beginner

COLLABORATORS			
	TITLE : beginner		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 22, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 Floating-Point Numbers

```
Floating-Point Numbers
**********************
```

   Floating-point or real numbers can be used to represent both very small fractions and very large numbers. However, unlike a LONG which can hold every integer in a certain range (see Variable types), floating-point numbers have limited accuracy. Be warned, though: using floating-point arithmetic in E is quite complicated and most problems can be solved without using floating-point numbers, so you may wish to skip this chapter until you really need to use them.

```
  Floating-Point Values
  Floating-Point Calculations
  Floating-Point Functions
  Accuracy and Range
```

1.2 Floating-Point Values

```
Floating-Point Values
=====================
```

   Floating-point values in E are written just like you might expect and are stored in LONG variables:

```
       DEF x
       x:=3.75
       x:=-0.0000367
       x:=275.0
```

You must remember to use a decimal point (without any spaces around it) in the number if you want it to be considered a floating-point number, and this is why a trailing .0 was used on the number in the last assignment. At present you can't express every floating-point value in this way; the

compiler may complain that the value does not fit in 32-bits if you try to use more than about nine digits in a single number. You can, however, use the various floating-point maths functions to calculate any value you want (see Floating-Point Functions).

1.3 Floating-Point Calculations

```
Floating-Point Calculations
===========================
```

Since a floating-point number is stored in a LONG variable it would normally be interpreted as an integer, and this interpretation will generally not give a number anything like the intended floating-point number. To use floating-point numbers in expressions you must use the (rather complicated) floating-point conversion operator, which is the ! character. This converts expressions and the normal maths and comparison operators to and from floating-point.

All expressions are, by default, integer expressions. That is, they represent LONG integer values, rather than floating-point values. The first time a ! occurs in an expression the value of the expression so far is converted to floating-point and all the operators and variables after this point are considered floating-point. The next time it occurs the (floating-point) value of the expression so far is converted to an integer, and the following operators and variables are considered integer again. You can use ! as often as necessary within an expression. Parts of an expression in parentheses are treated as separate expressions, so are, by default, integer expressions (this, includes function call arguments).

The integer/floating-point conversions performed by ! are not simple. They involve rounding and also bounding. Conversion, for example, from integer to floating-point and back again will generally not result in the original integer value.

Here's a few commented examples, where f always holds a floating-point number, and i and j always hold integers:

```
DEF f, i, j
i:=1
f:=1.0
f:=i!  -> i converted to floating-point (1.0)
f:=6.2
i:=!f! -> the expression f is floating-point,
       -> then converted to integer (6)
```

In the first assignment, the integer value one is assigned to i. In the second, the floating-point value 1.0 is assigned to f. The expression on the right-hand side of third assignment is considered to be an integer until the ! is met, at which point it is converted to the nearest floating-point value. So, f is assigned the floating-point value of one (i.e., 1.0), just like it is by the second assignment. The expression in the final assignment needs to start off as floating-point in order to interpret the value stored in f as floating-point. The expression

finishes by converting back to integer. The overall result is to turn the floating-point value of f into the nearest integer (in this case, six).

The assignments below are more complicated, but should be straight-forward to follow. Again, f always holds a floating-point number, and i and j always hold integers.

```
f:=!f*f -> the whole expression is floating-point,
        -> and f is squared (6.2*6.2)
f:=!f*(i!) -> the whole expression is floating-point,
           -> i is converted to floating-point and
           -> multiplied by f
j:=!f/(i!)! -> the whole division is floating-point,
            -> with the result converted to integer
j:=!f!/i -> floating-point f is converted to integer
         -> and is (integer) divided by i
IF !f<230.0 THEN RETURN 0  -> floating-point comparison <
IF !f>(i!)  THEN RETURN 0  -> i converted to floating-point,
                           -> then compared to f
```

If the ! were omitted from the first assignment, then not only would the value in f be interpreted (incorrectly) as integer, but the multiplication performed would be integer multiplication, rather than floating-point. In the second assignment, the parentheses around the expression involving i are crucial. Without the parentheses the value stored in i would be interpreted as floating-point. This would be wrong because i actually stores an integer value, so parentheses are used to start a new expression (which defaults to being integer). The value of i is then interpreted correctly, and finally converted to floating-point (by the ! just before the closing parenthesis). The (floating-point) multiplication then takes place with two floating-point values, and the result is stored in f. In the last two assignments (using division), j is assigned roughly the same value. However, the expression in the first assignment allows for greater accuracy, since it uses floating-point division. This means the result will be rounded, whereas it is truncated when integer division is used.

One important thing to know about floating-point numbers in E is that the following assignments store the same value in g (again, f stores a floating-point number). This is because no computation is performed and no conversion happens: the value in f is simply copied to g. This is especially important for function calls, as we shall see in the next section. Strictly speaking, however, the second version is better, since it shows (to the reader of the code) that the value in f is meant to be floating-point.

```
g:=f
g:=!f
```

1.4 Floating-Point Functions

```
Floating-Point Functions
========================
```

There are functions for formatting floating-point numbers to E-strings

(so that they can be printed) and for decoding floating-point numbers from strings. There are also a number of built-in, floating-point functions which compute some of the less common mathematical functions, such as the various trigonometric functions.

RealVal(string)
This works in a similar way to Val for extracting integers from a string. The decoded floating-point value is returned as the regular return value, and the number of characters of string that were read to make the number is returned as the first optional return value. If a floating-point value could not be decoded from the string then zero is returned as the optional return value and the regular return value will be zero (i.e., 0.0).

RealF(e-string,float,digits)
Converts the floating-point value float into a string which is stored in e-string. The number of digits to use after the decimal point is specified by digits, which can be zero to eight. The floating-point value is rounded to the specified number of digits. A value of zero for digits gives a result with no fractional part and no decimal point. The e-string is returned by this function, and this makes it easy to use with WriteF.

```
PROC main()
  DEF s[20]:STRING, f, i
  f:=21.60539
  FOR i:=0 TO 8
    WriteF('f is \s (using digits=\d)\n', RealF(s, f, i), i)
  ENDFOR
ENDPROC
```

Notice that the floating-point argument, f, to RealF does not need a leading ! because we are simply passing its value and not performing a computation with it. The program should generate the following output:

```
f is 22 (using digits=0)
f is 21.6 (using digits=1)
f is 21.61 (using digits=2)
f is 21.605 (using digits=3)
f is 21.6054 (using digits=4)
f is 21.60539 (using digits=5)
f is 21.605390 (using digits=6)
f is 21.6053900 (using digits=7)
f is 21.60539000 (using digits=8)
```

Fsin(float), Fcos(float), Ftan(float)
These compute the sine, cosine and tangent (respectively) of the supplied float angle, which is specified in radians.

Fabs(float)
Returns the absolute value of float, much like Abs does for integers.

Ffloor(float), Fceil(float)
The Ffloor function rounds a floating-point value down to the nearest, whole floating-point value. The Fceil function rounds it up.

```
Fsqrt(float)
     Returns the square root of float.

Fpow(x,y), Fexp(float)
     The Fpow function returns the value of x raised to the power of y
     (which are both floating-point values).  The Fexp function returns
     the value of e raised to the power of float, where e is the
     mathematically special value (roughly 2.718282).  'Raising to a
     power' is known as exponentiation.

Flog10(float), Flog(float)
     The Flog10 function returns the log to base ten of float (the
     common logarithm).  The Flog function returns the log to base e of
     float (the natural logarithm).  Flog10 and Fpow are linked in the
     following way (ignoring floating-point inaccuracies):

            x = Fpow(10.0, Flog10(x))

     Flog and Fexp are similarly related (Fexp could be used again, using
     2.718282 as the first argument in place of 10.0).

            x = Fexp(Flog(x))
```

Here's a small program which uses a few of the above functions, and shows how to define functions which use and/or return floating-point values.

```
DEF f, i, s[20]:STRING

PROC print_float()
  WriteF('\tf is \s\n', RealF(s, !f, 8))
ENDPROC

PROC print_both()
  WriteF('\ti is \d, ', i)
  print_float()
ENDPROC

/* Square a float */
PROC square_float(f) IS !f*f

/* Square an integer */
PROC square_integer(i) IS i*i

/* Converts a float to an integer */
PROC convert_to_integer(f) IS Val(RealF(s, !f, 0))

/* Converts an integer to a float */
PROC convert_to_float(i) IS RealVal(StringF(s, '\d', i))

/* This should be the same as Ftan */
PROC my_tan(f) IS !Fsin(!f)/Fcos(!f)

/* This should show float inaccuracies */
PROC inaccurate(f) IS Fexp(Flog(!f))
```

```
PROC main()
  WriteF('Next 2 lines should be the same\n')
  f:=2.7; i:=!f!
  print_both()
  f:=2.7; i:=convert_to_integer(!f)
  print_both()

  WriteF('Next 2 lines should be the same\n')
  i:=10;  f:=i!
  print_both()
  i:=10;  f:=convert_to_float(i)
  print_both()

  WriteF('f and i should be the same\n')
  i:=square_integer(i)
  f:=square_float(f)
  print_both()

  WriteF('Next 2 lines should be the same\n')
  f:=Ftan(.8)
  print_float()
  f:=my_tan(.8)
  print_float()

  WriteF('Next 2 lines should be the same\n')
  f:=.35
  print_float()
  f:=inaccurate(f)
  print_float()
ENDPROC
```

The convert_to_integer and convert_to_float functions perform similar conversions to those done by ! when it occurs in an expression. To make things more explicit, there are a lot of unnecessary uses of !, and these are when f is passed directly as a parameter to a function (in these cases, the ! could safely be omitted). All of the examples have the potential to give different results where they ought to give the same, and this is due to the inaccuracy of floating-point numbers. The last example has been carefully chosen to show this.

1.5 Accuracy and Range

```
Accuracy and Range
==================
```

A floating-point number is just another 32-bit value, so can be stored in LONG variables. It's just the interpretation of the 32-bits which makes them different. A floating-point number can range from numbers as small as 1.3E-38 to numbers as large as 3.4E+38 (that's very small and very large if you don't understand the scientific notation!). However, not every number in this range can accurately be represented, since the number of significant digits is roughly eight.

Accuracy is an important consideration when trying to compare two

floating-point numbers and when combining floating-point values after dividing them. It is usually best to check that a floating-point value is in a small range of values, rather than just a particular value. And when combining values, allow for a small amount of error due to rounding etc. See the 'Reference Manual' for more details about the implementation of floating-point numbers.