If-then-else expressions

If (1 < 2) then 5 else 10

If (1 < 2) then ["ab", "cd"] else ["x"]

If-then-else is also an expression!
Can use any expression in then, else branch

\[
\begin{align*}
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 & : T \\
\end{align*}
\]

• Then-subexp, Else-subexp must have same type!
  - Equals type of resulting expression

\[
\begin{align*}
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 & : T \\
\end{align*}
\]
News

- PA 1 due tomorrow 11:59pm
- PA 2 out, and due next Fri at 5pm

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 the value of expression e to the variable x”

Later declared expressions can use 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”

<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

How ML deals with variables:

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

Environments and Evaluation

ML begins in a “top-level” environment
- Some names bound

\[
\text{let } x = e;;
\]

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”

Example

```ml
# 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 this 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
```

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 this 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
```

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 …`) 

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
```

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 …`) 

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
```
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 …)

Q: Why is this a good thing?
A: Function behavior frozen at point of declaration
- Nothing you can enter afterwards affects function
- Same inputs will always produce the same outputs
  - Localizes debugging
  - Localizes reasoning about the program
  - No “sharing” means no evil aliasing effects

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 to efficiently implement these “no-sharing” semantics
- Your job is to use the simplified no-sharing semantics to write correct, cleaner, readable, extendable systems

Function bindings

Functions are values, can bind using val

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

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

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

Occurences of fname inside e bound to “this” definition

```ocs
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-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 → 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 $e_2$

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

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 $E [x \mapsto v : t]$ to evaluate $e_2$

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

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 $e_2$

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

Correct Formatting

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

Correct Formatting
Example

```ocaml
let rec filter f l =
  if l = [] then []
  else
    let h = hd l in
    let t = filter f (tl l) in
    if (f h) then h::t else t
```

Nested function bindings

```ocaml
let a = 20;;
let f x =
  let y = 10 in
  let g z = y + z in
  a + (g x)
  ;;
f 0;
```

Recap

• Variables are names for values
  - Environment: dictionary/phonebook
  - Most recent binding used
  - Entries never changed, new entries added

• Environment frozen at fun definition
  - Re-binding variables cannot change a function
  - Same I/O behavior at every call

• Build complex expressions with local bindings
  - let-in expression
  - The let-binding is visible (in scope) inside in-expression
  - Elsewhere the binding is not visible
**Static/Lexical Scoping**

- For each occurrence of a variable, there is a **unique** place in program text where the variable was 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

**Next: Functions**

![Diagram of expressions, values, and types]

**Q**: What’s the value of a function?

**Functions**

Two ways of writing function expressions:

1. **Anonymous functions**:
   ```
   let fname = fun x -> e
   ```

2. **Named functions**:
   ```
   let fname x = e
   ```

**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 \( e_1 \)
The type of any function is:

- $T_1$: the type of the “input”
- $T_2$: the type of the “output”

```
let f = fun x -> e
```

$T_1 \rightarrow T_2$

$T_1, T_2$ can be any types, including functions!

Whats an example of?

- `int -> int`
- `int * int -> bool`
- `(int -> int) -> (int -> int)`

**Type of function application**

Application: fancy word for “call”

```
(e1 e2)
```

- “apply” argument $e_2$ to function value $e_1$

```
e1 : T1 -> T2
(e1 e2) : T2
```

- 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 functions: Closures

- "Body" expression not evaluated until application
  - but type-checking takes place at compile time
  - i.e. when function is defined
- Function value =
  - \(<\text{code} + \text{environment at definition}>\>
  - "closure"

Free (vs. Bound) Variables

Inside a function:
A "bound" occurrence:
1. Formal variable
2. Variable bound in let-in
   \(x, a, z\) are "bound" inside \(f\)

A "free" occurrence:
• Not bound occurrence
  \(a\) is "free" inside \(f\)

Environment at definition, frozen inside "closure", is used for values of free variables

Nested function bindings

let \(a = 20;\)

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

f 0;

Q: Where do values of bound variables come from?

Bound variable values determined when fun evaluated ("executed")
• From arguments
• Local variable bindings
  • Obtained from evaluation
Values of function application

Application: fancy word for “call”

\((e_1 \ e_2)\)

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

Application Value:
1. Evaluate \(e_1\) in current env to get (function) \(v_1\)
   - \(v_1\) is code + env
     - code is (formal \(x\) + body \(e\)), env is \(E\)
2. Evaluate \(e_2\) in current env to get (argument) \(v_2\)
3. Evaluate body \(e\) in env \(E\) extended by binding \(x\) to \(v_2\)

Example 1

```
let x = 1;;
let f y = x + y;;
let x = 2;;
let y = 3;;
f (x + y);;
```

Example 2

```
let x = 1;;
let f y =
  let x = 2 in
  fun z -> x + y + z
;;

let y = 100;;
let g = (f 4);;
let y = 100;;
(g 1);;
```
Example 2

```plaintext
let x = 1;;
let f y =
  let x = 2 in
  f x + y + z
;;
let x = 100;;
let g = (f 4);;
let y = 100;;
(g 1);
```

Example 3

```plaintext
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