Recap: ML’s Holy Trinity

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

Expressions (Syntax) \[\rightarrow\] Values (Semantics)

Compile-time “Static” \[\rightarrow\] Types

Exec-time “Dynamic”

Story So Far...

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

Today:
- Finish Crash Course
- Datatypes

Next: functions, but remember ...

Expression \[\rightarrow\] Value

Type

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

A function is a value!
A shorthand for function binding

```ml
# 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

```ml
- 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 lisi = fn

# 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 = fn

# 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;
```

```ml
# 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, lets begin at the beginning ...

What about more complex data?

- Expressions
- Values
- Types

Many kinds of expressions:

1. Simple
2. Variables
3. Functions

News

- Ocaml-top issues?
- Please post questions to Piazza
- Seating: Don’t Worry!

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

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)

```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
| Email of string

Quiz: Here is a typedef ...

What is the type of: Name “Tony Stark”
(a) Syntax Error
(b) Type Error
(c) string
(d) attrib
(e) ’a

Constructing Datatypes

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

\( t \) 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 \( C_n \)

All have the type \( t \)
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 = ...
```

Type declaration:

```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:

```
Name
OR
34
OR
DOB (9,8,77)
```

All have type `attrib`.

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
One-of types

- We've defined a “one-of” type named \texttt{attrib}
- Elements are \texttt{one of}:
  - \texttt{string},
  - \texttt{int},
  - \texttt{int*int*int},
  - \texttt{float},
  - \texttt{bool} …
- Can create uniform \texttt{attrib} lists
- Say I want a function to print attribs…

Question: Here is a typedef …

\begin{verbatim}
datatype 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;
\end{verbatim}

What is the type of:
\begin{enumerate}
\item [Name “Ranjit”; Age 35; Dob(9,8,77)]
\end{enumerate}

(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 …

\begin{enumerate}
\item string?
\item or an int?
\item or an int*int*int?
\item or …
\end{enumerate}

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 ?

match e with
| Name s     -> printf "%s" s 
| Age i      -> printf "%d" i 
| ... to pattern variable 
–  matching result expression is evaluated

• Simultaneously test and extract contents of box

Pattern-match expression: check if e is of the form ...

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

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

Pattern-match expression: check if e is of the form ...

– 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

First case matches the tag (Name)
Evals branch with s "bound" to string contents

Beware! Handle All TAGS!

None of the cases matched the tag (Name)
Causes nasty Run-Time Error

How to TEST & TAKE what's in box?

BEWARE!!

Be sure to handle all TAGS!

None of the cases matched the tag (Name)
Causes nasty Run-Time Error

Compiler To The Rescue!!

Compile-time checks for:
missed cases: ML warns if you miss a case!
Q: What does this evaluate to?

```plaintext
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!!

```
# let printAttrib a =
match a with
  ... : string
  ...
    | 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
Q: What does this evaluate to?

```plaintext
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

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

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
```

Type Rule

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

“Recursive” types

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

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

Wait a minute! Zero of what?! Relax.

Means “empty box with label Zero”

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

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

What are values of \texttt{nat}?
One \texttt{nat} contains another!

\begin{center}
\begin{tikzpicture}
    \node (zero) at (0,0) {Zero};
    \node (one) at (1,0) {Succ};
    \node (two) at (2,0) {Succ};
    \node (three) at (3,0) {Succ};
    \node (four) at (4,0) {Succ};
    \node (five) at (5,0) {Succ};

    \draw (zero) -- (one);
    \draw (one) -- (two);
    \draw (two) -- (three);
    \draw (three) -- (four);
    \draw (four) -- (five);
\end{tikzpicture}
\end{center}
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 \( t = C_1 \text{ of } t_1 | C_2 \text{ of } t_2 \)
   Value of T contains value of \( T_1 \) or a value of \( T_2 \)

3. “Recursive” type \( t = ... | C \text{ of } (...*t) \)
   Value of T contains (sub)-value of same type T

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

```
to_int : nat -> int
let rec to_int n =
| Zero
| Succ of nat
```

Next: Lets get cosy with Recursion

Code Structure = Type Structure!!!
to_int : nat -> int

```
let rec to_int n =
```

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

let rec of_int n =

| Zero
| Succ of nat

let rec of_int n =

if n <= 0 then
else

| Zero
| Succ of nat

let rec of_int n =

if n <= 0 then
  Zero
else
  Succ (of_int (n-1))
plus : nat*nat -> nat

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

let rec plus (n,m) =

\[
\text{match m with} \\
| \text{Zero} \rightarrow n \\
| \text{Succ m'} \rightarrow \text{Succ (plus (n,m'))}
\]
times: \text{nat} \times \text{nat} \rightarrow \text{nat}

\begin{align*}
\text{type} \quad \text{nat} &= \ \\
&\mid \text{Zero} \\
&\mid \text{Succ of nat} \\
\end{align*}

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

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

let rec minus (n, m) =
```

**times**: \(\text{nat} \times \text{nat} \rightarrow \text{nat}\)

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

let rec minus (n, m) =
  match (n, m) with
    | (_, Zero) -> n
    | (Succ n', Succ m') -> minus(n', m')
```
Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

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

Patterns:
- Base pattern
- Inductive pattern

Base Expression: `match l with Nil -> 0`  
Inductive Expression: `match l with Cons(_,t) -> 1 + (len t)`

Matches everything, no binding  
Pattern-matching in order
- Must match with `Nil`
Some functions on Lists: Append

\[
\text{let rec append (l1, l2) =}
\]

- **Base pattern**: \[\text{(pattern + expression)}\]
- **Inductive pattern**: \[\text{(pattern + expression)}\]

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
Q: How is this tree represented?

(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
3
Node
Node
Leaf
2
Leaf
1
Leaf
3

Next: Lets get cosy with Recursion

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

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

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
```

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

```
let rec sum_leaf t =
```

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

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)

Recursive Code Mirrors Recursive Data

Code almost writes itself!
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?

```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?

```
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
```