CSE 341 Section 3

Nicholas Shahan
Spring 2016

Adapted from slides by Cody A. Schroeder, and Dan Grossman

Today's Agenda

- Standard Library Documentation (for HW3)
- Anonymous Functions
  - "Unnecessary Function Wrapping"
  - Returning Functions
- High-Order Functions
  - Map
  - Filter
  - Fold
- More Practice
  - Tree example
  - Expression example

2

What is in a Standard Library?

- Things that you simply can't implement on your own.
  - Creating a timer, opening a file, etc.
- Things that are so common a "standardized" version will save you time and effort
  - List.map, string concatenation, etc.
  - A standard library makes writing and reading code easier.
  - Common operations don't have to be implemented, and are immediately recognizable.

3

Standard Library Documentation

Online Documentation

- http://www.standardml.org/Basis/index.html
- http://www.smlnj.org/doc/smlnj-lib/Manual/toc.html

Helpful Subset

- Top-Level http://www.standardml.org/Basis/top-level-chapter.html
- List http://www.standardml.org/Basis/list.html
- ListPair http://www.standardml.org/Basis/list-pair.html
- Real http://www.standardml.org/Basis/real.html
- String http://www.standardml.org/Basis/string.html

4

Anonymous Functions

```
fn pattern => expression
```

- An expression that evaluates to a new function with no name
- Usually used as an argument or returned from a higher-order function
- Almost equivalent to the following:

```
let fun name pattern = expression in name end
```

- The difference is that anonymous functions cannot be recursive!

5

"Unnecessary Function Wrapping"

`fn x => f x` vs. `f`

- When called both functions will evaluate to the same result
- However, one creates an unnecessary function to wrap `tl`
- Compare to:

`if e1 then true else false` vs. `e1`

Bad Style: Lose Points	Good Style: Happy TA ☺
`if x > 0 then true else false`	`x > 0`
`n_times((fn ys => tl ys), 3, xs)`	`n_times(tl, 3, xs)`

6

Returning Functions

- Remember - Functions are first-class values
  - We can return them from functions
- Example:

```
fun double_or_triple  f =
     if f 7
     then fn x => 2 * x
     else fn x => 3 * x
```

- Has type **(int -> bool) -> (int -> int)**
- The REPL will print **(int -> bool) -> int -> int** because it never prints an unnecessary parenthesis

7

High-order Hall of Fame

```
fun map (f, xs) =
   case xs of
      [] => []
   | x::xs' => (f x)::(map(f, xs'))
```

```
fun filter (f, xs) =
   case xs of
      [] => []
   | x::xs' => if f x
               then x::(filter(f, xs'))
               else filter(f, xs')
```

8

Fold

- **Fold** (synonyms/close relatives *reduce*, *inject*, etc.) is another very famous iterator over recursive structures
- Accumulates an answer by repeatedly applying a function **f** to the answer so far
  - **fold(f, acc,[x1, x2, x3, x4])** computes **f(f(f(f(acc, x1),x2),x3),x4)**

```
fun fold (f, acc, xs) =
   case xs of
      []     => acc
    | x::xs'  => fold(f, f(acc, x), xs')
```

```
val fold = fn : ('a * 'b -> 'a) * 'a * 'b list -> 'a
```

9

Practice - Tree Example

```
(* Generic Binary Tree Type *)
datatype 'a tree = Empty
               | Node of 'a * ' a tree * ' a tree

(* Apply a function to each element in a tree. *)
val tree_map = fn: (' a -> 'b) * 'a tree -> 'b tree

(* Returns true iff the given predicate returns true
when applied to each element in a tree. *)
val tree_all = fn: (' a -> bool) * 'a tree -> bool
```

10

Practice - Expression Example

```
(* Modified expression datatype from lecture 5. Now
   there are variables. *)
datatype exp = Constant of int
             | Negate of exp
             | Add of exp * exp
             | Multiply of exp * exp
             | Var of string
(* Do a post order traversal of the given exp. At each
node, apply a function f to it and replace the node with
the result. *)
val visit_post_order = fn : (exp -> exp) * exp -> exp
(* Simplify the root of the expression if possible. *)
val simplify_once = fn : exp -> exp
(* Almost the same as evaluate but leaves variables
alone. *)
val simplify = fn : exp -> exp
```

11