withdrawal of support services.

Hardware Support
You are advised to seek a Field Maintenance contract with your dealer before implementing this package.

Program Security ROMs
This program is accompanied by a Commodore Program Security ROM. The end-user need not concern himself with this, as it will already have been installed in his machine by his Commodore Dealer before delivery of the Software is completed.
For the information of the dealer, however, this ROM must be placed in the middle (A000-AFFF hexadecimal) open ROM slot of the three normally available on the 32K CBM logic board assembly. This ROM must always be installed by the dealership, and not by the end-user.

NOTE
This manual was prepared, edited, and typed by the COMWORDPRO III program product.

Diskette Care

Mini-diskettes appear to be reasonably tough, but are in fact very delicate and require careful handling. They must at all times be either in a disk drive or in their protective envelope. Diskettes left lying about stand a very good chance of never functioning again.
The magnetic surface should never be touched - note that contrary to popular belief, programs and data are recorded on the UNDERSIDE of the diskette.
The environment of the diskettes should not have a temperature below 10 C (50 F), or above 52 C (125 F).
The diskette should never be bent or flexed.
Insertion of a diskette into a drive should always be done in a gentle, cautious manner.
Diskettes should always be kept away from all magnetic fields (e.g. Electric motors, high current cables, etc...).

Initialization

Users may sometimes note difficulties in initializing diskettes - in that the middle error light (L.E.D.) may go on. This problem is most frequently caused by corruption of the diskette, due to lack of proper care when handling it. On very early Commodore disk drives, however, this also occasionally happens due to improper alignment of the diskette within the drive. The recommended procedure in this event is to leave the door or flap of the disk drive open, closing it only when the initialize function has actually selected the drive and lighted the L.E.D.

commodore COMPUTER

CONTENTS

I. Introduction to TCL Pascal 8

II. Beginner's Guide to Pascal 11

1. Getting Started 11

WRITE statements, string 12
Integer arithmetic. +,-,*,DIV,MOD 13
Functions: ABS, SQR, ODD 14
Boolean expressions >,<,>=,<=,<>,=,AND,OR,NOT 15

2. Pascal Statements 17

Variables and assignment statements 18
FOR statement 18
IF statement 19
REPEAT and READ statements 19
CASE statement 21
Error messages and error correction 22
WHILE statement 24

3. More Variable Types 25

Real numbers 25
Real arithmetic. /, SQRT, SIN, ARCTAN, LN, EXP, ROUND, TRUNC. 25
Output formatting and constants 26
Characters 27
CBM graphics 27
Arrays 28
Enumerated types and subranges ORD, PRED, SUCC 29
Sets 30

4. Procedures And Functions 32

Procedures 32
VAR parameters 33
Functions 33
Recursion 34
Textfiles 34
Strings 36

5. Advanced Features 38

Records 38
Pointers and lists 39
GOTO statement 41
Extensions 41

6. Disk-based Operation 42

7. Editor Command Summary 44

8. Error Messages 51

9. Sample Programs 54

III. TCL Pascal Reference Manual 59

1. General 59

1.1 Pascal keywords 59
1.2 Pascal Identifiers 59
1.3 Other Special Symbols 60
1.4 Comments 60
1.5 Constants 60
integer 60
real 60
character and string 61
1.6 Blanks 61

2. Data Types and Operators 62

2.1 Integer 62
2.2 Real 62
2.3 Char 63
2.4 User-defined (enumerated) Types 63
2.5 Subrange Types 64
2.6 Boolean 64
2.7 Operator Precedence 64
2.8 Summary of Arithmetic
and Conversion Functions 66

3. Pascal Declarations and Statements 67

3.1 Pascal Programs 67
3.1.1 Constant Declarations 67
3.2 Type Declarations 68
3.3 Variable Declarations 68
3.4 Executable Statements 68
3.5 Assignment Statements 69
3.6 Compound Statements 69
3.7 "If" Statements 69
3.8 "Repeat" Statements 70
3.9 "While" Statements 70
3.10 "For" Statements 70
3.11 "Case" Statements 71
3.12 "Goto" Statements 71
label declarations 72

4. Input and Output of Text 73

4.1 Outputting to Textfiles 73
4.2 Inputting from Textfiles 73
4.3 Reading Other Data Types from Textfiles 74
4.4 Writing Other Data Types to Textfiles 74
4.5 Abbreviations 75
4.6 Manipulating Files 76

5. Structured Data Types 77

5.1 Arrays 77
5.2 Sets 78
5.3 Records 79
5.4 Packed Structures 80
5.5 Pack and Unpack 81

6. Functions and Procedures 82

6.1 Function and Procedure Definitions 82
6.2 Procedure and Function Calls 82
6.3 Parameters 83
6.4 Local Declarations 85
6.5 Recursion and Forward References 86

7. Dynamic Storage and Pointers 87

7.1 Pointers 87
7.2 "New" and "Dispose" 87

8. Disk Files 89

8.1 Declarations 89
8.2 Sequential Writing 89
8.3 Sequential Reading 90
8.4 External Files 90
8.5 Reading and Writing from other Devices 91
Disk textfile example 91

9. Extensions to Standard Pascal 93

9.1 Hexadecimal Constants 93
9.2 Memory VDU and port access 93
9.3 Hexadecimal Input and Output 94
9.4 Bit Manipulation 95
9.5 Catching I/O Errors 95
9.6 Keyboard Interrupts 96
9.7 Random Number Generator 96
9.8 Underscore 96
9.9 The CBM Internal Clock 96
9.10 Input of String Variables 97
9.11 Program Chaining 97

10. CBM Pascal Interface Guide 99

10.1 Assembly Language Format 99
10.2 Storage Formats 100

I. Introduction to TCL Pascal by Keith Frewin of Transam

PASCAL is a powerful high level computer language written by Niklaus Wirth of Zurich, Switzerland.*

It can be efficiently implemented on small computers as well as large mainframes, offering numerous advantages over other popular microcomputer languages such as BASIC.
Some of these advantages are:

ALGOL-like block structure

Meaningful variable names

Powerful data structuring techniques

User-defined data types and constants

Excellent function and subroutine linkage

Recursive calls

Clean, modern flow of control

Runtime error checking

Dynamic variable allocation

Greater standardisation

High speed of execution

Greater program legibility

TCL Pascal is an implementation of standard PASCAL designed specially for small computers with as little as 32K bytes of memory. It offers all the features of this powerful language together with some useful enhancements for the personal computer user.

The CBM version has two modes of operation. In the simplest mode the Pascal compiler co-resides in RAM with the user's program. This is ideal for learning the language or writing small programs which do not need the disk. Most Pascal commands are available in this mode except those involving diskette files. For more complex programs the disk-based compiler can be used to give the full power of the language.

* PASCAL USER MANUAL AND REPORT BY JENSEN AND WIRTH
---Springer-Verlag 1975

Hardware requirements

CBM Professional computer with 32K RAM and BASIC 2.0, plus a model 3040 floppy disk unit with DOS 1.0.

Some implementation information

MAXINT = 32767

type INTEGER = -32768...32767

type CHAR = the ASCII set (Extended to include CBM graphics)

set values: must be in 0..127(therefore set of char must be between chr(0)..chr(127))

real numbers: accuracy:9 digits
range: approx 1E-38 to 1E38

default output formats: integer : 7 characters
real : 12 characters
boolean : 6 characters
char : 1 character
string : size of string

program size and complexity: No restriction, apart from exceeding the total memory capacity of the system (STACK OVERFLOW is printed)

identifiers: first 8 characters must be unique

labels: first 8 digits must be unique

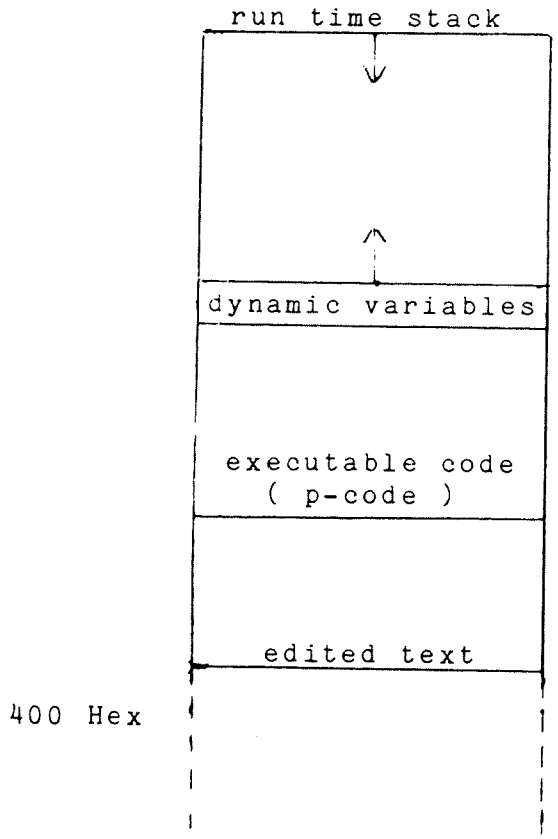
Extensions to standard Pascal

Dynamic specification of filenames
Input of strings
Hexadecimal numbers and hex I/O
Bit manipulation
Machine language interface
Memory and VDU screen access
Run-time I/O error detection
Random number generator
Program chaining
CBM clock interface
Separate compilation (linking)

PASCAL RAM USAGE

This is automatically optimised to use all available memory

Top of RAM



"Stack overflow" is printed when all memory has been exhausted.

II Beginner's Guide to Pascal

This section is a straightforward introduction to some of the features of Pascal.

1. Getting Started - Write Statements

Turn your computer on. It should be displaying the message.

```
### COMMODORE BASIC ###
    31743 BYTES FREE
READY.
```

To run Pascal, turn the disk unit on, insert the Pascal diskette in the right-hand drive (drive 0), and then type:

```
DLOAD "*",8 D0
```

followed by a carriage return. The computer should reply with:

```
SEARCHING FOR *
LOADING
READY.
```

This loads the first program on the diskette into memory, which should be the Pascal system. Now you just type:

```
RUN
```

followed by a carriage return. You should get a clear screen with the PASCAL SIGN-ON MESSAGE:

```
TCL Pascal
=== ======

(c) Copyright Transam 1980.
loading...
ready.
```

Modern computers can be very powerful, but they need to be "told" what to do by means of PROGRAMS. Computers work in a language of numbers called MACHINE LANGUAGE, but machine languages are generally quite difficult for humans to master, and they differ widely depending on the particular computer you are using.
It's much easier to talk to the computer in a HIGH LEVEL LANGUAGE such as COBOL or Pascal. These languages somewhat resemble English, but have stricter rules of grammar to prevent ambiguities.

Pascal was invented by Niklaus Wirth, of Zurich, Switzerland in

1968. (It is named after the 17th century French mathematician Blaise Pascal). Pascal is an ideal language for learning to write computer programs. Your Pascal programs are automatically translated by the computer into a machine language which it can interpret.

Let us start right away with a very simple programming example:

Example 1

First of all you must enter the program into the computer memory, and this is done using the EDITOR.

Type in the first line of the program, shown below. Follow it with the carriage return key (referred to in this manual as <return>):

```
10 begin
20 write ('Hi there !')
30 end.
```

Having typed in the first line, the editor should automatically prompt you with the next line number; which you should not re-type:

```
20
```

These line numbers have no significance in Pascal - They are purely for use by the editor, and they will be assumed in all future examples. Remember, if you make a mistake in typing the program you can correct it by using the screen editing commands INST, DEL, cursor up, cursor down, cursor left and cursor right, just as in BASIC. Pressing the DEL key, for example, will erase the last key you typed. See the CBM User's manual for a complete description.

Now enter the remaining two lines of the program, being especially careful about punctuation and spelling, and don't forget that full stop at the end!

```
write ('Hi there')
end.
```

When you have finished, type a blank line (just <return>) to turn off the line numbers.

All you have to do to run your program is to type

```
r<return>
```

If all goes well the computer should reply with something like the following:

```
Compiling
Program    0  0509
0 error(s)
Compilation complete.
```

Do not worry about the details, but what is happening is that the computer is scanning your program and converting it into a numeric form which it can efficiently execute. (If you don't get the message: "0 error(s)", then you probably made a mistake in typing. You could try typing "new <return>" and starting again!).
Now the computer should automatically run your program, and print the message:

```
Hi there!
```

Once your program has been compiled, it can be run as many times as you like by typing:

```
r<return>
```

Each time the computer should print:

```
Hi there!
```

Now let us look at the program in more detail. The main body of a Pascal program is always enclosed between the words BEGIN and END, the final END must be followed by a full stop. Pascal programs consist of a sequence of "statements" which are executed sequentially in the order they are written. Example 1 has one statement, a WRITE statement which tells the computer to write something on the screen, in this case the message "Hi there!". The object enclosed in the single quotes is called a STRING, and may contain any sequence of characters except <return>. Also, if a single quote is itself to be included in a string, it should be doubled up, so that the Pascal program

```
begin
write ('O''Brien''s string')
end.
```

would cause the message

```
O'Brien's string
```

to be printed on the screen.

Example 2

Other things can be printed besides strings. Try the following program. We will use the same steps as example 1 but we must remember to erase example 1 from the computer memory. So type:

```
new <return>
```

now type example 2 into the computer:

```
begin
write (3 + 4) ;
write (6 -  2  -  1)
end.
```

followed by

```
r<return>   (to compile and run the program)
```

When the program is run the computer should print

```
 7      3
```

Example 2 contains two statements, which must be separated by a semicolon. It has examples of INTEGER (whole number) arithmetic.

Now try the next example:

Example 3 multiplication and division

```
begin
write (6 * 7, 18 div 4, 18 mod 4, -(4 + 2) * 3)
end.
```

The computer should print

```
42      4       2     -18
```

In Pascal "*" means multiplication, DIV means integer division (ie with rounding towards zero), and "18 MOD 4" gives the remainder when 18 is divided by 4.

Note how brackets have been used to change the order of evaluating -4 + 6, or 2. This is because the computer does multiplications and divisions before it does additions and subtractions.

Any number of items can be printed using a single WRITE statement, provided that they are separated by commas.

Example 4 functions

```
begin
write (sqr (4 + 5), abs (- 44), abs (44), odd (3))
end.
```

The computer should print

```
81      44      44  TRUE
```

SQR, ABS and ODD are called "functions". There are many different functions in Pascal.

SQR, followed by a number in brackets, gives the square of the number.

ABS gives the absolute value of the number.

ODD (3) is TRUE because 3 is odd.

The last function, ODD, gives a Boolean, or logical result, that is it can either be TRUE or FALSE. Boolean values are used a lot in Pascal so let us look at them more closely.

Example 5 Boolean expressions

```
begin
writeln  (true, false, 3 = 3, 3 = 4);
write    (3<>4,  5<6, 9 >=10);
end.
```

Should print:

```
  TRUE FALSE  TRUE FALSE

  TRUE  TRUE FALSE
```

because: 3 is equal to itself
3 is not equal to 4
etc.

```
=  means  "equal to"
<  means  "less than"
>  means  "greater than"
>= means  "greater than or equal to"
<= means  "less than or equal to"
<> means  "not equal to"
```

WRITELN is like WRITE but also generates a new line after printing all the values in brackets.

Example 6 Boolean expressions

These can get a bit complicated, but the computer evaluates them using the rules of logic.

```
begin
write  ((3 = 3) and (3<5),(3 = 4) or (3>11));
write (not true, not false, not (1 = 2));
end.
```

Gives the result:

```
  TRUE FALSE FALSE  TRUE  TRUE
```

because both (3 = 3) and (3<5) are true, neither (3 = 4) nor (3>11) are true, and (1 = 2) is false so not (1 = 2) is true.

"x and y" is TRUE if both x and y are TRUE

"x or y" is TRUE if either x or y (or both) are TRUE

"not x" is TRUE if x is FALSE, and FALSE if x is TRUE

2. Pascal statements

First a word about symbols. These are the building blocks of Pascal programs, and there are three main kinds:

1. Pascal keywords, such as BEGIN and END, which are reserved and can't be altered by the user. A complete list of these is given in the reference manual, section 1.1.

2. Special symbols such as . ; := .. <> etc.

3. Identifiers, which are names chosen by the user. They can be any sequence of letters or digits, but must start with a letter. For example:

```
i
Henrythe8th
PI
```

WARNING Identifiers are unique only if they differ in the first 8 characters, so that Henrythe7th and Henrythe8th are the same identifier in TCL Pascal (and many other implementations).

Upper case letters are equivalent to their lower case counterparts so that PI, pi and Pi are all synonymous.

Some standard identifiers such as WRITE and WRITELN are predeclared in every version of Pascal.

These can be redefined by the user, however (in contrast to Pascal keywords).

IMPORTANT Pascal symbols can't contain imbedded blanks. "Henry the 8th" is not the same as "Henrythe8th", and "30 000" is not equivalent to the number 30000. ("30,000" would also be illegal). Note especially that ": =" cannot be used instead of ":=".

This aside, spaces, tabs and new lines may occur anywhere in a Pascal program, and are ignored.

Now we return to some actual examples of Pascal programs. Indentation is used by putting spaces in front of certain lines. This is optional, but helps to make the program clearer to humans.

Example 7 Variables and assignment.

```
var x,y :integer;
begin
  x:=3; y:=27;
  writeln (x,y);
  x:=4;
  y:=x+2;
  write (x,y, x+y)
end.
```

Should print:

```
 3      27
 4       6      10
```

The VAR declaration comes before the BEGIN, and informs the compiler that the identifiers x and y are "variables" which can take integer values. As the name implies, variables can change in value throughout the execution of the program. In line 3, the value of x is set to 3 and the value of y is set to 27. Then later, x is set to 4 and y is set to x + 2, or 4 + 2 = 6. Notice that y:=y+2 could also have been written setting y to 27 + 2 = 29. Variables can also be declared as BOOLEAN and many other types besides INTEGER.

Example 8 repetition using "FOR" loops.

```
var i : integer;
begin
  writeln ('going up');
  for i := 1 to 5 do writeln (i);
  writeln ('going down');
  for i := 5 downto -1 do writeln (i);
end.
```

Should print:

```
going up
       1
       2
       3
       4
       5

going down
       5
       4
       3
       2
       1
       0
      -1
```

-1

The statement following the "FOR .. DO" (in this case a WRITELN statement) is repeated once with each value of the variable i.

Example 9 "if" statements

```
var i : integer;
begin
  for i:= 1 to 11 do
   begin
    write (i);
    if odd (i) then writeln (' is odd')
    else writeln (' is even');
   end
end.
```

The result should be:

```
 1 is odd
 2 is even
 3 is odd
 4 is even
 5 is odd
 6 is even
 7 is odd
 8 is even
 9 is odd
10 is even
11 is odd
```

"if" statements give the computer a choice of two statements to do, depending on the value of the Boolean expression. (Remember, Boolean expressions can either be TRUE or FALSE). The "else" part of a conditional statement is optional but --IMPORTANT-- "else" is never preceded by a semi-colon.

The WRITE and IF statements in our example are enclosed in BEGIN...END to make them act as a single statement to be repeated by the FOR loop.

Example 10 finding the average

This example introduces keyboard input, and a more general sort of loop.

```
var total, count, x : integer;
begin
  total:=0; count:=0;
  write ('Type some numbers: ');
  repeat
    read (x);
    total:=total+x;
    if x>0 then count:=count+1;
  until x=0;
  writeln ('The average is', total/count);
end.
```

When you run this program, the computer should invite you to type a series of numbers. Try typing:

3 47 5 199 0 <return>

The computer should reply with

The average is 6.35000E+01

The statement read (x) tells the computer to accept an integer from the keyboard and place its value in the variable x. If you type something the computer doesn't recognise as an integer, you might get the message

INTEGER READ ERROR line 60

and the program will terminate.

The "/" operator gives division with a floating point, or REAL result (whereas DIV gives an integer result). More about REAL arithmetic later.

The "real" number was printed in "scientific" notation, which will be familiar to many calculator users. The number following the "E" represents a power of 10, so that 6.35E+01 means "6.35 (times 10 to the power of +1) or 63.5.

The printing format can be changed to make it more legible by specifying the total number of characters you would like printed and the number of digits after the decimal point. (Rounding is done automatically). Thus:

Writeln ('The average is', total/count : 10 : 3)

would have printed

The average is 63.500

The number is printed with 4 leading blanks to give the 10-character

field you specified.

The "repeat"..."until" loop simply executes the enclosed statements until the condition at the end turns out to be TRUE. In our example the loop is terminated when a zero is read from the keyboard.

Example 11 Case statement

This example introduces a slightly more elaborate way of choosing one of several statements:

```
var verse, i : integer;
begin
  for verse:=1 to 4 do
    begin
      writeln;
      for i := verse downto 0 do
       case i of
        3: writeln('three men');
        2: writeln('two men');
        1: writeln('one man');
        0: writeln('and his dog')
       end
    end
end.
```

This should result in the printout:

```
one man
and his dog

two men
one man
and his dog

three men
two men
one man
and his dog

case error line 70
```

The error message was caused because in the last verse i becomes 4 and there is no corresponding label in the CASE statement.
Case labels can also be combined, for example

```
4,5,6 : writeln ('Many men');
```

A note on error messages

The "CASE ERROR" message is called a "runtime" error message because it occurs while the program is actually running. There are several such messages which you may encounter (see section 8).

By now you may have started to experiment with your own programs. (This is probably the best way of finding out what is and is not possible in Pascal). If so, you will sooner or later get one of the compiler's error messages. This may also happen if you make a mistake in typing one of the examples. The simple program

```
var x : boolean;
    x : integer;
begin
  read (x);
  write (x)
end.
```

would cause the following message during compilation:

```
compiling
---ERROR TYPE 46 LINE 20 NEAR X
...IDENTIFIER DECLARED TWICE
program  0    050d
1 error(s)
Compilation complete
```

The error number given was 46. There are over 100 possible errors, and so an appropriate message is selected by the compiler from a disk file (PASCAL ERROR MSG), in this case: "Identifier declared twice". Referring to line 20 of the program, that's exactly what we have done.

NOTE---the line number may sometimes be out by 1 line or even more, depending on how long the compiler needed to detect the error.

Typing "L" in response to the prompt:

```
ready
```

displays the program on the VDU as well as compiling it. All the line numbers and errors are marked. In our example:

```
 10 var x :boolean;
 20     x <- ERROR 46
..IDENTIFIER DECLARED TWICE
 20     x : integer;
 30     begin
 40       read (x);
 50       write (x)
 60     end.
```

The full version of line 20 is retyped underneath the error report.

The computer will not let you run a program if there are any compiler errors.

Correcting errors

This can be done using the editor, without having to retype the whole program. To correct the small example above you might type:

```
list
```

to list the program on the screen:

```
 10 var  x : boolean;
 20      x : integer;
 30 begin
 40   read (x);
 50   write (x)
 60 end.
```

To delete the second line, type 20 followed by <return>. Typing "list" again should give:

```
 10 var  x : boolean;
 30 begin
 40   read (x);
 50   write (x)
 60 end.
```

Line 10 is still wrong. We want to read and write integer, so we type

```
change /boolean/integer/
```

which makes the substitution and retypes the line.

"list" now should print the correct version of the program.

```
10 var  x : integer;
30 begin
40   read (x);
50   write (x)
60 end.
```

The program doesn't do much, just reads a number from the keyboard and prints it out again.

For a more complete explanation of how the editor works, see the command summary (section 7). This also explains how to load and save your Pascal program on diskette.

while statement

There is another sort of loop in Pascal, besides REPEAT and FOR loops.

The WHILE statement is like the REPEAT statement except that the test is done at the beginning of the loop (so that the loop need not be executed at all). Also, like the FOR loop only one statement may be repeated (or a sequence of statements enclosed in BEGIN and END).

Example:

```
i:=1;
while i <= 5 do
  begin
    writeln (i);
    i:=i+1;
  end;
```

Has the same effect as

```
for i:=1 to 5 do writeln (i);
```

3. More about data types in Pascal

Example 12 Floating point numbers.

```
begin
  writeln (3.3, 33.0, 330.0, 0.33);
  writeln (-3.3E3, 3.3E-1, 4.5+2.1)
end.
```

The computer should print

```
 3.30000E+00 3.30000E+01 3.30000E+02 3.30000E-01
-3.30000E+03 3.30000E-01 6.60000E+00
```

The presence of either a decimal point or an exponent (the "E" part) in a number tells Pascal to treat it as a floating point or a REAL number.

3.3E3 means 3.3 times (10 to the power of 3)
in other words 3.3 x (10x10x10) or 3300

Floating point numbers in Pascal have an accuracy of 9 digits and may range in size from about 1E-38 to 1E38. In contrast to integer, you should not expect Pascal real arithmetic to be exact. This means, for example, that 4.0 may in fact be printed as 3.999999. Also, you can't rely on testing real numbers for equality. 2.0 + 2.0 = 4.0 may not always be true!

Example 13 floating point arithmetic

```
var x, y: real;
begin
    x := 9.1;
    y := 8.7;
    writeln (x+y: 7:2, x-y: 7:2, x*y: 7:2, x/y: 7:2);
    writeln (sqr (x) : 7:2, sqrt (x): 7:2, abs (x): 7:2);
    write (trunc (x), trunc (y), round (x), round (y))
end.
```

Should print:

```
17.80   0.40  79.17   1.05
82.81   3.02   9.10
    9      8      9      9
```

We have already met +, -, * and /. They are used to mean addition, subtraction, multiplication and floating point division (DIV means integer division. DIV and MOD shouldn't be used with reals).

```
SQR (X) means the square of X
SQRT (X) means the square root of X
ABS (X) gives the absolute value of X
TRUNC (X)  gives the integer (whole number) part of X
ROUND (X) rounds X to the nearest integer.
```

Some other useful mathematical functions are

```
SIN (X) gives the sine of X (X is in radians)
COS (X) gives the cosine of X (X is in radians)
ARCTAN (X) gives the angle whose tangent is X (in radians)
LN (X) gives the natural logarithm (base e) of X (for X > 0)
EXP (X) gives the number e raised to the xth power.
```

1 radian = 57.29578 degrees

e= 2.718281

Example 14 Output formatting, constants.

```
program waves;
const f1 = 0.5; f2 = 0.05; amplitude = 19;
var x1, x2, y :real;
begin
    x1:=0;  x2:=0;
    repeat
      x1:=x1 + f1;
      x2:=x2 + f2;
      y:=sin (x1) * sin (x2) * amplitude;
      writeln ('x': round (y) + amplitude)
    until false
end.
```

The program should print an amplitude - modulated sine wave.

Because of the REPEAT..UNTIL FALSE loop, example 14 will continue printing almost forever (at least until x1 or x2 becomes too large!) One way of stopping it would be to turn off the power, but if you did that you would lose the program. A better way is simply to press the STOP key.
The computer should print:

```
BREAK AT LINE xxxxxx
```

Where xxxxxx is the line it happened to be executing when you pressed STOP. (If this doesn't happen, try again)

The "Program" header is optional in TCL Pascal and in this case simply serves to give the program a name: WAVES. The name has no significance to the computer; it's merely there as an aid to documentation.

Any text enclosed between the pairs of symbols (* and *) is also ignored by the compiler. This facility can be used to write comments which help human readers to understand the program. Constants, introduced by the keyword CONST, are values which don't change throughout the program. It is an error to use a constant on the left of an assignment statement or as a parameter in the READ statement.

"CONST" declarations are useful for giving names to special values (for example PI = 3.1415926), and they make the program easier to change later. Try using the editor to alter the frequencies f1 and f2 and the amplitude to give different wave patterns in example 14.

Note how the program uses a field width specification (a colon followed by an integer value) to tell the computer how many characters to allocate to the 'x' when printing. If too many characters are asked for, enough spaces are printed to make up the difference. If not enough are asked for, the string is truncated on the right, for example

```
write ('Hi there' :5)
```

would print:

```
Hi th
```

Numeric values, however, are always printed in full even if too few characters are specified.

Example 15 CBM graphics

```
var line, i: integer;
begin
    page;
    for line:=1 to 24 do
      for i:=1 to 40 do
        if odd (line) and odd (i) or
         not odd(line) and not odd (i) then write(chr(177))
        else write(chr(178))
end.
```

This should fill the screen with a pattern. CHARACTERS in Pascal are strings of length 1, for example:

```
'x'  '?'  ''''
```

They belong to the data type "Char", which has 256 possible values in TCL Pascal, corresponding to the ASCII character set (see the CBM user's manual, page A-13).

The function ord (ch) gives the ASCII integer code (between 0 and

255) for the character ch, while chr (x) gives the character represented by the integer x. So ord ('?')=63 and correspondingly chr(63)='?'.

NOTE - On the CBM, the TCL Pascal data type "char" has been extended to the range 0..255 to allow the CBM graphics font to be used. Two of these characters were used in the program above. Try writing programs to give different patterns, using the available characters. A simple change would be replacing 177 and 178 by 173 and 174. The statement "page" simply clears the vdu screen.

Example 16 Arrays

Suppose you wanted to read in some numbers and print them out in reverse order. You would have to store the numbers somewhere because you can't start printing until the last number has been read. If you knew that there were always going to be three values, you could write:

```
var x1, x2, x3 :integer;
begin
    write ('Type 3 numbers : ');
    read (x1, x2, x3);
    writeln (x3);
    writeln (x2);
    writeln (x1)
end.
```

But for 50 values this would get a bit tedious!

The answer is to use an array variable:

```
const n=3;
var x : array [1..n] of integer;
    i : integer;
begin
    write ('Type ', n:1, ' numbers: ');
    for i:=1 to n do read (x[i]);
    for i:=n downto 1 do writeln (x[i])
end.
```

Running the program and typing the data:

```
463     79     980
```

Should give the result:

```
980
 79
463
```

The declaration of x really declares n variables which can be referred to by giving an index in square brackets. The elements of the array x are thus x[1], x[2], ..., x[n].

The constant n was used so that the number of values read in by the program can easily be changed by altering just one line.

Array elements can be any valid Pascal data type, including another array. This allows two dimensional (or indeed any dimensional) arrays, and a chessboard for example may be represented as:

```
var chessboard: array [1..8] of array [1..8] of chesspiece;
```

Where chesspiece is some suitable data type, probably a user defined type (more about this later). The 5th square of the 3rd row of the chessboard could then be referred to as:

```
Chessboard [3] [5]
```

Because arrays of arrays are used often in Pascal programs, the abbreviation "chessboard [3,5] " is allowed, and similarly in the declaration:

```
var chessboard : array [1..8, 1..8] of chesspiece;
```

This can be extended to arrays of any dimension.

Defining your own data type

None of the data types so far mentioned (integer, real, boolean, or even char) would be really suitable for describing a piece on a chessboard, so Pascal lets you define your own. This may be done in a TYPE DECLARATION, for example:

```
type chesspiece = (pawn, knight, bishop, rook, queen, king);
```

Then a variable of type CHESSPIECE could take any of these six values, for example:

```
var mypiece, yourpiece:chesspiece
begin
    :
    :
    mypiece:=rook;
    yourpiece:=queen;
    :
```

Type declarations come after constant declarations and before variable declarations. The identifiers used in an 'enumerated' data type like CHESSPIECE must be unique, they can't appear in other enumerated types or be declared as constants or variables.

Enumerated types are ordered so that our chess pieces can be compared using =, > etc:

```
king>queen
queen>rook
```

and so on:

```
:
```

Three functions are also defined: PRED, SUCC and ORD

pred (x) gives the value preceeding x.

succ (x) gives the value succeeding x.

ord (x) gives the position of x within the data type. (starting with pawn = 0)

so pred (bishop) = knight

succ (rook) = queen

but pred (pawn) and succ(king) are both meaningless

ord (knight) = 1

ord (rook) = 3, and so on.

Example 17 The sieve of Eratosthenes.

This program finds and prints all the prime numbers between 2 and 127.

```
program Eratosthenes;
const n=127;
var sieve : set of 2..n;
    number, i : integer;
begin
    sieve := [2..n];
    for number := 2 to n do if number in sieve then
     begin
      writeln (number);
      for i := 2 to n div number do
                  sieve := sieve - [i*number]
     end
end.
```

A prime number is divisible only by itself and 1. Our "sieve" used for finding the prime numbers, is a new type of variable called a SET variable.

Sets in Pascal are collections of objects enclosed in square

brackets. Either an object is in a set or it is not, so

```
[1,2,3]
[2,3,1]
```

and `[1,1,3,3,2]` are all equivalent.

The abbreviation x..y in a set means all the items between x and y inclusive, so

```
[1..4, 10] = [1,2,3,4, 10]
```

We can test whether an item is in a set by using the operator IN. Thus "4 in [1..5]" will give the Boolean result: TRUE.

The type of a set can be any scalar type (ie not an array or a set) except REAL. Values are restricted to the range 0..127 (so "set of char" from chr(0) to chr(127)) is acceptable).

Now back to our sieve program. Starting with the number 2 and working upwards, if a number is still in the sieve then it's a prime. We simply eliminate all multiples of that number from the sieve because they are not prime. Operations allowed on two sets x and y are:

`x + y` which gives the set of all items present in either x or y or both.

`x - y` which gives all items in x which are not also in y.

`x * y` gives all items present in x and also present in y.

`x = y` tests if two sets are equal.

`x<>y` tests if two sets are not equal.

`x<=y` tests if all items in x are also in y.

`x>=y` tests if all items in y are also in x.

4. Procedures and functions

Example 18 procedures

```
var ch: char;
procedure lineof (wotsit :char);
    var i: integer;
    begin
      for i:=1 to 30 do write (wotsit);
      writeln
    end; (* of procedure "lineof" *)
begin (* of main program *)
    lineof ('?');
    writeln;
    for ch:= 'a' to 'f' do lineof (ch)
end.
```

you don't need to type the comments (* ...*) if you don't want to. These are there to help explain the program.

The computer should print:

```
?????? ..........

aaaaaa ..........
bbbbbb ..........
cccccc ..........
dddddd ..........
eeeeee ..........
ffffff ..........
```

Procedures are used to separate sections of code from the main program, either to make what the program does clearer by dividing it up functionally, or to allow the same code to be "called" from various parts of the program. The procedure "lineof" has a PARAMETER "wotsit" (which takes the data type CHAR). When lineof is called it must be followed by a corresponding actual parameter in brackets. Then "lineof" simply writes a line of wotsit's on the screen.

If a procedure has no parameter then the brackets are omitted. The variable I is "local" to the procedure lineof, the main program doesn't know about it. However lineof could if necessary access the 'global' variable CH. Using local variables helps conserve storage, since they are destroyed when the procedure finishes. Procedures are really mini-programs in their own right. They can have their own constant and data type declarations and even their own procedures.

"WOTSIT" is called a VALUE parameter because a value is substituted for it when the procedure is called. Lineof could change the value of wotsit without affecting the main program. VARIABLE parameters on

the other hand are substituted with variables when the procedure is called.

Example 19 Variable parameters

```
var x,y :integer;
procedure swap (var a,b : integer);
    var temp: integer;
    begin
      temp:=a;
      a:=b;
      b:=temp
    end;
begin
      x:=4;  y:=77;
      writeln (x,y);
      swap (x,y);
      writeln (x,y)
end.
```

This should give the result

```
 4      77
77       4
```

Note that it is alright to have local variables, constants and parameters with the same names used in the main program. For example

```
procedure swap (var x,y : integer);
```

The computer won't get confused (but you might!). The variable parameters a and b are used by SWAP as a means of returning a result to the main program. Another way of returning a value is to define a function.

Example 20 Defining a function

```
var i : integer;
function cube (x : integer) : integer;
    begin
         cube := x*x*x;
    end;
begin
    for i:= 1 to 20 do writeln ('The cube of ',
                      i : 2,' is', cube (i))
end.
```

The program should print some numbers and their cubes. Apart from having to specify a return value, functions are just like procedures.

Example 21 Recursion

A recursive function or procedure is one that calls itself. Using recursion can give neat solutions to mind-bending problems like the "Towers of Hanoi". In this well known puzzle, there are three piles of discs. To start with piles 2 and 3 are empty, and the first pile has a number of discs stacked in order of size, smallest at the top. The game is to get all the discs in the same order (smallest on top) over to the 3rd pile, moving only one at a time, with no disc ever resting on a smaller disc.

```
program Hanoi;
var ndiscs: integer;
procedure move (source, destn, spare: 1..3; n:integer);
begin
    if n>1 then move (source, spare, destn, n-1);
    writeln('Moving from', source : 2, ' to ', destn : 2);
    if n>1 then move (spare, destn, source, n-1)
end;
begin
    write ('How many discs ? ');
    read (ndiscs);
    writeln;
    move (1,3,2,ndiscs)
end.
```

Moving one disc is trivial. To move n discs we first move the top (n-1) to the spare pile and then move the bottom one. Then the top (n-1) are moved using the same technique.

Recursive programs are not always the most efficient, though. They tend to gobble up memory because the computer has to save the variables for each call on the stack. If you make ndiscs too large the computer will run out of memory and print STACK OVERFLOW -- line xxxx. The same will happen if you declare more variables in a program than you have memory available, or if you try to compile too large a program.

Text Files

Text files are special Pascal variables having the data type TEXT which are essentially streams of characters with no fixed size. Two are preassigned in PET Pascal. "Input" and "output" are associated normally with the keyboard and the VDU display respectively.

By default, "Input" is implied in READ, READLN, EOLN and EOF and "Output" is implied in WRITE, WRITELN and PAGE. So for example,

```
EOF is really short for EOF (INPUT)
WRITELN ('Hi'!) is really short for WRITELN (OUTPUT, 'Hi!')
```

Each textfile has an associated buffer variable of type CHAR (the file name followed by an upward arrow), for example: input↑

The procedure call:

get (input) reads the next character from the keyboard and puts it in the variable input↑

put (output) writes the contents of output↑ to the VDU. So if X is a character variable:

```
read (x) is equivalent to x :=input↑; get (input)
write (x) is equivalent to output↑ :=x; put (output)
```

Newlines are special characters in textfiles. When a file buffer contains one, assignments like

```
x := input↑
```

will set x to a space. Also, the end of line function EOLN will return TRUE.

READLN is like READ but afterwards skips to the beginning of the next line by doing:

```
while not eoln do get (input);
get (input);
```

This is awkward for interactive programs because the next line of input must be typed before the program can proceed, since input↑ is supposed to contain the first character of the next line. It's better to use READ and skip any leading spaces before you read the next value. (This is done when reading numeric values anyway).

Initially when the program is run, input↑ contains the newline you typed at the end of the "RUN" command.

Example 22 strings

```
program revwords;
const linesize = 64;
type string=packed array [1..LINESIZE] of char;
var word:string;
    nchars, i :integer;

procedure skipblanks;
    begin while (input↑ = ' ') and not eoln do
        get (input) end;

procedure swap (var s:string;i,j:integer);
var temp:char;
begin
    temp:=S[i];
    S[i]:=S[j];
    S[j]:=temp
end;

begin
  repeat if eoln then readln;
    skipblanks;
    while not eoln do
      begin
        nchars:=0;
        repeat
          nchars:=nchars+1;
          read (word [nchars]);
        until input↑ = ' ';
        for i:=1 to nchars div 2 do
          swap (word,i,nchars -i+1);
        write (word:nchars, ' ')
      end;
    writeln;
  until false
end.
```

The program reads words and writes them out with the letters reversed. For example:

```
Mary had a little lamb <return>
```

would print,

```
yraM dah a elttil bmal
```

To stop the program, hit <stop>.

Strings of size n in Pascal are treated as "packed array [1..n] of char". Packed arrays are like ordinary arrays but are compressed to

optimize storage. Packed array ELEMENTS can't be used directly as VAR - parameters (but whole arrays can, as in the example).

Examples of string operations in Pascal:

```
var  str:  packed array [1..4] of char;
begin
     str:='when';
     writeln (str>'what')
end.
```

Would print TRUE because "when" is greater than "what" (Lexographically, i.e. dictionary order).

5. Advanced features

Example 23 Records

```
Program clock;
const delay = 950; (*approx. for CBM*)
var i:integer;
    clock: record
             hrs: 0..23;
             mins, secs :0..59;
    end;
begin
    write ('Enter the time, in hours minutes seconds : ');
    read (clock.hrs, clock.mins, clock.secs);
    with clock do repeat
          for i:= 1 to delay do (*nothing*);
          secs:=(secs + 1) mod 60;
          if secs=0 then
           begin
             mins:=(mins+1)mod 60;
             if mins=0 then
               hrs:=(hrs+1)mod 24;
           end;
         writeln (hrs:1,':',mins:1,':',secs:1)
   until false
end.
```

The program should print out the time roughly every second, for example:

```
Enter the time, in hours minutes seconds : 1 39 56
```

Should print:

```
1 : 39 : 57
1 : 39 : 58
1 : 39 : 59
1 : 40 :  0
   etc
```

This example is intended to be a demonstration of the use of records in Pascal, not a replacement for the CBM built-in clock! See the Reference Manual (III.9.9) for how to access the CBM clock.

Records are a way of combining several conceptually related variables into one structure. The record can then be treated as a whole or the parts can be accessed individually using the dot notion.

The WITH..DO statement tells the computer to treat the elements of

"clock" as though they were locally defined individual variables for that statement, removing the need for the "clock." prefix.

Record elements can be any type (for example other records or arrays), and in addition an optional "variant" part is allowed (see reference manual).

Example 24 Pointers

```
var p,q: ↑integer;
begin
    new (p); new (q);
    p↑ :=3;
    q↑ :=4;
    write (p↑,q↑)
end.
```

Should print

```
3       4
```

The variables p and q are not integers but 'pointers' to integer variables. The actual space for the variables to be stored at is created "dynamically" (in other words while the program is running) by the procedure NEW. This allows programs to create variables as required. A major use of pointers is in processing linked lists:

Example 25 Reversing a line of characters using a list

```
program revchars;
type itempointer =↑item;
     item = record
          value:char;
          next: itempointer
     end;
var list,p: itempointer;
begin
    list:=nil;
    repeat
         new (p);
         read (p↑.value);
         p↑.next:=list;
         list:=p
    until eoln;
    repeat
         write (p↑.value);
         p:=p↑.next
    until p=nil
end.
```

The input: Mary had a little lamb.

Should give the result: bmal elttil a dah yraM

This program defines a record containing a pointer to itself. (A recursive definition). Linked lists give very flexible storage but you have to keep careful track of what points to what.

In standard Pascal the procedure DISPOSE (P) releases the storage assigned to the pointer p↑ and can be used when p↑ is no longer needed.

In CBM Resident Pascal, dispose has no effect. (However, it is usually possible for programs themselves to implement some sort of "free list" of unwanted items). The Pascal keyword "nil" is a pointer value which points to no variable.

Example 26 "goto" statements

```
label 294, 33;
begin
    33: writeln ('This should be printed');
        goto 294;
        writeln ('This shouldn''t');
   294: writeln ('Stuck in a loop');
        goto 33
end.
```

This should print:

```
This should be printed
Stuck in a loop
This should be printed
Stuck in a loop
etc.
```

"labels" used with goto statements must be integers and should be declared before constants, data types and variables. GOTO'S should be avoided where possible because they destroy the structure of the program. A common use, however, is for "disaster" exits from nested procedures or statements. Jumping INTO a loop or a procedure will cause unpredictable results.

Extensions to standard Pascal

These are described in the Reference Manual (III.10). One useful procedure is VDU:

Example 27 "poking" the vdu screen

```
var i: integer;
begin page;
    for i:=1 to 40 do vdu (i mod 4,i,'x')
end.
```

This should produce a pattern of x's on the screen. Vdu (i,j,ch) stores the character ch at row i, column j.

Remember, PAGE clears the VDU screen.

6. Disk based operation

So far this manual has been concerned only with using the resident compiler, which is always in RAM. While this may provide an ideal environment for learning Pascal, it necessarily restricts the number of commands available, and the space remaining for user programs.

As you become familiar with Pascal, you will probably want to write larger programs. Using the disk-based compiler and linker, Pascal programs of 3000 lines or more may be run, and this may be extended even further by using program chaining.

You may also want to write programs which access diskette files. While this is possible in resident mode by opening channels to the disk unit, in disk mode the full Pascal file syntax is available, permitting files of any data type.

Disk mode is entered in the editor by typing:

```
disk
```

This removes the resident compiler from memory, making the space available for editing.

Once your program is edited, it MUST be saved on disk, for example:

```
put 0:prog
```

saves it in a file called "prog" on drive 0. To compile the program, type:

```
comp prog
```

The compiler output should be something like:

```
Pascal compiler v 1.4
(c) Copyright TCL Software 1980.

program   0   0009
0 error(s)
Compilation complete.
```

There are various compiler options for generating listings etc. (see command summary - section 7). If all goes well the compiler will put the object code in a file "prog.obj" which can then be executed. If there are any errors during compilation then the Pascal source must be corrected using the editor and re-compiled (Use the command "get prog" to read your Pascal text back into memory).

Finally, type:

ex prog

to execute the object file (prog.obj).

The name of each procedure or function is printed out as it is compiled, together with its static nesting level (0 for the main program, 1 for outer level functions and procedures, and so on). A hexadecimal address is also printed, giving a rough idea of its relative position in memory

The following table summarises the differences between resident and disk mode:

Resident Mode	Disk Mode
Compiler always in RAM	Compiler only in RAM during a compilation
Pascal source and object code in RAM	Source and object code held in disk files

Language differences (see Reference Manual for details):

Textfiles only	All file types supported
	Disk files fully supported
	PACK, UNPACK implemented
DISPOSE is a no-op	DISPOSE fully implemented
	Program chaining allowed

7. Editor command summary

Line numbers

A command beginning with a number is recognised by the editor as a new program line, and is inserted in the program text in the position corresponding to that number.

For example:

```
10 end.
 5 begin (* this command comes first *)
```

A line containing just a number has the effect of deleting that line from the program.

Auto

Enables or disables automatic generation of line numbers.

Examples:

auto 20 <return> - enables auto numbering with an increment of 20.
auto <return> - disables auto line numbering.

default - auto 10.

List

Lists the program currently in memory.

```
Examples: list          -lists entire program
          list 330      -lists line 330 only
          list 100-     -lists lines 100 onwards
          list 100-200  -lists lines 100 through 200
          list -200     -lists up to and including line 200
```

Note: The STOP key pressed during a listing will halt it completely, while pressing any other key freezes the listing until a second key is pressed.

Upper, Lower

Set upper or lower case display mode (The contents of the program memory are unaffected). Default : lower.

Basic

Return to BASIC, reverting to uppercase display. To re-load Pascal type : ↑PASCAL <return>.

Break

Enter the machine language monitor by executing a BRK instruction. (X command returns to PASCAL).

New

Erase the program memory.

Disk

Overwrite resident compiler, allowing larger programs to be edited, and permitting use of the disk based commands (COMP, EX, LINK).

Resident

Re-load the resident compiler, reversing the action of "disk". (If there is insufficient memory you may have to delete some or all of your program first).

Number

Renumber the program lines in memory.
for example:

```
number 1000,2000,30
```

renumbers lines 1000 onwards, starting the new numbers at 2000 and with an increment of 30.

Note that the new starting number must be greater than or equal to the old starting number.

Find

Find and print occurrences of a string in the program.

```
find /function/   -finds all occurrences of "function"
```

```
find /function/,100-250 - finds all occurrences in lines 100 through 250.
```

The "/" can in fact be any delimiting character not contained in the search string.

See note under "list" command to halt execution.

Change

As find, but substitutes a second string for all occurrences found of the first. For example:

```
change /function/procedure/,150
```

Changes all occurrences of "function" in line 150 to "procedure".

See note under "list" to halt command execution.

Delete

Deletes program lines (parameters as with LIST). "Delete" with no parameter is equivalent to NEW.

Put

Saves the program on diskette. For example:

```
put 0:sara   -saves in a file called "SARA" on drive 0. (drive 0
              must be initialised).

put @1:jim   -saves in an existing file called "JIM" on disk 1.
              (drive 1 must be initialised).
```

Note: the drive must be specified.

Get

Reads a program from diskette. For example

```
get sara        (searches both disks if required).
get 1:jim
```

R (or Run) (Resident mode only).

Run the program in memory (compiling first if necessary).

L (Resident mode) - compile, and display the program on the VDU.

P (Resident mode) - compile and list on the printer.

COMP (Disk mode) - compile a program

```
comp sara    - compiles file "sara" giving a relocatable object file
               "0:SARA.OBJ"

comp sara,l - the "l" option gives a listing on the VDU.

Other options:      P - list on printer
```

N - no object code
C - no range checking or line numbers in the object file (giving slightly faster and more compact code)
1 - object file on drive 1

Ex (Disk mode) - execute an object file

ex sara - executes SARA.OBJ

Note - COMP and EX both clear the text buffer.

Hex convert from decimal to hexadecimal.

hex 32 <return> gives the result 0020

Decimal convert from hexadecimal to decimal.

decimal 7f <return> gives the result 127

BASIC commands

Any of the BASIC direct-mode commands below may be used in the editor:

PRINT (or ?)
PRINT#
OPEN
CLOSE
CMD
POKE
SYS
FOR
LET

examples:

```
let ti$ = "120000"          - set clock to mid-day
?ti$                        - print the time
?fre (0)                    - print the number of bytes free
for i=1 to 20: ?i, i*i: next i
```

In addition, the usual DOS support commands are allowed:

```
>              -gives disk status
>$0            -gives directory listing of drive 0
>$1            -gives directory listing of drive 1
>s0:penny      -deletes file "0:PENNY"
↑as            -loads and runs file AS
```

/as -loads the file AS (without running it)
etc.

Link (disk mode)

For large programs it is desirable, (and often physically necessary) to have some form of modularization. Several Pascal source files with inter dependent functions and procedures may be compiled separately and their object files later "linked" into one file. The linker may also be used to produce directly executable CBM files.

Examples:

```
link 0:prog=myprog,yourprog,anyprog
```

links the files MYPROG.OBJ, YOURPROG.OBJ and ANYPROG.OBJ

into one object file PROG.OBJ on disk drive 0.

NOTE - "link" clears the text buffer.

Restrictions

(a) The programs being linked must have identical variable declarations at the outer program level.

(b) Each outer-level function or procedure may only be defined in one file.

(c) If the other files need to refer to this function or procedure, a duplicate header should be included, with the body replaced by the keyword "extern".

(d) The first file in the list is assumed to contain the main program. (The other files would normally just contain a dummy main program:

```
begin
end.)
```

Linker example:

```
file f1:
     program test (input, output);
     var i: integer;
     procedure x; extern; (* x is defined in the other file *)
     procedure y;
       begin
          write (i)
       end;
     begin     (* main program *)
       x
     end.

file f2:
     program testpart2(input, output);
     var i : integer; (* var's must be identical to f1 *)
     procedure y; extern; (* y is defined in f1 *)
     procedure x;
       begin
          i:=3;
          write ('three =');y;
       end;
     begin
     end.
```

The command sequence might be

```
comp f1
comp f2
link 0:test=f1,f2
ex test
```

The program should print "three = 3"

Including other files in a compilation (disk mode only)

If the character "#", followed immediately by a diskette file name, is placed at the beginning of a pascal source line, then this indicates to the compiler that the contents of the specified file are to be included at that point in the program.

This is extremely useful when program segments are to be linked, as global declarations (which need to be the same in each segment) can be kept in a separate file thus simplifying any alterations.

The facility cannot be nested (the included file must itself contain no #filename's).

Locate (disk mode) - make an executable file which can then be loaded under BASIC and executed by just typing "RUN". This command may even be used to create small Pascal programs which will run on a 16K CBM, since the runtime interpreter size is only 10K.

example:

```
locate 0:xyz=jane
```

Creates an executable file XYZ on drive 0 from JANE.OBJ

NOTE - "locate" clears the text buffer.

8. ERROR MESSAGES

? Syntax error - editor command is mis-spelled or has invalid parameters.

?Out of memory error - there is insufficient memory left to do the command you specified, for example inserting a new program line or "GETTING" a file.

With COMP,EX, LINK you should first use the "DISK" command.

?Illegal quantity error - bad numeric input to an editor command, for example "NUMBER".

?File data error - one of the Pascal library files (PASCAL LIB 01 etc) is not present on the disk or else has been corrupted.

Compiler not resident - The L,P and R commands may only be used in resident mode.

No source program - You typed L,P or R with no program text present in the computer's memory.

RUNTIME ERRORS

1. STACK OVERFLOW - (during compilation) program is too big
- (during execution) program needs too much variable space or uses too many levels of recursion.

2. INTEGER READ ERROR - an integer was expected from the keyboard.

3. INTEGER OVERFLOW - overflow when multiplying two integers, or DIVing or MODing by zero, or TRUNCating or ROUNDing too large a number.

4. ARRAY INDEX ERROR - an expression used to index an array is outside the declared range.

5. VARIABLE OUT OF RANGE - a variable, or a procedure or function parameter has been given a value outside the allowed range for that data type.

6. CASE ERROR - there is no label in a case statement corresponding to the value of the selection expression.

7. BAD PCODE - your program has been corrupted, or (hopefully not) a system bug. Occuring at random, this may indicate a memory fault.

8. SET VALUE ERROR - a set element has gone outside the range 0..127

9. FLOATING POINT OVERFLOW - may occur if the result of + - x / SQR or EXP is too large.

10. FLOATING POINT READ ERROR - a floating point constant was expected from the keyboard.

11. UNDEFINED GOTO - a GOTO statement referenced a non-existent label.

12. COMPLEX LOG OR SQUARE ROOT - attempt to take the log or square root of a negative number, or the log of zero.

13. FILE NOT OPEN FOR READING - READ or GET without a reset first.

14. FILE NOT OPEN FOR WRITING - WRITE or PUT without a rewrite first.

15. END OF FILE - attempt to read a file with EOF true.

16. NO FREE I/O CHANNELS - CBM operating system only allows ten files to be open at one time.

17. DEVICE READ ERROR - Bad status byte encountered while reading

data from the IEEE bus.

20 to 72. DISK ERROR - An error status has been detected on the floppy disk unit. Returns the error type and if possible the offending filename.

9. Sample programs

Example 1

The character 'o' should appear to "bounce" around the VDU screen. As a variation, try deleting line 13 to produce a pattern on the screen.

```
program bounce (input,output);
const thecowscomehome = false;
      DELAY = 30;
var row, col, i, j, d : integer;
begin
  row := 0;
  col := 0;
  i :=1; j:=1;
page;
repeat
  for d := 1 to DELAY do;
  vdu (row, col, ' ');
  col := col+j;
  row := row+i;
  if (row > 23) or (row < 0) then begin
     begin
       i := -i;
       row := row+i+i;
     end;
  if (col > 39) or (col < 0) then
     begin
       j := -j;
       col := col+j+j;
     end;
   vdu (row, col, 'o');
 until the cowscomehome
end.
```

Example 2 The game of Nim

```
program nim;
const NROWS = 24;
        delay = 1000;
        coin = 168;
var pile : array [1..3] of 0..NROWS;
        move : record
                ntaken, pileno : integer
        end;
        i : integer;
        key : char;
function gameover : boolean;
begin gameover := (pile[1] + pile [2] + pile [3] = 0) end;

function asc (n : integer) : char;
begin asc := chr (n + ord ('0')) end;
procedure display;
        var p, row, col, firstcol : integer;
begin
        page;
        for p := 1 to 3 do
            begin
                firstcol := p*10;
                for row :=0 to NROWS-1 do
                  if pile [p] >=ONROWS-row then
                    for col := firstcol +3 to
                    firstcol+5 do
                        vdu (row, col, chr (COIN));
                  if pile [p] >=    then
                      vdu (NROWS-1, firstcol, asc (pile[p]
div 10));

                  vdu (NROWS-1, firstcol+1, asc (pile[p] mod
10));

                end
end;
```

```
procedure signon;
        begin
        page;
        writeln ('          *** NIM ***');
        writeln;
        writeln;
        writeln ('I will set up three piles of coins ');
        writeln ('To move, take any number of coins away');
        writeln ('from any pile. The player who clears');
        writeln ('the screen wins. ');
        writeln;
        write (' Now hit any key to start : ');
        while getkey = chr (0) do;
        end;


procedure hismove;
        var ok : boolean;
        begin
                writeln ('Now enter your move :');
                with move do repeat
                        writeln;
                        write ('Pile (1,2 or 3)? ');
                        read (pileno);
                        ok := pileno in [1..3];
                        if ok then
                            begin
                                write ('Number to take away ?
');
                                read (ntaken);
                                ok   :=   ntaken   in   [1..pile
[pileno]];

                            end;
                        if not ok then writeln ('What ??');
                until ok;
                with move do pile [pileno] := pile [pileno]
                                - ntaken;
        end; (* of hismove *)
```

```
Procedure mymove;
var bit : array [1..3, 1..4] of boolean;
        parity : array [1..4] of boolean;
        firstbit, x, i, j : integer;
        begin
                for i := 1 to 3 do
                    begin
                        x := pile [i];
                        for j := 4 downto  1 do
                            begin
                                bit [i, j] := odd (x);
                                x := x div 2;
                            end;
                    end;
                for i := 1 to 4 do parity [i] :=
                         bit [1,i] <> (bit [2,i] <> [3,i]);
                    move.pileno := 1;
                    move.ntaken := 0;
                    with move do
                    if not (parity [1] or parity [2] or parity
                          [3] or parity [4])then
                        begin
                            while pile [pileno] = 0 do pileno
                                        := pileno + 1;
                            if pile [pileno] =1 then ntaken:= 1
                            else
                            ntaken    :=    random    mod    (pile
[pileno]-1)+1
                        end
                    else begin
                            firstbit := 1;
                            while not parity [firstbit] do
                                    firstbit := firstbit + 1;
                            while not bit [pileno, firstbit] do
                                    pileno := pileno + 1;
                                    for i:= firstbit to 4 do
                                        begin
                                            x := 1;
                                            for j := 3 downto i
do x:= x*2;

                                            if parity [i] then
                                            if bit [pileno, i]
then ntaken

                                                := ntaken + x
                                            else    ntaken    :=
ntaken - x;

                                         end
                        end;
                    with move do pile [pileno] := pile [pileno]
                                                  - ntaken;
            end; (* of mymove *)
```

```
      begin
      signon;
      repeat
              for i:= 1 to 3 do pile [i] := random mod 10 + 6;
              display;
              repeat
                      hismove;
                       if  gameover  then  writeln  ('Congratulations
...You win!')
                      else begin
                              display;
                              mymove;
                              for i := 1 to delay do;
                              display;
                              writeln ('My move was  ', move.ntaken
                                   :3,' from pile', move.pileno :2);

                              if  gameover  then  writeln  ('***  I
win.');
                              writeln;
                              writeln;
                           end;
              until gameover;
              write ('Another game ? ');
              while input ↑ = ' ' do get (input);
              read (key);
              while not eoln do get (input);
       until key = 'n';
       page;
       end.
```

III. TCL Pascal Reference Manual

This manual is intended to be used for quick reference by those familiar with Pascal or a similar programming language.

1. General

1.1 Pascal keywords

These are reserved words in Pascal and cannot be redefined. They must be written without embedded spaces or newlines. A complete list is:

and	do	function	nil	program	type
array	downto	goto	not	record	until
begin	else	if	of	repeat	var
case	end	in	or	set	while
const	file	label	packed	then	with
div	for	mod	procedure	to	

1.2 Pascal identifiers

These are names chosen by the programmer for variables, constants etc., and should consist of at least one letter, followed by zero or more letters or digits. Upper and lower case letters are equivalent. Identifiers should be unique in the first 8 characters, and must not contain embedded blanks.

The following identifiers are standard (but may be redefined):

abs	eoln	new	read	sqrt
arctan	exp	odd	readln	succ
boolean	false	ord	real	text
char	get	output	reset	true
chr	integer	pack	rewrite	trunc
cos	input	page	round	unpack
dispose	in	pred	sin	write
eof	maxint	put	sqr	writeln

(see also section 9 - extensions).

1.3 Other Special symbols

+	<	'(apostrophe)	[	:=
-	<=	.	]	;
*	>=	..	(	,
/	>	(*	)	:
=	<>	*)	↑	

These symbols should not contain embedded blanks.

1.4 Comments

Pascal comments are enclosed by the symbols (* and *).

Comments are totally ignored by the compiler. They can contain any characters except the closing delimiter "*)".

1.5 Constants

Integer constants

These consist of a sequence of digits, for example:

```
33  0001  0
```

No check is made to ensure that the value is less than 2**15. Integer constants must not contain embedded blanks or commas (see also section 9.1 on hex constants).

Real constants

These are of the form:

```
    <integer part> . <fractional part>
or  <integer part> E <exponent>
or  <integer part> . <fractional part> E <exponent>
```

The integer and fractional parts are non-null strings of digits. The "E" may be in upper or lower case in TCL Pascal. The exponent is a digit string which may be preceded by a sign [+ or -].

Real constants must not contain ANY embedded blanks.

Examples:

```
3.14159   4E-9    -387.4E11

1E+30
```

A real constant which is out of range (greater than about 1E38) will cause an error.

Character and string constants

These are enclosed in single quotes, and may contain any character except a newline. Single quotes are included in a string by writing them twice.

Examples:

'c', '$', '''' (character constants)

'Hi there!', 'Fred''s string' (string constants)

(see also section 9.1 on hex constants).

1.6 Blanks

Any number of spaces or newlines may separate two keywords, identifiers, constants or other symbols, but at least one blank is required between adjacent keywords, identifiers and numbers.

2. Data types and operators

2.1 Integer

Pascal integers are whole numbers in the range - MAXINT to + MAXINT, where MAXINT is an implementation defined constant (32767 in TCL Pascal).

TCL Pascal stores integers in 16-bit 2's complement form, so integers may range from -32768 to +32767.

Integer operators are

```
    +   addition
    -   subtraction
    *   multiplication
  div   integer division (result is rounded towards zero)
  mod   remainder operator
    -   (unary operator) negation
```

+ and - produce 2's complement results mod 2**16.
*, div and mod are defined only on values in the range -MAXINT..MAXINT, and the result must be in this range (otherwise an error occurs).

Division by zero causes an error.

```
x mod y = x - ((x div y) * y)
```

2.2 Real

Real numbers in TCL Pascal are held in floating point binary form with a 32-bit mantissa (9 digits). The exponent can range from -38 to +38.

The operators +, -, * behave as for integers, but produce a REAL result. (Which will cause an error if it is out of range).

The operator / denotes floating point division. Division by zero will cause an error.

Integer expressions and constants can be used wherever a real expression is acceptable, but real values can't be used with DIV or MOD.

Conversion from real to integer is done by the functions TRUNC and ROUND (section 2.8).

2.3 Char

The Pascal data type "char" operates on an ordered set of characters. In TCL Pascal the 128 character ASCII set is used. (Extended to 256 characters to include CBM graphics).

In all implementations of Pascal the digits '0' to '9' are guaranteed to be ordered and contiguous, and the letters 'A' to 'Z' are ordered (but not necessarily contiguous).

The standard functions ORD and CHR convert from character to integer and back.

For example, in TCL Pascal:

```
ord  ('A') = 65
chr  (36)  = '$'
```

Also, succ(x) gives the next character after x, and pred(x) gives the character before x, for example:

```
succ('3') = '4'
pred('1') = '0'
```

Note that in TCL CBM Pascal succ (chr (255)) and pred (chr (0)) are undefined, and chr (x) with x outside the range 0..255 is not allowed.

2.4 User-defined (enumerated) types.

These are usually defined by means of a TYPE declaration (section 3.2) for example:

```
type day = (monday, tuesday, wednesday, thursday, friday,
saturday, sunday);

colour   = (RED, GREEN, BLUE);
```

The data type "day" then has seven ordered values represented by the identifiers MONDAY, TUESDAY, etc.

The type "colour" has three values. The functions ORD, SUCC and PRED may be used on these types (see the previous section). For example:

```
succ (wednesday)  = thursday
pred (green)      = red
ord  (monday)     = 0
ord  (green)      = 1
ord  (sunday)     = 6
```

2.5 Subrange types

The user may define subranges over any scalar type except REAL. Examples:

```
type  year = 1970..1990;
      weekday = monday..friday;
```

These types have the same properties as their parent types, but often occupy less storage space, and values are checked at runtime to see that they fall in the required range. They also act as a convenient means of documentation.

2.6 Boolean

Boolean values in Pascal are represented by the standard identifiers TRUE and FALSE. In fact the data type Boolean may be thought of as resulting from the declaration:

```
type boolean = (false, true)
```

so true>false. The Boolean operators defined in Pascal are:

```
and     --   logical "and" operation
or      --   logical "or" operation
not     --   (unary operator) logical negation.
```

The relational operators

```
<       less than
>       greater than
=       equal to
<=      less than or equal to
>=      greater than or equal to
<>      not equal to
```

may be used with any scalar data type (integer, real, Boolean, char, user-defined), and give a Boolean result. They may also be used to compare strings (section 5.)

2.7 Operator precedence

The relational operators

```
<  >  <= >=  =  <>  in (see section 5)
```

have lowest precedence, followed by

```
+   -   or
```

then

* / div mod and

and finally the unary operator

not

Evaluation is otherwise left to right, and can be changed by using parentheses. Particular care should be taken with expressions like:

(x>3) and (y=2)

This would be illegal if the parentheses were omitted.

2.8 Summary of arithmetic and conversion functions

FUNCTION	PARAMETER	RESULT	MEANING
abs (x)	integer	integer	absolute value
abs (x)	real	real	absolute value
sqr (x)	integer	integer	square
sqr (x)	real	real	square
sqrt (x)	real or integer	real	square root (x>=0)
ln (x)	real or integer	real	natural logarithm (x>0)
exp (x)	real or integer	real	e raised to the xth power
sin (x)	real or integer	real	sine (x in radians)
cos (x)	real or integer	real	cosine (x in radians)
arctan (x)	real or integer	real	arctangent (0 to PI radians)
trunc (x)	real	integer	convert real to integer by truncation towards zero
round (x)	real	integer	convert real to integer by rounding
chr (x)	integer	char	convert ASCII value
odd (x)	integer	Boolean	TRUE if x is odd
ord (x)	scalar *	integer	position within a data type
pred (x)	scalar *	scalar	preceding value in a data type
succ (x)	scalar *	scalar	next value in a data type

(* can't be real)

3. Pascal declarations and statements

3.1 Pascal Programs

A Pascal program takes the form:

```
program header
label declaration part
constant declaration part
type declaration part
variable declaration part
function and procedure declarations
BEGIN
executable statements
END.
```

The declarations are all optional. Label declarations are discussed in III 3.12, functions and procedures in III 6.

The program header is optional in TCL Pascal. If it is included it consists of the keyword PROGRAM followed by a name (which can be any valid identifier) followed by a list of identifiers in brackets, for example:

```
program joe (input, output);
```

"Input" and "Output" are external files used by the program "joe". The header is terminated by a semicolon.

The final full stop after the program "end" is always required.

3.1.1 Constant declarations

These are used to assign values to identifiers which will not change throughout the program. They facilitate modifications to the program and provide a means of documentation.

The keyword "const" is followed by one or more declarations of the form

```
identifier = value;
```

"value" may be a signed or unsigned integer, real, a Boolean, character, string, a member of an enumerated type or a previously defined constant identifier.

Examples:

```
const    message = 'hi there!';
         ch      = '$';
         PI      = 3.14;
         MINUSPI = -PI
```

3.2 Type declarations

These are used to make an identifier synonymous with a given data type. The keyword "type" is followed by one or more declarations:

```
identifier = datatype;
```

Examples:

```
type suit  =  (SPADES, HEARTS, DIAMONDS, CLUBS);
     int   =  integer;
     byte  =  0..255;
```

3.3 Variable declarations

In Pascal all variables must be declared explicitly. This is sometimes annoying but makes the programmer's intention clearer and helps the compiler to detect errors.

The word "var" is followed by one or more declarations:

```
identifier list: datatype;
```

Examples:

```
type day = (monday, tuesday, wednesday, thursday, friday);
var x,y:real;
i : integer;
switch : Boolean;
today, tomorrow, payday:day;
favouritecolour : (BLUE, RED, GREEN, PINK);
date : 1970..1990;
```

The variables denoted by these identifiers can then take any of the allowed values for the corresponding data type.

3.4 Executable statements

The executable part of a Pascal program enclosed by the keywords BEGIN and END, consists of zero or more sequentially executed statements separated by semicolons. Redundant semicolons are always accepted and generate no code. There is no need for any correspondence between the logical structure of statements and their physical layout. Well formatted programs with one statement per

line are easier to read, however.

3.5 Assignment statements

The form of this statement is:

```
variable := expression
```

where the left and right hand sides must have compatible data types. This means that they must arise from the same type identifier, or be declared as variables in the same declaration. Exceptions are if the variable type is a subrange of the expression type, or they are sets with compatible base types, or if the left hand side is real and the right hand side is integer.

The value of the variable is set to the value of the expression, and future references to the variable will yield this value.

Examples:

```
x := 3/sqrt(36)          x is set to 0.5

y := x+4                 y is set to 4.5

y := y-2                 y is set to 2.5
```

x and y are "real" variables.

3.6 Compound statements

The construction:

```
BEGIN
Sequence of statements separated by semicolons
END
```

behaves as a single statement, which when executed causes the execution of all the enclosed statements in sequence.

3.7 "If" statements

The statement "IF Boolean expression THEN statement 1" causes statement 1 to be executed only if the expression is TRUE. Alternatively, "IF Boolean expression THEN statement 1 ELSE statement 2" causes statement 2 to be executed instead if the expression is FALSE.

IMPORTANT - no semicolon may be placed before the ELSE.

Statement 1 and statement 2 can be any Pascal statement, including another IF statement. For example: if x then if y then s1 else s2 -

is taken to mean:

```
if x then
  begin
     if y then s1 else s2
  end
```

3.8 "Repeat" statement

```
REPEAT
sequence of statements separated by semicolons
UNTIL Boolean expression
```

causes the sequence to be executed repeatedly (at least once) until the expression evaluates to TRUE when it is checked at the end of a loop.

3.9 "While" statement

```
WHILE Boolean expression DO statement 1
```

Statement 1 is repeated zero or more times until the expression turns out to be FALSE.

3.10 "For" statement

```
FOR variable := e1 TO e2 DO statement 1
```

The variable can be any scalar type except real. e1 and e2 are expressions of the same type as the variable. Statement 1 is executed exactly ord (e2) - ord (e1) + 1 times (zero times if e2<e1). On successive loops the value of the variable is e1, succ(e1), succ (succ(e1)),..., e2

An alternative form is:

```
FOR variable := e2 DOWNTO e1 DO statement 1
```

Where statement 1 is executed with successively decreasing values of the variable.

Statement 1 should not try to alter the variable, as in:

```
for i := 1 to 10 do i := i + 1 (* WRONG *)
```

Structure members (section 5) can't be used as control variables in FOR loops.

Also, control variables must be local to the current block (section 6.4).

3.11 "Case" statement

```
CASE expression OF
  constant list : statement;
  constant list : statement;
     :
     :
  constant list : statement;
END
```

A redundant semicolon may be included before the END, as shown.

Each constant list consists of one or more constants (which must be the same data type as the case expression), separated by commas. The case expression must be a scalar type (and can't be real). Each label in the case statement should be unique, and indicates that the statement it prefixes is the one to be executed if the case expression has that value. If no case labels match the expression value when the case statement is executed, a CASE ERROR occurs.

WARNING - Case statements with a wide spread of values should be avoided, for example:

```
Case n of
  1: statement 1;
  44,255: statement 2
end
```

This will generate a large jump table in memory with null entries for all the intermediate values (2,3 etc.). Generally, case statements are an efficient way of choosing one of many similar statements to execute.

3.12 "Goto" statement

Pascal statements may be prefixed by a label thus:

```
label : statement
```

The label is an unsigned integer which should differ from all other labels in the first 8 digits in TCL Pascal (4 digits in standard Pascal). Control can then be transferred to this statement from another part of the program by means of the "goto" statement.

```
GOTO label
```

All labels must be declared before use (see below).

The effect of jumping into a structured statement (FOR, WHILE, REPEAT, IF, CASE, WITH) or into a function or procedure is undefined.

The use of GOTO's is not recommended if it can be avoided, since programs quickly become unreadable and error detection becomes very difficult.

Jumping to an undefined (as against undeclared) label is signalled as a runtime error in TCL Pascal.

GOTO's can be used to exit from nested functions and procedures.

Label declaration

This takes the form:

```
LABEL list of labels;
```

The labels are separated by commas.

4. Input and Output of text

A file is a Pascal structured variable which (unlike an array) has no fixed size. Its elements are normally accessed sequentially and either reside on a disc or are associated with some physical I/O device such as the keyboard or display.

In this part we look mainly at textfiles, which are essentially files of characters (Pascal data type CHAR), but which give special treatment to the newline character. In particular the standard textfiles INPUT and OUTPUT, which are usually the keyboard and display, are discussed. Disc files are covered later in III.8 and III.9.

4.1 Outputting to textfiles

A textfile is a variable declared as type TEXT. Associated with any Pascal file f is a buffer variable f↑ which is used in transferring data to and from the file. The standard procedure call:

```
write (f, ch)
```

is equivalent to:

```
f↑:= ch; put (f)
```

```
writeln (f)
```

sends a newline character (ASCII carriage return followed by a line feed) to the file f.

```
page (f)
```

sends a form-feed (or clears the screen in the case of the display).

4.2 Inputting from textfiles

```
get (f)
```

reads the next item from the file f into f↑.

```
read (f, ch)
```

for a character variable ch is equivalent to:

```
ch := f↑; get (ch)
```

If the result of a GET is a newline character (a carriage return - linefeeds are ignored in textfiles), then f↑ appears to contain a space and the standard function EOLN(f) is set to TRUE. Otherwise

EOLN (f) is FALSE.

If the end of the file has been reached then get(f) will cause the standard function EOF(f) to become true, and f↑ will be undefined. Doing a get(f) while EOF(f) is TRUE will cause an error.

```
readln(f)
```

skips to the start of the next line. It means:

```
begin while not eoln(f) do get (f); get (f) end
```

4.3 Reading other data types from textfiles

Syntax:

```
read (f, variable list)
```

each variable in the list can be of type CHAR, INTEGER or REAL.

char : reads one character into the variable, as above.

integer : reads any valid (signed or unsigned) integer constant into the variable, skipping leading blanks and newlines.

real : reads any valid integer or real constant into the variable, skipping leading blanks and newlines.

f↑ is set in each case to the next character after the data read.

4.4 Writing other data types to textfiles

Syntax:

```
write (f, expression list).
```

each expression can be of a type CHAR, REAL, BOOLEAN, INTEGER or a string, and may be qualified by a field width

```
expression : w
```

where w is a non-negative integer expression giving the total number of characters to write to the file.

Character or string: Write sufficient spaces to give a total of w characters, then write the character or string. If w is too small then the string is truncated on the right. Default w = the size of the string.

Boolean: As with string, but one of 'TRUE ' or 'FALSE'

is written. Default w=6.

Integer: Write sufficient spaces first to give a total of w characters. Then write the number without leading zeros, preceded by a minus sign if it is negative. If w is too small print out the entire number with no spaces. Default w=7.

Real: There are two formats:
(i) Floating point - write a sign character (space or '-') followed by a digit, followed by a decimal point, followed by enough digits to give a total of w characters, followed by a 4-character exponent. If w is too small, at least one digit is still printed after the decimal point. Default w=12.

(ii) Fixed point - the number of decimal places must be specified:

```
expression : w : d
```

Enough spaces are first printed to give a total of w characters, followed by a minus sign if negative, followed by a decimal point and d fractional digits, with rounding if necessary. If w is too small, no spaces are printed but the entire number is still output.

4.5 Abbreviations

writeln (f,...) is short for write (f,...);writeln (f)

readln (f,...) is short for read (f,...);readln (f)

write (...) is short for write (output,...)

read (....) is short for read (input,...)

writeln is short for writeln (output)

readln is short for readln (input)

eoln is short for eoln (input)

eof is short for eof (input)

page is short for page (output)

4.6 Manipulating files

There is no problem in passing files as variable parameters. In TCL Pascal (but not in standard Pascal) assignment and passing as value parameters is also allowed. For example:

```
var sourcefile : text
    :
    :
begin
     sourcefile := input;
       :
       :
     read (sourcefile,x); (* reads from keyboard *)
```

5. Structured data types

5.1 Arrays

The syntax of array types is:

```
ARRAY (indextype) OF element type.
```

Where "indextype" can be any scalar or subrange type except real. If indextype has values ranging from m to n, say, then this defines an array of ord (n)-ord(m)+1 values of type "elementtype", which are referenced using the subscripts [m], [succ(m)],...,[n].
Alternatively arrays can be accessed as a whole:

Examples:

```
var x,z : array [1..64] of integer;

    y : array [0..3] of array [-4..2] of real,
begin

    x [1]   := 0;
    x [5]   := x [1] + 2;
    y [3] [1]   := 3.345;
    z := x;          (* Transfer whole array *)
```

"element type" may be any Pascal data type. N-dimensional arrays may be abbreviated as follows:

```
array [t1,t2,...,tn] of sometype
```

which is equivalent to:

```
array [t1] of array [t2] of ... array [tn] of sometype
```

example:

```
var x: array [1..7,4..9, boolean] of char;
```

references to n-dimensional arrays may also be abbreviated,

```
x[i,7,false] := '$';
```

5.2 Sets

The syntax is:-

```
SET OF elementtype
```

where "elementtype" should be a scalar or subrange type, but not

REAL. Sets are constructed from a collection of values in square brackets, for example:

```
var x : set of 0..127;
    y : set of (RED,GREEN,BLUE);
begin
    x:= [1,sqr(2), 6..74];
    y:= [BLUE, GREEN];
```

where 6..74 gives all the values between 6 and 74 inclusive. Set elements must have ordinal values between 0 and 127 inclusive. If their base types are compatible, then two sets are said to be compatible, and operations on compatible sets are:

```
*          Intersection (highest precedence)
+ and -    Union and difference
= <> <= >= Equality, inequality and inclusion tests.
```

The IN operator tests membership of a set. The left hand side should be a scalar compatible with the set's base type. IN has the same precedence as the relational operators <>,= etc.

Examples:

Assuming

```
var x,y : set of (APPLES, PEARS, ORANGES, BANANAS, FIGS);
begin
    x:=[APPLES, PEARS, BANANAS];
    y:=[BANANAS, FIGS];
```

Then

```
x+y is [APPLES, PEARS, BANANAS, FIGS]
x-y is [APPLES, PEARS]
x*y is [BANANAS]
x=y,x<=y and x>=y are all false
x<>y is true
y <= [APPLES, FIGS, BANANAS] is true
y<=y is true
y>=y is true
y=[BANANAS, FIGS] is true
BANANAS IN y is true
ORANGES IN x+y is false
```

5.3 Records

The basic syntax is:

```
RECORD
    identifier list : data type;
    identifier list : data type;
    :
    identifier list : data type
END
```

An optional semicolon may be placed before the END. The fields may be accessed by the field name preceeded by a dot, for example:

```
var x,y:record
    a,b:integer;
    c:real
end;
begin
    x.b := -33;
    x.c := 9E-20;
    x.a := x.b+2;
```

Entire records may also be assigned:

```
    x := y;
```

Several different record definitions may be combined using the following syntax:

```
RECORD
any fields common to all variants
CASE identifier:datatype OF
    constant list : (field list);
    constant list : (field list);
    :
    :
    constant list : (field list)
END
```

Again there can be a redundant semicolon before the END. The variant "field lists" may themselves contain nested variants, for example:

```
type date = record
    year : integer;
    month : (JAN,FEB,MAR,APR,MAY,JUN,JLY,AUG,SEP,OCT,NOV,DEC);
    day : 1..31;
end;
```

```
person = record
    name: packed array [1..30] of char;
    birthday : date;
    case status: (EMPLOYED,UNEMPLOYED,RETIRED,STUDENT) of
       UNEMPLOYED: (registered : date);
       EMPLOYED  :  (case selfemployed : boolean of
                       true  : (numberofemployees:integer);
                       false : (employer: packed array [1..30]
                                                    of char;
                                dateemployed : date))
end;
var his:person;
begin
    his.name  := 'Harry Johnson
    his.birthday.year  := 1938;
    his.birthday.month  := DEC;
    his.birthday.day  := 12;
    his.status := EMPLOYED;
    his.employer :=
          :
        etc.
```

WITH statements have the effect of declaring the fields of a record as local variables for that statement. For example:

```
with his, birthday do
    begin
       month := DEC;
       year  := 1938;
       day   := 12;
    end;
```

The record cannot however be referenced as a whole from inside the WITH statement.

```
WITH r1, r2, ..., rn DO statement
```

is equivalent to:

```
WITH r1 DO WITH r2 DO ... WITH rn DO statement
```

5.4 Packed structures.

Records, arrays, sets and files may be preceeded by the word "packed". This is a command to the compiler to optimise storage space for that structure, possibly at the expense of speed in accessing individual components of the structure. In TCL Pascal, "packed" has little effect on speed, but may cut storage by half in arrays of enumerated values, characters and subranges (0..255 and less). The disadvantage is that packed array elements can't be used as VAR parameters to procedures or functions (but whole packed

arrays can).

Packed arrays [1..n] of type CHAR are special in Pascal because they are considered to be string variables of length n.

Examples:

```
var x,y :packed array [1..4] of char;
    z : packed array [1..10] of char;
begin
    x  := 'how ';
    y  := 'when';
    z  := 'Hi there !';
```

y=x is false, y>x, y>=x and y<>x are true.
y>'what' is true, x<'why ' is true.

But note:

x and z are incompatible (different lengths)
x and 'hello' are incompatible.
z and 'who ' are incompatible.

5.5 Pack and Unpack (not available in resident mode)

Access to individual components of packed arrays may be costly, and the programmer is advised to pack or unpack a packed array in a single operation.

If U and P are array variables, for example:

```
type t =(some data type);
var U : array [m..n] of t;
    P:packed array [a..b] of t;
```

where (n-m) >= (b-a) then:

```
pack (U,i,P)
```

is equivalent to:

```
for j:=a to b do P [j] :=U [j-a+i]
```

and

```
unpack (P,U,i)
```

is equivalent to:

```
for j := a to b do U [j-a+i] := P[j]
```

6. Functions and Procedures

6.1 Function and procedure definitions

The syntax for each definition is the same as the syntax for a program, except that a function or procedure header is used instead of a program header, and also a semicolon appears at the end instead of a full stop:

```
procedure or function header
label declarations
const definitions
type definitions
variable declarations
procedure and function definitions
BEGIN
executable code for this procedure or function
END;
```

Any number of procedures or functions may be defined in a program. The definitions should occur between the variable declarations and the main "BEGIN" of the program.

A procedure header has the form:

```
PROCEDURE procedurename;
```

or

```
PROCEDURE procedurename (formal parameter list);
```

A function header has the form:

```
FUNCTION functionname : datatype;
```

or

```
FUNCTION functionname (formal parameter list) : data type;
```

6.2 Procedure and function calls.

Procedure calls are statements having the form:

```
procedurename
```

or

```
procedurename  (parameters)
```

The effect is to execute any code between the BEGIN and the END of the procedure definition, and then return to continue the program normally, from the statement after the call.

Function calls are expressions which have the data type specified in

the function header. To evaluate the function, any code between the BEGIN and the END of the definition is executed, and the value returned is the last value that was assigned to the function name. The value returned by a function must be a scalar or a pointer.

Examples:

```
procedure  x;
begin
    writeln ('xxxxx')
end;
begin x;
    writeln ('yyyyy');
    x
end.
```

Is equivalent to

```
begin
    writeln ('xxxxx');
    writeln ('yyyyy');
    writeln ('xxxxx')
end.
```

The following example will set i to the value 4:

```
var  i : integer;
function xyz : integer;
    begin
      xyz := 2;
      xyz := 4
    end;
begin
    i  := xyz
end.
```

6.3 Parameters

The usefulness of procedure and function calls can be extended by passing parameters. If these are used they must correspond in number, position and type with the formal (dummy) parameters in the definition.

The formal parameter list contains one or more parts separated by semicolons. Each part has one of the forms:

```
identifier list : datatype
VAR identifier list : datatype
FUNCTION identifier list : datatype
PROCEDURE identifier list
```

These correspond to four different classes of parameters, identifiers, variables, FUNCTION and PROCEDURE parameters which are substituted with expression values, variables, function and procedure names respectively when the function or procedure is called.

Examples:

```
const SIZE = 20;
type vec = array [1..SIZE] of integer;
var  v:vec ; i:integer;
function tan (x:real):real;
begin
    tan := sin (x)/cos(x)
end;
procedure zero (var a:vec);
begin
    for i := 1 to SIZE do a [i] := 0
end;
function square (x:integer):integer;
begin
    square := sqr(x)
end;
function sigma (function f:integer; n,m :integer):integer;
var sum, i:integer;
begin
    sum:=0;
    for i:= n to m do sum:=sum+f(i);
    sigma:=sum;
end;
```

Given the above definitions

tan (0.5) would give the tangent of 0.5 radians
(sin (0.5)/cos(0.5))
zero (v) would set the array v to be all zeros.

Note that passing large arrays (and records) as VAR parameters is a good idea, because the computer does not then have to copy the array.

```
sigma (square,1,20)
```

evaluates 1+4+9+16+...+400.

TCL Pascal (and many other Pascal systems) will not let you pass standard function and procedure names as parameters, hence the need for the function "square".

WARNING - Functions and procedures passed as parameters can themselves only have value parameters, and these are not checked.

So:

```
procedure X(a:real);
begin
:
end;
procedure y(procedure b);
begin
    b(4)
end;
:
begin
y (x)
```

will lead to disaster because x expects a real and gets an integer parameter (4).

6.4 Local declarations

Any variables, constants, labels, types, procedures and functions declared within a procedure or function are local to that procedure or function and cannot be referred to from outside it.

"Global" identifiers defined outside a function or procedure may also be referenced inside it, unless they have been redefined by local definitions.

Examples:

```
program example;
var i:integer; (* may be referenced by main prog, P1 and P2 *)
    j:real;    (* may be referenced by main prog and P3 *)
    k:boolean; (* may be referenced anywhere *)
procedure P1;
    var j:integer; (* may be referenced by P1 and P2 only *)
    procedure P2;
    var m:char; (* may be referenced by P2 only *)
    begin
    :
    end;
begin
:
end; (* ofP1 *)
procedure P3;
const i=49;     (* may be referenced by P3 only *)
:
```

P1 and P3 may be called from anywhere.
P2 may be called from P1 or P2 only.

6.5 Recursion and forward references.

Functions and procedures can call themselves recursively:

```
function factorial (x : integer):integer;
begin
    if x=0 then factorial := 1
    else factorial := factorial (x-1)*x
end;
```

factorial (4) gives 4*3*2*1 = 24

Sometimes it is helpful for a procedure to be able to call another procedure before the procedure being called is defined. The undefined procedure must previously have been declared with name and parameter list, together with "forward" - see example. The parameter list is not required on subsequent declaration of the procedure.

Example:

```
procedure x(parameters for x); forward;
procedure y(parameters for y);
begin
    (* calls x *)
end;
procedure x;
begin
    (* calls y *)
end;
```

x and y call one another (they are "mutually recursive").

7. Dynamic storage and Pointers

7.1 Pointers

Variables of a pointer type take as values the memory address of other variables. This can be used in Pascal to create variables as required while the program is running, since the compiler does not need to know the memory address in advance if it can be stored in a pointer. The syntax of a pointer type is:

```
↑ type pointed to
```

where "type pointed to" is an identifier which is the name of some data type (which could be declared later, allowing recursive definitions such as linked lists and trees).

Examples:

```
type treepointer = ↑tree;
     tree = record
       leftbranch, rightbranch : treepointer;
       data : sometype;
     end;
var oak : tree;
    p   : ↑integer;
```

The only way of giving a pointer a value in standard Pascal is to assign it the value "nil" (which is guaranteed to point to no variable) or to use the procedure "NEW".

In TCL Pascal, "nil" is the address 0000.

Pointers can, once assigned a value, be tested for equality (<> and =).

7.2 "New" and "dispose"

NEW allocates a new variable from the available storage (if any) and stores a pointer to it in the specified variable.

The variable created may then be referenced by the pointer variable followed by a ↑.

DISPOSE destroys the variable pointed to by the specified pointer and makes the storage available for other use. Of course you must be sure that the variable being DISPOSED is never referenced again.

Examples:

```
var p: ↑real;
begin
     new (p);
     p↑ :=103.7;
     write (p↑*p↑*p↑);
     dispose (p)
end.
```

Would print the cube of 103.7 and then destroy the space used to store it. P↑ means the variable whose address is in p.

8. Disk Files (sections 8.1 to 8.4 do not apply to resident mode)

8.1 Declarations

Disk files are declared as Pascal variables of type "file of X" where X is the base type of the file, and can be any structured or unstructured data type. For example:

```
type patient = record
       name : packed array [1..20] of char;
       wordnumber : integer
     end;
var f: file of integer;
    g,h:file of patient;
```

Every file f declared in Pascal has an associated buffer variable f↑ whose type is the base type of the file. Disk files can also be textfiles, for example:

```
var f1, f2 :text;      (see section 4.1)
```

8.2 Sequential writing

Before they can be read or written, disk files must be opened using one of the standard procedures RESET and REWRITE. Up to 5 sequential disk files may be open at any time.

```
rewrite (f)
```

creates an empty file which is then open for sequential writing. The end-of-file function eof(f) will return TRUE in this mode. The call put(f) writes the data in the file buffer (the variable f↑) to the file.

The sequence:

```
begin f↑ := expr;  put (f) end
```

may be abbreviated to:

```
write (f,expr)
```

IMPORTANT NOTE - in TCL Pascal, assignments should not be made to the buffer variable f↑ before a reset(f) or rewrite (f) has been done.

8.3 Sequential reading

The procedure call:

```
reset (f)
```

opens the file f for sequential reading. f must previously have been written by a REWRITE command, otherwise the error message FILE DOES NOT EXIST will be printed. The first record in the file will be placed in the variable f↑. (Or if f is empty, f↑ will be undefined and eof(f) will be true).

Successive records can be read into the buffer variable f↑ by the procedure call:

```
get (f)
```

read (f,x) is equivalent to x:=f↑;get (f)

The function eof(f) returns TRUE when there are no more records in the file. Attempts to read past an end-of-file will cause an error.

As an example the following program writes a file containing the numbers 1 to 10, and then reads them back displaying them on the Pet screen:

```
var i: integer;
    testfile : file of integer;
begin
    rewrite (testfile);
    for i := 1 to 10 do write (testfile , i);
    reset (testfile);
    while not eof (testfile) do
       begin
          read (testfile, i);
          writeln (i)
       end
end.
```

8.4 External files

The files described above are "internal" files, in other words temporary files which are normally deleted when the program (or procedure or function) in which they are defined finishes. Permanent diskette files may be created and/or accessed by giving a filename parameter to RESET or REWRITE. (The parameter may be either a string constant or a string variable). This is an extension to standard Pascal allowing specification of filenames, which can be useful in interactive programs.
Note that the filename cannot contain any imbedded spaces. If the filename is a string variable, it should be terminated by at least

one space.

Examples:

```
var fname : packed array [1..15] of char;
    f, g : file of sometype;
begin
    reset (f,'datafile')
    fname := '0:TEMP.HEX     ';
    rewrite (g,fname);
```

8.5 Reading and writing from other devices

Any device on the IEEE bus may be accessed by using RESET or REWRITE with the syntax:

```
    reset (f, devicenumber,secondaryaddress)
or  reset (f, devicenumber,secondaryaddress,filename)
```

where device number and secondary address are integer expressions. This syntax can also be used in resident mode.

Using a secondary address of 256 instead of 0 gives automatic switching between upper and lower case on the series 3022 printer, for example:

```
var printer:text;
begin
     rewrite (printer,4,256);
     writeln (printer,'Message with UPPER case!');
```

The rewrite command may be used to send commands to the floppy disk unit, for example:

```
const DISK = 8;    (* disk unit physical device # *)
      CC = 15;     (* command channel secondary address *)
var   f : text;
begin
      rewrite (f, disk, cc, 'I1');    (* Initialize drive 0 *)
      rewrite (f, disk, cc, 'R0:NEWNAME=OLDNAME');
                                      (* Rename a file *)
      rewrite (f, disk, cc, 'C1:COPY=0:FILE1,0:FILE2');
                                      (* Copy disk files *)
```

Disk textfile example

The following example program prompts the user for a disk file name, and then outputs an upper-and-lower-case textfile to the CBM printer.

8.6 CLOSE command

This command is an extension to standard Pascal. It may be used to explicitly close a file (without resetting or rewriting) if required. The syntax is:

close (f)

```
program printfile;
var  fname : packed array [1..80] of char;
     ch : char;
     f, printer : text;
begin
     writeln;
     writeln ('Filename ? ');
     read (fname);
     reset (f, fname);
     rewrite (printer, 4, 256);
     while not eof (f) do
        begin
           while not eoln (f) do
                begin
                     read (f, ch);
                     write (printer, ch);
                end;
           readln (f);
           writeln (printer);
        end
end.
```

9. Extensions to standard PASCAL

The features described in this section are specific to CBM Pascal and might not be implemented on other systems.

9.1 Hexadecimal constants These are introduced by the symbol $ (for integer constants) or a backslash (for character constants).

Their main application is probably in machine language and I/O interfacing

Examples:

```
const   portA=$e84f;
        linefeed=\a;
var     chardata:char;
begin
        .
        .
        .
        chardata:=linefeed; (* linefeed is a
                            constant of type CHAR *)
        poke (portA, $3f);
```

(writes the data 3f hex to VIA port "A" mapped at hex memory address E84F)

9.2 Memory VDU and port access

The standard functions/procedures PEEK, POKE, ORIGIN, GETKEY and VDU are provided for this purpose.

```
peek (x:integer):0..255
```

is a function which gives the contents of the physical memory location x, while the procedure:

```
poke (x:integer; y:0..255)
```

is used to change the contents of location x to the byte y. Poke should, of course, be used with great care to avoid corrupting your program.

```
origin(x : ↑sometype;y :integer)
```

sets the pointer x to point at the physical memory location y. x can be any pointer type. This should be used with care (see section 10).

The procedure VDU (x,y :integer; c :char) stores the character c in

the VDU memory row x, column y.

Finally, the function

getkey:char

returns a character read directly from the keyboard port. Chr(0) is returned if no character is ready.

Examples:

```
var x:0..255;
```

`begin poke($014c, $33);`	stores the byte 33 (hex) at address 14c (hex)
`x:= peek(47);`	sets x to the contents of decimal memory address 47
`page;`	clears the VDU screen
`VDU(0,3,'?');`	writes a question mark to the VDU row 0, column 3
`while getkey=chr(0)do;`	waits for someone to press a key

9.3 Hexadecimal input and output

The procedures WRHEX and WRHEX2, and the function RDHEX are provided.

```
wrhex (f:text; x:integer)
```

writes x as four hex. digits on the textfile f.

```
wrhex2 (f:text; x:0..255)
```

writes the byte x as two hex. digits.

Examples :

```
rewrite (printer,4,0);
wrhex (printer, -1); wrhex2 (output, 3)
```

prints FFFF on the printer and 03 on the Pet screen.

The function,

```
rdhex  (f:text):integer
```

reads a 16 bit value from the file f, skipping any leading blanks and discarding all but the last four digits read.

9.4 Bit manipulation

ANDB,ORB,XORB,NOTB,SHL, and SHR are functions operating on integers but treated as 16 bit logical data. The first four do bitwise AND, inclusive OR, exclusive OR and 1's complement.

SHL(x,y) shifts x left by y bits(zeros are shifted in)

SHR(x,y) shifts x right by y bits

SHL(x,-y) is equivalent to SHR(x,y)

examples:

```
andb ($fff0,$00ff)=$00f0

orb ($ff00,$000f) = $ff0f

xorb($ff00,$0ff0) = $f0f0

notb($f0f0) = $0f0f

shl(4,4) = $40

shl(4,-1) = 2

shr (4,-12) = $4000

shl(4,0) = shr(4,0)=4

shr ($4444,4) = $444
```

9.5 Catching I/O errors

Occasionally it is necessary for a program to protect itself against unexpected termination due to invalid input.
The procedure call:

```
iotrap(false)
```

turns off PASCAL error messages for real and integer read operations and disk I/O:

```
iotrap(true)
```

turns checking back on again. After each integer or floating point or hex read operation the function IOERROR may be used giving an

integer error number:

ioerror= 0-No error
2-Integer read error
10-Floating point read error
etc. (see section II.8 for a complete list of I/O runtime errors).

9.6 Keyboard interrupts

The calls:

```
breaks(true)
breaks(false)
```

enable and disable the stop key respectively.

The default is breaks (true).

9.7 Random Number Generator

The function random :0..255 gives a random no. between 0 and 255. A pseudo-random generating sequence is used but this is initialised by timing all keyboard inputs and is also "kicked" frequently by the PASCAL interpreter.

The construction

```
random+(random mod 128)*256
```

generates a random no. between 0 and MAXINT, while

```
random mod n+1
```

generates an (almost) random no. in the range 1..n if n is not too large.

9.8 Underscore

The character '_' (shifted $) is allowed as a letter in identifiers giving improved readability.

9.9 The CBM internal clock

The clock may be examined by using the three functions :

```
hours : integer
minutes : integer
seconds : integer
```

and may be set using the procedure settime (h,m,s : integer).

Example:

```
settime (12,47,00);
```

Sets the clock to 47 minutes past midday and

```
writeln (hours, ':', minutes, ':', seconds);
```

would print:

```
12:   47:    0
```

9.10 Input of String Variables

String variables (ie packed arrays [1..n] of char) may be read from textfiles in a similar manner to characters, integers and reals. Any leading spaces or newlines are first skipped, then an entire line of characters is read from the file into the string variable. If the string is too long, it is truncated on the right, if it is too short it is padded out with spaces.

A major application is for inputting file names from the console.

9.11 Program chaining (disk mode only)

The TCL Pascal command :

```
chain (filename)
```

stops execution of the current program and invokes the program named. The value of GLOBAL variables will be preserved only if declarations are identical in the old and new programs. All files are closed.

The filename can be either a string or a string variable. (If a string variable, at least one space must be used as terminator).

When used under the EX command, a ".obj" extension is implied.

When used in a LOCATED program, the chain command simply executes the DOS support function "↑", so that it is also possible to chain to a BASIC program.

Example:

file "prog1" (object code in "prog1.obj"):

```
begin
    writeln ('First program');
    chain ('Prog2')
end.
```

file "prog2" (object code in "prog2.obj"):

```
begin
    writeln ('Second program');
    chain ('Prog1')
end.
```

The command

```
ex prog1
```

would cause the following to be printed:

```
First program
Second program
First program
Second program
:
```

until the stop key is pressed.

Program chaining is a useful technique for splitting up large programs, or for menu-driven applications.

10. TCL Pascal interface guide

The purpose of this section is to provide all the necessary information to write 6502 machine language subroutines for TCL Pascal programs.

10.1 Assembly language format

Assembly language routines are declared as Pascal functions or procedures but the body is replaced by the word "extern" followed by an integer constant (the routine address). Any parameters are passed on the stack and should be removed by the assembly language routine. The routine should also push a return value on the stack if it is declared as a function. The best way to describe this is by example, so here is a simple function to add two integers:

```
program test;
function addxy (x,y:integer):integer;
    extern $7400;
begin
    write (addxy(3,4))
end.
```

This should result in the output:

```
7
```

Provided that the assembly language routine is correctly located at memory address 7400h:

```
sptr = $2A ; Pascal stack ptr
*    = $7400
addxy    clc
         ldy #0
         lda (sptr),y  ; low byte y
         ldy #2
         adc (sptr),y  ; low byte x
         sta (sptr),y  ; low byte result
         dey
         lda (sptr),y  ; hi byte y
         ldy #3
         adc (sptr),y  ; hi byte x
         sta (sptr);y  ; hi byte result
         clc
         lda  sptr     ; pop Y, leave result
         adc #2
         sta sptr
         bcc addrts
         inc sptr+1
addrts   rts
```

Note: the top-of-memory pointer at locations $34-35 should first be set to $7400 or below to prevent Pascal from overwriting these locations.

10.2 Storage formats

All scalar and subrange types (except REAL), and pointers are passed as 16-bit words in the usual low-high format.

Reals are passed as 6 bytes; in CBM BASIC format.

```
loc n+5: unused
loc n+4: LS mantissa
loc n+3: .
loc n+2: .
loc n+1: MS mantissa
loc n  : exponent
```

Arrays are stored row-by-row (the opposite to FORTRAN), the lowest element has the lowest address.

Arrays are byte-packed if their elements are scalars in the range 0..255 (eg. char), and "packed" was specified. In this case the size is always rounded up to an even number of bytes.

Records are stored with their fields in reverse order (first declared has highest address). Sets are passed as a 128-bit map, a "one" indicates membership. Odd and even bytes are reversed:

```
loc    n+15:        bit 15 .... bit 8
loc    n+14         bit 7  ...  bit 0
  :
loc    n+1          bit 127 ... bit 120
loc    n            bit 119 ... bit 112
```

IMPORTANT - pointers always point to the location above the highest byte used by the actual data. This also applies to VAR parameters,which are passed as addresses.

Example:

```
const  VDUSIZE = 1000;    (* 25 rows of 40 chars *)
type screen = packed array [1..VDUSIZE] of char;
var  vduptr :↑screen;
begin
    origin (vduptr, $8000 + VDUSIZE)
    :
    :
```

This declares an array based on the CBM vdu address 8000h. vduptr ↑ [1] is the first vdu location.