# Contents

# 1 Lecture 1 <2017-09-12 Tue>

This course is an introduction to the foundations and paradigms of programming languages.

- 5 assignments, 5% each
- 10% midterm

- 65% final
- You have two late days for the semester (cumulative)

## 1.1 Four main goals of COMP 302

1. Provide thorough introduction to fundamental concepts in programming languages
    - Higher-order functions
    - State-full vs state-free computation (most languages like Java we've seen are state-full)
    - Modeling objects and closures
    - Exceptions to defer control
    - Continuations to defer control
    - Polymorphism
    - Partial evaluation
    - Lazy programming
    - Modules
    - Etc.
    - Want to explore these concepts so you can recognize them in another language you study at some point
2. Show different ways to reason about programs
    - Type checking
        - One of the best inventions
        - Checks what it expects and will actually tell you where it expects something
        - Program is more likely to be correct now
    - Induction
        - Proving a program/transformation correct
    - Operational semantics
        - How a program is executed
    - QuickCheck
3. Introduce fundamental principles in programming language design

- Grammars
- Parsing
- Operational semantics and interpreters
- Type checking
- Polymorphism
- Subtyping

4. Expose students to a different way of thinking about problems
    - It's like going to the gym; it's good for you!

### 1.1.1 How we achieve these goals

- Functional programming in OCaml
    - Equal playing field
        * No one in the class really knows it, not affected by performance in previous classes like 250
    - Allows us to explain and model object-oriented and imperative programming
        * Isolates lots of the concepts individually
    - Isolates concepts such as state-full vs state-free, modules and functions, etc.
    - Statically typed language enforces disciplined programming
        * Also demonstrates that types are an important maintenance tool
    - Easy to reason about runtime behavior and cost

### 1.1.2 Guiding Principles

- No point in learning a programming language unless it changes how you view programming
- Simple and elegant solutions are more effective, but harder to find than the complicated ones, take more time.
    - You spend very little time testing OCaml code and more time compiling it

### 1.1.3 Why do I need to know this

- Science and craft of programming
- Skills you learn will help you become a better programmer
  - More productive
  - Code easier to maintain and read
  - Etc.
- Will be needed in some upper level courses
  - Like compilers, etc.
- It is cool and fun!
- You might even get a job!

## 1.2 Assignments

- Can do assignments in groups of 2

## 1.3 Misc

- Lectures won't be recorded.
- Slides may or may not be posted, but there are lecture notes on MyCourses (most essential reading)

# 2 Lecture 2 *<2017-09-14 Thu>*

## 2.1 What is OCaml

- Statically typed functional programming language

### 2.1.1 Statically typed

- Types approximate runtime behavior
- Analyze programs before executing them
- Find a fix bugs before testing
- Tries to rule out bad scenarios
- Very efficient, very good error messages, very good maintenance tool

### 2.1.2 Functional

- Primary expressions are functions!
- Functions are first-class!
  - Not only can we return base types like ints, we can return functions and pass them as arguments too
  - One of the key features of functional languages
- Pure vs Not Pure languages
  - Haskell is Pure
    - Doesn't give you ways to allocate memory or directly modify memory
  - OCaml is impure
    - Has arrays and consequences and stuff
- Call-By-Value vs Lazy
  - OCaml is call by value

### 2.1.3 Concepts for Today

- Writing and executing basic expressions
- Learn how to read error messages
- Names

### 2.1.4 OCaml demo in class

- Always have to finish a line with 2 semi colons ;;
- Can use interpreter by launching OCaml in shell
- Functional good for parallel computing
- Good to reason about these programs
- int:
  - 1 ;;
  - 1+3;;

- Strings:
  - "Hello";;
- Floats:
  - 3.14;;
- Booleans:
  - true;;
- if
  - if 0=0 then 1.4 else 2.1 ;;

1. Operators
   - +, -, /, *
     - Take as input 2 int, return int
   - 3.14 + 1 ;; → error
   - To specify for floating point operators, follow by a dot. Only works with floating points, no ints
     - 3.14 +. 2.4 ;;
2. Types
   - Approximate the runtime behaviour
   - Types classify expressions according to the value they will compute
   - Won't execute right away, will think of types you are returning to see if it's valid
   - if 0=0 then 1.4 else 3 ;;
     - Error, after reading 1.4 expects 3 to be float
   - if bool then T else T
     - Both Ts have to be the same type
   - Type checker will allow 1/0;; to run, but will have a runtime exception
     - int/int is not enough info to know that your dividing by 0
3. Vars

- let pi = 3.14 ;;
- let (pi : float) = 3.14 ;;
- let m = 3 in
  - let n=m * m in
  - let k=m*m in
  - k*n ;;

4. Binding
   - let m = 3 ;; puts it on the stack
   - let m = 3 in ...
     - m is a local variable now (temporary binding), once you hit ;;, won't have m anymore
     - Garbage collector
   - let x (name of a variable) = exp in exp (x is bound to this expression)
   - variables are bind to values, not assigned values
     - they look in the past!

5. Functions
   - let area = function r -> pi *. r * .r;;
     - Syntax error
   - let area = function r -> pi *. r *. r ;;
   - let area r = pi *. r *. r ;;
   - let a4 = area (2.0);;
   - If you redefine pi, like let pi = 6.0 ;;
   - area(2.0) will still give you the same thing
   - The function looks up in the past
   - Stack:

| | |
|---|---|
| pi | 6.0 |
| area | function r -> p *. r *. r |
| k | 5 |
| k | 4 |
| pi | 3.14 |

   - Can redefine the function though

# 3 Lecture 3 *<2017-09-15 Fri>*

## 3.1 Functions

- Functions are values
- Function names establish a binding of the function name to its body
  - let area (r:float)=pi*. r *. r ;;

### 3.1.1 Recursive functions

Recursive functions are declared using the keyword let rec

- let rec fact n =
  - if m = 0 them 1
    - else n*face(n-1)
- fact 2 needs to be stored on the stack
- fact 2 -> 2* fact 1
- fact 1 -> 1* fact 0 stored on stack
- fact 0 = 1
- Need to remember computation when you come back out of recursion, so need to store on the stack
  - What's the solution to this? How is functional programming efficient?

1. Tail-recursive functions A function is said to be "tail-recursive", if there is nothing to do except return the final value. Since the execution of the function is done, saving its stack frame (i.e. where we remember the work we still in general need to do), is redundant
   - Write efficient code
   - All recursive functions can be translated into tail-recursive form
2. Ex. Rewrite Factorial
   - let rec fact_tr n =
     - let rec f(n,m) -

        * if n=0 then
            · m
        * else f(n-1,n*m)
    - in
        * f(n,1)
- Second parameter to accumulate the results in the base case we simply return its result
- Avoids having to return a value from the recursive call and subsequently doing further computation
- Avoids building up a runtime stack to memorize what needs to be done once the recursive call returns a value
- f(2,1) -> fact(1, 2*1) -> fact(0,2)-> 2
- Whoever uses the function does not need to know how the function works, so you can use this more efficient way in the background
- What is the type of fact_tr? fact_tr: int(input) → int(output)
- Type of f? f: int * int (tuple input) → int
    - n-tuples don't need to be of the same type, can have 3 different types, like int*bool*string

### 3.1.2 Passing arguments

- ' means any type, i.e. 'a
- All args at same time
    - 'a*'b -> 'c
- One argument at a time
    - 'a -> 'b -> 'c
    - May not have a and b at the same time. Once it has both it will get c.
- We can translate any function from one to the other type, called currying (going from one at a time to all at once) and uncurrying (opposite).
    - Will see in 2 weeks

## 3.2 Data Types and Pattern Matching

### 3.2.1 Playing cards

- How can we model a collection of cards?
- Declare a new type together with its elements
- type suit = Clubs | Spades | Hearts | Diamonds
  - Called a user-defined (non-recursive) data type
  - Order of declaration does not matter
    - Like a set
  - We call clubs, spades, hearts, diamonds constructors (or constants), also called elements of this type
    - Constructors must begin with a capital letter in OCaml
- Use pattern matching to analyze elements of a given type.
- match <expression> with

  <pattern> -> <expression>
  <pattern> -> <expression>

  ...

  <pattern> -> <expression>

A pattern is either a variable or a ...

- Statements checked in order

1. Comparing suits Write a function dom of type suit*suit -> bool
   - dom(s1,s2) = true iff suit s1 beats or is equal to suit s2 relative to the ordering Spades > Hearts > Diamonds > Clubs
   - (Spades, _) means Spades and anything
   - (s1, s2) -> s1=s2 will return the result of s1=s2
   - Compiler gives you warning if it's not exhaustive and tells you some that aren't matched

# 4 Lecture 4 *<2017-09-19 Tue>*

## 4.1 Data Types and Pattern Matching Continued

- Type is unordered
- type suit = Clubs | Spades | Hearts | Diamonds
  - Order doesn't matter here, but they must start with capitals
- type rank = Two | Three | ...
- type card = rank * suit

What is a hand? A hand is either empty or if c is a card and h is a hand then Hand(c,h). Nothing else is a hand. Hand is a constructor. hand is a type. (capitalization mattersA)

- Recursive user defined data type
- Inductive or recursive definition of a hand
  - Add a card to something that is a hand, still a hand

### 4.1.1 Recursive data type

- type hand = Empty | Hand of card * hand

1. Typing into interpreter
   - Empty;;
     - hand = Empty
   - let h1 = Hand ((Ace, Spades),Empty);;
     - Want only 1 card, so include empty
     - Recursive data type, so it needs another hand in it
   - let h2 = Hand ((Queen, Hearts), Hand((Ace,Spades), Empty);;
     - Recursive
   - let h3 = Hand ((Joker, Hearts), h2) ;;
     - Error, Joker not defined
   - type 'a list = Nil | Cons of 'a * 'a list
   - Hand ((Queen, Hearts), (King, Spades), (Three, Diamonds));;
     - Hand has type? card * card * card
     - Get an error, because constructor Hand expects 2 arguments (card+hand)

### 4.1.2 Extract Example

- Given a hand, extract all cards of a certain suit
- extract: suit -> hand -> hand

```
let rec extract (s:suit) (h:hand) = match h with
  | Empty -> Empty (* We are constructing results, not destructing given hand *)
  (* Want to extract suit from first card *)
  | Hand ( (r0.s0) ,h) ->
     (*Make a hand with first card and remaining results of recursive ext*)
     if s0 = s then Hand( (r0, s0), extract s h0)
     else extract s h0
```

Hand is "destroyed" through this method, but old hand stays the same, it is not modified.

- Running extract Spades hand5;; will give a new hand with only spades
- Good exercise, write a function that counts how many cards in the hand
- Can we make this thing tail recursive?

```
let rec extract' (s:suit) (h:hand) acc = match h with
  | Empty -> acc (* Accumulator *)
  (* Want to extract suit from first card *)
  | Hand ( (r0.s0) ,h) ->
     (*Make a hand with first card and remaining results of recursive ext*)
     if s0 = s then extract' s h0 (Hand( (r0, s0), acc))
     else extract' s h0 acc
```

- extract' Spades hand5 Empty ;;
- Gives same cards but in the reverse order of extract
- extract Spades hand5 = extract' Spades hand5 Empty ;;
  - False
- Write a function find which when given a rank and a hand, finds the first card in hand of the specified rank and returns its corresponding suit.

What if no card exists?

- Optional Data Type (predefined)
- type 'a option = None | Some of 'a

## 5 Lecture 5 *<2017-09-21 Thu>*

```
(* type mylist = Nil | Cons of ? * list;; *)
(* Polymorphic lists:  *)
(* type 'a mylist = Nil | Cons of 'a * 'a my list *)
[] ;;
1 :: [] ;;
1 :: 2 :: 3 :: [] ;;
[1;2;3;4];;

(* These are only homogenous lists though, what if we want floats and ints? *)
(* type if_list = Nil | ICons of int * if_list | FCons of float * if_list *)
(* But here we can't use List libraries *)

(* So make an element that can be either *)
type elem = I of int | F of float;;

let rec append l1 l2 = match l1 with
  | [] -> l2
  | x::xs -> x :: append xs l2;;
(* Program execution *)
(* append 1::(2::[]) -> 1 :: append (2::[])             *)
let head l = match l with
  | [] -> None
  | x :: xs -> Some x;;

(* Write a function rev given a list l of type 'a list returns it's reverse  *)

(* Silly way of doing this             *)
let rec rev (l : 'a list) = match l with
  | [] -> []
  | hd :: tail -> rev (tail) @ [hd];;
(* Could we have written rev(tail) :: hd? No. Why? *)
(* a' : 'a list, left side has to be one element, right side  to be a list *)
```

```
(* 'a list @ 'alist *)

(* What is the type of rev? Is it 'a list? -No *)
(* It is 'a list -> 'a list *)

(* Is this a good program? Long running time, use tail recursion  *)


let rev_2 (l : 'a list) =
  let rec rev_tr l acc = match l with
    | [] -> acc
    | h::t -> rev_tr t (h::acc)
  in
  rev_tr l [];;

(* Exercises:
 * Write a function merge: 'a list -> 'a list -> 'a list
which given ordered lists l1 and l2, both of type 'a list,
it returns the sorted combination of both lists
 * Write a function split: 'a list -> 'a list * 'a list
which given a list l it splits into two sublists,
(every odd element, every even element)*)
```

### 5.1 Lists

What are lists?

- Nil([]) is a list
- Given an element x and a list l x::l is a list
- Nothing else is a list
- [] is an $'\alpha$ list
  - Given an element x of type $'\alpha$ and l of type $'\alpha$ list
    x l is an $'\alpha$ list (i.e. a list containing elements of type $'\alpha$)
- ; are syntactical sugar to separate elements of a list

### 5.2 Execution

Understand how a program is executed

- **Operational Semantics**

## 6 Lecture 6 *<2017-09-22 Fri>*

### 6.1 Proofs

#### 6.1.1 Demo: lookup & insert

```
(* Warm up *)
(* Write a function lookup: 'a -> ('a * 'b) list -> 'b option.
Given a key k of type 'a and a list l of key-value pairs,
return the corresponding value v in l (if it exists). *)
(* lookup : 'a -> ('a * 'b) list -> 'b option  *)
let rec lookup k l = match l with
  | [] -> None
  | (k',v')::t -> if k=k' then Some v' (* If it is the right key, return val*)
  else lookup k t;;

(* Write a function insert which
given a key k and a value v and an ordered list l of type ('a * 'b) list
it inserts the key-value pair (k,v) into the list l
preserving the order (ascending keys). *)
(* insert : ('a * 'b) -> ('a * 'b) list -> ('a * 'b) list
 insert (k,v) l = l'

 Precondition: l is ordered.

Postcondition: l' is also ordered and we inserted (k,v) at the right position in l*)

(* let rec insert (k,v) l = match l with
 *   | [] -> [(k,v)]
 *         (\* k = k' or k < k' or k' < k *\)
 *   | ((k',v') as h) :: t ->
 *      if k = k' then (k,v) :: l
 *      else
 *        if k' < k then (k,v) :: l
 *        else h :: insert (k,v) t;;
```

```
 *
 * let l = [(1,"anne") ; (7,"di")];;
 * l;;
 * let l0 = insert (3,"bob") l;;
 * insert (3,"tom") l0 ;;
 * (\* But now we'll have 2 entries with the same key *\) *)
(* Undesirable, better to replace the value if its a dictionary*)

let rec insert (k,v) l = match l with
  | [] -> [(k,v)]
(* k = k' or k < k' or k' < k *)
  | ((k',v') as h) :: t ->
     if k = k' then (k,v) :: t (* Replace *)
     else
       if k' < k then (k,v) :: l
       else h :: insert (k,v) t;;

(* Personal tail recursive attempt *)
let insert_t (k,v) l =
  let rec insert_acc (k,v) l acc = match l with
    | [] -> acc @ [(k,v)]
    | ((k',v') as h) :: t ->
       if k = k' then (k,v) :: t
       else
 if k' < k then (k,v) :: l
 else insert_acc (k,v) t (acc @ [h])
```

- What is the relationship between lookup and insert?

### 6.1.2 How to prove it?

1. Step 1 We need to understand how programs are executed (operational semantics)
   - $e \Downarrow v$ expression $e$ evaluates in multiple steps to the value $v$. (**Big-Step**)
   - $e \Rightarrow e'$ expression $ee$ evaluates in one steps to expression $e'$. (**Small-Step (single)**)
   - $e \Longrightarrow {}^{*}e'$ expression $e$ evaluates in multiple steps to expression $e'$ (**Small-Step (multiple)**)

For all l, v, k, lookup k (insert k v l) $\implies$ $*$ Some v Induction on what?

2. Step 2 $P(l) = lookupk$ (insert$(k, v)l$) $\Downarrow$ Some $v$
   - How to reason inductively about lists?
     - Analyze their structure!
     - The recipe ...
     - To prove a property $P(l)$ holds about a list $l$
       - Base Case: $l = []$
         - Show $P([])$ holds
       - Step Case: $l = x :: xs$
         - IH $P(xs)$ (Assume the property $P$ holds for lists smaller than $l$)
       - Show $P(x :: xs)$ holds (Show the property $P$ holds for the original list $l$)
3. Theorem For all $l, v, k$, lookup $k$ (insert $(k, v)l$) $\implies$ $*$ Some $v$
4. Proof Proof by structural inductional on the list $l$
   - Case: $l = []$
     - lookup k (insert(k,v)[])
     - $\overset{\text{By insert program}}{\implies}$ lookup k [(k,v)] (same as (k,v)::[]) $\overset{\text{By lookup}}{\implies}$ Some $v$
       - Would not hold if we didn't put the k=k case
   - Case: $l = h :: t$ where $h = (k', v')$
     - IH: For $k, v$ lookup $k$ (insert (k,v) t) $\Downarrow$ Some $v$
     - To show: lookup k (insert (k,v)) $\underbrace{(k', v') :: t}_{l}$ $\Downarrow$ Some $v$
     - Subcase: $k = k'$
       - lookup k (insert (k,v) ((k', v')::t))
       - $\overset{\text{By insert}}{\implies}$ lookup k ((k,v)::t)
       - $\overset{\text{By lookup}}{\implies}$ Some v (good)
     - Subcase: $k < k'$
       - lookup k (insert(k,v) ((k',v'):: t))
       - $\overset{\text{By insert}}{\implies}$ lookup k ((k,v)::l)

        * $\overset{\text{By lookup}}{\Longrightarrow}$ Some v (good)
    - Subcase: $k > k'$
        * lookup k (insert(k,v) ((k',v')::t))
        * $\overset{\text{By insert}}{\Longrightarrow}$ lookup k ((k', v')::insert (k,v) t)
        * $\overset{\text{By lookup}}{\Longrightarrow}$ lookup k (insert (k,v) t)
        * $\overset{\text{By IH}}{\Longrightarrow}$ Some $v$

5. Lesson to take away

    - State what you are doing induction on
        - Proof by structural induction in the list l
    - Consider the different cases!
    - For lists, there are two cases- either l =[] or l = h::t
    - State your induction hypothesis
        - IH: For all v,k, lookup insert (k,v) t $\Downarrow$ Some $v$
    - Justify your evaluation / reasoning steps by
        - Referring to evaluation of a given program
        - The induction hypothesis
        - Lemmas/ Properties (such as associativity, commutativity)

# 7 Lecture 7 *<2017-09-26 Tue>*

## 7.1 Structural Induction

- How do I prove that all slices of cake are tasty using structural induction?
    - Define a cake slice recursively
    - Prove that a single piece of cake is tasty
    - Use recursive definition of the set to prove that all slices are tasty
    - Conclude all are tasty

### 7.1.1 Example with rev

```
(* naive *)
(* rev: 'a list -> 'a list *)
let rec rev l = match l with
```

```
| [] -> []
| x::l -> (rev l) @ [x];;

(* tail recursive *)
(* rev': 'a list -> 'a list *)
let rev' l =
(* rev_tr: 'a list -> 'a list -> 'a list *)
let rec rev_tr l acc = match l with
| [] -> acc
| h::t -> rev_tr t (h::acc)
in
rev_tr 1 [];;

(* Define length *)
let rec length l = match l with
| [] -> 0
| h::t -> 1+length t
```

1. Theorem: For all lists l, rev l = rev' l.

   What is the relationship between l, acc and rev_tr l acc?

   - Invariant of rev
     - length l = length (rev l)
   - Invariant rev_tr
     - length l + length acc = length(rev_tr l acc)
   - How are these related?

---

   - rev l ⇓
   - rev_tr l acc ⇓ v
   - Not quite because:
     - rev [] ⇓ []
     - rev_tr [] acc ⇓ acc
     - Not returning the same thing given empty list
   - Slightly modified so it's right:
     - rev l @ acc ⇓ v

- rev_tr l acc ⇓ v

---

For all l, acc, (rev l) @ acc ⇓ v and rev_tr l acc⇓ v By induction on the list l.

- Case l=[]
  - rev [] @ acc
  - $\overset{\text{prog rev}}{\rightarrow}$ [] @ acc → acc
  - rev_tr [] acc
  - $\overset{\text{by prog rev\_tr}}{\rightarrow}$ acc
- Case $l = h :: t$
  - **IH: For all acc rev t @ acc ⇓ v and rev_tr t acc ⇓ v**
  - rev (h::t) @ acc $\overset{\text{by rev}}{\rightarrow}$ (rev t @ [h])@acc
    - $\overset{\text{By associativity of @}}{\rightarrow}$ rev t @ ([h] @ acc)
    - $\overset{@}{\rightarrow}$ rev t @ (h::acc)
  - rev_tr (h::t) acc → rev_tr t (h::acc)
  - By the IH rev t @ (h::acc) ⇓ v and rev_tr t (h::acc) ⇓ v

## 7.2 Trees

```
type 'a tree = Empty | Node of 'a = 'a tree * 'a tree;;
let t0 = Node (5, Node (3, Empty, Empty), Empty);;
(* 5 is at the head, has 3 as left child, all other children are empty *)

let rec size t = match t with
  | Empty -> 0
  | Node (v, l, r) -> 1 + size l + size r

     size t0;;
(* Since Ocaml is a stack,
  if you modify the tree type after declaring t0,
  then t0 will be the old type,
  so when you try to use size you'll get an error
  as it's not the same type *)

type 'a forest = Forest of ('a many_trees) list
```

```
and 'a many_trees = Empty | MoreTrees of 'a many_trees

(* Mutually recursive *)

let rec size_forest f = match f with
  | Forest trees -> match trees with
    | [] -> 0
    | h::t -> size_many_trees h + size_forest (Forest t)

and size_many_trees t = match t with
  | NoTree -> 0
  | MoreTrees f -> 1 + size_forest f
```

# 8 Lecture 8 *<2017-09-28 Thu>*

## 8.1 Binary Tree (Inductive definition)

- The empty binary tree empty is a binary tree
- If l and r are binary trees and v is a value of type 'a then Node(v, l, r) is a binary tree
- Nothing else is a binary tree

How to define a recursive data type for trees in OCaml?

```
type 'a tree =
Empty
| Node of 'a * 'a tree * 'a tree |
```

## 8.2 Insert

Want to make a function insert

- Given as input (x,dx), where x is key and dx is data and a binary search tree t
  - Return a binary search tree with (x,dx) inserted
  - What is insert's type?
    - (a' * b') → ('a × 'b)tree → ('a × 'b) tree

- Good exercise: write a function to check if a tree is a binary search tree or not
  - Good exam question

```
(* Data Types: Trees *)

type 'a tree = Empty | Node of 'a * 'a tree * 'a tree

let rec size t = match t with
  | Empty -> 0
  | Node (v, l, r) -> 1 + size l + size r
let rec insert ((x,dx) as e) t = match t with
  (* Tree is empty, root is now e *)
  | Empty -> Node (e, Empty, Empty)
  | Node ( (y,dy), l, r) ->
     (* Replace val that has same key *)
     (* No destructive updates, need to keep elements and remake tree *)
     if x = y then Node (e, l, r)
(* Go down left tree *)
     else (if x < y then Node ( (y,dy), insert e l, r)
      (* Go down right tree *)
   else Node ((y,dy), l, insert e r)
  )

    (* Can we still use these less than signs for any type?
     The node constructor uses any type
     Since we used comparison, we can*)

;;
3 < 4 ;;
Empty < Node (3, Empty, Empty);;
Node (3, Empty, Empty) < Node (4, Empty, Empty) ;;

[3 ; 4] < [2 ; 5];;
(* Why is this false? *)

[3 ; 5] < [4 ; 7];;
[3 ; 5] < [7];;
(* Doesn't look at length of list, looks at first number of list *)
```

```
(* Dangerous to have comparison on all these types. Can only compare built in data type
 OCaml will come up with something, but it might not be the correct thing
 For example, with the suits example we made a function to quantify
 what's bigger*)

(* lookup: 'a -> ('a x 'b)tree -> b' option *)
(* Option in case key isn't there *)
let rec lookup x t = match t with
  | Empty -> None
  | Node ( (y, dy), l, r) ->
     if x = y then Some dy
     else ( if x < y then
      lookup x l
    else lookup x r
  )
;;


(* collect: 'a tree -> 'a list
 What order do we want to return it in? In order traversal*)
let rec collect t = match t with
  | Empty -> []
  | Node (x, l, r) ->
     let l1 = collect l in
     let l2 = collect r in
     l1 @ l2
(* Incomplete, where to put x? *)
;;

collect (Node (5, Node (3, Empty, Empty), Empty));;
```

### 8.3 Proving

- How to reason inductively about trees? Analyze their structures!

### 8.4 Theorem

For all trees t, keys x, and data dx, lookup x(insert (x, dx) t) $\Rightarrow *$ Some dx

### 8.4.1 Proof by structural induction on the tree t

(You get points on an exam for mentioning what kind of induction, structural induction on tree, points for base case/case, points for stating induction hypothesis, perhaps multiple. Then show by a sequence of steps of how to get from what to show to the end)

- Case t = Empty
  - lookup x (insert (x, dx) Empty) $\overset{\text{By insert}}{\Rightarrow}$ lookup x (Node ((x,dx), Empty, Empty)) $\overset{\text{by lookup}}{\Rightarrow}$ Some dx
- Case t = Node ( (y,dy), l, r)
  - Both trees l and r are smaller than t
  - IH1: For all x, dx, lookup x (insert (x,dx) l) $\Rightarrow *$ Some dx
  - IH2: For all x, dx, lookup x (insert(x,dx) r) $\Rightarrow *$ Some dx
- Need to show lookup x (insert (x, dx) Node ((y, dy), l, r))
- Show 3 cases (x < y, x = y, y < x)
  - x < y $\Rightarrow$ lookup x (Node((y,dy), insert (x,dx) l, r)) $\overset{\text{By lookup}}{\Rightarrow}$ lookup x (insert (x,dx) l) $\overset{\text{by IH 1}}{\Rightarrow}$ Some dx
  - x=y lookup x(insert (x,dx) Node ((y, dy), l, r)) $\overset{\text{by ins}}{\Rightarrow}$ lookup x (Node ((x,dx), l, r)) $\overset{\text{by lookup}}{\Rightarrow}$ Some dx

---

Exercise: write a type for cake (2 slice of cake together become 1 slice), with weight

# 9 Lecture 9 *<2017-09-29 Fri>*

## 9.1 Higher-order functions

- Allows us to abstract over common functionality
- Programs can be very short and compact
- Very reusable, well-structured, modular
- Each significant piece implemented in one place

- Functions are first-class values!
  - Pass functions as arguments (today)
  - Return them as results (next week)

### 9.1.1 Abstracting over common functionality

Want to write a recursive function that sums up over an integer range: $\sum_{k=a}^{k=b} k$

```
let rec sum (a,b) =
if a > b then 0 else a + sum(a+1,b)
```

Now what if we want to make a sum of squares? $\sum_{k=a}^{k=b} k^2$

```
let rec sum (a,b) =
if a > b then 0 else square(a) + sum(a+1,b)
```

$\sum_{k=a}^{k=b} 2^k$

```
let rec sum (a,b) =
if a > b then 0 else exp(2,a) + sum(a+1,b)
```

- So you can reimplement the function every time, but it would be more useful to make a sum function that will sum up what you tell it to (what to do to each element)
- Non-Generic Sum (old)
  - int * int -> int
- Generic Sum using a function as an argument
  - (int -> int) -> int * int -> int

## 9.2 Demo

```
(* Arbitrary functions *)
(* cube, rcube, square, exp, sumInts, sumSquare, sumCubes, sumExp *)
let square x = x * x;;
let cube x = x * x * x;;
let rec exp (a, b) = match a with
  |
```

```
  (* Non-generalized sums  *)
  let rec sumInts (a,b) = if (a > b) then 0 else a + sumInts(a+1,b);;
let rec sumSquare(a,b) = if (a > b) then 0 else square(a) + sumSquare(a+1,b);;
let rec sumCubes(a,b) = if (a > b) then 0 else cube(a) + sumCubes(a+1, b);;


(* We will abstract over the function f (i.e. cube, square, exp etc)
 to get a general sum function*)

(* sum: (int -> int) -> int * int -> int  *)
let rec sum f(a,b) =
  if a > b then 0
  else f(a) + sum f(a+1, b);;

(* Call function on a *)

(* Identity function, returns Argo *)
let id x = x;;
let exp2 x = exp (2, x);;

(* let sumInts' (a,b) = sum id (a,b);; *)
(* anonymous functions *)
let sumInts' (a,b) = sum (fun x -> x) (a,b);;

(* let sumSquare' (a,b) = sum square(a,b);; *)
let sumSquare' (a,b) = sum (fun x -> x * x) (a,b)

let sumCubes' (a,b) = sum cube(a,b);;

(* let sumExp' (a,b) = sum exp2(a,b);; *)
let sumExp' (a,b) = sum (fun x-> exp (2,x)) (a,b);;

(* Inconvenient, we have to define a function beforehand *)
(* How can we define a function on the fly without naming it?
 -> Use anonymous functions*)

(* Different ways to make anonymous functions *)
fun x y -> x + y;;
function x -> x;;
```

```
fun x -> x;;
(* Can use function for pattern matching
 Don't need to write match
 Function can only take in one argument and implies pattern matching
 fun can take many *)
(function 0 -> 0 | n -> n+1);;
(* Equivalent to fun and match *)
(fun x -> match x with 0 -> 0 | n -> n+1);;


(* comb: is how we combine - either * or +
 f : is what we do to the a
 inc : is how we increment a to get to b
 base : is what we return when a > b *)
(* Make this tail recursive this time  *)
let rec series comb f (a,b) inc base =
  if a > b then base
  else series comb f (inc(a),b) inc (comb base (f a));;
(* Base acts as an accumulator *)
```

- How about only summing up odd numbers?

```
let rec sumOdd (a,b) =
if (a mod 2) = 1 then
sum (fun x -> x) (a, b)
else
sum (fun x -> x)(a+1, b)
```

- Adding increment function

```
let rec sum f (a, b) inc =
if (a > b) then 0 else (f a) + sum f (inc(a), b) inc


let rec sumOdd (a,b) =
if (a mod 2) = 1 then
sum (fun x -> x) (a, b) (fun x -> x+1)
else
sum (fun x -> x)(a+1, b) (fun x-> x+1)
```

- How about only multiplying?

```
let rec product f (a, b) inc =
if (a > b) then 1 else (f a) * product f (inc(a), b) inc
```

- Can make this tail recursive with accumulators for base (1 for prod, 0 for sum)

---

- Types:
  - (int -> int -> int) : comb
  - series: -> (int -> int) : f
  - int * int : a,b lower and upper bound
  - int -> int : inc
  - int : base

  Types can get crazy, too much abstraction may lead to less readability

### 9.3 Bonus

Approximating the integral

- $l = a + dx/2$
  - Use rectangles to approximate
  - Left side of l is above the rectangle, right side is below, approximation should almost cancel them
  - $\int_a^b f(x)dx \approx f(l) * dx + f(l+dx) * dx + f(l+dx+dx) * dx + \ldots = dx * (f(l) + f(l+dx) + f(l+2*dx) + f(l+3*dx)\ldots)$

Want: sum: $\underbrace{(float -> float)}_{f} -> (\underbrace{float}_{l} * \underbrace{float}_{u}) -> \underbrace{(float -> float)}_{inc} ->$ $float$

```
let integral f (a,b) dx =
dx * sum f (a+.(dx/2.),b) (fun x-> x+. dx)
(* Follows format of sum function above
Can easily write a short program like above*)
```

## 10 Lecture 10 *<2017-10-03 Tue>*

```
(* Common built in higher-order functions we'll be writing *)

(* map: ('a -> 'b) -> 'a list -> list, bracket does whatever function f does *)
(* map is the most important higher order function *)
let rec map f l = match l with
  | [] -> []
  (* Apply function to head and then prepend to what you get from recursive call *)
  | h :: t -> (f h) :: map f t;;

(* Increment all by one  *)
map (fun x -> x + 1) [1 ; 2 ; 3 ; 4];;

(* Convert to strings *)
map (fun x -> string_of_int x) [1 ; 2 ; 3 ; 4];;

(* filter: ('a -> bool) -> 'a list -> 'a list
 Want to filter out elements of a list
 function in bracket takes an argument and returns boolean whether it's good or not*)
(* Ex. filter (fun x-> x mod 2 = 0) [1 ; 2 ; 3 ; 4] should give [2 ; 4] *)
let rec filter p l = match l with
  | [] -> []
  | h :: t ->
     (* If it satisfies p, prepend to recursive call *)
     if p h then h :: filter p t
     else filter p t;;

(* Being on the safe side, we can write:
 * let pos l = filter (fun x -> x > 0) l *)

(* But we can also write , because it partially evaluates function
 * What we get back is a function from 'a list -> 'a list
 * and we can return a function *)
let pos = filter (fun x -> x > 0);;

pos [1 ; -1 ; 2 ; -3 ; -4 ; 7];;

(* fold_right: _f_ -> _base/init_ -> 'a list -> _result_
`* fold_right: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

```
 * Also known as reduce in some other languages
 * For example, if we want to sum over a list we'd write: *)

(* let rec sum l =
 *   let rec suma l acc = match l with
 *     | [] -> acc
 *     | h::t -> suma t (h+acc) in
 *   suma l 0;;
 *
 * (\* sum [1 ; 2 ; 3 ; 4];; *\)
 * (\* 1+(2+(3+(4+0))) *\)
 *
 * let rec prod l =
 *   let rec proda l acc = match l with
 *     | [] -> acc
 *     | h::t -> proda t (h*acc) in
 *   suma l 1;; *)
(* For a string, we'd concat instead of add or multiply
 * So we want to abstract this common functionality *)

(* 1, f(2, f(3,f(4))) *)

(* fold_right f init [x1 ; .... ; xn]
 * ==> f(x1, f(x2,... (f(xn, init))))
 * fold_right: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)

let rec fold_right f init l = match l with
  | [] -> init
  | h :: t -> f(h, fold_right f init t);;
(* Not really tail recursive, can make it tail recursive though *)


fold_right (fun (x,acc)->x+acc) 0 [1 ; 2 ; 3 ; 4 ; 5];; (* sum *)
fold_right (fun (x,acc)->x*acc) 1 [1 ; 2 ; 3 ; 4 ; 5];; (* prod *)

(* Concatenate as strings in list
 * Convert each int to a string and use ^ operator to concatenate 2 strings
 * init is empty string*)
fold_right (fun (x,acc)->(string_of_int x) ^ acc) "" [1 ; 2 ; 3 ; 4 ; 5];;
```

```
(* Function that adds two numbers, but doesn't work with fold_right
 * as it needs a function that takes a tuple, not 2 ints  *)
(+) 3 4;;

(* Folds the other way, will see difference with String function,
 * but not with commutative things like addition
 * fold_left f init [x1 ; .... ; xn] ==> f(xn, (f (xn-1, ... (f (x1, init)))))
 * fold_left: ('a * b' -> 'b) -> 'b -> 'a list -> 'b*)

let rec fold_left f init l = match l with
  | [] -> init
  | h::t -> fold_left f (f (h, init)) t;;

fold_left (fun (x,acc)->(string_of_int x) ^ acc) "" [1 ; 2 ; 3 ; 4 ; 5];;

(* for_all p l returns true if all elements in l satisfy p *)
(* let rec for_all p l  *)

(* exists p l returns true if there exists an elements in l satisfy p *)

(* Things in basic library *)
List.map;;
List.fold_right;;
List.fold_left;;
List.filter;;
List.for_all;;
List.exists;;
(* etc *)
(* Writing these functions is good practice *)
```

# 11 Lecture 11 *<2017-10-05 Thu>*

## 11.1 Lambda-Calculus

- Simple language consisting of variables, functions (written as $\lambda x.t$) and function application
- We can define all computable functions in the Lambda-Calculus
- Church Encoding of Booleans:

- T = $\lambda x.\lambda y.x$ Keeps first argument, throws the other.
- F = $\lambda x.\lambda y.y$ Keeps second argument, throws the other.
- Lambda-Calculus is Turing complete, can do everything with it

## 11.2 Back to the beginning

```
(*Binding variables to functions*)
let area : float -> float = function r -> pi *. r *. r
(*or*)
let area (r:float) = pi *. r *. r
```

- The variable name area is bound to the value function r -> pi *. r *. r, which OCaml prints as <fun>
  - The type is float->float
- Good question:
  - let plus x y = x + y
  - What is the type of plus?
    - An integer? (answer 1) Wrong
    - int -> int -> int (answer 2) Correct answer
    - A function (answer 3) Wrong, function is not the **type**
  - let plus' (x,y) = x+y
    - type is int * int -> int
- What are types?
  - Base types: Int, float, string...
  - If T is a type and S is a type then
    - T->S is a type
    - T*S is a type

## 11.3 Curry

$$\text{let curry } \underbrace{f}_{\text{'a * 'b -> 'c}} = fun \underbrace{x}_{\text{'a}} \underbrace{y}_{\text{'b}} \text{ -> } f \overbrace{(x,y)}^{\text{'a * b}}_{\text{'a -> 'b -> 'c}}$$

- curry $\underbrace{plus'}_{\text{int * int -> int}}$ : int -> int -> int

- fun x y -> plus' (x,y)
  - OCaml gives you <fun>
  - Shouldn't we continue evaluation plus'(x,y) and get as a final result fun x y -> x + y?
    * No, we never evaluate inside function bodies
    * When OCaml sees fun, it stops looking
      · It has a function, it's a value, it's done

### 11.4 Uncurrying

uncurry ('a -> 'b -> 'c) -> 'a * 'b -> 'c

- The type of functions is right associative
- **NOT** the same thing as 'a -> 'b -> 'c -> 'a * 'b -> 'c
- **Important** to know how to read functions.
  - Ex. plus function from earlier
  - Can also have plus x = fun y -> x + y
    * int -> (int -> int)
    * Makes a function from an int

### 11.5 Demo

We've already seen functions that return other functions: derivatives!

```
(* Write a function curry that takes as input
 * a function f:('a * 'b)->c'
 * and returns as a result a function
 * 'a->'b->'c*)
(* *)
(* curry : (('a * 'b)->'c)-> 'a -> 'b -> 'c
 * Note: Arrows are right-associative. *)
let curry f = (fun x y -> f (x,y))

let curry_version2 f x y = f (x,y)

let curry_version3 = fun f -> fun x -> fun y -> f (x,y)
```

```
  (* Uncurry *)
  (* uncurry ('a -> 'b -> 'c) -> 'a * 'b -> 'c *)
let uncurry f = (fun (x,y) -> f x y)

(* swap : ('a * 'b -> 'c) -> 'b * 'a -> 'c *)
let swap f = fun (b, a) -> f(a , b)

let plus' (x,y) = x + y

(* swap plus' ===> fun (b,a) -> plus' (a,b) *)
```

### 11.6 Partial evaluation

- A technique for optimizing and specializing programs
- Generate programs from other programs
- Produce new programs which run faster than originals and guaranteed to behave in same way
- What is the result of evaluation curry plus'?
  - $\Longrightarrow$ It's a function!
  - Result: fun x y -> plus' x y
    - Still waiting for x y
    - What if we just pass in 3? (plus 3)
      - fun y -> 3 + y
      - We generated a function!
- let plusSq x y = $\underbrace{x * x}_{horriblyExpensiveThing(x)}$ + y
  - fun y -> 3 * 3 + y (if we set x as 3), won't evaluate this expensive function until we give it a y
  - If we write:
    - plusSq 3 10
    - plusSq 3 15
    - plusSq 3 20
  - We'd have to evaluate horribly expensive function 3 times.
  - Why not store it and use it for the next computation?

#+BEGIN_SRC ocaml let betterPlusSq x = let x = horriblyExpensiveThing(x) in fun y -> x + y #+END_SRC ocaml

- Now we get:
- let x = horriblyExpensiveThing 3 in fun y -> x+y -> fun y ->9+y
  - Now we can use this function to quickly compute without having to do the expensive function
  - Partial evaluation is very important

# 12 Lecture 12 *<2017-10-06 Fri>*

- Review by Leila: Today
  - 6-7:30 pm, MC 103
- Cheat sheet correction, minimum 12 pt, not max

## 12.1 Review

Types of questions:

1. fun x -> x +. 3.3
   - What is the type?
     - float -> float
   - What does it evaluate to?
     - <fun> or fun x -> x +. 3.3
   - let x = 3 in x + 3
     - type: int
     - eval: 6
   - let x = 3 in x +. 3
     - type: error
     - eval: n/a
2. Programming in OCaml
   (a) Higher-order functions
   - Nothing too crazy since we haven't had any assignments on it

- Maybe like the built in functions we implemented the other day
- using map, for_all, filter, exists...

3. Induction proof

### 12.2 Demo, using higher order functions

```
(* simplified roulette *)
type colour = Red | Black

type result = colour option (* Result of run*)

type amt = int
type bet = amt * colour

type id = string
type player = id * amt * bet option

(* See who won *)
let compute (am, col : bet) : result -> int = function
  | None -> 0
  | Some col' -> if col = col' then am * 2 else 0

(* same as: *)

(* let compute (am, col) r = match r with
 *   | None -> 0
 *   | Some col' -> if col = col' then am * 2 else 0 *)

(* Solve all these questions without using recursion or
 * pattern matching on lists, but instead just use the HO functions we saw in class *)

let bets = [ ("Aliya", 1000, Some (400, Red)) ;
     ("Jerome", 800, Some (240, Black)) ;
     ("Mo", 900, Some (200, Black)) ;
     ("Andrea", 950, Some (100, Red))]

(* Q1: given a list of players compute the new amounts each player has and set their be
let compute_all_results (l : player list) (r : result) =
```

```
  (* Should map players to their new vals, player has name id and amt.
   * What function? Act on bet type
   * Keep id, add compute to amount and no more bet
   * Need to get bet out of bet option
   * Use pattern matching*)

  (* List.map (fun (id, amt, bopt) -> match bopt with
   *                                  | None -> (id, amt, bopt)
   *                                  | Some b -> (id, amt + compute b r , None)) l *)

  (* Alternative with function *)
  List.map (function (id, amt, Some b) -> (id, amt +compute b r , None)
   | (id, amt, None) -> (id, amt, None)) l
;;
compute_all_results bets (Some Red);;

(* Q2: given a list of bets and a result
compute a list of winning players with their bets *)

(* Use filter *)
let compute_winners (l : player list) (r : result) =
  List.filter(function (id, amt, Some b) -> compute b r > 0
     | (id, amt, None) -> false) l
;;
compute_winners bets (Some Red);;

(* Q3: given a list of bets and a result compute
 * how much money the casino needs to pay back*)

(* Use fold *)

(* Q4 : given a list of bets and a result
 * compute if nobody won *)
(* Check if there is a winner (exists ho func) or if everyone is a loser (for_all) *)
```