CSE341 – Section 3
Standard-Library Docs, Unnecessary Function Wrapping, Map, & More

Cody A. Schroeder

January 24th, 2013
1. SML Docs
   - Standard Basis

2. Interlude

3. First-Class Functions
   - Anonymous
   - Style Point
   - Higher-Order

4. Example
Online Documentation

http://www.standardml.org/Basis/index.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

Anonymous Functions

An Anonymous Function

\[
\text{fn \ pattern } \Rightarrow \text{ expression}
\]

- 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:

\[
\text{let fun \ name \ pattern } = \text{ expression \ in \ name \ end}
\]

- The difference is that anonymous functions cannot be recursive!!!

Simple Example

1. \text{fun doSomethingWithFive f = f 5;}
2. \text{val x1 = doSomethingWithFive (fn x => x*2); (\star x1=10 \star)}
3. \text{val x2 = (fn x => x+9) 6; (\star x2=15 \star)}
4. \text{val cube = fn x => x*x*x;}
5. \text{val x3 = cube 4; (\star x3=12 \star)}
6. \text{val x4 = doSomethingWithFive cube; (\star x4=15 \star)}
Anonymous Functions

What’s the difference between the following two bindings?

```ml
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 x2 = n_times (fn x => x+1, 4, 7);
3  val x3 = n_times (fn xs => tl xs, 2, [4,8,12,16]); (* Bad Style *)
```
What’s the difference between the following two expressions?

((fn xs => tl xs) vs. tl)

**STYLE POINTS!**

- Other than style, these two expressions result in the exact same thing.
- However, one creates an unnecessary function to wrap `tl`.
- This is very similar 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**: \((\alpha \rightarrow \beta) \times \alpha \text{ list} \rightarrow \beta \text{ list}\)
  - Applies a function to every element of a list and return a list of the resulting values.
  - Example: \(\text{map (fn x => x*3, [1,2,3])} === [3,6,9]\)

- **filter**: \((\alpha \rightarrow \text{bool}) \times \alpha \text{ list} \rightarrow \alpha \text{ list}\)
  - Returns the list of elements from the original list that, when a predicate function is applied, result in true.
  - Example: \(\text{filter (fn x => x>2, [~5,3,2,5])} === [3,5]\)

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`

**map**

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

**filter**

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

```ml
fun piecewise x = if x < 0.0
              then fn x => x * x
              else if x < 10.0
                   then fn x => x / 2.0
                   else fn x => 1.0 / x + x
```
Tree Example

1 (∗ Generic Binary Tree Type ∗)
2
3 (** Apply a function to each element in a tree. ∗)
4 val treeMap = fn : ('a -> 'b) * 'a tree -> 'b tree

5 (∗ Returns true iff the given predicate returns true when applied to
6 each element in a tree. ∗)
7 val treeAll = fn : ('a -> bool) * 'a tree -> bool
Modified expression datatype from lecture 5. Now there are variables.

```sml
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.

```sml
val visitPostOrder = fn : (exp -> exp) * exp -> exp
```

Simplify the root of the expression if possible.

```sml
val simplifyOnce = fn : exp -> exp
```

Almost the same as evaluate but leaves variables alone.

```sml
val simplify = fn : exp -> exp
```