Recap: Functions as “first-class” values
• Arguments, return values, bindings ...
• What are the benefits?

Parameterized, similar functions (e.g. Testers)
Iterator, Accumul, Reuse computation pattern w/o exposing local info

Creating, (Returning) Functions
Using, (Taking) Functions
Funcs taking/returning funcs
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 funcs (lt h), square etc.
Without requiring implement. details of data structure

Data Structure Library
Provides meta-functions: map, fold, filter
to traverse, accumulate over lists, trees etc.
Meta-functions don’t need client info (tester ? accumulator ?)
Higher-order functions enable modular code
• Each part only needs local information

“Map-Reduce” et al.

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

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

Higher Order Functions

Are Awesome...

..but how do they work

Next: Environments & Functions

Expressions ➔ Values

Types

Lets start with the humble variable...
**Variables and Bindings**

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

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

“Bind value of expr e to variable x”

**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 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 |

How is it different from C/Java’s “store”?
- No change or mutation
- Old binding frozen in `f`

---

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

| # 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 |

| # let f 0; |
|---|---|
| val it : int = 4 |

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
```

Binding used to eval (f …)

<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>
<tr>
<td>x</td>
<td>8   : int</td>
</tr>
</tbody>
</table>

Binding for subsequent x

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?

**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!”
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

Q: What is the value of res?

(a) 120
(b) 60
(c) 20
(d) 5
(e) 1

Let bindings

Functions are values, can bind using val

let f = fun x -> 1;;
let f = fun x -> if x<2 then 1 else (x * f(x-1));;
let res = f 5;;

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

let rec fname x = e ;;

Occurrences of fname inside e bound to “this” definition

let rec fac x = if x<=1 then 1 else x*fac (x-1)
Q: What is the value of \texttt{res}?

\begin{verbatim}
let y = let x = 10 in
    x + x ;;
let res = (x, y) ;;
\end{verbatim}

(a) Unbound Var Error
(b) (10,20)
(c) (10,10)
(d) Type Error

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-in is an expression!

Evaluating let-in in env \(E\):
1. Evaluate \(expr\ \texttt{e1}\) in env \(E\) to get value \(v : t\)
2. Use extended \(E[x :-> v : t]\) to evaluate \(e2\)

Let-in is an expression!

Evaluating let-in in env \(E\):
1. Evaluate \(expr\ \texttt{e1}\) in env \(E\) to get value \(v : t\)
2. Use extended \(E[x :-> v : t]\) to evaluate \(e2\)
Nested bindings

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

Example

```
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'
```

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

Next: Functions

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!”

Function Application

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

Two ways of writing function expressions:

1. Anonymous functions:

   \[
   \text{let } \text{fname} = \text{fun } x \rightarrow e
   \]

2. Named functions:

   \[
   \text{let } \text{fname}_x = e
   \]

Expressions

The type of any function is:

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

\[
\text{let } \text{fname} = \text{fun } x \rightarrow e
\]

\(T_1 \rightarrow T_2\)

\[
\text{let } \text{fname}_x = e
\]

\(T_1 \rightarrow T_2\)
**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?
- \( \text{int} \rightarrow \text{int} \)
- \( \text{int} * \text{int} \rightarrow \text{bool} \)
- \( (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \)

**Values**

Two questions about function values:

What is the value:
1. ... of a function?
2. ... of a function "application" (call)?

**Type of function application**

Application: fancy word for "call"

- "apply" argument \( e_2 \) to function value \( e_1 \)

\[
\begin{align*}
\text{e}_1 : T_1 \rightarrow T_2 & \quad \text{e}_2 : T_1 \\
(\text{e}_1 \text{ e}_2) : T_2
\end{align*}
\]

- Argument must have same type as "input" \( T_1 \)
- Result has the same type as "output" \( T_2 \)

**Values of function = “Closure”**

Two questions about function values:

What is the value:

1. ... of a function?

**Closure**

Code of Fun. \((\text{formal} \ x + \text{body} \ e)\)
+ Environment at Fun. Definition
Two questions about function values:

What is the value:

1. ... of a function?

\[
\text{fun } x \rightarrow e
\]

Closure = Code of Fun. (formal \(x\) + 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

Q: Which vars in env. of \(f\)?

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
<th>(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(y)</td>
<td>(x\ y)</td>
<td>(x\ y\ z)</td>
<td>None</td>
</tr>
</tbody>
</table>

Q: Vars in closure-env of \(f\)?

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
<th>(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a\ y)</td>
<td>(a)</td>
<td>(y)</td>
<td>(z)</td>
<td>(y\ z)</td>
</tr>
</tbody>
</table>

# Values of function = “Closure”

# Q: Which vars in env. of \(f\)?

# Q: Vars in closure-env of \(f\)?

Binding used to eval \(\langle f \rangle \ldots\): \(x\) : \text{int} = 4

Binding for subsequent \(x\): 4 : \text{int}

# let \(x\) = 2+2;;
val \(x\) : \text{int} = 4

# let \(f\) = fun \(y\) -> \(x\) + \(y\);;
val \(f\) : \text{int} -> \text{int} = fn

# let \(x\) = \(x\) + \(x\);;
val \(x\) : \text{int} = 8

# \(f\) 0;;
val it : \text{int} = 4
Free vs. Bound Variables

Environment frozen with function
Used to evaluate fun application
Which vars needed in frozen env?

\[
\begin{align*}
\text{let } & a = 20;; \\
\text{let } & f x = \\
& \quad \text{let } y = 1 \text{ in} \\
& \quad \text{let } g z = y + z \text{ in} \\
& \quad a + (g x) \\
\end{align*}
\]

\[f 0;;\]

\[ (e1 \ e2) \]

Free vs. Bound Variables

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

Q: Which vars are free in \(f\) ?

(a) \(a\)  
(b) \(x\)  
(c) \(y\)  
(d) \(z\)  
(e) None

Free vs. Bound Variables

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?

let \( a = 20 ;; \)

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

\( f \) 0;

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

1. Find closure of \( e_1 \)
2. Execute body of closure with param \( e_2 \)

Free values found in closure-environment

Bound values by executing closure-body

Values of function application

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{verbatim}
let x  = 1;;
let y  = 10;;
let f y = x + y;;
let x  = 2;;
let y  = 3;;
let res = f (x + y);;
\end{verbatim}

(a) 4  (b) 5  (c) 6  (d) 11  (e) 12

Q: What is the value of \texttt{res}?

\begin{verbatim}
let x  = 1;;
let y  = 10;;
let f y = x + y;;
let x  = 2;;
let y  = 3;;
let res = f (x + y);;
\end{verbatim}

\begin{verbatim}
Q: What is the value of \texttt{res}?
\end{verbatim}

(a) Syntax Error
(b) 102
(c) Type Error
(d) 2
(e) 100
Example 3

```
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!”