PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Section 1

TA Name

Winter 2018

Adapted from slides by Dan Grossman, Eric Mullen and Ryan Doenges

Introduction

About TA

Course Resources

- We have a ton of course resources. Please use them!
- If you get stuck or need help:
  - Ask questions in Google Group
  - Email the staff list! cse341-staff@cs.washington.edu
  - Come to Office Hours (on website, you don't need a list of topics before you decide to stop by)
- We're here for you

Agenda

- Setup: get everything running
- Emacs Basics
- ML development workflow
- Shadowing
- Comparison Operators
- Boolean Operators
- Debugging
- Testing

Setup

- Excellent guide located on the course website: https://courses.cs.washington.edu/courses/cse341/18wi/software-setup/sml_emacs.pdf
- We're going to spend about 5 minutes setting up now (so you can follow along for the rest of section)
- You need 3 things installed:
  - Emacs
  - SML
  - SML mode for Emacs

Emacs Basics

- Don't be scared!
- Commands have particular notation: C-x means hold Ctrl while pressing x
- Meta key is Alt (thus M-z means hold Alt, press z)
  - C-x C-s is Save File
  - C-x C-f is Open File
  - C-x C-c is Exit Emacs
- C-g is Escape (Abort any partial command you may have entered)
- Consult the installation guide

ML Development Workflow

- REPL is the general term for tools like "Run I/O" you have been using in jGRASP for CSE 142/3
- REPL means **R**ead **E**val **P**rint **L**oop
- Read: ask the user for semi colon terminated input
- Evaluate: try to run the input as ML code
- Print: show the user the result or any error messages produced by evaluation
- Loop

ML Development Workflow

- Demo of REPL with lecture 1 code
  - You can type in any ML code you want, it will evaluate it
  - Useful to put code in .sml file for reuse
  - Every command must end in a semicolon (;)
  - Load .sml files into REPL with `use` command

Shadowing

```
val a = 1;
val b = 2;
val a = 3;
```

a -> int
a -> int, b -> int
a -> int, b -> int, a -> int

- You can't change a variable, but you can add another with the same name
- When looking for a variable definition, most recent is always used
- Shadowing is usually considered bad style

Shadowing

```
val a = 1;
val b = 2;
val a = 3;
```

a -> 1
a -> 1, b -> 2
a -> 1, b -> 2, a -> 3

- You can't change a variable, but you can add another with the same name
- When looking for a variable definition, most recent is always used
- Shadowing is usually considered bad style

Shadowing

- This behavior, along with `use` in the REPL can lead to confusing effects
- Suppose I have the following program:

```
val x = 8;
val y = 2;
```

- I load that into the REPL with `use`. Now, I decide to change my program, and I delete a line, giving this:

```
val x = 8;
```

- I load that into the REPL without restarting the REPL. What goes wrong?
- (Hint: what is the value of y?)

Comparison Operators

- You can compare numbers in SML!
- Each of these operators has 2 subexpressions of type `int`, and produces a `bool`

= (Equality)	< (Less than)	<= (Less than or equal)
<> (Inequality)	> (Greater than)	>= (Greater than or equal)

Boolean Operators

- You can also perform logical operations over `bool`s!

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

- `and` is completely different, we will talk about it later
- `andalso/orelse` are SML built-ins as they use short-circuit evaluation, we will talk about why they have to be built-ins later

And… Those Bad Styles

- 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
```

- Using more concise forms generally much better style
- And definitely please do not do this:

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

Debugging

- DEMO
- Errors can occur at 3 stages:
  - Syntax: Your program is not “valid SML” in some (usually small and annoyingly nitpicky) way
  - Type Check: One of the type checking rules didn’t work out
  - Runtime: Your program did something while running that it shouldn’t
- The best way to debug is to read what you wrote carefully, and think about it.

Testing

- We don't have a unit testing framework (too heavyweight for 5 weeks)
- You should still test your code!

```
val test1 = ((4 div 4) = 1);

(* Neat trick for creating hard-fail tests: *)

val true = ((4 div 4) = 1);
```