8 = toi :-

- Learn concepts underlying programming languages?
- Study three major programming languages 1 330 2000 2432
  - · functional (ML)
  - doject oriented (Java) 2 M A 20 100/010 1 1 1000
    - · logic declarative (Prolog)
- plus a bit of

  porallel 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C and C++ are based on out of date assumptions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 high tevel and safe 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!

Languages Today

All languages except C and C++ are memory safe.

- pointers not exposed by language
- no dangling pointers south = to 6 to 1 slouds lay
- no way to get an undefined states 185 significant #
- Other languages might provide high-level abstractions functional programming: no mutation (hence no race conditions!)

```
2
```

Download ocaml from ocaml.org SEAS machines: /usr/local/cs/bin/ocam/ Ocaml is a dialect of the ML functional programming language # 3+5;; //Looks like we've got a fancy calculator so far. -: int = 8 Type inference: compiler figures out the types at compile time טומל לבור מורל למבומל מיו טוני פיף למוצב מבשנות מחב # let x = 3+5;; val x: int = 8 # x + 34;; thomatonic nativity a pi 2vi smorpore -: 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.... And one manay soft system system whatif... # let double : int > int = < fun> (Function x > x \* 3); val double: int > int = <fun> # quadruple 23; no way to get on underired states -: int = 92 - still works as intended, even though we made a new dable! # double 4 -: int = 12 - note that now the old valve function for double is now out of scope Note: we must tell ocam when we are about to use recursion

# let ret factorial =

function  $n \rightarrow i$ if n = 0 then 1 else n + factorial (n-1);

val factorial: int  $\rightarrow int = \langle fun \rangle$ 

There's actually a better way to do this, called pattern matching.

# let rec fact =

function n →

match n with

0 → 1

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 is Even = (function n -> ....) < function that tells if a function is even without doing a %

Answer:

Only needs to work for + numbers

# let rec is Even = (function n >

match n with

1 1  $\Rightarrow$  false 1 -  $\Rightarrow$  is Even (n-2);

to solve negatives as well, replace with 1 - 3 if n > 0 then is Even (n-2) else is Even (n+2);

```
Lists of positioning of help one on north hope list town on until
                  # [1, 2, 3]
                     -: int list = [1; 2; 3]
                 # 1:: (2:: (3::[])) i jort + a sale 1 and 0 = a 9
                     -: int list = [1; 2; 3]
             # let rec sum Let let = note that we didn't explicitly use the "function"

match let ::
                        match 1st 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)
                                                                                                + n + fact (n-1) is
                                                                                                                      South = to + to 1 tool low
              # let rec everyOther =
       function 1st >
                                                                         powstate abdius a se paidatem metrog to soul
                                  [] > 0
                                                                                  Such thousand the state of the 
               1 h1:: (h2::t) → h2 + every Other (t) //sums every other list
notation of state + O religion of a contract of a postonit ) = persist tol #
# let rec everyOther =
                          function 1st >
                                [] > #[]
                           | h1::(h2::t) > h2:: everyOther(t) // returns a list of every other item
                            1 - > #[]
                    [] > [] Mac 20 2 sistopar whee of
            [[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 I 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's 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 I but can have one of size O (this is called the unit tuple).

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

Chaining local variable
let x = 3 in declarations
let x = 4 in
x + y;;

# let add = (function (xy) > x+y);; -: Int = 7

val add: int \* int = < fun>

# add (3,4);

-: int=7

# let add = p = (3,4);;

# add (p);;

-: int = 7

Note: we will hover use void, because the purpose of having void is to mutate a function who returning anything.

In an immutable language, there is no point in having these!!!

3/30/20/6

Example problem

Assume that input lists are the same size. Write a function zip such that were state solution and and

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

(function (11, 12) -> match (11, 12) with  $([],[]) \rightarrow []$   $|(h1::t1, h2::t2) \rightarrow (h1, h2)::zip(t1, t2)$ 

Example problem is to start resource problem significant and and start Now write unzip, such that it takes a result and then reverses the process.

house that what nice about this language is that still very opener

let rec unzip O ser function 1st + me will be see to say the seed to make the match 1st with

we aren't building a list of tuples

 $[] \rightarrow ([],[])$ 1 (h1, h2)::t > (h1, h2):: unzipt match unzip t with  $(11,12) \rightarrow (h1::1,h2::12)$ 

and a partie propose of bound width to mutate

or atternatively let (11,12) = unzip t in (+8) + to (+11:11, h2:12)

```
or t(t x)
```

# let twice = function  $(f_{j}x) \rightarrow f(f(x))$ ;

val twice:  $(a \rightarrow a) * a \rightarrow a = \langle fun \rangle$ # twice (double, 3);;

# let compose =

function  $(f, g, x) \Rightarrow g(f(x));$ val compose :  $('a \Rightarrow 'b) * ('b \Rightarrow 'c) * 'a \Rightarrow 'c = < fun>$ 

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

We can also have functions return other functions!!!

# let returns Add() =

function (x, y) = x+y

which is shorth and for

# let returns Add

function () > Exty

# let odd = (function x > (Xxx x no Nonif) qom deil = rellant tol tol (function y > x+y);

#add 3 4

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

# let add to Three = add 3ij

```
8
```

```
# letrec inclist =
   function 1 = function (x, x) + x (x, x) motion = south to #
     match I with some of the land of the contraction of
    [] → []
| h::t → (h+1):: (incList +);;
val inclist : int list -> int list = < fun>
#inclist [1;2;3];
-: int list = [2;3;+] ((0)7)p = (4,0) actioning
          Val compose : (10 + b) * (5' e) 6) * (2 + b) : 2300000 /00
# let rec qt2=
                 (E ( ( x x x + x no incom a) altob) secures #
    function 1 >
      match I with
       [] \leftarrow []
     | bi: t > (b)2): (gt 2 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]
                       part (p. s) no its mil
# list.map (function x > x>2) [1;2;3]ii
 -: boo list [false; false; true]
# let inclist = list, map (function x > x+1);
                            (PLYXEY DOBOWY)
```

Somethic I can invoke port of the overall thing and then use this later

is boo = small to bo told

Discussion 1

4/1/2016

anothlong sot

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

Functional Programming Paradigm of the some of

- · 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 Merge Sort, recall this from CS180: Divide and Conquer

Yet another example: Mop 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 to stopes while addressed of

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.

Discossion I

- needs lots of sofety protocols
- lots of threads sit idle at the performance bottleneck

Compation as the evaluation of millionianian touchous

Yet another example: Map Reduce

. Mouther name state or how an deserval

TIFE DO THESE OCHIONS have side effects

· Allocating on arrive on the stock (lasers

Elemono et la sia partitoli "

Mineral on exception depend

I limingt and of blow offer many .

# How to Lood Sance Code in Ocam! # use "Filename.m!"

Socant filename.ml

\$ ocam) < filename.ml

## Builtin Data Structures

int	mine 1 n sh
string	as"
float	1.
bool	true, false
hst	[aj b]
pair	(a,b)

#### List

arpitalia -

Salbrar 1

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

· Use list append [list 1] [list 2] to append list 2 to list 1
This is also represented by the 2 symbol

#### Pair

- · Asit says, a pair of values
  - · How to access the first value:
    - -fst(1,2)
  - . How to occess the second value; - snd(1,2)
  - · Can we extract the first value via pattern matching. - let my-first (a, b) == a;;

Ist = moticly list with

· The types can be different

## **Functions**

- · let add = fun x y > x+yij
- · let add = funx > funy > x+y ij
- · let add = function x + function y → x+y; Note\* "function keyword · let odd x = fun y > x+y; only allowed to have one
- · let add x = function y > x+yjj argument

= Sy Iv go go = love tol

- · let add xy = x+yij
- · let add (x, y) = x+y;
- · let add = function | (x, x) > x+xii

# 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 h with -> n \* factorial (n-1);; let eval - op op v1 v2 =

match op with

"+"  $\rightarrow$  v1+v2

""  $\rightarrow$  v1-v2

""  $\rightarrow$  v1/v2

""  $\rightarrow$  v1+v2;

# Exploiting Pattern Matching: Iteration

• Iteration over a list

let rec sumOf lst = match lst with

|h::t → h + sumOf t

|[] → 0 ij

### Tail Recursion

· let rec make list n=ifn = 0 then [] else n: make-list (n-1);;
- make-list 1000000

(1-n) length of the site of most 0 = 17 71

· Con we calcust the first volue via pottem well

TYTE KNOW - & pro tol .

ii (1-a) Labration + a +

· let rec make\_list n list =

if n = 0 then list else make\_list (n-1) (n::list);

What are the differences?

brand of miland?" \* atold in year of aniland the milande is how to !

```
4/4/16
Lecture 3
                                          Concentration for things
 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 Java-Script ..., print its dements to conside to make
 > var things = [1, 2, 3, 4];
                             (2 level) = mi * points:
 > for (var i = 0; i < things.lengthi, i+t)
                      2004-10 replace printer printer side side sides
    console.log (things[i]);
                           1871 folylogo notady a yrine [ Ka
             thenology tent soll of noticent and HOAD I region
          The apply to that a tent of plago tel #
> things. in ap ((x) => console. log(x));
                                 I) (100 x x and) to real flogs #
                            The second of the second to the second
                        3 "ollar") ("1" x & x n(t) tolle pigno#
IEX: Given a pair of elements, add I to the first component of the pair
```

```
14
```

```
E exctose.
4/4/26
   Concatenation for strings
    -: string = "hello!"
   still tay between so me zoobenit zaolo tant - netrout
 +let exclaim First = fun p > nutar more another 2000 1000 1000
        match p with
 (x,y) \Rightarrow (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 finctions
                                       (Li Japaini) polisiono
  lex! write a function apply To First:
         input: function, and a pair
         output: apply the function to the first component
   # let apply To First = fun fp >
        match p with
            (a,b) > (fa,b) ii
  # apply To First (fun x + x+1) (1, "hello");;
    -: 10+ * string = (2, "hello") $
  #applyToFirst (fun x > x ^ "!") ("Hello", 5);;
  The Course of Stempers and the first company to make the
    -: string + int = ("hello!", 5)
                                            to not = tentopl tol #
  #apply To First (String, length);
    -: String * -a > int * -a = < fun>
  # apply To First (String length) ("hello", 0); july " -: int * int = (5,0)
```

```
#apply To First (+) (1,2);; and some some sold we sign I seemed to the sold with sold the sol
```

of a 1 st? well, we can actually do even better and implement a more generic version!

# let rec jose List f l = gmatch | with  $[] \rightarrow []$ 

| h:: t → (f h):: (apply To List f t) ii

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

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

# 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]]

Texil Take a list of integers and remove the negative clement from it

# let rec removeNegatives = fun 1 >

match I with

| h:: t > if h<0 then remove heaptives t else h: remove hegatives til

or atternatively the ( \* remains a seigned to) still by point and tall

# let rec remove Negatives = fun 1>

1 b :: t >

let t' = remover/regatives t in

if h<0 then t' else hitt'

11

-: 10t = 2

```
[ex:] remare Empties, which removes empty strings from the list of strings
        This will also take a very similar form to the previous cample!
     Icail Let's write the general form of both of these functions
    # let rec fitter = fun f / > du talloci monagan ou no wold lie
     som match I with an verted more ab elloung poo our let w Stall a so
            | h:: + >
         if fhthen h::t' clse t';;
     # filter (fun x > x>= 0) [-1:2:3:-4];
     -: jot list [2:3]
     this is identically the map throtten most use have broad so much about
    Ex: write combine Ints
          input: function from but - int a manual man an analy more
          list of ints
           $ map (map ( Fin x > x + 1 )) [[112:3]; [4:5:6]; [7 trains
            integer [ 1 2] [[ 1 2 2] . [ 4 : 8 : 6 ]] = 724 734 770. 1-
     #let rec combine Ints = fin (f: int > int) (1: int list) >
          [a] > a
hit > f h (combine Ints ft) ii
      Interesting tidbit: (* begins a comment, *) ends a comment
                                    Last remarkly anner 227 tol
     Homost
     # combine Ints (-) [1:2;3];;
```

It had still dell billy

Lecture +

4/6/2016

Guest lecturer again!

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

#### examples

- apply to First: applies an input function to the first component of an input poir.

To vice combine Inte to get length!

- map: applies a function to every element of a list
- -filter: takes a predicate and thims a list based on that predicate
- -combine Ints Left aggregate list elements with a left -associative binary operation.

Obestion: can we implement length using combine Ints? 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 ( = sumList (List. map ( fun \_ > 1) )

How can we make sumlist, prod List work for empty list?

- sumList [] = 0

- produst of ]= 1 tobs blot boo yet the gom to with

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

let rec combine Ints =

fun (f: int > int > int) (1: int list) (#Wil: int) >

match | with

1:N71 + []

| [x] > x = we can remove this, and now length can be found! in hall the first of hall (combine Ints f tl if Nil)

4/6/2016

To use combine Ints to get length:

(fun - length Of Tail > 1+ length Of Tail > 0;)

What if we write out the specialized version?

let tec length =

fune (1:intlist)  $\Rightarrow$ match 1 with

[] = 0

| hdi:t1  $\Rightarrow$  1 + (length t1);

so in fact, this combine Ints function is called reduce, fold, or in occupil,

#### Exercise:

- define fold\_left
  - extend combine Ints Left 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.

(1219 1+ + stallandows) by 7 + 15 Sbal

let map f 1 = List. fold\_right (fun hd map-f-th > f hd:: map-f-tl) 1. []

Exercise : # (misling) (poi toi ) (noi & toi & toi & toi )

-define filter using List. fold-right.

- the can remove this, and now length can be found on

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

Turion, enum! ( + too! + too! +) at 2010 an elarent & 21d

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

Enumin occum!:

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.

the combines the provides the encepts.

Now that we have this, what can we do? same constructor name.

- Check for equality

- Pattern matching

A type with some fields | data, like a structin ( type point = Point of (float + float);

Syrtax: < constructor-name> of <type>

```
We can do things like
```

let negate p =match p with

Point  $(x,y) \rightarrow Point (-,x,-,y)$ ii

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

What if we ob ...

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

type rulade Int = NoII | Non Null of inti;

This combines the provious two concepts. Note that Ocamil does not have NVL and remember that this is a good thing.

letinchvHableInt x =
montch x with

NUN > NUN

NonNull (+1)

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

let upd Wilable: In =
match in with
buil > Null

1 HonNull x > NonNull (fx)

Anullable is often used in functional programming
-in occanit:

"nullable" is called option
"Null" is called None
"Nonly" is called Some

define a function get that returns the 11th element of a list let rec get n 1 match n with 0 > {match I with mine towards & towards [] > None () Sydbory ("syort" olls In exercise the accompanion of the local termination and economical [] + None hit to get (n-1) to mon bottom sou ow mos wot We can clean this up by matching on both at the same time let rec get n =

martch (0, 1) with Some: this is imported for the type checker  $(0, h::-) \rightarrow T$  Some: this is imported for the type checker 1 (-, h:it) → get (n-1) t This teks if an elevieur x is inside a list! You let's try to return a 1st of all elements in a list that are prime index of an its or smithal half to lite and be used to be (I-a) Molart + I to = " Motour" tol # almis a dotpar SR (0=1 move ve a) (Maskin aninton South fre has an South So (Come) Wo Town) +