CSE 130 [Winter 2014] Programming Languages

Datatypes

Ravi Chugh

UCSDCSE
Computer Science and Engineering

Jan 16

What About More Complex Data?

- We've seen some base types and values:
  - Integers, Floats, Bool, String, etc.
- Some ways to build up types:
  - Tuples (products) and "lists"
  - Functions
  - Records (we will see in a few weeks)
- Design Principle: Orthogonality
  - Don't clutter core language with stuff
  - Few, powerful orthogonal building techniques
  - Put "derived" types, values, functions in libraries

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

Recursive Datatype

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

Supposed 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
- Can have multiple names, phones, emails, etc.
- Want to store them in a list. Can I?

Supposed 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;;
```

Clicker Question

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

What is the result of

`Name "Tony Stark"` ?

(a) Syntax Error
(b) Type Error
(c) `string`
(d) `attrib`
(e) `'a`

Clicker Question

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

What is the result of

```
Name "Tony" ^ Name "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**

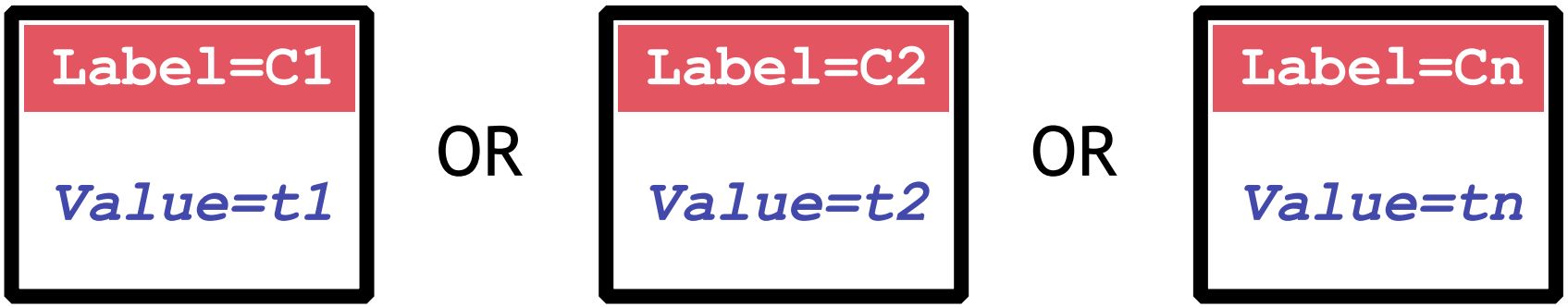
  Or a value of type ***t2*** placed in a box labeled **C2**

  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`

Clicker Question

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

What is the result of

```
Age "Tony Stark" ?
```

(a) Syntax Error
(b) Type Error
(c) `string`
(d) `attrib`
(e) `'a`

How to PUT values into box?

How to PUT values into box?

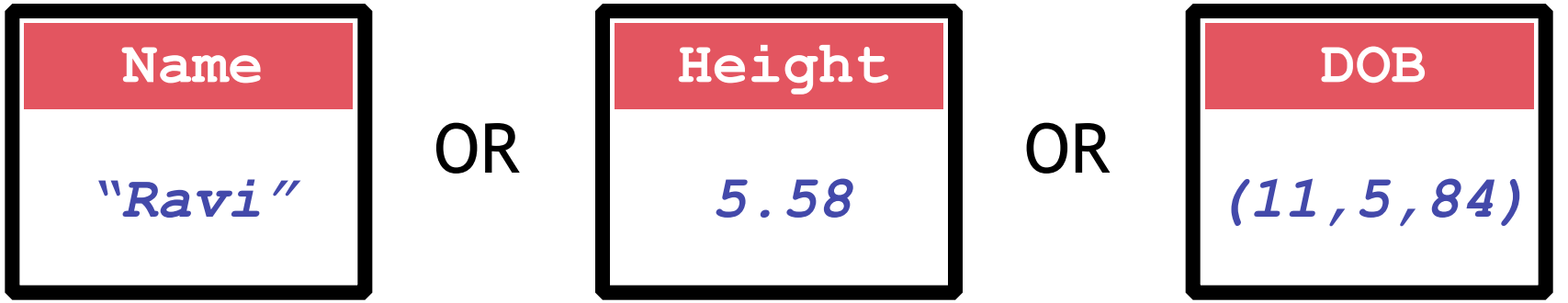
How to create values of type `attrib` ?

```
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;;
```

```
# let a1 = Name "Ravi" ;;
val a1 : attrib = Name "Ravi"
# let a2 = Height 5.58 ;;
val a2 : attrib = Height 5.58
# let year = 1984 ;;
val year : int = 1984
# let a3 = DOB (11,5,year) ;;
val a3 : attrib = DOB (11,5,1984)
# let attrs = [a1;a2;a3] ;;
val attrs : attrib list = …
```

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 "Ravi"
```

```
Height 5.58
```

```
DOB (11,5,84)
```

All have type `attrib`

Clicker Question

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

What is the result of

```
[Name "Ravi"; Height 5.58; DOB(11,5,84)]
```

?

(a) Syntax Error

(b) Type Error

(c) `attrib list`

(d) `(string*float*(int*int*int)) list`

(e) `'a list`

“One-of” Types

- We’ve defined a “one-of” type named `attrib`
- Elements are one of:
  - `string`
  - `int`
  - `int*int*int`
  - `float`
  - `bool` …

```
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;;
```

- Can create uniform `attrib` lists
- Say I want a function to print attributes …

How to TEST & GET what's in box?

Is it a ...

`string?`

or an

`int?`

or an

`int*int*int?`

or ...

How to TEST & GET what's in box?

Look at TAG!

Clicker Question

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

What is the result of

```
let welcome a = match a with
                    | Name s -> s
in welcome (Name “Ravi”)
```

?

(a) Type Error

(b) `Name “Ravi” : 'a`

(c) `Name “Ravi” : attrib`

(d) `“Ravi” : string`

(e) Run-time Error

How to TEST & GET what's in 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: float *)
| Alive b      -> … (*b: bool *)
| Phone(y,r)   -> … (*a: int, r: int *)
```

Pattern-match expression

- Simultaneously test and extract contents of box

If `e` matches the pattern form, then:

- value in box bound to pattern variable
- matching result expression is evaluated

Else: try next pattern

How to TEST & GET what's in box?

```
match e with
| Name s       -> printf ”Hello %s\n" 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
```

How to TEST & GET what's in box?

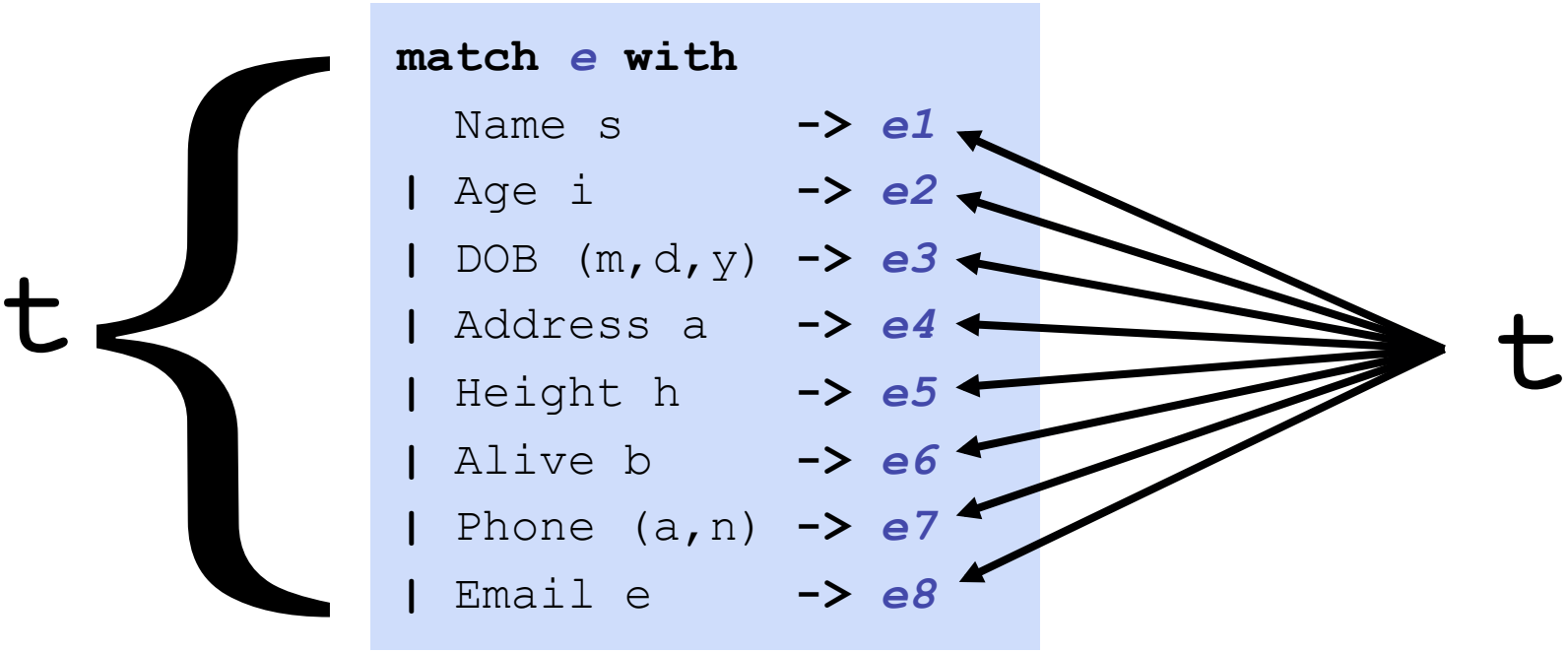
```
match (Name "Ravi") with
  | Name s       -> printf "Hello %s\n" s
  | Age i        -> printf "%d" i
  | DOB(d,m,y)   -> printf "%d/%d/%d" d m y
  | Address s    -> printf "%s" s
  | Height h     -> printf "%f" h
  | Alive b      -> printf "%b" b s
  | Phone(a,r)   -> printf "(%d)-%d" a r
```

```
Hello Ravi
- : unit = ()
```

First case matches the tag (Name)

Evals branch with s "bound" to string contents

`match-with` is an Expression



Type Rule

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

Clicker Question

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

What is the result of

```
let welcome a = match a with
                    | Name s -> s
in welcome (Age 29)
```

?

(a) Type Error

(b) Name “Ravi” : ’a

(c) Name “Ravi” : attrib

(d) “Ravi” : string

(e) Run-time Error

How to TEST & GET what's in box?

BEWARE!

Be sure to handle all TAGS!

Beware! Handle All Tags!

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

Exception: Match Failure!!

None of the cases matched the tag (Name), so crash with nasty **Run-Time Error**

Beware! Handle All Tags!

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

Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Phone (_, _)

Exception: Match Failure!!
```

ML gives a **compile-time** warning about missing cases!

Clicker Question

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

What is the result of

```
let welcome a = match a with
  | Name s -> “Hello, ” ^ s ^ “! ”
  | Name s -> “Hi, ”    ^ s ^ “! ”
in welcome (Name “Ravi”)
```

?

(a) Type Error

(b) `“Hello, Ravi! ”` `: string`

(c) `“Hi, Ravi! ”` `: string`

(d) `“Hello, Ravi! Hi, Ravi! ”` `: string`

(e) 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
    ...
    | Age i        -> Printf.printf "%d" i
;;

Warning U: this match case is unused.
val printAttrib : attrib -> unit = <fun>
```

ML gives a **compile-time** warning about **redundant** cases (which will never match)!

Clicker Question

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

What is the result of

```
let welcome a = match a with
                    | Name s -> s
                    | Age i  -> i
in welcome (Name “Ravi”)
```

?

(a) Type Error

(b) `Name “Ravi” : attrib`

(d) `“Ravi” : string`

(e) Run-time Error

Clicker Question

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

What is the result of

```
let welcome a = match a with
                    | Name s -> a
                    | Age i  -> a
in welcome (Name “Ravi”)
```

?

(a) Type Error

(b) `Name “Ravi” : attrib`

(d) `“Ravi” : string`

(e) Run-time Error

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
```

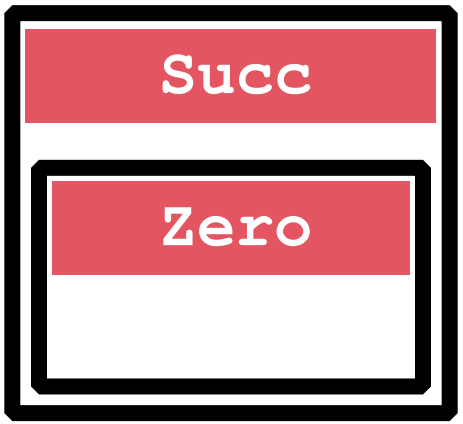
What are values of **`nat`** ?

Recursive Types

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

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

One **`nat`** contains another!

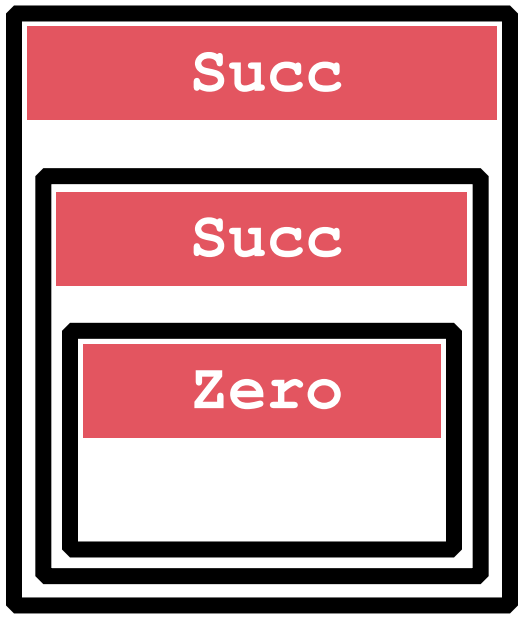


Recursive Types

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

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

One **`nat`** contains another!

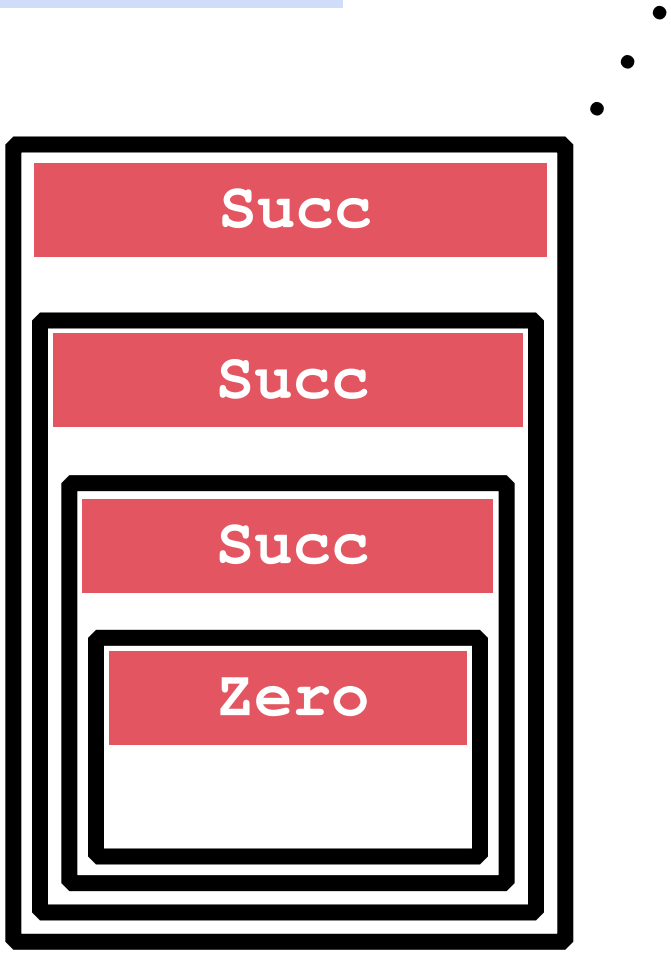


Recursive Types

```
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

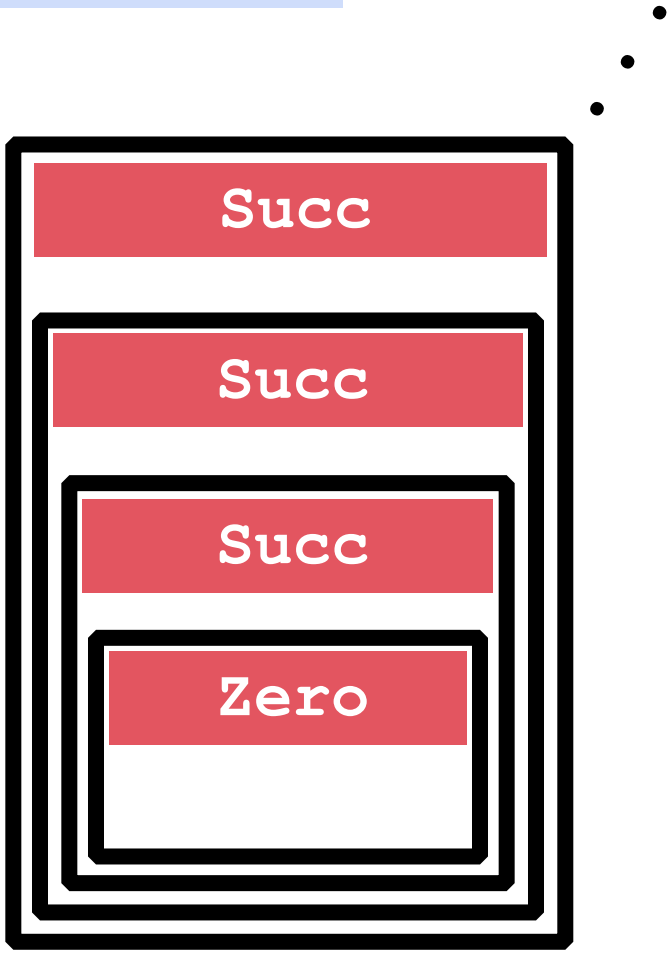
UCSDCSE
Computer Science and Engineering

Recursive Types

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

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

One **`nat`** contains another!



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

Base Expression

Inductive Pattern

Inductive Expression

```
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 Pattern

Inductive Pattern

Base Expression

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` ?

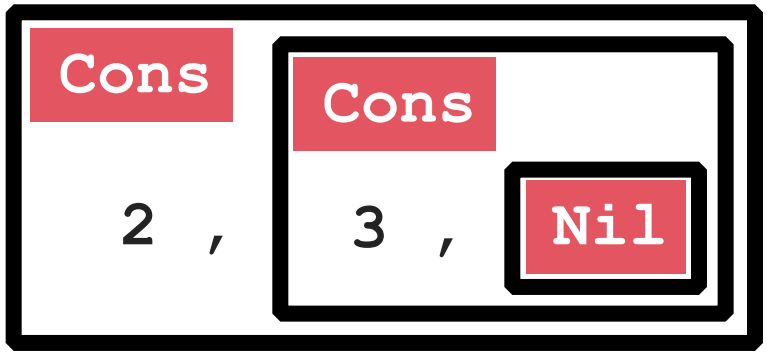
Cons(3,Nil) Nil

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

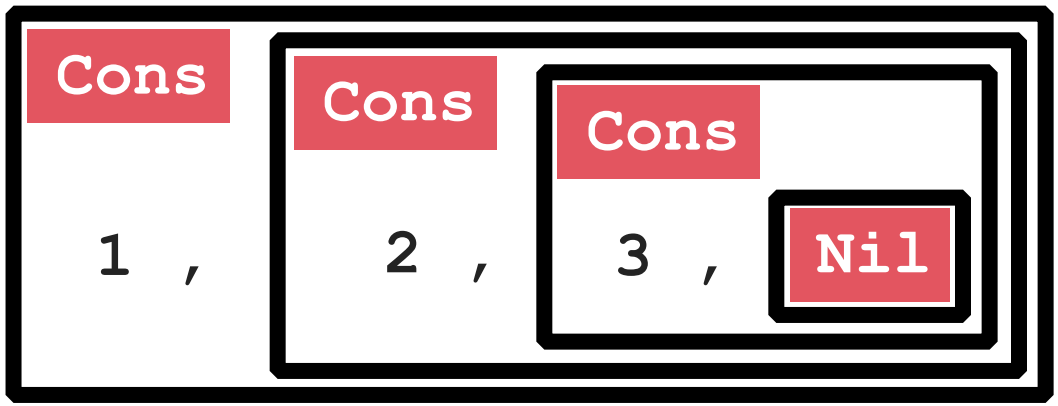


Lists are recursive types!

```
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*

Inductive Pattern

Inductive Expression

```
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 — *Base Expression*

Inductive Pattern — *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

Base Expression

Inductive Pattern

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
```

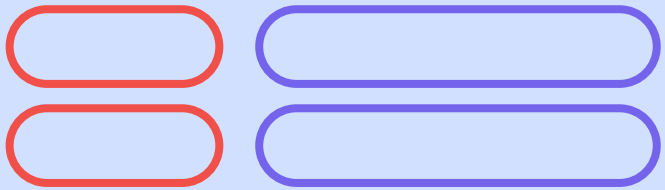
Some functions on Lists : mem

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

Well-designed datatype gives strategy

Some functions on Lists : mem

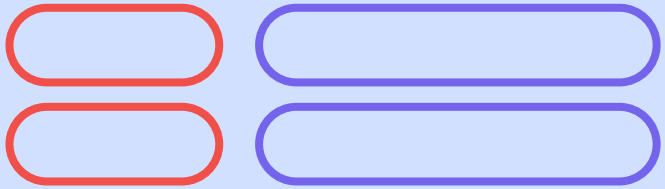
```
(* val insert : 'a -> 'a list -> 'a list *)
let rec insert x xs =
```



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 =
```

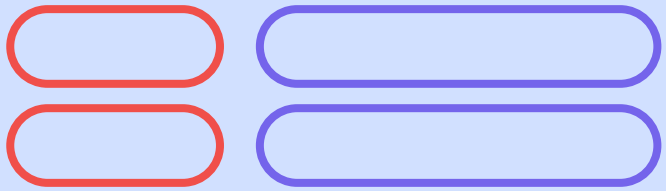


Some functions on Lists : mem

```
(* 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