I Haskell
Doing vs. Being

How are things?

How to change things?
Change vs. Description

### Quicksort in Java

```java
private void quicksort(int low, int high) {
    int i = low, j = high;
    int pivot = numbers[low + (high-low)/2];
    while (i <= j) {
        while (numbers[i] < pivot) {
            i++;
        }
        while (numbers[j] > pivot) {
            j--;
        }
        if (i <= j) {
            exchange(i, j);
            i++;
            j--;
        }
    }
    if (low < j) quicksort(low, j);
    if (i < high) quicksort(i, high);
}
private void exchange(int i, int j) {
    int temp = numbers[i];
    numbers[i] = numbers[j];
    numbers[j] = temp;
}
```

### Quicksort in Haskell

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y<-xs, y<=x] ++ [x] ++ qsort [y | y<-xs, y>x]
```

Robert Recorde invented the “=” sign in 1557.

Same symbol – completely different meaning!
The Meaning of “=’’

S → S - {(i, S(i))} U {(i, S(i)+1)}

i = i + 1

In Haskell:
No state, No assignment!

(There are monads ...)

i_{new} = i_{old} + 1

Java, C, ...
(... and maybe on Fox News)

Haskell

Math, Logic, Philosophy, ...
... and the rest of the rational world

undefined
So how do I do anything in Haskell?

You don’t!

Instead you describe!
How do I rearrange the elements in a list to obtain the reverse list?

**Example: reversing a list**

S → T

Function that maps values of type S to values of type T

**Description of Computation:**
Equations relating input to output

How are a list and its reverse related?

**Operational view**

How do I rearrange the elements in a list to obtain the reverse list?

**Declarative view**

How are a list and its reverse related?

reverse :: [a] → [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
Substituting Equals for Equals

Pattern Matching:
1) Conditional
2) Bindings

reverse :: [a] → [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

Pattern Matching:
(1) Conditional
(2) Bindings

reverse [1,2,3,4] =
reverse [2,3,4] ++ [1] =
reverse [3,4] ++ [2] ++ [1] =
reverse [4] ++ [3] ++ [2] ++ [1] =
reverse [] ++ [4] ++ [3] ++ [2] ++ [1] =
[4,3,2,1]
The Essence of Functional Programming

Core Features

- computation $\equiv$ function
- higher-order functions
- (no side effects)

Evaluation
- strict
- non-strict

Typing
- static
- dynamic

Haskell
4 Steps to Learning How to Program

1. How to evaluate programs?
2. How to run programs?
3. How to write programs?
4. How to define input?
4 Steps to Learning Haskell

1. How to evaluate programs?
   - Evaluate expressions

2. How to run programs?
   - Apply functions

3. How to write programs?
   - Define functions

4. How to define input?
   - Define types and values

Diagram:
- Program
  - Define functions
  - Apply functions
- Input
- Language Implementation
- Output
Defining Functions

Recursion

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

Higher-Order Functions

sum :: [Int] → Int
sum = foldr (+) 0

variables & recursion not needed!

head :: [a] → a
head (x:_):= x

tail :: [a] → [a]
tail (_:xs) = xs

(2) Data decomposition
Exercises

1. Define the function \( \text{length} :: [a] \to \text{Int} \)

\[
\text{length} \ [] = 0 \\
\text{length} \ (\_ : \text{xs}) = 1 + \text{length} \ \text{xs}
\]

2. Evaluate the expressions that don’t contain an error

- \( \text{sum} \ \text{xs} + \text{length} \ \text{xs} \)
- \( \text{xs} ++ \text{length} \ \text{xs} \)
- \( \text{xs} ++ \text{[length} \ \text{xs}] \)
- \( \text{[sum} \ \text{xs, length} \ \text{xs}] \)
- \( \text{[xs, length} \ \text{xs}] \)

\[
\begin{align*}
\text{sum} \ :: \ [\text{Int}] \to \text{Int} \\
\text{sum} \ [] &= 0 \\
\text{sum} \ (\text{x:xs}) &= \text{x} + \text{sum} \ \text{xs}
\end{align*}
\]

\[
\text{sum} = \text{foldr} \ (+) \ 0
\]

\[
\begin{align*}
\text{xs} &= [1,2,3] \\
5:xs \\
\text{xs:5} \\
[\text{tail} \ \text{xs,5}] \\
[\text{tail} \ \text{xs,\text{[5]}}] \\
\text{tail} \ \text{[xs,xs]}
\end{align*}
\]
Piazza Question

Haskell list functions
Higher-Order Functions

**map** :: \((a \rightarrow b) \rightarrow [a] \rightarrow [b]\)
\[
\text{map } f \text{ [] } = \text{ [] }
\]
\[
\text{map } f \text{ (x:xs) } = f \text{ x:map } f \text{ xs}
\]

**fold** :: \((a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b\)
\[
\text{fold } f \text{ u } \text{ [] } = u
\]
\[
\text{fold } f \text{ u } (x:xs) = x \text{ `f` } (\text{fold } f \text{ u } \text{ xs})
\]

**HOFs ≡ Control Structures**

Loop for processing elements independently

Loop for aggregating elements

\[
\text{sum } = \text{ fold } (+) \text{ 0}
\]
\[
\text{fac } n = \text{ fold } (*) \text{ 1 [2 .. n]}
\]
Higher-Order Functions

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

(·) :: (b → c) → (a → b) → a → c
f · g = \x → f (g x)

plus2 = succ . succ
odd = not . even
snd = head . tail
drop2 = tail . tail

succ :: Int → Int
even :: Int → Bool
not :: Bool → Bool
head :: [a] → a
tail :: [a] → [a]
Exercises

3. Is the function \( \text{th} \) well defined? If so, what does it do and what is its type?

\[
\text{th} :: ? \\
\text{th} = \text{tail} \ . \ \text{head}
\]

\[
(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c
\]

4. What does the expression \( \text{map} \ f \ . \ \text{map} \ g \) compute? How can it be rewritten?

\[
\text{map} f \ . \ \text{map} g = \text{map} (f \ . \ g)
\]

head :: [a] \rightarrow a
head (x:_ ) = x

tail :: [a] \rightarrow [a]
tail (_:xs) = xs
Exercises

5. Implement `revmap` using pattern matching

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]
\[
\text{map} \ f \ [] = []
\]
\[
\text{map} \ f \ (x:xs) = f \ x : \text{map} \ f \ xs
\]

\[
\text{revmap} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]
\[
\text{revmap} \ f \ [] = []
\]
\[
\text{revmap} \ f \ (x:xs) = \text{revmap} \ f \ xs \mathbin{\&}\mathbin; [f \ x]
\]

6. Implement `revmap` using function composition

\[
(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c
\]

\[
\text{reverse} :: [a] \rightarrow [a]
\]
\[
\text{reverse} \ [] = []
\]
\[
\text{reverse} \ (x:xs) = \text{reverse} \ xs \mathbin{\&}\mathbin{\&}\mathbin{\&} [x]
\]
Exercises

7. Find expressions to ...

... increment elements in xs by 1
... increment elements in ys by 1
... find the last element in xs

xs = [1,2,3]
yx = [xs,[7]]

8. Define the function

last :: [a] → a

9. Evaluate all the expressions that don’t contain an error

map sum xs
map sum ys
last ys
map last ys
last (last ys)
Data Constructors

\approx \text{Java object constructor, but:}
(1) Inspection by pattern matching!
(2) immutable!

“WORM”: Write Once, Read Many times

data Nat = Zero
| Succ Nat

data List = Empty
| Cons Int List

two = Succ (Succ Zero)

xs = Cons 31 (Cons 7 Empty)
More on Data Constructors

\[
data \text{Nat} = \text{Zero} \mid \text{Succ Nat}
\]

\[
data \text{List} = \text{Empty} \mid \text{Cons Int List}
\]

\[
data \text{List} = \text{Empty} \mid \text{Cons Nat List}
\]

\[
two = \text{Succ} (\text{Succ Zero})
\]

\[
xs = \text{Cons} 1 (\text{Cons} 2 \text{Empty})
\]

\[
ys = \text{Cons} \text{one} (\text{Cons} \text{two} \text{Empty})
\]
Avoiding Sharing

```haskell
data List = Empty
           | Cons Nat List

one = Succ Zero
two = Succ one
ys = Cons one (Cons two Empty)

one = Succ Zero
two = Succ (Succ Zero)
ys = Cons one (Cons two Empty)
```
Cyclic Data Structures

data List = Empty |
             Cons Int List

xs = Cons 1 (Cons 2 Empty)

Intensional description of an infinite list

ones = Cons 1 ones
morse = Cons 1 (Cons 0 morse)

zeros = Cons 0 zeros
big = Cons 1 zeros
Changing Data Structures

```
data List = Empty | Cons Int List
```

```
zs = Cons 1 (Cons 2 (Cons 3 Empty))
```

```
chgLast :: Int → List → List
chgLast y [x] = [y]
chgLast y (x:xs) = x:chgLast y xs
```

“WORM”: Write Once, Read Many times
Summary: Haskell so far

• Functions (vs. state manipulation)
• No side effects
• Higher-Order Functions (i.e. flexible control structures)
• Recursion
• Data Types (constructors and pattern matching)
• More Haskell features:
  list comprehensions, pattern guards, where blocks
Currying

The values of a type are the proofs for the proposition represented by it.

Proof for Proposition

Program : Type

even : Int → Bool

sort : List → SortedList

Curry-Howard Isomorphism

The values of a type are the proofs for the proposition represented by it.

(a, b) → c

A ∧ B ⇒ C

(A ∧ B) ⇒ C

¬(A ∧ B) ∨ C

¬A ∨ ¬B ∨ C

A ⇒ (¬B ∨ C)

A ⇒ (B ⇒ C)

a → b → c

≡

a → (b → c)
“Curried” Dinners are More Spicy

Experience dinner(Drink d, Entree e, Dessert f){…}

dinner(wine, pasta, pie)

Haskell

dinner :: Drink → Entree → Dessert → Experience

dinner wine :: Entree → Dessert → Experience

Must provide all arguments at once

Partial function application is possible