

# Linguagens e Ambientes de Programação (Aula Teórica 4)

**LEI - Licenciatura em Engenharia Informática**

**João Costa Seco ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))**



NOVA SCHOOL OF  
SCIENCE & TECHNOLOGY

# Agenda

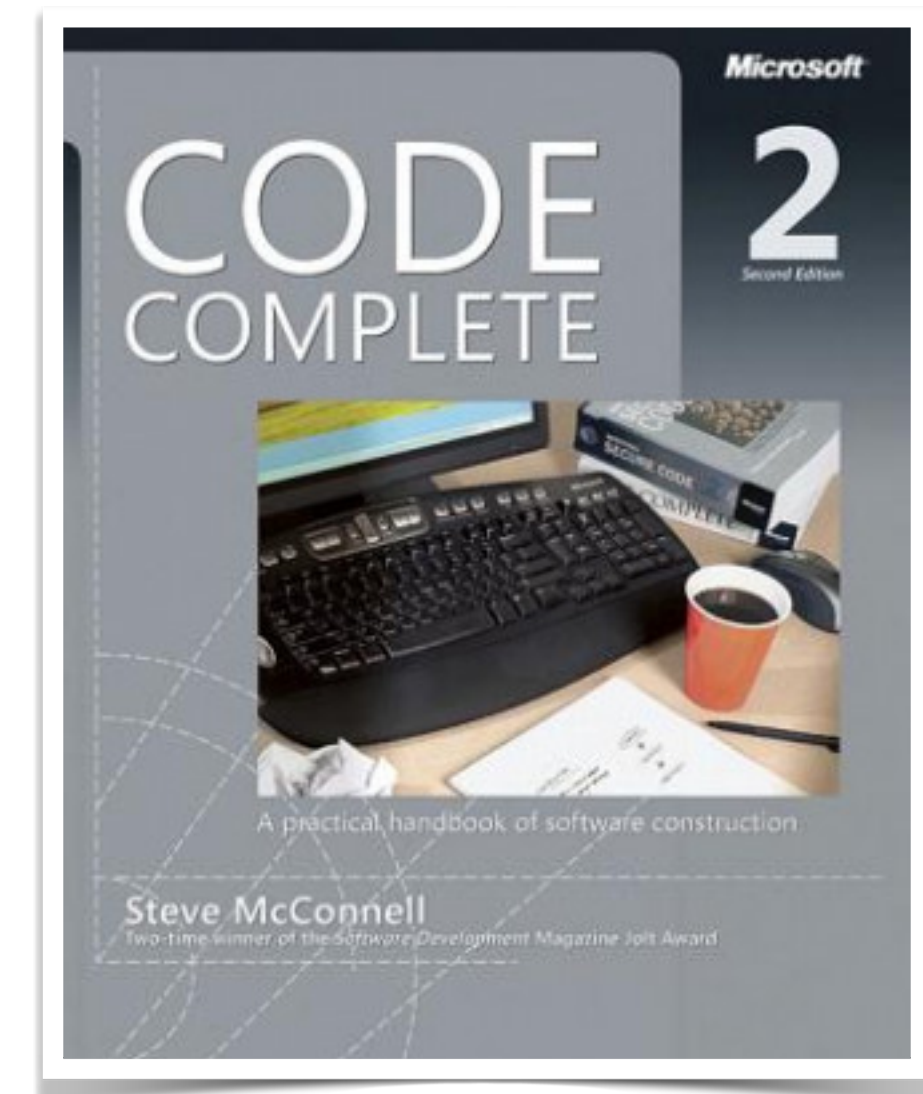
---

- Pré- e pós-condições.
- Correção de programas.
- Testes unitários.
- Funções como valores.
- Composição.

Correção

# Um slide sobre correção

- Programação por contrato (Design by Contract - Bertrand Meyer, 1986)
  - Verifica que todas as chamadas cumprem as pré-condições (se não, pára), garante as pós-condições.
- Programação com pré-condições e pós-condições (Lógica de Hoare, 1969)
  - Não assume o cumprimento das pré-condições, mas...
  - dá garantias de cumprimento das pós-condições no caso de cumprimento das pré-condições e em caso de terminação.
  - Há ferramentas que garantem a correção dos programas (os contratos) em tempo de compilação.
- Programação defensiva
  - Não assume nada sobre o input e testa todas as suas “pré-condições” em tempo de execução. Tem outputs/excepções para todas as entradas possíveis. (A verdadeira pré-condição é “true”).
  - Pode declarar pós-condições mas normalmente não testa os seus resultados.


$$\{ A \} P \{ B \}$$


# Mais um slide sobre correção (exemplos)

- Pré e pós-condições informais vs.

```
(** [fib n] is the [n]th fibonnaci number  
| requires n > 0 *)  
let rec fib n = if n <= 2 then 1 else fib (n - 1) + fib (n - 2)
```

- Uma linguagem que verifica formalmente a especificação de uma função/método (Dafny).
- Existem mais ferramentas de verificação: Verifast, Why3, Infer, Cameleer, ...

```
let rec fib n =  
| if n <= 2 then 1 else fib (n - 1) + fib (n - 2)  
(*@ requires n > 0 *)
```

```
function fib(n: nat): nat  
{  
    if n == 0 then 0  
    else if n == 1 then 1  
    else fib(n - 1) + fib(n - 2)  
}  
method ComputeFib(n: nat) returns (b: nat)  
    ensures b == fib(n)  
{  
    ...  
}
```

# Testes unitários



# Testes unitários

## Module Lap

```
module Lap: sig .. end
```

The first special comment of the file is the comment associated with the whole module. This is module LAP with sample code for LAP 2024

```
val fact : int -> int
```

fact n is the factorial of n Requires: n >= 0

```
val even : int -> bool
```

even x is true if x is even, false otherwise Requires: x >= 0

```
val odd : int -> bool
```

odd x is true if x is odd, false otherwise Requires: x >= 0

```
let _ = assert (true = even 2)
let _ = assert (false = odd 2)
let _ = assert (true = odd 57)
let _ = assert (false = even 57)
```

```
let _ = assert (1 = fact 0)
let _ = assert (720 = fact 6)
```

```
let _ = assert (1 = fib 1)
let _ = assert (1 = fib 2)
let _ = assert (2 = fib 3)
let _ = assert (3 = fib 4)
let _ = assert (5 = fib 5)
let _ = assert (8 = fib 6)
```

```
"two is even" >:: (fun _ -> assert_equal true (even 2))
```

# Testes unitários

```
module Lap: sig ..
  The first special c
  whole module. Th

val fact : int ->
  fact n is the fac

val even : int ->
  even x is true if

val odd : int -> b
  odd x is true if x

val sum : 'a -> 'b
  sum lst is the s
```

```
let tests =
[
  "two is even" >:: (fun _ -> assert_equal true (even 2));
  "two is not odd" >:: (fun _ -> assert_equal false (odd 2));
  "fifty seven is odd" >:: (fun _ -> assert_equal true (even 2));
  "fifty seven is not even" >:: (fun _ -> assert_equal false (odd 2));
  "factorial of 0 is 1" >:: (fun _ -> assert_equal 1 (fact 0));
  "factorial of 6 is 720" >:: (fun _ -> assert_equal 720 (fact 6));
  "fibonacci 1 is 1" >:: (fun _ -> assert_equal 1 (fib 1));
  "fibonacci 2 is 1" >:: (fun _ -> assert_equal 1 (fib 2));
  "fibonacci 3 is 2" >:: (fun _ -> assert_equal 2 (fib 3));
  "fibonacci 4 is 3" >:: (fun _ -> assert_equal 3 (fib 4));
  "fibonacci 5 is 5" >:: (fun _ -> assert_equal 5 (fib 5));
  "fibonacci 6 is 8" >:: (fun _ -> assert_equal 8 (fib 6))
]
```

```
let test_suit = "test suite for sum" >::: tests
```

```
let _ = run_test_tt_main test_suit
```

```
= even 2)
e = odd 2)
= odd 57)
e = even 57)

fact 0)
= fact 6)

fib 1)
fib 2)
fib 3)
fib 4)
fib 5)
fib 6)
```

```
jcs@joaos-imac lap2024 % dune exec ./test.exe
.....
Ran: 12 tests in: 0.11 seconds.
OK
```



# Funções++

# Funções como valores

- As funções são valores da linguagem, podem ser como qualquer outro valor.
- As funções podem ser usadas como parâmetros para outras funções.

```
let f x y = x+y
```

✓ 0.2s

```
val f : int → int → int = <fun>
```

```
let g x y = x * y
```

✓ 0.0s

```
val g : int → int → int = <fun>
```

```
let h i = 1 + i 1 1
```

✓ 0.0s

```
val h : (int → int → int) → int = <fun>
```

```
h f
```

✓ 0.0s

```
- : int = 3
```

```
h g
```

✓ 0.0s

```
- : int = 2
```

```
h (fun x y → x - y)
```

✓ 0.0s

```
- : int = 1
```

# Funções como valores

```
let f x = if x = 1 then fun x y → x + y else fun x y → x * y
```

✓ 0.0s

```
val f : int → int → int → int = <fun>
```

- As funções podem ser usadas como resultados de outras funções (já tínhamos visto).
- As funções podem “capturar” nomes do contexto de definição, chamam-se “closures”.

```
let g = f 1 in g 2 3
```

✓ 0.0s

```
- : int = 5
```

```
let g = f 2 in g 2 3
```

✓ 0.0s

```
- : int = 6
```

# Funções como valores

```
let f x = if x = 1 then fun x y → x + y else fun x y → x * y
```

✓ 0.0s

```
val f : int → int → int → int = <fun>
```

- As funções podem ser usadas como resultados de outras funções (já tínhamos visto).
- As funções podem “capturar” nomes do contexto de definição, chamam-se “closures”.

```
(f 1) 2 3
```

✓ 0.0s

```
- : int = 5
```

```
(f 2) 2 3
```

✓ 0.0s

```
- : int = 6
```

# Funções como valores

- A aplicação de funções aplica-se da esquerda para a direita, logo não é preciso usar parâmetros.

```
f 3 2 1
```

```
✓ 0.0s
```

```
- : int = 2
```

```
f 3 2 1 = (f 3) 2 1 && (f 3) 2 1 = ((f 3) 2) 1
```

```
✓ 0.0s
```

```
- : bool = true
```



# Composição de funções

```
let comp (f:int → int) (g:int->int) = fun x → f (g (x))
```

```
let dup = fun x → x + x
```

```
let quad = comp dup dup
```

```
let x = quad 2
```

✓ 0.0s

```
val comp : (int → int) → (int → int) → int → int = <fun>
```

```
val dup : int → int = <fun>
```

```
val quad : int → int = <fun>
```

```
val x : int = 8
```