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 My-Courses (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 <u>let rec</u>

- 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 all at once to one at a time) 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)**)
   - $e \implies {}^*e'$ 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)[])
     - $\xRightarrow{\text{By insert program}}$ lookup k [(k,v)] (same as (k,v)::[]) $\xRightarrow{\text{By lookup}}$ 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))
       * $\xRightarrow{\text{By insert}}$ lookup k ((k,v)::t)
       * $\xRightarrow{\text{By lookup}}$ Some v (good)
     - Subcase: $k < k'$
       * lookup k (insert(k,v) ((k',v'):: t))
       * $\xRightarrow{\text{By insert}}$ 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 $\Downarrow$
   - rev_tr l acc $\Downarrow$ v
   - Not quite because:
     - rev [] $\Downarrow$ []
     - rev_tr [] acc $\Downarrow$ acc
     - Not returning the same thing given empty list
   - Slightly modified so it's right:
     - rev l @ acc $\Downarrow$ v

– rev_tr l acc $\Downarrow$ v

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

- Case l=[]
  - rev [] @ acc
  - $\xrightarrow{\text{prog rev}}$ [] @ acc $\rightarrow$ acc
  - rev_tr [] acc
  - $\xrightarrow{\text{by prog rev\_tr}}$ acc
- Case $l = h :: t$
  - **IH: For all acc rev t @ acc $\Downarrow$ v and rev_tr t acc $\Downarrow$ v**
  - rev (h::t) @ acc $\xrightarrow{\text{by rev}}$ (rev t @ [h])@acc
    - $\xrightarrow{\text{By associativity of @}}$ rev t @ ([h] @ acc)
    - $\xrightarrow{@}$ rev t @ (h::acc)
  - rev_tr (h::t) acc $\rightarrow$ rev_tr t (h::acc)
  - By the IH rev t @ (h::acc) $\Downarrow$ v and rev_tr t (h::acc) $\Downarrow$ 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') $\rightarrow$ ('a $\times$ 'b)tree $\rightarrow$ ('a $\times$ '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) <u>Correct answer</u>
    - 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

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

- 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) *)
```

13 Lecture 13 *<2017-10-12 Thu>*

13.1 Midterm Review

See either 13.ml or Midterm.ml for the questions.

14 Lecture 14 *<2017-10-13 Fri>*

How can we do imperative programming (like C) in a functional language?

- So far, expressions in OCaml have:
  - An expression has a type
  - Expression evaluates to a value (or diverges)
- Today:
  - Expressions in OCaml may also have an effect (one effect is allocating values to memory and updating them)

14.1 Overshadowing

Recall:

```
let (k : int) = 4;;
let (k : int) = 3 in k * k;;
k;; (* This will be 4! *)
```

Binding in line 2 will be gone after line 2.

```
let pi = 3.14;;
let area (r:float) = pi *. r *. r;;

let a2 = area (2.0)

let (pi : float) = 6.0;;

let b1 = area (2.0) = a2 (* True *)

let area (r:float) = pi *. r *. r;;
let b2 = area (2.0) = a2 (* False *)
```

For b1, calling area will use the old definition of pi (it already evaluated it when we created the function).

14.2 State

How to program with state? We may want to update memory, for example if we have values that change or an array.

- How to allocate state?

```
let x = ref 0
```

Allocates a reference cell with the name x in memory and initializes it with 0. Not the address, cannot do address manipulation.

- How to compare 2 reference cells?
  - Compare their address: `r == s`
    - Succeeds if both are names for the same location in memory.
  - Compare their content: `r = s`

```
let x = ref 0
let y = ref 0
x = y (*true*)
x==y (*false*)
```

- How to read value stored in a reference cell?
  - `!x`
  - `let {contents = x} = r`
    - Pattern match on value that is stored in the reference cell with name x
- How to update value stored?
  - `x := 3`

14.3 Demo

```
let r = ref 0;;
let s = ref 0;;
r = s;; (* True *)
r == s;; (* False *)
r := 3;;
(* Update, but it returns a unit, uninteresting
 * Always true, what it says is it succeeded
 * As an effect, it changes the value in cell x.
 * But the _value_ of updating a cell is
 * (). unit is type, it evaluates to ()
```

```
 * Keep this in mind*)
!r;;
let x = !r + !s;; (* 3 *)

(* The following is not valid: *)
(* r := 3.4;; *)
(* This is because r is an int ref *)

r := !s;;
!r;;

r := 2+3;; (*2+3 evaluated before stored*)
!r;;

(* What's the value of r? (_IMPORTANT_)
 * The address/location in memory
 * So t = r will set t to the same address *)
let t = r;; (* Point to same loc in mem*)
t == r;;
t := 4 * 3;;
!t;;
!r;;

(* Polymorphic functions, will see later *)
let id = ref (fun x -> x);;
id := fun x -> x + 1;;
(* Will fix x to an int *)
(* This won't work: *)
(* id := fun x -> x +. 3.2;; *)

(* Can only do something like *)
id := fun x -> x + 2;;
(* Can also overshadow references *)
let id = ref (fun x -> x + 1);;

(* Back to the area example *)
let pi = ref 3.14;;
let area r = !pi *. r *. r;;

let a2 = area (2.0);; (*12.56*)
```

```
pi := 6.0;;
let a3 = area (2.0);; (*24.0*)
a2 = a3;; (* false *)

(* Now we can write C like
 *   programs using references *)

(* Purely functional,
 * changes addresses in triple *)
let rot (a,b,c) = (c,b,a);;

(* Purely rotten,
 * changes contents in triple *)
let rott (a,b,c) = let t = !a in (a := !c ; c := t ; (a,b,c)) ;;

let triple = (ref 1, ref 2, ref 3);;
rot triple;;
rott triple;;
(* They don't both do the same thing,
 * since one changes addresses vs contents *)


(*Imperative factorial
 * More complicated than purely functional ver
 * Considered bad style in functional
 * Harder to reason about its correctness
 * Harder to understand*)
let imperative_fact n =
  begin
    let result = ref 1 in
    let i = ref 0 in
    let rec loop () =
      if !i = n then ()
      else (i := !i + 1; result := !result * !i; loop ())
    in
    (loop (); !result)
  end
```

- Updating a cell in memory has
    - a value (i.e. unit, written in OCaml as `()`)

- an effect (i.e. changes the value in cell x)

- Types
  - `let r = ref 0`
    - Type of r: int ref
    - ref 0 is an int ref
    - 0 is an int
    - We cannot store a float in r
    - `!r` is an int
    - For `r:=3+2` to make sense, r should be an int ref and 3+2 must be an int. This returns a type of unit.

15 Lecture 15 <*2017-10-17 Tue*>

15.1 Warm up

Given the following expression write down its type, its value (i.e. what the expression evaluates to), and its effect, if it has any.

- Usually on exams, they should all type check. The error option is just in case they make a typo in typing the question.
- `3+2`
  - `int`
  - `5`
  - No effect
- `55`
  - `int`
  - `55`
  - No effect
- `fun x -> x+3 *2`
  - `int -> int`
  - `<fun>` or `fun x -> x + 3 * 2`
- `((fun x -> match x with [] -> true | y::ys -> false), 3.2 *. 2.0)`

  - ('a list -> bool) * float
  - (<fun>,6.4)
  - No effect

- let x = ref 3 in x := !x + 2
  - Example: let k=1 in k+2 is an int, and k=1 gets discarded after k+2
  - unit
  - ()
  - Effect? No, x is disposed of. Removed from the stack after evaluation in this example. x is now unbound

- fun x -> x := 3
  - int ref -> unit
  - <fun>
  - Effect: updated x to 3

- (fun x -> x := 3) y
  - type: unit
  - value: ()
  - Effect: updated y to 3

- fun x -> (x := 3; x) (returns x)
  - int ref -> int ref

- fun x -> (x := 3; !x) (returns !x)
  - int ref -> int

- let x = 3 in print_string (string_of_int x)
  - type: unit
  - value: ()
  - Effect: prints 3 to the screen

15.2 Demo

```
();;
(* unit *)

fun x -> x := 3;;
let y = ref 1;;
(fun x -> x := 3) y;;
y;;
let x = ref 1 in
    fun x -> (x := 3; x);;

(* Linked list *)
type 'a rlist = Empty | RCons of 'a * ('a rlist) ref;;
let l1 = ref (RCons (4, ref Empty));;
let l2 = ref (RCons (5, l1));;
(* The 'a rlist ref of l2 is l1, same address *)

(* What happens here? *)
l1 := !l2;;
(* We have created a circular list *)
!l1;;
```

16 Lecture 16 *<2017-10-19 Thu>*

16.1 Demo

- Mutable Data-Structures
- Closures and Objects

```
type 'a rlist = Empty | RCons of 'a * ('a rlist) ref

let l1 = ref (RCons (4, ref Empty))
let l2 = ref (RCons (5, l1));;

l1 := !l2;;
(* Value is (), effect is changing link to itself *)

(* Append for regular lists *)
let rec append l1 l2 = match l1 with
```

```
  | [] -> l2
  | x::xs -> x::(append xs l2)

(* Append for rlist *)
type 'a refList = ('a rlist) ref
(* Return unit, as the "result" is the effect *)
(* 'a refList->'a refList->unit *)
let rec rapp (r1 : 'a refList) (r2 : 'a refList) = match r1 with
  | {contents = Empty} -> r1 := !r2
  | {contents = RCons (x, xs)} -> rapp xs r2

(* 'a refList -> 'a refList -> 'a rlist *)
let rec rapp' (r1 : 'a refList) (r2 : 'a refList) = match r1 with
  | {contents = Empty} -> {contents = r2}
  | {contents = RCons (x, xs)} -> rapp' xs r2

let r = ref (RCons (2, ref Empty))
let r2 = ref (RCons(5, ref Empty));;


let r3 = rapp' r r2;;
r3;;
rapp r r2;;
r;;

let (tick, reset) =
  let counter = ref 0 in
  (* Input is unit, always true. Not the same as void *)
  let tick () = (counter := !counter + 1 ; !counter) in
  let reset () = counter := 0 in
  (tick, reset);;


(* Now we have 2 functions, tick and reset *)
tick ();;
tick ();;

type counter_obj = {tick : unit -> int ; reset : unit -> unit}

let makeCounter () =
```

```
  let counter = ref 0 in
  {tick = (fun () -> counter := !counter + 1 ; !counter);
   reset = (fun () -> counter := 0)};;

(* global variable *)
let global_counter = ref 0
let makeCounter' () =
  let counter = ref 0 in
  {tick = (fun () -> counter := !counter + 1 ; global_counter := !counter ; !counter);
   reset = (fun () -> counter := 0)};;

let c = makeCounter ();;
c.tick ();;
c.tick ();;
let d = makeCounter ();;
d.tick ();;
c.tick ();;
d.reset ();;
```

17 Lecture 17 *<2017-10-20 Fri>*

17.1 Exceptions

- Primary benefits:
  - Force you to consider the exceptional case
  - Allows you to segregate the special case from other cases in the code (avoids clutter!)
  - Diverting control flow!

17.1.1 Warm-up

- `3/0`
  - Type: `int`
  - Value: No value
  - Effect: Raises run-time exception `Division_by_zero`

```
let head_of_empty_list =
    let head (x::t) = x in
head []
```

- Type: `'a`
    - head is `'a list -> 'a`, so it returns `'a` in last line
    - Value: No value
    - Effect: raises run-time exception `Match_failure`
    - Would have been well-defined if we used option return type

17.1.2 Demo

1. Signal Error
    - Ex. raise Domain
2. Handle an exception
    - Try <exp> with Domain -> <exp>

```
exception Domain

let fact n =
  let rec f n =
    if n = 0 then 1
    else n * f (n-1)
  in
  if n < 0 then raise Domain
  else f(n)

let runFact n =
  try
    let r = fact n in
    print_string ("Factorial of "^ string_of_int n ^
    " is " ^ string_of_int r ^ "\n")
  with Domain ->
    print_string("Error: Trying to call factorial on a negative input \n");;

fact 0;;
fact (-1);;

(* let fact' n =
 *   let rec f n =
 *     if n = 0 then 1
```

```
 *     else n * f (n-1)
 *   in
 *   if n < 0 then raise (Error "Invalid Input")
 *   else f(n)
 *
 * let runFact' n =
 *   try
 *     let r = fact n in
 *     print_string ("Factorial of "^ string_of_int n ^
 *                     " is " ^ string_of_int r ^ "\n")
 *                 (\* with Error msg ->
 *                  *   print_string(msg ^ "\n");; *\)
 *                 (\* Can pattern match here too *\)
 *   with Error "Invalid Input" -> print_string ("Programmer says you passed an invalid
 *      | Error msg -> print_string(msg ^ "\n") *)

type key = int

type 'a btree =
  | Empty
  | Node of 'a btree * (key * 'a) * 'a btree



(* let l = Node (Node (Empty, (3, "3"), Empty), (7,"7"),
 *              Node (Empty, (4, "4")  *)

(* Binary search tree searching *)
exception NotFound
(* Can use exceptions for positive things as well *)
exception Found of int


(* let rec findOpt1 t k = match with
 *   | Empty -> raise NotFound
 *   | Node(l, (k',d),r) ->
 *      if k = k' then raise (Found d)
 *      else
 *        (if k < k' then findOpt1 l k else findOpt1 r k) *)
```

```
(* Now we don't assume that the tree is a binary search tree *)
let rec findOpt t k = match t with
  | Empty -> None
  | Node(l, (k',d), r) ->
     if k = k' then Some d
     else
       (match findOpt l k with
| None-> findOpt r k
| Some d -> Some d)

(* Doing it with exceptions *)

let rec find t k = match t with
  | Empty -> raise Not_Found
  | Node (l, (k', d), r) ->
     if k = k' then d
     else try (find l k with NotFound -> find r k)
```

18 Lecture 18 *<2017-10-24 Tue>*

18.1 Backtracking

- General algorithm for finding all (or some) solutions incrementally - abandons partial candidates as soon as it determines that it cannot lead to a successful solution
- Important tool to solve constraint satisfaction problems such as crosswords, puzzles, Sudoku, etc.
- Ex today:
  - Implement a function `change`. It takes as input a list of available coins and an amount `amt`. It returns the exact change for the amount (i.e. a list of available coins, `[c1;c2;...;cn]` such that `c1 + c2 + ... + cn = amt`), if possible; otherwise it raises an exception `Change`.

`change :` `int list` (list of coins) `->` `int` (amt) `->` `int list` (list of coins)

- Assumptions:
  - List of coins is ordered

    - Each coin in our list can be used as often as needed

- Good practice/exam question

```
let listToString l = match l with
  | [] -> ""
  | l ->
     let rec toString l = match l with
       | [h] -> string_of_int h
       | h::t -> string_of_int h ^ ", " ^ toString t
     in
     toString l

      (*       change [50;25;10;5;2;1] 43;;
       * [25; 10; 5; 2; 1]
       *   change [50;25;10;5;2;1] 13;;
 * [10;2;1]
 *   change [5;2;1] 13;;
 * [5;5;2;1] *)

exception Change

let rec change coins amt =
  if amt = 0 then []
  else
    (
      match coins with
      | [] -> raise Change
      (* Cannot print here, as it won't do any backtracking and
       * will also give you a type error since it's a unit and we're trying to build a
      (* raise Change is any type  *)
      | coin :: cs -> if coin > amt then change cs amt
      else (* coin <= amt*)
(* Prepend coin as we're trying to use this coin for result *)
try coin::(change coins (amt - coin) )
    (* Try, if you fail, try again without given coins*)
with Change -> change cs amt
    )
```

Backtracking with exceptions:

- Given `change[6,5,2] 9`
- `try 6::change[6;5;2] 3 with Change -> change [5;2] 9`
  - `change [5;2] 3`
    - `change[2] 3`
    - `try 2::change[2] 1` will not work, raise Change and get back to `change [5;2] 9` from before

Key thing to take away is:

- You can use exceptions for special cases like dividing by 0
- But a more interesting use is that you can divert control flow
- So you can use it to backtrack and solve problems

19 Lecture 19 *<2017-10-26 Thu>*

19.1 Modules

Primary Benefits:

- Control complexity of developing and maintaining software
- Split large programs into separate pieces
- Name space separation
- Allows for separate compilation
  - Don't always want to recompile the whole project after a small change
  - Incremental compilation & type checking
- Incremental development
- Clear specifications at module boundaries
- Programs are easier to maintain and reuse (!)
- Enforces abstractions
- Isolates bugs

19.2 Signatures (Module Types)

- Declarations can be more specific in the signature than what the module actually implements
- Tying a module to a module type we are hiding information!
  - We can change the module implementation in the future and users won't notice as long as it still implements what we specified in the signature
- Order of signature doesn't have to have the same order as module

19.3 Demo

```
(* Want to give an interface to this module
 * What functions do we want to expose?
 * Called signature in OCaml*)
module type STACK =
  sig
    type stack
    type el
    val empty : unit -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack option
    val push : el -> stack -> stack
(* val push : int -> stack -> stack *)
  (* If we change this to el -> stack -> stack
 * Stack.push 1 s wouldn't work, because 1 isn't a Stack.el*)
  end

(* 1 program unit with namespace separation *)
(* If you want to access the module need to write Stack.function *)

(* module Stack = *)

(* Specify that Stack implements STACK *)
(* module Stack : STACK = *)

(* Specify what type el is *)
module Stack : (STACK with type el = int) =
  struct
```

```
    type el = int
    type stack = int list

    let empty () : stack = []

    let push i (s : stack) = i::s

    let is_empty s = match s with
      | [] -> true
      | _::_ -> false

    let pop s = match s with
      | [] -> None
      | _::t -> Some t

    let top s = match s with
      | [] -> None
      | h::_ -> Some h

    let rec length s acc = match s with
      | [] -> acc
      | x::t -> length t 1+acc

    let size s = length s 0

    let stack2list(s:stack) = s
  end

let s = Stack.empty();;
(* empty;; -> unbound, packaged in module *)
let s1 = Stack.push 1 s;;
(* Will not show you that the stack is a list
 * Didn't specify in the signature *)
(* Cannot do: 1 :: s1;; *)
(* By tying module to signature, you are hiding information *)

module FloatStack : (STACK with type el = float) =
  struct
    type el = float
    type stack = float list
```

```
    let empty () : stack = []

    let push i (s : stack) = i::s

    let is_empty s = match s with
      | [] -> true
      | _::_ -> false

    let pop s = match s with
      | [] -> None
      | _::t -> Some t

    let top s = match s with
      | [] -> None
      | h::_ -> Some h

    let rec length s acc = match s with
      | [] -> acc
      | x::t -> length t 1+acc

    let size s = length s 0

    let stack2list(s:stack) = s
  end

module IS = Stack
module FS = FloatStack

(* How do we test the length function without a module specifying it?
 * DANGEROUS
 * Use open*)
(* open Stack;;
 * Now erases all namespace boundaries
 * Don't need to prefix anymore *)
```

20 Lecture 20 <*2017-10-27 Fri*>

20.1 More on Modules

- Can hide information when we bind to a module type
- Don't expose how we implement it
- Can even hide what elements we store in the stack
- Nice level of abstraction, so we can easily rip out the implementation and put in a new one
- Makes programs easy to maintain
- Modules are great for enforcing abstraction

20.2 Demo

- Want to implement different currencies
- Bank
- Money

```
module type CURRENCY =
  sig
    type t
    val unit : t
    val plus : t -> t -> t
    val prod : float -> t -> t
    val toString : t -> string
  end;;

(* Here, float is not yet tied to currency *)
(* Ideally, we'd also like to define multiple currencies *)
module Float =
  struct
    type t = float
    let unit = 1.0
    let plus = (+.)
    let prod = ( *. )
    let toString x = string_of_float x
```

```
  end;;

(* Abbreviation for a module *)
(* module Euro = Float *)
(* But we don't just want to abbreviate it,
 * want to also say it implements currency *)
module Euro = (Float : CURRENCY);;
module USD = (Float : CURRENCY);;
module CAD = (Float : CURRENCY);;
module BitCoins = (Float : CURRENCY);;
(* All these currencies use the same implementation of float,
 * but all referring to different implementations
 * Important that we specify CURRENCY here rather than in float
 * so that they can be different
 * Want to keep abstraction. We made plus and prod require Euros
 * so we don't just add floats*)
(* Isomorphic structures, but all accessed differently *)

(* Conversion functions  *)
let euro x = Euro.prod x Euro.unit
let usd x = USD.prod x USD.unit
let cad x = CAD.prod x CAD.unit
let bitcoins x = BitCoins.prod x BitCoins.unit

let x = Euro.plus (euro 10.0) (euro 20.5);;
(* Will not show result because it is abstract
 * Can show result by printing/showing with toString*)

Euro.toString x;;

(* Euro.plus (euro 10.0) (10.0) does not work *)

Euro.plus;;
(* Euro.t -> Euro.t -> Euro.t
 * Requires Euros *)

(* If we say that Float : CURRENCY when declaring module
 * will still get different types for
 * module Euro = Float;; module USD = Float;;
 * Just aliases for different types
```

```
 * Not actually different!
 * Binding to signature multiple times makes them all different
 * Isomorphic, can do the same stuff, but not the same type
 * But then you'll be able to add USDs and EUROs*)

(* Important principle since you're abstracting
 * implementations but also not mixing currencies *)

(* Now let's think of banks and their view
   and how we'll implement it for them *)
module type CLIENT = (* Client's view*)
  sig
    type t (* account *)
    type currency
    val deposit : t -> currency -> currency
    val retrieve : t -> currency -> currency
    val print_balance: t -> string
  end;;

module type BANK =
  sig
    include CLIENT (* Inheritance *)
(* Don't have to literally copy all things from the client module
 * Now has the same things as client*)

    val create : unit -> t

  end;;

(* We want banks of different currencies
 * with particular functions, adding 2 currencies
 * printing currencies, etc.*)

(* Parameterize a module Old_Bank with the functionality
   provided by the module type CURRENCY *)
(* Should not matter what type of currency,
   bank should be able to do the same thing regardless of currency *)

(* Module parametized by another module is also called a functor
 * Similar to higher order functions but for modules *)
```

```
module Old_Bank (M : CURRENCY) : (BANK with type currency = M.t) =
  (* M describes a module, can implement a module
     with a module for currency, M *)
  struct
    type currency = M.t
    type t = { mutable balance : currency }
    (* Could have made it a currency ref *)

    let zero = M.prod 0.0 M.unit
    and neg = M.prod (-1.0)

    let create() = { balance = zero }

    let deposit c x =
      if x > zero then
c.balance <- M.plus c.balance x; c.balance

    let retrieve c x =
      if c.balance > x then
deposit c (neg x)
      else
c.balance

    let print_balance c =
      M.toString c.balance
  end;;

(* How do we get an implementation of a bank now? *)

module Post = Old_Bank (Euro);;
(* How to make the client see less? *)
module Post_Client : (CLIENT with type currency = Post.currency and type t = Post.t) =
(* Tells you what is shared with Post *)

let my_account = Post.create () ;;
Post.deposit my_account (euro 100.0);;
Post_Client.deposit my_account (euro 10.00);;
Post.print_balance my_account;;
Post_Client.print_balance my_account;;
(* Shared functionality among the two, but different ways of accessing *)
```

```
module Citybank = Old_Bank (USD);;
module Citybank_Client : (CLIENT with type currency = Citybank.currency and type t = Ci

let my_cb_account = Citybank.create ();;
Citybank.deposit my_cb_account (usd 50.00);;
(* Citybank_Client.deposit my_account;; Won't work *)
```

21 Lecture 21 *<2017-10-31 Tue>*

21.1 Continuations

A **continuation** is a representation of the execution state of a program (for example a call stack) at a certain point in time.

Save the current state of execution into some object and restore the state from this object at a later point in time resuming its execution.

21.1.1 First-class Support for Continuations

C#	async/wait
Racket	call-with-current-continuation
Ruby	callcc
Scala	shift/reset
Scheme	callcc

Ocaml doesn't have first-class support, so we'll be using functions as continuations! Back to higher order functions.

- Back to the beginning: Recall what tail-recursive means. Can every recursive function be written tail-recursively?

```
let rec append l k = match l with
  | [] -> k
  | h::t -> h::append t k
```

- Not tail-recursive, because of `h::append t k`
- But still efficient
- Can we rewrite it tail-recursively?

21.1.2 Recipe

How to re-write a function tail-recursively?

- Add an additional argument, a **continuation**, which acts like an accumulator
- In the base case, we call the continuation
- In the recursive case, we build up the computation that still needs to be done.

A continuation is a stack of functions modeling the call stack, i.e. the work we still need to do upon returning.

- Not always easy to do this, the earlier attempt at tail recursion for append reversed the list.

21.1.3 Demo

```
(* append: 'a list -> 'a list -> 'a list *)
let rec append l k = match l with
  | [] -> k
  | h::t -> h::(append t k)

(* Tail recursive *)
(* app_tl: 'a list -> 'a list -> ('a list -> 'a list) -> 'a list *)
(* First 'a list in c is "waiting for result" of rec call
   and last one is final result *)
let rec app_tl l k c = match l with
(* | [] -> ?
 * | h::t -> app_tl t k ? *)
(* How to do this? Need something with a hole, i.e. a function *)
  (* When you do app [1;2] [3;4] ([3;4 = k]) -> 1::app [2] [3;4]
   * -> 1::2::app [] [3;4] What to give to c here? [3;4]*)
  (* c = (fun r -> 1::2::r) *)
  (* Parameterized function, can call and start using *)
  (* c k -> 1::2::[3,4] *)
  | [] -> c k (* Calling the continuation
       - passing to the call stack k*)
  (* What to put here? *)
  (* | h::t -> app_tl t k (fun r -> h :: c r) *)
```

```
  (* This gives reverse order *)

  (* Building up the call stack: *)
  (* app_tr [1;2] [3;4] (fun r -> r) (initial continuation), ident
   -> app_tr [2] [3;4] (fun r1 -> (fun r->r) (1::r1))
   -> app_tr [] [3;4] (fun r2-> (fun r1 -> (fun r->r) (1::r1))) (2::r2)
   Collapsing the call stack
   -> (fun r2 -> (fun r1 -> (fun r -> r) (1::r1)) (2::r2)) [3,4]
   -> (fun r1 -> (fun r-> r) (1::r1)) [2;3;4]
   -> (fun r->r) [1;2;3;4] -> [1;2;3;4] *)
  | h::t -> app_tl t k (fun r -> c (h::r))

let rec genList n acc =
  if n > 0 then genList (n-1) (n::acc) else acc;;

let l1 = genList 8000000 [];;
let l2 = genList 4000000 [];;

(* append l1 l2 gives stack overflow*)
(* So does l1 @ l2
 * Ocaml didn't implement it through tail-recursion
 * For short lists, it works faster
 * But it cannot append huge lists like this one
 * Program can crash vs program being a bit slower*)
app_tl l1 l2 (fun r -> r);;

let rec map l f = match l with
  | [] -> []
  | h::t -> (f h)::map t f

(* Past interview question, wanted to reimplement map tail recursively *)

let map' l f =
  let rec map_tl l f c = match l with
    | [] -> c []
    (* Build up calling stack *)
    | h::t -> map_tl t f (fun r -> c ((f h):: r))
  in
  map_tl l f (fun r -> r)
```

22 Lecture 22 *<2017-11-02 Thu>*

22.1 Continuation recap

- Building up the stack
- Once you hit base, it collapses
- Seen an example, tail-recursion: the continuation is a functional accumulator; it represents the call stack built when recursively calling a function and builds the final result
- Failure Continuation: the continuation keeps track of what to do upon failure and defers control to the continuation
- Success Continuation: the continuation keeps track of what to do upon success, defers control to the continuation, and builds the final result

22.1.1 Demo

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

let leaf n = Node (Empty, n, Empty)

let r = Node (leaf 22, 35, leaf 70)
let ll = Node (leaf 3, 5, leaf 7)
let l = Node(ll, 9, leaf 15)
let t = Node (l, 17, r)

(* Contrasting three different versions *)
(* Using options *)
(* find: ('a -> bool) -> 'a tree -> 'a option *)
(* Good exercise to understand a function, write types *)
let rec find p t = match t with
  | Empty -> None
  | Node (l, d, r) ->
     if (p d) then Some d
     else (match find p l with
   | None -> find p r
   | Some d' -> Some d')
```

```
(* Kind of messy and overcomplicated
 * Building up a call stack that's useless
 * Have to go down all of l and then only
 * r when we're done with l*)


(* Using exceptions *)
exception Fail

let rec find_ex p t = match t with
  | Empty -> raise Fail
  | Node(l,d,r) -> if (p d) then Some d
   else (try find_ex p l with Fail -> find_ex p r)
(* If you fail, backtrack on right tree *)
let find' p t =
  (try find_ex p t with Fail -> None)


(* Using failure continuation *)
(* Continuation: Base case, want to call call stack *)
(* In this case it's c *)
(* find_cont ('a -> bool) -> 'a tree -> (unit -> 'a option) -> 'a option *)
let rec find_cont p t c = match t with
  | Empty -> c ()
  (* Instead of raising and causing an effect here, we call continuation stack *)
  | Node(l,d,r) -> if (p d) then Some d
   else find_cont p l (fun () -> find_cont p r)
(* If left fails, then we go on left side
 * Building up call stack*)

let rec find'' p t = find_cont p t (fun () -> None)
       (* If we're at the very last leaf,
   will pass unit to call stack and will get None
 * Could have also made a fail exception*)


find'' (fun x -> x = 22) t;;
(* find_cont p t (fun (0 -> None))
 * -> find_cont p t (fun () -> None)
 * If this fails
```

```
 * -> find_cont p l (fun () -> find_cont p r (fun () -> None))
 * Building up stack in case going down left fails
 * Passing it up and getting ready for next time like with exceptions
 * -> find_cont p ll (fun () -> find) cont p lr ....
 * This consists of building up the call stack, remember what to do upon fail*)

(* Good reason to use this is to make it tail-recursive and not get out of memory probl

(* Finding all elements satisfying a given property *)
(* Recursive *)
(* findAll: ('a -> bool) -> 'a tree -> 'a list *)
let rec findAll p t = match t with
  | Empty -> []
  | Node (l, d, r) ->
     let el = findAll p l in
     let er = findAll p r in
     if (p d) then el @ (d :: er)
     else el @ er;;

findAll (fun x -> x mod 3 = 0) t;;

(* Continuations, but this time on success to build up result *)
let rec findAll' p t sc = match t with
  | Empty -> sc []
  | Node(l, d, r) ->
     if (p d) then
       findAll' p l (fun el -> findAll' p r (fun er -> sc (el @ (d::er))))
     else
       findAll' p l (fun el -> findAll' p r (fun er -> sc (el @ er)))
(* Figure this out yourself *)
```

23 Lecture 23 *<2017-11-03 Fri>*

23.1 Regular expressions

Going to implement regex with continuations. This is another example of a success continuation.

You might have used them in a Unix shell:

- `ls *.ml`, list files with suffix ml

- `ls hw[1-3].ml`, lists hw1.ml, hw2.ml, hw3.ml

Patterns for regular expressions:

- Singleton: Matching a specific character
- Alternation: choice between two patterns
- Concatenation: Succession of patterns
- Iteration: indefinite repetition of patterns

More rigorously: Regular expression $r ::= a|r_1 + r_2|r_1 r_2|r^*|0|1$

- Backus-Naur-Form
- This is an inductive definition
- How to read this?
- A regex is either:
  - a, a character (anything of type char)
  - r_1 r_2 is concatenation, have both regular expressions
  - $r_1 + r_2$ alternation, can have either
  - r^* iteration, indefinite amount
  - 0 None
  - 1 Success

Question: When does a string s match a regular expression r?

- If s is in the set of terms described by r. (When does this happen?)

23.1.1 Examples:

- $a(p^*)l(e + y)$ would match *apple* or *apply* or *ale*
- $g(1+r)(e+a)y$, either you succeed (skip it) or find r, *grey*,/gray/,/gay/
- $g(1 + o)^*(gle)$, *google*,/gogle/,/goooogle/,/ggle/ (1+o) doesn't matter here, could just be o
- $b(ob0 + oba)$, *boba* but not bob

23.1.2 Demo

```
(* BNF notation *)
type regexp =
  Char of char | Times of regexp * regexp | One | Zero |
  Plus of regexp * regexp | Star of regexp

(* How to write? *)
(* String s:
 * Never matches 0
 * Matches 1 if s = empty
 * Matches a iff s = a
 * Matches r1+r2 iff s matches r1 or r2
 * Matches r1 r2 iff s1 = s1 s2
   with s1 matching r1 and s2 matching r2
 * Matches r* iff s= empty or s = s1 s2
   with s1 matching r1 and s2 matches r* *)

(* acc: regexp -> char list (each char that makes up string)
   -> (char list -> bool) (success continuation) -> bool *)
(* Accumulating things on call stack,
   return type of success continuation must be
   the same as return type of acc, i.e. bool *)
(* acc (Times(r1,r2)) s *)
(* Idea: check if a prefix of s matches r1 *)
(* But we still need to check the remaining part of s with r2
 * So we pass the whole char list to k *)
(* We use continuations as backtracking, so we don't have to
   worry about where we split the string *)
let rec acc r clist k = match r, clist with
  | Char c, [] -> false (* Expected some characters, got nothing *)
  | Char c, c1::s -> (c = c1) && k s (* At the bottom of the stack
If we succeed, pass rest of string to                                       call stack*)
  | Times (r1, r2), s -> acc r1 s (fun s2 -> acc r2 s2 k)
  (* Can we match remaining string as well?
Pass to continuation and also call k, whatever it still has to do*)
  | One, s -> k s (* Nothing to do, call continuation with whole string *)
  | Plus (r1, r2), s -> acc r1 s k || acc r2 s k (*Only has to be 1*)
  | Zero, s -> false
  | Star r, s ->
```

```
    (k s) || acc r s (fun s2 -> not (s = s2) && acc (Star r) s2 k)
(* Make sure we're consuming something, s2 is smaller *)
(* Note that (ap* )(l (e+g)) is represented as:
 * Times(Times(Char "a", Star(Char "p")),Times(Char "l",Plus(Char "e", Char "y")))*)

let string_explode s =
  tabulate (fun n -> String.get s n) ((String.length s) - 1)

let string_implode l =
  List.fold_right (fun c s -> Char.escape c ^ s) l ""

  (* let accept r s =
   *   acc r string_explode s  *)
```

24 Lecture 24 *<2017-11-07 Tue>*

24.1 Lazy Programming

24.1.1 Eager vs Lazy

- Eager Evaluation
  - Evaluate expressions by call-by-value
  - Variables are bound to values
  - Ex. `let x = 3+2 in x * 2`
    1. Evaluate expression `3+2` to the value `5`
    2. Evaluate expression `x*2` in an environment where variable `x` is bound to the value `5` to the final value `10`
  - Ex. let x = horribleComp (345) in 5
    1. Evaluate expression horribleComp (345) to some value `777`
    2. Evaluate expression `5` in an environment where variable `x` is bound to the value `777` to the final value `5`. Here we bind an expression we don't need.
- Lazy Computation
  - Ex. `let x = horribleComp (345) in 5`
  - Bind variables to unevaluated expressions (not values!)
  - Suspend computation `horribleComp (345)` until needed

- Memoize results because:
    - `let x = horribleComp (345) in x + x` will recompute horribleComp twice
- Lazy usually doesn't go well with state, which is why most imperative languages are eager
- Harder to reason about but very useful for:
    - Infinite data (ex. representing all prime numbers, reading part of a file instead of the whole thing)
    - Interactive data (ex. sequence or stream of inputs)

24.1.2 Finite vs Infinite

- Finite Data
    - `type 'a list = Nil | Cons of 'a * 'a list`
        - Encodes an inductive definition of finite lists
        - `Nil` is a list of type `'a list`
        - If `x` is of type `'a` and `xs` is a list of type `'a list` then `cons(x,xs)` is a list of type `'a list`
        - Nothing else is a list
    - How do we take apart lists? By pattern matching.
    - How do we reason with lists? By induction on the structure of lists
- Infinite Data
    - Instead of saying how to construct infinite data, we define it by the observations we make about them
    - Given a stream `1,2,3,4,5,...` we can ask:
        - The head of the stream: `1`
        - The tail of the stream: `2, 3, 4, 5, ...`
    - Can we always make an observation? Does this terminate eventually? No, does not terminate.
        - Observations should be productive, shouldn't go into infinite loops. We can make an observation at each step.

24.1.3 Suspending

How to suspend and prevent evaluation of an expression?

```
  type 'a susp = Susp of unit -> 'a

  (* Force evaluation of suspended computation *)
  let force (Susp f) = f ()

(* Example of suspending and forcing computation *)
let x = Susp (fun () -> horribleComp(345)) in force x + force x
```

Wrap your function in lazy programming so that it suspends computation

- Infinite Streams: `type 'a str = {hd :  'a ; tl :  ('a str) susp}`
  - Encodes a coinductive definition of infinite streams using the two observations `hd` and `tl`
    - Asking for the head using the observation `hd` returns element of type `'a`
    - Asking for the tail using the observation `tl` returns a suspended stream of type `('a str) susp`
    - If you want more elements you need to ask for more
    - Very demand driven

24.2 DEMO

```
type 'a susp = Susp of (unit -> 'a)
(* Delay computation *)

(* Force computation *)
let force (Susp f) = f ()

type 'a str =
  {hd : 'a ; tl : ('a str) susp}

    (* Stream of 1, 1, 1, 1, 1 .... *)
    (* ones : int str *)

let rec ones =
  {hd = 1 ;
```

```
   tl = Susp (fun () -> ones) (* Suspend comp to generate more ones *)
  }

(* numsFrom n  generates stream n, n+1, n+2, ... *)
let rec numsFrom n =
  {hd = n ;
   tl = Susp (fun () -> numsFrom (n+1))} (* Ocaml stops evaluating at function *)

let nats = numsFrom 0

(* series starting with n and i-th element is k^i *)
let rec pow_seq n k =
  {hd = n ;
   tl = Susp (fun () -> pow_seq (n*k) (k))}

(* add: int str -> int str -> int str *)
let rec add s1 s2 =
  {hd = s1.hd + s2.hd;
   (* tl = Susp (fun() -> add s1.tl s2.tl)  -> this gives you an error, need to force i
   tl = Susp (fun() -> add (force s1.tl) (force s2.tl))
  }

(* smap: ('a -> 'b) -> 'a str -> 'b str *)
let rec smap f s =
  {hd = f (s.hd) ;
   tl = Susp (fun () -> smap f (force s.tl))
  }
(* take: int -> 'a str -> 'a list *)
(* peels off n elements from stream *)
let rec take n s = if n = 0 then []
   else s.hd :: take (n-1) (force s.tl);;

take 10 ones;;
take 10 (numsFrom 1);;
take 10 (pow_seq 1 2);;
take 10 (add nats ones);;
take 10 (add nats nats);;
take 10 (smap (fun x -> x*2) nats);;
```

25 Lecture 25 *<2017-11-09 Thu>*

25.1 Demo

```
(* Continuation of last class *)
#use "24.ml";;

(* Generate: 1, 1/2, 1/4, 1/8, 1/16 *)
(* geom_series: float -> float str *)
let rec geom_series x =
  {hd = (1.0 /. x) ;
   tl = Susp (fun () -> geom_series (x *. 2.0))
  };;

take 10 (geom_series 1.0);;

(* zip: ('a * 'b -> 'c) -> 'a str -> 'b str -> 'c str *)
let rec zip f s1 s2 =
  {hd = f (s1.hd,s2.hd) ;
   tl = Susp (fun () -> zip f (force s1.tl) (force s2.tl))
  }

(* sfilter: ('a -> bool) -> 'a str -> 'a str *)
let rec sfilter f s =
  let h,t = find_hd f s in
  {hd = h ;
   tl = Susp (fun () -> sfilter f (force t))
  }

(* find_hd: ('a bool) -> 'a str -> 'a * ('a str) susp *)
(* Use to find first element in stream satisfying f, return with tail     *)
and find_hd f s =
  if (f s.hd) then (s.hd, s.tl) else find_hd f (force s.tl);;

(* All even numbers *)
take 10 (sfilter (fun x -> x mod 2 = 0) nats);;

(* Will filter always be productive? No. It may never be true in a stream
 * Your responsibility now *)
```

```
(* Sieve of Eratosthenes for prime numbers *)
(* Start with natural numbers from 2 *)
(* Take head and filter out all elements divisible by head *)
(* Repeat *)
let rec sieve s =
  {hd = s.hd ;
   tl = Susp(fun() -> sfilter (fun x -> not (x mod s.hd = 0)) (sieve (force s.tl)))
  }
let nats2 = numsFrom 2
let primes = sieve nats2;;

take 10 primes;;
(* take 1000 primes;; *)

(* fib(n+1)=fib(n)+fib(n-1) *)
(* fibs 0 1 1 2 3 5 *)
(* Make a copy of this but shifted *)
(* Add both entries to get next fib number *)
(* Essentially adding two streams *)
let rec fibs =
  {hd = 0 ;
   tl = Susp (fun () -> fibs')
  }
and fibs' =
  {hd = 1 ;
   tl = Susp (fun () -> add fibs fibs')};;

take 10 fibs;;
```

25.1.1 Sieve of Eratosthenes:

Generate a stream of prime numbers

- Start with natural numbers starting at 2
- 234567891011121314151617...
- First number, 2 is a hit.
- Now look at tail and filter out all even numbers.
  - Now you have a stream of odd numbers.

- Head this time is 3. Now remove all things that are multiples of 3

26 Lecture 26 *<2017-11-10 Fri>*

26.1 Fundamental Principles in Programming Language Design

Three key questions:

- What are the syntactically legal expression? What expressions does the parser accept?
  - Grammar
- What are well-typed expressions? What expressions does the type-checker accept?
  - Static semantics
- How is an expression executed?
  - Dynamic semantics
  - Why isn't a program working? Seems logically correct, but then how do we delve further?

Definitely need to know the answers to all 3 questions if you're writing a compiler or your own language.

26.1.1 Syntactically Legal Expressions

Def. (Not the definition for Ocaml, but similar and essentially for a basic language)

The set of expressions is defined inductively by the following clauses

1. A number n is an expression
2. The booleans `true` and `false` are expressions.
3. If e_1 and e_2 are expressions, then e_1 op e_2 is an expression where $op = \{+, =, -, *, <\}$
4. If e, e_1 and e_2 are expressions, then `if` e then e_1 `else` e_2 is an expression

Example of legal expressions:

- `if 3 then 2+3 else 5`
  - This is syntactically fine, but it's a type error

When we define languages, we typically have 3 stages:

1. source -> lexer -> parser (syntax errors here)
   - After parse, you have an abstract syntax tree
   - How do I read `3+2=7`? `(3+2)=7` or `3+(2=7)`?
   - The definition above is tree like (with the if statements)
   - We won't talk much about parsing and lexing, as most of that has been solved
2. -> type checker (if it exists, also type errors are here) -> interpreter & evaluation (run-time errors/run time exceptions)

26.1.2 Backus-Naur Form (BNF)

Alternative to the previous definition.
Operations op ::= $+ | - | * | < | =$
Expressions e ::= $n | e_1 \ op \ e_2 | true | false | if \ e \ then \ e_1 \ else \ e_2$

1. Representation in OCaml

```
type primop = Equals | LessThan | Plus | Minus | Times

type exp =
| Int of int (* 0 | 1 | 2 | ... *)
| Bool of bool (* true | false *)
| If of exp * exp * exp (* if e then e_1 else e_2 *)
| Primop of primop * exp list (*e_1 primop e_2 or primop e *)
```

   Example:

   - `if 3<0 then 1 else 0` is represented as `If (Primop (LessThan, [Int 3 ; Int 0]), Int 1, Int 0)`
   - `if true then 3` cannot be represented, no else

26.1.3 Evaluation

How to evaluate an expression?

- How to describe evaluation of expressions?
- We want to say: "Expression e evaluates to a value v"
- What are values?
  - Values $v ::= n|true|false$

Definition

Evaluation of the expression e to a value v is defined inductively by the following clauses:

- A value v evaluates to itself.
- If expression e evaluates to the value `true` and expression e_1 evaluates to a value v, then `if` e `then` e_1 `else` e_2 evaluates to the value v.
- If expression e evaluates to the value `false` and expression e_2 evaluates to a value v, then `if` e `then` e_1 `else` e_2 evaluates to the value v.

Too verbose and doesn't even talk about primitive operators. Need more compact notation.

$e \Downarrow v$ for "Expression e evaluates to value v"

Definition with new notation:

$e \Downarrow v$ is defined inductively by the following clauses:

- $v \Downarrow v$
- If $e \Downarrow$ `true` and $e_1 \Downarrow v$, then `if` e `then` e_1 `else` $e_2 \Downarrow v$
- If $e \Downarrow$ `false` and $e_2 \Downarrow v$, then `if` e `then` e_1 `else` $e_2 \Downarrow v$

Turning informal description into a formal one

$$\frac{premise_1 \ \ldots \ premise_n}{conclusion}\text{name}$$

Read as: If $premise_1$ and $premise_2$ and $\ldots premise_n$, then conclusion.

$$\frac{}{v \Downarrow v}\text{B-VAL}$$

$$\frac{e \Downarrow true \ e_1 \Downarrow v}{if \ e \ then \ e_1 \ else \ e_2 \Downarrow v}\text{B-IFTRUE}$$

$$\frac{e \Downarrow false \;\; e_2 \Downarrow v}{if \;\; e \;\; then \;\; e_1 \;\; else \;\; e_2 \Downarrow v}\text{B-IFFALSE}$$

Here we only talk about the good things, `true` or `false`. We don't define what would happen if $e \Downarrow$ something else, like a number. Then we would raise not defined.

27 Lecture 27 *<2017-11-14 Tue>*

27.1 Designing our own language

(Continuation from last class)

We transformed evaluation into a formal description with premises and a conclusion. Note that evaluation rules do not impose an order on the premises. This works if we don't have state.

Extending further:

- If $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$, then $e_1 \; op \; e_2 \Downarrow v$ where $v = \overline{v_1 \; op \; v_2}$
- $\frac{e_1 \Downarrow v_1 \;\; e_2 \Downarrow v_2}{e_1 \; op \; e_2 \Downarrow \overline{v_1 \; op \; v_2}}$ B-OP

How to use these rules?

- Example in slides

  Our operational semantics ($e \Downarrow v$) is what we formalized.

  Task: Implement a function `eval` that does what $e \Downarrow v$ describes.

```
let rec eval e = match e with
| Int _ -> e
| Bool _ -> e
| If (e, e1, e2) ->
    (match eval e with
    | Bool true -> eval e1
    | Bool false -> eval e2
    | _ -> raise (Stuck "guard is not  bool"))
```

27.1.1 Why do we care about a formal description?

- More compact

Establish properties and formal guarantees

- Coverage: For all expressions `e` there exists an evaluation rule.

- Determinacy: If $e \Downarrow v_1$ and $e \Downarrow v_2$ then $v_1 = v_2$
- Value Soundness: If $e \Downarrow v$ then v is a value.
  - Should always return a value

27.1.2 Advantages of an implementation

We can run it!

27.1.3 Static Type Checking

- Types approximate runtime behavior
- Lightweight tool for reasoning about programs
- Detect errors statically, early in the development cycle
- Great for code maintenance
- Precise error messages
- Checkable documentation of code

How do we make types work?

- Types classify expressions according to the kinds of values they compute
  - What are values though?
    - Values v ::= n | true | false
  - Hence, there are only two basic types.
    - Types T ::= int | bool
    - In OCaml -> ints, floats, functions, unit, ref, etc.

1. Types e:T, expression e has type T

   Def:

   (a) n : int
   (b) true : bool and false : bool
   (c) If e : bool and $e_1 : T$ and $e_2 : T$, then if e then e_1 else $e_2 : T$
   (d) If e_1 : int and e_2 : int, then $e_1 + e_2$: int

(e) If $e_1 : T$ and $e_2 : T$, then $e_1 = e_2 :$ bool

Can be converted to premises

- $\frac{}{true{:}bool}$ T-T
- $\frac{}{false{:}bool}$ T-F
- $\frac{e_1{:}T \;\; e_2{:}T}{e_1{=}e_2{:}bool}$ T-EQ
- $\frac{}{n{:}int}$ T-NUM
- $\frac{e_1{:}int \;\; e_2{:}int}{e_1{+}e_2{:}int}$ T-PLUS
- $\frac{e{:}bool \;\; e_1{:}T \;\; e_2{:}T}{if \;\; e \;\; then \;\; e_1 \;\; else \;\; e_2{:}T}$ T-IF

Two readings of typing

- Type Checking $e : T$
  - Given the expression e and the type T, we check that e does have type T
- Type Inference $e : T$
  - Given the expression e, we infer its type T
- OCaml does both and well
- Checking is difficult for us with T-EQ, we don't know what type T is.

27.2 Demo

```
(* Defining a new language in OCaml *)
module Exp =
  struct
    type name = string
    type primop = Equals | LessThan | Plus | Minus | Times | Negate

    type exp =
      | Var of name (* From Lecture 28 *)
      | Int of int (* 0 | 1 | 2 | ... *)
      | Bool of bool (* true | false *)
      | If of exp * exp * exp (* if e then e_1 else e_2 *)
      | Primop of primop * exp list (*e_1 primop e_2 or primop e *)
      | Let of dec * exp (* From L28 let dec in e end *)
```

```
    and dec = (* From L28 *)
      | Val of exp * name (* val x =e *)

    (* How to evaluate? *)
    module Eval =
      struct
open exp

exception Stuck of string

let evalOp op = match op with
  | (Equals, [Int i; Int i']) -> Some (Bool (i = i'))
  | (LessThan, [Int i; Int i']) -> Some (Bool (i < i'))
  | (Plus, [Int i; Int i']) -> Some (Int (i + i'))
  | (Minus, [Int i; Int i']) -> Some (Int (i - i'))
  | (Times, [Int i; Int i']) -> Some (Int (i * i'))
  | (Negate, [Int i]) -> Some (Int (-i))
  | _ -> None

let rec eval e = match e with
  | Int _ -> e
  | Bool _ -> e
  | If (e, e1, e2) ->
     (match eval e with
      | Bool true -> eval e1
      | Bool false -> eval e2
      | _ -> raise (Stuck "guard is not  bool"))
  (* Primitive operations *)
  | Primop (po, args) ->
     let argvalues = List.map eval args in
     (match evalOp (po, argvalues) with
      | Some v -> v
      | None -> raise (Stuck "wrong arguments passed to prim. op"))

      end
(* The rest is on course page *)
```

28 Lecture 28 <*2017-11-16 Thu*>

Extending our language with variables, let-expressions, and functions!

- Add the following expressions: $x | let\ x = e_1\ in\ e_2\ end$

Are the following well-formed?

- `let x = x in x + 1 end`
  - x is not defined, not well-formed
- `let x = y in x + 1 end`
  - Same problem, y is not defined.
  - Is this equal to the line above? What kind of renaming do we want?
- `let x = 1 in let y = x end in x + y end`
  - Legal
- `let x = 1 in let x = x + 2 end in x + x end`
  - Legal, important to know where things are bound, the first x bounds the x in $x + 2$. The second let x statement bounds the x in $x + x$

28.1 Free Variables

Variable names should not matter

These are all the same:

- `let x = 5 in (let y = x + 3 in y + y end) end`
- `let x = 5 in (let x = x + 3 in x + x end) end`
- `let v = 5 in (let w = v + 3 in w + w end) end`

Defining Free Variables:

- $FV(e)$ returns the set of the free variable names occurring in the expression `e`.
  - It is defined inductively based on the structure of the expression `e`.

- $FV(e_1 \ op \ e_2) = FV(e_1) \cup FV(e_2)$
- $FV(n) = \{ \}$
- $FV(x) = \{x\}$
- $FV(let \ x = e_1 \ in \ e_2)$
- $FV(e_1) \cup FV(e_1) \setminus \{x\}$
  - `let x = 1 in let y = x+1 end in x+y end`
    * First look at `x+y` -> x and y are free
    * Now looking at `let y = x + 1`, `x` is free here.
    * Apply the function to this let expression, get that `x` is free
      · $\{x\} \cup \{x, y\} \setminus \{y\}$
    * Then look at the outer `let x`. Nothing free there.
    * Apply the function again, get $\{\} \cup \{x\} \setminus \{x\}$
- $FV(if \ e \ then \ e_1 \ else \ e_2) = FV(e) \cup FV(e_1) \cup FV(e_2)$

In OCaml, `let`, functions and pattern matching bound variables. In programming languages, no variables should be free, not bound.

28.2 Substitution

Define as: `[e'/x]e` Replace all **free** occurrences of the variable `x` in the expression `e` with expression `e'`

Can be read as a function:

- Input: Expression `e'`, variable `x`, and expression `e`
- Output: an expression where all **free** occurrences of the variable `x` in expression `e` have been replaced by `e'`

Substitution is defined by considering different cases for `e`.