Recursion, see Recursion.

--Amit Saha (amitsaha.in@gmail.com, http://amitksaha.wordpress.com)

Version History:

--Version 1.0: May 28, 2009

The study of Recursion abounds in such statements as:

"Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law. " ---Douglas Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid.

They are amusing and thought provoking at the same time. The idea of recursion is simple when you think of expressing something complex in terms of simpler versions of itself. You will then see recursion in action in the simple things we do everyday, the music we create, the language we speak and the things that happen at sub-atomic levels. In the chapter titled "Recursive Structures and Processes", of Gödel, Escher, Bach (http://en.wikipedia.org/wiki/Gödel,_Escher,_Bach), Douglas Hofstadter (http://en.wikipedia.org/wiki/Douglas_Hofstadter)takes you into a World of Music, Art, Physics and Programming where recursion abound. The underlying concept is to decompose a big problem into indivisible sub-problems, solve them and then combine each of these partial solutions to get the holistic solution. This combination operator is not a universal operator and is specific to the problem at hand. As a crude example, consider the factorial function, fact(N). It is elementary to observe that fact(N)   = N * fact(N-1) = N * (N-1) * fact(N-2)...

As is apparent, we are combining smaller fact()'s by the multiplication operator to obtain fact(N). Here our combination operator is multiplication operator- '*'. In the rest of this paper, I shall touch upon recursion in Formal Languages, Linguistics, Mathematics and Computer Science. Except for the last one, in which I have had my education, I am just a guy who takes keen interest in these fields. So, please take my words with cum grano salis. (and correct me wherever I may have gone wrong).

Recursion in Formal Languages and Linguistics

The grammar G (See http://en.wikipedia.org/wiki/Formal_grammar) of a Language L, is a generative system which, via rules, known as production rules emits countably infinite, finite-length strings. The grammar of a language is thus a compact system of representation for all the strings in a language. Applying these rules and the allowed symbols, also called the alphabet, we can construct strings of the language.

Let us now see a very basic grammar for a language containing 2 rules, and 2 allowed symbols-'a' and 'b' and a special symbol, called the starting symbol, 'S'. Here are the rules:

```
S -> aSSb
S -> ab
```

Let us work these out:

We start with:

```
=> S -> aSSb
=> S -> aabSb (Applying S->ab once)
=> S- > aabaabSbb  (Applying S->aabSb; this is a recursive implementation)
=> S -> aabaababbb (Terminating with S->ab; we could have continued
for as long as we wanted)
```

It is very easy to see, that using the above rules, we can generate as many strings we want of the type: [a(ab)*b], where (ab)* denotes various combinations of (ab). We rely on the recursive usage of 'S' for generating such strings. Likewise, we could introduce other such symbols (for generating complex patterns or implementing larger functionalities).

A classic formalisation of such grammar is due to Naom Chomsky (http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=1056813). The reader will realise that recursion is basic to the study of Formal Languages.

The usage of recursion in linguistics dates back to Panini, an Indian Sanskrit grammarian (http://www-history.mcs.st-andrews.ac.uk/Biographies/Panini.html). According to http://www.infinityfoundation.com/mandala/t_es/t_es_rao-t_syntax.htm- "Panini's grammar uses a variety of formal techniques including recursion, transformations, and metarules."

Recursion in Mathematics & Computer Science

Recursive Procedures & Processes: In Structure and Interpretation of Computer Programs (http://mitpress.mit.edu/sicp/full-text/book/book.html) , the authors draw a distinction between Recursive Procedures and Recursive Processes. Recursive procedures refer to the syntactic fact that the procedure calls itself a finite number of times. The process may evolve in a iterative manner, that is its state at any given point of time of its execution is completely described by its variables. On the other hand, a Recursive Process denotes the evolution of the process in a recursive manner, that is its state at any point of the execution is dependent on some yet to be computed value. Thus, an iteration or a looping construct may as well be described efficiently by a recursive procedure instead of using special looping constructs. This is not encouraged by languages such as C,C++ and Python where each procedure call to itself consumes more space every time, even though optimisation may be possible. These languages hence have special looping constructs which are used for iteration. Please refer to section 1.2 of the text for the complete discussion on the topic.

Recurrence relations: Consider a function defined on a variable- 'x' (where x >=0), f(x), defined as follows: (This is the all familiar, Fibonacci sequence/series)

```
f(x) = f(x-1) + f(x-2)
f(0)=1
f(1)=1
```

So, what do we have here? We have a function which is defined as: the value at some point 'x' is obtained by summing the values of the function at points 'x-1' and 'x-2'. The same idea, of building upon solutions to smaller problems to obtain the solution to a large problem, like:

```
f(2)=f(1)+f(0)
f(3)=f(2)+f(1)
f(4)=f(3)+f(2)
.
.
.
```

..and so on.

(Note: Do you see a computational cost going waste here? We are calculating each f(x) twice. Memoization (also called as Tabulation) can be used to avoid incurring such costs.)

Such equations which definite a sequence recursively are said to be recurrence relations, due to the fact that they express the relations between the elements recursively.

Tail Recursion: A ballooning stack is often the result of a deeply recursive procedure (towards solving a problem recursively) and will eventually cause a stack overflow, blurted out in various forms of the English language. When the last executed statement in a procedure is to itself, its called Tail Recursion. Some compilers recognise this and employ tricks so that the recursion is eliminated completely. In terms of Stacks, the current stack frame being used for the caller is replaced with that of the callee, thus reusing a stack frame, instead of creating a new one.

We shall now consider a tail recursive implementation of a program to calculate the factorial of a integer in a language, whose compiler recognises (and hence, acts too) the Tail Recursion- Scheme(using 'mit-scheme') and and one that doesn't- Python.

```
<code>
;; Factorial using linear recursion
;; fact-tail.scm (syntesised on 'mit-scheme')
;;

(define (factorial n)
   (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
   (if (> counter max-count)
           product
                 (fact-iter (* counter product)
                              (+ counter 1)
                               max-count)))

</code>
```

Try running the above code for any large value of 'n', it won't break.

Now, consider the following Python code:

```
<code>
## Factorial using linear recursion (tail called)
## fact-tail.py (synthesized on 'Python 2.6.2')

def fact_tail(n):
        print fact_iter(1,1,n)


def fact_iter(product, counter, max_count):
        if (counter >= max_count):
                return product
        else:
                return fact_iter(counter * product, counter + 1, max_count)
```

</code>

Run this code for any n > 998, and you should see "RuntimeError: maximum recursion depth exceeded". By default, maximum recursion depth of the Python interpreter is set to 1000. (the remaining 2 "slots" are used up in calling fact_tail, which calls fact_iter)

```
<code>
>>> import sys
>>> print sys.getrecursionlimit()
1000
</code>
```

You can increase the recursion limit to a large value N (using sys.setrecursionlimit(N)) , but for any number which is greater than (N-2), the above execution will break.

(As a fun exercise, you may refer http://amitksaha.wordpress.com/2009/05/23/tail-recursion-optimization-gcc-gdb/ for running a Tail Call optimized C code in the GNU Debuggers-'gdb')

Tail Recursion and the related discussions on its optimization belongs to the broader area of Tail Call Optimization. Please refer http://c2.com/cgi/wiki?TailCallOptimization and http://lambda-the-ultimate.org/classic/message1532.html I might have messed up a bit with the terminology here on my best attempts to be correct as far as I could possibly be.

Divide & Conquer Algorithms: This is a class of algorithms where the central idea is to divide a (large) problem into (two or more) sub-problems, solve them and then combine them to obtain the solution to the problem. Recursion is hence an obvious tool to employ here. Infact, multi-branched recursion is the base on which Divide and Conquer algorithms are formulated. Recurrence relations, mentioned earlier is used to estimate the computational cost of Divide and Conquer algorithms. These are very well described at http://www.cs.berkeley.edu/~vazirani/algorithms/chap2.pdf

Ending Notes

Thank you for reading this far. I think this article doesn't give the feeling of continuity while reading it. Its more like a set of notes put together after reading some books on the subject. But, I have tried hard and this is not a book, anyway :). One topic in particular that I would have liked to include was that of "Recursively Enumerable Sets". I haven't understood enough of it to write about it here. So may be in a next revision of this article, I shall incorporate it.

If you have come across any form of error or misinformation, please email me at amitsaha.in@gmail.com. It will be very much appreciated.

Bibliography

These are the texts I have referred to:

1. Douglas Hofstadter, "Gödel, Escher, Bach: An Eternal Golden Braid (commonly GEB)" (Chapter titled "Recursive Structures and Processes")
2. Harold Abelson, Gerald Jay Sussman, with Julie Sussman, "Structure and Interpretation of Computer Programs" (Section 1.2 mainly)