Cornell CS 3110 - Functional Programming

Ivin Lee

October 10, 2020

No, I'm not from Cornell. But I think this set of lectures on OCaml is really useful for learning functional programming, which is why I am including it in these notes. I studied this course before my Java course because I think it will add value to learn Java course with prior knowledge of the functional programming paradigm.

1 Introduction

Functional languages:

- define computations as mathematical functions

- avoid mutable states

The former means that every computation takes in values and outputs values (more on this later).

The latter means that you never write $x = x + 1$, because when you do so you are changing (muting) the value of x.

Imperative languages:

- mutable states

- functions have side effects

The latter means that in

```
int addValue(Wallet wallet, int change) {
    wallet.money += change;
    return wallet.money;
}
```

apart from returning the new value of the wallet, it also changes the value in the Wallet instance.

It is claimed that functional programming allows one to write correct code easier, because:

- variables never change values

- functions never change other things (and cause trouble!)

It is possible to write code following the functional programming paradigm in any programming language, but some languages have been designed to make this easier. e.g.OCaml

Benefits of OCaml:

1. Immutable programming inbuilt (can't accidentally re-write a variable after being declared)

2. Functions passed as values

3. **Automatic type inference** - quite useful from experience

2   Before We Begin

You can compile OCaml programs with `ocamlc` and `ocamlopt`. The difference is that `ocamlc` is the bytecode compiler while `ocamlopt` is the native code compiler. For reference, native code is code which works only on your CPU architecture. In contrast, bytecode is CPU independent, and needs another round of parsing to become native code that runs on your computer's hardware.

To compile a program, type
```
ocamlopt -o program.exe module1.ml module2.ml
```

You may also see
```
ocamlopt -c module1.ml module2.ml
ocamlopt -o program.exe module1.cmx module2.cmx
```

This just means that each module is compiled first, then linked up.

3   Functions

Value - expression that does not need any further evaluation

```
1  let x = if e1 then e2 else e3
2  let y = e2 +. e3
```

There is no need to declare the type as the type is inferred from e2 and e3 (e.g. if e2 and e3 are ints, then the type of x is inferred to be int).

The +. is a binary infix operator that takes in two floats and returns a float. In order for the type of y to be inferred correctly, it is necessary to use the operator, which tells the compiler(?) the return type of this expression.

Function Definitions

```
1  let rec pow x y =
2      if y = 0 then 1
3      else x * pow x (y - 1)
```

Note that you don't put brackets around the arguments. Also, all functions only return one thing, which matches the mathematical definition of a function.

```
1  let rec f x1 x2 ... xn = e
2  val f : t1 -> t2 -> ... tn -> te
```

The second line is output in the command line. Type `t -> u` is the type of a function that takes an input of type t and returns an output of type u.

Anonymous Functions

```
1  fun x -> x + 1
2  let inc = fun x -> x = 1
```

Can be used when you don't need the function to have a name, e.g. defining the function in another function's argument like in

```
1  let f x y =
2    x + y
3  f ((fun x -> x + 1) 2) 3 (* returns 6*)
```

A function is a value. In the above example, $x + 1$ is not evaluated until 2 was passed to the anonymous function.

Note that the anonymous expression syntax is analogous to lambda expres-

sions in math:

$$\lambda x.e$$
$$\text{fun } x-> e$$

Function Application Operator

```
1  f e (*equivalent to*) e |> f
2  5 |> inc |> square
```

5 is passed to inc and square

Functions are Values
Implication: functions can take functions as arguments and can return functions as results

4 Lists

Lists are constructed recursively, so the following constructors are equivalent

```
1  let x = [1;2;3]
2  let x = 1::[2;3]
3  let x = 1::2::[3]
4  let x = 1::2::3::[]
5  let x = h::t (* h: head, t: tail)
```

The type of x is a' list, where a' is the type of the elements in the tail of the list (note: the type of h is a' but the type of t is a' list!)

Pattern Matching

```
1  let f inputList =
2      match inputList with
3      |[] -> -1
4      |h::t -> h
```

Basically pattern matching allows you to look at the first element of the list, and if that doesn't satisfy you, you can recurse and look at the first element of the tail list.

Linked List
As expected, the linked list data structure is good for sequential access.

Function Keyword Another way to write this is:

```
1  let f = function
2      |[] -> -1
3      |h::t -> h
```

This function takes in one argument, matches the argument to a pattern and returns the corresponding value. The main difference is that using `match ... with` can take multiple inputs but `function` only takes in one input.

Patterns

```
1  a::[] (*matches lists with 1 element*)
2  a::b::[] (*matches all lists with 2 elements*)
3  x (*matches anything that has a value*)
4  _ (*matches everything*)
```

Pattern Matching Warnings

Not all cases considered: inexhaustive pattern-match warning

Duplicated cases: unused match case warning

5 Let Expressions

Let definitions have been used until now

```
1  let x = 2 in x + x
```

This returns the value $x + x$ so this statement is an expression

```
1  let x = 2 in x = 1
2  (fun x -> x + 1) 2
```

These two equations are the same. Basically all these expressions are secretly functions!

Variant

```
1  type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
2  let int_of_day d =
3      match d with
4      |Sun -> 1
5      |Mon -> 2
6      |Tue -> 3
7      |Wed -> 4
8      |Thu -> 5
9      |Fri -> 6
10     |Sat -> 7
```

Each of the Sun/Mon/Tues are called constructors, which are already a value

Records
Need to define a record type for type inference

```
1  type contact = {name: string; hp: int}
2  let nick = {name="Nick"; hp=81234567}
3  nick.name (*returns "Nick"*)
4  let get_hp m =
5      match m with
6      | {name=_; hp = h} -> h
7  let get_hp m =
8      match m with
9      |{name; hp} -> hp
10 let get_hp m =
11     match m with
12     |{hp} -> hp
13 let get_hp m = m.hp
```

All the functions are the same, check if you understand how this pattern matching works!

Tuples

```
1  (1,2,10) : int*int*int
2  (true, "Hello") : bool*string
3  ([1;2;3], (0.5, 'X')) : int list * (float*char)
4  let f t =
5      match t with
6      | (x, y, z) -> z
7  let f t =
8      let (x, y, z) = t in z
9  let f t =
10     let (_, _, z) = t in z
11 let f (_, _, z) = z
12 f (1, 2, 3) = 3
```

Extended Syntax for Let
Previously x was used as a variable, but actually any pattern will work

```
1  let x = e1 in e2
2  let rec f x1 ... xn = e1 in e2
```

so replacing all the x's with patterns will work as well

```
1  let add t =
2      let (x, y, z) = t
```

```
3       in x + y + z
4  let add (x, y, z) = x + y + z
```

There are built-in functions for accessing first and second element of a tuple (not sure why you need it...)

```
1  let fst (x, _) = x
2  let snd (_, y) = y
```

Type synonyms
Can define types so you don't have to type float list list for matrix etc

```
1  type point = float * float
2  type vector = float list
3  type matrix = float list list
```

Of course, you will have to specify the type because the compiler isn't smart enough to read your mind!

```
1  type point = float * float
2  let f x : point =
3    let y = x +. x in
4    (y, y)
5    (* if : point is not written, the type of f will be
          float -> float*float instead of float -> point!*)
```

Back to variants
Recall: variants are enumerated sets of values. In particular, you can have a variant that enumerates values that of different types.

```
1  type point = float * float
2  type shape =
3      | Point of point
4      | Circle of point * float
5      | Rect of point * point
6
7  let pt1 : point = (1., 2.)
8  (*pt1 = point(1., 2.) is invalid!! *)
9  let pt2 : shape = Point(1., 2.)
10 let pt3 = Circle (pt1, 2.)
11 let pt4 = Circle ((1., 2.), 3.)
12 (*note that since Circle's type is point * float, you
      don't need to define the tuple as a point since it is
      already defined!*)
```

The data type of shape is called an **algebraic data type** because it is a **tagged union** (Google nonintersecting union if you're not sure)

This allows you to classify things under other things. This is similar to inheritance with classes in Java. We may combine this with functions to allow a function to take in arguments of various types (patterns), which is similar to overloading in Java.

```
1  let area = function
2      | Point _ -> 0.0
3      | Circle (_,r) -> pi *. (r ** 2.0)
4      | Rect ((x1,y1),(x2,y2)) ->
5      let w = x2 -. x1 in
6      let h = y2 -. y1 in
7      w *. h
```

Recursive Variants

We may implement the linked list data type with a recursive variant definition.

```
1  type intlist = Nil | Cons of int * intlist
2
3  let emp = Nil
4  let l3 = Cons (3, Nil) (* 3::[] or [3]*)
5  let l123 = Cons(1, Cons(2, l3)) (* [1;2;3] *)
6
7  let rec sum (l:intlist) =
8      match l with
9      | Nil -> 0
10     | Cons(h,t) -> h + sum t
11
12 let rec length = function
13     | Nil -> 0
14     | Cons (_,t) -> 1 + length t
15     (* length : intlist -> int *)
16
17 let empty = function
18     | Nil -> true
19     | Cons _ -> false
20     (* empty: intlist -> bool *)
```

We may also use 'a to allow our list to accept other types.

```
1  type 'a mylist = Nil | Cons of 'a * 'a mylist
2  int mylist
```

Mylist is known as a **type constructor**, because it takes a type as an input and returns a type.

6   Higher-order Programming

Recall that functions are values, so we can pass them as arguments. This is an example of functions being 'first-class citizens' lol.

```
1  let square x = x * x
2  let quad x = (square x) * (square x)
3  let twice f x = f (f x)
4  let quad2 x = twice square x
```

The slides didn't mention this, but I believe that the lack of distinction between functions and values is why braces around function arguments were not implemented.

Map and Fold

Note: map is not a method to store key - value pairs!

```
1  let rec map f = function
2      |[] -> []
3      | h::t -> (f h) :: (map f t)
4
5  let add1 = List.map (fun x -> x + 1)
6  let list1 = [1; 2; 3; 4]
7  add1 list1 (*returns [2; 3; 4; 5]*)
```

Filter

```
1  let rec filter f = function
2      |[] -> []
3      | h::t ->
4          if f h
5          then h::(filter f t)
6          else filter f t
7
8  let filter1 = List.filter (fun x -> x > 2)
9  let list1 = [1; 2; 3; 4]
10 filter1 list1 (*returns [3; 4]*)
```

These are called iterators.

Combining Elements

```
1  let rec combine init op = function
2      | [] -> init
3      | h::t -> op h (combine init op t)
4
5  let sum = combine 0 (+)
```

```
6  let concat = combine "" (^)
```

Notice once again how all these functions can be ordered around as values. With braces, the code would be more confusing than without, which is quite interesting! Notice also the use of () to convert an infix operator to a prefix operator.

OCaml has a `fold_right` / `fold_left` operator to do this explicitly.

```
1  let rec fold_right f t acc =
2      match t with
3      | [] -> acc
4      | h2::t2 -> f h2 (fold_right f t2 acc)
5
6  List.fold_right f [a;b;c] init
7  (*computes f a (f b (f c init))*)
8
9  let rec fold_left f acc t =
10     match t with
11     | [] -> acc
12     | h2::t2 -> fold_left f (f acc h2) t2
13 (*note the difference between fold_left and fold_right's
       implementation - fold_right has to reach the
      rightmost element before evaluating f but fold_left
      evaluates the left-most element straight away*)
```

7   Modular Programming

OCaml uses **functional data structures** which means that data structures never change (again, to avoid having multi-valued things)

```
1  module MyStack = struct
2    type 'a stack =
3      | Empty
4      | Entry of 'a * 'a stack
5    let empty = Empty
6    let is_empty s = s = Empty
7    let push x s = Entry (x, s)
8    let peek = function
9      | Empty -> failwith "Empty"
10     | Entry(x,_) -> x
11   let pop = function
12     | Empty -> failwith "Empty"
```

```
13        | Entry(_,s) -> s
14    end
15
16    let test1 = MyStack.empty;;
17    let test2 = MyStack.push 1 test1;;
18    let test3 = MyStack.push 2 test2;;
19
20    (*returns*)
21    val test3 : int MyStack.stack =
22      MyStack.Entry (2, MyStack.Entry (1, MyStack.Empty))
```

- Note the `module...struct` keywords; which is equivalent to classes in Java. Specifically, a module creates a new **namespace**.

- In particular, you need to define the type which is needed for type inference.

- Note the last 3 lines: it is not possible to push an element into the original stack, because the stack is not mutable. (If you tried to assign `test1 = MyStack.push 1 test1` you would get `false`! The equate sign is interpreted as checking equality.)

Function Signature
It is possible to define the function signature, i.e. what types of variables it can input and output, as well as what functions to be accessed from outside the module.
The 2 benefits are:

- input and output types are correct

- unnecessary methods are hidden

```
1    module type S1 = sig
2      val x:int
3      val y:int
4    end
5    module M1 : S1 = struct
6      let x = 42
7    end
8    (*Error: Signature mismatch:
9    Modules do not match: sig val x : int end is not
        included in S1
10   The value 'y' is required but not provided*)
```

11

```
11  module type S2 = sig
12    val x:int
13  end
14  module M2 : S2 = struct
15    let x = 42
16    let y = 7
17  end
18  M2.y
19  (*Error: Unbound value M2.y*)
```

Abstract Types

There is one more step to abstraction: not revealing the data structure in the module signature.

```
1  module type ListStackSig = sig
2    val empty : 'a list
3    val is_empty : 'a list -> bool
4    val push : 'a -> 'a list -> 'a list
5    val peek : 'a list -> 'a
6    val pop : 'a list -> 'a list
7  end
8
9  module ListStack = struct
10   let empty = []
11   let is_empty s = s = []
12   let push x s = x :: s
13   let peek = function
14     | [] -> failwith "Empty"
15     | x::_ -> x
16   let pop = function
17     | [] -> failwith "Empty"
18     | _::xs -> xs
19  end
20    (*client's code*)
21  let x = ListStack.empty;;
22  let y = ListStack.push 1 x;;
23  let z = (2 :: y);;
```

In this example, after realising that `ListStack` is implemented with list, the client decides to use the `::` function to concatenate lists. However, if you decide to change the type used in `ListStack` to something else, all of the client's code that uses `::` will break. Hence, the type used in implementation shouldn't be revealed.

```
1  module type Stack = sig
```

```
2      type 'a t
3      val empty : 'a t
4      val is_empty : 'a t -> bool
5      val push : 'a -> 'a t -> 'a t
6      val peek : 'a t -> 'a
7      val pop : 'a t -> 'a t
8    end
9
10   module ListStack : Stack = struct
11      type 'a t = 'a list (*this line informs the compiler
            that all the ts refer to lists*)
12         ...
```

Here, we give an arbitrary type `stack` so the client would not know what type we have used. By convention `t` is used to refer to an abstract type.

8   Functors

Interface Inheritance

```
1   module type Ring = sig
2      type t
3      val zero : t
4      val one : t
5      val add : t -> t -> t
6      val mult : t -> t -> t
7      val neg : t -> t
8    end
9   module type Field = sig
10     include Ring
11     val div : t -> t -> t
12   end
13
14  module FloatRing = struct
15     type t = float
16     let zero = 0.
17     let one = 1.
18     let add = (+.)
19     let mult = ( *. )
20     let neg = (~-.)
21   end
22  module FloatField = struct
23     include FloatRing
24     let div = (/.)
```

```
25  end
```

So `include` works for both module signatures and modules.

Functors
Functors take structures as inputs and output another structure. So even modules are treated like functions!

```
1  module type X =
2  sig val x : int
3  end
4
5  module IncX (M : X) = struct
6     let x = M.x + 1
7  end
8
9  (*in anonymous form as well*)
10  module IncX = functor (M : X) -> struct
11     let x = M.x + 1
12  end
13
14  module A = struct let x = 0 end
15  module B = IncX(A) (* B.x is 1 *)
16  module C = IncX(B) (* C.x is 2 *)
```

One use of functors is to test whether modules of a certain signature are working.

```
1  assert (MyStack.(empty |> push 1 |> peek) = 1)
2  (*The output of MyStack.empty is piped to MyStack.push 1
       using the pipe operator. Assert throws an exception
     if the expression evaluates to false.*)
3
4  (*instead of*)
5  assert (MyStack.(
6     empty |> push 1 |> peek) = 1) ;;
7  assert (ListStack.(
8     empty |> push 1 |> peek) = 1);;
9  assert (WhateverStack.(
10     empty |> push 1 |> peek) = 1)
11
12  (*we could write*)
13  module StackTester (S:StackSig) = struct
14     assert (S.(empty |> push 1 |> peek) = 1)
15  end
```

```
16  module MyStackTester
17    = StackTester(MyStack)
18  module ListStackTester
19    = StackTester(ListStack)
20
21  (*note that this works because when instantiating the
        StackTester module, it evaluates each expression
        within it, even if it isn't a let x = ... expression.
        *)
```

`include` works with functors as well... But to be fair it isn't surprising as a functor is just another module.

```
1  module type Sig = sig ... end
2  module Ext (M:Sig) = struct
3    include M
4    let f =
5      ...
6  end
```

9   Abstraction and Specification

You should document your code properly if you want it to be readable. The following sections have guidelines on how to write specifications to make it easy for someone else to read.

```
1  (**
2   * returns: [hd lst] is the head of [lst].
3   * example: hd [1; 2; 3] is 1.
4   * requires: [lst] is non-empty.
5   * raises: [Failure "hd"] if [lst] is empty.
6   * effects: ...
7   *)
8  val hd : 'a list -> 'a
9  (*this last line would be in your module signature*)
```

The starry formatting allows html or others to parse the comments. The square brackets will format the words as `words`.

10 Abstraction Functions and Representation Invariants

Not critical to the Cambridge OCaml course. Will add in later if I feel like it.

11 Testing

Not critical to the Cambridge OCaml course. Will add in later if I feel like it.

12 Streams and Laziness

It is possible to generate infinite lists, also known as streams. To not use up memory, the next item in the list is only generated when a function is evaluated.

```
1  type 'a stream =
2      Cons of 'a * (unit -> 'a stream)
3  (By convention, we type stream. Not that it is not a
       default type.*)
4
5  let rec from n =
6    Cons(n, fun () -> from (n + 1))
7  (*The anonymous function works, because functions are
       already values and do not get evaluated until a value
        is fed in. Hence, the following iterations of from (
       n + 1) are not evaluated.*)
8
9  let nats = from 0
10 let hd (Cons (h, _)) = h
11 let tl (Cons (_, tf)) = tf ()
12 (*To evaluate tf, we pass a unit argument, (). A
       function that takes a unit argument is called a thunk
       .*)
13
14 let rec take n s =
15   if n = 0 then []
16   else hd s :: take (n - 1) (tl s)
17
18 let rec drop n s =
```

```
19    if n = 0 then s
20    else drop (n - 1) (tl s)
21
22   (*Make sure you can figure out what take and drop does.
        *)
```

We can apply operations like arithmetic and mapping to streams as though they were lists, by using the following methods:

```
1  let rec square (Cons (h, tf)) =
2    Cons (h*h, fun () -> square (tf ()))
3
4  let rec sum (Cons (h1, tf1)) (Cons (h2, tf2)) =
5    Cons (h1+h2, fun () -> sum (tf1 ()) (tf2 ()))
6
7  let rec map f (Cons (h, tf)) =
8      Cons (f h, fun () -> map f (tf ()))
9
10 let rec nats = Cons (0, fun () -> map (fun x -> x + 1)
      nats)
11 (*This was difficult to understand; to figure this out
      imagine nats = [0; 1; 2; 3;...]. Then map (fun x -> x
      + 1) nats = [1; 2; 3; 4;...].*)
```

One issue with this is that `fun () -> ...` is not memoized, so every time it is called the value will be recalculated. OCaml has a high-level implementation to solve this problem called `Lazy`.

```
1  let x = lazy (5 + 4) ;;
2  (*val x : int lazy_t = <lazy>*)
3
4  Lazy.force x;;
5  (*int = 9*)
6
7  x;;
8  (*int lazy_t = lazy 9*)
```

On instantiating, x is of type `int lazy_t` and returns a value of `<lazy>`. We may use `Lazy.force` to evaluate x. After that, x returns a new value, `lazy 9`. This means that when x is called again, it does not do the calculation.

The lecture recommended trying to implement `lazylist` for fibonacci numbers. No solution was given so here is my solution:

```
1  type 'a lazylist =
```

```
2       Cons of 'a * 'a lazylist Lazy.t;;
3       (*Lazy.t means lazy type, which is lazy_t, so you
            could type lazy_t instead!*)
4
5   let hd (Cons (h, _)) = h;;
6
7   let tl (Cons (_, tf)) = Lazy.force tf;;
8
9   let rec take n s =
10      if n = 0 then []
11      else hd s :: take (n - 1) (tl s);;
12
13  let rec drop n s =
14      if n = 0 then s
15      else drop (n - 1) (tl s);;
16
17  (*both methods for sum should work*)
18  let rec sum list1 list2 =
19      Cons ((hd list1) + (hd list2), lazy (sum (tl list1) (
            tl list2)));;
20
21  let rec sum (Cons (h1, t1)) (Cons (h2, t2)) =
22      Cons (h1 + h2, lazy (sum (Lazy.force t1) (Lazy.force
            t2)));;
23
24  let rec fibs =
25      Cons(1, lazy (
26          Cons(1, lazy (
27              sum (tl fibs) fibs))))
28
29  let fib30lazyfast = take 30 fibs
30  let fib29lazyfast = take 29 fibs ;;
```

13   Balanced Trees

Various data structures should be able to use the following functions:

```
1   module type Set = sig
2       type 'a t
3       val empty : 'a t
4       val insert : 'a -> 'a t -> 'a t
5       val mem : 'a -> 'a t -> bool
6       (*mem stands for membership*)
```

```
7  end
8
9  module ListSet : Set = struct
10    ...
11 end
12 module BstSet : Set = struct
13    ...
14 end
15 module RbSet: Set = struct
16    ...
17 end
```

Binary Search Tree

```
1  module BstSet = struct
2  (* AF: [Leaf] represents the empty set.
3  * [Node (l, v, r)] represents
4  * the set $AF(l) \union {v} \union
5  * AF(r)$.
6  * RI: for every [Node (l, v, r)],
7  * all the values in [l] are strictly less than
8  * [v], and all the values in [r] are strictly
9  * greater than [v]. *)
10    type 'a t =
11      | Leaf
12      | Node of 'a t * 'a * 'a t
13    let rec mem x = function
14      | Leaf -> false
15      | Node (l, v, r) ->
16          if x < v then mem x l
17          else if x > v then mem x r
18          else true
19    let rec insert x = function
20      | Leaf -> Node (Leaf, x, Leaf)
21      | Node (l, v, r) ->
22          if x < v then Node(insert x l, v, r )
23          else if x > v then Node(l, v, insert x r)
24          else Node(l, x, r )
25 end
26
27 let x = BstSet.Leaf;;
28 let y = BstSet.insert 4 x |> BstSet.insert 5;;
29 (*val y : int BstSet.t =
30   BstSet.Node (BstSet.Leaf, 4, BstSet.Node (BstSet.Leaf,
```

```
5, BstSet.Leaf))*)
```

We will go through the time complexities of worst, average and best cases briefly.

Worst Case
If you insert an ascending list of numbers, the next number will always be on the right side of the previous number so you have to climb down very far - $\mathcal{O}(n)$ - to reach the leaf!

Average Case
A random set of numbers would lead to a rather 'balanced' tree. The height is $\log n$ so the time complexity is also $\mathcal{O}(\log n)$.

Best Case
If you leaf the inputs properly you will definitely be able to get a tree with height $\log n$ and time complexity of $\mathcal{O}(\log n)$.

In order to ensure that the time complexity is $\mathcal{O}(\log n)$, we may balance the tree. Common data structures that implement these are:

- 2-3 Trees: all paths have the same length

- AVL Trees: length of shortest and longest path from any node differ by at most one

- Red-Black Trees: length of shortest and longest path from any node differ at most by factor of two

Red-Black Trees

- Binary search tree with each node coloured red or black (1 bit)

- Conventions:

  - Root is always black
  - Empty leaves are black

- Invariants:

  - No red node has a red child
  - Every path from root to leaf has the same number of black nodes

- Thus the length of the longest path is at most twice the length of the shortest path (interleave red nodes between black nodes)

Implementation

```
1  module RbSet = struct
2    type color = Red | Blk
3    type 'a t =
4      | Leaf
5      | Node of (color * 'a t * 'a * 'a t)
6    let empty = Leaf
7    let rec mem x = function
8      | Leaf -> false
9      | Node (_, l, v, r) ->
10        if x < v then mem x l
11        else if x > v then mem x r
12        else true
13   let rec insert' x = function
14     | Leaf -> Node(Red, Leaf, x, Leaf)
15     | Node (col, l, v, r) ->
16        if x < v
17        then balance (col, (insert' x l), v, r)
18        else if x > v
19        then balance (col, l, v, (insert' x r))
20        else Node (col, l, x, r)
21   let insert x s =
22     match insert' x s with
23     | Node (_, l, v, r) -> Node(Blk, l, v, r)
24     | Leaf -> failwith "impossible"
25   let balance = function
26     | (Blk, Node (Red, Node (Red, a, x, b), y, c), z, d)
27     | (Blk, Node (Red, a, x, Node (Red, b, y, c)), z, d)
28     | (Blk, a, x, Node (Red, Node (Red, b, y, c), z, d))
29     | (Blk, a, x, Node (Red, b, y, Node (Red, c, z, d)))
30       -> Node (Red, Node (Blk, a, x, b), y, Node (Blk, c
            , z, d))
31     | t -> Node t
32 (*pattern matching makes the reassignments so easy!*)
```

2-3 Trees
Two types of nodes:

- 1 parent 2 children: obeys BST

- 2 parents 3 children (p1 p2 c1 c2 c3):

    - c1 < p1
    - p1 <= c2 <= p2

– p2 < c3

14   Mutable Data Types

OCaml variables are immutable, but its references, fields and arrays are mutable.

```
1  let x = ref 0;;
2  val x : int ref = {contents = 0}
3  (*The type of ref 0 is int ref*)
4
5  x;;
6  - : int ref = {contents = 0}
7  !x;;
8  - : int = 0
9  x:=2;;
10  - : unit = ()
11  x;;
12  - : int ref = {contents = 2}
13  !x;;
14  - : int = 2
```

The binding of x to the pointer is immutable, but the contents of the memory may change. This is useful in implementing a counter.

```
1  let next_val =
2    let counter = ref 0
3    in fun () ->
4      incr counter;
5      !counter
```

References may have aliases, i.e. another variable that points to the same location in memory. So if the value stored in that location changes, both variables when dereferenced will point to the new value.

```
1  let r1 = ref 3110;;
2  let r2 = ref 3110;;
3  r1 == r2;; (*false*)
4  r1 = r2;; (*true*)
```

== tests for physical equality, i.e. if the pointers point to the same memory location. = tests for structural equality, i.e. if the value they point to is the same. Most of the time you only need to compare values with structural inequality.

Mutable Fields

```
1  # type point = {x:int; y:int; mutable c:string};;
2  type point = {x:int; y:int; mutable c:string; }
3  # let p = {x=0; y=0; c="red"};;
4  val p : point = {x=0; y=0; c="red"}
5  # p.c <- "white";;
6  - : unit = ()
7  # p;;
8  val p : point = {x=0; y=0; c="white"}
9  # p.x <- 3;;
```

With this, we realise that refs are just implementations of mutables.

```
1  type 'a ref = { mutable contents: 'a }
2  let ref x = { contents = x }
3  let ( ! ) r = r.contents
4  let ( := ) r newval = r.contents <- newval
```

Arrays

Quick note: arrays are not lists!

```
1  # let v = [|0.; 1.|];;
2  val v : float array = [|0.; 1.|]
3  # v.(0) <- 5.;;
4  - : unit = ()
5  # v;;
6  - : float array = [|5.; 1.|]
7
8  while e1 do e2 done
9  for x=e1 to e2 do e3 done
10 for x=e1 downto e2 do e3 done
11 (*loops are back!*)
```

15   Interpreters

Compilers compile the source into the target program, before the target program can take in inputs and produce outputs. Interpreters can take in both the source program and inputs while producing outputs.

This leads to possibly better error messages and better debuggers.

Mixed Compilation and Interpretation

Source program compiles into an intermediate program. Then a virtual machine can take in the intermediate program and input while producing outputs.

Architecture

- Front end: translate source code into abstract syntax trees (AST)

- Back end: translate AST into machine code

Front end involves:

- Lexical analysis with lexer

- Syntactic analysis with parser

- Semantic analysis

Functional languages are well-suited to implement compilers and interpreters

- Recursive functions can be converted into ASTs

- Recursive functions can be used to interpret ASTs

To interpret the following AST `Add (Int 5) (Add (Int 6) (Int 7))` We can use the following code:

```
type expr =
  | Int of int
  | Add of expr * expr

let rec eval e =
  if is_value e
  then e
  else eval (step e)

let is_value = function
  | Int _ -> true
  | Add _ -> false

let rec step = function
  | Int n -> failwith "Does not step"
  | Add(Int n1, Int n2) -> Int (n1+n2)
  | Add(Int n1, e2) -> Add(Int n1, step e2)
  | Add(e1, e2) -> Add(step e1, e2)
```

Another example: `let x = 5 in x + 1`, which is equivalent to
`Let "x" (Int 5) (Add (Var "x") (Int 1))`

```ocaml
type expr =
  | Int of int
  | Add of expr * expr
  | Var of string
  | Let of string * expr * expr

let rec eval e =
  if is_value e
  then e
  else eval (step e)
let is_value = function
  | Int _ -> true
  | Add _ | Var _ | Let _ -> false

  (* [subst e v x] is e{v/x}, that is,
 * [e] with [v] substituted for [x]. *)
let rec subst e v x = match e with
  | Var y -> if x=y then v else e
  | Int n -> Int n
  | Add(el,er) ->
      Add(subst el v x, subst er v x)
  | Let(y,ebind,ebody) ->
      let ebind'
        = subst ebind v x in
      if x=y
      then Let(y, ebind', ebody)
      else Let(y, ebind', subst ebody v x)

let rec step = function
  | Int n -> failwith "Does not step"
  | Add(Int n1, Int n2) -> Int (n1 + n2)
  | Add(Int n1, e2) -> Add (Int n1, step e2)
  | Add(e1, e2) -> Add (step e1, e2)
  | Var _ -> failwith "Unbound variable"
  | Let(x, Int n, e2) -> subst e2 (Int n) x
  | Let(x, e1, e2) -> Let (x, step e1, e2)
```

After all this work we can finally interpret 2 syntax trees...

Btw the following sections are beyond Cambridge's OCaml syllabus so I will
not make notes for them yet (This is a 3rd/4th year Cornell course!)

16   The Substitution Model

17   The Environment Model

18   Futures

19   Monads