1. SML Docs
   • Standard Basis

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

1. Examples

---

**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!!!

---

**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?

(\( fn \) \( xs \) \( => \) \( tl \) \( xs \) \) ) \( \text{vs.} \) \( \text{tl} \)

**STYLE POINTS!**

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

```
(if ex then true else false) \( \text{vs.} \) \( \text{ex} \)
**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: \( \text{map (fn x => x*3, \{1,2,3\})} \equiv \{3,6,9\} \)

- **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: \( \text{filter (fn x => x>2, \{-5,3,2,5\})} \equiv \{3,5\} \)

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

**Broader Idea**

**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.

(Let's see some examples!)

**Polymorphic Datatypes**

(*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
```