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 Recursion

```
Recursion
*********

   A recursive function is very much like a function which uses a loop.
Basically, a recursive function calls itself (usually after some
manipulation of data) rather than iterating a bit of code using a loop.
There are also recursive types, which are objects with elements which have
the object type (in E these would be pointers to objects).  We've already
seen a recursive type: linked lists, where each element in the list
contains a pointer to the next element (see Linked Lists).

   Recursive definitions are normally much more understandable than an
equivalent iterative definition, and it's usually easier to use recursive
functions to manipulate this data from a recursive type.  However,
recursion is by no means a simple topic.  Read on at your own peril!


  Factorial Example
  Mutual Recursion
  Binary Trees
  Stack (and Crashing)
  Stack and Exceptions
```

1.2 Factorial Example

```
Factorial Example
=================

   The normal example for a recursive definition is the factorial
function, so let's not be different.  In school mathematics the symbol !
is used after a number to denote the factorial of that number (and only
positive integers have factorials).  n! is n-factorial, which is defined
as follows:
```

```
n! = n * (n-1) * (n-2) * ... * 1     (for n >= 1)
```

So, 4! is 4*3*2*1, which is 24. And, 5! is 5*4*3*2*1, which is 120.

Here's the iterative definition of a factorial function (we'll Raise an exception is the number is not positive, but you can safely leave this check out if you are sure the function will be called only with positive numbers):

```
PROC fact_iter(n)
  DEF i, result=1
  IF n<=0 THEN Raise("FACT")
  FOR i:=1 TO n
    result:=result*i
  ENDFOR
ENDPROC result
```

We've used a FOR loop to generate the numbers one to n (the parameter to the fact_iter), and result holds the intermediate and final results. The final result is returned, so check that fact_iter(4) returns 24 and fact_iter(5) returns 120 using a main procedure something like this:

```
PROC main()
  WriteF('4! is \d\n5! is\d\n', fact_iter(4), fact_iter(5))
ENDPROC
```

If you're really observant you might have noticed that 5! is 5*4!, and, in general, n! is n*(n-1)!. This is our first glimpse of a recursive definition--we can define the factorial function in terms of itself. The real definition of factorial is (the reason why this is the real definition is because the `...' in the previous definition is not sufficiently precise for a mathematical definition):

```
1! = 1
n! = n * (n-1)!    (for n > 1)
```

Notice that there are now two cases to consider. The first case is called the base case and gives an easily calculated value (i.e., no recursion is used). The second case is the recursive case and gives a definition in terms of a number nearer the base case (i.e., (n-1) is nearer 1 than n, for n>1). The normal problem people get into when using recursion is they forget the base case. Without the base case the definition is meaningless. Without a base case in a recursive program the machine is likely to crash! (See Stack (and Crashing).)

We can now define the recursive version of the fact_iter function (again, we'll use a Raise if the number parameter is not positive):

```
PROC fact_rec(n)
  IF n=1
    RETURN 1
  ELSEIF n>=2
    RETURN n*fact_rec(n-1)
  ELSE
    Raise("FACT")
  ENDIF
ENDPROC
```

Notice how this looks just like the mathematical definition, and is nice and compact. We can even make a one-line function definition (if we omit the check on the parameter being positive):

```
PROC fact_rec2(n) RETURN IF n=1 THEN 1 ELSE n*fact_rec2(n-1)
```

You might be tempted to omit the base case and write something like this:

```
/* Don't do this! */
PROC fact_bad(n) RETURN n*fact_bad(n-1)
```

The problem is the recursion will never end. The function fact_bad will be called with every number from n to zero and then all the negative integers. A value will never be returned, and the machine will crash after a while. The precise reason why it will crash is given later (see Stack (and Crashing)).

1.3 Mutual Recursion

Mutual Recursion
==========

In the previous section we saw the function fact_rec which called itself. If you have two functions, fun1 and fun2, and fun1 calls fun2, and fun2 calls fun1, then this pair of functions are mutually recursive. This extends to any amount of functions linked in this way.

This is a rather contrived example of a pair of mutually recursive functions.

```
PROC f(n)
  IF n=1
    RETURN 1
  ELSEIF n>=2
    RETURN n*g(n-1)
  ELSE
    Raise("F")
  ENDIF
ENDPROC

PROC g(n)
  IF n=1
    RETURN 2*1
  ELSEIF n>=2
    RETURN 2*n*f(n-1)
  ELSE
    Raise("G")
  ENDIF
ENDPROC
```

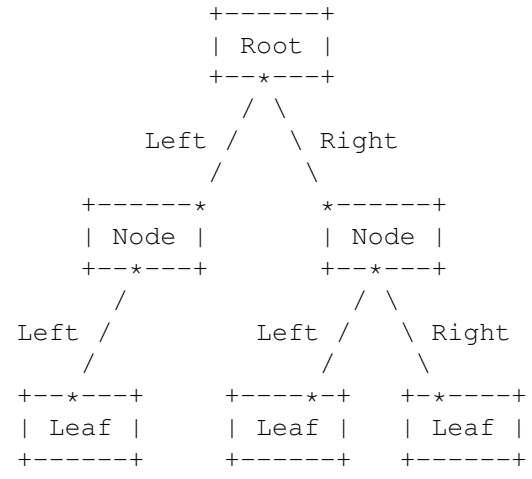
Both functions are very similar to the fact_rec function, but g returns double the normal values. The overall effect is that every other value in long version of the multiplication is doubled. So, f(n) computes

n*(2*(n-1))*(n-2)*(2*(n-3))*...*2 which probably isn't all that interesting.

1.4 Binary Trees

```
Binary Trees
============

   This is an example of a recursive type and the effect it has on
functions which manipulate this type of data.  A binary tree is like a
linked list, but instead of each element containing only one link to
another element there are two links in each element of a binary tree
(which point to smaller trees called branches).  The first link points
to the left branch and the second points to the right branch.  Each
element of the tree is called a node and there are two kinds of special
node: the start point, called the root of the tree (like the head of a
list), and the nodes which do not have left or right branches (i.e., NIL
pointers for both links), called leaves.  Every node of the tree
contains some kind of data (just as the linked lists contained an E-string
or E-list in each element).  The following diagram illustrates a small
tree.
```




```
Notice that a node might have only one branch (it doesn't have to have
both the left and the right).  Also, the leaves on the example were all at
the same level, but this doesn't have to be the case.  Any of the leaves
could easily have been a node which had a lot of nodes branching off it.

   So, how can a tree structure like this be written as an E object?
Well, the general outline is this:

     OBJECT tree
       data
       left:PTR TO tree, right:PTR TO tree
     ENDOBJECT

The left and right elements are pointers to the left and right branches
(which will be tree objects, too).  The data element is some data for each
node.  This could equally well be a pointer, an ARRAY or a number of
```

different data elements.

So, what use can be made of such a tree? Well, a common use is for holding a sorted collection of data that needs to be able to have elements added quickly. As an example, the data at each node could be an integer, so a tree of this kind could hold a sorted set of integers. To make the tree sorted, constraints must be placed on the left and right branches of a node. The left branch should contain only nodes with data that is less than the parent node's data, and, similarly, the right branch should contain only nodes with data that is greater. Nodes with the same data could be included in one of the branches, but for our example we'll disallow them. We are now ready to write some functions to manipulate our tree.

The first function is one which starts off a new set of integers (i.e., begins a new tree). This should take an integer as a parameter and return a pointer to the root node of new tree (with the integer as that node's data).

```
PROC new_set(int)
  DEF root:PTR TO tree
  NEW root
  root.data:=int
ENDPROC root
```

The memory for the new tree element must be allocated dynamically, so this is a good example of a use of NEW. Since NEW clears the memory it allocates all elements of the new object will be zero. In particular, the left and right pointers will be NIL, so the root node will also be a leaf. If the NEW fails a "MEM" exception is raised; otherwise the data is set to the supplied value and a pointer to the root node is returned.

To add a new integer to such a set we need to find the appropriate position to insert it and set the left and right branches correctly. This is because if the integer is new to the set it will be added as a new leaf, and so one of the existing nodes will change its left or right branch.

```
PROC add(i, set:PTR TO tree)
  IF set=NIL
    RETURN new_set(i)
  ELSE
    IF i<set.data
      set.left:=add(i, set.left)
    ELSEIF i>set.data
      set.right:=add(i, set.right)
    ENDIF
    RETURN set
  ENDIF
ENDPROC
```

This function returns a pointer to the set to which it added the integer. If this set was initially empty a new set is created; otherwise the original pointer is returned. The appropriate branches are corrected as the search progresses. Only the last assignment to the left or right branch is significant (all others do not change the value of the pointer), since it is this assignment that adds the new leaf. Here's an iterative

version of this function:

```
PROC add_iter(i, set:PTR TO tree)
  DEF node:PTR TO tree
  IF set=NIL
    RETURN new_set(i)
  ELSE
    node:=set
    LOOP
      IF i<node.data
        IF node.left=NIL
          node.left:=new_set(i)
          RETURN set
        ELSE
          node:=node.left
        ENDIF
      ELSEIF i>node.data
        IF node.right=NIL
          node.right:=new_set(i)
          RETURN set
        ELSE
          node:=node.right
        ENDIF
      ELSE
        RETURN set
      ENDIF
    ENDLOOP
  ENDIF
ENDPROC
```

As you can see, it's quite a bit messier. Recursive functions work well with manipulation of recursive types.

Another really neat example is printing the contents of the set. It's deceptively simple:

```
PROC show(set:PTR TO tree)
  IF set<>NIL
    show(set.left)
    WriteF('\d ', set.data)
    show(set.right)
  ENDIF
ENDPROC
```

The integers in the nodes will get printed in order (providing they were added using the add function). The left-hand nodes contain the smallest elements so the data they contain is printed first, followed by the data at the current node, and then that in the right-hand nodes. Try writing an iterative version of this function if you fancy a really tough problem.

Putting everything together, here's a main procedure which can be used to test the above functions:

```
PROC main() HANDLE
  DEF s, i, j
  Rnd(-999999)    /* Initialise seed */
  s:=new_set(10)  /* Initialise set s to contain the number 10 */
```

```
    WriteF('Input:\n')
    FOR i:=1 TO 50  /* Generate 50 random numbers and add them to set s */
      j:=Rnd(100)
      add(j, s)
      WriteF('\d ',j)
    ENDFOR
    WriteF('\nOutput:\n')
    show(s)         /* Show the contents of the (sorted) set s */
    WriteF('\n')
  EXCEPT
    IF exception="NEW" THEN WriteF('Ran out of memory\n')
  ENDPROC
```

1.5 Stack (and Crashing)

```
Stack (and Crashing)
====================

   When you call a procedure you use up a bit of the program's stack.
The stack is used to keep track of procedures in a program which haven't
finished, and real problems can arise when the stack space runs out.
Normally, the amount of stack available to each program is sufficient,
since the E compiler handles all the fiddly bits quite well.  However,
programs which use a lot of recursion can quite easily run out of stack.

   For example, the fact_rec(10) will need enough stack for ten calls of
fact_rec, nine of which are recursively called.  This is because each call
does not finish until the return value has been computed, so all recursive
calls up to fact_rec(1) need to be kept on the stack until fact_rec(1)
returns one.  Then each procedure will be taken off the stack as they
finish.  If you try to compute fact_rec(40000), not only will this take a
long time, but it will probably run out of stack space.  When it does run
out of stack, the machine will probably crash or do other weird things.
The iterative version, fact_iter does not have these problems, since it
only takes one procedure call to calculate a factorial using this function.

   If there is the possibility of running out of stack space you can use
the FreeStack (built-in) function call (see System support functions).
This returns the amount of free stack space.  If it drops below about 1KB
then you might like to stop the recursion or whatever else is using up the
stack.  Also, you can specify amount of stack your program gets (and
override what the compiler might decide is appropriate) using the OPT
STACK option.  See the 'Reference Manual' for more details on E's
stack organisation.
```

1.6 Stack and Exceptions

```
Stack and Exceptions
====================
```

The concept ‘recent’ used earlier is connected with the stack (see Raising an Exception). A recent procedure is one which is on the stack, the most recent being the current procedure. So, when Raise is called it looks through the stack until it finds a procedure with an exception handler. That handler will then be used, and all procedures before the selected one on the stack are taken off the stack.

Therefore, a recursive function with an exception handler can use Raise in the handler to call the handler in the previous (recursive) call of the function. So anything that has been recursively allocated can be ‘recursively’ deallocated by exception handlers. This is a very powerful and important feature of exception handlers.