Linguagens e Ambientes de Programação (Aula Teórica 4)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Agenda

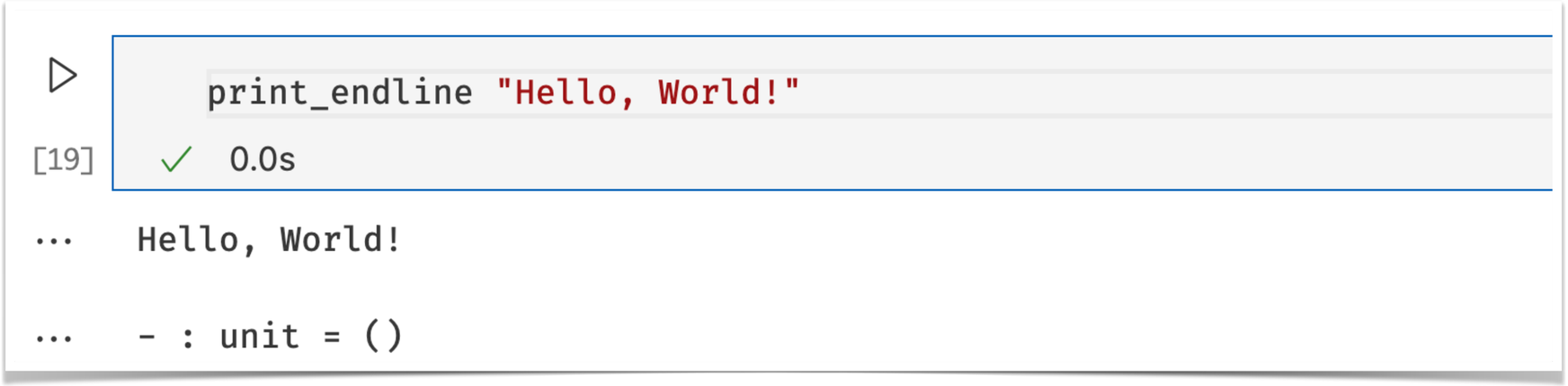
- Input/output básico
- Unit, Sequências, como ignorar valores intermédios
- Documentação
- Pré- e pós-condições.
- Correção de programas.
- Testes unitários.
- Funções como valores.
- Composição.

Input/Output & Unit

Input/Output Básico

- O tipo unit só faz sentido em expressões que têm efeitos laterais, a impressão é um exemplo típico disso

- `print_endline`
- `print_string`
- `print_char`
- `print_int`
- `print_float`



Declaração como operador de sequência.

- declarações que ignoram resultados

```
let () = print_string "hello, " in print_endline "world!"
```

[14] ✓ 0.0s

```
hello, world!

- : unit = ()
```

```
let _ = uma_funcao_com_efeitos_laterais () in 4
```

[13] ✓ 0.0s

```
- : int = 4
```

```
print_string "hello, "; print_endline "world!"
```

[15] ✓ 0.0s

```
hello, world!

- : unit = ()
```

Declaração como operador de sequência.

- declarações que ignoram resultados

```
[18]  1; 2
      ✓ 0.0s

...   File "[18]", line 1, characters 0-1:
      1 | 1; 2
          ^

      Warning 10 [non-unit-statement]: this expression should have type unit.
      File "[18]", line 1, characters 0-1:
      1 | 1; 2
          ^

      Warning 10 [non-unit-statement]: this expression should have type unit.

...   - : int = 2
```

Declaração como operador de sequência.

- declarações que ignoram resultados

```
let () = print_string "hello, " in print_endline "world!"
```

[14] ✓ 0.0s

```
hello, world!
```

```
(ignore 1); 2
```

[21] ✓ 0.0s

```
- : int = 2
```

```
print_string "hello, "; print_endline "world!"
```

[15] ✓ 0.0s

```
hello, world!

- : unit = ()
```

Documentação

OCamldoc - documentação real em ocaml

- A documentação do código serve para ajudar a ler o código de uma função, mas também para perceber a funcionalidade de um módulo inteiro.

```
(** 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 *)

(** [fact n] is the factorial of [n]
    requires: [n >= 0] *)
let rec fact x = if x = 0 then 1 else x * fact(x-1)

(** [even x] is true if [x] is even, false otherwise
    requires: [x >= 0] *)
let rec even x = if x = 0 then true else if x = 1 then fal

(** [odd x] is true if [x] is odd, false otherwise
    requires: [x >= 0] *)
and odd x = if x = 0 then false else if x = 1 then true el
```

```
jcs@joaos-imac lap2024 % ocamldoc -html lap.ml
```

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`

OCamldoc - documentação real em ocaml

- Tags que se podem usar:

2.5 Documentation tags (@-tags)

Predefined tags

The following table gives the list of predefined @-tags, with their syntax and meaning.

`@author` *string*	The author of the element. One author per `@author` tag. There may be several `@author` tags for the same element.
`@deprecated` *text*	The *text* should describe when the element was deprecated, what to use as a replacement, and possibly the reason for deprecation.
`@param` *id text*	Associate the given description (*text*) to the given parameter name *id*. This tag is used for functions, methods, classes and functors.
`@raise` *Exc text*	Explain that the element may raise the exception *Exc*.
`@return` *text*	Describe the return value and its possible values. This tag is used for functions and methods.
`@see <` *URL* `>` *text*	Add a reference to the *URL* with the given *text* as comment.
`@see '`*filename*`'` *text*	Add a reference to the given file name (written between single quotes), with the given *text* as comment.
`@see "`*document-name*`"` *text*	Add a reference to the given document name (written between double quotes), with the given *text* as comment.
`@since` *string*	Indicate when the element was introduced.
`@before` *version text*	Associate the given description (*text*) to the given *version* in order to document compatibility issues.
`@version` *string*	The version number for the element.

https://v2.ocaml.org/manual/ocamldoc.html

OCamldoc - preconditions and post-conditions

- A documentação de uma função também pode/deve indicar as suas pré-condições e pós-condições. Estas condições são informais.

```
(** 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 *)

(** [fact n] is the factorial of [n]
    requires: [n >= 0] *)
let rec fact x = if x = 0 then 1 else x * fact(x-1)

(** [even x] is true if [x] is even, false otherwise
    requires: [x >= 0] *)
let rec even x = if x = 0 then true else if x = 1 then false else odd(x-1)

(** [odd x] is true if [x] is odd, false otherwise
    requires: [x >= 0] *)
and odd x = if x = 0 then false else if x = 1 then true else even(x-1)
```

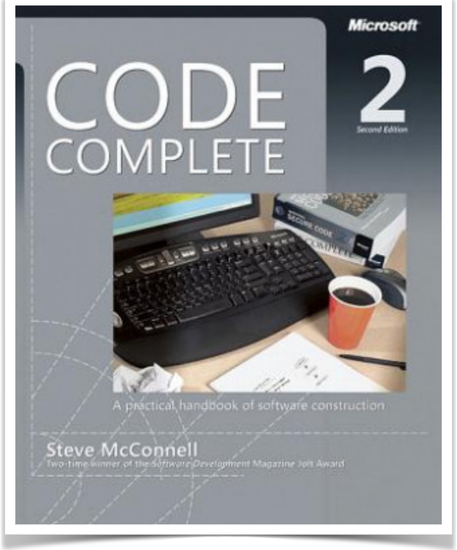
Correção

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.

Eiffel

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



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)
= odd 2)
= odd 57)
= 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
```

https://cs3110.github.io/textbook/chapters/data/ounit.html

Funções (outra vez)

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
```

Agenda

- Input/output básico
- Unit, Sequências, como ignorar valores intermédios
- Documentação
- Pré- e pós-condições.
- Correção de programas.
- Testes unitários.
- Funções como valores.
- Composição.