Story So Far...

- Simple Expressions
- Branches
- Let-Bindings ...

Today:
- Finish Crash Course
- Datatypes
Next class: functions, but remember ...

Expression ➔ Value ➔ Type

Everything is an expression
Everything has a value
Everything has a type

A function is a value!
Complex types: Functions!

Parameter (formal) | Body Expr
--- | ---
fun \( x \rightarrow x+1 \); | fn

int \(-\) int

# let inc = fun x -> x+1 ;
val inc : int \(-\) int = fn

# inc 0;
val it : int = 1

# inc 10;
val it : int = 11
A Problem

Parameter (formal) | Body Expr
--|--
fun x -> x+1 ;; | fn

int -> int

Functions only have ONE parameter ?!

How a call (“application”) is evaluated:
1. Evaluate argument
2. Bind formal to arg value
3. Evaluate “Body expr”
A Solution: Simultaneous Binding

Parameter (formal)  Body Expr

fun \((x, y)\) \(\rightarrow\) \(x < y\);

(int * int) \(\rightarrow\) bool

How a call ("application") is evaluated:
1. Evaluate argument
2. Bind formal to arg value
3. Evaluate "Body expr"

Functions only have ONE parameter?
Another Solution ("Currying")

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>(formal)</td>
<td>Expr</td>
</tr>
<tr>
<td></td>
<td>fun x -&gt; fun y -&gt; x &lt; y;</td>
</tr>
<tr>
<td></td>
<td>fn</td>
</tr>
</tbody>
</table>

Int -> (Int -> Bool)

Whoa! A function can return a function

```ocaml
# let lt = fun x -> fun y -> x < y ;
val lt : int -> int -> bool = fn
# let is5Lt = lt 5;
val is5lt : int -> bool = fn;
# is5lt 10;
val it : bool = true;
# is5lt 2;
val it : bool = false;
```
Question: What is result of?

\((\text{fun } x \rightarrow \text{not } x)\)

(a) Syntax Error
(b) \(<\text{fun}> : \text{int} \rightarrow \text{int}\>
(c) \(<\text{fun}> : \text{int} \rightarrow \text{bool}\>
(d) \(<\text{fun}> : \text{bool} \rightarrow \text{int}\>
(e) \(<\text{fun}> : \text{bool} \rightarrow \text{bool}\>
And how about…

A function can also take a function argument

```ocaml
# let neg = fun f -> fun x -> not (f x); val lt : 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;
(*...odd, even ...*)
```
Question: What is result of?

\((\text{fun } \, f \, \rightarrow \, (\text{fun } \, x \, \rightarrow \, (f \, x) + x))\)

(a) Syntax Error
(b) Type Error
(c) \(<\text{fun}> : \text{int} \rightarrow \text{int} \rightarrow \text{int}\>
(d) \(<\text{fun}> : \text{int} \rightarrow \text{int}\>
(e) \(<\text{fun}> : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}\>
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…

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 list * 'a list = fn

# 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 ...

```
# 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

```ocaml
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**
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 \((T_1 \times T_2)\)
   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)

```
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 ...

    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

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

t is a new datatype.

A value of type \( t \) is either:

- a value of type \( t_1 \) placed in a box labeled \( C_1 \)
- a value of type \( t_2 \) placed in a box labeled \( C_2 \)
- \( \ldots \)
- a value of type \( t_n \) placed in a box labeled \( C_n \)
Constructing Datatypes

\[ \text{type } t = \text{C1 of } t1 \mid \text{C2 of } t2 \mid \ldots \mid \text{Cn of } tn \]

All have the type \( t \)
How to PUT values into box?
Question: Here is a typedef ...

```plaintext
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

\[
\text{type attrib} = \begin{align*}
\text{Name} & \text{ of } \text{string} & | \text{Age} & \text{ of } \text{int} & | \text{DOB} & \text{ of } \text{int*int*int} \\
\text{Address} & \text{ of } \text{string} & | \text{Height} & \text{ of } \text{float} & | \text{Alive} & \text{ of } \text{bool} \\
\text{Phone} & \text{ of } \text{int*int} & | \text{Email} & \text{ of } \text{string}; & ;
\end{align*}
\]

Name \quad “\text{Ranjit}” \quad \text{OR} \quad \text{Age} \quad 34 \quad \text{OR} \quad \text{DOB} \quad (9,8,77)

All have type \quad \text{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...

```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 ...

Wednesday, January 16, 2013
How to TEST & TAKE whats in 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 whats in the box?

<table>
<thead>
<tr>
<th><strong>type</strong> attrib =</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of string</td>
</tr>
<tr>
<td>Age of int</td>
</tr>
<tr>
<td>DOB of int<em>int</em>int</td>
</tr>
<tr>
<td>Address of string</td>
</tr>
<tr>
<td>Height of float</td>
</tr>
<tr>
<td>Alive of bool</td>
</tr>
<tr>
<td>Phone of int*int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>match e with</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name s -&gt; ...(*s: string *)</td>
</tr>
<tr>
<td>Age i -&gt; ...(*i: int *)</td>
</tr>
<tr>
<td>DOB(d,m,y) -&gt; ...(<em>d: int,m: int,y: int</em>)</td>
</tr>
<tr>
<td>Address a -&gt; ...(<em>a: string</em>)</td>
</tr>
<tr>
<td>Height h -&gt; ...(*h: int *)</td>
</tr>
<tr>
<td>Alive b -&gt; ...(<em>b: bool</em>)</td>
</tr>
<tr>
<td>Phone(y,r) -&gt; ...(<em>a: int, r: int</em>)</td>
</tr>
</tbody>
</table>

**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?

```c
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

in welcome (Age 34)

(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

```ocaml
# match (Name "Ranjit") with
| Name s  -> printf "Hello %s\n" s
| Age i   -> printf "%d years old" i
;;

Hello Ranjit
- : unit = ()
```

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*

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

Exception: Match Failure!!
```
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?

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”“Ranjit” : string
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
  ...;
  | 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`

```
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
match-with is an Expression

match e with
  C1 x1 -> e1
| C2 x2 -> e2
| ...
| Cn xn -> en
Q: What does this evaluate to?

```ocaml
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 $\text{type } t = C_1 \text{ of } t_1 \mid C_2 \text{ of } t_2$
   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

```plaintext
type nat = Zero | Succ of nat
```
“Recursive” types

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

Wait a minute! \textbf{Zero} of what ?!
“Recursive” types

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

Wait a minute! \textbf{Zero} of what?!

Relax.

Means “empty box with label \textbf{Zero}”
“Recursive” types

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

What are values of \text{nat}?
“Recursive” types

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

What are values of \text{nat}?
"Recursive" types

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

What are values of `nat`?

One `nat` contains another!
“Recursive” types

```haskell
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!
“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 $T_1 \text{ and a value of } T_2$

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

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

Recursive Code Mirrors Recursive Data
Next: Lets 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
let rec of_int n =

type nat =
  | Zero
  | Succ of nat

let rec of_int n =
```

*Base pattern*

*Inductive pattern*
of_int : int -> nat

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

**Type**

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

**Base pattern**

```
if n <= 0 then
```

**Inductive pattern**

```
else
```
of_int : int -> nat

let rec of_int n =
  if n <= 0 then Zero
  else Succ (of_int (n-1))
to_int : nat -> int

type nat =
  | Zero
  | Succ of nat

let rec to_int n =
to_int : nat -> int

let rec to_int n =

\[
\text{type nat =}
\begin{align*}
&\mid \text{Zero} \\
&\mid \text{Succ of nat}
\end{align*}
\]
to_int : nat -> int

Base pattern
Inductive pattern

let rec to_int n = match n with
| Zero  ->
| Succ m ->

Base pattern
Inductive pattern

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

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

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

\text{Base pattern}  \\
\text{Inductive pattern}
plus : nat*nat -> nat

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

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

\textbf{Base pattern}

\textbf{Inductive pattern}

\texttt{type nat =}
\begin{itemize}
  \item \texttt{Zero}
  \item \texttt{Succ} of nat
\end{itemize}

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

\begin{itemize}
  \item \texttt{type nat =}
  \begin{itemize}
    \item \texttt{Zero}
    \item \texttt{Succ of nat}
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \texttt{let rec plus (n,m) =}
  \begin{itemize}
    \item \texttt{match m with}
    \begin{itemize}
      \item \texttt{Zero} \to
      \item \texttt{Succ m'} \to
    \end{itemize}
  \end{itemize}
\end{itemize}
plus : nat*nat -> nat

\[
\text{type nat =}
\begin{array}{l}
\mid \text{Zero} \\
\mid \text{Succ} \text{ of nat}
\end{array}
\]

let rec plus (n,m) =
\[
\begin{array}{l}
\text{match m with} \\
\mid \text{Zero} \rightarrow n \\
\mid \text{Succ m'} \rightarrow \text{Succ (plus (n,m'))}
\end{array}
\]
times: nat*nat -> nat

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

```haskell
let rec times (n,m) =
```
\textbf{times}: \textit{nat}*\textit{nat} \rightarrow \textit{nat}

\begin{itemize}
  \item \textit{Base pattern}
  \hspace{1cm} \textbf{type} \textit{nat} =
  \begin{itemize}
    \item \textbf{Zero}
    \item \textbf{Succ of nat}
  \end{itemize}
  \item \textit{Inductive pattern}
\end{itemize}

\begin{itemize}
  \item \textit{let rec times (n,m) =}
\end{itemize}
times: nat*nat -> nat

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

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

type nat =
| Zero
| Succ of nat

let rec times (n, m) =
match m with
| Zero -> Zero
| Succ m' -> plus n (times (n, m'))
Q: How would you write:

\[ \text{minus}: \text{n}at^*\text{n}at \rightarrow \text{n}at \]
times: nat*nat -> nat

type nat =
| Zero
| Succ of nat

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

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

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
\textbf{times:} \texttt{nat*nat -> nat}

\textbf{Type} \texttt{nat} =
\begin{itemize}
  \item \texttt{Zero}
  \item \texttt{Succ \ of \ nat}
\end{itemize}

\textbf{Let rec} \texttt{minus (n,m) =}
\begin{align*}
  \text{match (n, m) with} \\
  &\text{\texttt{\_\_ \ Zero) \ -> } n} \\
  &\text{\texttt{(Succ n\', Succ m') \ -> } \texttt{minus(n',m')}}
\end{align*}
Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data
Lists are recursive types!

```ml
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!

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(_,t) -> 1 + (len t)
```

**Base pattern**

**Base Expression**

**Ind pattern**

**Inductive Expression**

Matches everything, no binding

Pattern-matching in order
- Must match with \texttt{Nil}
Some functions on Lists: Max

\[
\text{let rec max xs =}
\]

- Find the right \textit{induction} strategy
  - Base case: pattern + expression
  - Induction case: pattern + expression

Well designed datatype gives strategy
Some functions on Lists: Append

- 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: Let's get cosy with Recursion

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

(a) \((1, 2), 3\)
(b) \((\text{Leaf 1}, \text{Leaf 2}), \text{Leaf 3}\)
(c) \(\text{Node (Node (Leaf 1, Leaf 2), Leaf 3)}\)
(d) \(\text{Node ((Leaf 1, Leaf 2), Leaf 3)}\)
(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

Leaf of int

Node of tree*tree
Representing Trees

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

\[
\text{Node(Leaf 1, Leaf 2)}
\]
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\^2

Node(Node(Leaf 1, Leaf 2), Leaf 3)
Next: Lets 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.

# 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

```ocaml
let rec sum_leaf t =
```
**sum_leaf**: tree -> int

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

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

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

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

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:

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

What's a ML TYPE for REPRESENTING expressions?
Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

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

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:

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

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?

```haskell
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?

```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 ->
| 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 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
Random Art from Expressions

PA #2

Build more funky expressions, evaluate them, to produce:

Wednesday, January 16, 2013