1 Haskell

PAUL KLEE

DIESER STERN LEHRT BEUGEN 1940

Change vs. Description

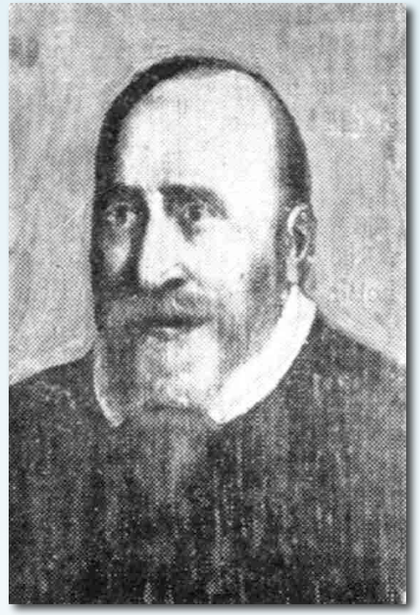
```
private void quicksort(int low, int high) {
     int i = low, j = high;
     int pivot = numbers[low + (high-low)/2];

     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

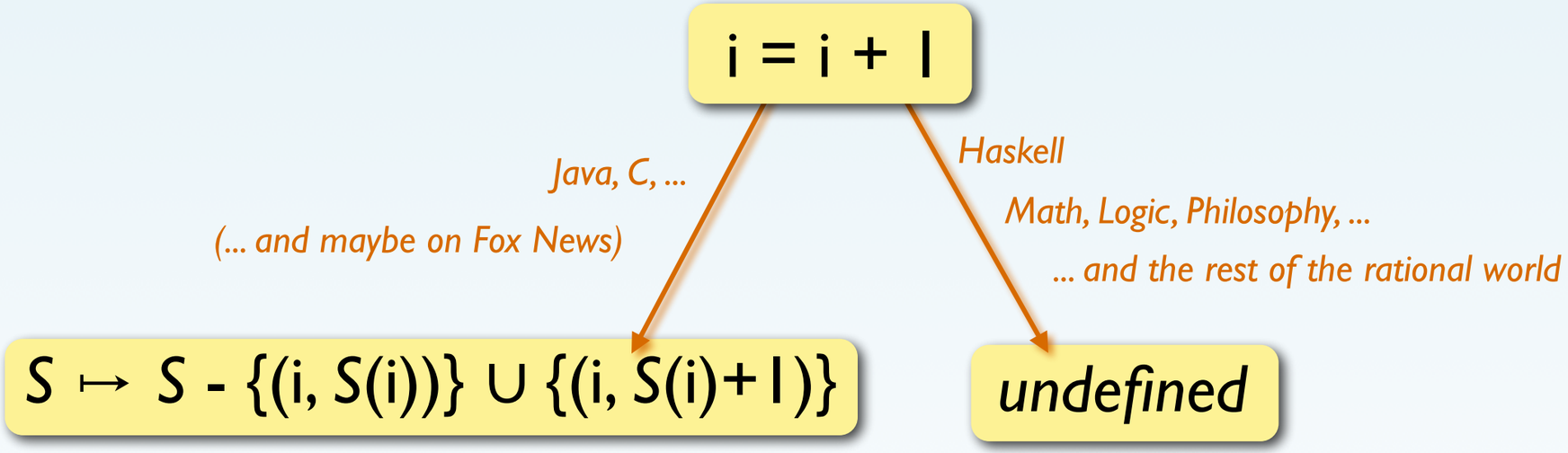
Same symbol – completely different meaning!

```
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 "="



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

In Haskell:
No state, No assignment!

(There are monads ...)

Computation in Haskell

$S \rightarrow T$

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

Description of Computation:
Equations of how to construct output from input

`[Int]` : *list of Int*
`[a]` : *list of anything*

Example: reversing a list

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

Concatenate 2 lists

list of single element `x`

Empty list

Nonempty list with first element and `x` *rest list* `xs`

Zoom Poll

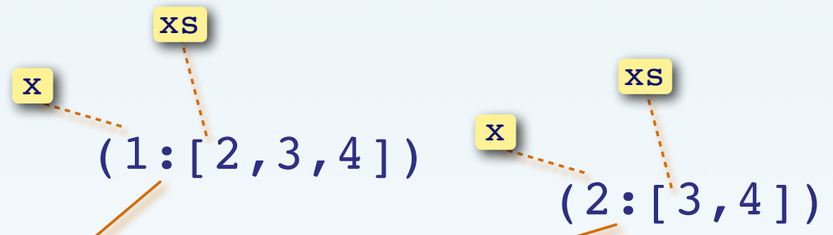
About Recursion

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]
```




```
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]
```

Learning Haskell: First Steps

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Learning Haskell

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions (Type Signatures, Parameters, Equations)*
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types (Constructors, Pattern Matching)*

Exercises

Write an expression that computes the larger value of 5 and 6 (don't use the function max)

*Write an expression that tests whether 2*3 is equal to 3+3*

Learning Haskell

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Exercises

Define a function that tests whether a number is positive using pattern guards

```
positive :: Int → Bool
positive x = if x>=0 then True else False
```

Learning Haskell

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Exercises

Define a function for computing Fibonacci numbers

Define a function that tests whether a number is even

Learning Haskell

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Exercises

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

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

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

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

Define the function `length :: [a] → Int`

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]
```

Zoom Poll

Haskell list functions

From Iteration to Recursion

f

```
result = Start
while Condition[argument] do {
  result = Update[result,argument]
  argument = Simplify[argument]
}
```

```
f argument | Condition[argument] = Update[f(Simplify[argument]),argument]
           | otherwise            = Start
```

fac

```
result = 1
while argument > 1 do {
  result = result * argument
  argument = argument - 1
}
```

```
fac argument | argument > 1 = fac(argument - 1) * argument
             | otherwise    = 1
```

```
fac 1 = 1
fac n = n * fac (n-1)
```

From Iteration to Recursion

f

```
result = Start
while Condition[argument] do {
  result = Update[result,argument]
  argument = Simplify[argument]
}
```

length

```
result = 0
while not (null (list)) do {
  result = result + 1
  list = list.next
}
```

```
f argument | Condition[argument] = Update[f(Simplify[argument]),argument]
           | otherwise           = Start
```

```
length list | not (null list) = length (tail list) + 1
            | otherwise       = 0
```

```
length []     = 0
length (_:xs) = 1 + length xs
```

3 Ways to Define Functions

Conditional

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

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

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

Pattern Matching

(1) *Case analysis*

```
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!

Learning Haskell

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

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
```

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

Loop for aggregating elements

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

Higher-Order Functions

Function composition

```
(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

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
```

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

Exercises

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]
```

Implement `revmap` *using function composition*

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

Exercises

Find expressions to …

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

… *increment elements in xs by 1*

… *increment elements in ys by 1*

… *find the last element in xs*

Define the function

```
last :: [a] → a
```

Evaluate all the expressions that don't contain an error

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

“Curried” Dinners are More Spicy

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

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

Must provide all arguments at once

Curried function definition

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


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

Partial function application is possible

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
```

$A \wedge B \Rightarrow C$

$(A \wedge B) \Rightarrow C$

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

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

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

$A \Rightarrow (B \Rightarrow C)$

```
a → b → c
```

≡

```
a → (b → c)
```

Learning Haskell

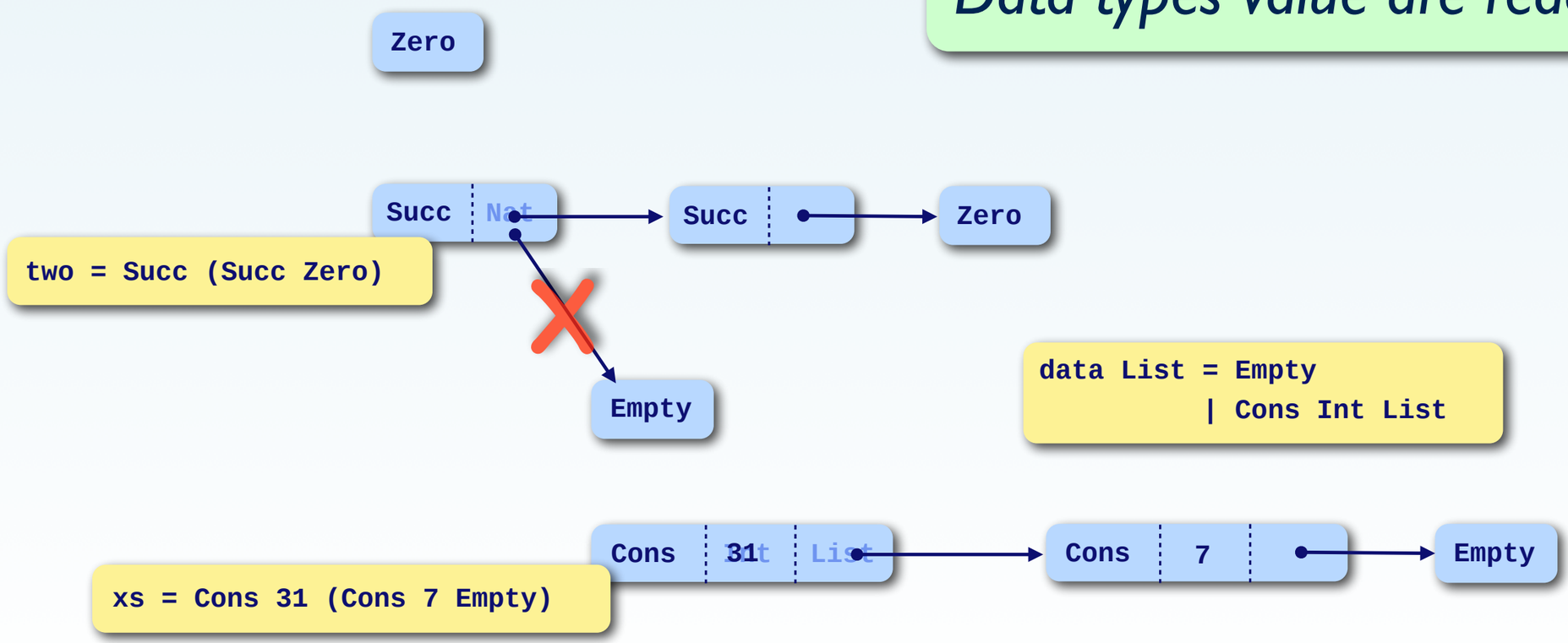
- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Data Constructors

≈ C++/Java object constructor, but:
(1) Inspection by pattern matching!
(2) Immutable!

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

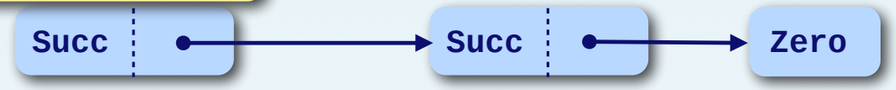
Data types value are read-only



More on Data Constructors

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

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

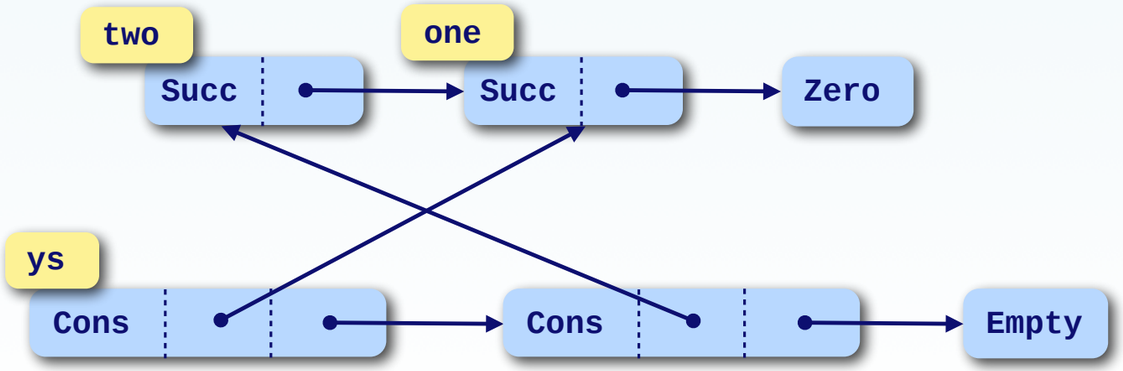



```
data List = Empty
          | Cons Int List
```

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


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

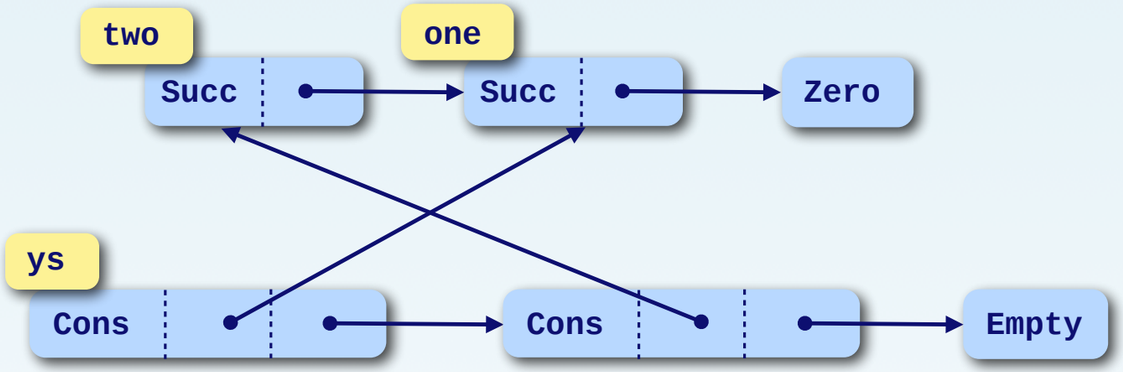
```
one = Succ Zero
two = Succ one
ys = Cons one (Cons two Empty)
```



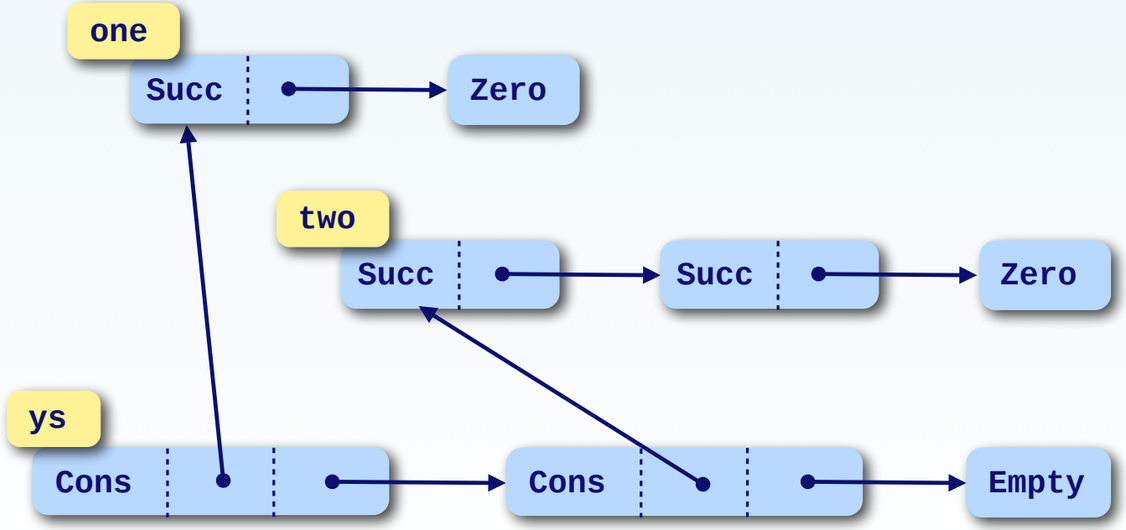
Avoiding Sharing

```
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)
```



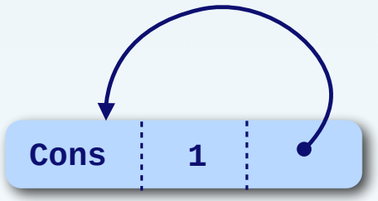
Cyclic Data Structures

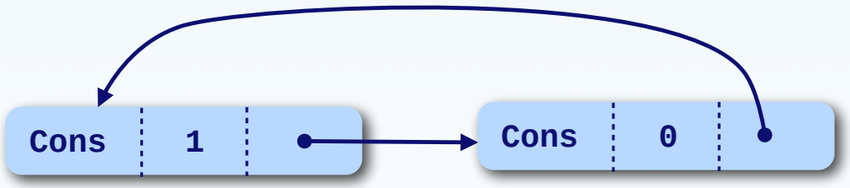
```
data List = Empty
          | Cons Int List
```

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

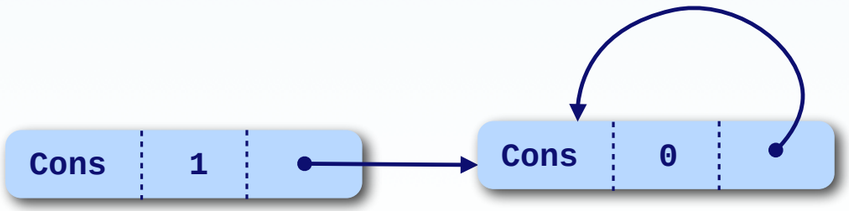
Intensional description of an infinite list

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






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

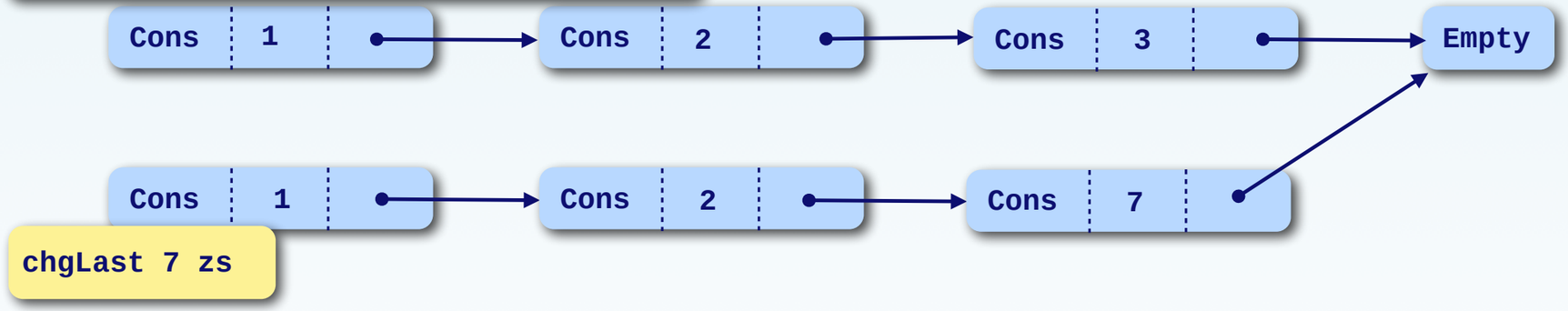


Changing Data Structures

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

Data types value are read-only

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




```
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, *where blocks*, …

Haskell?

Research in PL ?

Research Experience as Undergrad ?

Grad School ?