

CSE 130 : Spring 2012

Programming Languages

Lecture 5: Recursion

Ranjit Jhala
UC San Diego

News

On webpage:

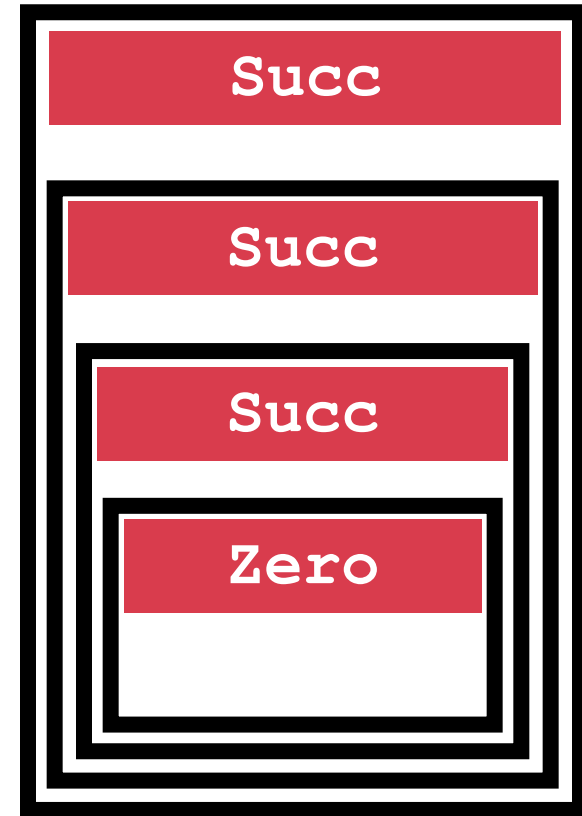
- PA #1 (**due Friday 4/20**)
- In Class Exercises, bring laptops

“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

of_int : int -> nat

```
let rec of_int n =
```

Base pattern **if n <= 0 then**

Zero *Base Expression*

Inductive pattern **else** *Inductive Expression*

Succ (of_int (n-1))

of_int 0 ==> Zero

of_int 1 ==> Succ (of_int 0)
==> Succ (Zero)

of_int 2 ==> Succ (of_int 1)
==> Succ (Succ (Zero))

to_int : nat -> int

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

Base pattern | *Base Expression*

Inductive pattern | *Inductive Expression*

to_int Zero ==> 0

to_int (Succ Zero) ==> 1 + to_int Zero
==> 1 + 0
==> 1

to_int (Succ (Succ Zero)) ==> 1 + to_int (Succ Zero)
==> 1 + 1
==> 2

plus: nat -> nat -> nat

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

Base pattern (around Zero)

Inductive pattern (around Succ m')

Base Expression (around n)

Inductive Expression (around Succ (plus n m'))

```
plus Zero (Succ (Succ Zero))  
====> Succ (Succ Zero)
```

```
plus (Succ Zero) (Succ (Succ Zero))  
====> Succ (plus Zero (Succ (Succ Zero)))  
====> Succ (Succ (Succ Zero))
```

`minus: nat*nat -> nat`

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

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

`minus: nat*nat -> nat`

Base pattern `| Zero`
Inductive pattern `| Succ` of nat

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

minus: nat*nat -> nat

```
type nat =
```

Base pattern | Zero

Inductive pattern | Succ of nat

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

```
match (n, m) with
```

Base pattern | (_, Zero) -> n

Inductive pattern | (Succ n', Succ m') -> minus(n',m')

minus: nat*nat -> nat

```
type nat =
```

Base pattern | Zero

Inductive pattern | Succ of nat

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

```
match (n, m) with
```

Base pattern | (_, Zero) -> n *Base Expression*

Inductive pattern | (Succ n', Succ m') -> minus(n', m') *Inductive Expression*

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

Lists are recursive types!

```
type 'a list =  
  Nil  
| Cons of 'a * 'a list
```

Think about this! What are values of `int list` ?

Lists are recursive types!

```
type 'a list =  
  Nil  
| Cons of 'a * 'a list
```

Think about this! What are values of `int list` ?

Nil

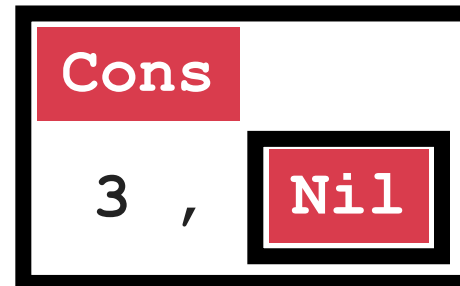
Nil

Lists are recursive types!

```
type 'a list =  
  Nil  
| Cons of 'a * 'a list
```

Think about this! What are values of `int list` ?

Cons(3,Nil) Nil

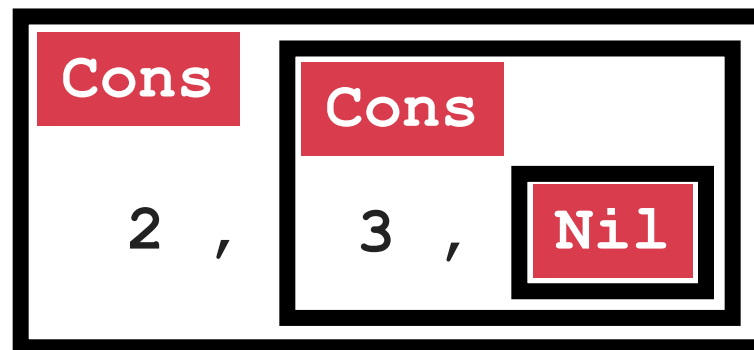


Lists are recursive types!

```
type 'a list =  
  Nil  
| Cons of 'a * 'a list
```

Think about this! What are values of `int list` ?

`Cons(2,Cons(3,Nil))` `Cons(3,Nil)` `Nil`



Lists are recursive types!

```
type 'a list =  
  Nil  
| Cons of 'a * 'a list
```

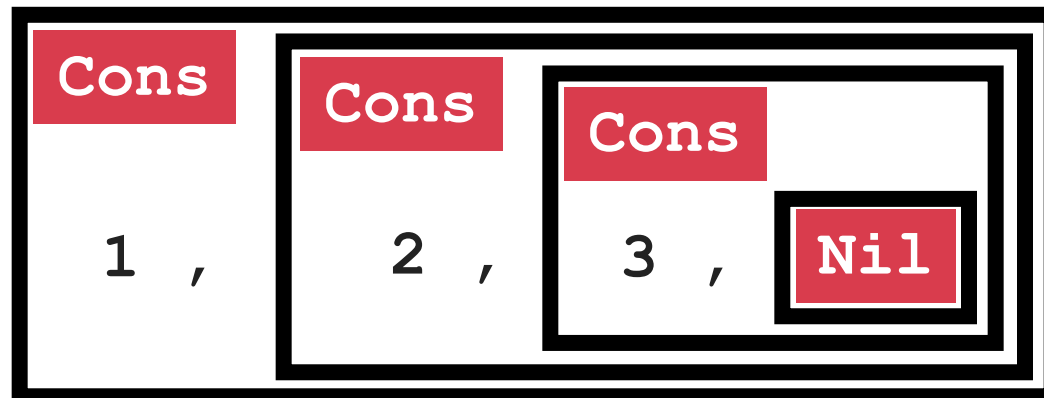
Think about this! What are values of `int list` ?

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

`Cons(2,Cons(3,Nil))`

`Cons(3,Nil)`

`Nil`



Lists aren't built-in !

```
type 'a list =  
  Nil  
| Cons of 'a * 'a 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”

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

Some functions on Lists : len

Base pattern

```
let rec len l =  
  match l with  
  | [] -> 0  
  | h::t -> 1 + (len t)
```

Base Expression

Ind pattern

Inductive Expression

```
let rec len l =  
  match l with  
  | [] -> 0  
  | _ :: t -> 1 + (len t)
```

Matches everything, no binding

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

Pattern-matching in order

- Must match with []

Some functions on Lists : len

```
let rec len l =  
  match l with  
    | [] -> 0  
    | h::t -> 1 + (len t)
```

Base pattern *Base Expression*

Ind pattern *Inductive Expression*

`len [] ==> 0`

`len ("cat" :: []) ==> 1 + (len [])`
`==> 1 + 0`
`==> 1`

`len ("dog":: "cat" ::[]) ==> 1 + len ("cat" :: [])`
`==> 1 + 1`
`==> 2`

Some functions on Lists : sum

```
(* val sum : int list -> int *)  
let rec sum xs =  
  match xs with  
  | [] -> 0  
  | h::t -> h + (sum t)
```

Base pattern *Base Expression*

Ind pattern *Inductive Expression*

sum [] ==> 0

sum (2::[]) ==> 2 + sum []
 ==> 2 + 0
 ==> 2

sum (1::2::[]) ==> 1 + sum (2::[])
 ==> 1 + 2
 ==> 3

Some functions on Lists : member

```
(* val mem : 'a -> 'a list -> bool*)
```

```
let rec mem x ys =
```

```
match ys with
```

```
| [] -> false
```

```
| y::ys' -> if x=y then true else mem x ys'
```

Base Expression

Inductive Expression

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

Well designed datatype gives strategy

Some functions on Lists : member

```
(* val mem : 'a -> 'a list -> bool*)
```

```
let rec mem x ys =
```

```
match ys with
```

```
| [] -> false
```

```
| y::ys' -> if x=y then true else mem x ys'
```

Base Expression

Inductive Expression

```
mem 2 (2::[]) ==> true
```

```
mem 2 (1::2::[]) ==> mem 2 (2::[])
```

```
==> true
```

Some functions on Lists : member

```
(* val mem : 'a -> 'a list -> bool*)
```

```
let rec mem x ys =
```

```
match ys with
```

```
| [] -> false
```

```
| y::ys' -> if x=y then true else mem x ys'
```

Base Expression

Inductive Expression

`mem 4 [] ==> false`

`mem 4 (2::[]) ==> mem 4 []`
`==> false`

`mem 4 (1::2::[]) ==> mem 4 (2::[])`
`==> false`

Some functions on Lists : insert

```
(* val insert : 'a -> 'a list -> 'a list *)  
let rec member x xs =
```

Base pattern

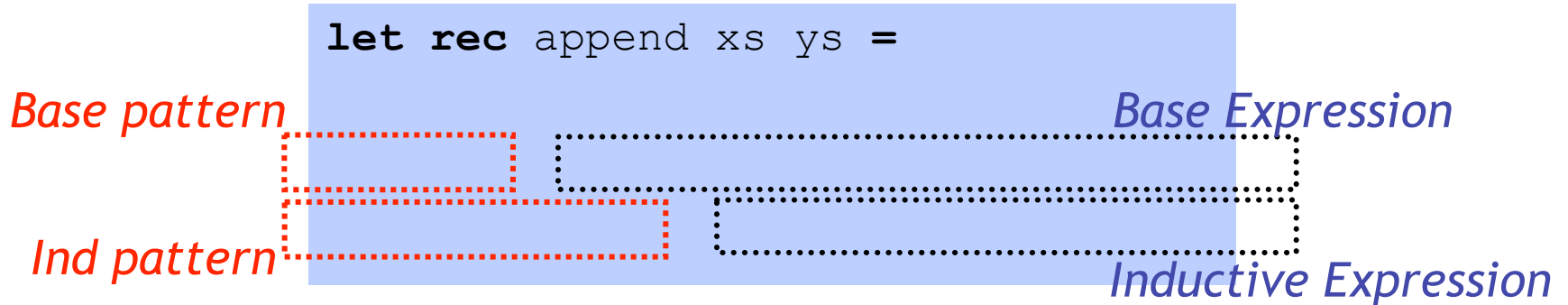
Base Expression

Ind pattern

Inductive Expression

Online

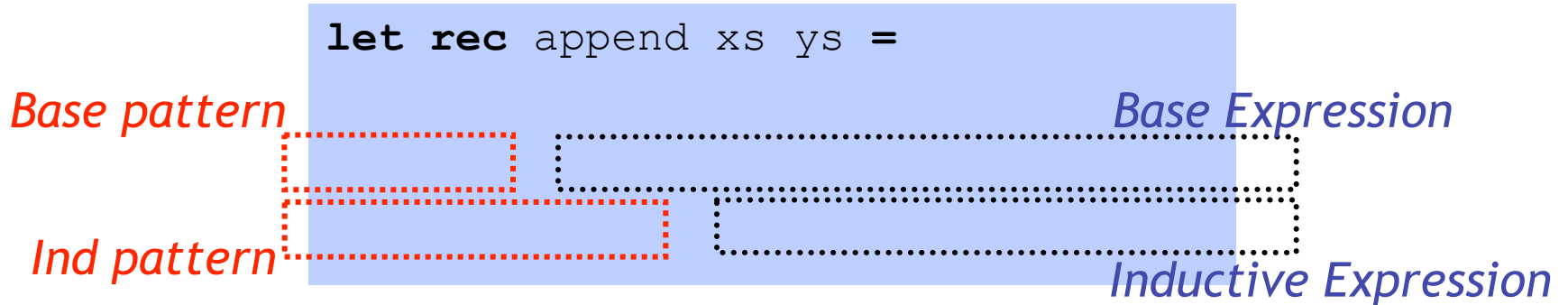
Some functions on Lists : append



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

Well designed datatype gives strategy

Some functions on Lists : append



`append [] [] ==> []`

`append [] (3::4::[]) ==> 3::4::[]`

`append (2::[]) (3::4::[]) ==> 2::(3::4::[])`
`==> 2::3::4::[]`

`append (1::2::[]) (3::4::[]) ==> 1::2::(3::4::[])`
`==> 1::2::3::4::[]`

Some functions on Lists : clone

```
(* val clone : int -> 'a -> 'a list *)  
let rec clone n x =
```

Online

null, hd, tl are all functions ...

Bad ML style: More than aesthetics !

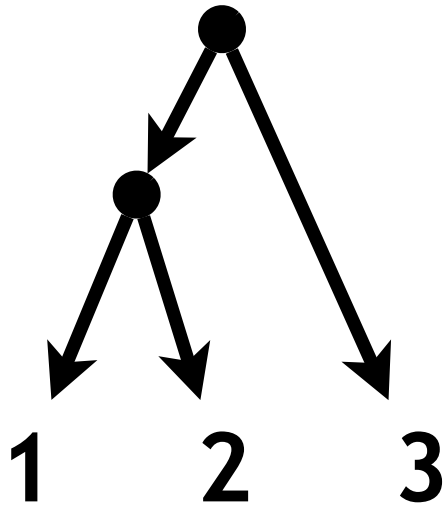
Pattern-matching better than test-extract:

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

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

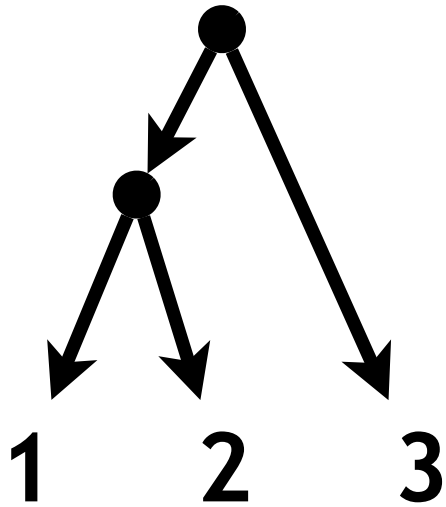
Representing Trees



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

Leaf 1

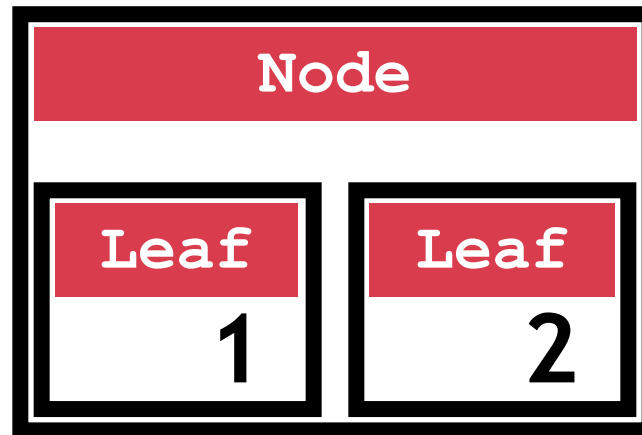
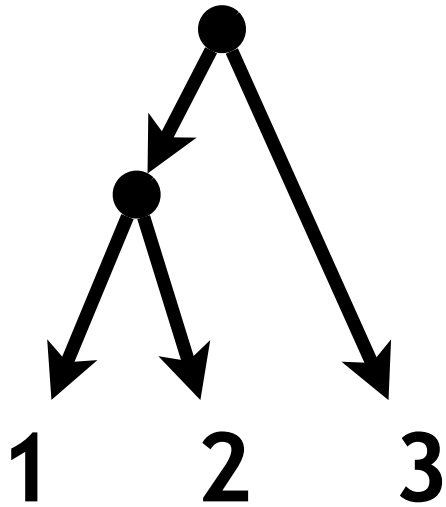
Representing Trees



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

Leaf 2

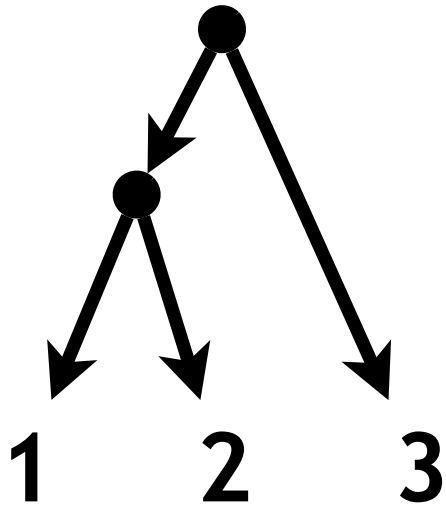
Representing Trees



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

Node (Leaf 1, Leaf 2)

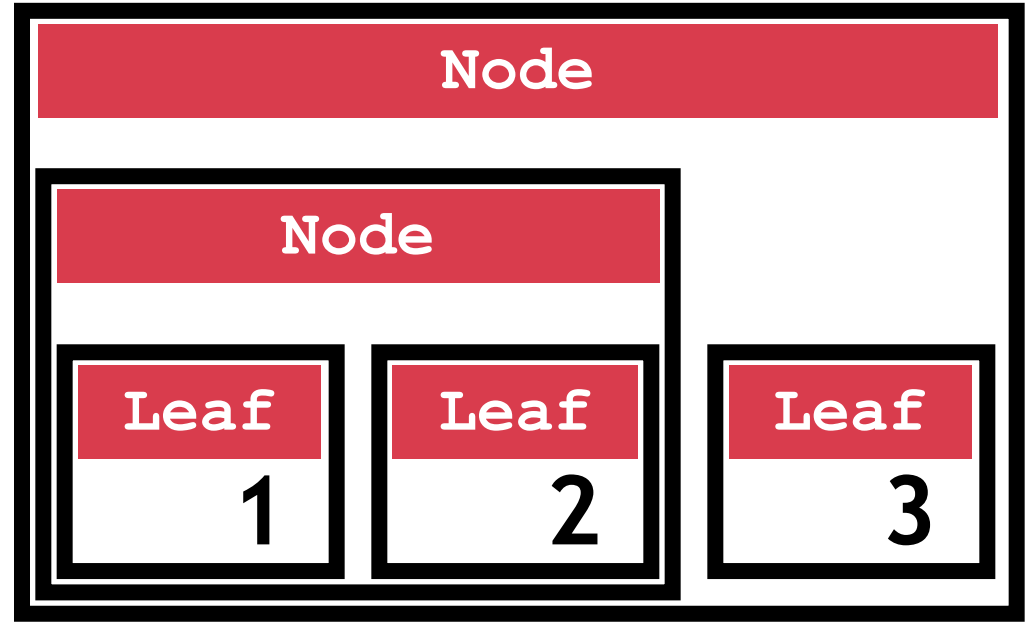
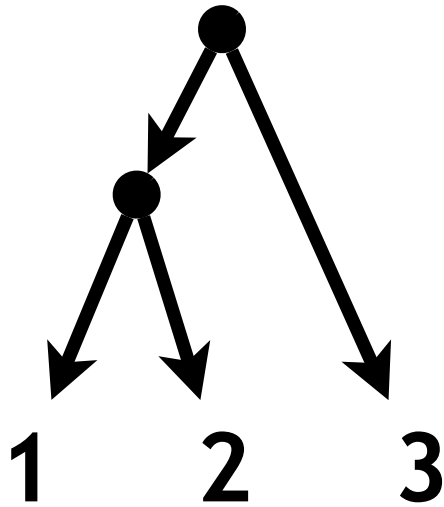
Representing Trees



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

Leaf 3

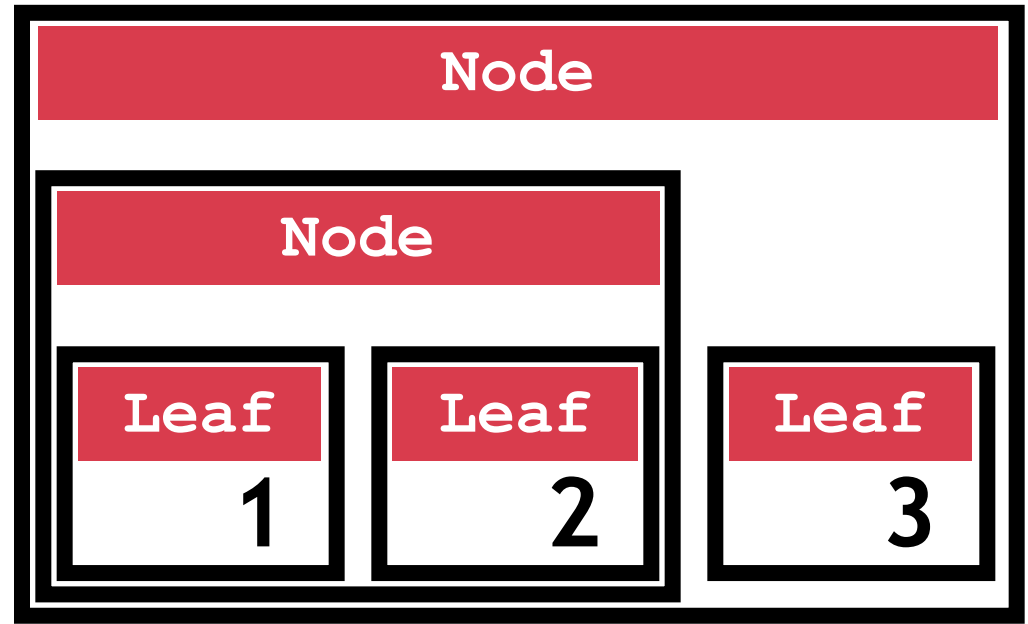
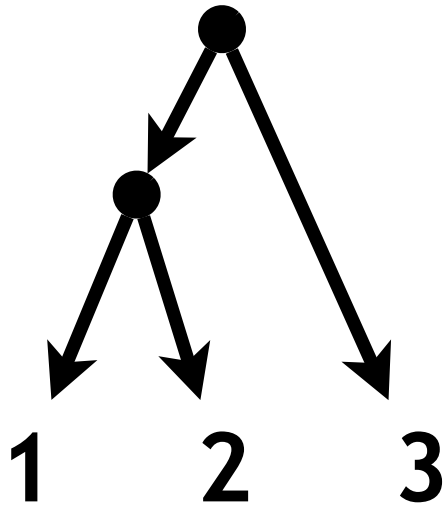
Representing Trees



```
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

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

Base pattern | **Leaf** of int
Inductive pattern | **Node** of tree*tree

```
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

```
type tree =
```

Base pattern

```
| Leaf of int
```

Inductive pattern

```
| Node of tree*tree
```

```
let rec sum_leaf t =
```

```
match t with
```

Base pattern

```
| Leaf n -> n Base Expression
```

Inductive pattern

```
| Node(t1,t2) -> sum_leaf t1 + sum_leaf t2
```

Inductive Expression

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:

- $4.0 + 2.9 \implies 6.9$
- $3.78 - 5.92 \implies -2.14$
- $(4.0 + 2.9) * (3.78 - 5.92) \implies -14.766$

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

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- $4.0 + 2.9 \implies 6.9$
- $3.78 - 5.92 \implies -2.14$
- $(4.0 + 2.9) * (3.78 - 5.92) \implies -14.766$

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:

- $4.0 + 2.9 \implies 6.9$
- $3.78 - 5.92 \implies -2.14$
- $(4.0 + 2.9) * (3.78 - 5.92) \implies -14.766$

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

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

Online

Random Art from Expressions

PA #2

Build more funky expressions, evaluate them, to produce:

