Week 1

Brendon Faleiro

Functional Programming

- Functional Programming is a paradigm that treats computation as the evaluation of mathematical functions and avoids changing and mutable data.
- Thus, it is side-effect free!

OCaml

- http://protz.github.io/ocaml-installer/

Ocaml Grammar

- E ::= C | X | E OP E | function P -> E | if E then E else E
  | match E with P -> E '|' ... '|' P -> E
  | [] | E :: E | let P = E in E | (E1,...,En) | E E
- C ::= 0 | 1 | 2 | ... | true | false | "hi" | ...
- X ::= variable names
- OP ::= + | - | * | < | ...
- P ::= C | _ | X | P::P | [P1;P2;...;Pn] | (P1,...,Pn)

Primitives

- ListsCons (::)
  - examples:-
    - 1::2::[]
    - [1]::[[2]]
    - [1;2]::[[]]
    - ‘a :: ‘a list
    - Adds in front of list

Demystifying `List` syntax (`::`, `;` and `[]`) :-

- `::` has an element on left and list on right- `::` is right associative-
  - 1::2::3::[] = [1; 2; 3] = 1::(2::(3::[]))
  - 1::2 is not a valid expression! (neither an `int list`, nor an `int`)
  - 1::2::[3; 4] is valid and has type `int list`.
  - [1::2]::[[3; 4]] is not a valid expression.
  - [1;2]::[[3; 4]] is valid and has type `int list list`.

- Append (@) examples:-
  - Appends two lists-
    - [1; 2] @ [3; 4] = [1; 2; 3; 4]
    - [1; 2]::[3; 4] is invalid

Pattern Matching

- h;t is invalid syntax
- [h;t] matches a 2 element list only
- [h::t] matches a list of lists, with at least 1 element in outer list
- h::t matches a list with at least one element
- h@t is invalid syntax
- _ matches anything
- _::t matches the top-most (i.e. the first)
- h::m::t - h and m are first 2 elements, t is the rest
- (x,y)::t first element matched to tuple directly

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

# square 3;;
- : int = 9

# let add x y = x + y;;
val add : int -> int -> int = <fun>

# add 1 2;;
- : int = 3

# let add (x,y) = x + y;;
val add : int * int -> int = <fun>

# add (3,4);;
- : int = 7

# add 5 6;;
Error: This function has type int * int -> int
       It is applied to too many arguments; maybe you
forgot a `;'.

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

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

Higher order functions

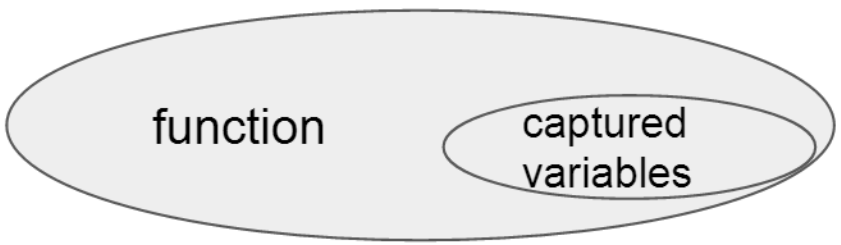
- A first-order function is one whose parameters and results are all “data”.
- In general, a higher-order function has one or more functions as parameters or results.
- Functional Languages support higher order functions
  - Ocaml supports passing functions as arguments to other functions and returning them as the values from the functions

```
# let twice f x = f (f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>

#twice (function i -> i+1) 5;;
- : int =7
```

Currying and dynamic scoping?

- let add = fun x y -> x + y
- let addTwo = add 2
  - (function x-> function y-> x+y) 2
  - returns closure → (function y->x+y, {x:2})
- what will happen if our language uses dynamic scoping?
  - let x = 5 in addTwo 3
    - The new definition of x should not affect the value of the original ‘x’.



- Closure
  - Closures are functions that refer to independent (free) variables. In other words, the function defined in the closure 'remembers' the environment in which it was created (*from wikipedia*)

Type Annotation

- Better errors: Instead of “inferred” types, you see “expected” types

- Easier programming: the arguments and return values are annotated, so when calling these functions, it’s easy to look up the types in signature!

- type point = float * float;;
- let get_x_annot ((x1,y1) :point) :float = x1;;
- let get_x1 (x1,y1) = x1;;

(* get_x1 in this example can return any type, but it was intended to return type float in the original design *)

- # get_x1 3.0;;

Error: This expression has type float but an expression was expected of type 'a * 'b

- # get_x_annot 3.0;;

Error: This expression has type float but an expression was expected of type point = float * float

- # get_x1 ("a", 3);;

- : string = "a"

- # get_x_annot ("a", 3);;

Error: This expression has type string but an expression was expected of type float