CSE 341: Programming Languages

Section 1

Miranda Edwards

Thanks to Dan Grossman, Josiah Adams, and Cody A. Schroeder for the majority of this content
Hi, I’m Miranda

- I’m a senior in CS
- I was born and raised in Washington
- I’m a boxer, fought on UW’s team
- I run cross-country
- I think PL is pretty cool
Today’s Agenda

• Why Study PL?
• ML Development Workflow
  – Emacs
  – Using `use`
  – The REPL (Read–Eval–Print Loop)

• More ML
  – Shadowing Variables
  – Debugging Tips
  – Boolean Operations
  – Comparison Operations
Why study programming languages?

- Gets you out of OO-land
- Helps you pick the right language (tool) for a given task
- It’s (maybe) fun
Emacs Demo

- Recommended (not required) editor for this course
- Powerful, but the learning curve can at first be intimidating
- Not as scary as it looks
- Helpful Resource: CSE341 Emacs Guide
Important Emacs Commands

- **C-x C-s** - Save
- **C-x C-c** - Close
- **M-w** - Copy the highlighted text
- **C-y** – Paste
- **C-shift- ("ctrl-shift-dash")** - Undo

Note: M (Meta) is most likely the (Non-Mac) alt key, or (Mac) the flowery button or the option key.
Using `use`

```ml
use "foo.sml";
```

- Enters bindings from the file `foo.sml`
  - Like typing the variable bindings one at a time in sequential order into the REPL (more on this in a moment)

- Result is `()` bound to variable `it`
  - Ignorable
The REPL

• Read-Eval-Print-Loop is well named

• Conveniently run programs
  – Useful to quickly try something out
  – Save code for reuse by moving it into a persistent .sml file

• Expects semicolons

• For reasons discussed later, it’s dangerous to reuse use without restarting the REPL session
Debugging Errors

Your mistake could be:

- Syntax: What you wrote means nothing or not the construct you intended
- Type-checking: What you wrote does not type-check
- Evaluation: It runs but produces wrong answer, or an exception, or an infinite loop

Keep these straight when debugging even if sometimes one kind of mistake appears to be another
Play around

Best way to learn something: Try lots of things and don’t be afraid of errors

Work on developing resilience to mistakes
  – Slow down
  – Don’t panic
  – Read what you wrote very carefully

Maybe watching me make a few mistakes will help…
**Shadowing of Variable Bindings**

```ml
val a = 1; (* a -> 1 *)
val b = a; (* a -> 1, b -> 1 *)
val a = 2; (* a -> 2, b -> 1 *)
```

1. Expressions in variable bindings are evaluated “eagerly”
   - Before the variable binding “finishes”
   - Afterwards, the expression producing the value is irrelevant

2. Multiple variable bindings to the same variable name, or “shadowing”, is allowed
   - When looking up a variable, ML uses the latest binding by that name in the current environment

3. Remember, there is no way to “assign to” a variable in ML
   - Can only shadow it in a later environment
   - After binding, a variable’s value is an immutable constant
Try to Avoid Shadowing

- Shadowing can be confusing and is often poor style

- Why? Reintroducing variable bindings in the same REPL session may:
  - make it seem like wrong code is correct; or
  - make it seem like correct code is wrong.

```ocaml
val x = "Hello World";
val x = 2; (* is this a type error? *)
val res = x * 2; (* is this 4 or a type error? *)
```
Using a Shadowed Variable

- Is it ever possible to use a shadowed variable? Yes! And no…
- It can be possible to uncover a shadowed variable when the latest binding goes out of scope

```ocaml
val x = "Hello World";
fun add1(x : int) = x + 1; (* shadow x in func body *)
val y = add1 2;
val z = x^"!!"; (* "Hello World!!" *)
```
Use **use** Wisely

- **Warning:** Variable shadowing makes it dangerous to call `use` more than once without *restarting* the REPL session.

- *It may* be fine to repeatedly call `use` in the same REPL session, but unless you know what you’re doing, *be safe!*
  - Ex: loading multiple distinct files (with independent variable bindings) at the beginning of a session
  - `use`’s behavior is well-defined, but even expert programmers can get confused

- Restart your REPL session before repeated calls to `use`
### Boolean Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Type-checking</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>andalso</td>
<td>e1 andalso e2</td>
<td>e1 and e2 must have type bool</td>
<td>Same as Java’s e1 &amp;&amp; e2</td>
</tr>
<tr>
<td>orelse</td>
<td>e1 orelse e2</td>
<td>e1 and e2 must have type bool</td>
<td>Same as Java’s e1</td>
</tr>
<tr>
<td>not</td>
<td>not e1</td>
<td>e1 must have type bool</td>
<td>Same as Java’s !e1</td>
</tr>
</tbody>
</table>

- **not** is just a pre-defined function, but **andalso** and **orelse** must be built-in operations since they cannot be implemented as a function in ML.
  - Why? Because **andalso** and **orelse** “short-circuit” their evaluation and may not evaluate *both* e1 and e2.

- Be careful to always use **andalso** instead of **and**.
- **and** is completely different. We will get back to it later.
Style with Booleans

Language does not need `andalso`, `orelse`, or `not`

\[
(* \text{e1 andalso e2} *) \\
\text{if e1} \\
\text{then e2} \\
\text{else false}
\]

\[
(* \text{e1 orelse e2} *) \\
\text{if e1} \\
\text{then true} \\
\text{else e2}
\]

\[
(* \text{not e1} *) \\
\text{if e1} \\
\text{then false} \\
\text{else true}
\]

Using more concise forms generally much better style

And definitely please \[
(* \text{just say e (!!!)} *) \\
\text{if e} \\
\text{then true} \\
\text{else false}
\]
Comparisons

For comparing int values:

\[
= \ < > \ < \ >= \ <=
\]

You might see weird error messages because comparators can be used with some other types too:

- \( > < >= <= \) can be used with real, but not 1 int and 1 real
- \( = <> \) can be used with any “equality type” but not with real
  - Let’s not discuss equality types yet