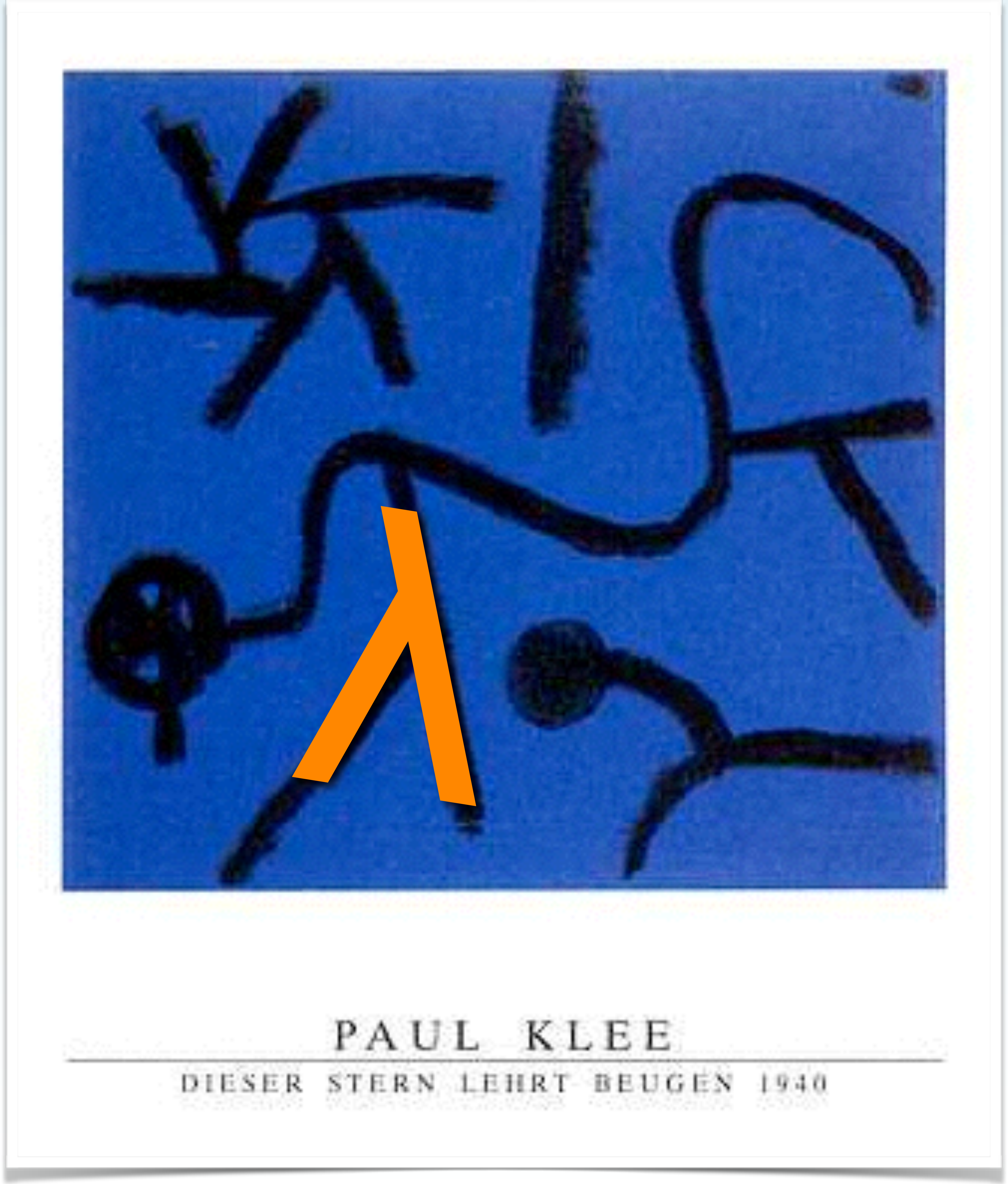
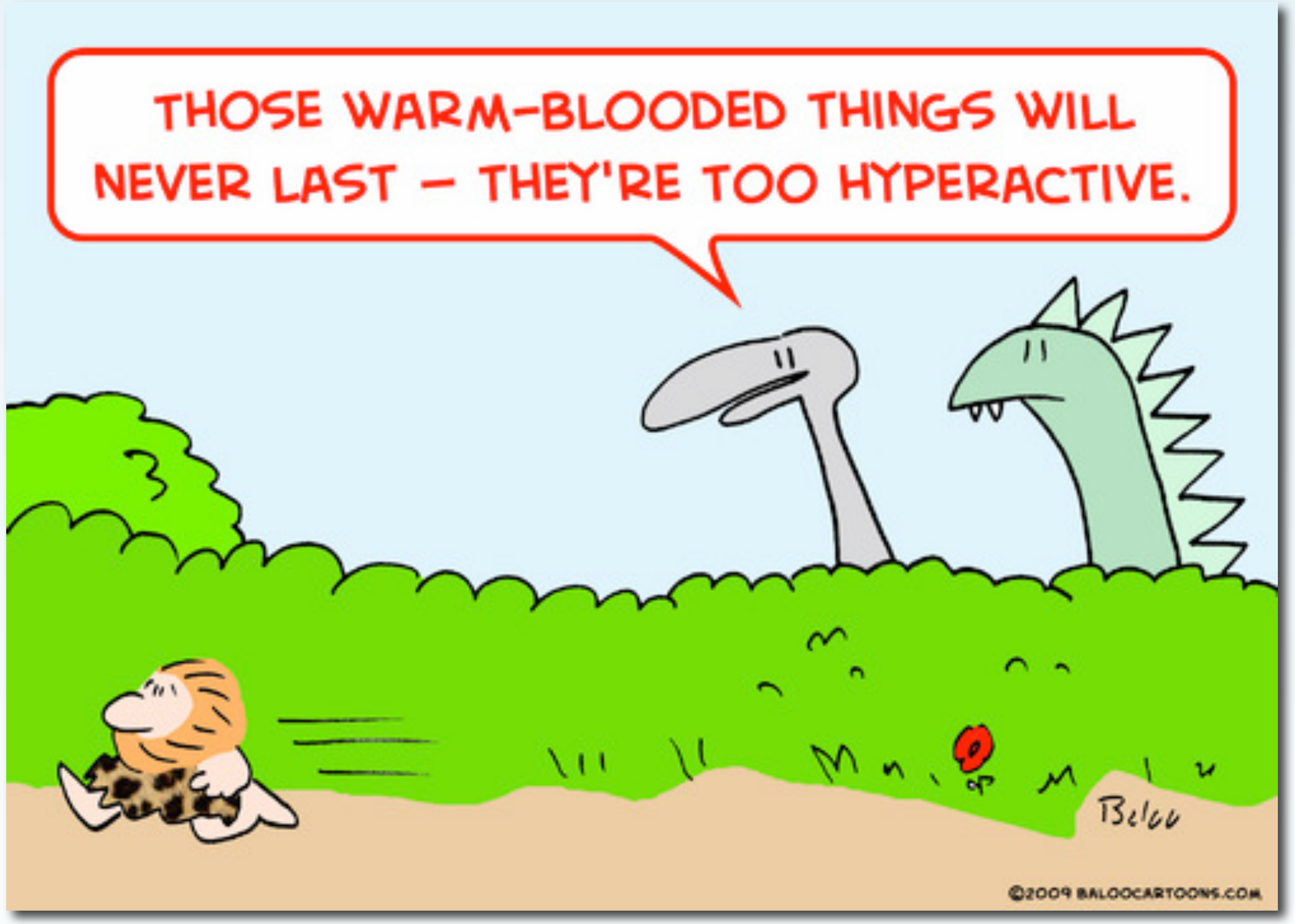
1 Haskell



Doing vs. Being



How to change things?

How are things?

Change vs. Description

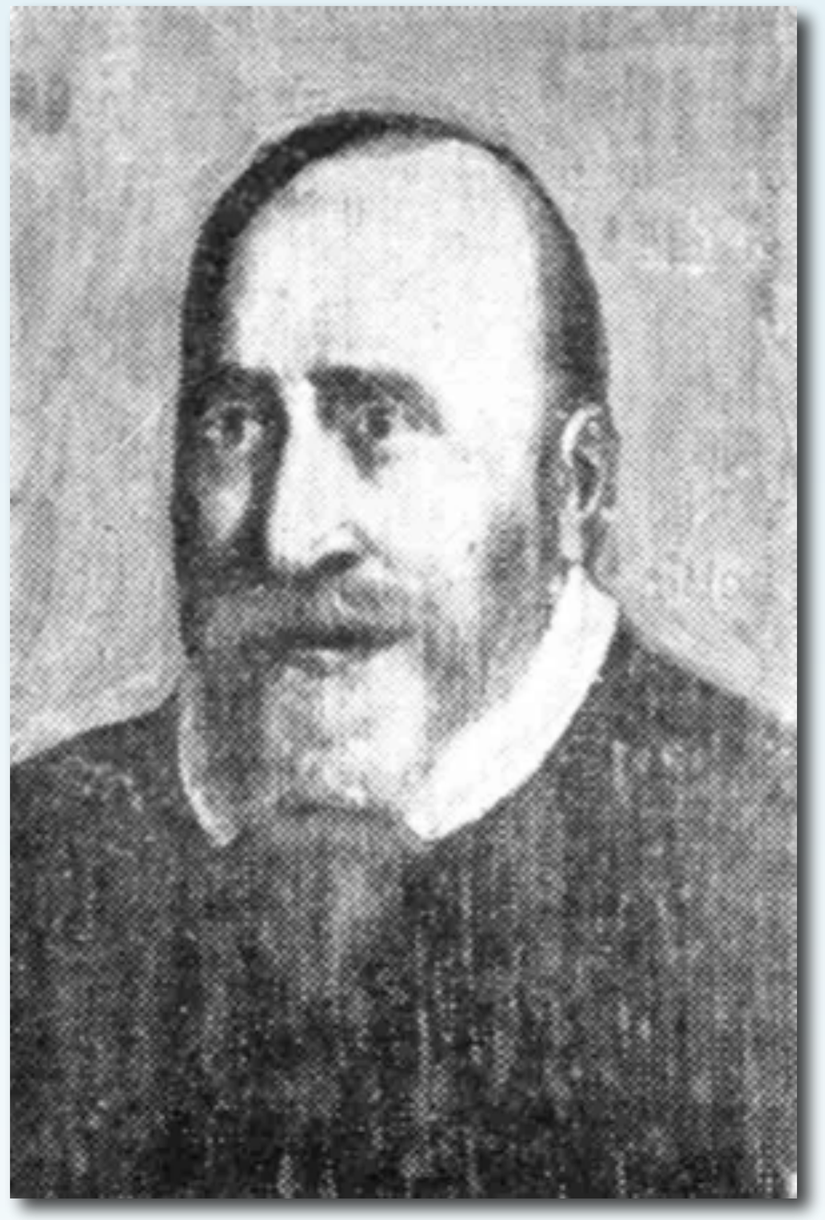

```java
private void quicksort(int low, int high) {
     int i = low, j = high;
     int pivot = numbers[low + (high-low)/2];

     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 Java

invented the "=" sign in 1557



Robert Recorde

Same symbol – completely different meaning!

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

Quicksort in Haskell

The Meaning of “=”

$i = i + 1$

Java, C, ...

(... and maybe on Fox News)

$S \mapsto S - \{(i, S(i))\} \cup \{(i, S(i)+1)\}$

Haskell

Math, Logic, Philosophy, ...

... and the rest of the rational world

undefined

$i_{new} = i_{old} + 1$

In Haskell:
No state, No assignment!

(There are monads ...)

So how do I *do* anything in Haskell ?

You don't !

Instead you *describe* !

Computation in Haskell

$S \rightarrow T$

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

Description of Computation: Equations relating input to output

Example: reversing a list

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

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

Pattern Matching:
(1) Conditional
(2) Bindings

```
(:)  ::  a  → [a] → [a]
(++) :: [a] → [a] → [a]
```

```
  x    xs
(1:[2,3,4])
             x    xs
           (2:[3,4])
```

```
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] =
[4,3,2,1]
```

The Essence of Functional Programming

Core Features

computation ≡ function
higher-order functions
(no side effects)

Evaluation

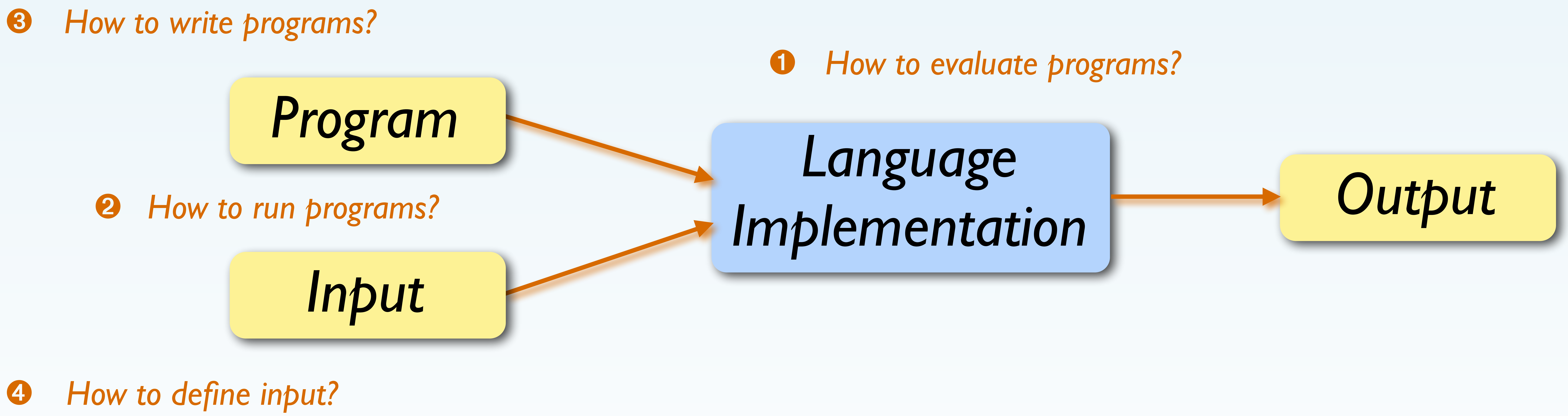
strict
non-strict

Typing

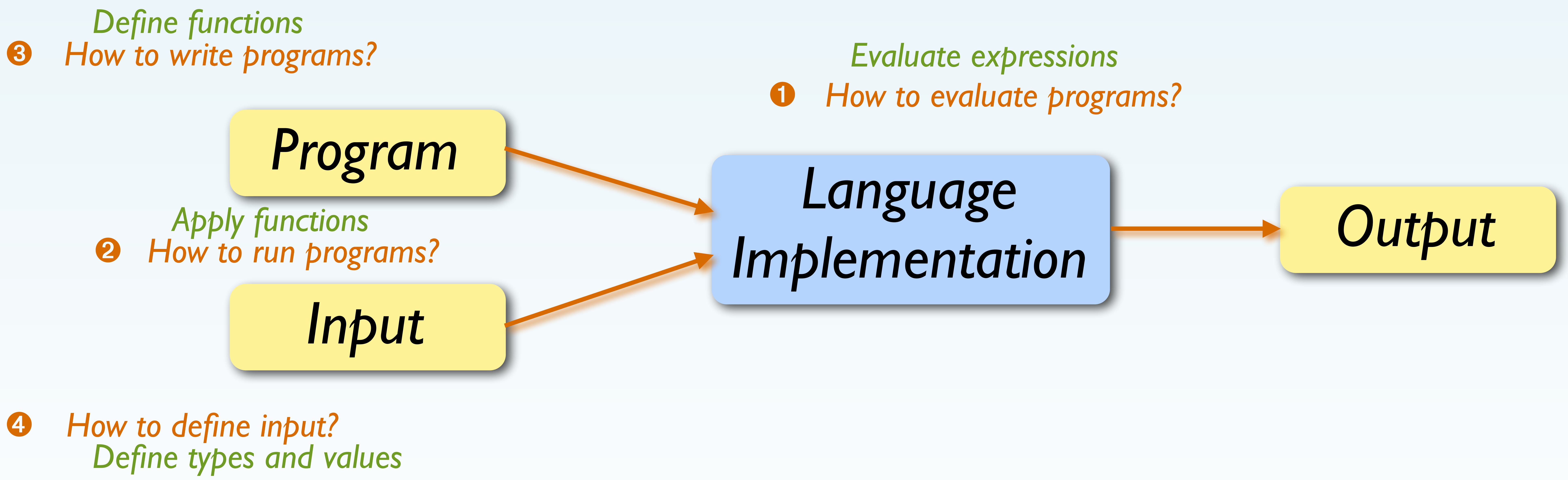
static
dynamic

Haskell

4 Steps to Learning How to Program



4 Steps to Learning Haskell



Defining Functions


```
head :: [a] → a
head (x:_) = x
```

```
tail :: [a] → [a]
tail (_:xs) = xs
```

Recursion

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

(1) *Case analysis*

Pattern Matching

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

(2) *Data decomposition*

Higher-Order Functions

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

variables & recursion not needed!

Exercises


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

1. *Define the function* `length :: [a] → Int`

Pointfree

```
sum = foldr (+) 0
```

2. *Evaluate the expressions that don't contain an error*

```
xs = [1,2,3]
```

```
sum xs + length xs
xs ++ length xs
xs ++ [length xs]
[sum xs, length xs]
[xs, length xs]
```

```
5:xs
xs:5
[tail xs,5]
[tail xs,[5]]
tail [xs,xs]
```

Higher-Order Functions

HOFs ≡ Control Structures

```
map f [x1, ..., xk] = [f x1, ..., f xk]
```

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

Loop for processing elements independently

```
fold f u [x1, ..., xk] = x1 `f` ... `f` xk `f` u
```

Loop for aggregating elements

```
fold :: (a → b → b) → b → [a] → b
fold f u []     = u
fold f u (x:xs) = x `f` (fold f u xs)
```

```
sum = fold (+) 0
fac n = fold (*) 1 [2 .. n]
```

Higher-Order Functions


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

Function composition

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

Pointfree

```
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]
```

Pause for Point Free Demo

Exercises

3. *Is the function* `th` *well defined?*
If so, what does it do and what is its type?

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

```
(.) :: (b → c) → (a → b) → a → c
```

```
head :: [a] → a
head (x:_) = x
```

```
tail :: [a] → [a]
tail (_:xs) = xs
```

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

Exercises

5. *Implement* `revmap` *using pattern matching*

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

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

6. *Implement* `revmap` *using function composition*

```
(.) :: (b → c) → (a → b) → a → c
```

Exercises

```
xs = [1,2,3]
ys = [xs,[7]]
```

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

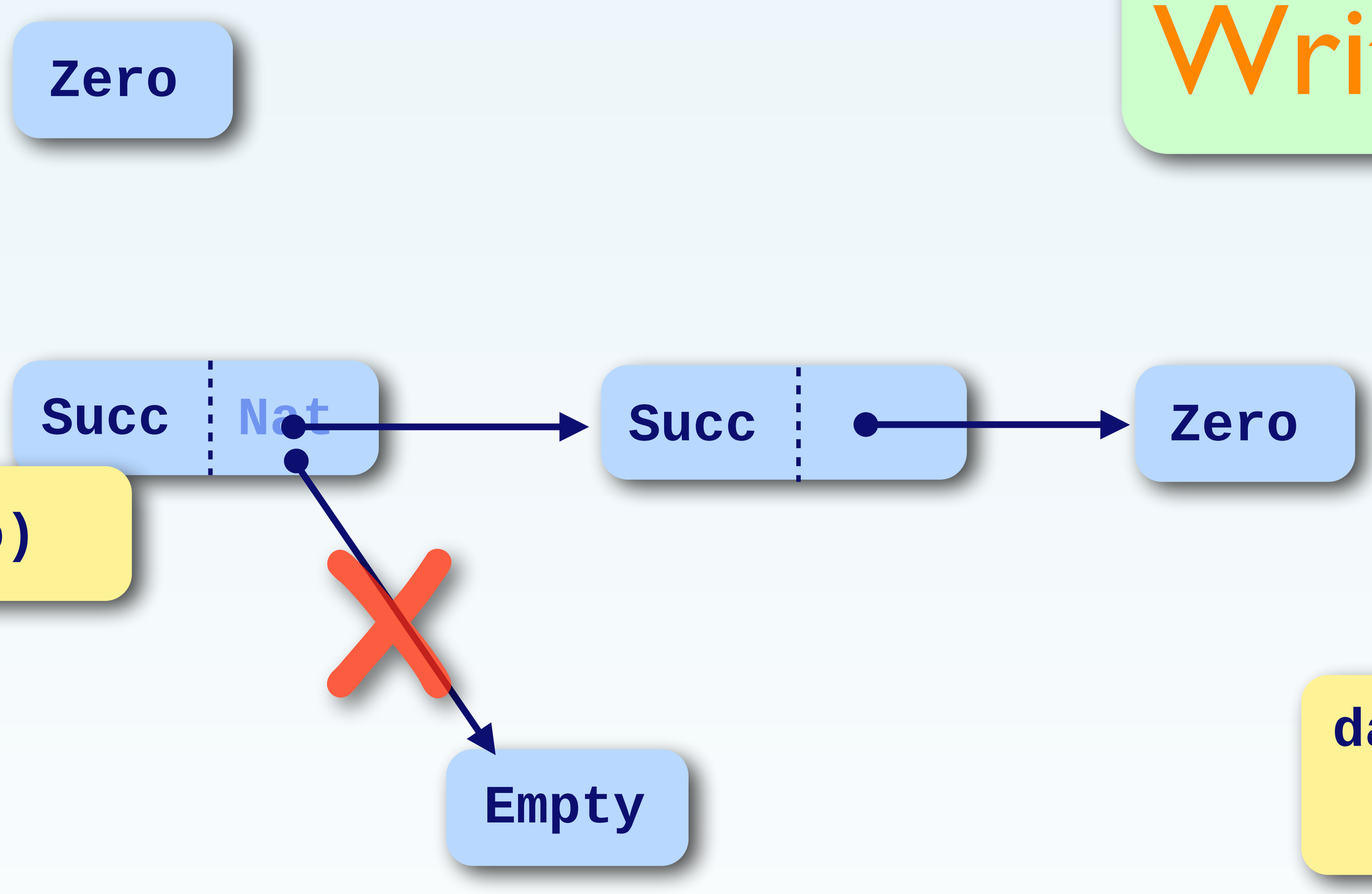
```
map sum xs
map sum ys
last ys
map last ys
last (last ys)
```

Data Constructors

≈ Java object constructor, but:
(1) Inspection by pattern matching!
(2) immutable!

"WORM":
Write Once, Read Many times

```
data Nat = Zero
         | Succ Nat
```




```
two = Succ (Succ Zero)
```

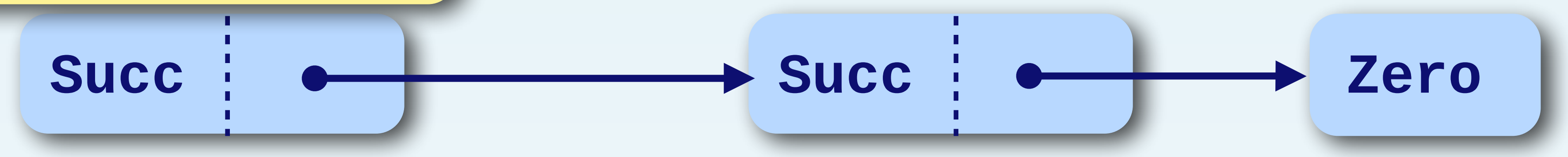
```
data List = Empty
          | Cons Int List
```


```
xs = Cons 31 (Cons 7 Empty)
```

More on Data Constructors

```
data Nat = Zero
         | Succ Nat
```

```
two = Succ (Succ Zero)
```

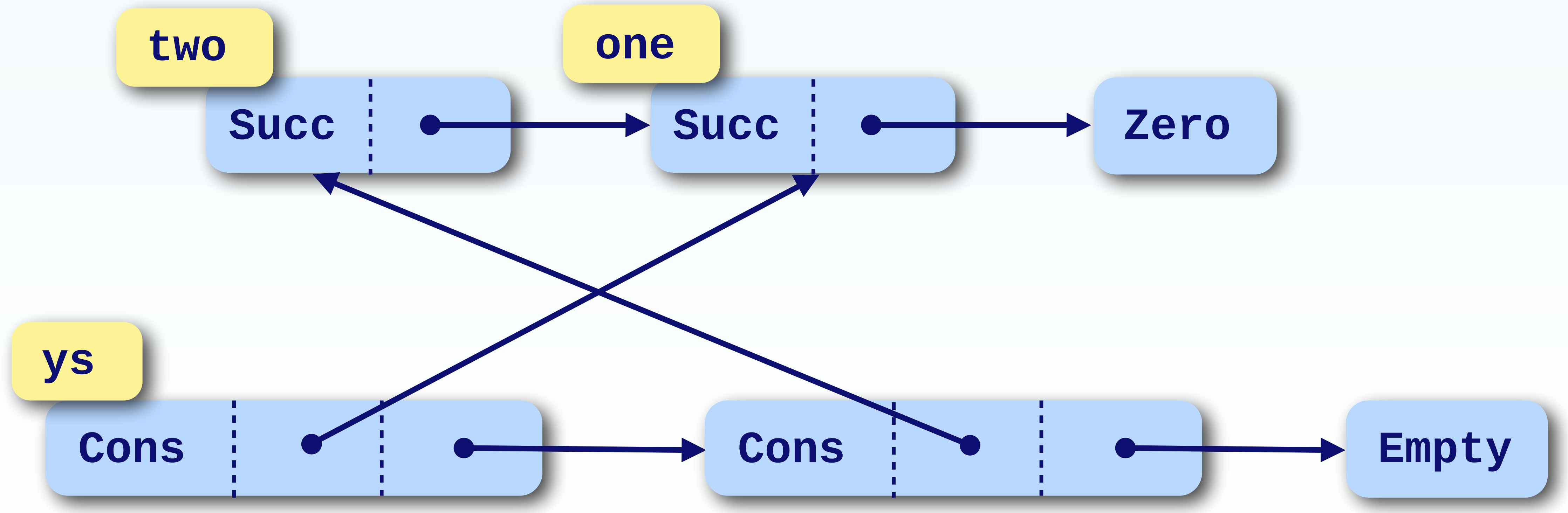



```
data List = Empty
          | Cons Int List
```

```
xs = Cons 1 (Cons 2 Empty)
```


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

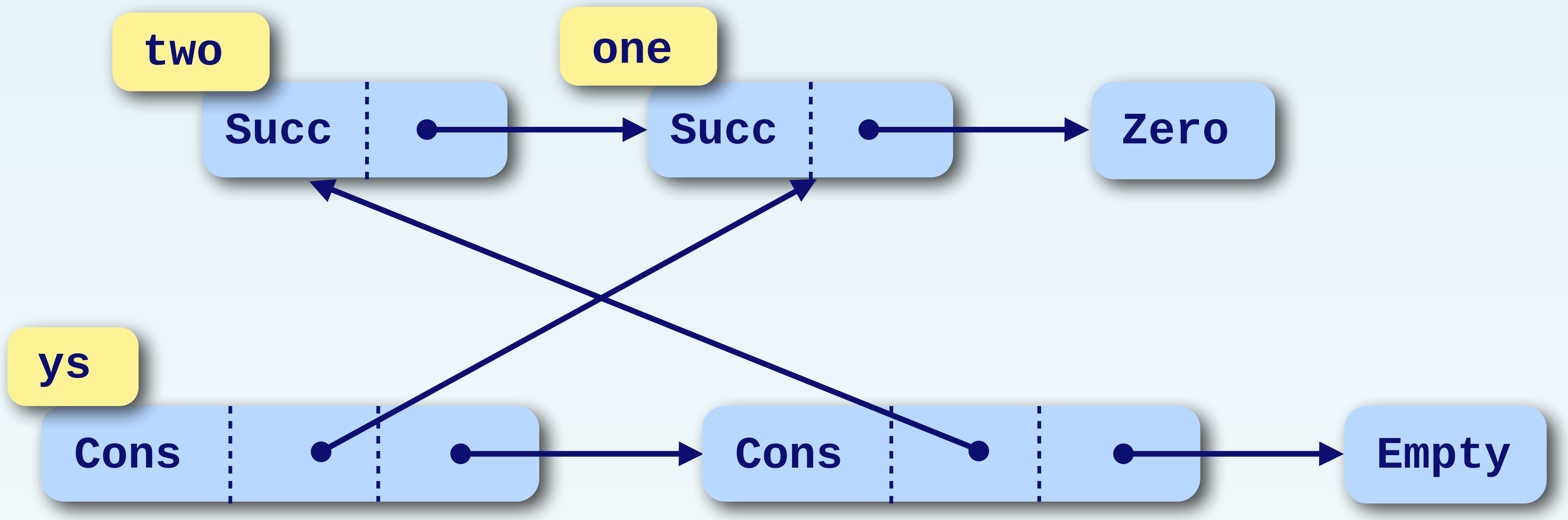
```
one = Succ Zero
two = Succ one
ys = Cons one (Cons two Empty)
```



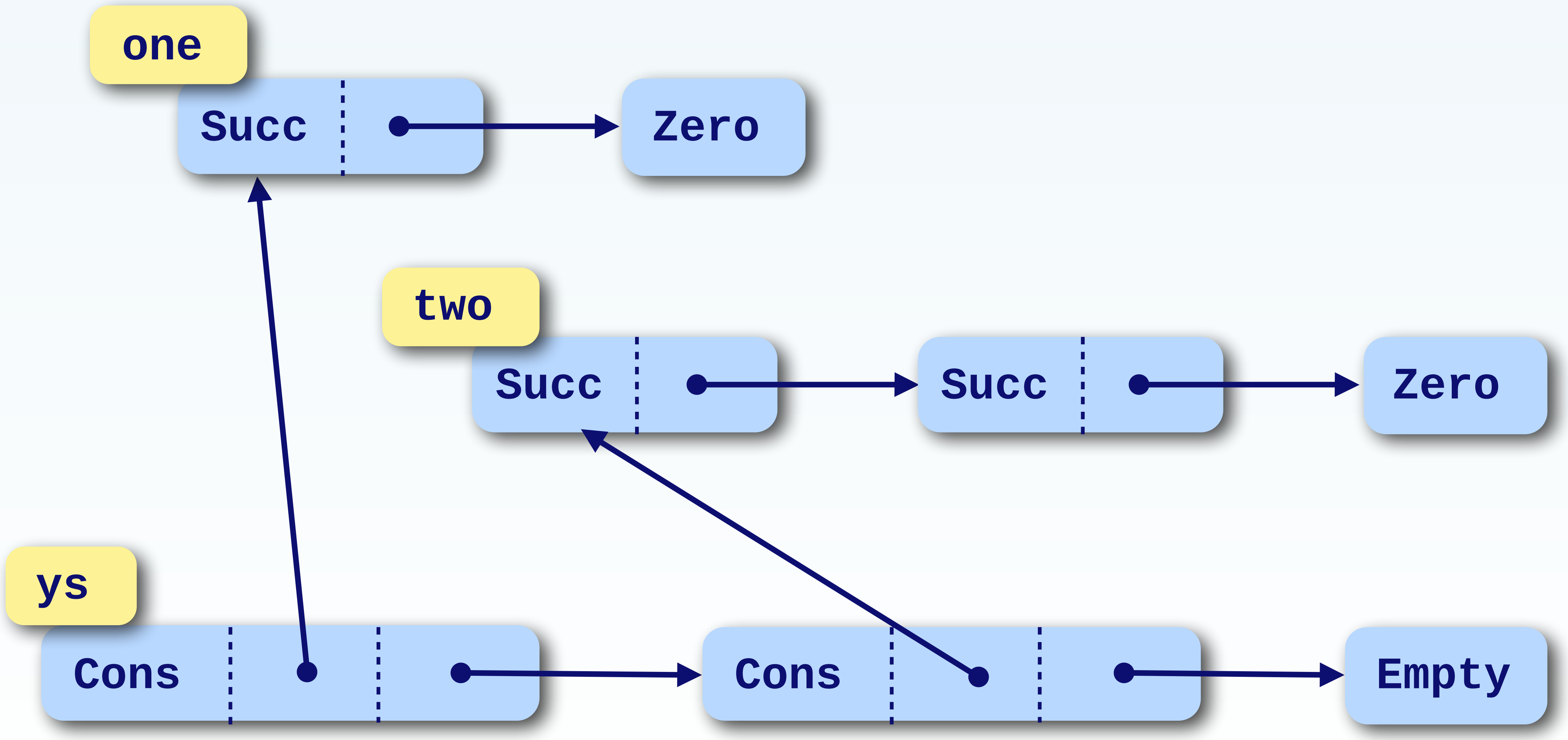
Avoiding Sharing

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

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



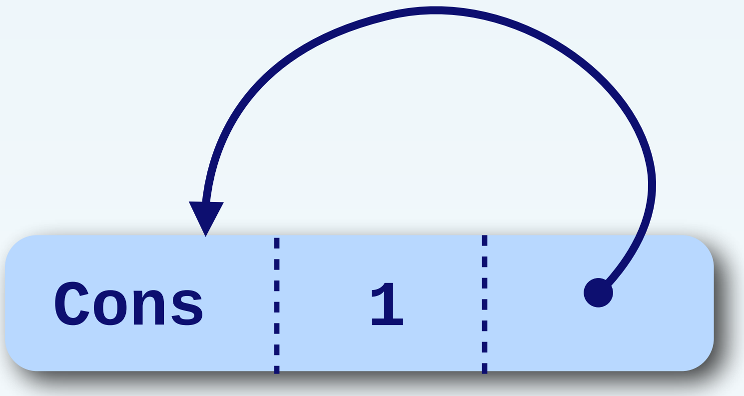

```
one = Succ Zero
two = Succ (Succ Zero)
ys = Cons one (Cons two Empty)
```



Cyclic Data Structures

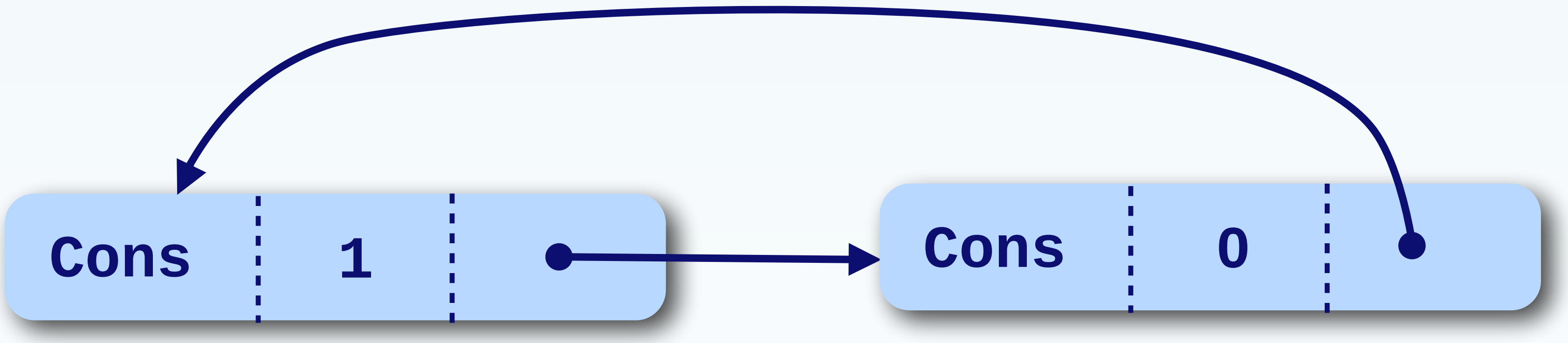
```
data List = Empty
          | Cons Int List
```

```
xs = Cons 1 (Cons 2 Empty)
```

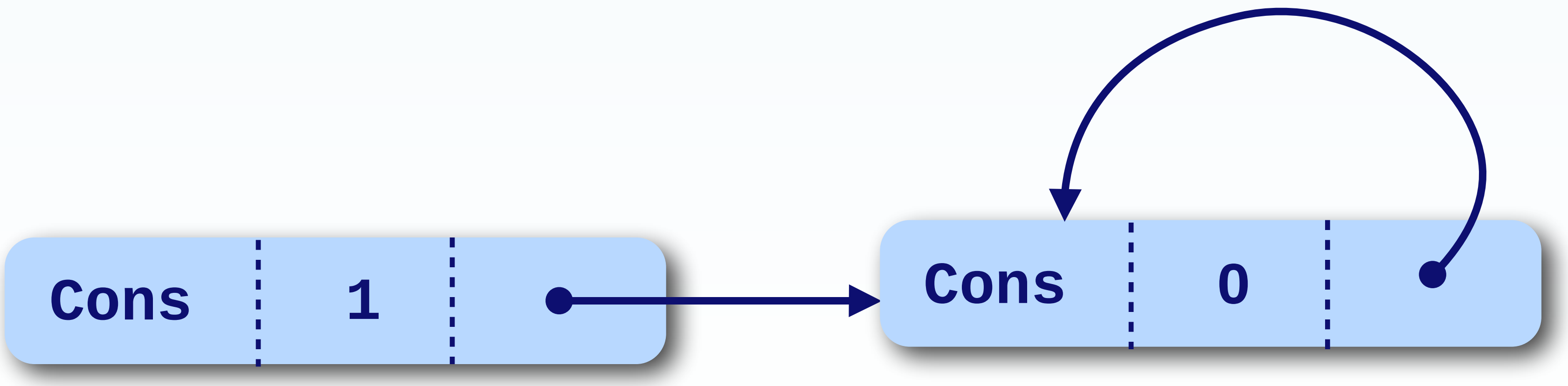


Intensional description of an infinite list

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




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

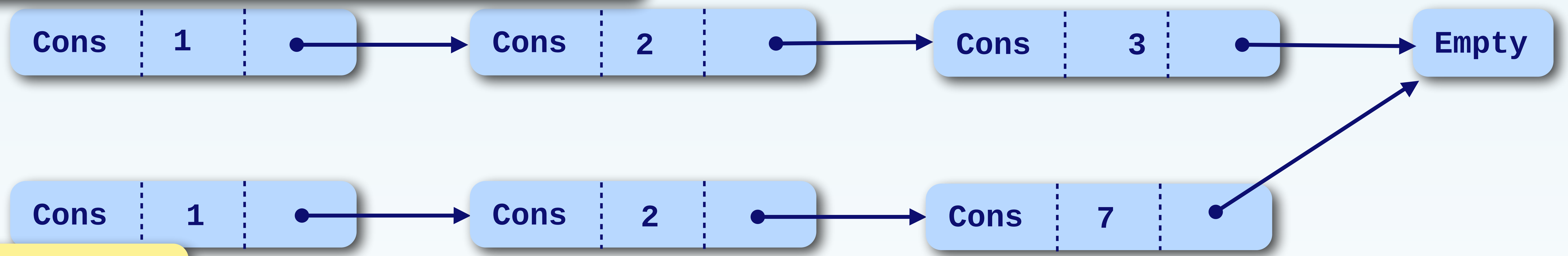


Changing Data Structures

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

“WORM”:
Write Once, Read Many times

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




```
chgLast 7 zs
```

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

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)`

$A \wedge B \Rightarrow C$

$(A \wedge B) \Rightarrow C$

$\neg(A \wedge B) \vee C$

$\neg A \vee \neg B \vee C$

$A \Rightarrow (\neg B \vee C)$

$A \Rightarrow (B \Rightarrow C)$

“Curried” Dinners are More Spicy

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

```
dinner(wine, pasta, pie)
```

Must provide all arguments at once

Java

```
dinner :: Drink → Entree → Dessert → Experience
```

```
dinner wine :: Entree → Dessert → Experience
```

Partial function application is possible

Haskell