Introduction to OCaml
(Continued)
Announcements

• HW #1 due Mon Jan 20

• Post questions/discussion to Piazza

• Check Piazza for TA/Tutor lab hours

• **Must** bring clickers starting Thurs
let \( x = e ; ; \)

“Bind the value of expression \( e \) to the variable \( x \)”

```ml
# let x = 2 + 2 ;;
val x : int = 4
```
Local Bindings: `let x = e1 in e2`

... 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

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

17424 int
```
Clicker Frequency for This Room

1. Press and hold until blinking...

2. Enter BD
What is the result of

```haskell
let x = 10 in
(le t z = 10 in x + z) + z
```

(a) Syntax Error
(b) 30
(c) Unbound Var Error (x)
(d) Unbound Var Error (z)
(e) Other Type Error
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]
```
But:

```ml
# 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?

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

In general, \( xS \) may be empty (match failure!)

Another useful early warning
What About ... Functions
Functions are values!

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

Remember the Holy Trinity
Complex Type: Functions!

Parameter (formal) | Body (Expr)
--- | ---

fun \( x \rightarrow x + 1 \)

\[ \text{int} \rightarrow \text{int} \]

How a call (“application”) is evaluated:

1. Evaluate argument
2. Bind formal to arg value
3. Evaluate “Body expr”
Complex Type: Functions!

Functions only have ONE parameter?!!

Parameter (formal) Body

fun x -> x+1

int -> int
Solution #1: Simultaneous Binding

Parameter
(formal)

fun \((x, y)\) \rightarrow x \lt y

(int * int) \rightarrow \text{bool}

Functions only have ONE parameter ?!?
Solution #2: “Currying”

Whoa! A function can return a function

```ml
# let lt = fun x -> fun y -> x < y ;;
val lt : int -> int -> bool = <fun>
# let is5Lt = lt 5 ;;
val is5lt : int -> bool = <fun>
# is5lt 10 ;;
- : bool = true
# is5lt 2 ;;
- : bool = false
```
What is the result of

(fun x -> not x) ?

(a) Syntax Error
(b) <fun> : int -> int
(c) <fun> : int -> bool
(d) <fun> : bool -> int
(e) <fun> : bool -> bool
Higher-Order Function

Parameter (formal)   Body Expr

fun f -> fun x -> not(f x)

('a -> bool) -> ('a -> bool)

A function can also take a function argument

# let neg = fun f -> fun x -> not(f x) ;;
val lt : int -> int -> bool = <fun>
# let is5gte = neg is5lt ;;
val is5gte : int -> bool = <fun>
# is5gte 10 ;;
- : bool = false
# is5gte 2 ;;
- : bool = true
(*... odd, even ...*)
What is the result of

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

(a) Syntax Error
(b) \(<\text{fun}> : \text{int} \rightarrow \text{int} \rightarrow \text{int}\>
(c) \(<\text{fun}> : \text{int} \rightarrow \text{int}\>
(d) \(<\text{fun}> : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}\>
# let neg = fun f -> fun x -> not (f x);;
...

# let neg f x = not (f x) ;;
val neg : int -> int -> bool = <fun>

# let is5gte = neg is5lt ;;
val is5gte : int -> bool = <fun>
# is5gte 10 ;;
- : bool = false
# is5gte 2 ;;
- : bool = true
# 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 = <fun>

# let list1 = [1; 31; 12; 4; 7; 2; 10] ;;
# filter is5lt list1 ;;
- : int list = [31; 12; 7; 10]
# filter is5gte list1 ;;
- : int list = [1; 4; 2]
# filter even list1 ;;
- : int list = [12; 4; 2; 10]
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

# let list1 = [1; 31; 12; 4; 7; 2; 10] ;;
- ...

# partition is5lt list1 ;;
- : (int list * int list) = ([31;12;7;10],[1;4;2])

# partition even list1 ;;
- : (int list * int list) = ([12;4;2;10],[1;31;7])
```
“Operators” are Functions

```ocaml
# 2 <= 3 ;;
- : bool = true

# "ba" <= "ab" ;;
- : bool = false

# let lt = (<) ;;
val lt : 'a -> 'a -> bool = <fun>

# lt 2 3 ;;
- : bool = true

# lt "ba" "ab" ;;
- : bool = false

# let is5Lt = lt 5 ;;
val is5lt : int -> bool = <fun>

# is5lt 10 ;;
- : bool = true

# is5lt 2 ;;
- : bool = false
```
let rec sort xs =
    match xs with
    | [] -> []
    | (h::t) ->
        let (l,r) = partition ((<) h) t in
        (sort l) @ (h::(sort r))
Datatypes
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**

```plaintext
type t = (t1 * t2)
```

Value of `t` contains value of `t1 and` a value of `t2`

**Data (a.k.a. “One-of”, “Variant”) Type**

```plaintext
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?
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

```plaintext
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

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

- \( t \) is a new datatype
- A value of type \( t \) is either:
  - a value of type \( t1 \) placed in a box labeled \( \text{C1} \)
  - a value of type \( t2 \) placed in a box labeled \( \text{C2} \)
  - ... 
  - a value of type \( tn \) placed in a box labeled \( \text{Cn} \)
Constructing Datatypes

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

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 create values of type `attrib`?

```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;;
```

```ocaml
# 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

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

All have type \text{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
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 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!
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?**

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

**match e with**

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

**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?

```plaintext
match e with
default {
  printf "Hello %s\n" s
}
def Name s {
  printf "Hello %s\n" s
}
def Age i {
  printf "%d\n" i
}
def DOB(d,m,y) {
  printf "%d/%d/%d\n" d m y
}
def Address s {
  printf "Address: %s\n" s
}
def Height h {
  printf "%f\n" h
}
def Alive b {
  printf "Alive: %b\n" b
}
def Phone(a,r) {
  printf "Phone: (%d)-%d\n" 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

- \( e_1, e_2, \ldots, e_n \) must have same type \( t \)
- Type of whole expression is \( t \)
What is the result of

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

What is the result of

\[
\text{let welcome } a = \text{match } a \text{ with}
\]
\[
| \text{Name } s \rightarrow \text{“Hello, ” } ^s ^\text{“! ”} \\
| \text{Name } s \rightarrow \text{“Hi, ” } ^s ^\text{“! ”}
\]

\[
\text{in welcome (Name “Ravi”)}
\]

(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!

```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.
val printAttrib : attrib -> unit = <fun>
```

ML gives a compile-time warning about **redundant** cases
(which will never match)!
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
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
1. Simultaneous \texttt{test-extract-bind}

2. Compile-time checks for:
   - \textit{missed} cases: ML warns if you \texttt{miss a \texttt{t} value}
   - \textit{redundant} cases: ML warns if a case \texttt{never matches}

\begin{align*}
\text{type } \texttt{t} &= \texttt{| C1 of \texttt{t1} \\
& | C2 of \texttt{t2} \\
& | \ldots \\
& | Cn of \texttt{tn}} \\
\text{match } \texttt{e} \text{ with} &= \texttt{| C1 \ x1 \ -> \ e1 \\
& | C2 \ x2 \ -> \ e2 \\
& | \ldots \\
& | Cn \ xn \ -> \ en}
\end{align*}
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

\[
type\ t = \ldots \ | \ C\ of\ (\ldots * t)
\]
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

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

What are values of \text{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

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

What are values of \text{nat}?

One \text{nat} contains another!
3 Ways to Build Complex Values

**Tuple (a.k.a. “Each-of”, “Product”) Type**

```plaintext
type t = (t1 * t2)
```

Value of `t` contains value of `t1` and a value of `t2`

**Data (a.k.a. “One-of”, “Variant”) Type**

```plaintext
type t = C1 of t1 | C2 of t2
```

Value of `t` contains value of `t1` or a value of `t2`

**Recursive Datatype**

```plaintext
type t = ... | C of (... * t)
```

Value of `t` contains (sub)-value of same type `t`