News?

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

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

Story So Far...

- Simple Expressions
- Branches
- Let-Bindings …

• Today:
  - Finish Crash Course
  - Datatypes
Variables and bindings

```plaintext
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

```plaintext
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

```plaintext
# 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:

```plaintext
# let h::t = [1;2;3];;
```

```
val h : int = 1
val t : int list = [2;3]
```

Why is it whining?

```plaintext
# let h::t = [];
Exception: Match_failure
```

```plaintext
# 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

In general xs may be empty (match failure!)

Another useful early warning
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

```plaintext
# 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

- 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

```plaintext
# 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

```ocaml
# 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 ...

```ocaml
# 2 <= 3;; ...
val it : bool = true
# "ba" <= "ab";;
val it : bool = false

# 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, lets begin at the beginning ...

What about more complex data?

**Expressions**

- Simple
- Variables
- Functions

**Values**

- Expressions
- Values

**Types**

- Expressions
- Values
- Types

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
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?

Attributes:
- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (real)
- Alive (boolean)
- Phone (int-int)
- Email (string)

Question: Here is a typedef ...

```
type attrib = Name of string | Age of int | ... | Phone of int-int | Email of string;
```

What is the type of: Name “Tony Stark”

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

Constructing Datatypes

```
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

\[
\text{type } t = \text{C1 of } t_1 \mid \text{C2 of } t_2 \mid \ldots \mid \text{Cn of } t_n
\]

All have the type \( t \)

How to PUT values into box?

How to create values of type \( \text{attrib} \)?

```ocaml
# let a1 = Name "Ranjit";;
val a1 : attrib = Name "Ranjit"
# let a2 = Height 5.83;;
val a2 : attrib = Height 5.83
# let a3 = DOB (9,8,1977) ;;
val a3 : attrib = DOB (9,8,1977)
# let a_l = [a1;a2;a3];;
val a_l : attrib list = ...
```

Question: Here is a typedef ...

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

What is the type of: Age “Tony Stark”

(a) Syntax Error
(b) Type Error
(c) string
(d) attrib
(e) ’a
Constructing Datatypes

```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; 
```

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 `attrib`s...

Question: Here is a typedef ...

```plaintext
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 whats in box?

Look at TAG!

How to tell whats in the box?

**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 (Name “Ranjit”)
```

(a) Name “Ranjit” : ‘a
(b) Type Error
(c) Name “Ranjit” : attrib
(d) “Ranjit” : string
(e) Runtime Error
Question: Here is a typedef ...

```plaintext
type attrib = Name of string | Age of int | ...
```

What does this evaluate to?

```plaintext
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**

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

```plaintext
# match (Name “Ranjit”) with
    | Name s -> printf "Hello %s\n" s
    | Age i -> printf "%d years old" i
    ;;
Hello Ranjit
- : unit = ()
```

**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!!
```

**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
Q: What does this evaluate to?

```ml
type attrib = Name of string | ... # 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 (_, _)

(a) Type Error
(b) "Welcome!Mickey" : string
(c) Runtime Error
(d) "Hello!Mickey" : string
(e) "Hello!MickeyWelcome!Mickey" Ranjit" : string

Benefits of `match-with`

1. Simultaneous test-extract-bind
2. Compile-time checks for:
   - missed cases: ML warns if you miss a case
   - redundant cases: ML warns if a case never matches

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

```ml
type t =
  C1 of t1
  | C2 of t2
  | ... 
  | Cn of tn
```

```ml
# let welcome a = match a with
  | Name s -> "Hello!" ^ s
  | Name s -> "Welcome!" ^ s
in welcome (Name "Mickey")
```
**match-with is an Expression**

Q: What does this evaluate to?

```occaml
type attrib = Name of string | Age of int | ...
let welcome a = match a with
  ... 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
“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?

One nat contains another!

Succ
Zero
“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 \text{t1} \times \text{t2}
   Value of \text{T} contains value of \text{T1} and a value of \text{T2}

2. “One-of” types \text{type t = C1 of t1} \mid \text{C2 of t2}
   Value of \text{T} contains value of \text{T1} or a value of \text{T2}

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

---

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

Code Structure = Type Structure!!!
Q: What does this evaluate to?

```plaintext
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**

```plaintext
let rec of_int n =
  if n <= 0 then !
  else !
type nat =
  | Zero
  | Succ of nat
```

Base pattern
Inductive pattern
Base pattern
Inductive pattern
Base pattern
Inductive pattern
of_int : int -> nat

**Type Definition**

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

**Recursive Function Definition**

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

to_int : nat -> int

**Type Definition**

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

**Recursive Function Definition**

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

to_int : nat -> int

**Type Definition**

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

**Recursive Function Definition**

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

---

**Base Pattern**

- `Type Definition`:
  - `type nat =` (Base pattern)

**Inductive Pattern**

- `Zero` and `Succ` (Inductive pattern)
**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) =  
match m with  
| Zero    -> ...
| Succ m' -> ...
```
**plus**: \(\text{nat*nat} \rightarrow \text{nat}\)

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

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

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

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

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

minus : nat*nat -> 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')
```

times : nat*nat -> nat

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

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)  ... -> minus(n',m')

Base pattern
Inductive pattern
Base pattern
Inductive pattern
Base pattern
Inductive pattern

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

Lists are recursive types!

type int_list =
| Nil
| Cons of int * int_list

Lists aren’t built-in!

datatype 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 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
Ind pattern

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

Base pattern
Ind pattern

Pattern-matching in order
Matches everything, no binding

Well designed datatype gives strategy
```

Some functions on Lists: Append

```
let rec append (l1,l2) =  
  Base Expression 
Inductive Expression 
Base pattern 
Ind pattern

• 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 =  
  Base Expression 
Inductive Expression 
Base pattern 
Ind pattern 

• 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?

1 2 3

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

(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

1 2 3

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

Leaf 1

Leaf 2
Representing Trees

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

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

```
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

Q: What does this evaluate to?

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

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

```
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:

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

Whats a ML FUNCTION for EVALUATING expressions?

```plaintext
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:

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

Whats a ML FUNCTION for EVALUATING expressions?

```plaintext
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 \times eval e2
```