

Linguagens e Ambientes de Programação (Aula Teórica 3)

LEI - Licenciatura em Engenharia Informática

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



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Agenda para hoje

- Declaração de funções, com e sem parâmetros.
- Avaliação de expressões por substituição.
- Funções como valores (primeira vez).
- Avaliação parcial de funções
- Input/output básico
- Unit, Sequências, como ignorar valores intermédios
- Documentação

Funções

Declaração e chamada (aplicação) de funções

- A declaração de um nome (**f**) é limitada ao corpo da declaração (**e2**)
- O nome (**f**) não é visível na expressão que define o valor (**e1**)
- Os parâmetros são listados na declaração.

let f x = e1 in e2

let f x = x + 1 in f (1 + 1)

Declaração e chamada (aplicação) de funções

- A declaração de um nome (**f**) é limitada ao corpo da declaração (**e2**)
- O nome (**f**) não é visível na expressão que define o valor (**e1**)
- Os parâmetros são listados na declaração.
- As funções “sem parâmetros” têm um parâmetro de tipo **unit**.

`let x = 1 in let f () = 1 + x in f ()`

Declaração e chamada (aplicação) de funções

- A declaração de um nome (**f**) é limitada ao corpo da declaração (**e2**)
- O nome (**f**) não é visível na expressão que define o valor (**e1**)
- Os parâmetros são listados na declaração.
- A declaração com parâmetros é uma forma sintática alternativa à utilização de valores de tipo função (o símbolo seta é composto por dois caracteres ->)

let f = fun x → x +1 in f (1 + 1)

Definição e chamada (aplicação) de funções

- A aplicação de funções pode ser definida por substituição do parâmetro pelo valor do argumento.
- OCaml implementa uma estratégia de avaliação *call-by-value*, o que quer dizer que os argumentos são avaliados antes de expandir o corpo da função.

```
(fun x → x + 1) (1 + 1)
  (fun x → x + 1) 2
    2 + 1
      3
```

Declaração recursivas (âmbito)

- A declaração de um nome (**x**) é visível no corpo da declaração (**e2**) e na expressão de definição do nome (**e1**).

let rec x = e1 in e2

(* [fact x] computes the factorial of x
Requires: [x >= 0] *)

let rec fact x = if x = 0 then 1 else x * fact (x - 1)

Declaração recursivas (âmbito)

- A declaração de um nome (**x**) é visível no corpo da declaração (**e2**) e na expressão de definição do nome (**e1**).

let rec x = e1 in e2

```
(* [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)
```

Declarações mutuamente recursivas em C

- Tem que se declarar a função sem a definir.

```
bool odd(int x);
```

```
bool even(int x) {  
    if( x == 0 ) {  
        return false;  
    } else if( x == 1 ) {  
        return false;  
    } else {  
        return odd(x-1);  
    }  
}
```

```
bool odd(int x) {  
    if( x == 0 ) {  
        return false;  
    } else if( x == 1 ) {  
        return true;  
    } else {  
        return even(x-1);  
    }  
}
```

Declaração e chamada (aplicação) de funções

- A declaração de um nome (**f**) é limitada ao corpo da declaração (**e2**)
- O nome (**f**) não é visível na expressão que define o valor (**e1**)
- Os parâmetros são listados na declaração.

let f x y = x + y in f 1 1

Declaração e chamada (aplicação) de funções

- A declaração de um nome (**f**) é limitada ao corpo da declaração (**e2**)
- O nome (**f**) não é visível na expressão que define o valor (**e1**)
- Os parâmetros são listados na declaração.

let f = fun x y → x + y in f 1 1

Declaração e chamada (aplicação) de funções

- A declaração de um nome (**f**) é limitada ao corpo da declaração (**e2**)
- O nome (**f**) não é visível na expressão que define o valor (**e1**)
- Os parâmetros são listados na declaração.

```
let f = fun x → fun y → x + y in f 1 1
```

Avaliação parcial de funções

- Funções com vários parâmetros são na realidade composição de várias funções.
- Os parâmetros podem ser instanciados um de cada vez resultando em funções que aceitam os restantes parâmetros até produzir um resultado final.

```
[9] let add x y = x + y
    ✓ 0.0s
... val add : int → int → int = <fun>
```

```
[10] add 2 3
    ✓ 0.0s
... - : int = 5
```

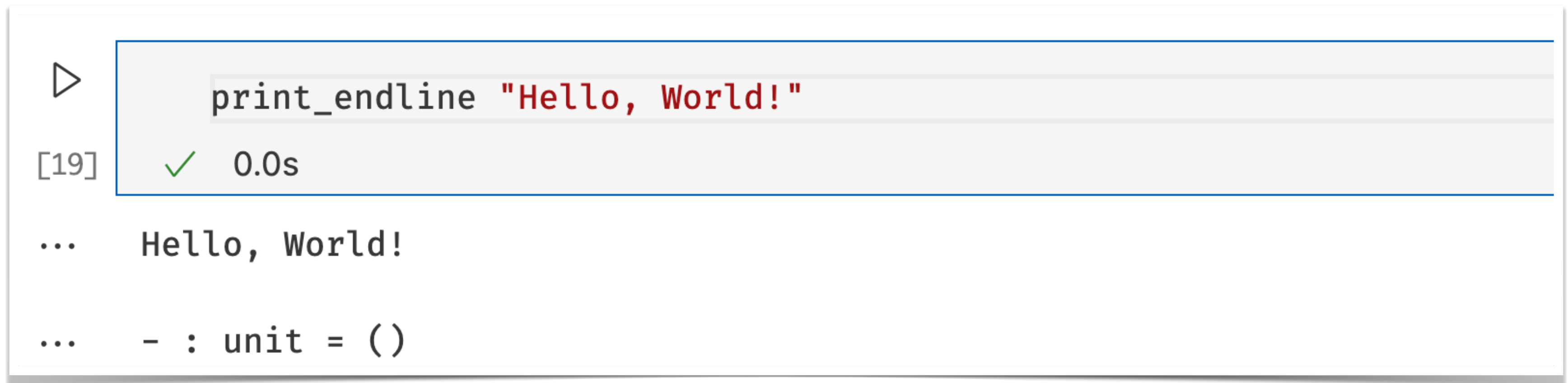
```
▷ [12] let add1 = add 1
    ✓ 0.0s
... val add1 : int → int = <fun>
```

```
▷ [13] add1 2
    ✓ 0.0s
... - : int = 3
```

Input/Output & Unit

Input/Output Básico

- O tipo `unit` só faz sentido em expressões que têm efeitos laterais, a impressão é um exemplo típico
- `print_endline`
- `print_string`
- `print_char`
- `print_int`
- `print_float`



```
▶ print_endline "Hello, World!"  
[19] ✓ 0.0s  
... Hello, World!  
... - : unit = ()
```


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

OCaml doc - 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 false
```

```
(** [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
```

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
```

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


OCamlDoc - Tags

- As tags fornecem metadados sobre funções, parâmetros, valores de retorno, exceções, etc.
- Ajudam a organizar a informação, tornando mais fácil gerar documentação clara e consistente.
- São colocadas dentro de comentários de documentação, ou seja, (`** ... *`), começando com um `@`.

2.5 Documentation tags (@-tags)

Predefined tags

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

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

OCaml doc - Pré e Pós condições

- 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)
```