15-150 Fall 2017

Stephen Brookes
Dilsun Kaynar

LECTURE 2
Thursday, August 31
Announcements

• HOMEWORK 1 is out...

• Due Tuesday 5 September, 11:59pm

Must be your OWN work

Read course policy and hand in signed commitment
Academic Integrity

- No collaboration on writing
- No re-use of old solutions
- It’s OK to discuss lectures, concepts, but NOT OK to copy code or proofs
- Rules are enforced, with penalties
- Ask us if you’re not sure
Your weekly plan

- Class! Labs! Study!
  - review slides and notes (posted online)
- Homework
  - start early, end on time
- Office hours
  - check availability, plan accordingly
- Use Piazza only AFTER honest effort, and not as deadline approaches!
Walk-in tutoring

- Offered by Academic Development
- Hours to be announced
- Tutoring is by undergrad peers, not TAs
  - may help with conceptual material

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

\[ t ::= \text{int} \]
\[ \mid \text{real} \]
\[ \mid t_1 * t_2 * \ldots * t_k \]
\[ \mid t \text{ list} \]
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...)

functions are values
Examples

<table>
<thead>
<tr>
<th>expression</th>
<th>value : type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3 + 4) * 6</td>
<td>42 : int</td>
</tr>
<tr>
<td>(3.0 + 4.0) * 6.0</td>
<td>42.0 : real</td>
</tr>
<tr>
<td>(42, 2+3)</td>
<td>(42, 5) : int * int</td>
</tr>
<tr>
<td>fn x:int =&gt; x+42</td>
<td>fn x:int =&gt; x+42 : int -&gt; int</td>
</tr>
<tr>
<td>fn x:int =&gt; 2+2</td>
<td>fn x:int =&gt; 2+2 : int -&gt; int</td>
</tr>
</tbody>
</table>

A function value has form  \( \text{fn } x: t \Rightarrow e \)
Standard ML of New Jersey [...]  
- 225 + 193 ;  
val it = 418 : int

Don’t forget the semi-colon.
ML reports the type and value.

225 + 193 = 418
225 + 193 =>* 418

runtime behavior consistent with math
Standard ML of New Jersey [...] 
- \textbf{fn} (x:int):int => 2+2;

val it = \textbf{fn} - : int \rightarrow int

ML reports the type and \textbf{fn} - 
2+2 does not get evaluated (yet)

- it 99;
val it = 4 : int
Examples

function

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

- has type int*int -> int*int
- is a value
```

application

```plaintext
(fn (x:int, y:int) : int*int => (x div y, x mod y)) (42, 5)

- has type int*int
- evaluates to the value (8, 2)
```
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
Shadowing

```plaintext
val pi : real = 3.14;  (* binds pi to 3.14 *)

fun area(r:real) : real = pi * r * r;
(* binds area to a function that uses 3.14 for value of pi *)

area 1.0;            (* val it = 3.14 : real *)

val pi : real = 3.14159;  (* binds pi to 3.14159 *)

fun new_area(r:real) : real = pi * r * r;
(* binds new_area to a function that uses 3.14159 for pi *)

area 1.0;            (* val it = 3.14 : real *)

new_area 1.0;        (* val it = 3.14159 : real *)
```
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

every call to circ evaluates 2.0*pi
every call to circ evaluates 2.0*pi

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

2.0*pi only gets evaluated once
Summary

• An expression of type \( t \) can be evaluated

• If evaluation terminates, we get a value of type \( t \)

• ML reports the type and value
  - \( \text{val it = 3 : int} \)
  - \( \text{val it = fn - : int -> int} \)

• Declarations produce bindings

• Bindings are statically scoped

*Use well scoped declarations to avoid re-evaluating code repeatedly*
List expressions

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

also \text{null}, \text{hd}, \text{tl}
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 to a value

• If the match succeeds, it produces bindings

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

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

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

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

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

- Matching $p_1::p_2$ to $v_1::v_2$ succeeds
  with bindings $L_1@L_2$
  if matching $p_1$ to $v_1$ succeeds with $L_1$
  and matching $p_2$ to $v_2$ 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.
  
  \[(x::L) : \text{int list}\]
  
  \[(x:\text{int}, y:\text{real})\]