Agenda

1. SML Docs
   • Standard Basis

2. First-Class Functions
   • Anonymous
   • Style Points
   • Higher-Order

3. Examples
Standard Basis Documentation

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
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:

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

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

Simple Example

```ml
fun doSomethingWithFive f = f 5;
val x1 = doSomethingWithFive (fn x => x*2); (* x1=10 *)
val x2 = (fn x => x+9) 6; (* x2=15 *)
val cube = fn x => x*x*x;
val x3 = cube 4; (* x3=64 *)
val x4 = doSomethingWithFive cube; (* x4=125 *)
```
Anonymous Functions

What's the difference between the following two bindings?

```plaintext
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

```ml
fun n_times (f,n,x) = if n=0
  then x
  else f (n_times (f, n-1, x));

fun square x = x*x;
fun increment x = x+1;

val x1 = n_times (square, 2, 3);
val x2 = n_times (increment, 4, 7);
val x3 = n_times (tl, 2, [4,8,12,16,20]);
```

With Anonymous Functions

```ml
val x1 = n_times (fn x => x*x, 2, 3);
val x2 = n_times (fn x => x+1, 4, 7);
val x3 = n_times (fn xs => tl xs, 2, [4,8,12,16,20]);(*Bad Style*)
```
Unnecessary Function Wrapping

What's the difference between the following two expressions?

\[(\text{fn } xs \to \text{tl } xs)\quad \text{vs.} \quad \text{tl}\]

**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:
  
  \[(\text{if } ex \text{ then true else false})\quad \text{vs.} \quad ex\]
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

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

filter
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

```plaintext
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

(*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 treeMap = fn : ('a -> 'b) * 'a tree -> 'b tree```

(* Returns true iff the given predicate returns true when applied to each element in a tree. *)

```val treeAll = fn : ('a -> bool) * 'a tree -> bool```