CSE 341: Programming Langs

Section 1
Justin Harjanto

Hi! I’m Justin ^_^

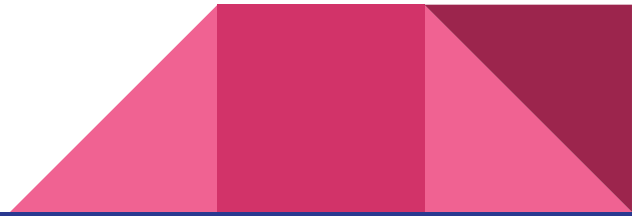
Senior in CS

Love PL!!!

- Huge Haskell fan

Third time TAing for 341

Vim!



Today

Motivation for this course

SML workflow, errors, and booleans

What’s so exciting about this class and why should I care?

Functional programming!

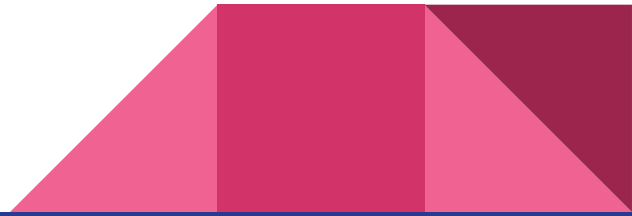
- Completely different style from what you’re probably used to
  - No loops, only recursion, no mutation etc...
  - Concise code!!!

Lot of features present in other languages

- May never write a line of the langs we cover again
- But features from FP languages have seeped into “mainstream” languages
  - Will highlight

What does this do?

```
let f = filterM $ const [True, False]
```



Java solution:

```
public static <T> Set<Set<T>> powerSet(Set<T> originalSet) {
	Set<Set<T>> sets = new HashSet<Set<T>>();
	if (originalSet.isEmpty()) {
		sets.add(new HashSet<T>());
		return sets;
	}
	List<T> list = new ArrayList<T>(originalSet);
	T head = list.get(0);
	Set<T> rest = new HashSet<T>(list.subList(1, list.size()));
	for (Set<T> set : powerSet(rest)) {
		Set<T> newSet = new HashSet<T>();
		newSet.add(head);
		newSet.addAll(set);
		sets.add(newSet);
		sets.add(set);
	}
	return sets;
}
```

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

The REPL

- Read-Eval-Print-Loop is well named
- Conveniently run programs: **`C-c C-s`**
  - 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
  - End the REPL session with **`C-d`**

Shadowing of Variable Bindings

```
val a = 1; (* a -> 1 *)
val b = a * 10; (* a -> 1, b -> 10 *)
val a = 2; (* a -> 2, b -> 10 *)
```

- Expressions in variable bindings are evaluated “eagerly”
  - Before the variable binding “finishes”
  - Afterwards, the expression producing the value is irrelevant
- Multiple variable bindings to the same variable name, or “shadowing”, is allowed
  - When looking up a variable, ML uses the most recent binding by that name in the current environment
- 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 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
  - The behavior of `use` is well-defined, but even expert programmers can get confused
- Restart your REPL session before repeated calls to `use`

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…

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? 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`

```
(* e1 andalso e2 *)
if e1
then e2
else false
```

```
(* e1 orelse e2 *)
if e1
then true
else e2
```

```
(* not e1 *)
if e1
then false
else true
```

```
(* just say e (!!!) *)
if e
then true
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 a mixture of 1 **`int`** and 1 **`real`**
- **`= <>`** can be used with any “equality type” but not with **`real`**
  - Let’s not discuss equality types yet