I Haskell
Doing vs. Being

THOSE WARM-BLOODED THINGS WILL NEVER LAST — THEY'RE TOO HYPERACTIVE.

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;
}
```

Robert Recorde invented the “=” sign in 1557

Same symbol – completely different meaning!

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]
```

Haskell
The Meaning of “=”

\[ S \rightarrow S - \{(i, S(i))\} \cup \{(i, S(i)+1)\} \]

i = i + 1

\[ i_{\text{new}} = i_{\text{old}} + 1 \]

In Haskell:
No state, No assignment!

(There are monads ...)

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

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

Description of Computation:
Equations relating input to output

Example: reversing a list

How are a list and its reverse related?

Haskell

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

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

Pattern Matching:
(1) Conditional
(2) Bindings

(+) :: a → [a] → [a]
(++) :: [a] → [a] → [a]
The Essence of Functional Programming

Core Features
- computation \equiv \text{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

Program -> Language Implementation -> Output
**Defining Functions**

### Recursion

Sum function defined recursively:

\[
\text{sum} :: [\text{Int}] \to \text{Int} \\
\text{sum} \ x = \begin{cases} 
0 & \text{null} \ x \\
\text{head} \ x + \text{sum} \ (\text{tail} \ x) & \text{else}
\end{cases}
\]

- **Case analysis**
- **Data decomposition**

### Pattern Matching

Sum function defined with pattern matching:

\[
\text{sum} :: [\text{Int}] \to \text{Int} \\
\text{sum} \ [] = 0 \\
\text{sum} \ (x:xs) = x + \text{sum} \ xs
\]

- **Case analysis**
- **Data decomposition**

### Higher-Order Functions

Sum function defined with higher-order function:

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

- **Variables & recursion not needed!**

Additional functions:

- **head**:
  \[
  \text{head} :: [\text{a}] \to \text{a} \\
  \text{head} (x:_\) = x
  \]

- **tail**:
  \[
  \text{tail} :: [\text{a}] \to [\text{a}] \\
  \text{tail} (_\:xs) = xs
  \]
1. Define the function \( \text{length} :: [a] \rightarrow \text{Int} \)

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

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

\[
\text{sum} :: [\text{Int}] \rightarrow \text{Int} \\
\text{sum} [] = 0 \\
\text{sum} (x : xs) = x + \text{sum} xs \\
\text{sum} = \text{foldr} (+) 0 \\
\text{xs} = [1,2,3] \\
\]

\[
\text{sum xs} + \text{length xs} \\
\text{xs ++ length xs} \\
\text{xs ++ [length xs]} \\
[\text{sum xs}, \text{length xs}] \\
[xs, \text{length xs}] \\
\text{5:xs} \\
\text{xs:5} \\
[\text{tail} \text{xs,5}] \\
[\text{tail} \text{xs,}[5]] \\
\text{tail} [\text{xs},\text{xs}] 
\]
Higher-Order Functions

HOFs ≡ Control Structures

Loop for processing elements independently

\[
\text{map } f \ [x_1, \ldots, x_k] = [f \ x_1, \ldots, f \ x_k]
\]

\[
\begin{align*}
\text{map} & \:: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map} & \ [\ ] \quad = \ [\ ] \\
\text{map} & \ (x:xs) \quad = \ f \ x \cdot \text{map} \ f \ xs
\end{align*}
\]

Loop for aggregating elements

\[
\text{fold } f \ u \ [x_1, \ldots, x_k] = x_1 \cdot f \cdot \ldots \cdot f \cdot x_k \cdot f \cdot u
\]

\[
\begin{align*}
\text{fold} & \:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
\text{fold} & \ u \ [\ ] \quad = \ u \\
\text{fold} & \ u \ (x:xs) \quad = \ x \cdot f \cdot (\text{fold} \ f \ u \ xs)
\end{align*}
\]

\[
\begin{align*}
\text{sum} & \quad = \ \text{fold} \ (+) \ 0 \\
\text{fac} \ n & \quad = \ \text{fold} \ (*) \ 1 \ [2 \ \ldots \ \ n]
\end{align*}
\]
Higher-Order Functions

Function composition

\[(f \cdot g) x = f (g x)\]

\[\cdot :: (b \to c) \to (a \to b) \to a \to c\]
\[f \cdot g = \lambda x \to f (g x)\]

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

succ :: Int \to Int
even :: Int \to Bool
not :: Bool \to Bool
head :: [a] \to a
tail :: [a] \to [a]
3. Is the function `th` well defined? If so, what does it do and what is its type?

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

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

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

\[ \text{tail} :: [a] \rightarrow [a] \]
\[ \text{tail} (_:xs) = xs \]

4. What does the expression `map f \cdot map g` compute? How can it be rewritten?

\[ \text{map f} \cdot \text{map g} = \text{map (f \cdot g)} \]
5. Implement \texttt{revmap} using pattern matching

\begin{verbatim}
map :: (a → b) → [a] → [b]
map f []   = []
map f (x:xs) = f x:map f xs
\end{verbatim}

\begin{verbatim}
reverse :: [a] → [a]
reverse []   = []
reverse (x:xs) = reverse xs ++ [x]
\end{verbatim}

6. Implement \texttt{revmap} using function composition

\begin{verbatim}
(·) :: (b → c) → (a → b) → a → c
\end{verbatim}
Exercises

7. Find expressions to …
   ... increment elements in xs by 1
   ... increment elements in ys by 1
   ... find the last element in xs

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)
Haskell

Data Constructors

≈ 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

tax = Succ (Succ Zero)

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

```haskell
data Nat = Zero | Succ Nat

Succ
Succ Zero

Succ
Succ
Succ

Succ
Succ
Succ

Succ
Succ
Succ

data List = Empty | Cons Int List

Cons 1
Cons 2

Cons 1
Cons 2

Cons 1
Cons 2

one = Succ Zero

Cons
Cons
Cons

Cons
Cons
Cons

Cons
Cons
Cons

two = Succ (Succ Zero)

xs = Cons 1 (Cons 2 Empty)

ys = Cons one (Cons two Empty)
```
Avoiding Sharing

\[
\begin{align*}
\text{data } \text{List} & = \text{Empty} \\
& | \text{ Cons Nat List} \\
\text{one} & = \text{Succ Zero} \\
\text{two} & = \text{Succ one} \\
\text{ys} & = \text{Cons one (Cons two Empty)} \\
\end{align*}
\]
Cyclic Data Structures

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

xs = Cons 1 (Cons 2 Empty)

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

zeros = Cons 0 zeros
big = Cons 1 zeros
```

Intensional description of an infinite list.

Haskell
Changing Data Structures

```haskell
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

Curry-Howard Isomorphism

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

(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){…}

Experience dinner(wine, pasta, pie)

Must provide all arguments at once

Haskell

dinner :: Drink → Entree → Dessert → Experience

dinner wine :: Entree → Dessert → Experience

Partial function application is possible