beginner

COLLABORATORS			

	TITLE : beginner		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 1, 2023	

REVISION HISTORY			

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 More About Statements and Expressions

```
                More About Statements and Expressions
**************************************

   This chapter details various E statements and expressions that were not
covered in Part One.  It also completes some of the partial descriptions
given in Part One.
```

```
                Turning an Expression into a Statement

                Initialised Declarations

                Assignments

                More Expressions

                More Statements

                Unification

                Quoted Expressions

                Assembly Statements
```

1.2 Turning an Expression into a Statement

```
Turning an Expression into a Statement
======================================

   The VOID operator converts an expression to a statement.  It does this
by evaluating the expression and then throwing the result away.  This may
not seem very useful, but in fact we've done it a lot already.  We didn't
use VOID explicitly because E does this automatically if it finds an
```

expression where it was expecting a statement (normally when it is on a
line by itself). Some of the expressions we've turned into statements
were the procedure calls (to WriteF and fred) and the use of ++. Remember
that all procedure calls denote values because they're really functions
that return zero by default (see Functions).

   For example, the following code fragments are equivalent:

```
    VOID WriteF('Hello world\n')
    VOID x++

    WriteF('Hello world\n')
    x++
```

Since E automatically uses VOID it's a bit of a waste of time writing it
in, although there may be occasions when you would want to use it to make
this voiding process more explicit (to the reader). The important thing
is the fact that expressions can validly be used as statements in E.

1.3 Initialised Declarations

```
Initialised Declarations
========================
```

   Some variables can be initialised using constants in their declarations.
The variables you cannot initialise in this way are array and complex type
variables (and procedure parameters, obviously). All the other kinds can
be initialised, whether they are local or global. An initialised
declaration looks very much like a constant definition, with the value
following the variable name and a = character joining them. The following
example illustrates initialised declarations:

```
    SET ENGLISH, FRENCH, GERMAN, JAPANESE, RUSSIAN

    CONST FREDLANGS=ENGLISH OR FRENCH OR GERMAN

    DEF fredspeak=FREDLANGS,
        p=NIL:PTR TO LONG, q=0:PTR TO rec

    PROC fred()
      DEF x=1, y=88
      /* Rest of procedure */
    ENDPROC
```

Notice how you need to use a constant like FREDLANGS in order to
initialise the declaration of fredspeak to something mildly complicated.
Also, notice the initialisation of the pointers p and q, and the position
of the type information.

   Of course, if you want to initialise variables with anything more than
a simple constant you can use assignments at the start of the code.
Generally, you should always initialise your variables (using either
method) so that they are guaranteed to have a sensible value when you use
them. Using the value of a variable that you haven't initialised in some

way will probably get you in to a lot of trouble, because the value will
just be something random that happened to be in the memory which is now
being used by the variable. There are rules for how E initialises some
kinds of variables (see the 'Reference Manual', but it's wise to
explicitly initialise even those, as (strangely enough!) this will make
your program more readable.

1.4 Assignments

```
Assignments
===========
```

   We've already seen some assignments--these were assignment statements.
Assignment expressions are similar except (as you've guessed) they can be
used in expressions. This is because they return the value on the
right-hand side of the assignment as well as performing the assignment.
This is useful for efficiently checking that the value that's been
assigned is sensible. For instance, the following code fragments are
equivalent, but the first uses an assignment expression instead of a
normal assignment statement.

```
        IF (x:=y*z)=0
          WriteF('Error: y*z is zero (and x is zero)\n')
        ELSE
          WriteF('OK: y*z is not zero (and x is y*z)\n')
        ENDIF

        x:=y*z
        IF x=0
          WriteF('Error: y*z is zero (and x is zero)\n')
        ELSE
          WriteF('OK: y*z is not zero (and x is y*z)\n')
        ENDIF
```

You can easily tell the assignment expression: it's in parentheses and not
on a line by itself. Notice the use of parentheses to group the
assignment expression. Technically, the assignment operator has a very
low precedence. Less technically, it will take as much as it can of the
right-hand side to form the value to be assigned, so you need to use
parentheses to stop x getting the value ((y*z)=0) (which will be TRUE or
FALSE, i.e., -1 or zero).

   Assignment expressions, however, don't allow as rich a left-hand side
as assignment statements. The only thing allowed on the left-hand side of
an assignment expression is a variable name, whereas the statement form
allows:

```
        var
        var [ expression ]
        var . obj_element_name
        ^ var
```

(With as many repetitions of object element selection and/or array
indexing as the elements' types allow.) Each of these may end with ++ or

--. Therefore, the following are all valid assignments (the last
three use assignment expressions):

```
    x:=2
    x--:=1
    x[a*b]:=rubble
    x.apple++:=3
    x[22].apple:=y*z
    x[].banana.basket[6]:=3+full(9)
    x[].pear--:=fred(2,4)

    x.pear:=(y:=2)
    x[y*z].table[1].orange:=(IF (y:=z)=2 THEN 77 ELSE 33)
    WriteF('x is now \d\n', x:=1+(y:=(z:=fred(3,5)/2)*8))
```

You may be wondering what the ++ or -- affect. Well, it's very simple:
they only affect the var, which is x in all of the assignment statements
above. Notice that x[].pear-- is the same as x.pear--, for the same
reasons mentioned earlier (see Element selection and element types).

1.5 More Expressions

```
              More Expressions
================
```

   This section discusses side-effects, details two new operators (BUT and
SIZEOF) and completes the description of the AND and OR operators.

```
              Side-effects

              BUT expression

              Bitwise AND and OR

              SIZEOF expression
```

1.6 Side-effects

```
Side-effects
------------
```

   If evaluating an expression causes the contents of variables to change
then that expression is said to have side-effects. An assignment
expression is a simple example of an expression with side-effects. Less
obvious ones involve function calls with pointers to variables, where the
function alters the data being pointed to.

   Generally, expressions with side-effects should be avoided unless it is

really obvious what is happening. This is because it can be difficult to
find problems with your program's code if subtleties are buried in
complicated expressions. On the other hand, side-effecting expressions
are concise and often very elegant. They are also useful for obfuscating
your code (i.e., making it difficult to understand--a form of copy
protection!).

1.7 BUT expression

```
          BUT expression
--------------
```

   BUT is used to sequence two expressions. exp1 BUT exp2 evaluates
exp1, and then evaluates and returns the value of exp2. This may not
seem very useful at first sight, but if the first expression is an
assignment it allows for a more general assignment expression. For
example, the following code fragments are equivalent:

```
      fred((x:=12*3) BUT x+y)


      x:=12*3
      fred(x+y)
```

Notice that parentheses need to be used around the assignment expression
(in the first fragment) for the reasons given earlier (see
                Assignments
).

1.8 Bitwise AND and OR

```
Bitwise AND and OR
------------------
```

   As hinted in the earlier chapters, the operators AND and OR are not
simply logical operators. In fact, they are both bit-wise operators,
where a bit is a binary digit (i.e., the zeroes or ones in the binary
form of a number). So, to see how they work we should look at what
happens to zeroes and ones:

```
    x   y   x OR y   x AND y
   -----------------------
    1  1    1        1
    1  0    1        0
    0  1    1        0
    0  0    0        0
```

   Now, when you AND or OR two numbers the corresponding bits (binary
digits) of the numbers are compared individually, according to the above
table. So if x were %0111010 and y were %1010010 then x AND y would be

%0010010 and x OR y would be %1111010:

```
        %0111010          %0111010
      AND                OR
        %1010010          %1010010
        -------           -------
        %0010010          %1111010
```

The numbers (in binary form) are lined up above each other, just like you
do additions with normal numbers (i.e., starting with the right-hand
digits, and maybe padding with zeroes on the left-hand side). The two
bits in each column are AND-ed or OR-ed to give the result below the line.

   So, how does this work for TRUE and FALSE, and logic operations? Well,
FALSE is the number zero, so all the bits of FALSE are zeroes, and TRUE is
-1, which has all 32 bits as ones (these numbers are LONG so they are
32-bit quantities). So AND-ing and OR-ing these values always gives
numbers which have all zero bits (i.e., FALSE) or all one bits (i.e.,
TRUE), as appropriate. It's only when you start mixing numbers that
aren't zero or -1 that you can muck up the logic. The non-zero numbers
one and four are (by themselves) considered to be true, but 4 AND 1 is
%100 AND %001 which is zero (i.e., false). So when you use AND as the
logical operator it's not strictly true that all non-zero numbers
represent true. OR does not give such problems so all non-zero numbers
are treated as true. Run this example to see why you should be careful:

```
    PROC main()
      test(TRUE,          'TRUE\t\t')
      test(FALSE,         'FALSE\t\t')
      test(1,             '1\t\t')
      test(4,             '4\t\t')
      test(TRUE OR TRUE,  'TRUE OR TRUE\t')
      test(TRUE AND TRUE, 'TRUE AND TRUE\t')
      test(1 OR 4,        '1 OR 4\t\t')
      test(1 AND 4,       '1 AND 4\t\t')
    ENDPROC

    PROC test(x, title)
      WriteF(title)
      WriteF(IF x THEN ' is TRUE\n' ELSE ' is FALSE\n')
    ENDPROC
```

Here's the output that should be generated:

```
    TRUE            is TRUE
    FALSE           is FALSE
    1               is TRUE
    4               is TRUE
    TRUE OR TRUE    is TRUE
    TRUE AND TRUE   is TRUE
    1 OR 4          is TRUE
    1 AND 4         is FALSE
```

   So, AND and OR are primarily bit-wise operators, but they can be used
as logical operators under most circumstances, with zero representing
false and all other numbers representing true. Care must be taken when
using AND with some pairs of non-zero numbers, since the bit-wise AND of

such numbers does not always give a non-zero (or true) result.

   You can easily turn any value into a real truth value using the expression x<>FALSE, where x represents the value to be converted. For example, this expression is true: (1<>FALSE) AND (4<>FALSE).

1.9 SIZEOF expression

```
SIZEOF expression
-----------------
```

   SIZEOF returns the size, in bytes (8-bits, like a CHAR), of an OBJECT or a built-in type (like LONG). This can be useful for determining storage requirements. For instance, the following code fragment prints the size of the object rec:

```
    OBJECT rec
      tag, check
      table[8]:ARRAY
      data:LONG
    ENDOBJECT

    PROC main()
      WriteF('Size of rec object is \d bytes\n', SIZEOF rec)
    ENDPROC
```

   You may think that SIZEOF is unnecessary because you can easily calculate the size of an object just by looking at the sizes of the elements. Whilst this is generally true (it was for the rec object), there is one thing to be careful about: alignment. This means that ARRAY, INT, LONG and object typed elements must start at an even memory address. Normally this isn't a problem, but if you have an odd number of consecutive CHAR typed elements or an odd sized ARRAY OF CHAR, an extra, pad byte is introduced into the object so that the following element is aligned properly. For an ARRAY OF CHAR this pad byte could be considered part of the array, so in effect this means array sizes are rounded up to the nearest even number. Otherwise, pad bytes are just an unusable part of an object, and their presence means the object size is not quite what you'd expect. Try the following program:

```
    OBJECT rec2
      tag, check
      table[7]:ARRAY
      data:LONG
    ENDOBJECT

    PROC main()
      WriteF('Size of rec2 object is \d bytes\n', SIZEOF rec2)
    ENDPROC
```

The only difference between the rec and rec2 objects is that the array size is seven in rec2. If you run the program you'll see that the size of the object has not changed. We might just as well have declared the table element to be a slightly bigger array (i.e., have eight elements).

1.10   More Statements

```
                More Statements
===============
```

   This section details five new statements (INC, DEC, JUMP, EXIT and
LOOP) and describes the use of labelling.


```
                 INC and DEC statements

                 Labelling and the JUMP statement

                 EXIT statement

                 LOOP block
```


1.11   INC and DEC statements

```
INC and DEC statements
--------------------
```

   INC x is the same as the statement x:=x+1. However, because it doesn't
do an addition it's a bit more efficient. Similarly, DEC x is the same as
x:=x−1.

   The observant reader may think that INC and DEC are the same as ++ and
−−. But there's one important difference: INC x always increases x by
one, whereas x++ may increase x by more than one depending on the type to
which x points. For example, if x were a pointer to INT then x++ would
increase x by two (INT is 16−bit, which is two bytes).

1.12   Labelling and the JUMP statement

```
                Labelling and the JUMP statement
------------------------------
```

   A label names a position in a program, and these names are global
(they can be referred to in any procedure). The most common use of label
is with the JUMP statement, but you can also use labels to mark the
position of some data (see
                Assembly Statements
). To define a label you
write a name followed by a colon immediately before the position you want
to mark. This must be just before the beginning of a statement, either on

the previous line (by itself) or the start of the same line.

   The JUMP statement makes execution continue from the position marked by
a label. This position must be in the same procedure as the JUMP
statement, but it can be, for instance, outside of a loop (and the JUMP
will then have terminated that loop). For example, the following code
fragments are equivalent:

```
    x:=1
    y:=2
    JUMP rubble
    x:=9999
    y:=1234
  rubble:
    z:=88

    x:=1
    y:=2
    z:=88
```

As you can see the JUMP statement has caused the second group of
assignments to x and y to be skipped. A more useful example uses JUMP to
help terminate a loop:

```
    x:=1
    y:=2
    WHILE x<10
      IF y<10
        WriteF('x is \d, y is \d\n', x, y)
      ELSE
        JUMP end
      ENDIF
      x:=x+2
      y:=y+2
    ENDWHILE
  end:
    WriteF('Finished!\n')
```

This loop terminates if x is not less than ten (the WHILE check), or if y
is not less than ten (the JUMP in the IF block). This may seem pretty
familiar, because it's practically the same as an example earlier (see
WHILE loop). In fact, it's equivalent to:

```
    x:=1
    y:=2
    WHILE (x<10) AND (y<10)
      WriteF('x is \d, y is \d\n', x, y)
      x:=x+2
      y:=y+2
    ENDWHILE
    WriteF('Finished!\n')
```

1.13 EXIT statement

```
EXIT statement
--------------
```

   As noted above, you can use the JUMP statement and labelling to break
out of a loop prematurely. However, a much nicer mechanism exists for
WHILE and FOR loops: the EXIT statement. This statement will terminate
the closest one of these loops (of which it is part) if the supplied
expression evaluates to true (i.e., a non-zero value). Any loop using
EXIT can be re-written without it, but sometimes at the expense of
readability.

   The following fragments of code are equivalent:

```
    FOR x:=1 TO 10
      y:=f(x)
    EXIT y=-1
      WriteF('x=\d, f(x)=\d\n', x, y)
    ENDFOR

    FOR x:=1 TO 10
      y:=f(x)
      IF y=-1 THEN JUMP end
      WriteF('x=\d, f(x)=\d\n', x, y)
    ENDFOR
  end:
```

This example shows a situation which is arguably more readable using
something like EXIT. It can be rewritten using a WHILE loop, as below,
but the code is a bit less clear.

```
    going:=TRUE
    x:=1
    WHILE going AND (x<=10)
      y:=f(x)
      IF y=-1
        going:=FALSE
      ELSE
        WriteF('x=\d, f(x)=\d\n', x, y)
        INC x
      ENDIF
    ENDWHILE
```

1.14 LOOP block

```
LOOP block
----------
```

   A LOOP block is a multi-line statement. It's the general form of loops
like the WHILE loop, and it builds a loop with no check. So, this kind of
loop would normally never end. However, as we now know, you can terminate
a LOOP block using the JUMP statement. As an example, the following two

code fragments are equivalent:

```
    x:=0
    LOOP
      IF x<100
        WriteF('x is \d\n', x++)
      ELSE
        JUMP end
      ENDIF
    ENDLOOP
  end:
    WriteF('Finished\n')

    x:=0
    WHILE x<100
      WriteF('x is \d\n', x++)
    ENDWHILE
    WriteF('Finished\n')
```

1.15   Unification

```
Unification
===========
```

   In E, unification is a way of doing complicated, conditional
assignments. It may also be referred to as pattern matching because
that is what it does: it matches patterns and tries to fit values to the
variables mentioned in those patterns. The result of a unification is
true or false, depending on whether the pattern was successfully matched.

   The basic form of a unification expression is:

```
    expression <=> pattern
```

The only things that can be used in a pattern are constants and variable
names, and lists of patterns. (Strictly speaking, lisp-cells are also
allowed, but this variant of unification is beyond the scope of this
Guide.) The pattern is matched against the expression as follows:

  * If pattern is a constant then the match succeeds only if
    expression evaluates to the same value. So, the simple
    unification expression x<=>1 is similar to an equality check x=1.

  * If pattern is a variable name then the match is always successful
    and the variable is assigned the value of expression. So, the
    simple unification expression 1<=>x is similar to an assignment x:=1.

  * If pattern is a list then expression is assumed to be a list, and
    each element of pattern is taken to be a pattern to be
    (recursively) matched against the corresponding element (by index) of
    the expression list. The match succeeds only if the pattern list
    and the expression list are the same length and all the elements
    match. (It is a serious programming error if pattern is a list but
    expression does not represent a list. In this case, strange things

   may happen and the program may crash.)

So, the things in pattern that control whether a match succeeds are the
constants and the lists.

   If a match succeeds then all variables mentioned in the pattern will
be assigned the appropriate values. However, if a match fails you should
consider all variables involved in the pattern to have undefined values
(so you may need to initialise them to safely use their values again).
This is because the actual way that unification is implemented may not
follow the rules above in the obvious way, but will have the same effect
in the successful case and will affect only the variables mentioned in the
pattern if the match fails.

   For example, the following program shows a couple of simple unification
expressions in use:

```
PROC main()
  DEF x, lt
  x:=0
  WriteF('x is \d\n', x)
  lt:=[9,-1,7,4]

  /* The next line uses unification */
  IF lt <=> [9,-1,x,4]
    WriteF('First match succeeded\n')
    WriteF('1) x is now \d\n', x)
  ELSE
    WriteF('First match failed\n')
    /* To be safe, reset x */
    x:=0
  ENDIF

  /* The next line uses unification */
  IF lt <=> [1,x,6,4]
    WriteF('Second match succeeded\n')
    WriteF('2) x is now \d\n', x)
  ELSE
    WriteF('Second match failed\n')
    /* To be safe, reset x */
    x:=0
  ENDIF
ENDPROC
```

The first match will succeed in this example, and there will be a
side-effect of assigning seven to x. The second match will not succeed
because, for instance, the first element of lt is not one.

   We can rewrite the above example without using the unification operator
(to show why unification is so useful). This code follows the rules in
one particular way, so is not guaranteed to have the same effect as the
unification version if any of the matches fail.

```
PROC main()
  DEF x, lt, match
  x:=0
  WriteF('x is \d\n', x)
```

```
      lt:=[9,-1,7,4]

      /* The next lines mimic: lt <=> [9,-1,x,4] */
      match:=FALSE
      IF ListLen(lt)=4
        IF ListItem(lt, 0)=9
          IF ListItem(lt, 1)=-1
            x:=ListItem(lt,2)
            IF ListItem(lt, 3)=4 THEN match:=TRUE
          ENDIF
        ENDIF
      ENDIF
      IF match
        WriteF('First match succeeded\n')
        WriteF('1) x is now \d\n', x)
      ELSE
        WriteF('First match failed\n')
        /* To be safe, reset x */
        x:=0
      ENDIF

      /* The next lines mimic: lt <=> [1,x,6,4] */
      match:=FALSE
      IF ListLen(lt)=4
        IF ListItem(lt, 0)=1
          x:=ListItem(lt, 1)
          IF ListItem(lt, 2)=6
            IF ListItem(lt, 3)=4 THEN match:=TRUE
          ENDIF
        ENDIF
      ENDIF
      IF match
        WriteF('Second match succeeded\n')
        WriteF('2) x is now \d\n', x)
      ELSE
        WriteF('Second match failed\n')
        /* To be safe, reset x */
        x:=0
      ENDIF
    ENDPROC
```

Here's a slightly more complicated example, which shows how you might use patterns made up of nested lists. Remember that if the pattern is a list then the expression to be matched must be a list. If this is not the case (e.g., if the expression represents NIL) then your program could behave strangely or even crash your computer. A similar, but less disastrous, problem is if the converse happens: the pattern is not a list but the expression to be matched is a list. In this case the pointer (to the list) is matched against the pattern constant or assigned to the pattern variable.

```
    PROC main()
      DEF x=10, y=-3, p=NIL:PTR TO LONG, lt, i
      WriteF('x is \d, y is \d\n', x, y)
      lt:=[[23,x],y]

      /* This basically swaps x and y */
```

```
        IF lt <=> [[23,y],x]
          WriteF('First match succeeded\n')
          WriteF('1) Now x is \d, y is \d\n', x, y)
        ELSE
          WriteF('First match failed\n')
          /* To be safe, reset x and y */
          x:=10;   y:=-3
        ENDIF

        /* This will make p point to the sub-list of lt */
        IF lt <=> [p,-3]
          WriteF('Second match succeeded\n')
          WriteF('2) p is now $\h (a pointer to a list)\n', p)
          FOR i:=0 TO ListLen(p)-1
            WriteF('   Element \d of the list p is \d\n', i, p[i])
          ENDFOR
        ELSE
          WriteF('First match failed\n')
          /* To be safe, reset p */
          p:=NIL
        ENDIF
      ENDPROC
```

1.16 Quoted Expressions

```
              Quoted Expressions
==================
```

   Quoted expressions are a powerful feature of the E language, and they
require quite a bit of advanced knowledge. Basically, you can quote any
expression by starting it with the back-quote character ` (be careful not
to get it mixed up with the quote character ' which is used for strings).
This quoting action does not evaluate the expression; instead, the address
of the code for the expression is returned. This address can be used just
like any other address, so you can, for instance, store it in a variable
and pass it to procedures. Of course, at some point you will use the
address to execute the code and get the value of the expression.

   The idea of quoted expressions was borrowed from the functional
programming language Lisp. Also borrowed were some powerful functions
which combine lists with quoted expressions to give very concise and
readable statements.

```
              Evaluation

              Quotable expressions

              Lists and quoted expressions
```

1.17 Evaluation

```
Evaluation
----------
```

   When you've quoted an expression you have the address of the code which
calculates the value of the expression. To evaluate the expression you
pass this address to the Eval function. So now we have a round-about way
of calculating the value of an expression. (If you have a GB keyboard you
can get the ` character by holding down the ALT key and pressing the '
key, which is in the corner just below the ESC key. On a US and most
European keyboards it's on the same key but you don't have to press the
ALT key at the same time.)

```
    PROC main()
      DEF adr, x, y
      x:=0; y:=0
      adr:=`1+(fred(x,1)*8)-y
      x:=2; y:=7
      WriteF('The value is \d\n', Eval(adr))
      x:=1; y:=100
      WriteF('The value is now \d\n', Eval(adr))
    ENDPROC

    PROC fred(a,b) RETURN (a+b)*a+20
```

This is the output that should be generated:

```
    The value is 202
    The value is now 77
```

This example shows a quite complicated expression being quoted. The
address of the expression is stored in the variable adr, and the
expression is evaluated using Eval in the calls to WriteF. The values of
the variables x and y when the expression is quoted are irrelevant--only
their values each time Eval is used are significant. Therefore, when Eval
is used in the second call to WriteF the values of x and y have changed so
the resulting value is different.

   Repeatedly evaluating the same expression is the most obvious use of
quoted expressions. Another common use is when you want to do the same
thing for a variety of different expressions. For example, if you wanted
to discover the amount of time it takes to calculate the results of
various expressions it would be best to use quoted expressions, something
like this:

```
    DEF x,y

    PROC main()
      x:=999; y:=173
      time(`x+y,     'Addition')
      time(`x*y,     'Multiplication')
      time(`fred(x), 'Procedure call')
    ENDPROC

    PROC time(exp, message)
```

```
      WriteF(message)
      /* Find current time */
      Eval(exp)
      /* Find new time and calculate difference, t */
      WriteF(': time taken \d\n', t)
   ENDPROC
```

This is just the outline of a program--it's not complete so don't bother
running it. The complete version is given as a worked example later (see
Timing Expressions).

1.18 Quotable expressions

```
Quotable expressions
--------------------
```

   There is no restriction on the kinds of expression you can quote,
except that you need to be careful about variable scoping. If you use
local variables in a quoted expression you can only Eval it within the
same procedure (so the variables are in scope). However, if you use only
global variables you can Eval it in any procedure. Therefore, if you are
going to pass a quoted expression to a procedure and do something with it,
you should use only global variables.

   A word of warning: Eval does not check to see if the address it's been
given is really the address of an expression. You can therefore get in a
real mess if you pass it, say, the address of a variable using {var }.
You need to check all uses of things like Eval yourself, because the E
compiler lets you write things like Eval(x+9), where you probably meant to
write Eval('x+9). That's because you might want the address you pass to
Eval to be the result of complicated expressions. So you may have meant
to pass x+9 as the parameter!

1.19 Lists and quoted expressions

```
Lists and quoted expressions
----------------------------
```

   There are several E built-in functions which use lists and quoted
expressions in powerful ways. These functions are similar to functional
programming constructs and, basically, they allow for very readable code,
which otherwise would require iterative algorithms (i.e., loops).

```
MapList(address,list,e-list,quoted-exp)
      The address is the address of a variable (e.g., {x}), list is a
      list or E-list (the source), e-list is an E-list variable (the
      destination), and quoted-exp is the address of an expression which
      involves the addressed variable (e.g., 'x+2).  The effect of the
      function is to take, in turn, a value from list, store it at
      address, evaluate the quoted expression and store the result in the
```

```
        destination list.  The resulting list is also returned (for
        convenience).

        For example:

                MapList({y}, [1,2,3,a,99,1+c], lt, `y*y)

        results in lt taking the value:

                [1,4,9,a*a,9801,(1+c)*(1+c)]

        Functional programmers would say that MapList mapped the function
        (the quoted expression) across the (source) list.

ForAll(address,list,quoted-exp)
        Works just like MapList except that the resulting list is not stored.
        Instead, ForAll returns TRUE if every element of the resulting list is
        true (i.e., non-zero), and FALSE otherwise.  In this way it decides
        whether the quoted expression is true for all elements of the
        source list.  For example, the following are TRUE:

                ForAll({x}, [1,2,-13,8,0], `x<10)
                ForAll({x}, [1,2,-13,8,0], `x<=8)
                ForAll({x}, [1,2,-13,8,0], `x>-20)

        and these are FALSE:

                ForAll({x}, [1,2,-13,8,0], `x OR x)
                ForAll({x}, [1,2,-13,8,0], `x=2)
                ForAll({x}, [1,2,-13,8,0], `x<>2)

Exists(address,list,quoted-exp)
        Works just like ForAll except it returns TRUE if the quoted
        expression is true (i.e., non-zero) for at least one of the source
        list elements, and FALSE otherwise.  For example, the following are
        TRUE:

                Exists({x}, [1,2,-13,8,0], `x<10)
                Exists({x}, [1,2,-13,8,0], `x=2)
                Exists({x}, [1,2,-13,8,0], `x>0)

        and these are FALSE:

                Exists({x}, [1,2,-13,8,0], `x<-20)
                Exists({x}, [1,2,-13,8,0], `x=4)
                Exists({x}, [1,2,-13,8,0], `x>8)

SelectList(address,list,e-list,quoted-exp)
        Works just like MapList except the quoted-exp is used to decide
        which elements from list are copied to e-list.  The only elements
        which are copied are those for which quoted-exp is true (i.e.,
        non-zero).  The resulting list is also returned (for convenience).

        For example:

                SelectList({y}, [99,6,1,2,7,1,1,6,6], lt, `y>5)
```

        results in lt taking the value:

                [99,6,7,6,6]

1.20   Assembly Statements

                 Assembly Statements
    ===================

    The E language incorporates an assembler so you can write Assembly
mnemonics as E statements. You can even write complete Assembly programs
and compile them using the E compiler. More powerfully, you can use E
variables as part of the mnemonics, so you can really mix Assembly
statements with normal E statements.

    This is not really the place to discuss Assembly programming, so if you
plan to use this feature of E you should get yourself a good book,
preferably on Amiga Assembly. Remember that the Amiga uses the Motorola
68000 CPU, so you need to learn the Assembly language for that CPU. More
powerful and newer Amigas use more advanced CPUs (such as the 68020) which
have extra mnemonics. Programs written using just 68000 CPU mnemonics
will work on all Amigas.

    If you don't know 68000 Assembly language you ought to skip this
section and just bear in mind that E statements you don't recognise are
probably Assembly mnemonics.

                   Assembly and the E language

                   Static memory

                   Things to watch out for

1.21   Assembly and the E language

Assembly and the E language

    You can reference E variables simply by using them in an operand.
Follow the comments in the next example (the comments are on the same
lines as the Assembly mnemonics, the other lines are normal E statements):

    PROC main()
      DEF x
      x:=1
      MOVE.L x, D0 /* Copy the value in x to register D0   */
      ADD.L D0, D0 /* Double the value in D0   */
      MOVE.L D0, x /* Copy the value in D0 back to variable x */

```
        WriteF('x is now \d\n', x)
      ENDPROC
```

Constants can also be referenced but you need to precede the constant with
a #.

```
      CONST APPLE=2

      PROC main()
        DEF x
        MOVE.L #APPLE, D0 /* Copy the constant APPLE to register D0 */
        ADD.L  D0, D0     /* Double the value in D0                 */
        MOVE.L D0, x      /* Copy the value in D0 to variable x     */
        WriteF('x is now \d\n', x)
      ENDPROC
```

Labels and procedures can similarly be referenced, but these are
PC-relative so you can only address them in this way. The following
example illustrates this, but doesn't do anything useful:

```
      PROC main()
        DEF x
        LEA main(PC), A0 /* Copy the address of main to register A0 */
        MOVE.L A0, x      /* Copy the value in A0 to variable x     */
        WriteF('x is now \d\n', x)
      ENDPROC
```

You can call Amiga system functions in the same way as you would normally
in Assembly. You need to load the A6 register with the appropriate
library base, load the other registers with appropriate data and then JSR
to the system routine. This next example uses the E built-in variable
intuitionbase and the Intuition library routine DisplayBeep. When you run
it the screen flashes (or, with Workbench 2.1 and above, you might get a
beep or a sampled sound, depending on your Workbench setup).

```
      PROC main()
        MOVE.L #NIL, A0
        MOVE.L intuitionbase, A6
        JSR DisplayBeep(A6)
      ENDPROC
```

1.22 Static memory

```
Static memory
-------------
```

   Assembly programs reserve static memory for things like strings using
DC mnemonics. However, these aren't real mnemonics. They are, in
fact, compiler directives and they can vary from compiler to compiler.
The E versions are LONG, INT and CHAR (the type names), which take a list
of values, reserve the appropriate amount of static memory and fill in
this memory with the supplied values. The CHAR form also allows a list of
characters to be supplied more easily as a string. In this case, the
string will automatically be aligned to an even memory location, although

you are responsible for null-terminating it. You can also include a whole
file as static data using INCBIN (and the file named using this statement
must exist when the program is compiled). To get at the data you mark it
with a label.

   This next example is a bit contrived, but illustrates some static data:

```
PROC main()
  DEF x:PTR TO CHAR
  LEA datatable(PC), A0
  MOVE.L A0, x
  WriteF(x)
ENDPROC

datatable:
  CHAR 'Hello world\n', 0
moredata:
  LONG 1,5,-999,0;    INT -1,222
  INCBIN 'file.data'; CHAR 0,7,-8
```

The Assembly stuff to get the label address is not really necessary, so
the example could have been just:

```
PROC main()
  WriteF({datatable})
ENDPROC

datatable:
  CHAR 'Hello world\n', 0
```

1.23 Things to watch out for

Things to watch out for

   There are a few things to be aware of when using Assembly with E:

 * All mnemonics and registers must be in uppercase, whilst, of course,
   E variables etc., follow the normal E rules.

 * Most standard Assemblers use ; to mark the rest of the line as a
   comment. In E you can use -> for the same effect, or you can use the
   /* and */ delimiters.

 * Registers A4 and A5 are used internally by E, so should not be messed
   with if you are mixing E and Assembly code. Other registers might
   also be used, especially if you've used the REG keyword. Refer to
   the 'Reference Manual' for more details.

 * E function calls like WriteF can affect registers. Refer to the
   'Reference Manual' for more details.