Lecture 3: Datatypes

Ranjit Jhala
UC San Diego
News?

- PA #2 *(coming tonight)*
- Ocaml-top issues?
- Please post questions to Piazza
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
Variables and bindings

```
let x = e;
```

“Bind the value of expression e to the variable x”

```
# let x = 2+2;;
val x : int = 4
```
Local bindings

... for expressions using “temporary” variables

let tempVar = x + 2 * y
in tempVar * tempVar

• tempVar is bound only inside expr body from in ...
• Not visible (“not in scope”) outside
Binding by Pattern-Matching

Simultaneously bind several variables

```ocaml
# let (x, y, z) = (2+3, "a"^"b", 1::[2]);;

val x : int = 5
val y : string = "ab"
val z : int list = [1;2]
```
Binding by Pattern-Matching

But what of:

```
# let h::t = [1;2;3];;
Warning P: this pattern-matching not exhaustive.
val h : int = 1
val t : int list = [2;3]
```

Why is it whining?

```
# let h::t = [];
Exception: Match_failure
# let XS = [1;2;3];
val xs = [1;2;3]: list
- val h::t = xs;
Warning: Binding not exhaustive
val h = 1 : int
val t = [2;3] : int
```

In general $XS$ may be empty (match failure!)

Another useful early warning
Functions
Next class: 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

```ocaml
# 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” ...then use

this pattern... ...this Body Expr

```ocaml
- 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 list = 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

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

# 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 ...

```ml
# 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))
```

Now, let's 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 \((T1 \times T2)\)
   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)

```haskell
import Data.Type

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;
```
Question: Here is a typedef ...

```plaintext
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

definition

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
- a value of type t2 placed in a box labeled C2
- ...
- a value of type tn placed in a box labeled Cn
Constructing Datatypes

\[ \text{type } t = C_1 \text{ of } t_1 \mid C_2 \text{ of } t_2 \mid \ldots \mid C_n \text{ of } t_n \]

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?

```ocaml
# 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 = ...
```

```ocaml
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

```markdown
**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 “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 ...

- Can create uniform `attrib` lists

- Say I want a function to print attribs...

```python
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;
```
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 what's in box?

Is it a ...
string?
or an int?
or an int*int*int?
or ...
How to TEST & TAKE what's in the 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 what's in the box?

```plaintext
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 what's in the box?

```haskell
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
    | Age s -> s

in welcome (Age 34)

(a) \text{Name} "Ranjit" : \text{`a}
(b) Type Error
(c) \text{Name} "Ranjit" : attrib
(d) "Ranjit" : \text{string}
(e) Runtime Error
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!

None of the cases matched the tag (Name)
Causes nasty **Run-Time Error**
Compiler To The Rescue!!

```ocaml
# 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?

```ocaml
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" : 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`

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

Type Rule

- \( e_1, e_2, \ldots, e_n \) 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 \( t_1 \times t_2 \)
Value of \( T \) contains value of \( T_1 \) and a value of \( T_2 \)

2. “One-of” types \( \text{type } t = C_1 \text{ of } t_1 | C_2 \text{ of } t_2 \)
Value of \( T \) contains value of \( T_1 \) or a value of \( T_2 \)

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

define 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

\[
\text{type } \texttt{nat} = \texttt{Zero} \mid \texttt{Succ} \text{ of } \texttt{nat}
\]

What are values of \texttt{nat}?

One \texttt{nat} contains another!
“Recursive” types

type nat = Zero | Succ of nat

What are values of nat?
One nat contains another!
“Recursive” types

\[
\text{type } \text{nat} = \text{Zero} \mid \text{Succ of nat}
\]

What are values of \text{nat}?

One \text{nat} contains another!

\text{nat} = \text{recursive type}
Next: Building datatypes

Three key ways to build complex types/values

1. “Each-of” types \( t_1 \times t_2 \)
   Value of \( T \) contains value of \( T_1 \) \textcolor{blue}{\textbf{and}} a value of \( T_2 \)

2. “One-of” types \textbf{type}\ \( t = C_1 \text{ of } t_1 \mid C_2 \text{ of } t_2 \)
   Value of \( T \) contains value of \( T_1 \) \textcolor{blue}{\textbf{or}} a value of \( T_2 \)

3. “Recursive” type \textbf{type}\ \( t = \ldots | C \text{ of } (\ldots \ast t) \)
   Value of \( T \) contains \textcolor{blue}{\textbf{(sub)-value of same type}} \( T \)
Next: Let's get cosy with Recursion

Recursive Code Mirrors Recursive Data
Next: Let's get cosy with Recursion

Code Structure = Type Structure!!!
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

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

let rec of_int n =
```
of_int : int -> nat

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

let rec of_int n =
if n <= 0 then
else
```
of_int : int -> nat

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

type nat =
| Zero
| Succ of nat
to_int : nat -> int

\[
\text{type nat =}
| \text{Zero}
| \text{Succ of nat}
\]

let rec to_int n =
to_int : nat -> int

let rec to_int n =

type nat =
| Zero
| Succ of nat
to_int : nat -> int

<table>
<thead>
<tr>
<th>Base pattern</th>
<th>Inductive pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>type nat =</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zero</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Base pattern</th>
<th>Inductive pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>let rec to_int n = match n with</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zero -&gt;</td>
</tr>
</tbody>
</table>
to_int : nat -> int

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

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

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

```
let rec plus (n,m) =
```
plus : nat*nat -> nat

\[
\begin{align*}
type \ \text{nat} &= \\
| \ \text{Zero} & \\
| \ \text{Succ} \ \text{of} \ \text{nat}
\end{align*}
\]

let rec plus (n,m) =
plus : nat*nat -> nat

type nat =  
  | Zero  
  | Succ of nat  

let rec plus (n,m) =  
  match m with  
  | Zero ->  
  | Succ m' ->
plus : nat*nat -> nat

type nat =
| Zero
| Succ of nat

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

let rec times (n,m) =
\textbf{times:} \texttt{nat*nat -> nat}

\begin{itemize}
\item \textbf{Type:}
\begin{verbatim}
type nat =
| Zero
| Succ of nat
\end{verbatim}
\item \textbf{Inductive Pattern:}
\begin{verbatim}
let rec times (n,m) =
match m with
| Zero    ->
| Succ m' ->
\end{verbatim}
\end{itemize}
\textit{plus : nat*nat -> nat}

\begin{align*}
\text{type } \text{nat} &= \text{Zero} \\
&\lor \text{Succ of nat} \\
\text{let rec times (n,m) =} \\
&\text{match m with} \\
&\quad | \text{Zero} \rightarrow \text{Zero} \\
&\quad | \text{Succ m'} \rightarrow \text{plus n (times (n,m'))}
\end{align*}
minus : nat*nat -> nat

**type nat =**
| Zero |
| Succ of nat |

**let rec minus (n,m) =**
times: nat*nat -> nat

\[
\text{type } \text{n} \text{at =}
\begin{align*}
& \mid \text{Zero} \\
& \mid \text{Suc} \text{c of nat}
\end{align*}
\]

let rec minus (n,m) =
times: nat*nat -> nat

type nat =
| Zero
| Succ of nat

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

Base pattern
Inductive pattern

Base pattern
Inductive pattern
times: nat*nat -> nat

\[
\text{type } \text{nat } = \\
| \text{Zero} \\
| \text{Succ of nat}
\]

let rec minus (n,m) = 
match (n, m) with 
| (\_, \text{Zero}) -> n \\
| (\text{Succ n'}, \text{Succ m'}) -> \text{minus(n',m')}
Next: Let's get cosy with Recursion

Recursive Code Mirrors Recursive Data
Lists are recursive types!

\[
\text{type } \text{int}_\text{list} = \\
\quad \text{Nil} \\
\mid \text{Cons} \text{ of } \text{int} \times \text{int}_\text{list}
\]

Think about this! What are values of \text{int}_\text{list}?

\begin{align*}
\text{Cons}(1, \text{Cons}(2, \text{Cons}(3, \text{Nil}))) & \quad \text{Cons}(2, \text{Cons}(3, \text{Nil})) \quad \text{Cons}(3, \text{Nil}) \quad \text{Nil} \\
\end{align*}
Lists aren’t built-in!

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

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

- **Base pattern**: Nil
- **Ind pattern**: Cons(_, t)
- **Base expression**: `0`
- **Inductive expression**: `1 + (len t)`

**Matches everything, no binding**

**Pattern-matching in order**
- Must match with `Nil`
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: \texttt{Max}

\begin{align*}
\text{let rec } \texttt{max} \; \texttt{xs} =
\end{align*}

- \textbf{Base pattern} + \textbf{Base expression}
- \textbf{Ind pattern} + \textbf{Inductive expression}

\begin{itemize}
\item Find the right \textbf{induction} strategy
  \begin{itemize}
  \item \textbf{Base case}: pattern + expression
  \item \textbf{Induction case}: pattern + expression
  \end{itemize}
\end{itemize}

\textbf{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: Let's get cozy with Recursion

Recursive Code Mirrors Recursive Data
Q: How is this tree represented?

1
2
3

(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            Leaf 1
| Node of tree*tree
```
Representing Trees

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

Leaf 2
Representing Trees

\[
\text{type } \text{tree} = \begin{cases} \\
\text{Leaf of int} \\
\text{Node of tree*tree} \\
\end{cases}
\]

Node(Leaf 1, Leaf 2)
Representing Trees

```haskell
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: Let's get cosy with Recursion

Recursive Code Mirrors Recursive Data
Q: What does this evaluate to?

```ocaml
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”. E.g.

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

```ocaml
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

let rec sum_leaf t =
sum_leaf : tree -> int

\[
\text{type } \text{tree} = \\
| \text{Leaf} \text{ of } \text{int} \\
| \text{Node} \text{ of } \text{tree*tree}
\]

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

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

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

**Base pattern**
- **Leaf**
- **Node**

**Inductive pattern**
- Base pattern
- Inductive pattern
Recursive Code Mirrors Recursive Data

Code almost writes itself!
Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- \(4.0 + 2.9\)
- \(3.78 - 5.92\)
- \((4.0 + 2.9) \times (3.78 - 5.92)\)
Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- $4.0 + 2.9 \rightarrow 6.9$
- $3.78 - 5.92 \rightarrow -2.14$
- $(4.0 + 2.9) \times (3.78 - 5.92) \rightarrow -14.766$

Whats a ML TYPE for REPRESENTING expressions?
Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- \(4.0 + 2.9 \rightarrow 6.9\)
- \(3.78 - 5.92 \rightarrow -2.14\)
- \((4.0 + 2.9) \times (3.78 - 5.92) \rightarrow -14.766\)

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

```ml
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:

• \(4.0 + 2.9 \Rightarrow 6.9\)
• \(3.78 - 5.92 \Rightarrow -2.14\)
• \((4.0 + 2.9) \times (3.78 - 5.92) \Rightarrow -14.766\)

What's 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:

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

What's 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:

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

What's a ML FUNCTION for EVALUATING expressions?

```plaintext
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:

- \[4.0 + 2.9 \Rightarrow 6.9\]
- \[3.78 - 5.92 \Rightarrow -2.14\]
- \[(4.0 + 2.9) \times (3.78 - 5.92) \Rightarrow -14.766\]

What's a ML FUNCTION for EVALUATING expressions?

```ml
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
```
Random Art from Expressions

PA #2

Build more funky expressions, evaluate them, to produce: