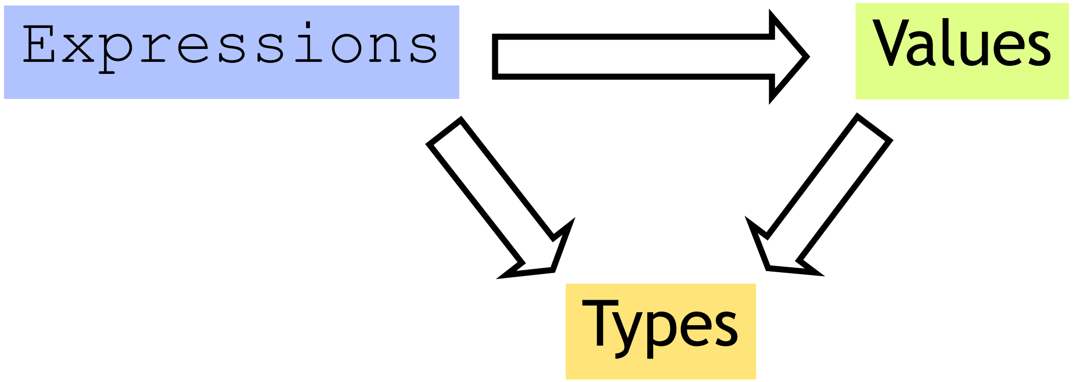
CSE 130
Programming Languages

Datatypes

UCSD

Review so far



Many kinds of expressions:

1. Simple
2. Variables
3. Functions

Review so far

- 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

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)
- …

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 ?

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`

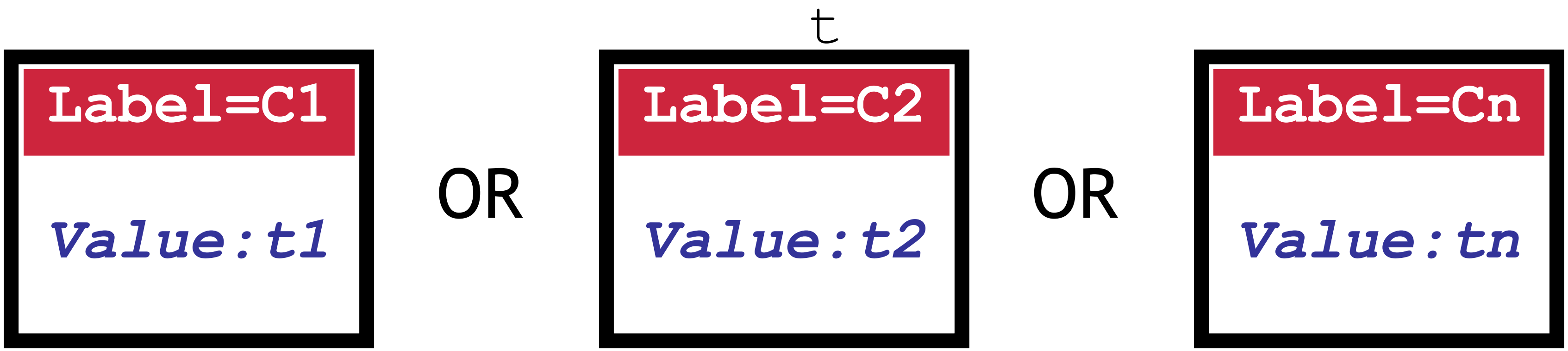
Or a value of type `t2` placed in a box labeled `C2`

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**

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;;
```

How to PUT values into box?

How to PUT values into box?

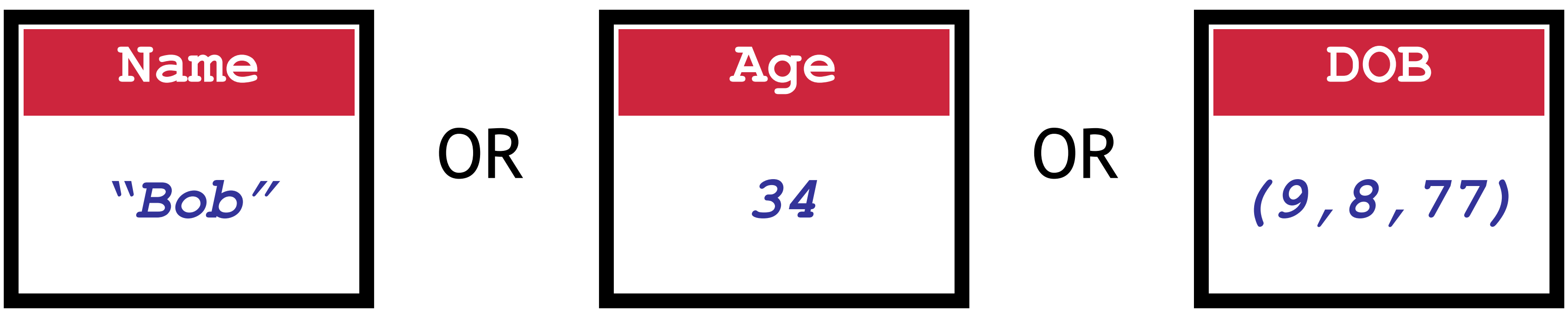
How to create values of type `attrib` ?

```
# let a1 = Name "Bob";;
val x : attrib = Name "Bob"
# 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 "Bob"`   `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...

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!

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

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(a,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 (Name “Bob”) with
   | Name s -> printf "Hello %s\n" s
   | Age i  -> printf "%d years old" i
   ;;
Hello Bob
- : unit = ()
```

None of the cases matched the tag (Name)

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

How to TEST & TAKE whats in box?

BEWARE!!
Be sure to handle all TAGS!

Beware! Handle All TAGS!

```
# match (Name “Bob”) 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!

```
# match (Name “Bob”) 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!

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

Another Few Examples

```
# 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.
```

See code text file

`match-with` is an Expression

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

Type Rule

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

`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

T

Type Rule

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

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

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
```

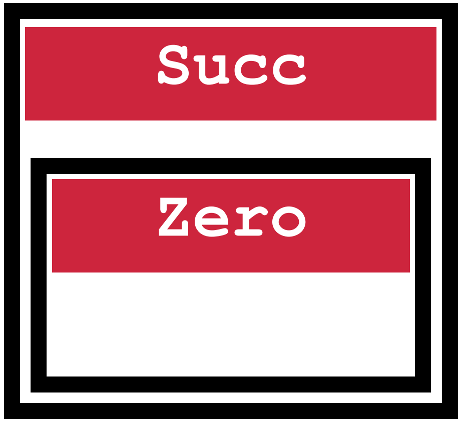
What are values of **`nat`** ?

"Recursive" types

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

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

One **`nat`** contains another!

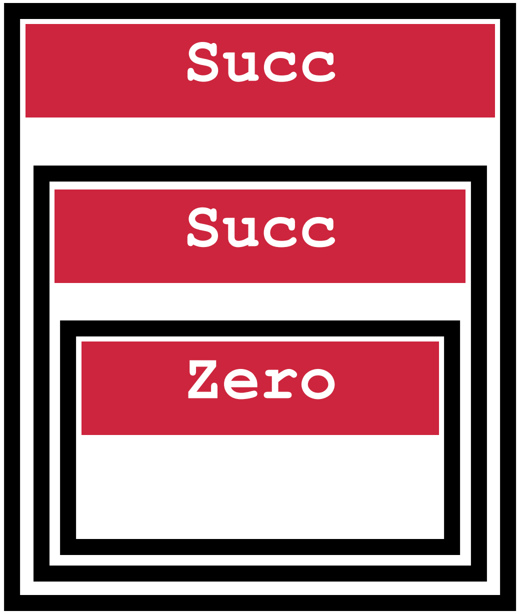


"Recursive" types

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

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

One **`nat`** contains another!

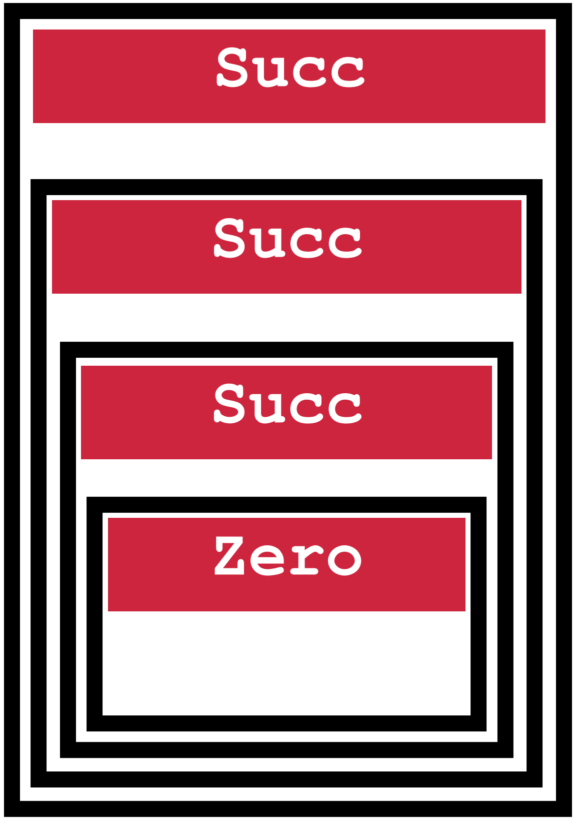


“Recursive” types

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

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

One **`nat`** contains another!



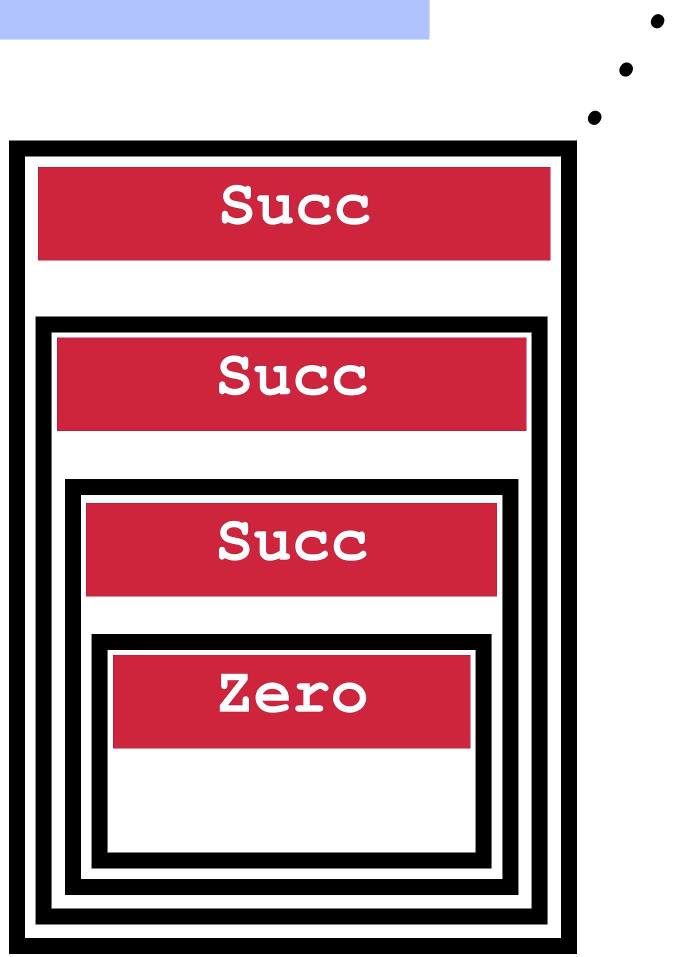
"Recursive" types

```
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

Base pattern

Inductive pattern

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

```
let rec to_int n =
```

to_int : nat -> int

Base pattern
Inductive pattern

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

Base pattern
Inductive pattern

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

Base Expression
Inductive Expression

of_int : int -> nat

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

```
let rec of_int n =
```

of_int : int -> nat

Base pattern

Inductive pattern

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

```
let rec of_int n =
```

of_int : int -> nat

Base pattern

Inductive pattern

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

Base pattern

Inductive pattern

```
let rec of_int n =
  if n <= 0 then

  else
```

of_int : int -> nat

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

Base pattern: `| Zero`

Inductive pattern: `| Succ`

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

Base pattern: `if n <= 0`

Base Expression: `Zero`

Inductive pattern: `else`

Inductive Expression: `Succ (of_int (n-1))`

plus : nat*nat -> nat

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

```
let rec plus n m =
```

plus : nat*nat -> nat

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

Base pattern (circled: `| Zero`)

Inductive pattern (circled: `| Succ`)

```
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

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

Base Expression
Inductive Expression

times: nat*nat -> nat

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

```
let rec times n m =
```

times: nat*nat -> nat

Base pattern

Inductive pattern

```
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 =
match m with
| Zero     ->
| Succ m'  ->
```

Base pattern: `Zero`

Inductive pattern: `Succ 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     -> 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’)`

Next: Lets get cosy with Recursion

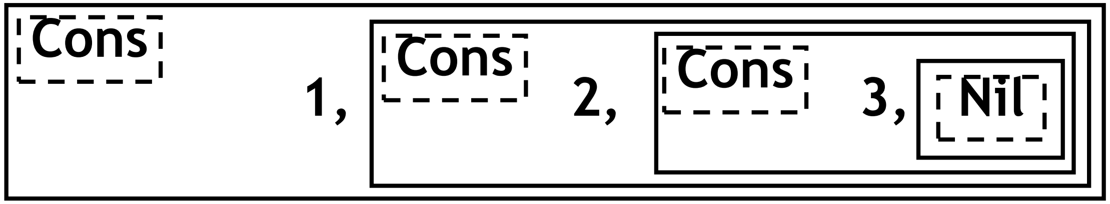
Recursive Code Mirrors Recursive Data

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

Inductive pattern — Cons(h,t)

Base Expression — 0

Inductive Expression — 1 + (len t)

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

No binding for head

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

Pattern-matching in order

Some functions on Lists : Append

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

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

Well designed datatype gives strategy

Some functions on Lists : Max

```
let rec max xs =
```

- 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 !

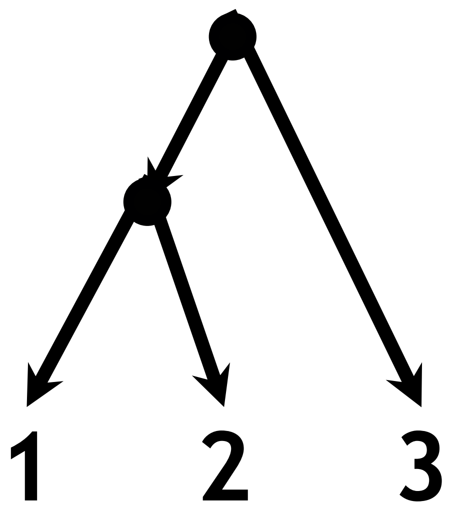
Pattern-matching better than test-extract:

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

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

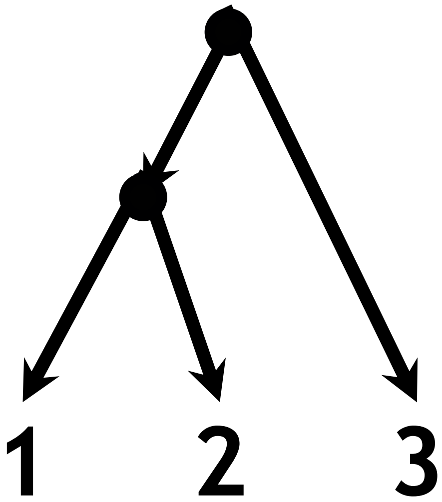
Representing Trees




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

Leaf 1

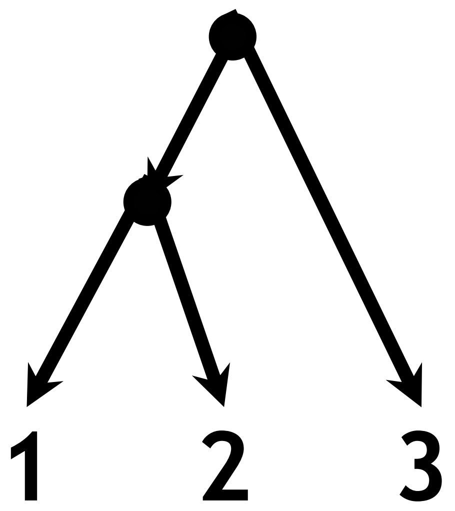
Representing Trees

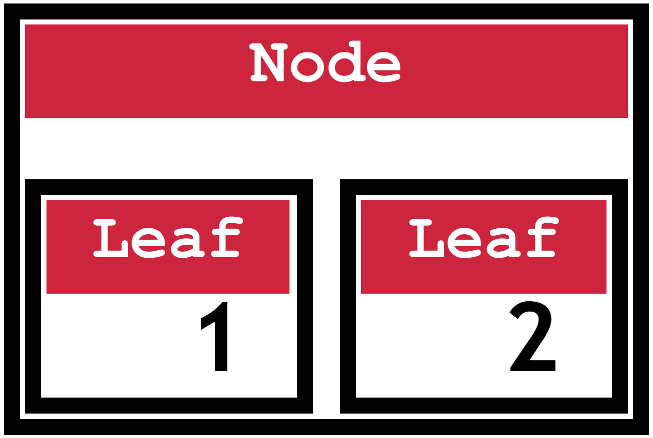



```
type tree =
| Leaf of int
| Node of tree*tree
```

`Leaf 2`

Representing Trees

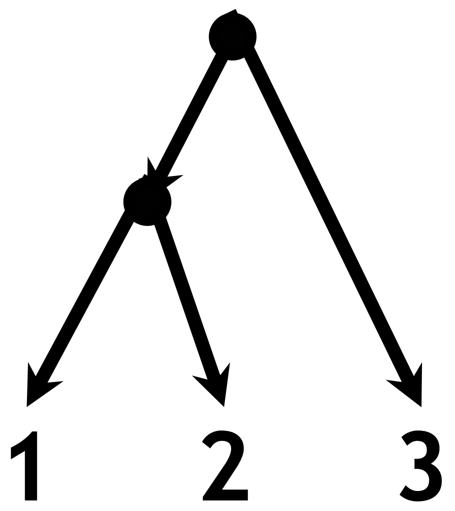





```
type tree =
| Leaf of int
| Node of tree*tree
```

`Node(Leaf 1, Leaf 2)`

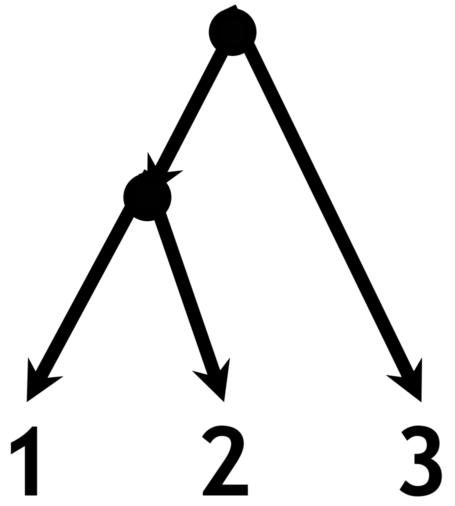
Representing Trees

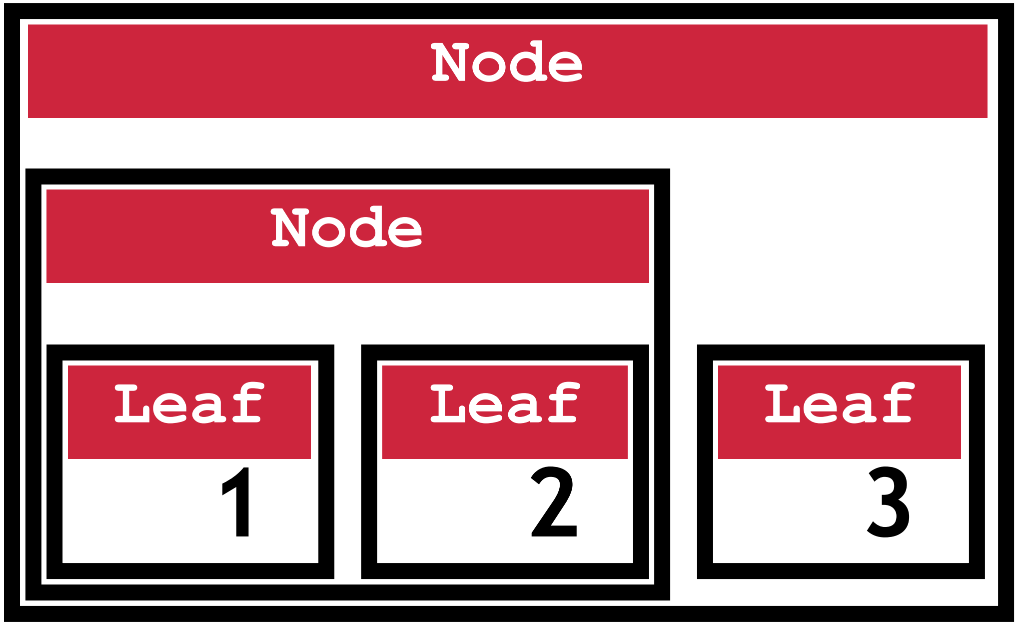



```
type tree =
| Leaf of int
| Node of tree*tree
```

Leaf 3

Representing Trees






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

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

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);;
- : 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 (circles "| Leaf")

Inductive pattern (circles "| 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

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

Base Expression
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 ====> ***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
```

```
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
```