CSE 341:
Programming Languages

Section AC with Nate Yazdani

about me

- CSE BS/MS student (last year of BS)
- I am *really* into programming languages
- I am *really* into research on programming languages
  - program synthesis
  - formal verification
  - crazy theoretical stuff (“homotopy type theory??”)

PLSE

about you?

why are we here

- to get a bit more interactive learning
- to supplement the material from lecture
- to take a closer look at important but subtle details
- to ask questions (please)! :-)

agenda

- ML development workflow
  - emacs
  - using **`use`**
  - REPL
- some more ML
  - variable shadowing
  - debugging with the REPL
  - boolean operations
  - comparison operations

emacs

- recommended (not required) editor for this course
- a powerful tool for programming
- learning curve may seem steep, but you get the hang of it more quickly than you'd think
- Dan's emacs guide is super helpful
- if you need help with setup, please let us know

emacs demo time

using `use`

```
use “foo.sml”;
```

- parses the local file `foo.sml` and then evaluates the bindings one after another
- result is the dummy value `()`
  - automatically bound to variable `it`
  - completely safe to ignore

the REPL

- stands for the “read-eval[uate]-print-loop”
  - it reads, evaluates, prints, and loops!

- works with both expressions and bindings
  - expects semicolons to know when to evaluate

- handy to quickly try stuff out
  - in emacs, start with `C-c C-s` and end with `C-d`

- as we will see in a bit, **`use`**-ing multiple files without restarting your REPL session is dangerous

shadowing

```
val a = 1; (* a -> 1 *)
val b = a * 10; (* a -> 1, b -> 10 *)
val a = 42; (* a -> 1, b -> 10, a -> 42 *)
```

- eager" evaluation of expressions in variable bindings
  - computes the value and *then* binds the name to that value
  - afterwards, the original expression is forgotten

- multiple variable bindings to the same variable name is called "**shadowing**"
  - affects both static and dynamic environments
  - ML will use the most-recently bound value in the current environment

- remember: there is no variable "assignment" in ML
  - you can only shadow it in a later environment
  - once bound, a variable's value is an immutable constant

avoid shadowing

- it can confuse yourself and (especially) others
- it's often considered poor style
- why? shadowing variables in a REPL session may
  - make *wrong* code seem *correct*
  - make *correct* code seem *wrong*
  - this can easily happen when you re-**`use`** a file

using a shadowed variable

- is it ever possible to use a shadowed variable?
  - yes!
  - and also no…
    - when the shadowing binding of a variable name goes out of **scope**, the shadowed binding is available again
      - environments are like a “lookup stack”

```
val x = “Hello World”;
fun plus1 (x : int) = x + 1;
val y = plus1 2;
val z = x ^ “!!”; (* …, z -> “Hello World!!” *)
```

be careful with `use`

- **warning:** variable shadowing makes it dangerous to call `use` multiple times without restarting the REPL session

- it *might* be safe to call `use` more than once in the same REPL session, but think twice about it
  - at the beginning of a session, loading distinct files with distinct variable names is probably fine
  - while the behavior of `use` is well-defined, even experts can easily get confused

- best to always restart the REPL session

debugging errors

- your mistake could be
  - syntactic: the source code means nothing (not in the ML grammar) or something unintended

```
val 0 = x
```

  - typing: the code fails to typecheck

```
3 + true
```

  - semantic (evaluation): the program's behavior is not what you want, *e.g.*, raises an exception, computes the wrong value, or loops infinitely

```
val three = 2 + 2
```

- keep these straight when debugging
- sometimes one kind of mistake will appear 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
  - reconsider what assumptions you're making

- maybe it will help to see me make some mistakes?

let’s give it a try

boolean operations

operation	syntax	type-checking	semantics (evaluation)
conjunction	`e1 andalso e2`	**e1** and **e2** must have type **bool**	same as **&&** in Java
disjunction	`e1 orelse e2`	**e1** and **e2** must have type **bool**	same as **||** in Java
negation	`not e`	**e** must have type **bool**	same as **!e** in Java

- **not** is essentially just a pre-defined function
- **andalso** and **orelse** must be built in, because they "short-circuit" and may not always evaluate **e2**
- be careful to not use **and** instead of **andalso**
  - they mean totally different things

booleans with ***style***

- ML 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
```

- more concise forms are generally better style
- definitely please don’t do this

```
(* just say e (!!!)” *)
if e then true else false
```

comparisons

- you can compare two **`int`** values with

  `=  <>  >  <  >=  <=`

- you might get weird errors messages because these operators work with some other types too

- **`>  <  >=  <=`** also work with two real values but *not* with one **`int`** and one **`real`**

- **`=  <>`** work with any two values of the same "equality type" but not with **`real`**
  - we'll hear more about equality types later

thanks!