CSE 341: Programming Languages

Section 1

Spencer Pearson

(Thu 9:30-10:30, CSE 220)

Thanks to Dan Grossman, Konstantin Weitz, Josiah Adams, and Cody A. Schroeder for the majority of this content

Today's Agenda

- ML Development Workflow
  - The REPL (Read–Eval–Print Loop)
  - Emacs
  - Using `use`

- More ML
  - Shadowing Variables
  - Debugging
  - Comparison Operations
  - Boolean Operations
  - Testing

The REPL

- Read-Eval-Print-Loop is well named
- Useful for quickly trying things out
  (but save code for reuse by putting it in a .sml file)
  - Expects semicolons

- (P.S.: `rlwrap` *might* be useful.)

Emacs Demo

- Recommended (not required) editor for this course
- Powerful, but the learning curve can at first be intimidating

Using `use`

```
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
- It's dangerous to reuse `use` without restarting the REPL session! Definitions linger.

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

Work on developing resilience to mistakes:

- Slow down
- Don’t panic
- Read what you wrote very carefully
- Preventative medicine: testing!

Shadowing of Variable Bindings

```
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

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

1. 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

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

- 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*.

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

```
val threshold = 10;
(* threshold -> 10 *)
fun is_big(x : int) = x > threshold;
(* threshold -> 10, is_big -> (function) *)
val threshold = 20;
(* threshold -> 20, is_big -> (function) *)
val z = is_big 15;
```

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`

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

Boolean Operations

Operation	Syntax	Type-checking	Evaluation
`andalso`	`e1 andalso e2`	`e1` and `e2` must have type `bool`	Same as Java's `e1 && e2`
`orelse`	`e1 orelse e2`	`e1` and `e2` must have type `bool`	Same as Java's `e1 || e2`
`not`	`not e1`	`e1` must have type `bool`	Same as Java's `!e1`

- `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?

  `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 different. We will get back to it later.

Testing

Write tests for your code!

```
val test1 = (abs 2 = 2);
val test2 = (abs 0 = 0);
```