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

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

Compose Functions:
Flexible way to build Complex functions from primitives.
Higher-order functions enable modular code
- Each part only needs local information

**Funcs taking/returning funcs**

- **Data Structure**
- **Client**
- **Uses list**

**Uses** meta-functions:
- map, fold, filter

With locally-dependent funs
- (lt h), square etc.

Without requiring implementing
- details of data structure

**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.**

- **Map-Reduce**
- **Library**

**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
Let's start with the humble variable...

```
# 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 `x`
- Most recent “bound” value used for evaluation

Sounds like C/Java?
NO!

**Variables and Bindings**

<table>
<thead>
<tr>
<th>x</th>
<th>4 : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>64 : int</td>
</tr>
<tr>
<td>z</td>
<td>[4;64;68] : int list</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

**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...)

```ocaml
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”

---

```
Q: What is the value of res ?

<table>
<thead>
<tr>
<th>let x = 0      ;;</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>let y = x + 1     ;;</td>
<td>...)</td>
<td></td>
</tr>
<tr>
<td>let z = (x, y)    ;;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>let x = 100       ;;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>let res = z       ;;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(a) (0 , 1)
(b) (100, 101)
(c) (0, 100)
(d) (1, 100)

---

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!
Q: What is the value of \( \text{res} \) ?

\[
\begin{align*}
\text{let } x &= 0 ; ; \\
\text{let } y &= x + 1 ; ; \\
\text{let } z a &= (x, y) ; ; \\
\text{let } x &= 100 ; ; \\
\text{let } \text{res} &= z \ [] ; ; \\
\end{align*}
\]

(a) (0, 1)  
(b) (100, 101)  
(c) (0, 100)  
(d) (100, 1)

Environment

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”?

```plaintext
# 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
```

New binding:  
- No change or mutation  
- Old binding frozen in \( f \)

Environment

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”?

```plaintext
# 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 \( f \) ...

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!”

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 \texttt{val}

\begin{verbatim}
let fname = fun x -> e ;;
\end{verbatim}

Problem: Can’t define recursive functions!

• \texttt{fname} is bound after computing rhs value
• no (or “old”) binding for occurrences of \texttt{fname} inside \texttt{e}

\begin{verbatim}
let rec fname x = e ;;
\end{verbatim}

Occurences of \texttt{fname} inside \texttt{e} bound to “this” definition

\begin{verbatim}
let rec fac x = if x<=1 then 1 else x*fac (x-1)
\end{verbatim}

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

\begin{verbatim}
let f x = 1;; let f x = if x<2 then 1 else (x * f(x-1));; let res = f 5;;
\end{verbatim}

(a) 120
(b) 60
(c) 20
(d) 5
(d) 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) Syntax 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 $e_1$ in env $E$ to get value $v : t$
2. Use extended $E [x |-> v : t]$ (only) to evaluate $e_2$

Nested bindings

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

```plaintext
let
  x = 10
in
let
  y = 20
in
  x * y
;;
```

GOOD Formatting

BAD Formatting

Example

```plaintext
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!”
Functions

Two ways of writing function expressions:

1. Anonymous functions:
   - Parameter (formal)
   - Body Expr
   ```
   let fname = fun \( x \) -> e
   ```

2. Named functions:
   - Parameter (formal)
   - Body Expr
   ```
   let fname \( x \) = e
   ```

Function Application

Application: fancy word for “call”

```
(e1 e2)
```

- Function value \( e1 \)
- Argument \( e2 \)
- “apply” argument \( e2 \) to function value \( e1 \)

Functions Type

The type of any function is:
- \( T1 \) : the type of the “input”
- \( T2 \) : the type of the “output”

```
let fname = fun \( x \) -> e
```

- \( T1 \) -> \( T2 \)

```
let fname \( x \) = e
```

- \( T1 \) -> \( T2 \)

T1, T2 can be any types, including functions!

What's an example of?
- \( int \) -> \( int \)
- \( int * int \) -> \( bool \)
- \( (int -> int) -> (int -> int) \)
Type of function application

Application: fancy word for “call”

\[(e_1 e_2)\]

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

\[
\begin{align*}
  & e_1 : T_1 \rightarrow T_2 \\
  & e_2 : T_1 \\
  & (e_1 e_2) : T_2
\end{align*}
\]

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

Functions

Two questions about function values:

What is the value:

1. ... of a function ?  \[\text{fun } x \rightarrow e\]
2. ... of a function “application” (call) ?  \[(e_1 e_2)\]

Values of function = “Closure”

Two questions about function values:

What is the value:

1. ... of a function ?  \[\text{fun } x \rightarrow e\]

Closure =

Code of Fun. \((\text{formal } x + \text{body } e)\)
+ Environment at Fun. Definition
Q: Which vars in closure of f?

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

Values of functions: Closures

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

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

Free vs. Bound Variables

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

let $a = 20$;;
let $f$ $x$ =
  let $y = 1$ in
  let $g$ $z$ = $y + z$ in
    $a + (g$ $x$)
;;
f 0;;

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

Where do bound-vars values come from?

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

let $a = 20$;;
let $f$ $x$ =
  let $a = 1$ in
  let $g$ $z$ = $a + z$ in
    $a + (g$ $x$)
;;
f 0;;
Two questions about function values:

What is the value:

1. ... of a function?
2. ... of a function "application" (call)?

(\text{fun } x \rightarrow e)

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

1. Find \textit{closure} of e1
2. Execute body of \textit{closure} with param e2

Free values found in \textit{closure-environment}

Bound values by executing \textit{closure-body}

Q: What is the value of res?

1. Evaluate e1 in current-env to get (\textit{closure})
   \[= \text{code (formal } x \text{ + body } e \text{) + env } E\]
2. Evaluate e2 in current-env to get (argument) v2
3. Evaluate body e in env E extended with x := v2

(a) 4 \hspace{1cm} (b) 5 \hspace{1cm} (c) 6 \hspace{1cm} (d) 11 \hspace{1cm} (e) 12
Q: What is the value of \( \text{res} \)?

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

Application: \( f \ (x + y) \)

Eval \( \text{body} \) in \( \text{env} \) extended with \( \text{formal} \mapsto 5 \)

Eval \( x + y \) in \([x|->1, \ y|->5]\) \( \mapsto 6 \)

Q: What is the value of \( \text{res} \)?

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

Example 3

\[
\begin{align*}
\text{let } f \ g &= \\
& \begin{align*}
\text{let } x &= 0 \text{ in } \\
& \text{g } 2 \\
\end{align*} \\
\text{let } x &= 100;; \\
\text{let } h \ y &= x + y;; \\
\text{let } \text{res} &= f \ h;;
\end{align*}
\]

Example

\[
\begin{align*}
\text{let } x &= 1;; \\
\text{let } f \ y &= \\
& \begin{align*}
\text{let } x &= 2 \text{ in } \\
& \text{fun } z -> x + y + z \\
\end{align*} \\
\text{let } x &= 100;; \\
\text{let } g &= f \ 4;; \\
\text{let } y &= 100;; \\
\text{let } \text{res} &= f \ h;; \\
\end{align*}
\]

Example 3

\[
\begin{align*}
\text{let } f \ g &= \\
& \begin{align*}
\text{let } x &= 0 \text{ in } \\
& \text{g } 2 \\
\end{align*} \\
\text{let } x &= 100;; \\
\text{let } h \ y &= x + y;; \\
f \ h;;
\end{align*}
\]
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!”