Announcements

• HOMEWORK 1 is out...

Read course policy
Must be your OWN work
Integrity

• No collaboration (only limited discussion)
• Rules will be enforced, with penalties
• No re-use of old solutions
• OK to discuss lectures, self-tests, concepts
• Ask us if not sure
Your weekly plan

• Class! Labs!
  • Study lecture slides and notes
  • Do the self-tests

• Homework
  • Start early, end on time

• Office hours

• Piazza only AFTER honest effort!
Walk-in tutoring

• Offered by Academic Development
• Hours to be announced
• Tutoring is by undergrad peers, not TAs
  • may help with conceptual material
• Also: use TA and staff office hours as needed

Don’t ask tutors to help with homework!
Today

• Types, expressions and values
• Declarations, binding and scope
• Patterns and matching
• Equality in ML
Types

• basic types    int, real, bool
• tuple types    int * int, int * int * real
• function types int -> int, real -> int * int
• list types     int list, (int -> int) list

t ::= int | bool | real
    | t1 * t2 * ... * tk
    | t1 -> t2
    | t list
Values

• For each type $t$ there is a set of (syntactic) values

• An expression of type $t$ evaluates to a value of type $t$ (or fails to terminate)
Values

- int: integers
- real: real numbers
- int list: lists of integers
- int -> int: functions from integers to integers
- (int -> int) -> (int -> int): functions from (functions...) to (functions...)
# Examples

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(3 + 4) \times 6$</td>
<td>$42 : \text{int}$</td>
</tr>
<tr>
<td>$(3.0 + 4.0) \times 6.0$</td>
<td>$42.0 : \text{real}$</td>
</tr>
<tr>
<td>$(42, 2+3)$</td>
<td>$(42, 5) : \text{int} \times \text{int}$</td>
</tr>
<tr>
<td>$\text{fn } x: \text{int} \Rightarrow x+42$</td>
<td>$\text{fn } x: \text{int} \Rightarrow x+42 : \text{int} \rightarrow \text{int}$</td>
</tr>
<tr>
<td>$\text{fn } x: \text{int} \Rightarrow 2+2$</td>
<td>$\text{fn } x: \text{int} \Rightarrow 2+2 : \text{int} \rightarrow \text{int}$</td>
</tr>
</tbody>
</table>

A function value has form $\text{fn } x:t \Rightarrow e$
# Examples

<table>
<thead>
<tr>
<th>expression</th>
<th>ML says value : type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fn (x:int):int =&gt; x+1</code></td>
<td><code>fn - : int -&gt; int</code></td>
</tr>
<tr>
<td><code>fn (x:real):real =&gt; x+1.0</code></td>
<td><code>fn - : real -&gt; real</code></td>
</tr>
<tr>
<td><code>Math.sin</code></td>
<td><code>fn - : real -&gt; real</code></td>
</tr>
</tbody>
</table>
Examples

function

\( \text{fn} \ (x: \text{int}, y: \text{int}) : \text{int}*\text{int} => (x \ \text{div} \ y, x \ \text{mod} \ y) \)

- has type \( \text{int}*\text{int} \rightarrow \text{int}*\text{int} \)
- is a value

application

\((\text{fn} \ (x: \text{int}, y: \text{int}) : \text{int}*\text{int} => (x \ \text{div} \ y, x \ \text{mod} \ y)) (42, 5)\)

- has type \( \text{int}*\text{int} \)
- evaluates to the value \((8, 2)\)
Declarations

fun divmod(x:int, y:int) : int*int = (x div y, x mod y)

binds divmod to the function value

fn (x:int, y:int) : int*int => (x div y, x mod y)

In scope of this declaration,

val (q:int, r:int) = divmod(42, 5)

binds q to 8, r to 2
• Bindings have static (syntactically fixed) scope

val x = 3.14;

let
  val x = 3.14
in
  end

local
  val x = 3.14
in
  end

(expressions and declarations using x)

(let
  val x = 3.14
in
  end

(an expression using x)

local
  val x = 3.14
in
  end

(a declaration using x)
Design issues

fun circ(r:real):real = 2.0 * pi * r

fun circ(r:real):real =
let
  val pi2:real = 2.0 * pi
in
  pi2 * r
end

local
  val pi2:real = 2.0 * pi
in
  fun circ(r:real):real = pi2 * r
end

2.0*pi only gets evaluated once

every call to circ evaluates 2.0*pi

every call to circ evaluates 2.0*pi
Summary

- An expression of type t can be evaluated
- If it terminates, we get a value of type t
- ML reports type and value
  - `val it = 3 : int`
  - `val it = fn - : int -> int`
- Declarations produce bindings
- Bindings are statically scoped

Use well scoped declarations to avoid re-evaluating code repeatedly
Lists

- \([1, 3, 2, 1, 21+21]\) : int list
- \([\text{true}, \text{false}, \text{true}]\) : bool list
- \([[[1],[2, 3]]]\) : (int list) list
- \([\ ]\) : int list, \([\ ]\) : bool list, \ldots
- \(1::[2, 3], 1::(2::[3]), 1::2::[3], 1::2::3::\text{nil}\)
- \([1, 2]@[3, 4]\)
- \(\text{nil} = [\ ]\)
Patterns

• Wildcard: _
• Variable: x
• Constant: 42, true, ~3
• Tuple: \((p_1, ..., p_k)\)
• List: nil, p_1::p_2, [p_1, ..., p_k]

Syntactic constraint:
no variable appears twice
in the same pattern

no constant patterns for reals or functions

where \(k \geq 0\) and \(p_1, ..., p_k\) are patterns
Matching

- A *pattern* can be *matched* against a *value*
- If the match *succeeds*, it produces *bindings*

  matching \texttt{d::L} against the value \( [2,4] \)
  succeeds with the bindings \( d:2, L:[4] \)

  matching \texttt{d::L} against the value \( [ ] \)
  fails
Matching

• Matching 42 against the value 42 succeeds
• Matching 42 against the value 0 fails
• Matching x against any value v succeeds with the binding x:v
• Matching _ against any value succeeds
Matching

- Matching $p_1::p_2$ against $[]$ fails

- Matching $p_1::p_2$ against $v_1::v_2$ fails
  if matching $p_1$ against $v_1$ fails,
  or matching $p_2$ against $v_2$ fails

- Matching $p_1::p_2$ against $v_1::v_2$ succeeds
  with bindings $L_1$ and $L_2$
  if matching $p_1$ against $v_1$ succeeds with $L_1$
  and matching $p_2$ against $v_2$ succeeds with $L_2$
Notes

• Use *patterns with variables* to give names to values that you *need* to use

• Use *wildcard* pattern when you don’t need to use the value

• Can annotate patterns with *types*, e.g.

  \[(x::L) : \text{int list}\]

  \[(x:\text{int}, y:\text{real})\]
\textbf{ML} =

- Infix operator \( e_1 = e_2 \)

- Only for expressions whose type is an \textit{equality} type, having a \textit{computable} check for exact equality

- Equality types include all types built from \texttt{int}, \texttt{bool}
  using tuple and list constructors, e.g.
  \texttt{int list}
  \texttt{int * bool * int}
  \texttt{(int * bool * int) list}

Standard ML…
\[
3 = 4; \\
\text{val it} = \text{false : bool}
\]
Using patterns

- `val p = e`
- `fn p => e`
- `fun f p = e`

or, more generally

```
fn p_1 => e_1 | ... | p_k => e_k
```

```
fun f p_1 = e_1 | ... | f p_k = e_k
```
Using patterns

```ml
fun check (x:int, y:int):bool =
  let
    val (q:int, r:int) = divmod(x, y)
  in
    (x = q*y + r)
  end
```

Introduces `check : int * int -> bool`

Binds `check` to a function value of this type

What does this function do?