```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Standard Equivalences

Syntactic sugar

Using or not using syntactic sugar is always equivalent

- By definition, else not syntactic sugar

Example:

```
fun f x =
    x andalso g x
```

$=$

```
fun f x =
    if x
    then g x
    else false
```

But be careful about evaluation order

```
fun f x =
    x andalso g x
```

$\neq$

```
fun f x =
    if g x
    then x
    else false
```

Standard equivalences

Three general equivalences that always work for functions

- In any (?) decent language

1. Consistently rename bound variables and uses

```
val y = 14
fun f x = x+y+x
```

$\equiv$

```
val y = 14
fun f z = z+y+z
```

But notice you can't use a variable name already used in the function body to refer to something else

```
val y = 14
fun f x = x+y+x
```

$\not\equiv$

```
val y = 14
fun f y = y+y+y
```

```
fun f x =
  let val y = 3
  in x+y end
```

$\not\equiv$

```
fun f y =
  let val y = 3
  in y+y end
```

Standard equivalences

Three general equivalences that always work for functions

- In (any?) decent language

2. Use a helper function or do not

```
val y = 14
fun g z = (z+y+z)+z
```

$\equiv$

```
val y = 14
fun f x = x+y+x
fun g z = (f z)+z
```

But notice you need to be careful about environments

```
val y = 14
val y = 7
fun g z = (z+y+z)+z
```

$\not\equiv$

```
val y = 14
fun f x = x+y+x
val y = 7
fun g z = (f z)+z
```

Standard equivalences

Three general equivalences that always work for functions

- In (any?) decent language

3. Unnecessary function wrapping

```
fun f x = x+x
fun g y = f y
```

$\equiv$

```
fun f x = x+x
val g = f
```

But notice that if you compute the function to call and *that computation* has side-effects, you have to be careful

```
fun f x = x+x
fun h () = (print "hi";
            f)
fun g y = (h()) y
```

$\not\equiv$

```
fun f x = x+x
fun h () = (print "hi";
            f)
val g = (h())
```

One more

If we ignore types, then ML let-bindings can be syntactic sugar for calling an anonymous function:

```
let val x = e1
in e2 end
```

```
(fn x => e2) e1
```

- These both evaluate **`e1`** to **`v1`**, then evaluate **`e2`** in an environment extended to map **`x`** to **`v1`**
- So *exactly* the same evaluation of expressions and result

But in ML, there is a type-system difference:

- **`x`** on the left can have a polymorphic type, but not on the right
- Can always go from right to left
- If **`x`** need not be polymorphic, can go from left to right