CSE341 – Section 3

Standard-Library Docs, Unnecessary Function Wrapping, Map, & More

Cody A. Schroeder

January $24^{th}, 2013$



Interlude



- Anonymous
- Style Point
- Higher-Order



Standard Basis 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

Interlude

Questions

- How's life?
- Tail-recursion?
- Pattern-matching?

Note

• Extra Lecture Material: http://www.cs.washington.edu/ education/courses/cse341/13wi/videos/unit3/

(* x2=15 *)

(* ×3=12 *)

 $(* \times 4 = 15 *)$

Anonymous Functions

An Anonymous Function

fn pattern => expression

- An expression that creates a new function with no name.
- Usually used as an argument to 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!!!

Simple Example

- ¹ **fun** doSomethingWithFive f = f 5;
- ² val x1 = doSomethingWithFive (fn x => x*2); (* x1=10 *)
- ³ val $x^2 = (fn x => x+9) 6;$
- 4 val cube = fn x => x*x*x;
- $_5$ val x3 = cube 4;
- 6 **val** x4 = doSomethingWithFive cube;

Anonymous Functions

What's the difference between the following two bindings?

val name = fn pattern => expression; fun name pattern = expression;

- Once again, the difference is recursion.
- However, excluding recursion, a **fun** binding could just be syntactic sugar for a **val** binding and an anonymous function.
- This is because there are no recursive val bindings in SML.

Anonymous Functions (cont.)

Previous Example

```
1 fun n_times (f,n,x) = if n=0
2 then x
3 else f (x_times (f, n-1, x));
4
5 fun square x = x*x;
6 fun increment x = x+1;
7
8 val x1 = n_times (square, 4, 7);
9 val x2 = n_times (increment, 4, 7);
10 val x3 = n_times (tl, 2, [4,8,12,16]);
```

With Anonymous Functions

1 val
$$x1 = n_{times}$$
 (fn $x = x x, 4, 7$);

² val
$$x^2 = n_{times}$$
 (fn $x = > x+1, 4, 7$);

$$_{3}$$
 val x3 = n_times (fn xs => tl xs, 2, [4,8,12,16]); (* Bad Style *)

Unnecessary Function Wrapping



- Other than style, these two expressions result in the exact same thing.
- However, one creates an unnecessary function to wrap tl.
- This is very similiar to this style issue:

(**if** ex **then** true **else** false)

VS.

ex

Higher-Order Functions

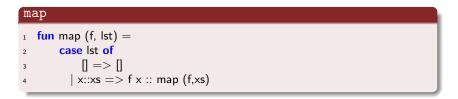
• A function that returns a function or takes a function as an argument.

Two Canonical Examples

- map : ('a -> 'b) * 'a list -> 'b list
 - Applies a function to every element of a list and return a list of the resulting values.
 - Example: map (fn x => x*3, [1,2,3]) === [3,6,9]
- filter : ('a -> bool) * 'a list -> 'a list
 - Returns the list of elements from the original list that, when a predicate function is applied, result in true.
 - Example: filter (fn x => x>2, [~5,3,2,5]) === [3,5]

Note: List.map and List.filter are similarly defined in SML but use currying. We'll cover these later in the course.

Defining map and filter



filter

$\begin{array}{ll} & \text{case Ist of} \\ & & [] => [] \\ & & x::xs => \text{if } f x \\ & & \text{then } x::filter (f, xs) \end{array}$
$4 \qquad x::xs => if f x$
5 then x::filter (f. xs)
6 else filter (f, xs)

Broader Idea

Functions are Awesome!

- SML functions can be passed around like any other value.
- They can be passed as function arguments, returned, and even stored in data structures or variables.
- Functions like map are very pervasive in functional languages.
 - A function like map can even be written for other data structures such as trees.

Returning a function

Tree Example

exp Example

```
1 (* Modified expression datatype from lecture 5. Now there are
       variables . *)
2
   datatype exp = Constant of int
3
                   Negate of exp
4
                   Add of exp * exp
                   Multiply of exp * exp
6
                  Var of string
8
   (* Do a post-order traversal of the given exp. At each node, apply a
9
      function f to it and replace the node with the result . *)
10
   val visitPostOrder = fn : (exp -> exp) * exp -> exp
11
12
   (* Simplify the root of the expression if possible. *)
13
   val simplifyOnce = \mathbf{fn} : exp -> exp
14
15
  (* Almost the same as evaluate but leaves variables alone. *)
16
   val simplify = fn : exp -> exp
17
```