Intro to OCaml and Functional Programming (Recitation 1 and 2)
Topics (week 1)

- Imperative vs. Functional
- OCaml expressions
- Evaluating expressions
- Tuples, records
- Defining new data types
- Pattern matching
- Type inference
Imperative vs. Functional programs

- **Imperative**
  - $c_1; c_2; \ldots; c_n$
  - sequential

- **Functional**
  - $e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow v$
  - a series of reductions

- **functions are first-class objects**
  - pass around functions
  - partially apply functions
  - and a looooot more
Imperative vs. Functional styles

- No side effects
  - void function(T x) vs. T function (T x)
- Much easier to reason about
- Powerful paradigms like MapReduce
OCaml expression

● Values
  ○ bool: true, false
  ○ int: 0, 1, ...
  ○ float:
  ○ string, char

● Expressions
  ○ BNF (Backus-Naur form)
  ○ $e ::= c \mid \text{unop } e \mid e_1 \text{ binop } e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid (e)$
More expressions

- Declarations
  - let \( x = e \)

- Local bindings in expressions
  - let \( x = e \) in
  - some expression using \( x \)
  - \( e ::= \ldots \mid let \ id = e_1 \ in \ e_2 \)

- A typical OCaml program consists of a list of declarations (and module definitions)
Tuples

- ex. pair\langle A,B\rangle in C++
- type: 'a * 'b
- Extracting parts:
  - let (x,y) = tuple in ....
  - fst tuple
  - snd tuple
- Easy to combine multiple data
Records

- similar to primitive C structs, Java classes
- *Unordered*, unlike tuples
- \texttt{type student = \{id:int ; name:string\}}
- Each field has a label
- More convenient than tuples when extracting individual parts
Lists

- basic datatype in functional languages
- can add/retrieve the head (front) of the list
- no random access
- still can do a lot!
Some words about types

- Every (well-formed) expression has a type
- Different from imperative languages
  - what is the type of `int x; x = 2; cout << x ;`
Practice with types

- 5
- > int
- ("abc", 0.9)
- > string * float
- None
- > 'a option
- fun x y = x^y
- > string -> string -> string
- List.hd
- > 'a list -> 'a
- fun x = x x
- type answer = Yes | No
- "sum" types
- type num = Int of int | Real of float
- type ('a, 'b) either = Left of 'a | Right of 'b
Pattern matching

type t = A | B | C

let f (x : t) =
  match x in
  | A -> ...
  | A -> ...
  | B -> ...
  | C -> ...
The "underscore" case

Use | _ -> .... as a wildcard to match "all others."

- Useful when you don't distinguish the remaining cases

```ocaml
match lst with
  | h1::h2::t -> ....
  | _ -> failwith "Not enough args"
```

- It may be better practice spell out all cases
type 'a option = Some of 'a | None

forces you to deal with the "null" case

useful for returning "if yes, this is the answer, otherwise, nothing"
Type inference

- Types
  - int, bool, string
  - int -> int list
  - 'a list -> int, 'a -> 'b -> 'a * 'b

- You can annotate types explicitly...
  - let f (x : string) (y: int) : float = ...

- Or OCaml can infer for you!
  - let f x = x ** 2 ;;
  - val f : float -> float <fun>

- Why annotating is good, nonetheless
  - readability
  - to validate that your function/expression has correct type