Declarations

Variable declarations

- Global declarations

```
let x = 42
```

```
[12] ✓ 0.0s
```

```
val x : int = 42
```

- Local declarations

```
let x = 42 in (string_of_int x)^": the ultimate question of life, the universe, and everything"
```

```
[13] ✓ 0.0s
```

```
- : string =
"42: the ultimate question of life, the universe, and everything"
```

```
let x = 1 in (let x = 2 in x + 1) + x
```

```
[14] ✓ 0.0s
```

```
- : int = 4
```

```
let x = 1 in let y = 2 in x + y
```

```
[15] ✓ 0.0s
```

```
- : int = 3
```

Scoping

- A declaration of a name (`x`) is limited to the body of the declaration (`e2`).
- It is not, for instance, visible in the expression that defines its value (`e1`).

```
let x = e1 in e2
```

```
let y = let y = 1 in y + 1 in let y = y + 2 in y + 2
```

- Declarations follow the principle of name irrelevance, meaning that the chosen names should not affect the evaluation of an expression.

Scoping

- A declaration of a name (`x`) is limited to the body of the declaration (`e2`).
- It is not, for instance, visible in the expression that defines its value (`e1`).

```
let x = e1 in e2
```

```
let y = let y = 1 in y + 1 in let y = y + 2 in y + 2
```

- Declarations follow the principle of name irrelevance, meaning that the chosen names should not affect the evaluation of an expression.

Scoping of local variables

```
let x =
  let y = 1 in
  let z = 2 in
  y + z
in
let w = 3+x in
  w + x
```

x

Scoping of local variables

```
let x =
   let y = 1 in
   let z = 2 in
   y + z
in
let w = 3+x in
   w + x
```

Y

Scoping of local variables

```
let x =
  let y = 1 in
  let z = 2 in
  y + z                                    z
in
let w = 3+x in
  w + x
```

Scoping of local variables

```
let x =
  let y = 1 in
  let z = 2 in
  y + z
in
let w = 3+x in
  w + x                                    w
```

Evaluation by substitution

```
let x =
  let y = 1 in
  let z = 2 in
  y + z
in
let w = 3+x in
   w + x
```

Evaluation by substitution

```
let x =
  let y = 1 in
  let z = 2 in
  y + z
in
let w = 3+x in
  w + x
```

Evaluation by substitution

```
let x =

  let z = 2 in
  1 + z
in
let w = 3+x in
   w + x
```

Evaluation by substitution

```
let x =

  let z = 2 in
  1 + z
in
let w = 3+x in
  w + x
```

Evaluation by substitution

```
let x =

  1 + 2
in
let w = 3+x in
  w + x
```

Evaluation by substitution

```
let x =

  3
in
let w = 3+x in
  w + x
```

Evaluation by substitution

```
let w = 3+3 in
  w + 3
```

Evaluation by substitution

```
let w = 3+3 in
  w + 3
```

Evaluation by substitution

```
let w = 6 in
  w + 3
```

Evaluation by substitution

```
6 + 3
```

Evaluation by substitution

9

Programming Languages and Environments (Lecture 3)

LEI - Licenciatura em Engenharia Informática

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

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Syllabus

- Function declaration, with and without parameters.
- Evaluation of expression by substitution.
- Functions as values.
- Function partial applications.

Functions

Declaration and application of functions

- The name declaration (`f`) is limited to the body (`e2`).
- The name (`f`) is not visible in the expression that defines the value (`e1`).
- Parameters are listed in the declaration.

```
let f x = e1 in e2
```

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

Declaration and application of functions

- The name declaration (`f`) is limited to the body (`e2`).
- The name (`f`) is not visible in the expression that defines the value (`e1`).
- Parameters are listed in the declaration.
- Functions with "no parameters", have a parameter of type unit.

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

Declaration and application of functions

- The name declaration (`f`) is limited to the body (`e2`).
- The name (`f`) is not visible in the expression that defines the value (`e1`).
- Parameters are listed in the declaration.
- Declarations with parameters is a syntactic alternative to using functions as values (the arrow type is composed by two characters ->).

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

Definition and evaluation of functions

- The application of functions can be defined by the substitution of the parameter by the value of the argument.
- OCaml implements *call-by-value* evaluation strategy, meaning that the arguments are evaluated before expanding the body of the function.

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

Recursive definitions (scoping)

- The declaration of a name (`x`) is visible in the body of the declaration (`e2`) and in the body of the declaration (`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)
```

Mutually recursive declarations (scoping)

- The declaration of a name (`x`) is visible in the body of the declaration (`e2`) and in the body of the declaration (`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)
```

Mutually recursive declarations in C

- Declare a function without defining it.

```c
bool odd(int x);

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

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

Declaration and application of functions

- The name declaration (`f`) is limited to the body (`e2`).
- The name (`f`) is not visible in the expression that defines the value (`e1`).
- Parameters are listed in the declaration.

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

Declaration and application of functions

- The name declaration (`f`) is limited to the body (`e2`).
- The name (`f`) is not visible in the expression that defines the value (`e1`).
- Parameters are listed in the declaration.

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

Declaration and application of functions

- The name declaration (`f`) is limited to the body (`e2`).
- The name (`f`) is not visible in the expression that defines the value (`e1`).
- Parameters are listed in the declaration.

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

Partial evaluation of functions

- A function with multiple parameters is essentially the composition of multiple functions.
- Parameters can be instantiated one at a time, resulting in partial applications until the evaluation is complete.

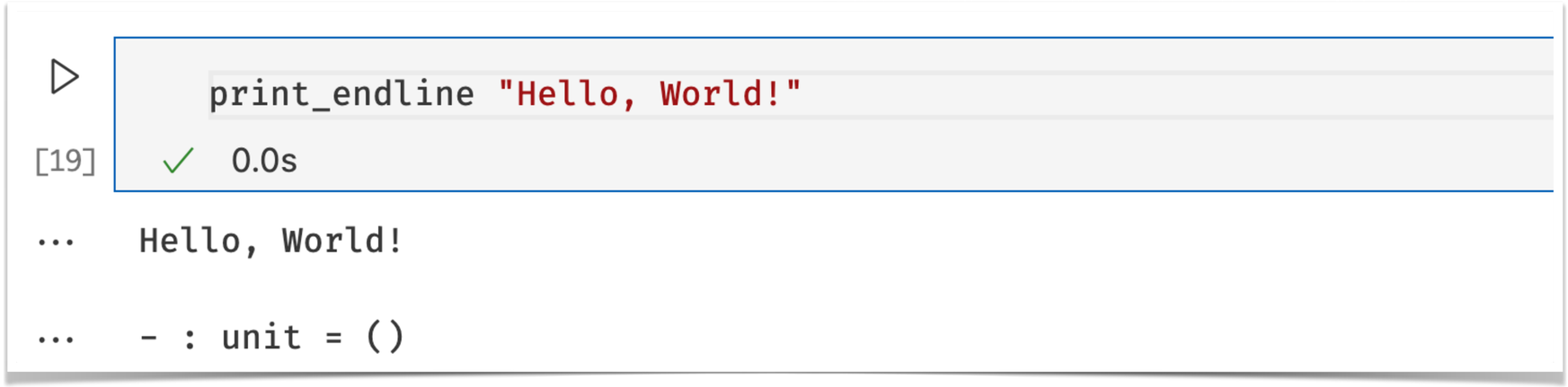
```
[9]  let add x y = x + y
     ✓ 0.0s
...  val add : int → int → int = <fun>

[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

Basic Input/Output

- Functions with return type unit are typically have side-effects. Printing to the standard output is one such example.

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



Declarations with sequencing

- Declarations that discard the result.

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

[14] ✓ 0.0s

```
hello, world!
- : unit = ()
```

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

[13] ✓ 0.0s

```
- : int = 4
```

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

[15] ✓ 0.0s

```
hello, world!
- : unit = ()
```

Declarations with sequencing

- Declarations that discard the result.

```
[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
```

Declarations with sequencing

- Declarations that discard the result.

```
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 = ()
```

Documentation

OCamldoc

- Documentation helps with reading the code of a function, but also with understanding the functionality of an entire module.

```
(** 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 - Tags

- Tags provide metadata about functions, parameters, return values, exceptions, etc.
- They help organize information, making it easier to generate clear and consistent documentation.
- Tags are placed within documentation comments, i.e. `(** ... *)`, starting with an `@`.

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 - Pre and Post-conditions

- The documentation of a function cal also state its pre and post-conditions. Though these conditions are informal and not enforced.

```
(** 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)
```

Summary

- Name declarations
- Function declaration, with and without parameters.
- Evaluation of expression by substitution.
- Functions as values.
- Function partial applications.