Lecture 1 3/28/2016

- Learn concepts underlying programming languages
- Study three major programming languages
  - functional (ML)
  - object oriented (Java)
  - logic/declarative (Prolog)
- plus a bit of
  - parallel programming
  - scripting

C and C++ are based on out of date assumptions
- memory is scarce
- computers are slow
- programs live in a trusted environment

These languages are <u>low-level</u> and <u>unsafe</u>

Today, the opposite is true... we want <u>high-level</u> and <u>safe</u> languages.
- provide powerful abstractions that hide low-level details
- ensure safety properties for ALL programs (even buggy ones)

Programs tend to be simpler and easier to understand!

<u>Languages Today</u>

All languages except C and C++ are memory safe.
- pointers not exposed by language
- no dangling pointers
- no way to get an undefined states

Other languages might provide high-level abstractions
- functional programming: no mutation (hence no race conditions!)

Download ocaml from ocaml.org
SEAS machines : /usr/local/cs/bin/ocaml

Ocaml is a dialect of the ML functional programming language

```
# 3+5;;
  -: int = 8
```
// Looks like we've got a fancy calculator so far.

Type inference: compiler figures out the types at compile time

```
# let x = 3+5;;
 val x: int = 8
# x + 34;;
 -:int =42
# let y = x * x;;
 val y : int = 64
# let double = (function x -> x * 2);;
val double : int -> int = <fun>
# let quadruple = (function v -> (double v) + (double v));;
 val quadruple : int -> int = <fun>
 # quadruple 23;;
 -: int = 92
```

Whatif....

```
# let double : ~~int -> int = <fun>~~ (function x -> x * 3);;
 val double : int -> int = <fun>
# quadruple 23;;
 -: int = 92 ← still works as intended, even though we made a new double!
# double 4
 -:int = 12 ← note that now the old value/function for double is now out of scope
```

Note: we must tell ocaml when we are about to use recursion

```
# let rec factorial =
    function n ->
        if n = 0 then 1 else n * factorial (n-1);;
val factorial : int -> int = <fun>
```

There's actually a better way to do this, called <u>pattern matching</u>.

```
# let rec fact =
    function n ->
      match n with
          0 -> 1
        | _ -> n * fact (n-1);;
val fact : int -> int = <fun>
```

Think of pattern matching as a switch statement in C or C++. We will see later that matching is more powerful.

```
# let rec isEven = (function n -> .....)
```

← function that tells if a function is even without doing a %

Only needs to work for + numbers

Answer:

```
# let rec isEven = (function n ->
                      match n with
                          0 -> true
                        | 1 -> false
                        | _ -> isEven (n-2));;
```

↑ to solve negatives as well, replace with

```
        | _ -> if n>0 then isEven(n-2) else isEven(n+2));;
```

Lists

```
# [1, 2, 3]
 - : int list = [1; 2; 3]
# 1::(2::(3::[]));;
 - : int list = [1; 2; 3]
```

note that we didn't explicitly use the "function" keyword. This is syntactic shorthand.

```
#let rec sumLst lst =
  match lst with
    [] -> 0
  | h::t -> h + sumLst t
```

← h goes to head, t goes to tail

Aside: pattern for a list of at least 2 elements:

```
h1::(h2::t)
```

```
# let rec everyOther =
  function lst ->
    [] -> 0
  | h1::(h2::t) -> h2 + everyOther(t)
  | _ -> 0
```

//sums every other list

```
# let rec everyOther =
  function lst ->
    [] -> [redacted][]
  | h1::(h2::t) -> h2::everyOther(t)
  | _ -> [redacted][]
```

// returns a list of every other item

or

```
  [] -> []
| [h] -> [h]
| h1::(h2::t) -> h1::everyother(t)
```

Lecture 2 3/30/2016

How do we build programs in a functional programming style? Answer: lots and lots of little data structures.

Recall the quadruple implementation from Lecture 1 for a moment

We will rewrite it in let X = E1 in E2 in the following way.

```
let d = double n
    in d+d;;
```

In this way, if we include it in a function we have effectively created a local variable

Notice that what is nice about this language is that it is very general. We can have multiple local variables, lists of lists, even lists of functions!

What if we want a function to take multiple parameters?
We can use tuples.
Note that we cannot have tuples of size 1 but can have one of size 0 (this is called the unit tuple).

```
# (1, "hi", true)
- : int * string * bool = (1, "hi", true)

#let add = (function (x,y) -> x+y);;
   val add : int * int -> int = <fun>
# add (3,4);;
- : int = 7
# let ~~add =~~ p = (3,4);;
  val p .....
# add (p);;
- : int = 7
```

Chaining local variable declarations

```
let x = 3 in
  let x = 4 in
    x+y;;
- : int = 7
```

Note: we will never use void, because the purpose of having void is to mutate a function w/o returning anything.
In an immutable langage, there is no point in having these!!!

Example problem

Assume that input lists are the same size.
Write a function zip such that

zip ([1;2;3], [4;5;6]) returns [(1,4);(2,5);(3,6)]

```
let rec zip =
  (function (l1, l2) →
      match (l1, l2) with
        ([],[]) → []
      | (h1::t1, h2::t2) → (h1, h2)::zip(t1,t2)
  )
```

Example problem

Now write unzip, such that it takes a result and then reverses the process.

```
let rec unzip
  function lst →
    match lst with
      [] → ([],[])
    | (h1,h2)::t → ~~(h1,h2)::unzip t~~
                   match unzip t with
                   (l1,l2) → (h1::l1, h2::l2)
```

we aren't building a list of tuples

or alternatively

```
let (l1,l2) = unzip t in
  (h1::l1, h2::l2)
```

or f(f x)

↓

```
# let twice = function (f,x) -> f(f(x));;
  val twice : ('a -> 'a) * 'a -> 'a = <fun>
# twice (double, 3) ;;
 -: int = 12
```

```
# let compose =
    function (f,g,x) -> g(f(x));;
val compose : ('a -> 'b) * ('b -> 'c) * 'a -> 'c = <fun>

# compose (double, (function x -> x * x), 3);;
  - : int = 36
```

We can also have functions return other functions!!!

~~# let returnsAdd : unit -> int * int -> int = <fun>~~

```
# let returnsAdd() =
    function (x,y) = x+y
 which is shorthand for
# let returnsAdd
    function () ->
      function (x,y) = x+y

# let add =
    function x ->
      (function y -> x+y);;
# add 3 4
-: int = 7
```

Benefit? I can invoke part of the overall thing and then use this later on!!

```
# let addToThree = add 3;;
```

8.

```
# let rec incList =
    function l
      match l with
        [] -> []
    | h::t -> (h+1)::(incList t);;
val incList : int list -> int list = <fun>
# incList [1;2;3];;
- : int list = [2;3;4]

# let rec gt2 =
    function l ->
      match l with
        [] -> []
    | h::t -> (h>2)::(gt2 t);;
.....
# gt2 [1;2;3];;
  - : bool list [false; false; true]
```

Cooler way...

```
# list.map (function x -> x+1) [1;2;3];;
 - : int list = [2;3;4]
# list.map (function x -> x>2) [1;2;3];;
 - : boo list  [false; false; true]
# let incList = list.map (function x -> x+1);;
 .....
```

Discussion 1 4/1/2016

See the slides he uploads for Discussion notes. He moves too quickly to handwrite the code.

Functional Programming Paradigm

- Computation as the evaluation of mathematical functions.
- Avoids changing-state and mutable data

Recall the Tower of Hanoi problem and how it can be solved recursively.

Another example is MergeSort, recall this from CS180: Divide and Conquer

Yet another example: Map Reduce

Side Effects

- Modifies some state or has an observable interaction with calling functions.

T/F: Do these actions have side effects?

- Changing a global variable T
- Allocating an array on the stack (assuming no overflow) F
- Throwing an exception depends on the situation (is there a handler?)
- Modifying one of its arguments T
- Print "hello world" to the terminal T
- Calling another function T

Referential Transparency

- When a function can be replaced by the value it would have produced without changing the behavior of a program

1) No side effects.
2) The function is pure (always returns the same result on the same input)

Race Conditions

- Occurs with parallelism and is very error prone.
- Happens when modifying the same memory with different processes.
  - needs lots of safety protocols
  - lots of threads sit idle at the performance bottleneck.

How to Load Source Code in Ocaml

```
# use "filename.ml"
```

```
$ ocaml filename.ml
```

```
$ ocaml < filename.ml
```

Builtin Data Structures

int	1
string	"1"
float	1.
bool	true, false
list	[a; b]
pair	(a, b)

List

- An immutable, finite sequence of elements of the same type.

```
[1; 2; 3];;        1::(2::(3::[]));;
```

- Use list.append [list 1] [list 2] to append list 2 to list 1
  This is also represented by the @ symbol

```
[1; 2; 3] @ [4; 5; 6]
```

Pair

- As it says, a pair of values
  - How to access the first value:
    - `fst(1,2)`
  - How to access the second value;
    - `snd(1,2)`
  - Can we extract the first value via pattern matching?
    - `let my-first (a, b) = a;;`
  - The types can be different

Functions

- `let add = fun x y -> x+y;;`
- `let add = fun x -> fun y -> x+y;;`
- `let add = function x -> function y -> x+y;;` Note* "function" keyword only allowed to have one argument
- `let add x = fun y -> x+y;;`
- `let add x = function y -> x+y;;`
- `let add x y = x + y;;`
- `let add (x,y) = x+y;;`
- `let add = function | (x,y) -> x+y;;`

Conditions and Pattern Matching

```
let rec factorial =
  function n ->
    if n = 0 then 1 else n * factorial(n-1);;
```

```
let rec factorial =
  function n ->
    match n with
      0 -> 1
    | _ -> n * factorial (n-1);;
```

```
let eval_op op v1 v2 =
  match op with
    "+" -> v1+v2
  | "-" -> v1-v2
  | "/" -> v1/v2
  | "*" -> v1*v2 ;;
```

Exploiting Pattern Matching: Iteration

- Iteration over a list

```
let rec sumOf lst = match lst with
    |h::t -> h+ sumOf t
    |[] -> 0 ;;
```

Tail Recursion

- `let rec make_list n = if n = 0 then [] else n:: make_list (n-1);;`
  `- make_list 1000000`
- `let rec make_list n list =`
  `if n = 0 then list else make_list (n-1) (n::list);;`

What are the differences?

Lecture 3 4/4/16

Guest lecture! :)

First class function - first class functions can be treated just like regular data by the programming language!
Higher order functions - functions that return other functions

Ex:
Given a list in JavaScript ... , print its elements to console!

```
> var things = [1, 2, 3, 4];
> for (var i = 0; i < things.length; i++)
  {
      console.log (things[i]);
  }
1
2
3
4
> things.map ((x) => console.log(x));
1
2
3
4
```

Ex: Given a pair of elements, add 1 to the first component of the pair

```
# let incFirst = fun p ->
    match p with
      (x,y) -> (x+1, y) ;;
#incFirst (1, "hello");;
-: int * string = (2, "hello")
```

Concatenation for strings

```
# "hello" ^ "!" ;;
- : string = "hello!"
```

ex:

```
#let exclaimFirst = fun p →
      match p with
        (x,y) → (x ^ "!", y);;
# exclaimFirst ("hello", 5);;
- : string * int = ("hello!", 5)
```

Note that this is annoyingly similar to the previous example!
We will resolve this using higher order functions

ex: write a function applyToFirst:
input: function, and a pair
output: apply the function to the first component

```
# let applyToFirst = fun f p →
    match p with
      (a,b) → (f a, b);;

# applyToFirst (fun x → x+1) (1, "hello");;
- : int * string = (2, "hello")
# applyToFirst (fun x → x ^ "!") ("hello", 5);;
- : string * int = ("hello!", 5)

# applyToFirst (String.length);;
- : string * 'a → int * 'a = <fun>
# applyToFirst (String.length) ("hello", 0);;
- : int * int = (5, 0)
```

```
# applyToFirst (+) (1,2);;
- : (int->int)*int = (<fun>, 2)
```

ex: How can we implement incList, which adds 1 to each element of a list? Well, we can actually do even better and implement a more generic version!

```
# let rec applyToList f l =
    match l with
      [] -> []
    | h::t -> (f h)::(applyToList f t);;
```

This is identically the map function that we have heard so much about. ☺

How do we increment every element in the list of lists?

ex:

```
# map (map (fun x -> x+1)) [[1;2;3];[4;5;6];[7;8;9]];;
- : int list list = [[2;3;4];[5;6;7];[8;9;10]]
```

ex: Take a list of integers and remove the negative elements from it

```
# let rec removeNegatives = fun l ->
    match l with
      h::t -> if h<0 then removeNegatives t else h::removeNegatives t ;;
```

or alternatively

```
# let rec removeNegatives = fun l ->
    match l with
      [] -> []
    | h::t ->
      let t' = removeNegatives t in
      if h<0 then t' else h::t'
;;
```

ex: removeEmpties, which removes empty strings from the list of strings
This will also take a very similar form to the previous example!!

ex: Let's write the general form of both of these functions

```
# let rec filter = fun f l ->
    match l with
      [] -> []
    | h::t ->
      let t' = filter f t in
      if f h then h::t' else t';;
# filter (fun x -> x >= 0) [-1;2;3;-4];;
- : int list [2; 3]
```

ex: write combineInts

input: function from int -> int
list of ints
output
integer

```
#let rec combineInts = fun (f: int -> int -> int) (l: int list) ->
    match l with
      [a] -> a
    | h::t -> f h (combineInts f t);;
```

Interesting tidbit: (* begins a comment, *) ends a comment

```
#combi
#combineInts (-) [1; 2; 3];;
- : int = 2
```

Lecture 4 4/6/2016

Guest lecturer again! ☺

Recall - a higher order function can take functions as parameters and return other functions.

examples:

- applyToFirst : applies an input function to the first component of an input pair.
- map: applies a function to every element of a list
- filter: takes a predicate and trims a list based on that predicate
- combineIntsLeft - aggregate list elements with a left-associative binary operation.

Question: can we implement length using combineInts? If yes, how? If no, why not?
No! The base case returns the value stored in the single list, so we would need to modify this case to make it compatible!!

Here's a working version...

```
let length l = sumList (List.map (fun _ -> 1) l)
```

How can we make sumList, prodList work for empty list?

- sumList [] = 0
- prodList [] = 1

How can we update combineInts to reflect this behavior? We can add a parameter that tells us what to do!

```
let rec combineInts =
  fun (f: int -> int -> int) (l : int list) (ifNil : int) ->
    match l with
      [] -> ifNil
      ~~| [x] -> x~~ ← we can remove this, and now length can be found! ☺
      | hd::tl -> f hd (combineInts f tl ifNil)
;;
```

To use combineInts to get length:

```
let length l = combineInts (fun _ lengthOfTail -> 1+ lengthOfTail)
  0;;
```

What if we write out the specialized version?

```
let rec length =
  fun (l : int list) ->
    match l with
      [] -> 0
    | hd::tl -> 1 + (length tl) ;;
```

So in fact, this combineInts function is called reduce, fold, or in ocaml, (fold_right)

Exercise:

- define fold_left
- extend combineIntsLeft to work with []
- type should be polymorphic
- should be able to define length (and length should work on any list).

Which of map, filter, and fold_right is most powerful?

Well, it turns out we can use fold_right to define map and filter.

```
let map f l = List.fold_right (fun hd map_f_tl -> f hd :: map_f_tl) l []
```

Exercise:

- define filter using List.fold_right.

Data Types

User defined data types give us a way to define our own abstractions: a type with associated operations.

- managing complexity of programs
  - decomposing
- preserving invariants of programs

Aside: -. = unary float subtraction
+. = float addition

What kind of user-defined data types are supported in other languages?

- class/object
- struct
- union, enum

Ocaml has these kinds of things, but they are specialized for:

- immutability
-

Enum in ocaml:

```
type sign = Pos | Neg | Zero ;;
```

Note *: type names must start with a lower case letter

- Pos/Neg/Zero are called the constructors of sign
- Constructor names must begin with an upper case letter.
- Two types cannot share the same constructor name.

Now that we have this, what can we do?

- Check for equality
- Pattern matching

A type with some fields/data, like a struct in C

```
type point = Point of (float * float);;
```

Syntax: <constructor_name> of <type>

We can do things like

```
let negate p =
  match p with
    Point (x,y) -> Point (-.x, -.y)
;;
```

or even

```
let negate (Point (x,y)) = Point (-.x, -.y);;
```

What if we do...

```
type point2 = float * float;;
```

?

This is merely an alias for (float * float), like a typedef in C

```
type nullableInt = Null | NonNull of int;;
```

This combines the previous two concepts. Note that Ocaml does not have NULL and remember that this is a good thing.

```
let incNullableInt x =
  match x with
    Null -> Null
  | NonNull i -> NonNull (i+1)
```

```
type 'a nullable = Null | NonNull of 'a;;
```

```
let updNullable f n =
  match n with
    Null -> Null
  | NonNull x -> NonNull (f x)
;;
```

A nullable is often used in functional programming

- in ocaml:
  - "nullable" is called option
  - "Null" is called None
  - "NonNull" is called Some

define a function get that returns the nth element of a list.

```
let rec get n l =
    match n with
      0 -> {match l with
              [] -> None
            | h::t -> h }
    | _ -> {
             match l with
               [] -> None
             | h::t -> get (n-1) t
           }
;;
```

We can clean this up by matching on both at the same time

```
let rec get n l =
  match (n, l) with
    (0, h::_) -> h      Some: this is important for the type checker
  | (_, h::t) -> get (n-1) t
  | (_, []) -> None
;;
```