

# Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 5 - Sequence 3: Sequences and iterations



# One after the other, and round and round

We have functions with side effect

We can now

- ▶ create *sequences*
- ▶ write *loops*

# Sequences of expressions

What if we want to print several values?

```
# let _ = print_int 1 in let _ = print_int 2 in print_int 3;;
```

```
123- : unit = ()
```

More concisely

```
# print_int 1; print_int 2; print_int 3;;
```

```
123- : unit = ()
```

# Sequences of expressions

The *expression sequence*

$e_1; e_2; \dots ; e_n$

- ▶ evaluates each  $e_i$  in turn
- ▶ drops all the results but the last one
- ▶ returns the result of  $e_n$
- ▶ *all intermediate expressions should be of type unit*
- ▶ otherwise the *OCaml* compiler prints a warning

# Correct parsing of sequences: `begin ... end`

Beware of the `;` precedence

```
# if 3>5 then print_string "3 is greater than 5";  
              print_string ".";;
```

```
.- : unit = ()
```

- ▶ use `( e1; e2; ... en )`
- ▶ or the more verbose `begin e1; e2; ... en end`

```
# if 3>5 then  
  begin print_string "3 is greater than 5";  
         print_string "."  
  end;;
```

```
- : unit = ()
```

# Iterations

What if we want to print all integers from 1 to 10?

```
# let foreach starti endi f =  
  let rec aux =  
    function n -> if n <= endi  
                  then (f n; aux (n+1))  
                  else ()  
  in aux starti;;
```

```
val foreach : int -> int -> (int -> 'a) -> unit = <fun>
```

```
# foreach 1 10 (fun i -> print_int i);;
```

```
12345678910- : unit = ()
```

# The for loop

More concisely

```
# for i=1 to 10 do
  print_int i
done;;
```

```
12345678910- : unit = ()
```

**for** *id* = *e1* **to** *e2* **do** *e3* **done**

- ▶ the *loop identifier* *id* takes all integer values from *e1* to *e2* in turn, and **cannot be otherwise altered**
- ▶ the *loop body* *e3* is evaluated for each value of *id*
- ▶ the type of the for loop is `unit`
- ▶ the type of the loop body is expected to be `unit`
- ▶ otherwise the *OCaml* compiler prints a warning

# The for loop, alternative

We can also go backwards

```
# for i=10 downto 1 do
  print_int i
done;;
```

```
10987654321- : unit = ()
```

**for** *id* = *e1* **to** *e2* **downto** *e3* **done**

- ▶ the *loop identifier* *id* takes all integer values from *e1* down to *e2* in turn
- ▶ the *loop body* *e3* is evaluated for each value of *id*
- ▶ the type of the for loop is `unit`
- ▶ the type of the loop body is expected to be `unit`
- ▶ otherwise the *OCaml* compiler prints a warning



# The while loop

We can also write while loops

```
while e1 do e2 done
```

- ▶ the *condition* `e1` is evaluated
- ▶ if true, the *loop body* `e2` is evaluated, and the loop repeated
- ▶ if false, the loop stops
- ▶ the type of the while loop is `unit`
- ▶ the type of the loop body is expected to be `unit`
- ▶ otherwise the *OCaml* compiler prints a warning

# Ignoring values

In some cases, we may really want to use in the body of a loop an expression that has not a `unit` type.

To make our intention explicit, and avoid the warning, we can use the `ignore` function:

```
# ignore;;
```

```
- : 'a -> unit = <fun>
```

# Summary

- ▶ Sequences `e1; e2; ... ; en`
- ▶ For loops `for id = e1 to e2 to/downto e3 done`
- ▶ While loops `while e1 do e2 done`
- ▶ *The body of the loops, and the intermediate expressions, should all be of type unit*
- ▶ *and we can use `ignore` to make sure they are*