Introduction to Functional Programming

http://cl-informatik.uibk.ac.at/teaching/ws05/idp/

Nao Hirokawa

office hours: Fridays 15:00 – 17:00 (3M09)

nao.hirokawa@uibk.ac.at

Purpose

PURPOSE

- unlearn imperative programming
- learn functional programming
- learn theories: λ and type system

Schedule

SCHEDULE

w	date	topic
7	November 25	introduction
8	December 2	higher-order functions, lists, trees
9	December 9	graphs, combinatorics
10	December 16	program reasoning
11	January 13	λ and interpreter
12	January 20	type system
13	January 27	exam part 2

EVALUATION

- [10×5 POINTS] homework weeks 7,8,9,10,11
- [50 POINTS] exam

Hello World

EXAMPLE

```
$ cat > hello.ml
print_string "hello world\n"
```

- run on interpreter

```
$ ocaml hello.ml
```

- byte-compile

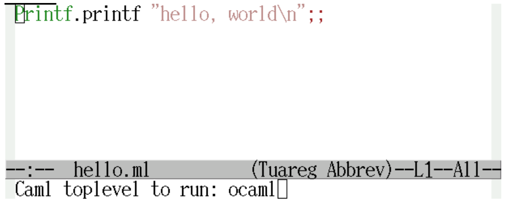
```
$ ocamlc hello.ml
$ ./a.out
```

- native-compile

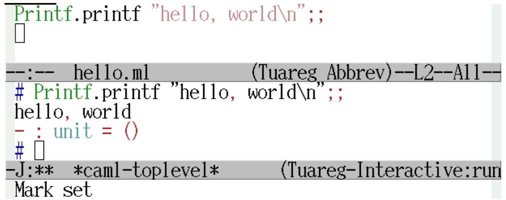
```
$ ocamlopt hello.ml
$ ./a.out
```

tuareg-mode (OCaml mode for Emacs)

- C-M-x or C-x C-e evaluates expression



- press enter



Functions

- function is declared by let

```
# let square x = x * x;;
val square : int -> int = <fun>

# square 10;;
- : int = 100

# let hello s = Printf.printf "Hello, %s\n" s;;
val hello : string -> unit = <fun>

# hello "world";;
Hello, world
- : unit = ()
```


Recursive Functions

- recursive function is declared by let rec

EXAMPLE

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

```
# let rec factorial n =
    if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>

# factorial 10;;
- : int = 3628800

# let rec factorial = function
    | 0 -> 1
    | n -> n * factorial (n - 1);;
```


Computational Model

- program is expression
- execution is rewriting

EXAMPLE

```
# let rec factorial = function
    | 0 -> 1
    | n -> n * factorial (n - 1);;

# factorial 3;;
- : int = 6
```

$$\begin{aligned} & \text{factorial } 3 \\ \rightarrow\ & 3 * \text{factorial } 2 \\ \rightarrow\ & 3 * 2 * \text{factorial } 1 \\ \rightarrow\ & 3 * 2 * 1 * \text{factorial } 0 \\ \rightarrow\ & 3 * 2 * 1 * 1 \\ \rightarrow\ & \cdots \\ \rightarrow\ & 6 \end{aligned}$$

Trace

EXAMPLE

```
# factorial 40;;
- : int = 0 !?
# #trace factorial;;
factorial is now traced.
# factorial 2;;
factorial <-- 2
factorial <-- 1
factorial <-- 0
factorial --> 1
factorial --> 1
factorial --> 2
- : int = 2
# factorial 40;;
```


Lists

- list is of the form $x_1 :: \cdots :: x_n :: []$, or $[x_1; \cdots; x_n]$
- $x_1, \ldots, x_n$ must have same type

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
# [1] :: [[2; 3]];;
- : int list = [[1]; [2; 3]]
# ["abc"; "def"];;
- : string list = ["abc"; "def"]
# [1; 2; "abc"];;
This expression has type string but is here used with type int
```


Length

$$\begin{aligned}
\text{length } ([]) &= 0 \\
\text{length } (3 :: []) &= 1 \\
\text{length } (2 :: 3 :: []) &= 2 \\
\text{length } (1 :: 2 :: 3 :: []) &= 3
\end{aligned}$$

```
let rec length = function
  | [] ->
  | x :: xs ->
```


Append

$$[\,] \mathbin{@} 3 :: 4 :: [\,] = 3 :: 4 :: [\,]$$
$$2 :: [\,] \mathbin{@} 3 :: 4 :: [\,] = 2 :: 3 :: 4 :: [\,]$$
$$1 :: 2 :: [\,] \mathbin{@} 3 :: 4 :: [\,] = 1 :: 2 :: 3 :: 4 :: [\,]$$

```
let rec (@) xs ys =
  match xs with
  | [] ->
  | x :: xs' ->
```