Begin at the beginning ...

1. Programmer enters expression
2. ML checks if expression is “well-typed”
   - Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expr
3. ML evaluates expression to compute value
   - Of the same “type” found in step 2

Base Types

News

• PA 1 due (next) Fri 01/16, 11:59pm
• PA 2 out soon
• Office hours posted on Webpage:
  - Held in lab (CSE 250)
### Base Type: int

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2;</td>
<td>2</td>
</tr>
<tr>
<td>2+3;</td>
<td>5</td>
</tr>
<tr>
<td>7-4;</td>
<td>3</td>
</tr>
<tr>
<td>(2+3)*(7-4);</td>
<td>15</td>
</tr>
</tbody>
</table>

**Expressions built from sub-expressions**

**Types computed from types of sub-expressions**

**Values computed from values of sub-expressions**

---

### Base Type: float

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0 2.0 r</td>
<td>2.0 2.0</td>
</tr>
</tbody>
</table>

**Expressions built from sub-expressions**

**Types computed from types of sub-expressions**

**Values computed from values of sub-expressions**

---

### Base Type: string

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>“ab” “ab” s</td>
<td>“ab” “abcd”</td>
</tr>
</tbody>
</table>

**Expressions built from sub-expressions**

**Types computed from types of sub-expressions**

**Values computed from values of sub-expressions**

---

### Base Type: bool

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true true b</td>
<td>“ab”=”cd”</td>
</tr>
</tbody>
</table>

**Expressions built from sub-expressions**

**Types computed from types of sub-expressions**

**Values computed from values of sub-expressions**
**Base Type: bool**

- Equality testing is built-in for all expr, values, types
  - but compared expressions must have same type
- ...except for ?
  - function values ... why?

```
(“ab”=“cd”)  false  e1 = e2
```

```
e1:T  e2:T  e1⇒v1  e2⇒v2
```

```
e1=e2  ⇒  v1=v2
```

**Type Errors**

```
“pq” ^ 9;
```

```
e1:string  e2:string
```

```
(2 + “a”);
```

```
e1:int  e2:int
e1 + e2 : int
```

- Expressions built from sub-expressions
- Types computed from types of sub-expression
- If a sub-expression is not well-typed then whole expression is not well-typed

```
0 * (2 + “a”);
```

**Complex types: Tuples**

- Can be of any fixed size

```
(2+2 , 7>8);
```

```
int * bool
```

```
(9-3,”ab”^“cd”,7>8)
```

```
(int * string * bool)
```

- Elements can have different types
- Tuples can be nested in other tuples

```
e1:T1  e2:T2  ...  en:Tn
(e1, e2, ..., en) : T1 * T2 * ... * Tn
```

```
e1⇒v1  e2⇒v2  ...  en⇒vn
```

```
(e1, e2, ..., en) ⇒ (v1, v2, ..., vn)
```
Complex types: Records

```plaintext
{name="sarah";age=31;pass=false}
```

Records are tuples with named elements...

```plaintext
{name="sarah";age=31;pass=false}.age 31 int
{age=31;name="sarah";pass=false}.age 31 int
{age=31;name="sarah";pass=false}.pass false bool
```

Complex types: Lists

```plaintext
[1;"pq"]
```

- Unbounded size
- Can have lists of anything (e.g. lists of lists)

Complex types: list ..construct

```
1::[2;3] [1;2;3] Cons “operator”
```

All elements have the same type

```
[1;"pq"]
```

Can only “cons” element to a list of same type
Complex types: list construct

Append “operator”

\[1;2]@\[3;4]\] \[1;2;3;4\]

int list

\[e_1 : T \text{ list} \quad e_2 : T \text{ list}\]
\[e_1@e_2 \quad : T \text{ list}\]
\[e_1@v_1 \quad e_2 \quad \Rightarrow \quad v_2\]
\[e_1@e_2 \quad \Rightarrow \quad v_1@v_2\]

1@[“b”; “cd”];

[1]@[“b”; “cd”];

Can only append lists of the same type

Complex types: list deconstruct

Reading the elements of a list:

• Two “operators”: \(hd\) (head) and \(tl\) (tail)

<table>
<thead>
<tr>
<th>([1;2;3;4;5])</th>
<th>(hd) ([1;2;3;4;5])</th>
<th>(1)</th>
<th>(tl) ([1;2;3;4;5])</th>
<th>([2;3;4;5])</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td></td>
<td>int list</td>
<td>int list</td>
</tr>
<tr>
<td>“a”</td>
<td>“a”</td>
<td></td>
<td>“a”</td>
<td>“a”</td>
</tr>
<tr>
<td>“b”; “cd”</td>
<td>“b”; “cd”</td>
<td></td>
<td>“b”; “cd”</td>
<td>“b”; “cd”</td>
</tr>
<tr>
<td>([1,”a”];(7,”c”])</td>
<td>([1,”a”];(7,”c”])</td>
<td>1</td>
<td>([1,”a”];(7,”c”])</td>
<td>([7; “c”])</td>
</tr>
<tr>
<td>(int * string) list</td>
<td>(int * string) list</td>
<td></td>
<td>int list</td>
<td>int list</td>
</tr>
<tr>
<td>([[];[1;2;3];[4;5]])</td>
<td>([[];[1;2;3];[4;5]])</td>
<td>1</td>
<td>([[];[1;2;3];[4;5]])</td>
<td>([2;3;4;5])</td>
</tr>
</tbody>
</table>

List: Heads and Tails

Head

\[e : T \text{ list}\]
\[hd \quad e \quad : T\]
\[e \quad \Rightarrow \quad v_1::v_2\]

\(hd \quad e \quad \Rightarrow \quad v_1\)

Tail

\[e : T \text{ list}\]
\[tl \quad e \quad : T \text{ list}\]
\[e \quad \Rightarrow \quad v_1::v_2\]

\(tl \quad e \quad \Rightarrow \quad v_2\)

\((hd \quad [[];[1;2;3]]\) = \((hd \quad [[];[“a”]]))\)

Recap

1. Programmer enters expression
2. ML checks if expression is “well-typed”
   • Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expr
3. ML evaluates expression to compute value
   • Of the same “type” found in step 2
Recap

- Integers: +, -, *
- Floats: +, -, *
- Booleans: =, <, andalso, orelse, not
- Strings: ^

- Tuples, Records: 
  - Fixed number of values, of different types
- Lists: ::, @, hd, tl, null
  - Unbounded number of values, of same type

If-then-else expressions

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

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

<table>
<thead>
<tr>
<th>then-subexp</th>
<th>else-subexp</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1: bool</td>
<td>e2: T</td>
</tr>
<tr>
<td>if e1 then e2 else e3</td>
<td></td>
</tr>
</tbody>
</table>

If (1 < 2) then 5 else 10

5

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

[“ab”, “cd”]

string list

If-then-else expressions

- if (1 < 2) then 
- else expressions

if (1 < 2) then 5 else 10 5

if (1 < 2) then [“ab”, “cd”] else [“x”] [“ab”, “cd”]

string list

If-then-else expressions

- if 1>2 then [1, 2] else []
  - ...which is the type of resulting expression

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

if 1>2 then [1, 2] else []

[]

if 1<2 then [] else [“a”]

[]

int list

string list

(int 1>2 then [1, 2] else [])=(if 1<2 then [] else [“a”])
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”

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td>4 : int</td>
</tr>
<tr>
<td><code>y</code></td>
<td>64 : int</td>
</tr>
<tr>
<td><code>z</code></td>
<td>[4;64;68] : int list</td>
</tr>
<tr>
<td><code>x</code></td>
<td>8 : int</td>
</tr>
</tbody>
</table>
Environments and Evaluation

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

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

ML program = Sequence of variable bindings

Program evaluated by evaluating bindings in order
1. Evaluate `expr` 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

```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

```
  x 4 : int
  y 64 : int
  z [4;64;68] : int list
```

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

```
  x 4 : int
  y 64 : int
  z [4;64;68] : int list
```

```ml
# 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
```

Binding used to eval (f ...)

```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 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 ...)

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

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

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)

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 ↦ v : t] (only) to evaluate e2

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 $e_2$

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

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 $e_2$

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

Nested bindings

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

Example

```
let rec filter (f, l) =
    if l = [] then []
else
    let h = hd l in
    let y = 20 in
    let t = filter (f, tl l) in
    if (f h) then h::t else t
```
Nested function bindings

```plaintext
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

Recap

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