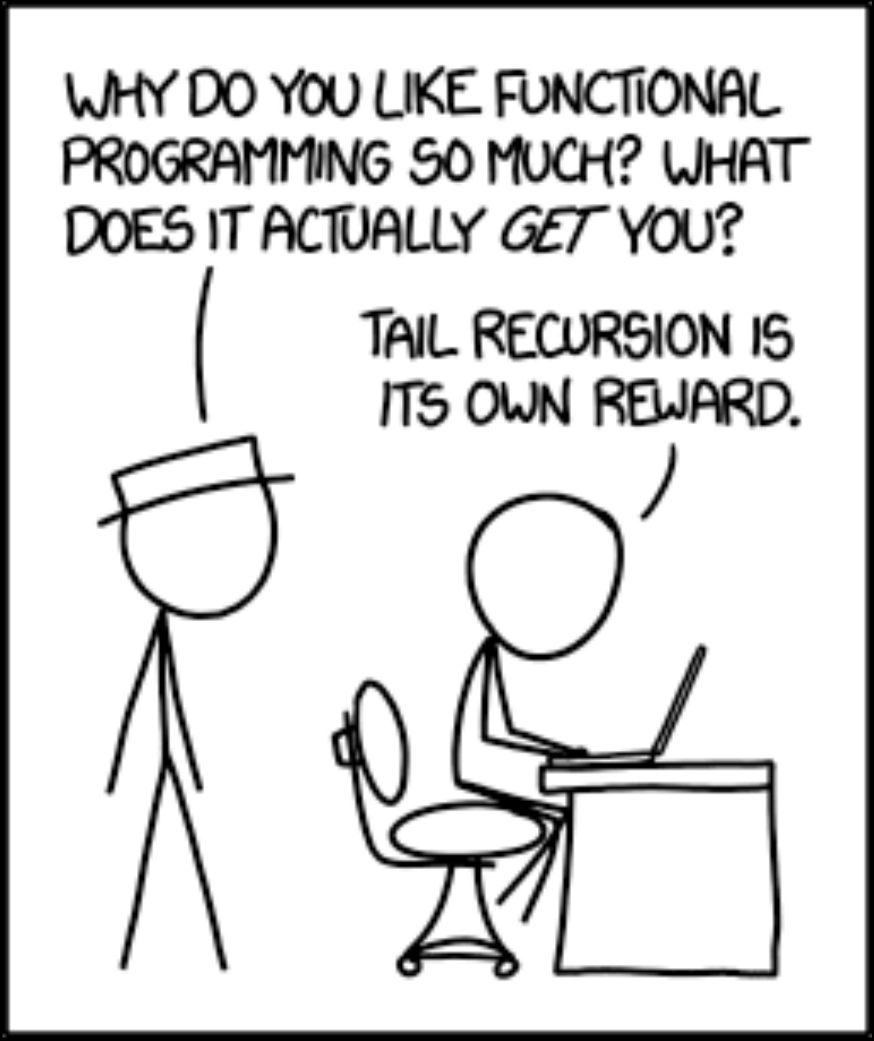


http://xkcd.com/1270/

CS 252:

Advanced Programming Language Principles

Lambdas & Higher-Order Functions

Prof. Tom Austin

San José State University

Lambdas

- Based on the lambda calculus
- Analogous to anonymous classes in Java

Lambda Example

```
Prelude> (\x -> x+1) 1
2
Prelude> (\x y -> x*y) 2 3
6
Prelude>
```

Function composition

```
f(g(x))
```

can be rewritten as

```
(f . g) x
```

Points-free style

```
inc x = x + 1
incByTwo = inc . inc
```

Points-free: no function argument

Lambdas & Function Composition

```
Prelude> let f = (\x -> x - 5)
                 . (\y -> y * 2)
Prelude> f 7
9
Prelude> let f = (\x y -> x - y)
                 . (\z -> z * (-1))
Prelude> f 3 4
-7
```

Tail Recursion

Iterative solutions tend to be more efficient than recursive solutions.

However, compilers are very good at optimizing a tail recursive functions.

In tail recursion,
the recursive call is
the last step performed
before returning
a value.

Is this function tail-recursive?

```
public int factorial(int n) {
    if (n==1) return 1;
    else {
        return n * factorial(n-1);
    }
}
```

No: the last step is multiplication

Is this function tail-recursive?

```
public int factorialAcc(int n, int acc)
{
    if (n==1) return acc;
    else {
        return factorialAcc(n-1, n*acc);
    }
}
```

Yes: the recursive step is the last thing we do

Which version is tail-recursive?

```
fact :: Integer -> Integer
fact 1 = 1
fact n = n * (fact $ n - 1)

fact' :: Integer -> Integer -> Integer
fact' 0 acc = acc
fact' n acc = fact' (n - 1) (n * acc)
```

Is this version tail-recursive?

```
fact2 :: Integer -> Integer -> Integer
fact2 n acc = if n == 0
     then acc
     else fact2 (n - 1) (n * acc)
```

This argument is called an "accumulator" – common design pattern to make your functions tail-recursive.

Higher-order functions

Programs as Functions

Functional languages treat programs as mathematical functions.

Definition: *A function is a rule that associates to each x from some set X of values a unique y from a set of Y values.*

Definition: *A function is a rule that associates to each x from some set X of values a unique y from a set of Y values.*

f is the name of the function

$$y = f(x)$$

Definition: *A function is a rule that associates to each x from some set X of values a unique y from a set of Y values.*

x is a variable in the set X

$$y = f(x)$$

X is the *domain* of f.
$x \in X$ is the *independent variable*.

Definition: *A function is a rule that associates to each x from some set X of values a unique y from a set of Y values.*

$$y = f(x)$$

y is a variable in the set Y

Y is the *range* of f.

$y \in Y$ is the *dependent variable*.

Qualities of Functional Programing

1. Functions clearly distinguish
   - incoming values (parameters)
   - outgoing values (results)
2. No assignment
3. No loops
4. Return value depends only on params
5. *Functions are first class values*

Functions are first-class data values, so we can:

- Pass as arguments to a function
- Return from a function
- Construct new functions dynamically

A function that either takes a function as a parameter or returns a function as its result is a **higher-order function**

Consider:

```
addNums x y = x + y
```

I mean to type

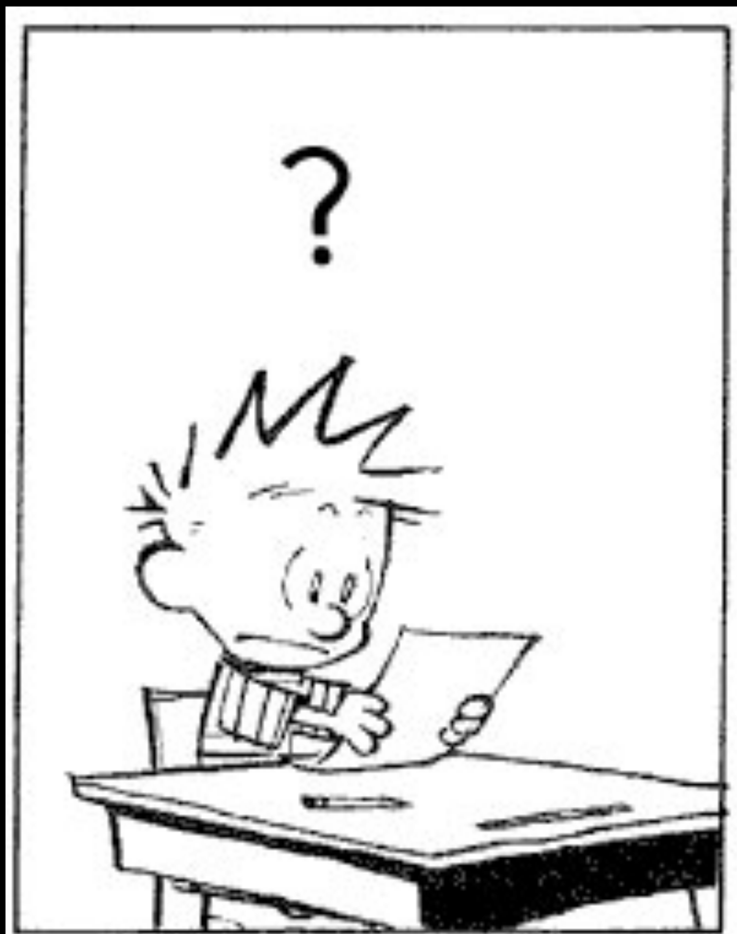
```
3 * addNums 5 2
```

But accidentally type

```
3 * addNums 52
```

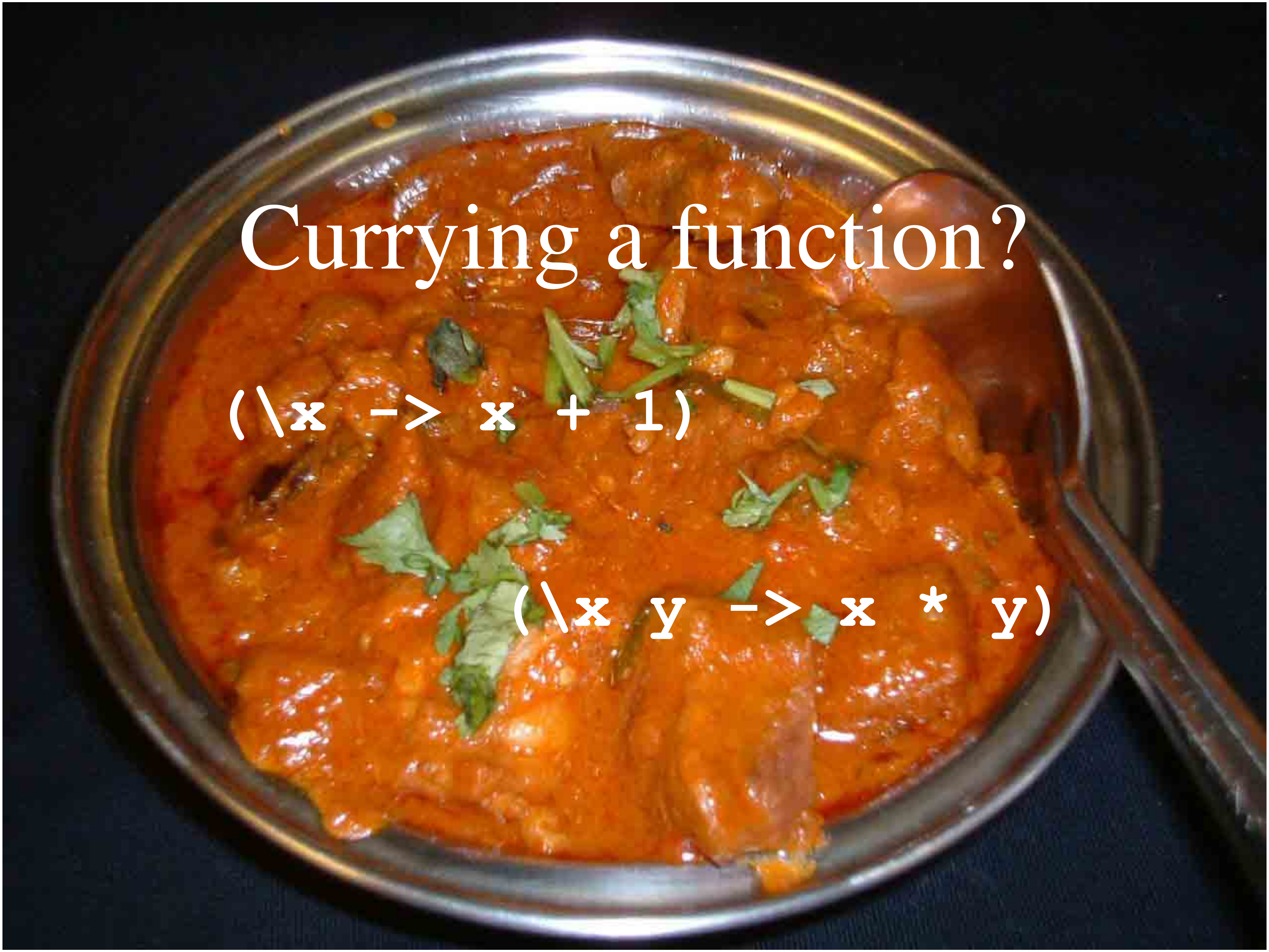
What happens?

```
Non type-variable argument in the
constraint: Num (a -> a)
(Use FlexibleContexts to permit
this)
When checking that ‘it’ has the
inferred type
 it :: forall a.
   (Num a, Num (a -> a)) => a -> a
```



Why does Haskell give such strange error messages?

The answer is that Haskell *curries* functions.


Currying a function?
(\x -> x + 1)
(\x y -> x * y)

Function Currying

Transform a function w/ multiple arguments into multiple functions

Haskell Brooks Curry

Function currying

- Note the type of our Haskell function
  - `addNums :: Num a => a -> a -> a`
- `addNums` is a function that takes in a number *and returns a function that takes another number*

Higher order functions

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Motivation for higher order functions

(in-class)

Fold left

`foldl` applies a function to each sequential pair of elements in a list

This is the *accumulator*

- `foldl (\x y -> x+y) 0 [1,2,3]`
- `foldl (\x y -> x+y) (0+1) [2,3]`
- `foldl (\x y -> x+y) ((0+1)+2) [3]`
- `foldl (\x y -> x+y) (((0+1)+2)+3) []`
- `(((0+1)+2)+3)`
- `6`

Fold right

`foldr` folds from the right, and works on infinite lists

Note that we can pass '+' as a function

- `foldr (+) 0 [1,2,3]`
- `1+(foldr (+) 0 [2,3]))`
- `1+(2+(foldr (+) 0 [3]))`
- `1+(2+(3+(foldr (+) 0 []))))`
- `1+(2+(3+(0)))`
- `6`

`foldr` on an infinite list

- `take 3 $ foldr (:) [] [1..]`
- `take 3 $ 1:foldr (:) [] [2..]`
- `take 3 $ 1:2:foldr (:) [] [3..]`
- `take 3 $ 1:2:3:foldr (:) [] [4..]`
- `[1,2,3]`

`foldl` (& `foldr`) build a *thunk* rather than calculate the results as it goes.

> let z = foldl (+) 0 [1..10000000]

Returns quickly

> z

Slow – result needs to be computed

Definition: a *thunk* is a *delayed computation*.

`foldl'` – Efficient left fold

- `foldl'` evaluates its results *eagerly* rather than *lazily*.
- To use, first:
```
import Data.List
```
- https://wiki.haskell.org/Foldr_Foldl_Foldl' has more details.

Which fold should I use?

- **`foldr`** – "`foldr` is not only the right fold, it is almost commonly the *right* fold to use…"
- **`foldl'`** – large, but finite lists
- **`foldl`** – specialized cases only

Related reading

- Learn You a Haskell, Chapter 6 (online)
- https://wiki.haskell.org/Foldr_Foldl_Foldl'
- https://wiki.haskell.org/Foldl_as_foldr

Lab 3: Higher order functions

Available in Canvas and on the course website.

http://www.cs.sjsu.edu/~austin/cs252-fall17/labs/lab3/lab3.lhs