CS131

Week 1
Saketh Kasibatla

Today

Recap Last Lecture (with a few more things about OCaml)

Exercises

More about grammars

This Class

Homework 1 is due **today at 23:55**

Submit by tomorrow @ 23:55, -1%; day after @ 23:55, -2% etc.

Homework 2 is due on 10/20 (2 weeks from today)

This homework will be graded with automated scripts

- not compiling → no credit
- function signatures must match, or no credit
- your code should behave exactly according to the spec
- if developing locally, test your homework out on SEASnet before submitting

OCaml

static typing → catch lots of errors at compile time

- type erasure → most type information is erased before the program is run

type inference → the system figures out type annotations for you

functional programming → more terse, easier to reason about

- referential transparency → call the same function with the same arguments, get the same result

Constants, Operators, Functions

```
let a = (5 * 100)
val a : int = 500

let b = (50 * 10)
val b : int = 500

let equal a b = a = b
val equal : ‘a -> ‘a -> bool = <fun>

let c = if equal a b then “equal” else “not equal”
val c : string = “equal”

let c = if equal then “equal” else 2
ERROR
```

More About Functions

Currying → calling a function with fewer arguments gives another function

```
let plus a b = a + b
val plus : int -> int -> int = <fun>

let plus5 = plus 5
val plus5 : int -> int = <fun>

let ans = plus5 10
val ans : int = 15

let plus = fun a -> fun b -> a + b
val plus : int -> int -> int = <fun>
```

More About Functions

```
let some_or_zero = function
  | Some x -> x
  | None -> 0
val some_or_zero : int option -> int
```

Lists, Tuples, Match

```
let a = []
val a : ‘a list = []

let b = [1;2;3]
val b : int list = [1; 2; 3]

let c = (42, 1)
val c : int * int = (42, 1)

let d = (42, “hi”, Some 42)
val d : int * string * int option = (42, “hi”, Some 42)
```

Exercises

```
let filter list predicate = ?
val filter : ‘a list -> (‘a -> bool) -> ‘a list = <fun>

let filter_test_1 = filter [1;2;3] (fun a -> a mod 2 = 0) = [2]
let filter_test_2 = filter [1;2;3] (fun a -> a mod 2 <> 0) = [1;3]

let fold list aggregator initialValue = ?
val fold : ‘a list -> (‘b -> ‘a -> ‘b) -> ‘b -> ‘b = <fun>

let fold_test_1 = fold [1;2;3] (+) 0 = 6
let fold_test_2 = fold [1;2;3] (*) 1 = 6
```

Solutions

```
let filter list predicate = match list with
  | [] -> []
  | x::xs -> if predicate x
    then x::(filter xs predicate)
    else (filter xs predicate)
val filter : ‘a list -> (‘a -> bool) -> ‘a list = <fun>

let filter_test_1 = filter [1;2;3] (fun a -> a mod 2 = 0) = [2]
let filter_test_2 = filter [1;2;3] (fun a -> a mod 2 <> 0) = [1;3]
```

Solutions

```
let fold list aggregator initialValue = match list with
  | [] -> initialValue
  | x::xs -> fold xs aggregator (aggregator initialValue x)
val fold : ‘a list -> (‘b -> ‘a -> ‘b) -> ‘b -> ‘b = <fun>

let fold_test_1 = fold [1;2;3] (+) 0 = 6
let fold_test_2 = fold [1;2;3] (*) 1 = 6
```

Custom Types

```
type (‘nonterminal, ‘terminal) symbol =
  | N of ‘nonterminal
  | T of ‘terminal
type (‘nonterminal, ‘terminal) symbol = N of ‘nonterminal | T of ‘terminal

type intList =
  | Empty
  | Cons of int * intList
type intList = Empty | Cons of int * intList
```

Exercise: Dictionaries

Association List → List of key value pairs

```
let d = [(hello, 5)]
val d : ? = [(hello, 5)]

type (‘key, ‘value) dict1 = ?
type (‘key, ‘value) dict1 = ?

let empty1 () = ?
val empty1 : unit -> (‘a, ‘b) dict1

let dicta = empty1 ()
val dicta : (‘a, ‘b) dict1 = ?
```

Exercise: Dictionaries

```
let put1 key value dict = ?
val put1 : ‘a -> ‘b -> (‘a, ‘b) dict -> (‘a, ‘b) dict = <fun>

let dictb = put1 42 “meow” dicta
val dictb : (int, string) dict = ?

let rec get1 key dict = ?
val get1 : ‘a -> (‘a, ‘b) dict -> ‘b option = <fun>

let ans = get1 42 dictb
val ans : string option = Some “meow”

let ans2 = get1 9 dictb
val ans2 : string option = None
```

Exercise: Dictionaries (Solutions)

```
let d = [(hello, 5)]
val d : (string, int) list = [(hello, 5)]

type (‘key, ‘value) dict1 = (‘key, ‘value) list
type (‘key, ‘value) dict1 = (‘key, ‘value) list

let empty1 () = []
val empty1 : unit -> ‘a list

let dicta = empty1 ()
val dicta : ‘a list = []
```

Exercise: Dictionaries (Solutions)

```
let put1 key value dict = (key, value)::dict
val put1 : ‘a -> ‘b -> (‘a, ‘b) list -> (‘a, ‘b) list = <fun>

let dictb = put1 42 “meow” dicta
val dictb : (int, string) list = [(42, “meow”)]

let rec get1 key dict = match dict with
  | [] -> None
  | (k, v)::xs -> if k = key
    then Some v
    else get1 key xs
val get1 : ‘a -> (‘a, ‘b) list -> ‘b option = <fun>

let ans = get1 42 dictb
val ans : string option = Some “meow”

let ans2 = get1 9 dictb
val ans2 : string option = None
```

Exercise: Dictionaries

Dictionaries as functions

```
type (‘a, ‘b) dict2 = (‘a -> ‘b option)
```

```
type (‘a, ‘b) dict2 = (‘a -> ‘b option)
```

```
let empty2 () requestKey = ?
val empty2 : unit -> (‘a, ‘b) dict2 = <fun>

let empty_test1 = empty2 () 42 = None
val empty_test1 : bool = true

let emptyDict = empty2 ()
val emptyDict : (‘a, ‘b) dict2 = <fun>
```

Exercise: Dictionaries

```
let put2 key value dict requestKey = ?
val put2 : ‘a -> ‘b -> (‘a, ‘b) dict2 -> (‘a, ‘b) dict2 = <fun>

let a = put2 42 “meow” empty
val a : (int, string) dict2

let get2 requestKey dict = ?
val get2 : ‘a -> (‘a, ‘b) dict2 -> ‘b option

let b = get2 42 a
val b : string option = Some “meow”

let c = get2 9 a
val c : string option = None
```

Exercise: Dictionaries (Solutions)

```
type (‘a, ‘b) dict2 = (‘a -> ‘b option)
type (‘a, ‘b) dict2 = (‘a -> ‘b option)

let empty2 () = fun requestKey -> None
val empty2 : unit -> ‘a -> ‘b option = <fun>

let empty_test1 = empty2 () 42 = None
val empty_test1 : bool = true

let emptyDict = empty2 ()
val emptyDict : (‘a, ‘b) dict2 = <fun>
```

Exercise: Dictionaries (Solutions)

```
let put2 key value dict = fun requestKey ->
  if requestKey = key
  then Some value
  else dict requestKey
val put2 : ‘a -> ‘b -> (‘a -> ‘b option) -> (‘a -> ‘b option) = <fun>

let a = put2 42 “meow” empty
val a : (int, string) dict2

let get2 requestKey dict = dict requestKey
val get2 : ‘a -> (‘a -> ‘b) -> ‘b

let b = get2 42 a
val b : string option = Some “meow”

let c = get2 9 a
val c : string option = None
```

Exercise: Binary Trees

```
type binaryTree = ?
type binaryTree = ?

let rec preorder tree = match tree with ?
val preorderTree : binaryTree -> int list = <fun>

let rec inorder tree = match tree with ?
val preorderTree : binaryTree -> int list = <fun>
```

Exercise: Binary Trees

Binary Search Tree - left children of each node have lower values, right children have higher values

```
let rec is_binary_search_tree tree = match tree with ?
```

```
let rec binary_st_to_list tree = match tree with ?
```

Exercise: Binary Trees (Solutions)

```
type binaryTree =
  | Leaf of int
  | Node of int * binaryTree * binaryTree
type binaryTree = Leaf of int | Node of int * binaryTree * binaryTree

let rec preorder tree = match tree with
  | Leaf x -> [x]
  | Node (x, left, right) -> [x] @ (preorder left) @ (preorder right)
val preorderTree : binaryTree -> int list = <fun>

let rec inorder tree = match tree with
  | Leaf x -> [x]
  | Node (x, left, right) -> (preorder left) @ [x] @ (preorder right)
val preorderTree : binaryTree -> int list = <fun>
```

Exercise: Binary Trees (Solution)

```
let rec is_ordered list = match list with
  | [] -> true
  | x::[] -> true
  | x::y::rest -> (x < y) && (is_ordered (y::rest))
let is_binary_search_tree tree = is_ordered (inorder tree)

let rec binary_st_to_list tree = inorder tree
```