CSE341
Section 3
Standard-Library Docs, First-Class Functions, & More
Agenda

1. SML Docs
   • Standard Basis
2. Polymorphic Datatypes
3. First-Class Functions
   • Anonymous
   • Style Points
   • Higher-Order
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
Higher-Order Functions

**Definition:** A function that returns a function or takes a function as an argument.

- 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.
- Generalized functions such as these are **very** pervasive in functional languages (and are starting to creep into more Object-Oriented ones too ala Java!)

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

Canonical Higher-Order Functions
map

- map : ('a -> 'b) * 'a list -> 'b list

**What does the type tell us?**
- What are the arguments?
- What is the return type?
- What could be a hypothesis for what this function is supposed to do?

- **map** 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

- **filter** 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]

**What could be the type of this function?**

- What are the arguments?
- What is the return type?
- What could be a hypothesis for what this function is supposed to do?

- filter : ('a -> bool) * 'a list -> 'a list
fold

- **fold**: ('a * 'b -> 'a) * 'a * 'b list -> 'a
  - Returns a “thing” that is the accumulation of the first argument applied to the third arguments elements stored in the second argument.
  - Example: `fold((fn (a,b) => a + b), 0, [1,2,3]) === 6`
Polymorphic Datatypes

Suppose we want to create a tree datatype

• A node can be a leaf
• A node can be the root of a subtree
Polymorphic Datatypes

We solve this problem by having polymorphic datatypes:

datatype (‘a, ‘b) tree =
  Leaf of ‘a
  | Node of ‘b * (‘a, ‘b) tree * (‘a, ‘b) tree
Anonymous Functions

An Anonymous Function

\texttt{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:
\texttt{let fun name pattern = expression in name end}

What’s the difference? What can you do with one that you can’t do with the other?
- The difference is that anonymous functions cannot be recursive!!!
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.
Unnecessary Function Wrapping

What's the difference between the following two expressions?

\[(\text{fn} \; \text{xs} \Rightarrow \text{tl} \; \text{xs})\] vs. \[\text{tl}\]

**STYLE POINTS!**

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

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