CS 381: Programming Language Fundamentals

Summer 2015

Introduction to Functional Programming in Haskell

June 23, 2015

Outline

Basics.hs

Haskell Basics

What is functional programming?

- What is a function?
- Equational reasoning
- First-order vs. higher-order functions
- Lazy evaluation

How to functional program

Outline

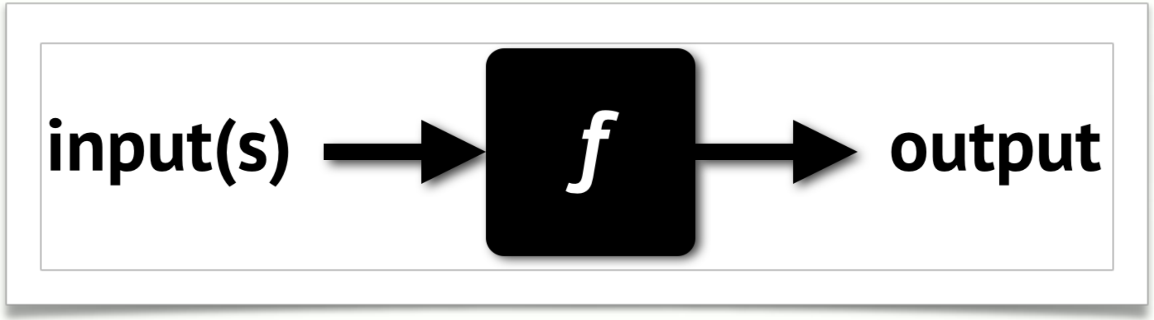
Haskell Basics

What is functional programming?

- What is a function?
- Equational reasoning
- First-order vs. higher-order functions
- Lazy evaluation

How to functional program

What is a (pure) function?



A function is **pure** if:

- it *always* returns the same output for the same inputs
- it doesn't do anything else — no "side effects"

In Haskell: whenever we say "function" we mean **pure function**!

What are (and aren't) functions?

Always functions:

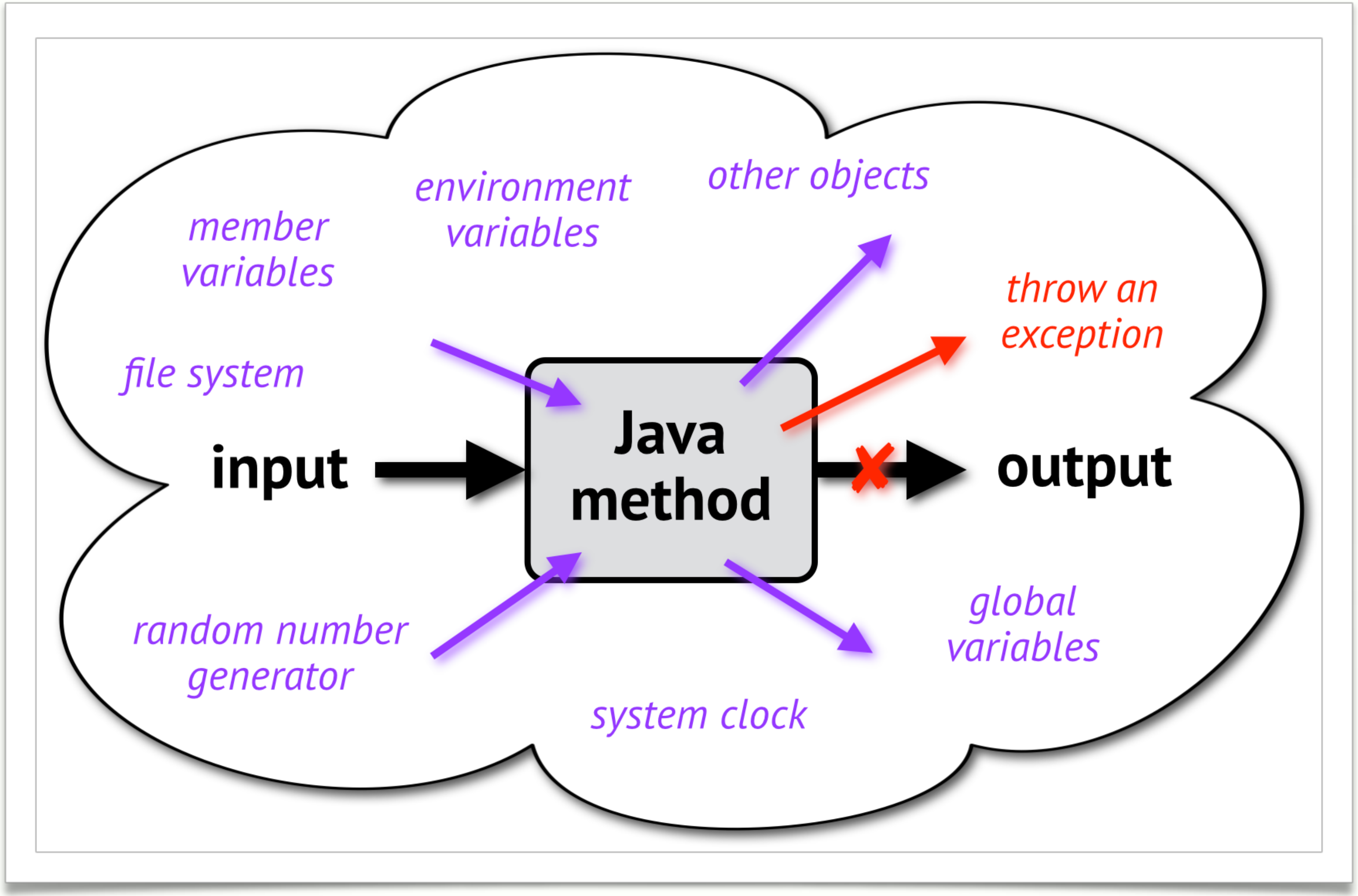
- mathematical functions $f(x) = x^2 + 2x + 3$
- encryption and compression algorithms

Usually not functions:

- C, Python, JavaScript,… "functions" (procedures)
- Java, C#, Ruby,… methods

Haskell **only** allows you to write (pure) functions!

Why procedures/methods aren't functions



Outline

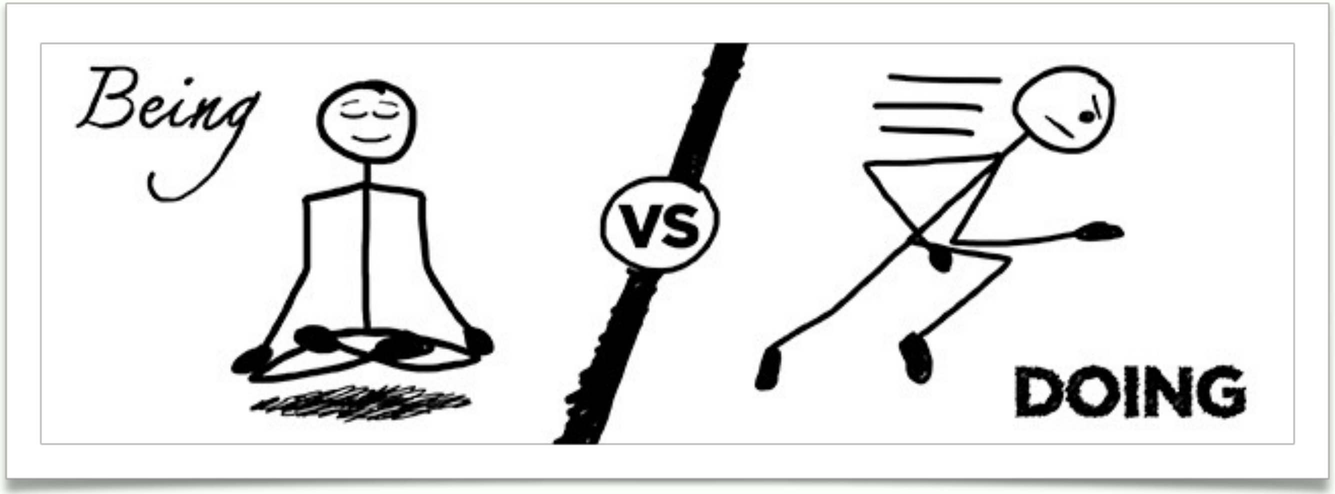
Haskell Basics

What is functional programming?

- What is a function?
- Equational reasoning
- First-order vs. higher-order functions
- Lazy evaluation

How to functional program

Getting into the Haskell mindset



Haskell

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x ++ sum xs
```

Java

```
int sum (List < Int > xs) {
    int s = 0;
    for (int x : xs) {
        s = s + x;
    }
    return s;
}
```

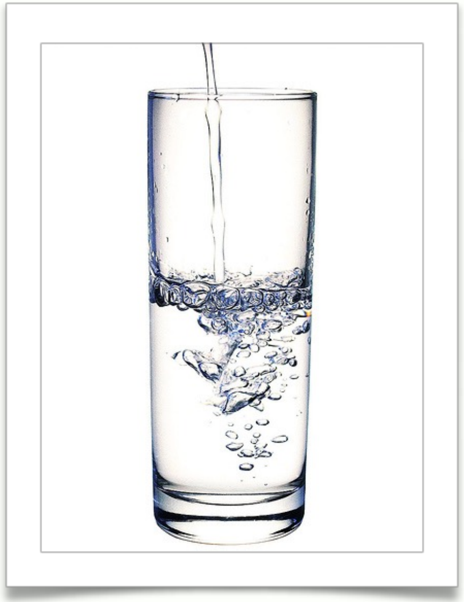
Same symbol, different meaning!

Referential transparency

An expression can be replaced by its **value** without changing the overall program behavior (**value** a.k.a **referent**)

```
  length [1,2,3] + 4
⇒              3 + 4
```

what if `length` was a Java method?



Corollary: an expression can be replaced by **any expression** with the same value without changing program behavior

Supports **equational reasoning**

Equational reasoning

Computation is just **substitution**!

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x ++ sum xs
```

```
  sum [2,3,4]
⇒ sum (2:(3:(4:[])))
⇒ 2 + sum (3:(4:[]))
⇒ 2 + 3 + sum (4:[])
⇒ 2 + 3 + 4 + sum []
⇒ 2 + 3 + 4 + 0
⇒ 9
```

So then how to I *do anything* in Haskell?

Simple answer…**you don't**!

Instead you **describe**!

Describing computations

Function definition: a list of **equations** that relate input to output

Example: reversing a list

- **imperative view**: how do I rearrange the elements in a list?
- **functional view**: how is a list related to its reversal?

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Exercise: use equational reasoning to compute the reverse of the list `[2,3,4,5]`

Exercise: using equational reasoning

```
reverse :: [a] -> [a]
reverse []     =[]
reverse (x:xs) = reverse xs ++ [x]
```

Pattern matching:

1. conditional
2. bindings

```
reverse [2,3,4,5]                     =
reverse [3,4,5] ++ [2]                =
reverse [4,5] ++ [3] ++ [2]           =
reverse [5] ++ [4] ++ [3] ++ [2]      =
reverse [] ++ [5] ++ [4] ++ [3] ++ [2] =
[] ++ [5] ++ [4] ++ [3] ++ [2]        = [5,4,3,2]
```


Oregon State University

Four steps to learning how to program

Language implementation — how to evaluate programs

Output — how to run programs

Program — how to write programs

Input — how to define programs

Four steps to learning Haskell

Language implementation — how to evaluate programs

how to evaluate **expressions**

Output — how to run programs

how to **apply functions**

Program — how to write programs

how to **define functions**

Input — how to define programs

how to **define types** and **values**

Outline

Haskell Basics

What is functional programming?

- What is a function?
- Equational reasoning
- First-order vs. higher-order functions
- Lazy evaluation

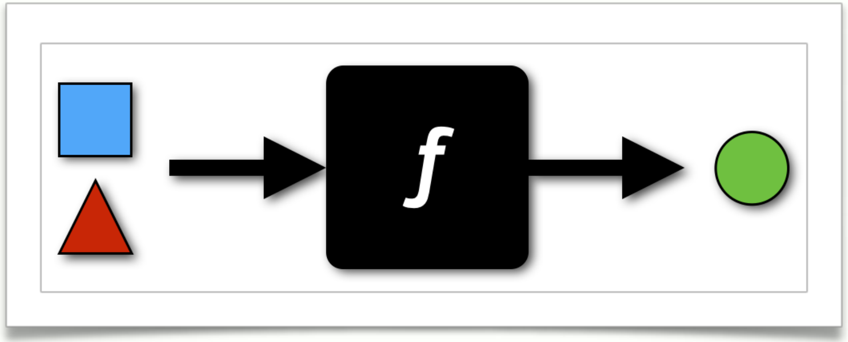
How to functional program

First-order functions

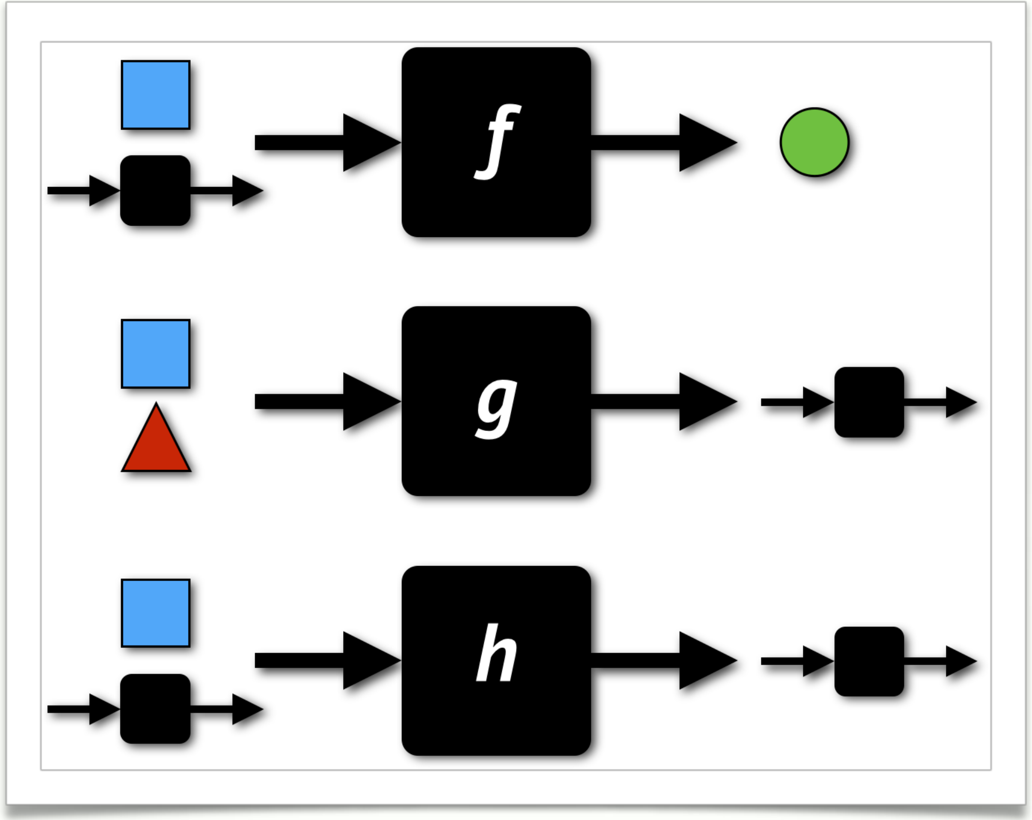
Basics.hs

Examples

```
cos    :: Float -> Float
even   :: Int -> Bool
length :: [a] -> Int
```

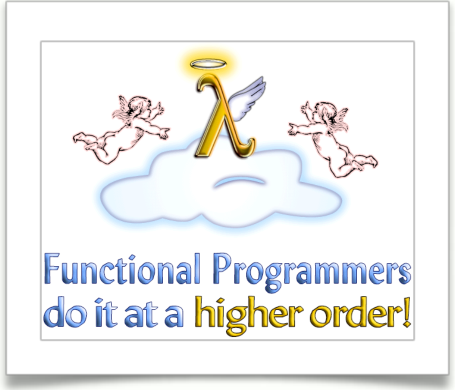


Higher-order functions



Examples

```
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
(.)    :: (b -> c) -> (a -> b) -> a -> c
```



Higher-order functions as control structures

Basics.hs

`map`: *loop for doing something to each element in a list*

```
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

```
map f [2,3,4,5] = [f 2,f 3,f 4,f 5]
```

```
map even [2,3,4,5]
                    = [even 2, even 3, even 4, even 5]
                    = [True,False,True,False]
```

`foldr`: *loop for aggregating elements in a list*

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f y []      = y
foldr f y (x:xs) = f x (foldr f y xs)
```

```
foldr f y [2,3,4] = f 2(f 3(f 4 y))
```

```
foldr (+) [2,3,4]
                  = (+) 2 ((+) 3 ((+) 4 0))
                  = 2 (3 + (4 + 0))
                  = 9
```


Oregon State University

Function composition

Create new functions by **composing** existing functions

- apply the *second* function to the input
- *then* apply the *first* function to output

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

Function composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Existing functions (types)

```
not  :: Bool -> Bool
succ :: Int -> Int
even :: Int -> Bool
head :: [a] -> a
tail :: [a] -> [a]
```

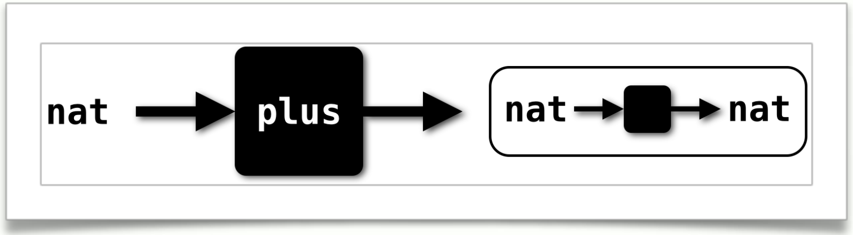
New function definitions

```
plus2  = succ . succ
odd    = not . even
second = head . tail
drop2  = tail . tail
```

Currying/partial application

In Haskell, functions that take multiple arguments are **implicitly higher order**

```
plus :: Int -> Int -> Int
```



Curried `plus 2 3`
```
plus :: Int -> Int -> Int
```

Uncurried `plus (2,3)`
```
plus :: (Int,Int) -> Int
```

Partial application
```
increment :: Int -> Int
increment = plus 1
```

Haskell Curry

Exercises

Is the function **`th`** well defined? Yes

If so, what does it do and what is its type? Takes the tail of a list's head

```
th :: ??
th = tail . head
```

```
th :: [[a]] -> [a]
th = tail . head
```

```
head :: [a] -> a
tail :: [a] -> [a]
```

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Exercises

Implement **revmap** using *pattern matching*

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map (x:xs) = f x : map f xs
```

```
revmap :: (a -> b) -> [a] -> [b]
revmap f [] = []
revmap (x:xs) = revmap f xs ++ [f x]
```

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ x
```

…using *function composition*

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
revmap :: (a -> b) -> [a] -> [b]
revmap f = map f . reverse
```

Outline

Haskell Basics

What is functional programming?

- What is a function?
- Equational reasoning
- First-order vs. higher-order functions
- Lazy evaluation

How to functional program

Lazy evaluation

InfList.hs

In Haskell expressions are **reduced** (evaluated):

- only when needed
- at most once

```
calculate :: Int -> Int -> Int
calculate a b = if a < 100 then a + a
                           else b
```

Supports:

- infinite data structures
- separation of concerns (maybe later)

Outline

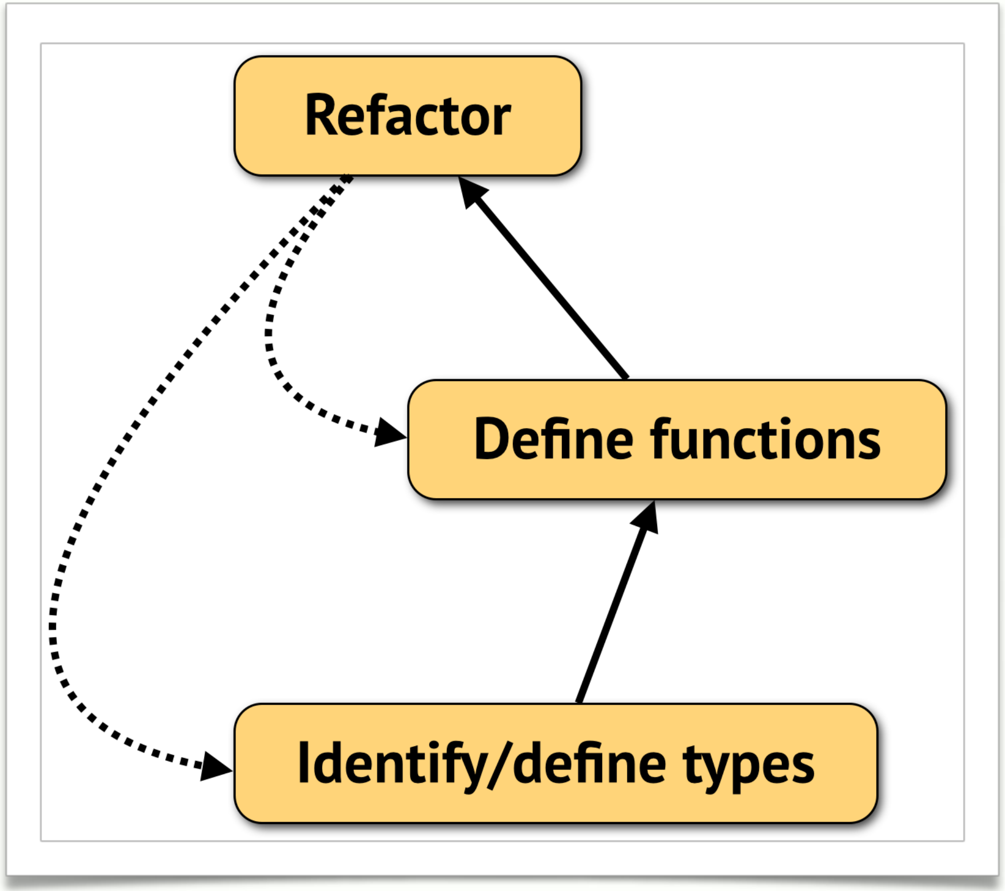
Haskell Basics

What is functional programming?

- What is a function?
- Equational reasoning
- First-order vs. higher-order functions
- Lazy evaluation

How to functional program

Functional programming workflow



Warning: may lead to *"obsessive compulsive refactoring disorder"*

Anatomy of a data type

Shape.hs

type name

```
data Expr = Lit Int
          | Plus Expr Expr
```

cases

constructor

types of arguments

Example: `2 + 3 + 4`

```
Plus (Lit 2) (Plus (Lit 3) (Lit 4))
```

Type parameters

Specialized lists

```
type IntList         = List Int
type CharList        = List Char
type RaggedMatrix a = List (List a)
```

type parameter

```
data List a = Nil
            | Cons a (List a)
```

reference to type parameter

recursive reference to type

Tools for defining functions

Recursion and other functions

```
sum :: [Int] -> Int
sum xs = if null xs then 0
         else head xs + sum (tail xs)
```

Pattern matching

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

1. case analysis

2. decomposition

Higher-order functions

```
sum :: [Int] -> Int
sum = foldr (+) 0
```

no recursion **or** variables needed!