Chapter 2

Recursivity

2.1 Principle

In mathematics, definitions by induction (by recursion or recursively) are quite common. For example the following sequence $F(n)$ called Fibonacci is recurrent because the function itself is involved in its definition:

$$\begin{cases} F(0) = F(1) = 1 \\ F(n) = F(n-1) + F(n-2) \quad \forall n > 1 \end{cases}$$

In computer science, recursion (i.e. recursivity) is an important concept, it represents an elegant style of programming. It consists of subdividing a problem into several sub-problems then solving each sub-problem and then combining the small solutions to construct the overall solution. This principle is called "*Divide and conquer*" or "*Divide and solve*".

2.2 Recursive algorithm

A program (a function, a procedure) is called recursive **if it calls itself**, that is to say if it contains a call to itself in its definition, otherwise it is called **iterative**.

In the following example, we want to calculate the sum of integers from 0 to n. The $SumI$ function calculates the sum **iteratively** while the $SumR$ function calculates it recursively according to the following principle: "*The sum of the integers from 0 to n is equal to n plus the sum of integers from 0 to $n-1$*".

$$\sum_{i=0}^{n} = n + \sum_{i=0}^{n-1}$$
$$\sum_{i=0}^{0} = 0$$

```
Function SumI( n : integer) : integer;
Var  S,i : integer;
Begin
    S ← 0;
    For i From 1 to n Do
        S ← S + i;
    End For;
    SumI ← S;
End;
// Iterative sum function
```

```
Function SumR( n : integer) : integer;
Begin
    If (n = 0) Then
        SumR ← 0;
    Else
        SumR ← n + SumR(n − 1);
    End If;
End;
// Recursive sum function
```

Executing a $SumR(5)$ call for example is done as follows:

- Call $SumR(5)$
  - 5 + ($SumR(4)$ = ?)
  - Call $SumR(4)$
    - 4 + ($SumR(3)$ = ?)
    - Call $SumR(3)$
      - 3 + ($SumR(2)$ = ?)
      - Call $SumR(2)$
        - 2 + ($SumR(1)$ = ?)
        - Call $SumR(1)$
          - 1 + ($SumR(0)$ = ?)
          - Call $SumR(0)$
            - **Return 0**
        - Return 1 (1 + 0)
      - Return 3 (2 + 1)
    - Return de 6 (3 + 3)
  - Return de 10 (4 + 6)

- Return 15 (5+10)

Saving the different values during these calls is done in the program execution stack.

> The **execution stack** of the current program is a memory location intended to store the parameters, local variables as well as the return addresses of the functions currently being executed. It works according to the LIFO (Last-In-First-Out) principle: last in first out.

The stack of the PP program calling the $SumR(3)$ function will contain the address of the calling instruction, i.e. @RetPP, and the values of its variables (VV) at the time of the call. Likewise, a call from the $SumR$ function will add to the stack the address of the $SumR$ function, the address of the call instruction, i.e. @RetR and the value of n. The evolution of the stack will therefore be as follows:

					SumR, @RetR, n=0						
				SumR, @RetR, n=1	SumR, @RetR, n=1	SumR, @RetR, n=1					
			SumR, @RetR, n=2	SumR, @RetR, n=2	SumR, @RetR, n=2	SumR, @RetR, n=2	SumR, @RetR, n=2				
		SumR, @RetR, n=3	SumR, @RetR, n=3	SumR, @RetR, n=3	SumR, @RetR, n=3	SumR, @RetR, n=3	SumR, @RetR, n=3	SumR, @RetR, n=3	SommeR, @RetR, n=3		
	PP, @RetPP, VV	PP, @RetPP, VV	PP, @RetPP, VV	PP, @RetPP, VV	PP, @RetPP, VV	PP, @RetPP, VV	PP, @RetPP, VV	PP, @RetPP, VV	PP, @RetPP, VV	PP, @RetPP, VV	
t=0	1	2	3	4	5	6	7	8	9	10	11

Attention ! The stack has a fixed size, improper use of recursion can lead to stack overflow.

2.3 Termination

As in the case of a loop, you need a stopping case where you do not make a recursive call. The evolution of recursive calls must necessarily lead to an initial case where no recursive calls are made. Otherwise the calls will be infinite and there will be a stack overflow. The following function, for example, produces a stack overflow with the call $SumR(2)$ since n

starts from 2 and is incremented with each call and the condition $(n = 0)$ will never be satisfied :

```
Function SumR( n : integer) : integer;
Begin
    If (n = 0) Then
        SumR ← 0;
    Else
        SumR ← n + SumR(n + 1);
    End If;
End;
```

2.4 Examples

2.4.1 The factorial

Factorial n, denoted $n!$, is the product of all integers from 1 up to n, i.e. $1 \times 2 \times 3 \times 4 \times ... \times n$

For example : 5! = 1 * 2 * 3 * 4 * 5 = 120

Then : $n! = (n-1)! \times n$ and

So for this example : 5! = 4!*5

```
Function FactI( n : integer) : integer;
Var  M,i : integer;
Begin
    M ← 1;
    For i From 1 to n Do
        M ← M × i;
    End For;
    FactI ← M;
End;
// Iterative factorial function
```

```
Function FactR( n : integer) : integer;
Begin
    If (n ≤ 1) Then
        FactR ← 1;
    Else
        FactR ← n × FactR(n − 1);
    End If;
End;
// Recursive factorial function
```

2.4.2 GCD

The Greatest Common Divisor (GCD) of two integers A and B is calculated by making successive subtractions between A and B until arriving at two equal integers which represent the GCD.

For example:

A	42	18	18	12	6
B	24	24	6	6	6

We notice that the GCD of 42 and 18 is the same as that of 18 and 24 and the same as that of 18 and 6...and so on, hence the recursive definition:

$$GCD(A,B) = \begin{cases} A & if\ A = B \\ GCD(A-B,B) & if\ A > B \\ GCD(A,B-A) & else \end{cases}$$

And hence the recursive function R_GCD:

```
Function It_GCD( A, B : integer) : integer;
Begin
    While (A ≠ B) Do
        If (A > B) Then
            A ← A − B
        Else
            B ← B − A
        End If;
    End While;
    It_GCD ← A;
End;

// Iterative GCD function
```

```
Function R_GCD( A, B : integer) : integer;
Begin
    If (A = B) Then
        R_GCD ← A;
    Else
        If (A > B) Then
            R_GCD ← R_GCD(A − B, B);
        Else
            R_GCD ← R_GCD(A, B − A);
        End If;
    End If;
End;
// Recursive GCD function
```

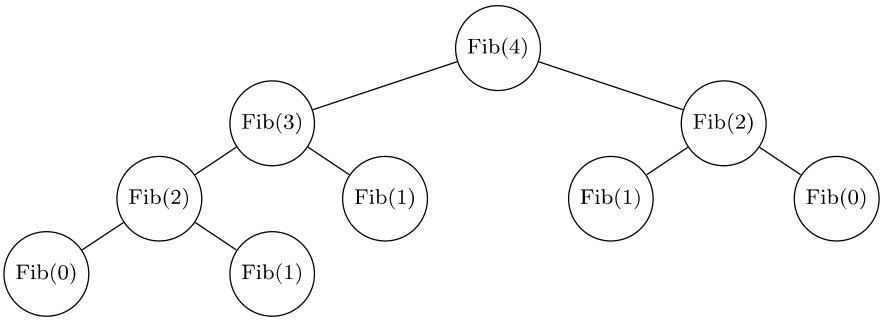
2.4.3 Fibonacci sequence

The calculation of the Fibonacci sequence seen previously is recursive in nature. Writing the corresponding algorithm is intuitive:

```
Function Fib( n : integer) : integer;
Begin
    If (n ≤ 1) Then
        Fib ← 1;
    Else
        Fib ← Fib(n − 1) + Fib(n − 2);
    End If;
End;

// Recursive Fibonacci Function
```

The execution of the $Fib(4)$ call can be imagined as follows:



Note: The recursive solution is not always the most efficient. For example, in the case of this function the calculation of Fib(1) is done several times.

2.5 Importance of order of recursive calls

The order of a recursive call in relation to the other instructions of an algorithm considerably influences the result obtained. The following two procedures illustrate an example:

```
Procedure Depict( n : integer);
Begin
    If (n ≥ 1) Then
        Write(n);
        Depict(n-1);
    End If;
End;
```

Results for n = 5: 5 4 3 2 1

```
Procedure Depict( n : integer);
Begin
    If (n ≥ 1) Then
        Depict(n-1);
        Write(n);
    End If;
End;
```

Results for n = 5: 1 2 3 4 5

2.6 Types of recursion

Depending on the number of calls and their location the recursion can be of several types:

2.6.1 Simple recursion

The procedure or function is called itself only once without a body such as the *FactR* function in the previous example.

2.6.2 Multiple recursion

Recursion is said to be multiple if the procedure or function contains more than one recursive call in its body such as the *Fib* function in the previous example.

2.6.3 Nested recursion

Recursion is said to be nested if the recursive call contains a parameter which is also a recursive call such as:

$$\mathrm{F} \leftarrow \mathrm{F}(\mathrm{n}, \mathrm{F}(\mathrm{n}));$$

2.6.4 Cross recursion

Recursion is said to be crossed between two procedures (function) if each of them calls the other as in the following example:

```
Function Even( n : integer) : boolean;
Begin
    If (n = 0) Then
        Even ← true;
    Else
        Even ← Odd(n-1);
    End If;
End;

Function Odd( n : integer) : boolean;
Begin
    If (n = 0) Then
        Odd ← false;
    Else
        Even ← Even(n-1);
    End If;
End;
```

Although neither function calls itself, recursion exists. It is also called mutual.