News

• PA 3 due **THIS** Friday (10/26) at 5pm

• Midterm on **11/6**
Recap: Functions as “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions

Using, (Taking) Functions

Iterator, Accumul, Reuse computation pattern w/o exposing local info
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Parameterized, similar functions (e.g. Testers)

Compose Functions:
Flexible way to build Complex functions from primitives.

Iterator, Accumul, Reuse computation pattern w/o exposing local info
Higher-order funcs enable modular code

- Each part only needs local information

**Data Structure**

- Client
- Uses list
  - Uses meta-functions: map, fold, filter
  - With locally-dependent funs (lt h), square etc.
  - Without requiring Implement. details of data structure

**Data Structure**

- Library
- list
  - Provides meta-functions: map, fold, filter
  - to traverse, accumulate over lists, trees etc.
  - Meta-functions don’t need client info (tester ? accumulator ?)
“Map-Reduce” et al.

Higher-order funcs enable modular code
• Each part only needs local information

Map-Reduce Client

Web Analytics “Queries”
Clustering, Page Rank, etc
as map/reduce + ops

Map-Reduce Library

Provides: map, reduce
to traverse, accumulate
over WWW (“Big Data”)
Distributed across “cloud”

Tuesday, October 23, 2012
Higher Order Functions Are Awesome...
Higher Order Functions
..but how do they work
Next: Environments & Functions

Let's start with the humble variable...
Variables and Bindings

Q: How to use variables in ML?
Q: How to “assign” to a variable?

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

let x = e;;
```

“Bind value of expr \(e\) to variable \(x\)”
Variables and Bindings

```ocaml
# let x = 2+2;;
val x : int = 4
# let y = x * x * x;;
val y : int = 64
# let z = [x;y;x+y];;
val z : int list = [4;64;68]
```

Later expressions can use \texttt{x}

- Most recent “bound” value used for evaluation

Sounds like C/Java?

NO!
Environments (“Phone Book”)

How ML deals with variables

• Variables = “names”
• Values = “phone number”
Environments and Evaluation

ML begins in a “top-level” environment

- Some names bound (e.g. +, -, print_string...)

```ml
let x = e;;
```

ML program = Sequence of variable bindings

Program evaluated by evaluating bindings in order

1. Evaluate `expr e` in current env to get value `v : t`
2. Extend env to bind `x` to `v : t`
(Repeat with next binding)
Environments

“Phone book”
- Variables = “names”
- Values = “phone number”

1. Evaluate:
Find and use most recent value of variable

2. Extend:
Add new binding at end of “phone book”
Example

# let x = 2+2;;
val x : int = 4

# let y = x * x * x;;
val y : int = 64

# let z = [x;y;x+y];;
val z : int list = [4;64;68]

# let x = x + x ;;
val x : int = 8

New binding!
Environments

1. **Evaluate**: Use most recent bound value of var
2. **Extend**: Add new binding at end

How is it different from C/Java’s “store”?

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

# let f = fun y -> x + y;
val f : int -> int = fn

# let x = x + x ;
val x : int = 8

# f 0;
val it : int = 4
```

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4 : int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4 : int</td>
</tr>
<tr>
<td>f</td>
<td>fn &lt;code, ↑ &gt;: int-&gt;int</td>
</tr>
</tbody>
</table>

New binding:
- No change or mutation
- Old binding frozen in `f`
Environments

1. **Evaluate**: Use most recent bound value of var
2. **Extend**: Add new binding at end

How is it different from C/Java’s “store”?

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

# let f = fun y -> x + y;
val f : int -> int = fn

# let x = x + x ;
val x : int = 8;

# f 0;
val it : int = 4
```
Environments

1. Evaluate: Use most recent bound value of var
2. Extend: Add new binding at end

How is it different from C/Java’s “store”?

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

# let f = fun y -> x + y;;
val f : int -> int = fn

# let x = x + x ;
val x : int = 8

# f 0;
val it : int = 4
```

Binding used to eval (f …)

<table>
<thead>
<tr>
<th>Binding for subsequent x</th>
<th>Binding for subsequent x</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4 : int</td>
</tr>
<tr>
<td>f</td>
<td>fn &lt;code, &gt;: int-&gt;int</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

Tuesday, October 23, 2012
Cannot change the world

Cannot “assign” to variables
• Can extend the env by adding a fresh binding
• Does not affect previous uses of variable

Environment at fun declaration frozen inside fun “value”
• Frozen env used to evaluate application \((f \ e)\)

Q: Why is this a good thing?

```ocaml
# let x = 2+2;;
val x : int = 4
# let f = fun y -> x + y;;
val f : int -> int = fn
# let x = x + x ;;
val x : int = 8;
# f 0;;
val it : int = 4
```

Binding used to eval \((f \ e)\)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
<tr>
<td>f</td>
<td>fn &lt;code, -&gt; int</td>
</tr>
</tbody>
</table>

Binding for subsequent \(x\)
Cannot change the world

Q: Why is this a good thing?
A: Function behavior frozen at declaration
Immutability: The Colbert Principle

“A function behaves the same way on Wednesday, as it behaved on Monday, no matter what happened on Tuesday!”

Tuesday, October 23, 2012
Cannot change the world

Q: Why is this a **good thing**?
A: Function behavior frozen at declaration

- Nothing entered afterwards affects function
- Same inputs **always produce same outputs**
  - Localizes debugging
  - Localizes reasoning about the program
  - No “sharing” means no evil aliasing
Examples of no sharing

Remember: No addresses, no sharing.
• Each variable is bound to a “fresh instance” of a value

Tuples, Lists ...
• Efficient implementation without sharing?
  • There is sharing and pointers but hidden from you

• Compiler’s job is to optimize code
  • Efficiently implement these “no-sharing” semantics

• Your job is to use the simplified semantics
  • Write correct, cleaner, readable, extendable systems
Function bindings

Functions are values, can bind using `val`

```ocaml
let fname = fun x -> e ;;
```

**Problem**: Can’t define recursive functions!
- `fname` is bound after computing rhs value
- no (or “old”) binding for occurrences of `fname` inside `e`

```ocaml
let rec fname x = e ;;
```

Occurrences of `fname` inside `e` bound to “this” definition

```ocaml
let rec fac x = if x<=1 then 1 else x*fac (x-1)
```
Local bindings

So far: bindings that remain until a re-binding (“global”)

Local, “temporary” variables are useful inside functions

• Avoid repeating computations
• Make functions more readable

let x = e₁ in
  e₂
;;

Let-in is an expression!

Evaluating let-in in env \( E \):

1. Evaluate expr \( e₁ \) in env \( E \) to get value \( v : t \)
2. Use extended \( E [x \mapsto v : t] \) (only) to evaluate \( e₂ \)
Local bindings

Evaluating let-in in env $E$:

1. Evaluate expr $e_1$ in env $E$ to get value $v : t$
2. Use extended $E[ x \mapsto v : t]$ to evaluate $e_2$
Let-in is an expression!

Evaluating let-in in env \( E \):
1. Evaluate expr \( e_1 \) in env \( E \) to get value \( v : t \)
2. Use extended \( E [x \rightarrow v : t] \) to evaluate \( e_2 \)

```ml
let y =
let
  x = 10
in
  x * x
;;
```

\[ \begin{array}{c|c}
  \cdots & \cdots \\
  \hline
  x & 10 : int \\
  \hline
  \cdots & \cdots \\
  \hline
  y & 100 : int \\
\end{array} \]
Nested bindings

Evaluating let-in in env $E$:

1. Evaluate expr $e_1$ in env $E$ to get value $v : t$
2. Use extended $E \left[ x \mapsto v : t \right]$ to evaluate $e_2$

```
let
  x = 10
in
  (let
    y = 20
  in
    x * y)
+ x
;;
```
Nested bindings

GOOD Formatting

BAD Formatting
Example

```ocaml
let rec filter f xs =
    match xs with
    | [] -> []
    | x::xs' -> let ys = if f x then [x] else [] in
              let ys' = filter f xs in
              ys @ ys'
```

Tuesday, October 23, 2012
Recap 1: Variables are names for values

- Environment: dictionary/phonebook
- Most recent binding used
- Entries never change
- New entries added
Recap 2: Big Exprs With Local Bindings

- let-in expression
- Variable “in-scope” in-expression
- Outside, variable not “in-scope”
Recap 3: Env Frozen at Func Definition

- Re-binding vars cannot change function
- Identical I/O behavior at every call
- Predictable code, localized debugging
Static/Lexical Scoping

• For each occurrence of a variable, **A unique place** where variable was defined!
  - Most recent binding in environment

• **Static/Lexical**: Determined from program text
  - Without executing the program

• Very useful for **readability, debugging**:
  - Don’t have to figure out “where” a variable got assigned
  - Unique, **statically** known definition for each occurrence
Q: What’s the value of a function?
Immutability: The Colbert Principle

“A function behaves the same way on Wednesday, as it behaved on Monday, no matter what happened on Tuesday!”
Two ways of writing function expressions:

1. Anonymous functions:

   let fname = fun x -> e

2. Named functions:

   let fname x = e
Function Application

Expressions

Application: fancy word for “call”

\( (e_1 \ e_2) \)

- Function value \( e_1 \)
- Argument \( e_2 \)
- “apply” argument \( e_2 \) to function value \( e_1 \)
Functions

The type of any function is:

- $T1$: the type of the “input”
- $T2$: the type of the “output”

$$\texttt{let} \ \texttt{fname} = \texttt{fun} \ x \rightarrow e$$

$T1 \rightarrow T2$

$$\texttt{let} \ \texttt{fname} \ x = e$$

$T1 \rightarrow T2$
Functions

The type of any function is:

- \( T_1 \) : the type of the “input”
- \( T_2 \) : the type of the “output”

\( T_1, T_2 \) can be any types, including functions!

What's an example of?

- \( \text{int} \rightarrow \text{int} \)
- \( \text{int} \times \text{int} \rightarrow \text{bool} \)
- \( (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \)
Type of function application

Application: fancy word for “call”

\((e_1 \ e_2)\)

- “apply” argument \(e_2\) to function value \(e_1\)

\[
\begin{array}{c}
e_1 : T_1 \rightarrow T_2 \\
e_2 : T_1 \\
(e_1 \ e_2) : T_2
\end{array}
\]

- **Argument** must have same type as “input” \(T_1\)
- **Result** has the same type as “output” \(T_2\)
Two questions about function values:

What is the value:

1. ... of a function?

2. ... of a function “application” (call)?
Values of function = “Closure”

Two questions about function values:

What is the value:

1. ... of a function?

Closure = Code of Fun. (formal x + body e) + Environment at Fun. Definition
Two questions about function values:

What is the value:

1. ... of a function?

Closure = Code of Fun. (\texttt{formal } x + \texttt{body } e) + Environment at Fun. Definition
Values of functions: Closures

• Function value = “Closure”
  - <code + environment at definition>

• Body not evaluated until application
  - But type-checking when function is defined

```ocaml
# let x = 2+2;;
val x : int = 4
# let f = fun y -> x + y;;
val f : int -> int = fn
# let x = x + x;;
val x : int = 8
# f 0;;
val it : int = 4
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>4 : int</td>
</tr>
<tr>
<td>(f)</td>
<td>fn &lt;code,</td>
</tr>
<tr>
<td></td>
<td>&gt;: int-&gt;int</td>
</tr>
<tr>
<td>(x)</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

Binding used to eval \((f \ldots)\)

Binding for subsequent \(x\)
Two questions about function values:

What is the value:

1. ... of a function?

2. ... of a function “application” (call)?

fun x -> e

(e1 e2)
Free vs. Bound Variables

let \( a = 20 ;; \)

let \( f \ x = \)
\( \quad \text{let } y = 1 \text{ in} \)
\( \quad \text{let } g \ z = y + z \text{ in} \)
\( \quad a + (g \ x) \)
\n\( f \ 0 ;; \)

Environment frozen with function

Used to evaluate fun application

Which vars needed in frozen env?
Free vs. Bound Variables

```
let a = 20;;

let f x =
  let y = 1 in
  let g z = y + z in
  a + (g x)
;;

f 0;;
```

Inside a function:

A “bound” occurrence:
1. Formal variable
2. Variable bound in `let-in`

x, y, z are “bound” inside \( f \)

A “free” occurrence:
- Non-bound occurrence
a is “free” inside \( f \)

Frozen Environment
needed for values of free vars
Free vs. Bound Variables

let a = 20;;

let f x =
  let a = 1 in
  let g z = a + z in
  a + (g x)
  ;;

f 0;

Inside a function:
A “bound” occurrence:
1. Formal variable
2. Variable bound in let-in-end

x, a, z are “bound” inside f

A “free” occurrence:
Not bound occurrence
nothing is “free” inside f
Where do bound-vars values come from?

```ml
let a = 20;;

let f x =
  let a = 1 in
  let g z = a + z in
  a + (g x)
;;

f 0;
```

Bound values determined when function is evaluated (“called”)
- Arguments
- Local variable bindings
Values of function application

Two questions about function values:

What is the value:

1. ... of a function?

2. ... of a function “application” (call)?

“apply” the argument \( e_2 \) to the (function) \( e_1 \)
Values of function application

Value of a function “application” (call) \((e1 \ e2)\)

1. Find closure of \(e1\)
2. Execute body of closure with param \(e2\)

Free values found in closure-environment

Bound values by executing closure-body
Values of function application

Value of a function “application” (call) \((e_1, e_2)\)

1. Evaluate \(e_1\) in current-env to get (closure)
   \[= \text{code (formal } x \text{ + body } e) + \text{ env } E\]

2. Evaluate \(e_2\) in current-env to get (argument) \(v_2\)

3. Evaluate body \(e\) in env \(E\) extended with \(x := v_2\)
Q: What is the value of \texttt{res} ?

\begin{align*}
\text{let } x &= 1 ;; \\
\text{let } y &= 10 ;; \\
\text{let } f y &= x + y ;; \\
\text{let } x &= 2 ;; \\
\text{let } y &= 3 ;; \\
\text{let } \texttt{res} = f (x + y) ;;
\end{align*}

\begin{align*}
\text{let } x &= 2 \\
\text{let } y &= 3 \\
\text{let } \texttt{res} = f (x + y) ;;
\end{align*}

\text{Application: } f (x + y) \\
\text{Eval } \texttt{body} \text{ in } \texttt{env} \text{ extended with } \texttt{formal} \rightarrow 5 \\
\text{Eval } x+y \text{ in } [x \rightarrow 1, \, y \rightarrow 5] \quad \Longrightarrow \quad 6
Example

```ocaml
let x = 1;;
let f y =
  let x = 2 in
  fun z -> x + y + z
;;
let x = 100;;
let g = f 4;;
let y = 100;;
(g 1);;
```

Q: Closure value of \( g \)?

**formal** \( z \)

**body** \( x + y + z \)

**env** \([x|->2,\ y|->4]\]

Eval **body** in **env** extended with **formal**|-> 1
Eval **x+y+z** in **env** \([x|->2,\ y|->4,\ z|->1]\) \Longrightarrow 7
Example 3

```ocaml
let f g =
  let x = 0 in
  g 2
;;
let x = 100;;

let h y = x + y;;

f h;;
```
Static/Lexical Scoping

- For each occurrence of a variable,
  - Unique place in program text where variable defined
  - Most recent binding in environment

- Static/Lexical: Determined from the program text
  - Without executing the program

- Very useful for readability, debugging:
  - Don’t have to figure out “where” a variable got assigned
  - Unique, statically known definition for each occurrence
Immutability: The Colbert Principle

“A function behaves the same way on Wednesday, as it behaved on Monday, no matter what happened on Tuesday!”