Computing Fundamentals
Advanced functions & Recursion

Salvatore Filippone

salvatore.filippone@uniroma2.it

2014–2015

Anonymous functions

Useful for one-line functions:

```
fc = @(x) (cos(x)+sin(x))

y=fc(2*pi);
```

You can get a *handle* for an existing function

```
facth=@factorial;
my_root(facth);
facth(4);
```

The (activation) Stack

A *stack* is a fundamental data structure holding a list of items, on which two operations are defined:

PUSH:	add a new item to the stack;
POP:	Take out the most recently added item;
Check:	if the stack is empty;

This is the basis for the management of memory needed by functions:

Functions and Memory

A function file defines the interface and behaviour of a function.

> *An instance of a function is the invocation of a function together with the data it will act upon*

- Each invocation gets a *stack frame* which is filled with the input arguments, and hosts the local variables;
- If the function invokes an auxiliary one, another stack frame is allocated;
- In particular, a function may call itself: such a function is called *recursive*.

Recursion

Recursion is the general technique in which a function is defined in terms of (another instance of) itself

It is a computing counterpart to the principle of *mathematical induction*:

If a property is true of the number 0, and if the truth for number n implies the truth for number $n + 1$, then the property is true for all natural numbers.

Recursion

Basic rules:

- There must be a *base* for which the solution to the problem is known directly;
- The problem must be decomposed into versions of itself with different arguments;
- The arguments of the recursive calls should change in a way that makes progress towards the *recursion base*;

The last condition is especially important: it ensures *termination* of the recursive call chain.

Recursion

First (classical) example: the factorial

```
function nfact=factorial(m)
  if m==0
     nfact=1;
  else
     nfact=m*factorial(m-1);
  end
end
```

Fibonacci

Famous problem by Leonardo Pisano defined by the following equation:

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

Coding:

```
function fn=fibonacci(n)
   if n==0
       fn=0;
   elseif n==1
       fn=1;
   else
       fn=fibonacci(n-1)+fibonacci(n-2);
   end
end
```

If anything can go wrong...

Recursion is a very powerful and elegant idea, but:

- Sometimes very significant memory occupancy;
- If you get the termination conditions wrong you're in (deep) trouble;
- Performance may be an issue;

What is wrong with the fibonacci function?

Starting a recursion

It is often necessary to do something different on the first invocation of a recursive function:

- Do you trust your user(s)? *Bad idea!*
- Do you check your data on each invocation?

The solution often lies in writing a *wrapper* function:

> *An external (user-visible) function that performs any checks and/or setup necessary, then calls the actual recursive function.*

Tail recursion

Recursion is powerful, iteration is efficient: how to get the best of both worlds?

Tail Recursion: The recursive call is the last executable statement in the function, and the result from the recursive call is the same as the function result

In this case the interpreter can overlay the new stack frame over the current one.

> *Tail recursion gets the best of both worlds; it almost always needs a wrapper function.*

Examples: fibonacci, factorial, bisection

Recursion

Variations on the concept of recursion:

Mutual recursion: Two (or more) functions calling each other in a ring: at least one of them has to have a terminating condition;

Generative recursion: It is necessarily not the case that each step moves directly towards termnination

Exception handling

What do you do when things go wrong?

- You may return an error indication (e.g. 0: everything went well, not 0: something went wrong): but this uses the function value!
- You can set a global variable: but the caller still needs to check for it. Moreover, what value do you return when things have gone wrong?

The mechanism available in Octave/Matlab:

- *Throw* an exception with `error` within a `try` block;
- *Catch* an exception in a `catch` block

Exception handling

```
try
  val=ifibo(-19)
catch
 disp('Wrong call to ifibo')
end
```

How does it work?

- When the exception is thrown, execution is immediately halted;
- The stack frames are travelled backwards (and freed) until a `catch` block is found;
- The `catch` block is executed, or the error is displayed at the top level