Notes on Recursion

David F. Brailsford

1. Introduction

The notion of 'recursion' and a 'recurrence relationship' has been known to mathematics for centuries. Perhaps the best known example of a recursive definition is the one for the *factorial* function:

$$factorial\,(n) = n \times factorial\,(n-1)$$

$$factorial\,(0) = 1$$

The essence of recursion is that one has to retain information about a current instance of a recursive function to be combined with yet more information from lower levels of the recursion. In the case of *factorial* there is, for every given n, a 'pending multiplication' to be performed, which can only be completed when the succeeding $factorial\,(n-1)$ call returns an answer. And the further key realisation is that each $n-1$ value of the function argument becomes a 'new n' for the next recursive invocation.

It was clear from the earliest days of computing that keeping the various values of n, and all the 'pending multiplications' for something like *factorial*, would require a *stack* data structure. When the earliest compiler was being written for FORTRAN, in the middle 1950s, it soon dawned on the compiler writers that the act of parsing the program was akin to traversing a tree and they found themselves inventing stack-like mechanisms to cope. Even so they shrank away from providing recursion at the user level i.e. within the language itself. However, ALGOL60 and all its descendent languages from that time (PASCAL, ALGOL68, C, C++, Java etc.) have made user-level recursion available and have accepted that, for this to work, a run-time stack (and the routines for managing it) have to be provided also. FORTRAN finally caught up with the modern world and made run-time recursion available in its FORTRAN77 version.

It's important to realise, however, that a run-time stack is used nowadays not only for functions that call themselves recursively but also for *every* call of a function or procedure — even those that have no intention of calling themselves recursively. Each managed area on the stack (called a *stack frame*) provides a place where the actual parameters of a function, and any local variables it defines, can safely be stored. So, if procedure *A* calls procedure *B* which in turn calls procedure *C*, and so on, even this non-recursive chain sets up a stack frame for each procedure call. With this in mind we can now imagine how a function calling itself recursively develops lots of stack frames piled one on top of another and these are succesively popped off the stack as the answer from a given invocation is passed back to the previous stack frame (to complete the 'pending multiply' in the case of *factorial*). Thus, in C, the *factorial* function can be programmed as:

```
int factorial (int n)
{
        if (n==0) return 1
        else
        return n * factorial(n-1)
}
```

2. Fibonacci numbers

The classic definition of the *n*th Fibonacci number, F_n is:

$$F_n = F_{n-1} + F_{n-2}$$
$$F_1 = 1$$
$$F_2 = 1$$

This definition is fine, mathematically, and generates the required sequence of 1, 1, 2, 3, 5, 8, 13 etc. However, if it is programmed as a recursive procedure, in exactly this form, it sets off *two* strands of recursion, each of which duplicates much of the work of the other. To condense this down to one strand of recursion we require to return *two* values from every recursive call (i.e. in the above formula we need to return F_{n-1} while somehow accumulating F_{n-2} as well.

Languages such as PostScript give us full access to the run-time stack (and this is true of assembler languages also) so we can return the values 0 and 1 from a call of `fibb(1)`. Once the recursion returns the `fibb` routine has to shift values on the stack (and add other values appropriately) so that a call of `fibb(n)` at any level returns the *n*th Fibonacci number at top of stack and the $n-1$th Fibonacci number below it. At the topmost level, in the main program, where the `fibb(n)` value is required, we have to remember to clean up the stack by getting rid of the `fibb(n-1)` value just below the stack top.

3. Finally

The above material (and the simple examples of implementing recursion as part of the G51CSA course) should have given you some idea of how stack frames and local variables within routines (even recursive ones) can be made to work in practice. If nothing else you might now be grateful for all the work that high-level languages do, on your behalf, in managing parameters and stack frames. And remember: whereas recursion is a 'luxury extra' in imperative languages, it is absolutely and utterly essential 'behind the scenes' to the correct working of functional languages. The best-known progenitor of all functional languages was John McCarthy's 1960 LISP which was a major landmark in Computer Science and is still in use today (particularly in the AI community). For many Computer Scientists LISP was the first dawning of a realisation of what recursion could do, not just in computing mathematical functions such as *factorial* but also in processing lists, traversing trees etc. This tradition continues in more modern functional languages such as Haskell. As you can easily imagine the task of getting the run time system correct (and reasonably efficient), for a functional language, is a considerable challenge.

Examples such as *factorial* and *fibb* are very simple examples of recursion which can easily be converted into iterative versions, using *for* loops, in the language of your choice. But there are other 'killer' examples in recursion function theory that will make you hair stand on end. A classic example is Ackermann's function:

```
int ack(int m,int n)
{
int ans;
if (m == 0) ans=n+1;
else if (n == 0) ans =ack(m-1,1);
else ans = ack(m-1, ack(m,n-1));
return (ans);
}
```

```
int main (int argc, char** argv)

{ int i,j;
for (i=0; i<5; i++)
for (j=0;j<5; j++)

printf ("ackerman (%d,%d) is: %d\n",i,j, ack(i,j));
}
```

Notice how, in the general case, a recursive call is needed merely to evaluate what the second argument to yet another recursive call should be ! If you have a few minutes to spare, and have a talent for not losing track of large amounts of detail, try calculating `ack(2,3)` by hand, from the above definition. Even moderate values of the arguments to `ack` will cause you to run out of stack memory, or to take an intolerable time to compute the answer, or both. For example the fact that $A(4,1) = 65533$ has just taken the best part of 20 CPU minutes to compute on my SPARCBlade workstation. Just like simplistic *fibonacci*, only more so, this function repeats, many times over, the computation of certain $A(x,y)$ intermediate results. A more efficient implementation would store each $A(x,y)$ as it is calculated, in a separate array. The algorithm then descend recursively if, and only if, the required result is not already present in the array.

The difference between *factorial* and *ackermann* points up the distinction between knowing, and not knowing, what depth of recursion will be reached when evaluating a function. In computing `factorial(n)` we know that we shall need $n+1$ stack frames, the final one being for *factorial*$(0) = 1$. In the case of Ackermann's function it is, to put it mildly, far from obvious what the eventual number of stack frames will be. (If you are truly masochistic, and only when you have finished all the courseworks, you could always try programming `ackermann` in ARM assembler).

More generally still, a recursive function may involve an indefinite depth of recursion depending on the arguments. Indeed some recursive definitions may terminate for some inputs and not for others. This observation lies at the very heart of Turing machines and Undecidability beacuse it was shown, very early on, that Turing machine arguments could be equivalently recast as questions and theorems in recursive function theory. What we are saying is that, in the most general case, it is undecidable whether a given recursive function will terminate, though it may be possible to show that it does so for certain *specific* values of the arguments. Functions which sometimes terminate and sometimes loop (recursively) for ever are called *recursively enumerable* functions — they are the most general form of computer program that can exist (and are called Type 0 programs in the Chomsky hierarchy).

4. Acknowledgements

The material on Ackermann's function is adapted from David Barron's 1975 monograph " Recursive Techniques in Programming (2nd Edn.)", (Macdonald/Elsevier) now sadly out of print.