SML Docs Interlude First-Class Functions Example

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

Cody A. Schroeder

January 24th, 2013

SML Docs

Interlude

First-Class Functions
Anonymous Style Point Higher-Order

Example

Standard Basis

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

Questions
How’s life?
Tail-recursion?
Pattern-matching?

Note
Anonymous Functions

**Anonymous Function**

- 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:
  
  ```sml```
  ```
  let fun name pattern = expression
  in
  name
  end
  ```
  
  The difference is that anonymous functions cannot be recursive!!!

**Simple Example**

```sml```
```ml```
fun doSomethingWithFive f = f 5;
val x1 = doSomethingWithFive (fn x = x * 2);
val x2 = doSomethingWithFive (fn x = x + 9) 6;
val cube = fn x = x * x * x;
val x3 = cube 4;
val x4 = doSomethingWithFive cube;
```ml```
```sml```

What's the difference between the following two bindings?

```sml```
```ml```
val name = fn pattern => expression;
fun name pattern = expression;
```ml```
```sml```
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.

Unnecessary Function Wrapping

What's the difference between the following two expressions?

```sml```
```ml```
(fn xs => tl xs)
vs.
```ml```
```sml```
This is very similar to this style issue:

```sml```
```ml```
(fn x => if x then x else false)
vs.
```ml```
```sml```
```ml```
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.

Broader Idea

- 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

```
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

```
datatype 'a tree = Empty
  | Node of 'a * 'a tree * 'a tree

val treeMap = fn : ('a -> 'b) * 'a tree -> 'b tree
val treeAll = fn : ('a -> bool) * 'a tree -> bool
```
 datatype exp = Constant of int
 | Negate of exp
 | Add of exp * exp
 | Multiply of exp * exp
 | Var of string

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

 val simplifyOnce = fn : exp -> exp

 val simplify = fn : exp -> exp