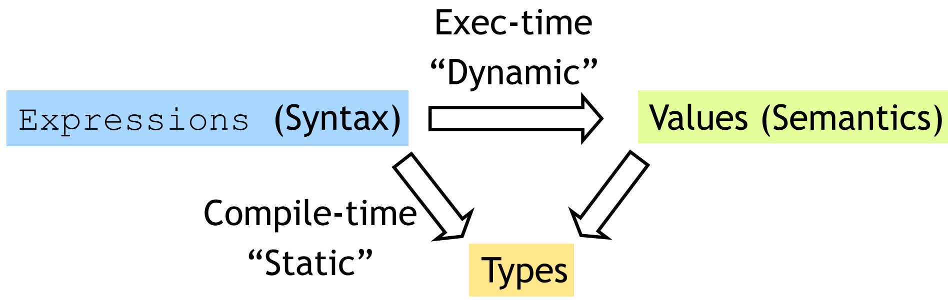
CSE 130 Programming Languages

Datatypes

Ranjit Jhala
UC San Diego

Recap: ML's Holy Trinity



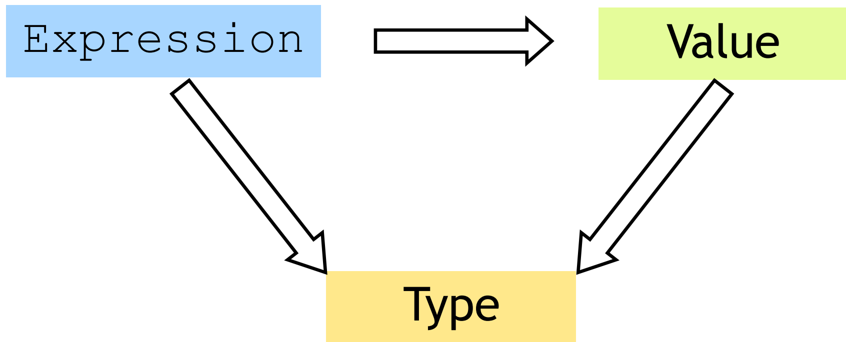
1. Programmer enters expression
2. ML checks if expression is "well-typed"
   - Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expr
3. ML evaluates expression to compute value
   - Of the same "type" found in step 2

Story So Far...

- Simple Expressions
- Branches
- Let-Bindings ...

- Today:
  - Finish Crash Course
  - Datatypes

Next: functions, but remember ...



Everything is an expression
Everything has a value
Everything has a type

A function is a value!

A shorthand for function binding

```
# let neg = fun f -> fun x -> not (f x);
…
# let neg f x = not (f x);
val neg : int -> int -> bool = fn

# let is5gte = neg is5lt;
val is5gte : int -> bool = fn;
# is5gte 10;
val it : bool = false;
# is5gte 2;
val it : bool = true;
```

Put it together: a "filter" function

If arg "matches" this pattern… …then use this Body Expr

```
- let rec filter f xs =
    match xs with
    | []       -> []
    | (x::xs')-> if f x
                 then x::(filter f xs')
                 else (filter f xs');;

val filter : ('a->bool)->'a list->'a lisi) = fn

# let list1 = [1;31;12;4;7;2;10];;
# filter is5lt list1 ;;
val it : int list = [31;12;7;10]
# filter is5gte list1;;
val it : int list = [1;4;2]
# filter even list1;;
val it : int list = [12;4;2;10]
```

Put it together: a "partition" function

```
# let partition f l = (filter f l, filter (neg f) l);
val partition :('a->bool)->'a list->'a lisi * 'a list =
fn

# let list1 = [1,31,12,4,7,2,10];
- …
# partition is5lt list1 ;
val it : (int list * int list) = ([31,12,7,10],[1,2,10]

# partition even list1;
val it : (int list * int list) = ([12,4,2,10],[1,31,7])
```

A little trick …

```
# 2 <= 3;; …
val it : bool = true
# "ba" <= "ab";;
val it : bool = false

# let lt = (<) ;;
val it : 'a -> 'a -> bool = fn

# lt 2 3;;
val it : bool = true;
# lt "ba" "ab" ;;
val it : bool = false;

# let is5Lt = lt 5;
val is5lt : int -> bool = fn;
# is5lt 10;
val it : bool = true;
# is5lt 2;
val it : bool = false;
```

Put it together: a “quicksort” function

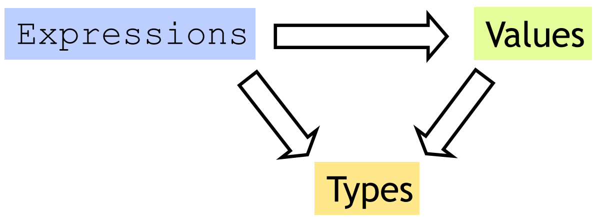
```
let rec sort xs =
  match xs with
  | []     -> []
  | (h::t) -> let (l,r) = partition ((<) h) t in
              (sort l)@(h::(sort r))
```

News

- Ocaml-top issues?
- Please post questions to Piazza
- Seating: Don’t Worry!

Now, lets begin at the beginning …

What about more complex data ?



Many kinds of expressions:

1. Simple
2. Variables
3. Functions

What about more complex data ?

- We’ve seen some base types and values:
  - Integers, Floats, Bool, String etc.
- Some ways to build up types:
  - Products (tuples), records, “lists”
  - Functions
- Design Principle: Orthogonality
  - Don’t clutter core language with stuff
  - Few, powerful orthogonal building techniques
  - Put “derived” types, values, functions in libraries

What about more complex data ?

- We've seen some base types and values:
  - Integers, Floats, Bool, String etc.
- Some ways to build up types:
  - Products (tuples), records, "lists"
  - Functions

Next: Building datatypes

Three key ways to build complex types/values

1. "Each-of" types
Value of T contains value of T1 **and** a value of T2

2. "One-of" types
Value of T contains value of T1 **or** a value of T2

3. "Recursive"
Value of T contains (sub)-value of **same type** T

Next: Building datatypes

Three key ways to build complex types/values

1. "Each-of" types **(T1 * T2)**
Value of T contains value of T1 **and** a value of T2

2. "One-of" types
Value of T contains value of T1 **or** a value of T2

3. "Recursive"
Value of T contains (sub)-value of **same type** T

Suppose I wanted …

… a program that processed lists of attributes

- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (float)
- Alive (boolean)
- Phone (int-int)
- email (string)

Many kinds of attributes (too many to put in a record)

- can have multiple names, addresses, phones, emails etc.

Want to store them in a list. Can I ?

Suppose I wanted …

Attributes:

- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (real)
- Alive (boolean)
- Phone (int-int)
- email (string)

```
type attrib =
   Name of string
|  Age of int
|  DOB of int*int*int
|  Address of string
|  Height of float
|  Alive of bool
|  Phone of int*int
|  Email of string;;
```

Quiz: Here is a typedef …

```
type attrib = Name   of string
            | Age    of int
            | Height of float
```

What is the type of: **Name "Tony Stark"**

(a) Syntax Error

(b) Type Error

(c) `string`

(d) `attrib`

(e) `'a`

Constructing Datatypes

```
type t = C1 of t1 | C2 of t2 | … | Cn of tn
```

`t` is a new datatype.

A value of type `t` is either:

a value of type *t1* placed in a box labeled `C1`

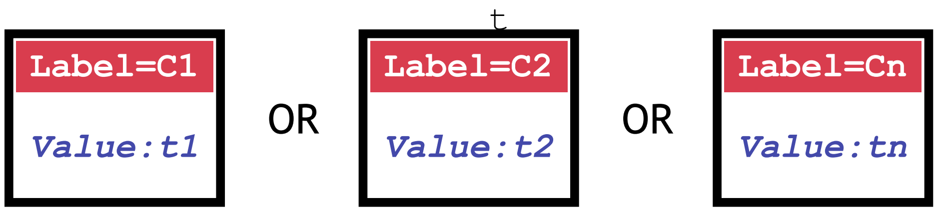
Or a value of type *t2* placed in a box labeled `C2`

Or …

Or a value of type *tn* placed in a box labeled `Cn`

Constructing Datatypes

```
type t = C1 of t1 | C2 of t2 | … | Cn of tn
```



All have the type **t**

How to PUT values into box?

Question: Here is a typedef ...

```
type attrib = Name   of string
            | Age    of int
            | Height of float
```

What is the type of: **Age "Tony Stark"**

(a) Syntax Error

(b) Type Error

(c) **`string`**

(d) **`attrib`**

(e) **`'a`**

How to PUT values into box?

How to create values of type `attrib` ?

```
# let a1 = Name "Ranjit";;
val x : attrib = Name "Ranjit"
# let a2 = Height 5.83;;
val a2 : attrib = Height 5.83
# let year = 1977 ;;
val year : int = 1977
# let a3 = DOB (9,8,year) ;;
val a3 : attrib = DOB (9,8,1977)
# let a_l = [a1;a2;a3];;
val a3 : attrib list = …
```

```
type attrib =
  Name of string
| Age of int
| DOB of int*int*int
| Address of string
| Height of float
| Alive of bool
| Phone of int*int
| Email of string;;
```

Constructing Datatypes

```
type attrib
  = Name of string     | Age of int      | DOB of int*int*int
  | Address of string  | Height of float | Alive of bool
  | Phone of int*int   | Email of string;;
```

Name		Age		DOB
"Ranjit"	OR	34	OR	(9,8,77)

`Name "Ranjit"` `Age 34` `DOB (9,8,77)`

All have type **`attrib`**

One-of types

- We've defined a "one-of" type named `attrib`
- Elements are one of:
  - string,
  - int,
  - int*int*int,
  - float,
  - bool ...

```
datatype attrib =
   Name of string
 | Age of int
 | DOB of int*int*int
 | Address of string
 | Height of real
 | Alive of bool
 | Phone of int*int
 | Email of string;
```

- Can create uniform `attrib` lists
- Say I want a function to print attribs...

Question: Here is a typedef ...

```
type attrib = Name of string
            | Age of int
            | Height of float
```

What is the type of:

```
[Name "Ranjit"; Age 35; Dob(9,8,77)]
```

(a) Syntax Error

(b) Type Error

(c) **`string * int * (int*int*int) list`**

(d) **`'a list`**

(e) **`attrib list`**

How to TEST & TAKE whats in box?

Is it a ...

`string?`

or an

`int?`

or an

`int*int*int?`

or ...

How to TEST & TAKE whats in box?

Look at TAG!

Question: Here is a typedef ...

```
type attrib = Name of string | Age of int | ...
```

What does this evaluate to?

```
let welcome a = match a with
                | Name s -> s
  in welcome (Name "Ranjit")
```

(a) **Name "Ranjit" : 'a**

(b) Type Error

(c) **Name "Ranjit" : attrib**

(d) **"Ranjit" : string**

(e) Runtime Error

How to tell whats in the box ?

```
type attrib =
  Name of string
| Age of int
| DOB of int*int*int
| Address of string
| Height of float
| Alive of bool
| Phone of int*int
```

```
match e with
| Name s     -> ...(*s: string *)
| Age i      -> ...(*i: int *)
| DOB(d,m,y)-> ...(*d: int,m: int,y: int*)
| Address a -> ...(*a: string*)
| Height h   -> ...(*h: int *)
| Alive b    -> ...(*b: bool*)
| Phone(y,r)-> ...(*a: int, r: int*)
```

Pattern-match expression: check if e is of the form ...

- On match:
  - value in box bound to pattern variable
  - matching result expression is evaluated
- Simultaneously test and extract contents of box

How to tell whats in the box ?

```
match e with
| Name s      -> printf "%s" s
| Age i       -> printf "%d" i
| DOB(d,m,y) -> printf "%d/%d/%d" d m y
| Address s  -> printf "%s" s
| Height h    -> printf "%f" h
| Alive b     -> printf "%b" b s
| Phone(a,r) -> printf "(%d)-%d" a r
```

Pattern-match expression: check if e is of the form ...

- On match:
  - value in box bound to pattern variable
  - matching result expression is evaluated
- Simultaneously test and extract contents of box

Question: Here is a typedef ...

type attrib = Name of string | Age of int | ...

What does this evaluate to?

```
let welcome a = match a with
                | Name s -> s
  in welcome (Age 34)
```

(a) **Name "Ranjit" : 'a**

(b) Type Error

(c) **Name "Ranjit" : attrib**

(d) **"Ranjit" : string**

(e) Runtime Error

How to tell whats in the box

```
# match (Name “Ranjit”) with
  | Name s -> printf "Hello %s\n" s
  | Age i  -> printf "%d years old" i
  ;;

Hello Ranjit
- : unit = ()
```

First case matches the tag (Name)

Evals branch with s “bound” to string contents

How to TEST & TAKE whats in box?

BEWARE!!
Be sure to handle all TAGS!

Beware! Handle All TAGS!

```
# match (Name “Ranjit”) with
  | Age i   -> Printf.printf "%d" i
  | Email s -> Printf.printf "%s" s
  ;;

Exception: Match Failure!!
```

None of the cases matched the tag (Name)

Causes nasty ***Run-Time Error***

Compiler To The Rescue!!

```
# let printAttrib a =
    match a with
      Name s -> Printf.printf "%s" s
    | Age i  -> Printf.printf "%d" i
    | DOB (d,m,y) -> Printf.printf "%d / %d / %d" d m y
    | Address addr -> Printf.printf "%s" addr
    | Height h -> Printf.printf "%f" h
    | Alive b -> Printf.printf "%b" b
    | Email e -> Printf.printf "%s" e
  ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Phone (_, _)
```

Compile-time checks for:

missed cases: ML warns if you miss a case!

Q: What does this evaluate to ?

```
type attrib = Name of string | ...

    let welcome a = match a with
                  | Name s -> "Hello!" ^ s
                  | Name s -> "Welcome!" ^ s
    in welcome (Name "Mickey")
```

(a) Type Error

(b) **"Welcome!Mickey" : string**

(c) Runtime Error

(d) **"Hello!Mickey" : string**

(e) **"Hello!MickeyWelcome!Mickey"Ranjit" : string**

Compiler To The Rescue!!

```
# let printAttrib a =
    match a with
      Name s -> Printf.printf "%s" s
    | Age i  -> Printf.printf "%d" i
    | DOB (d,m,y) -> Printf.printf "%d / %d / %d" d m y
    ...
    | Age i  -> Printf.printf "%d" i
  ;;

Warning U: this match case is unused.
```

Compile-time checks for:

redundant cases: ML warns if a case never matches

Benefits of **match-with**

```
match e with
  C1 x1 -> e1
| C2 x2 -> e2
| …
| Cn xn -> en
```

```
type t =
  C1 of t1
| C2 of t2
| …
| Cn of tn
```

1. Simultaneous test-extract-bind
2. Compile-time checks for:

   missed cases: ML warns if you miss a **t** value

   redundant cases: ML warns if a case never matches

match-with is an Expression

```
match e with
  C1 x1 -> e1
| C2 x2 -> e2
| …
| Cn xn -> en
```

Q: What does this evaluate to ?

```
type attrib = Name of string | Age of int | ...

    let welcome a = match a with
                  | Name s -> s
                  | Age i  -> i
    in welcome (Name "Ranjit")
```

(a) **`"Ranjit" : string`**

(b) Type Error

(c) **`Name "Ranjit" : attrib`**

(d) Runtime Error

`match-with` is an Expression

```
match e with
  Name s       -> e1
| Age i        -> e2
| DOB (m,d,y)  -> e3
| Address a    -> e4
| Height h     -> e5
| Alive b      -> e6
| Phone (a,n)  -> e7
| Email e      -> e8
```

T { (whole match expression) ← T (each of e1 … e8)

Type Rule

- ***e1***, ***e2,…,en*** must have same type `T`
- Type of whole expression is `T`

Next: Building datatypes

Three key ways to build complex types/values

1. "Each-of" types **t1 * t2**
Value of T contains value of T1 **and** a value of T2

2. "One-of" types **type t = C1 of t1 | C2 of t2**
Value of T contains value of T1 **or** a value of T2

3. "Recursive" type
Value of T contains (sub)-value of **same type** T

"Recursive" types

```
type nat = Zero | Succ of nat
```

"Recursive" types

```
type nat = Zero | Succ of nat
```

Wait a minute! **Zero** of what ?!

"Recursive" types

```
type nat = Zero | Succ of nat
```

Wait a minute! **Zero** of what ?!

Relax.

Means "empty box with label **Zero**"

"Recursive" types

```
type nat = Zero | Succ of nat
```

What are values of **nat** ?

"Recursive" types

```
type nat = Zero | Succ of nat
```

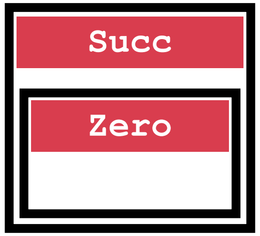
What are values of **nat** ?

“Recursive” types

`type nat = Zero | Succ of nat`

What are values of **`nat`** ?

One **`nat`** contains another!

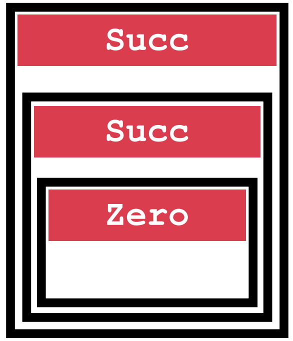


“Recursive” types

`type nat = Zero | Succ of nat`

What are values of **`nat`** ?

One **`nat`** contains another!

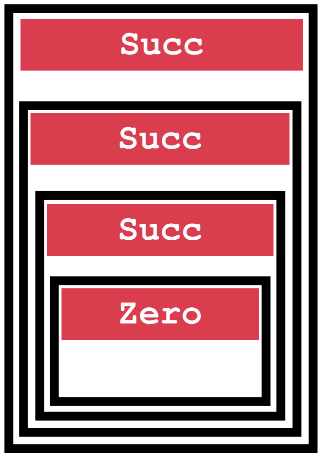


“Recursive” types

`type nat = Zero | Succ of nat`

What are values of **`nat`** ?

One **`nat`** contains another!



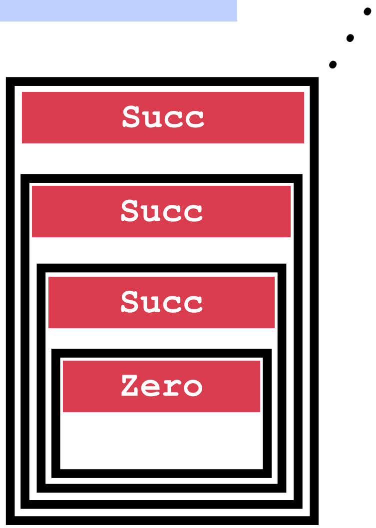
“Recursive” types

`type nat = Zero | Succ of nat`

What are values of **`nat`** ?

One **`nat`** contains another!

`nat` = recursive type



Next: Building datatypes

Three key ways to build complex types/values

1. "Each-of" types **t1 * t2**
Value of T contains value of T1 **and** a value of T2

2. "One-of" types **type t = C1 of t1 | C2 of t2**
Value of T contains value of T1 **or** a value of T2

3. "Recursive" type **type t = ...| C of (...*t)**
Value of T contains (sub)-value of **same type** T

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

Next: Lets get cosy with Recursion

Code Structure = Type Structure!!!

to_int : nat -> int

```
type nat =
| Zero
| Succ of nat
```

```
let rec to_int n =
```

to_int : nat -> int

```
type nat =
| Zero
| Succ of nat
```

Base pattern: `| Zero`
Inductive pattern: `| Succ`

```
let rec to_int n =
```

to_int : nat -> int

```
type nat =
| Zero
| Succ of nat
```

Base pattern: `| Zero`
Inductive pattern: `| Succ`

```
let rec to_int n = match n with
| Zero   ->
| Succ m ->
```

Base pattern: `| Zero`
Inductive pattern: `| Succ m`

to_int : nat -> int

```
type nat =
| Zero
| Succ of nat
```

Base pattern: `| Zero`
Inductive pattern: `| Succ`

```
let rec to_int n = match n with
| Zero   -> 0
| Succ m -> 1 + to_int m
```

Base pattern: `| Zero`; Base Expression: `0`
Inductive pattern: `| Succ m`; Inductive Expression: `1 + to_int m`

Q: What does this evaluate to ?

```
let rec foo n =
  if n<=0 then Zero else Succ(foo(n-1))
in foo 2
```

(a) **Zero : nat**

(b) **Type Error**

(c) **2 : nat**

(c) **Succ(Zero): nat**

(c) **Succ(Succ(Zero)) : nat**

of_int : int -> nat

```
type nat =
| Zero
| Succ of nat
```

```
let rec of_int n =
```

of_int : int -> nat

```
type nat =
| Zero          (* Base pattern *)
| Succ of nat   (* Inductive pattern *)
```

```
let rec of_int n =
```

of_int : int -> nat

```
type nat =
| Zero          (* Base pattern *)
| Succ of nat   (* Inductive pattern *)
```

```
let rec of_int n =
  if n <= 0 then   (* Base pattern *)

  else             (* Inductive pattern *)
```

of_int : int -> nat

```
type nat =
| Zero          (* Base pattern *)
| Succ of nat   (* Inductive pattern *)
```

```
let rec of_int n =
  if n <= 0 then          (* Base pattern *)
    Zero                  (* Base Expression *)
  else                    (* Inductive pattern *)
    Succ (of_int (n-1))   (* Inductive Expression *)
```

plus : nat*nat -> nat

```
type nat =
| Zero
| Succ of nat
```

```
let rec plus (n,m) =
```

plus : nat*nat -> nat

Base pattern

Inductive pattern

```
type nat =
| Zero
| Succ of nat
```

```
let rec plus (n,m) =
```

plus : nat*nat -> nat

Base pattern

Inductive pattern

```
type nat =
| Zero
| Succ of nat
```

Base pattern

Inductive pattern

```
let rec plus (n,m) =
match m with
| Zero     ->
| Succ m’  ->
```

plus : nat*nat -> nat

Base pattern

Inductive pattern

```
type nat =
| Zero
| Succ of nat
```

Base pattern

Inductive pattern

Base Expression

Inductive Expression

```
let rec plus (n,m) =
match m with
| Zero     -> n
| Succ m’  -> Succ (plus (n,m’))
```

times: nat*nat -> nat

```
type nat =
| Zero
| Succ of nat
```

```
let rec times (n,m) =
```

times: nat*nat -> nat

```
type nat =
| Zero
| Succ of nat
```

Base pattern: `| Zero`

Inductive pattern: `| Succ`

```
let rec times (n,m) =
```

times: nat*nat -> nat

```
type nat =
| Zero
| Succ of nat
```

Base pattern: `| Zero`

Inductive pattern: `| Succ`

```
let rec times (n,m) =
match m with
| Zero    ->
| Succ m' ->
```

Base pattern: `| Zero`

Inductive pattern: `| Succ m'`

plus : nat*nat -> nat

```
type nat =
| Zero
| Succ of nat
```

Base pattern: `| Zero`

Inductive pattern: `| Succ`

```
let rec times (n,m) =
match m with
| Zero    -> Zero
| Succ m' -> plus n (times (n,m'))
```

Base pattern: `| Zero`

Inductive pattern: `| Succ m'`

Base Expression: `Zero`

Inductive Expression: `plus n (times (n,m'))`

minus: nat*nat -> nat

```
type nat =
| Zero
| Succ of nat
```

```
let rec minus (n,m) =
```

times: nat*nat -> nat

Base pattern (| Zero)
Inductive pattern (| Succ) of nat

```
type nat =
| Zero
| Succ of nat
```

```
let rec minus (n,m) =
```

times: nat*nat -> nat

Base pattern (| Zero)
Inductive pattern (| Succ) of nat

```
type nat =
| Zero
| Succ of nat
```

Base pattern: | (_, Zero)
Inductive pattern: | (Succ n', Succ m')

```
let rec minus (n,m) =
match (n, m) with
| (_, Zero)             -> n
| (Succ n', Succ m')    -> minus(n',m')
```

times: nat*nat -> nat

Base pattern (| Zero)
Inductive pattern (| Succ) of nat

```
type nat =
| Zero
| Succ of nat
```

Base pattern: | (_, Zero) — *Base Expression*: n
Inductive pattern: | (Succ n', Succ m') — *Inductive Expression*: minus(n',m')

```
let rec minus (n,m) =
match (n, m) with
| (_, Zero)             -> n
| (Succ n', Succ m')    -> minus(n',m')
```

Next: Lets get cosy with Recursion

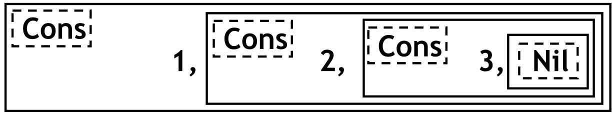
Recursive Code Mirrors Recursive Data

Lists are recursive types!

```
type int_list =
  Nil
| Cons of int * int_list
```

Think about this! What are values of `int_list` ?

Cons(1,Cons(2,Cons(3,Nil))) Cons(2,Cons(3,Nil)) Cons(3,Nil) Nil



Lists aren't built-in !

```
datatype int_list =
  Nil
| Cons of int * int_list
```

Lists are a derived type: built using elegant core!

1. Each-of
2. One-of
3. Recursive

`::` is just a pretty way to say "Cons"

`[]` is just a pretty way to say "Nil"

Some functions on Lists : Length

```
let rec len l =
  match l with
    Nil -> 0
  | Cons(h,t) -> 1 + (len t)
```

Base pattern — `Nil` ; *Base Expression* — `0`

Ind pattern — `| Cons(h,t)` ; *Inductive Expression* — `1 + (len t)`

```
let rec len l =
  match l with
    Nil -> 0
  | Cons(_,t) -> 1 + (len t)
```

Matches everything, no binding

```
let rec len l =
  match l with
    Cons(_,t) -> 1 + (len t)
  | _ -> 0
```

Pattern-matching in order
- Must match with `Nil`

Some functions on Lists : Append

```
let rec append (l1,l2) =
```

Base pattern — *Base Expression*

Ind pattern — *Inductive Expression*

- Find the right induction strategy
  - Base case: pattern + expression
  - Induction case: pattern + expression

Well designed datatype gives strategy

null, hd, tl are all functions …

Bad ML style: More than aesthetics !

Pattern-matching better than test-extract:

- ML checks all cases covered
- ML checks no redundant cases
- …at compile-time:
  - fewer errors (crashes) during execution
  - get the bugs out ASAP!

Some functions on Lists : Max

```
let rec max xs =
```

Base pattern — *Base Expression*

Ind pattern — *Inductive Expression*

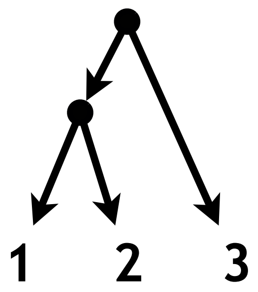
- Find the right induction strategy
  - Base case: pattern + expression
  - Induction case: pattern + expression

Well designed datatype gives strategy

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

Q: How is this tree represented ?




```
type tree =
| Leaf of int
| Node of tree*tree
```

(a) `(1, 2), 3`

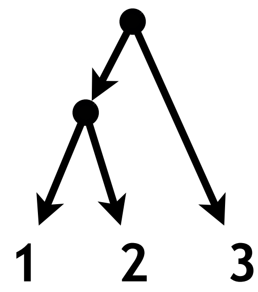
(b) `(Leaf 1, Leaf 2), Leaf 3`

(c) `Node (Node (Leaf 1, Leaf 2), Leaf 3)`

(d) `Node ((Leaf 1, Leaf 2), Leaf 3)`

(e) `None of the above`

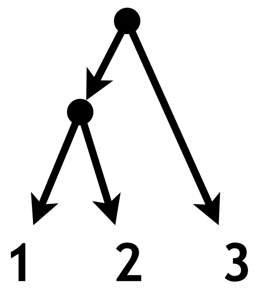
Representing Trees




```
type tree =
| Leaf of int
| Node of tree*tree
```

`Leaf 1`

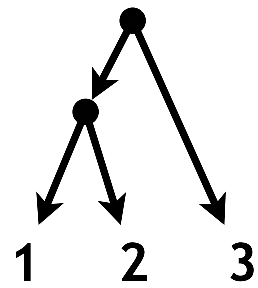
Representing Trees

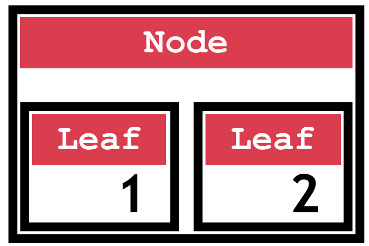



```
type tree =
| Leaf of int
| Node of tree*tree
```

`Leaf 2`

Representing Trees

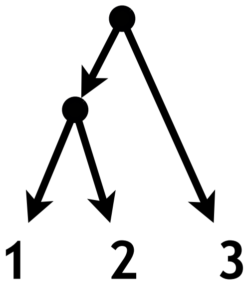





```
type tree =
| Leaf of int
| Node of tree*tree
```

`Node(Leaf 1, Leaf 2)`

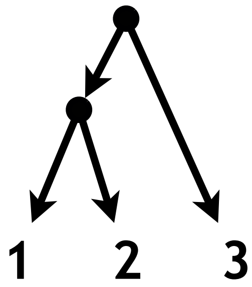
Representing Trees

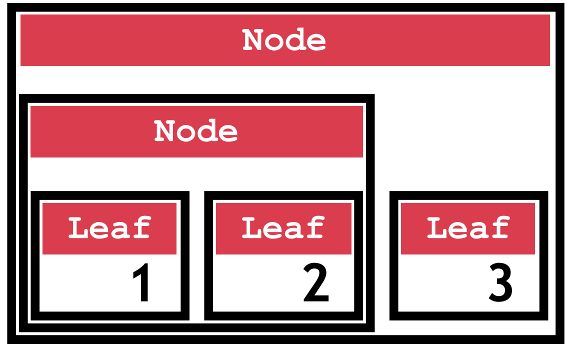



```
type tree =
| Leaf of int
| Node of tree*tree
```

`Leaf 3`

Representing Trees

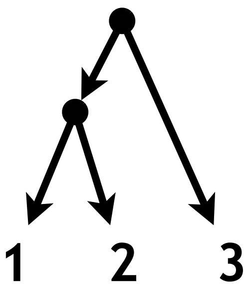


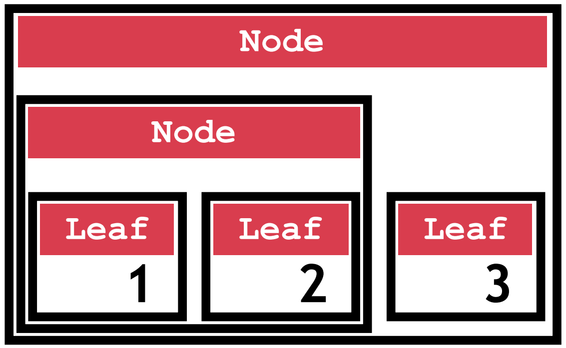



```
type tree =
| Leaf of int
| Node of tree*tree
```

`Node(Node(Leaf 1, Leaf 2), Leaf 3)`

Representing Trees






```
type tree =
| Leaf of int
| Node of tree*tree
```

`Node(Node(Leaf 1, Leaf 2), Leaf 3)`

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

Q: What does this evaluate to ?

```
let rec foo t = match t with
  | Leaf n         -> 1
  | Node (t1, t2) -> foo t1 + foo t2

foo (Node(Node(Leaf 1,Leaf 2),Leaf 3))
```

(a) **Type Error**

(b) **1 : int**

(c) **3 : int**

(d) **6 : int**

sum_leaf: tree -> int

"Sum up the leaf values". E.g.

```
# let t0 = Node(Node(Leaf 1, Leaf 2), Leaf 3);;
# sum_leaf t0 ;;
- : int = 6
```

sum_leaf: tree -> int

```
type tree =
| Leaf of int
| Node of tree*tree
```

```
let rec sum_leaf t =
```

sum_leaf: tree -> int

```
type tree =
| Leaf of int
| Node of tree*tree
```

Base pattern: | Leaf

Inductive pattern: | Node

```
let rec sum_leaf t =
```

sum_leaf: tree -> int

Base pattern / *Inductive pattern*

```
type tree =
| Leaf of int
| Node of tree*tree
```

Base pattern / *Inductive pattern*

```
let rec sum_leaf t =
match t with
|Leaf n        ->
|Node(t1,t2)->
```

sum_leaf: tree -> int

Base pattern / *Inductive pattern*

```
type tree =
| Leaf of int
| Node of tree*tree
```

Base pattern / *Inductive pattern* / *Base Expression* / *Inductive Expression*

```
let rec sum_leaf t =
match t with
|Leaf n        -> n
|Node(t1,t2)-> sum_leaf t1 + sum_leaf t2
```

Recursive Code Mirrors Recursive Data

Code almost writes itself!

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- 4.0 + 2.9
- 3.78 - 5.92
- (4.0 + 2.9) * (3.78 -5.92)

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- 4.0 + 2.9 ====> ***6.9***
- 3.78 – 5.92 ====> ***-2.14***
- (4.0 + 2.9) * (3.78 -5.92) ====> ***-14.766***

Whats a ML **TYPE** for **REPRESENTING** expressions ?

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- 4.0 + 2.9 ====> ***6.9***
- 3.78 – 5.92 ====> ***-2.14***
- (4.0 + 2.9) * (3.78 -5.92) ====> ***-14.766***

Whats a ML **TYPE** for **REPRESENTING** expressions ?


```
type expr =
| Num of float
| Add of expr*expr
| Sub of expr*expr
| Mul of expr*expr
```

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- 4.0 + 2.9 ====> ***6.9***
- 3.78 – 5.92 ====> ***-2.14***
- (4.0 + 2.9) * (3.78 -5.92) ====> ***-14.766***

Whats a ML **FUNCTION** for **EVALUATING** expressions ?


```
type expr =
| Num of float
| Add of expr*expr
| Sub of expr*expr
| Mul of expr*expr
```

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- 4.0 + 2.9 ====> ***6.9***
- 3.78 – 5.92 ====> ***-2.14***
- (4.0 + 2.9) * (3.78 -5.92) ====> ***-14.766***

Whats a ML **FUNCTION** for **EVALUATING** expressions ?

```
type expr =
| Num of float
| Add of expr*expr
| Sub of expr*expr
| Mul of expr*expr
```

Another Example: Calculator

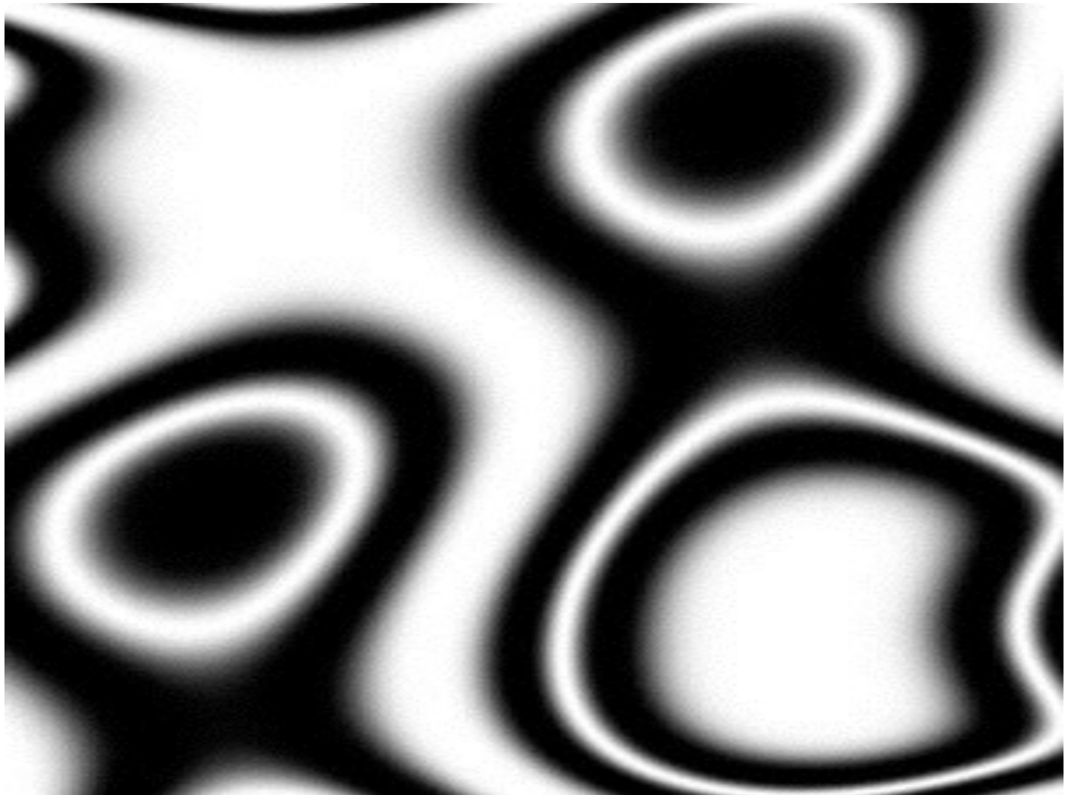
Want an arithmetic calculator to evaluate expressions like:

- 4.0 + 2.9 ====> ***6.9***
- 3.78 – 5.92 ====> ***-2.14***
- (4.0 + 2.9) * (3.78 -5.92) ====> ***-14.766***

Whats a ML **FUNCTION** for **EVALUATING** expressions ?

```
type expr =
| Num of float
| Add of expr*expr
| Sub of expr*expr
| Mul of expr*expr
```

```
let rec eval e = match e with
|Num f       ->
|Add(e1,e2)->
|Sub(e1,e2)->
|Mul(e1,e2)->
```



Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- 4.0 + 2.9 ====> ***6.9***
- 3.78 – 5.92 ====> ***-2.14***
- (4.0 + 2.9) * (3.78 -5.92) ====> ***-14.766***

Whats a ML **FUNCTION** for **EVALUATING** expressions ?

```
type expr =
| Num of float
| Add of expr*expr
| Sub of expr*expr
| Mul of expr*expr
```


```
let rec eval e = match e with
|Num f       -> f
|Add(e1,e2)-> eval e1 +. eval e2
|Sub(e1,e2)-> eval e1 -. eval e2
|Mul(e1,e2)-> eval e1 *. eval e2
```