CSE 341 AC

Yuma & Taylor
University of Washington

Today’s agenda

- Homework 2 check-in
- SML standard library documentation
- Polymorphic datatypes
- Higher-order functions
  - The basics
  - Unnecessary function wrapping
  - Returning functions from functions
  - `map`, `flat_map`, `filter`, `fold`

Homework #2 check-in

Due this Friday at 11:00pm PST. Homework #1 feedback should be out (or will be soon).

- How are things going?
- Any questions before we dive in?

SML Standard Library

- Standard ML Basis Library: http://sml-family.org/Basis/
- Modules/structures/signatures… we'll get to this shortly, and you can ignore it for now.
- Look in "Required Structures", click the link you're interested in.

Example: http://sml-family.org/Basis/string.html#SIG:STRING.explode:VAL

Polymorphic datatypes

- You can use 'a, 'b, etc when defining your own datatypes!
- Example: defining a binary tree that can store different type data in its leaf nodes (data of type 'a) vs branch nodes (data of type 'b)

```
datatype ('a, 'b) tree = Leaf of 'a
                       | Node of 'b * ('a, 'b) tree
                                    * ('a, 'b) tree
```

```
datatype ('a, 'b) tree = Leaf of 'a
                       | Node of 'b * ('a, 'b) tree
                                  * ('a, 'b) tree
```

You can create trees:

```
Node("hi", Leaf true, Leaf false) : (bool, string) tree
```

```
Node("hi", Leaf true, Leaf 7) : does not typecheck!
```

```
datatype ('a, 'b) tree = Leaf of 'a
                       | Node of 'b * ('a, 'b) tree
                                    * ('a, 'b) tree
```

You can create trees:

`Node("hi", Leaf true, Leaf false) : (bool, string) tree`

`Node("hi", Leaf true, Leaf 7)` : does not typecheck!

Higher-order functions: overview

Recall that, up until now, we have seen functions types like:

```
val tomorrow = fn : date -> date
```

or

```
val add = fn : (int * int) -> int
```

Higher-order functions: overview

- But! Functions are *first-class* citizens in SML, meaning they can be passed as values to anything that accepts them.
- Examples:
  - `val map = fn : ((‘a -> ‘b) * ‘a list) -> ‘b list`
  - `val filter = fn : ((‘a -> bool) * ‘a list) -> ‘a list`

  Don’t worry if you don’t understand these yet, we’ll go through them one-by-one.

Higher-order functions: unnecessary function wrapping

Recall earlier that we encouraged boolean zen, i.e., to rewrite

```
if e then true else false
```

as

```
e
```

Higher-order functions: unnecessary function wrapping

The same applies to functions! If you create an anonymous function to pass as an argument elsewhere, like:

```
fn x => f x
```

you can instead write:

```
f
```

Higher-order functions: returning functions

We can return functions from other functions:

```
fun f x = (* int -> (int -> int) *)
  if x > 0
  then fn y => 2 * y
  else fn y => 42
```

What does this do?

Higher-order functions: demo

Let's write `map`, `flat_map`, `filter`, `fold`.

(Code posted afterwords on the course webpage.)