More on Functions; WHILE instructions

Pedro Barahona
DI/FCT/UNL

Métodos Computacionais
1st Semestre 2017/2018

Iterative Execution - WHILE

- In many cases, although a block of instructions is to be repeated, it is not known before hand how many times it should be iterated.
- For example, to find an element in an array or matrix (or a word in a sequence of text), one might not have to look at **all** the elements of the array/matrix/text, since the element may be found before. In this case, the use of a FOR instruction (although possible) might not be desirable.
- In these cases, the WHILE instruction should be used. The WHILE instruction has the following syntax in MATLAB

```
while <CONDITION>
   WHILE-BLOCK
end;
```

- This instruction is illustrated soon in the **Euclid's algorithm** to find the maximum common divider between two integers.

Iterative Execution - WHILE

```
while <CONDITION>
    WHILE-BLOCK
end;
```

- The behaviour of this instruction is quite intuitive. When the program reaches this instruction
    1. The CONDITION is assessed
    2. If the condition is not satisfied the WHILE-BLOCK is not executed and the program “jumps” to the next instruction.
    3. Otherwise, the WHILE-BLOCK is executed.
    4. After executing the block, the program goes back to step 1 (to assess the CONDITON again, …).
- **NOTE**: Care has to be taken in the specification of the condition and the WHILE-BLOCK. In particular, if this block does not change the variables involved in the CONDITION, so as to make it eventually false, the program **loops forever**!

Euclid's Algorithm

- The Maximum Common Divider (MCD) of two integers, can be obtained by the following algorithm.

1. Take the two numbers, and make them A and B, ensuring that A is no less than B.
2. While A is greater than B
   - Obtain C, the difference between A and B (i.e. C = A – B);
   - Rename the numbers B and C, such that A becomes the larger of them and B the smallest.
   - Check again the condition and iterate as many times as needed.

- When one gets A equal to B, the iterations stop.
- The MCD of the initial numbers is A.

Euclid's Algorithm

Example:

- Let the numbers be 270 and 72, and see the evolution of the values of **a**, **b** and **c**.

a	b	c = a-b
270	72	198
198	72	126
126	72	54
72	54	18
54	18	36
36	18	18
18	18	0

- Hence 18 is the MCD between 270 and 72.

Euclid's Algorithm - WHILE

- The Euclid's Algorithm can be implemented with the following function:

```
function m = euclid(p, q)
% m = euclid(p, q)
% this function computes m, the maximum
% common divider between p and q.
   a = max(p,q);
   b = min(p,q);
   while a > b
      c = b - a;
      if c < b
         a = b;   % the order between these two
         b = c;   % assignments cannot change!
      else
         a = c;   % and b remains b
      end
   end               % at this point a = b
   m = b;
end
```

Euclid’s Algorithm - WHILE

- A trace of the function execution shows how the values of f2, f1 and f are maintained

```
while a > b
   c = a - b;
   if c < b
      a = b;
      b = c;
   else
      a = c;
      b = b;
   end
end
```

```
>> d = euclid(270, 72)
a =  270
b =  72  % before first iteration
a =  198
b =  72  % after first iteration
a =  126
b =  72   % after second iteration
a =  72
b =  54   % after third iteration
a =  54
b =  18   % after fourth iteration
a =  36
b =  18   % after fifth iteration
a =  18
b =  18   % after sixt iteration
d =  18
```

Iterative Execution - WHILE

- We can go back to the problem referred above of finding a value in an array.
- In particular we are interested in specifying a function **find/2** that takes
  - A number as first argument; and
  - An array as second argument;

  and returns
  - The index of the first position where that element appears.
- **Note**: If there is no such element the function should return 0.
- Some examples:
  - `find(3, [5, 8, 4, 3, 6, 8, 2])` → `4`
  - `find(8, [5, 8, 4, 3, 6, 8, 2])` → `2`
  - `find(9, [5, 8, 4, 3, 6, 8, 2])` → `0`

Iterative Execution - WHILE

- Before implementing the function we may design a convenient algorithm to solve this problem. Informally
  - While you have not found it and there is a next element
    - Look at the next element of the array to see if it is the intended one
  - Report the index of the element where you found it
- Although the skeleton of the algorithm is there, a few points must be taken care
  1. Where do we start from
  2. What if the element is not in the array
- Firstly, we must guarantee that we look at the first element, … if there is one!
- Secondly, if there are no more elements to look at, the algorithm must return 0.

- These issues may be dealt with in the specification of the **find/2** function

Iterative Execution - WHILE

- The algorithm can now be implemented as function find/2, shown below

```
function k = find(v, V)
% k = find(v, V)
% this function returns k, the first position, where
% v is in array V. It returns 0 if v is not present.
   found = false;
   k = 0;
   i = 1;
   n = length(V);
   while i <= n && !found % while not found and
      if v == V(i)        % there is a next element to check
         k = i;
         found = true;
      else
         i = i + 1 ;
      end
   end
end
```

WHILE vs. FOR

- Sometimes, namely when it is known the maximum number of times a cycle might be repeated, an instruction FOR might be used to force this (max) number of cycles
- In this case, when the condition to stop the cycle becomes True, then the cycle should be interrupted.
- In the context of a function, this such interruption may be achieved with instruction return, as below

```
function k = find_2(v, V)
% k = find(v, V)
% this function returns k, the first position, where
% v is in array V. It returns 0 if v is not present.
   k = 0;              % initially, the element is yet to find
   for i = 1:length(V)
      if v == V(i)     % if the element is found in position i
         k = i;        % assign the value of the function to i
         return;       % and return (finish the function)
      end
   end
end
```

Iterative Execution - WHILE

- A last note on the condition that could have been used in the WHILE

```
while i <= length(V) && V(i) != v
```

- As we know, trying to read an element of an array past its size reports an error

```
>> A = [ 4 7 5];
>> A(4)
    error: A(I): index out of bounds; value 4 out of bound 3
```

- Hence it is important that testing the value of the element in a certain index is only done after being sure that such index is within the bounds of the vector.

- In MATLAB the Boolean expression A && B (resp. A || B) is executed as follows
  1. Firstly, the Boolean expression A is assessed;
  2. If A is False (resp. True) the condition is False (resp. True)
  3. Otherwise B is assessed.
  4. The value of the condition is the value of B

Nested Functions

- As functions become more complex, their design relies on other functions, either system defined functions or user functions previously defined.
- For example if the **sin/1** function has been defined then the **tang/1** function can be defined in the obvious way.

```
function t = tang(x)
% t = tang(x)
% this function returns t, the tangent of the angle x
   s = sin(x);
   c = sqrt(1-s^2)
   t = s/c;
end
```

- Hence functions can call **other** functions. Assuming the called functions terminate, the calling functions will also terminate.
- However, what happens when a function calls itself?

Recursive Functions: Factorial

- When functions call themselves, i.e. they are defined recursively, one must be careful so that they do terminate.
- Take for example the case of the function **fact/1** defined recursively to obtain the factorial of a non-negative integer (i.e the factorial/1 function, that is already pre-defined in MATLAB).
- This functionality could of course be defined **iteratively**, by means of the **accumulation** technique that we have seen in the previous class, implemented with a for loop.

```
function f = fact_1(n)
% f = fact_1(n)
% this function returns f, the factorial of number n
   p = 1;
   for i = 1:n;
      p = p * i;
   end;
   f = p;
end
```

Recursive Functions: Factorial

- A more "mathematical" definition could however be used to guide the function implementation:

$$n! = \begin{cases} 1 & \text{if } n <= 1 \\ n * (n-1)! & \text{if } n > 1 \end{cases}$$

```
function f = fact (n)
% f = fact (n)
% this function returns f, the factorial of number n
   if n <= 1;
      f = 1;
   else
      f = n * fact(n-1);
   end
end
```

- Notice that in the implementation of this recursive function, the termination condition must be tested **before** the recursive call is made.

- Otherwise the program **loops forever**!

Recursive Functions: Factorial

- A more “mathematical” definition could however be used to guide the function implementation:

$$n! = \begin{cases} 1 & \text{if } n \le 1 \\ n * (n-1)! & \text{if } n > 1 \end{cases}$$

```
function f = fact (n)
% f = fact (n)
% this function returns f, the factorial of number n
   if n <= 1;
      f = n;
   else
      f = n * fact(n-1);
   end
end
```

- Important: In the implementation of a recursive function, the termination condition is tested **before** the recursive call is made. Otherwise the program **loops forever**!

- **Note**: MATLAB has a predefined variable, **max_recursion_depth,** with a (default) value of 256, stops recursion if the depth is exceeded, thus preventing endless loops.

Recursive Functions: Maximum Common Divider

- The same recursive technique may be used to define the MCD of two numbers, taking into account that :

$$\texttt{mdc(m,n)} = \begin{cases} \texttt{m} & \texttt{if m = n} \\ \texttt{mdc(min(m,n),abs(m-n))} & \texttt{if m} \neq \texttt{n} \end{cases}$$

```
function d = mdc(m, n)
% d = mdc (m,n)
% this function returns d, the maximum common divider
% of integers m and n
   if m == n
      d = m;
   else
      p = min(m , n);
      q = abs(m - n);
      d = mdc(p , q);
   end
end
```

- Note again that in this recursive function, the termination condition is tested **before** the recursive call is made

Doubly Recursive Functions: Fibonacci Numbers

- A final example of a function that is defined recursively returns the n^th Fibonacci element of the series

```
1, 1, 2, 3, 5, 8,13, 21, 34, 55 …
```

- Note that in this series, every element is the sum of the two previous elements.
- Hence the function can be defined recursively as

```
fib(n) =  { 1                      if n <= 2
          { fib(m-1)+ fib(m-2)     if n > 2
```

- There is a (significant) difference in this case, which is the fact that the function is recursively called twice, as we will analyse later.
- But from a modelling point of view, the recursively defined function can be implemented as before.

Doubly Recursive Functions: Fibonacci Numbers

```
                 ⎧ 1                        if n <= 2
fib(n)  =        ⎨
                 ⎩ fib(m-1)+ fib(m-2)       if n > 2
```

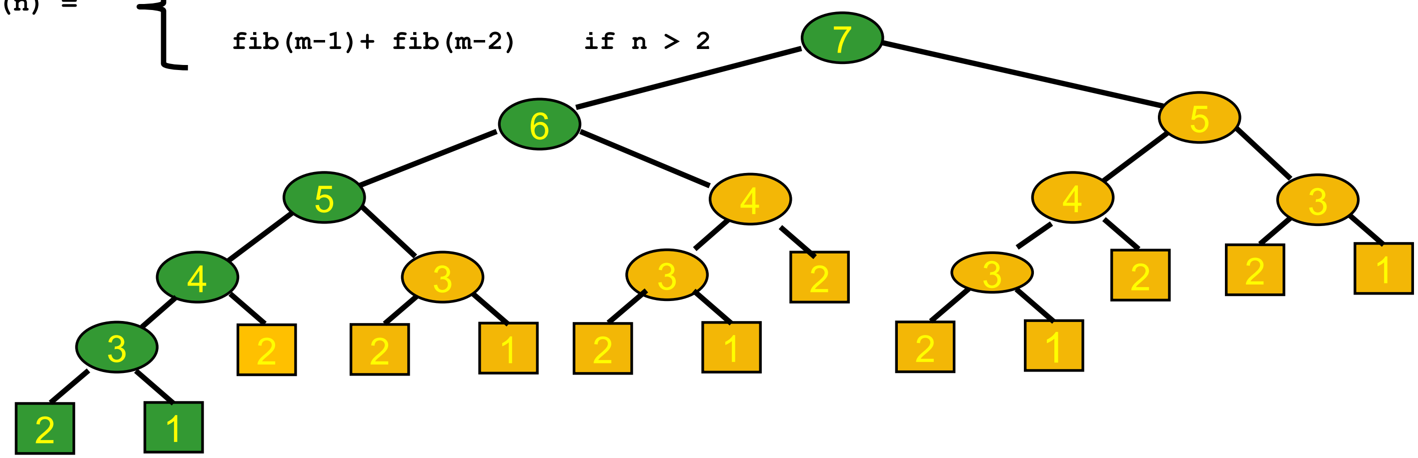
```
function f = fib(n)
% f = fib(n)
% this function returns f, the nth fibonnaci number
   if n <= 2
      f = 1;
   else
      f = fib(n-1) + fib(n-2);
   end
end
```

- Although the termination condition is tested **before** the recursive calls are made, now there are two recursive calls and this has a big impact on the execution
- In particular, many instances of function fib, *with the same input arguments*, are called several times, in fact an **exponential** number of times!

Doubly Recursive Functions: Fibonacci Numbers

- In fact, we can trace the computation, and see that the following calls are made

```
           ⎧ 1                        if n <= 2
fib(n) =  ⎨
           ⎩ fib(m-1)+ fib(m-2)       if n > 2
```



- fib(7) is called 1 time
- fib(6) is called 1 times
- fib(5) is called 2 times
- fib(4) is called 3 times
- fib(3) is called 5 times

- In general,
  - fib(3) is called fib(n-2) times
  - fib(4) is called fib(n-3) times, …
- and fib(n) grows exponentially!

`1, 1, 2, 3, 5, 8,13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, …`

Double Recursive Functions: Fibonacci Numbers

- There are two ways of avoiding this exponential explosion with double recursive functions
  1. use the iterative version for modelling the function
  2. memorize the values of the previous calls

- The iterative version, shown below, maintaining the previous 2 fibonacci numbers in two variables f2 and f1 that are added to obtain the current finonacci number.

```
function f = fib_ite(n)
% f = fib(n)
% this function returns f, the nth fibonnaci number
% using an iterative modelling
   f = 1; f2 = 1; f1 = 1;
   for i = 3:n
      f = f2 + f1;
      f2 = f1;
      f1 = f;
   end
end
```

- Note that the iterations only take place for n >= 3.

Double Recursive Functions: Fibonacci Numbers

- A trace of the function execution shows how the values of f2, f1 and f are maintained

```
f = 1;
f2 = 1;
f1 = 1;
for i = 3:n
    f = f2 + f1;
    f2 = f1;
    f1 = f;
end
```

```
>> n = fib_ite(7)
f =  1
f2 =  1
f1 =  1    % before first iteration
f =  2
f2 =  1
f1 =  2    % after iteration i = 3
f =  3
f2 =  2
f1 =  3    % after iteration i = 4
f =  5
f2 =  3
f1 =  5    % after iteration i = 5
f =  8
f2 =  5
f1 =  8    % after iteration i = 6
f =  13
f2 =  8
f1 =  13   % after iteration i = 7
n =  13
```

Double Recursive Functions: Fibonacci Numbers

- The recursive version with memorization maintains a vector as a **global** variable, i.e. a variable that is defined in the global context, and is thus visible from inside any function.
- Let us call this vector variable fib_vec, and define it in the outer context (initializing the first two numbers in the fibonacci sequence to 1)

```
>> global fib_vec = zeros(1,7)
>> fb_vec(1:2) = 1
fib_vec = 1 1 0 0 0 0 0
```

- Now, any function can read from and write into this function if it identifies the variable as global, **inside** the function body.
- This is done through a global declaration, inside the function body

```
function ...
  global fib_vec;
end
```

Double Recursive Functions: Fibonacci Numbers

- Now the recursive version with memorisation is easily explained.

  If the value has not been computed yet (i.e. `n > 2 && fib_vec(n) ≠ 0`) then

  it is is computed by the (double) recursive call, and

  written in fib_vec

  now the value in fib_vec, can be returned

```
function f = fib_mem(n);
   global fib_vec;               % fib_vec identified as global
   if n > 2 && fib_vec(n) == 0 % value has yet computed
      fib_vec(n) = fib_mem(n-1)+fib_mem(n-2);
   end
   f = fib_vec(n) = f;
end;
```

Global Variables

- A last note on global variables, which have a *state* and the following life cycle.

1. Variables are created, in the outer context, with the declaration **global**.
2. Then they are assessed, either in the outer context, or within some function body.
   a. In this case, they must be identified as global (not to be created again, only to be identified)
3. Eventually, they are destroyed, either because the outer context is finished, or the user wants to reset them.
   a. In the latter case, the instruction **clear** must be used.

```
>> global vec = [ 1 2 3];
>> vec
 vec = 1 2 3
>> clear vec
>> vec
error: Invalid call to vec.  Correct usage is:
 ...
```

Note: Some predefined variables (**pi**, **e**) are predefined global variables. If they are redefined by some assignment, they may be cleared to return to their predefined values.