CSE 130 [Winter 2014] Programming Languages

Datatypes

(Continued)

Ravi Chugh

UCSDCSE

Computer Science and Engineering

Jan 21

Announcements

- Theme for this week: Recursion!
- HW #2 due Mon Jan 27
- Reminder: Clicker Frequency BD

Benefits of `match-with`

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

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

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

3 Ways to Build Complex Values

Tuple (a.k.a. "Each-of", "Product") Type

```
type t = (t1 * t2)
```

Value of `t` contains value of `t1` **and** a value of `t2`

Data (a.k.a. "One-of", "Variant") Type

```
type t = C1 of t1 | C2 of t2
```

Value of `t` contains value of `t1` **or** a value of `t2`

Recursive Datatype

```
type t = ... | C of (... * t)
```

Value of `t` contains (sub)-value of **same type** `t`

Recursive Types

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

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

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

Recursive Types

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

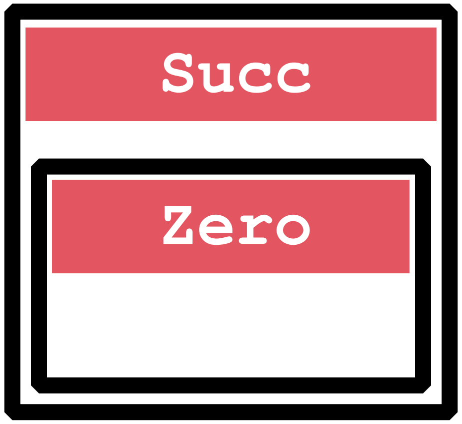
What are values of **`nat`** ?

Recursive Types

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

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

One **`nat`** contains another!

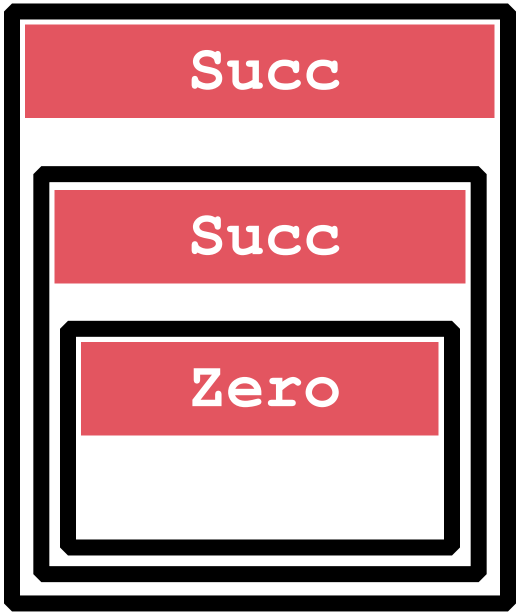


Recursive Types

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

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

One **`nat`** contains another!

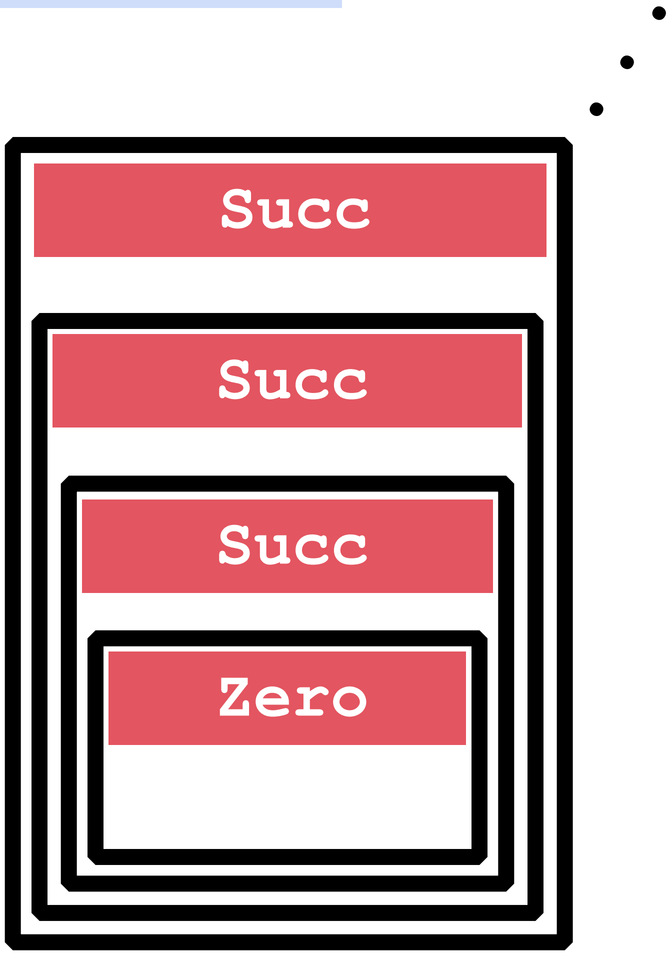


Recursive Types

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

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

One **`nat`** contains another!



3 Ways to Build Complex Values

Tuple (a.k.a. "Each-of", "Product") Type

```
type t = (t1 * t2)
```

Value of `t` contains value of `t1` **and** a value of `t2`

Data (a.k.a. "One-of", "Variant") Type

```
type t = C1 of t1 | C2 of t2
```

Value of `t` contains value of `t1` **or** a value of `t2`

Recursive Datatype

```
type t = ... | C of (... * t)
```

Value of `t` contains (sub)-value of **same type** `t`

CSE 130 [Winter 2014] Programming Languages

Recursion

Ravi Chugh

UCSDCSE

Computer Science and Engineering

3 Ways to Build Complex Values

Tuple (a.k.a. “Each-of”, “Product”) Type

```
type t = (t1 * t2)
```

Value of `t` contains value of `t1` **and** a value of `t2`

Data (a.k.a. “One-of”, “Variant”) Type

```
type t = C1 of t1 | C2 of t2
```

Value of `t` contains value of `t1` **or** a value of `t2`

Recursive Datatype

```
type t = ... | C of (... * t)
```

Value of `t` contains (sub)-value of **same type** `t`

Recursive Code Mirrors Recursive Data

of_int : int -> nat

Base Pattern

Inductive Pattern

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

Base Expression

Inductive Expression

```
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

Base Pattern

Inductive Pattern

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

Base Expression

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
```

Clicker Question

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

(a) **`Zero`**

(b) **`Succ Zero`**

(c) **`Succ (Succ Zero)`**

(d) **`Succ (Succ (Succ Zero))`**

(e) Type Error

plus: nat -> nat -> nat

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

Base Pattern: `Zero` — *Base Expression*: `m`

Inductive Pattern: `Succ n'` — *Inductive Expression*: `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

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

Base Expression

Base Pattern

Inductive Pattern

Inductive Expression

Try this out at home!

Recursive Code Mirrors Recursive Data

Lists are recursive types!

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

What are values of `int list` ?

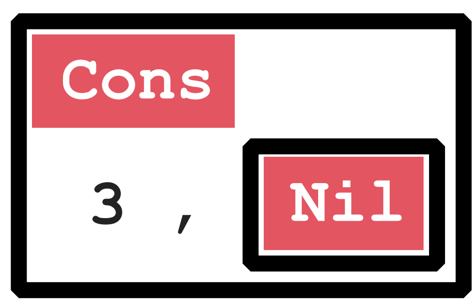
Nil

Lists are recursive types!

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

What are values of `int list` ?

Cons(3,Nil)   Nil

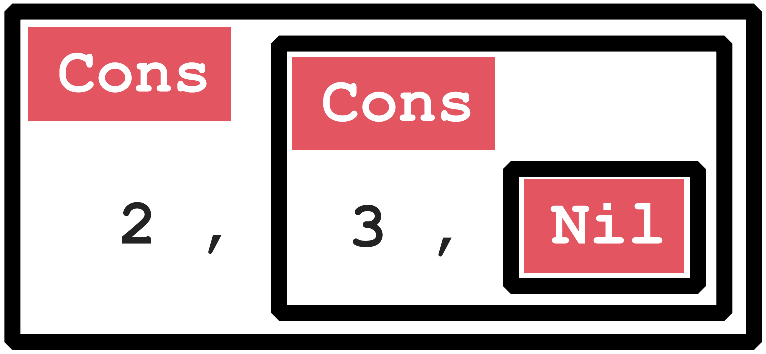


Lists are recursive types!

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

What are values of `int list` ?

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

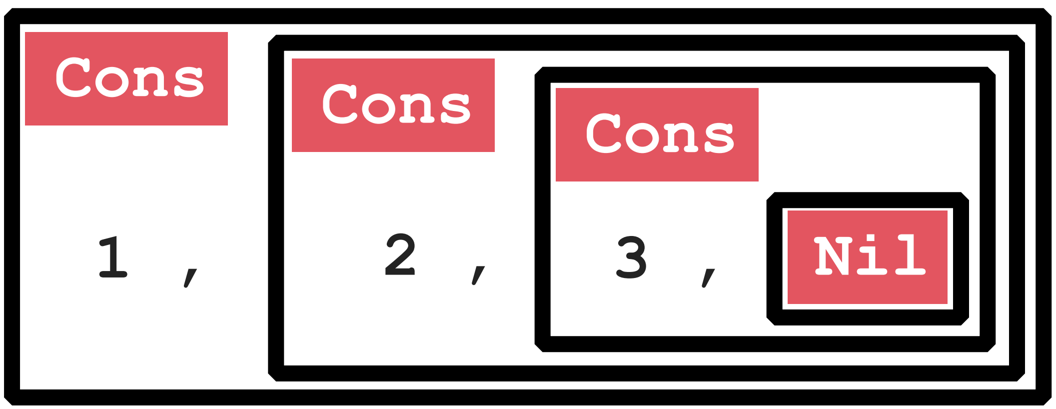


Lists are recursive types!

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

What are values of `int list` ?

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



Lists are not built-in!

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

Think about this!

Lists are a derived type, built using elegant core!

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

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

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

Recursive Code Mirrors Recursive Data

Clicker Question

```
let rec bar xs = match xs with
  | _::xs’ -> 1 + bar xs’
  | _      -> 0
in bar [10;20;30;40]
```

(a) Infinite Loop (Stack Overflow)

(b) `0`

(c) Runtime Error (Match Failure)

(d) `4`

(e) `100`

Some functions on Lists : `len`

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

Base Pattern — `[]`; *Base Expression* — `0`

Inductive Pattern — `h::t`; *Inductive Expression* — `1 + (len t)`

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

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

"_" matches any value, without binding to variable

pattern-matching in order, so last case must match `[]`

Some functions on Lists : len

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

```
                  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 l =
  match l with
  | []   -> 0
  | h::t -> h + (sum t)
```

Base Pattern

Inductive Pattern

Base Expression

Inductive Expression

```
sum [] ====> 0

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

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

Clicker Question

```
let rec baz x ys = match ys with
  | y::ys' -> (x=y) || baz x ys'
  | _      -> false
in bar 30 [10;20;30;40]
```

(a) Infinite Loop (Stack Overflow)

(b) **`false`**

(c) Type Error

(d) **`4`**

(e) **`true`**

Some functions on Lists : mem

```
(* 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 Pattern

Inductive Pattern

Base Expression

Inductive Expression

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

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

Some functions on Lists : mem

```
(* 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 Pattern | Base Expression

Inductive Pattern | Inductive Expression

```
mem 4 [] ====> false

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

Some functions on Lists : mem

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

Well-designed datatype gives strategy

Some functions on Lists : mem

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

Some functions on Lists : mem

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

```
(* val append : 'a list -> 'a list -> 'a list *)
let rec append xs ys =
```

Some functions on Lists : mem

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

```
(* val append : 'a list -> 'a list -> 'a list *)
let rec append xs ys =
```

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

Try these at home!

Remember `hd` and `tl` ?

These functions crash when applied to `[]`

- Bad ML style (more than just aesthetics!)

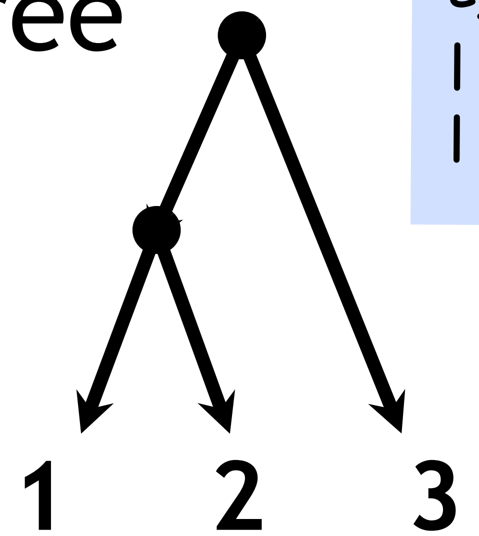
Pattern-matching better than test-extract:

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

Recursive Code Mirrors Recursive Data

Clicker Question

How is this tree represented?




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

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

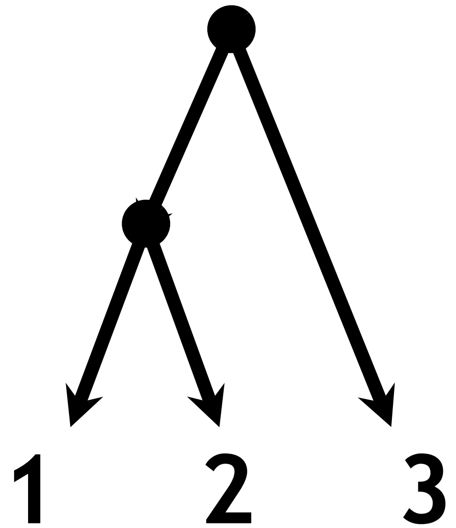
(b) `((Leaf 1, Leaf 2), Leaf 3)`

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

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

(e) None of the above

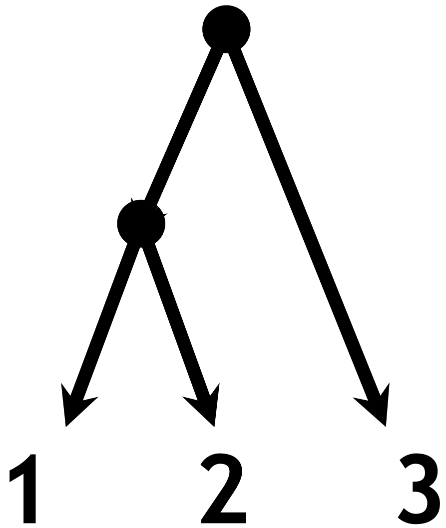
Representing Trees




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

`Leaf 1`

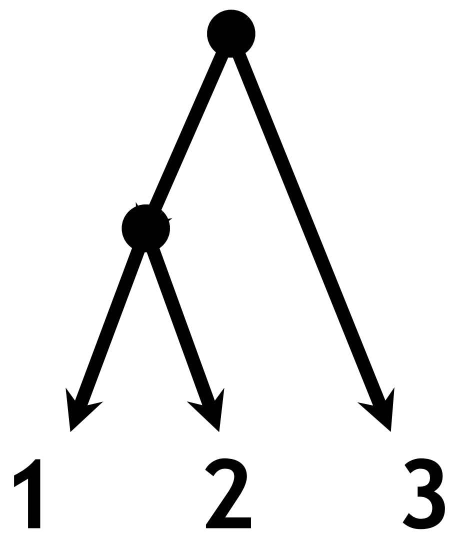
Representing Trees

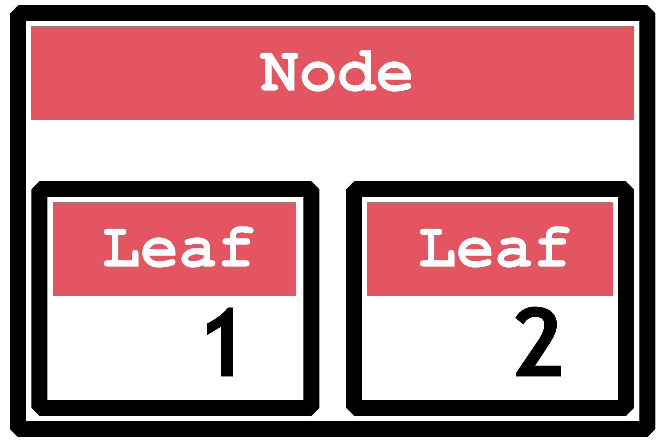



```
type tree =
| Leaf of int
| Node of tree*tree
```

`Leaf 2`

Representing Trees

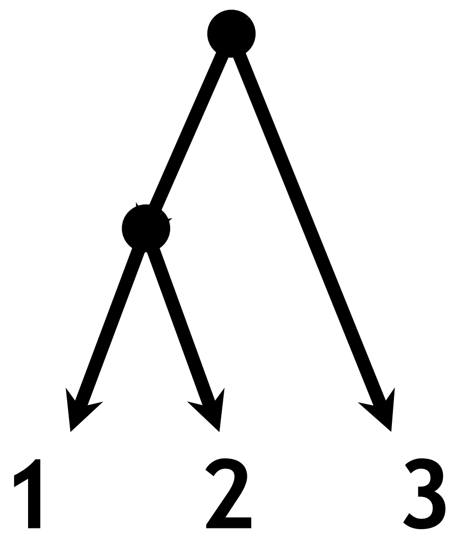





```
type tree =
| Leaf of int
| Node of tree*tree
```

`Node(Leaf 1, Leaf 2)`

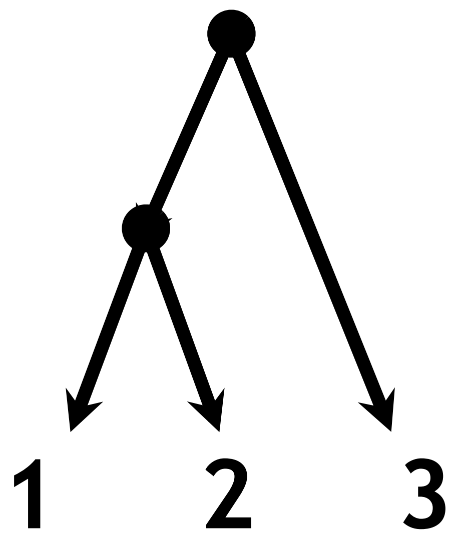
Representing Trees

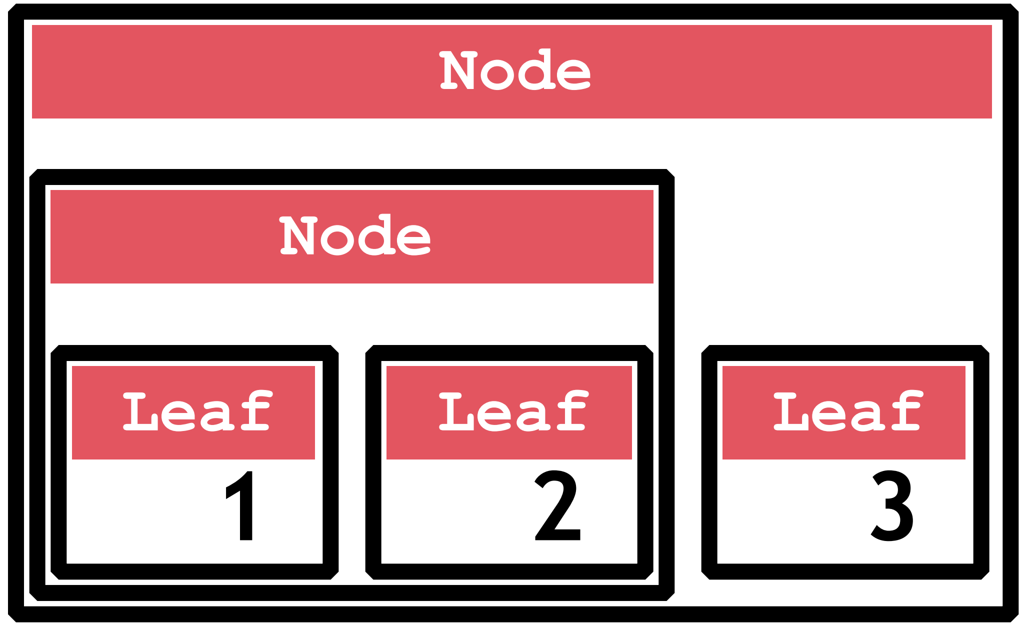



```
type tree =
| Leaf of int
| Node of tree*tree
```

Leaf 3

Representing Trees






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

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

Recursive Code Mirrors Recursive Data

Clicker Question

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

```
# 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
```

Base Pattern (Leaf)
Inductive Pattern (Node)

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

Base Pattern (Leaf n)
Inductive Pattern (Node(t1,t2))

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

Recursive Code Mirrors Recursive Data

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*

What's an 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*

What's an ML **TYPE** for **REPRESENTING** expressions ?

What's an 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*

What's an ML **TYPE** for **REPRESENTING** expressions ?

What's an 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
```

Recursion

- A way of life
- A different way to view computation

  Solutions for bigger problems

  from solutions for sub-problems

Why know about it ?

1. Often far simpler, cleaner than loops
   - But not always…
2. Forces you to factor code into reusable units
   - Only way to "reuse" loop is via cut-paste

Clicker Question

```
let rec foo i j =
  if i >= j then []
  else i::(foo (i+1) j)
in foo 0 3
```

(a) `[0;1;2]`

(b) `[0;0;0]`

(c) `[]`

(d) `[2;2;2]`

(e) `[2;1;0]`

Recursion

```
let rec range i j =
  if i >= j then []
  else i::(range (i+1) j)
```

```
range 3 3  ====> []
range 2 3  ====> 2::(range 3 3) ====> 2::[]
range 1 3  ====> 1::(range 2 3) ====> 1::2::[]
range 0 3  ====> 0::(range 1 3) ====> 0::1::2::[]
```

Tail Recursion

- A function is "**tail recursive**" if:
  - all recursive calls are immediately followed by return
  - that is, each recursive call is in "tail position"
  - so cannot do **anything** between call and return

Tail Recursion

```
let rec range i j =
  if i >= j then []
  else i::(range (i+1) j)
```

Tail Recursive?

Tail Recursion

```
let range lo hi =
  let rec helper res j =
    if lo >= j then res
    else helper (j::res) (j-1)
  in helper [] hi
```

Tail Recursive!

Tail Recursion

- A function is "**tail recursive**" if:
  - all recursive calls are immediately followed by return
  - that is, each recursive call is in "tail position"
  - so cannot do **anything** between call and return

- Why do we care?
  - Compiler can transform recursion into a loop
  - **You** write readable code
  - **Compiler** optimizes into fast code!

Tail Recursion

```
let fact n =
  let rec helper res j =
    if j <= 1
    then res
    else helper (j*res) (j-1)
  in
    helper 1 n
```

```
function fact(n) {
  var res := 1;
  while (true) {
    if (n <= 0)
    then { return res; }
    else { n := n-1; res := (n*res); }
  }
}
```