CSCI 2041

Sebastiaan Joosten

September 7 2023: introduction to Ocaml

University of Minnesota

Outline today

- Printing
- Using ocamlc
- More on function application
- Debugging

Printing

- Here’s a way to ‘print’ things in utop:

```
utop # print_endline "Hello world"
;;
Hello world
- : unit = ()
```

- Note that the value returned is ()
- Its type is unit
- … so let’s write the hello world program and compile it

ocamlc

- Let’s try this file “hello.ml”:

```
print_endline "Hello world”
```

- In the terminal:

```
% ocamlc hello.ml
% ./a.out
Hello world
```

- … ugh, I wasn’t expecting that to work
- I was hoping to tell you that you can only write definitions at the top level. I guess that’s not true for my ocaml version.
- Not sure if this works on older ocaml versions…

ocamlc

- Let's 'repair' our file "hello.ml":

```
let () = print_endline "Hello world"
```

- In the terminal:

```
% ocamlc hello.ml
% ./a.out
Hello world
```

- … good, still works

let () = …

- What's going on here?

```
let () = print_endline "Hello world"
```

- Let's take a look at the print_endline function:

```
utop # print_endline;;
- : string -> unit = <fun>
```

- Takes a string as argument, produces something of type unit… What's unit?

- Unit is a type with only one value (hence it's name)
- It's value is ()

let () = …

- Unit:

```
utop # ();;
- : unit = ()
```

- Any unit value is ()
  - so functions … -> unit all give the same result
  - mathematically, all functions … -> unit are equal!
- However, we care about the *side effects* of most computations if the result is unit.
  - This breaks mathematical elegance!

Reasoning about ocaml code

- We pretend that side-effects don't exist
  - All important code runs without side-effects
- We introduce side-effects at the last possible moment
  - Code that has side-effects should document it well
  - … and ideally be of type … -> unit
  - Complicated reasoning should be done when side-effects are introduced

Typical ocaml program

```
let foo x y
  = (* some complicated function involving x and y *)
    x + y

let plus x y
  = string_of_int x ^ " plus " ^
    string_of_int y ^ " is " ^
    string_of_int (foo x y)

let user_number_x = read_int ()
let user_number_y = read_int ()
let () = print_endline
            (plus user_number_x user_number_y)
```

running printthings.ml

```
% ocamlc printingthings.ml -o pt
% ./pt
3
5
3 plus 5 is 8
```

Me typing

in utop…

```
utop # let foo x y
  = (* some complicated function involving x and y *)
    x + y

let plus x y
  = string_of_int x ^ " plus " ^
    string_of_int y ^ " is " ^
    string_of_int (foo x y);;
val foo : int -> int -> int = <fun>
val plus : int -> int -> string = <fun>

utop # plus 3 4;;
- : string = "3 plus 4 is 7"
```

in utop… (2)

```
utop # let foo x y
  = (* some code removed for readability *)

let user_number_x = read_int ()
let user_number_y = read_int ()
let () = print_endline
           (plus user_number_x user_number_y);;
3
4
3 plus 4 is 7
val foo : int -> int -> int = <fun>
val plus : int -> int -> string = <fun>
val user_number_x : int = 3
val user_number_y : int = 4
```

in utop… (3)

```
utop # #use "printingthings.ml";;
val foo : int -> int -> int = <fun>
val plus : int -> int -> string = <fun>
3
val user_number_x : int = 3
4
val user_number_y : int = 4
3 plus 4 is 7
```

Function application

```
let greet greeting name
    = greeting ^ " " ^ name ^ "!"
let greet_en = greet "Hello"
let greet_nl = greet "Hallo"

val greet : string -> string -> string = <fun>
val greet_en : string -> string = <fun>
val greet_nl : string -> string = <fun>

utop # greet "Hi" "Jane";;
- : string = "Hi Jane!"
utop # greet_en "Jane";;
- : string = "Hello Jane!”
```

Partial function application / currying

- If arguments to a function are ‘missing’, the result in OCaml is again a function. This is called partial function application.

- the -> binds to the right, so these are equivalent:

```
val greet : string -> string -> string = <fun>
val greet : string -> (string -> string) = <fun>
```

- Strictly speaking, ‘greet’ takes one argument, and produces a function.
- The idea that functions just take 1 argument (but sometimes produce functions) is called currying.

Function arguments

- We can pass functions as arguments:

```
let greet greeting name
     = greeting ^ " " ^ name ^ "!"
let greet_en = greet "Hello"
let greet_nl = greet "Hallo"
let with_jane greeter = greeter "Jane"

utop # with_jane greet_en;;
- : string = "Hello Jane!”
```

- We will do more of this in week 3.

Polymorphic functions

- Let’s take another look at this:

```
let with_jane greeter = greeter "Jane"
```

```
utop # let with_jane greeter = greeter "Jane";;
val with_jane : (string -> 'a) -> 'a = <fun>
```

Polymorphic functions

- The type 'a stands for *any* type!

```
val with_jane : (string -> 'a) -> 'a = <fun>

utop # with_jane print_endline;;
Jane
- : unit = ()
```

- In this example, that type is ‘unit’ (and the returned value is ())

An easier example of a polymorphic function

```
utop # let f x y = x;;
val f : 'a -> 'b -> 'a = <fun>
```

More examples of polymorphic functions

```
utop # let f2 x y = y;;
val f2 : 'a -> 'b -> 'b = <fun>
utop # let f3 f y = f y;;
val f3 : ('a -> 'b) -> 'a -> 'b = <fun>
utop # let f4 f y z = f z;;
val f4 : ('a -> 'b) -> 'c -> 'a -> 'b = <fun>
utop # let f5 f g x = f (g x);;
val f5 : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Note: f5 is function composition, and f3 is function application

Quiz time!

Debugging

- Rather than debugging, it's good to do the following:
  - Let your code compile without warnings
  - Write tests for functions:
    Make a script that provides input to functions, run it.
    Ideally have the script tell you if the output is 'right'.
  - Structure your code to improve testability (small tasks per function)
  - Analyze your code:
    Think!
  - Ask a TA for help / visit office hours
- If you run out of these things, there are more techniques!

Debugging with print statements

- One of the ways to do debugging, is to add ‘print’ statements. That has some downsides:
  - The type of the variable you’re interested in is not always representable (printable)
  - The print statements can mess up your code
  - If you want to print something more involved, it can even change the code

Debugging with a tracer

- One of the most tedious ways to debug is to use #trace. Textbook's example:

```
utop # let rec fib x = if x <= 1 then 1 else fib (x - 1) +
fib (x - 2);;
val fib : int -> int = <fun>
utop # #trace fib;;
fib is now traced.
utop # fib 2;;
fib <-- 2
fib <-- 0
fib --> 1
fib <-- 1
fib --> 1
fib --> 2
- : int = 2
utop # #untrace fib;;
fib is no longer traced.
```

Next week

- More on types:
  - Lists
  - Writing your own types
  - How does ocaml know the types?