COMP3258
Functional Programming

Tutorial Session 4: List Comprehension and Higher-order Functions

List Comprehensions

$$\texttt{[x\^{}2 | }\underbrace{\texttt{x} \leftarrow \texttt{[1..5]}}_{\text{Generators}}\texttt{]}$$

List Comprehensions

$$[(x,\ y) \mid \underbrace{x \leftarrow [1..5],\ y \leftarrow [1..10]}_{\text{Multiple (Dependent) Generators}}]$$

List Comprehensions

```
[(x, y) | x ← [1..5] | y ← [1..10]]
```

Parallel Generators (with ParallelListComp extension)

Refer to: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/parallel_list_comprehensions.html

List Comprehensions

$$[x\text{^}2 \mid x \leftarrow [1..5],\ \underbrace{\text{even } x}_{\text{Guards}}]$$

List Comprehensions

```
[y | x ← [1..5], let y = x^2]
```

Local Declaration

Question 1 (3 mins)

A triple (x, y, z) of positive integers is called *Pythagorean* if $x^2 + y^2 = z^2$.

Use list comprehension to implement the function pythagoreans that finds all pythagorean triples with x, y, and z all less than or equal to the parameter.

```
pythagoreans :: Int → [(Int, Int, Int)]

> pythagoreans 5
[(3,4,5),(4,3,5)]
```

```
pythagoreans :: Int → [(Int, Int, Int)]
pythagoreans n = [(x, y, z) | x ← [1..n], y ← [1..n], z ← [1..n], x^2 + y^2 == z^2]
```

Question 2 (3 mins)

A positive integer is *perfect* if it's equal to the sum of all of its factors, excluding the number itself.

Use list comprehension, to implement a function perfects that finds all perfect numbers less than its parameter.

```
perfects :: Int → [Int]

> perfects 500
[6,28,496]
```

```
perfects :: Int → [Int]
perfects n = [x | x ← [1..n], sum (factors x) == x]
```

High-order Functions

- `map, filter, all, any, zipWith`

map

```haskell
----------------------------------------------
--              map
----------------------------------------------

-- | \(\mathcal{O}(n)\). 'map' @f xs@ is the list obtained by applying @f@ to
-- each element of @xs@, i.e.,
--
-- > map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
-- > map f [x1, x2, ...] == [f x1, f x2, ...]
--
-- >>> map (+1) [1, 2, 3]
-- [2,3,4]
map :: (a -> b) -> [a] -> [b]
{-# NOINLINE [0] map #-}
  -- We want the RULEs "map" and "map/coerce" to fire first.
  -- map is recursive, so won't inline anyway,
  -- but saying so is more explicit, and silences warnings
map _ []     = []
map f (x:xs) = f x : map f xs
```

filter

```haskell
-- | \(\mathcal{O}(n)\). 'filter', applied to a predicate and a list, returns
-- the list of those elements that satisfy the predicate; i.e.,
--
-- > filter p xs = [ x | x <- xs, p x]
--
-- >>> filter odd [1, 2, 3]
-- [1,3]
{-# NOINLINE [1] filter #-}
filter :: (a -> Bool) -> [a] -> [a]
filter _pred []    = []
filter pred (x:xs)
  | pred x         = x : filter pred xs
  | otherwise      = filter pred xs
```

zipWith

```haskell
-- >>> zipWith (+) [1, 2, 3] [4, 5, 6]
-- [5,7,9]
--
-- 'zipWith' is right-lazy:
--
-- >>> let f = undefined
-- >>> zipWith f [] undefined
-- []
--
-- 'zipWith' is capable of list fusion, but it is restricted to its
-- first list argument and its resulting list.
{-# NOINLINE [1] zipWith #-}  -- See Note [Fusion for zipN/zipWithN]
zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith f = go
  where
    go [] _ = []
    go _ [] = []
    go (x:xs) (y:ys) = f x y : go xs ys
```

Question 3 (2 mins)

Define a function filtmap that takes expressions like the list comprehension *[f x | x <- xs, p x]* using the functions map and filter.

```
filtmap :: (a → b) → (a → Bool) → [a] → [b]
```

```
filtmap :: (a → b) → (a → Bool) → [a] → [b]
filtmap f p = map f . filter p
```

Question 4 (2 mins)

Implement the library function zipWith with zip and map.

```
zipWith :: (a → b → c) → [a] → [b] → [c]
```

```
zipWith :: (a → b → c) → [a] → [b] → [c]
zipWith f xs ys = map (\(x,y) → f x y) (zip xs ys)
```

Folding Right and Left

Folding Right Patterns

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

product :: [Int] → Int
product []     = 1
product (x:xs) = x * (product xs)

all :: (a → Bool) → [a] → Bool
all p []     = True
all p (x:xs) = (p x) && (all p xs)
```

Folding Right Patterns

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

product :: [Int] -> Int
product []     = 1
product (x:xs) = (*) x (product xs)

all :: (a -> Bool) -> [a] -> Bool
all p []     = True
all p (x:xs) = (&&) (p x) (all p xs)
```

Folding Right Patterns

```
func []     = z
func (x:xs) = f x (func xs)

foldr :: (a → b → b) → b → [a] → b
```

Using foldr

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

product :: [Int] -> Int
product xs = foldr (*) 1 xs

all :: (a -> Bool) -> [a] -> Bool
all p xs = foldr (\x r -> p x && r) True xs
```

Folding Left Patterns

```haskell
sum :: [Int] → Int
sum = sum' 0
    where
        sum' v [] = v
        sum' v (x:xs) = sum' (v+x) xs

product :: [Int] → Int
product = product' 1
    where
        product' v [] = v
        product' v (x:xs) = product' (v*x) xs

all :: (a → Bool) → [a] → Bool
all p xs = all' True
    where
        all' v [] = v
        all' v (x:xs) = all' (v && p x) xs
```

Folding Left Patterns

```
func v []     = v
func v (x:xs) = func (f v x) xs

foldr :: (a → b → b) → b → [a] → b
```

foldr vs. foldl

```
foldr :: (a → b → b) → b → [a] → b

foldr (-) 0 (1 : (2 : (3 : [])))
= 1 - (2 - (3 - 0))
= 2

foldl :: (b → a → b) → b → [a] → b

foldl (-) 0 (1 : (2 : (3 : [])))
= ((0 - 1) - 2) - 3
= -6
```

foldr vs. foldl

- Traverse (same direction) + Folding (different)
- Circuit Cut

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Question 5 (5 mins)

Re-implement the following library functions with a single fold (foldl)

- `length, filter, unzip, reverse`

```haskell
length :: [a] -> Int
length = foldl (\r x -> r + 1) 0

filter :: (a -> Bool) -> [a] -> [a]
filter p = foldl (\r x -> if p x then r ++ [x] else r) []

unzip :: [(a, b)] -> ([a], [b])
unzip = foldl (\(as, bs) (xa, xb) -> (as ++ [xa], bs ++ [xb])) ([],[])

reverse :: [a] -> [a]
reverse = foldl (\r x -> x : r) []
```

Lazy Evaluation

Lazy Evaluation

- Avoids doing unnecessary evaluation;
- Ensures termination whenever possible;
- Supports programming with infinite lists;
- Allows programs to be more modular

Lazy Evaluation (Recipe)

- It evaluates outside-in instead of inside-out.
- It evaluates inside only when needed.
- It evaluates only enough.

```
square x = x * x
```

Outside-in (aka. outermost)

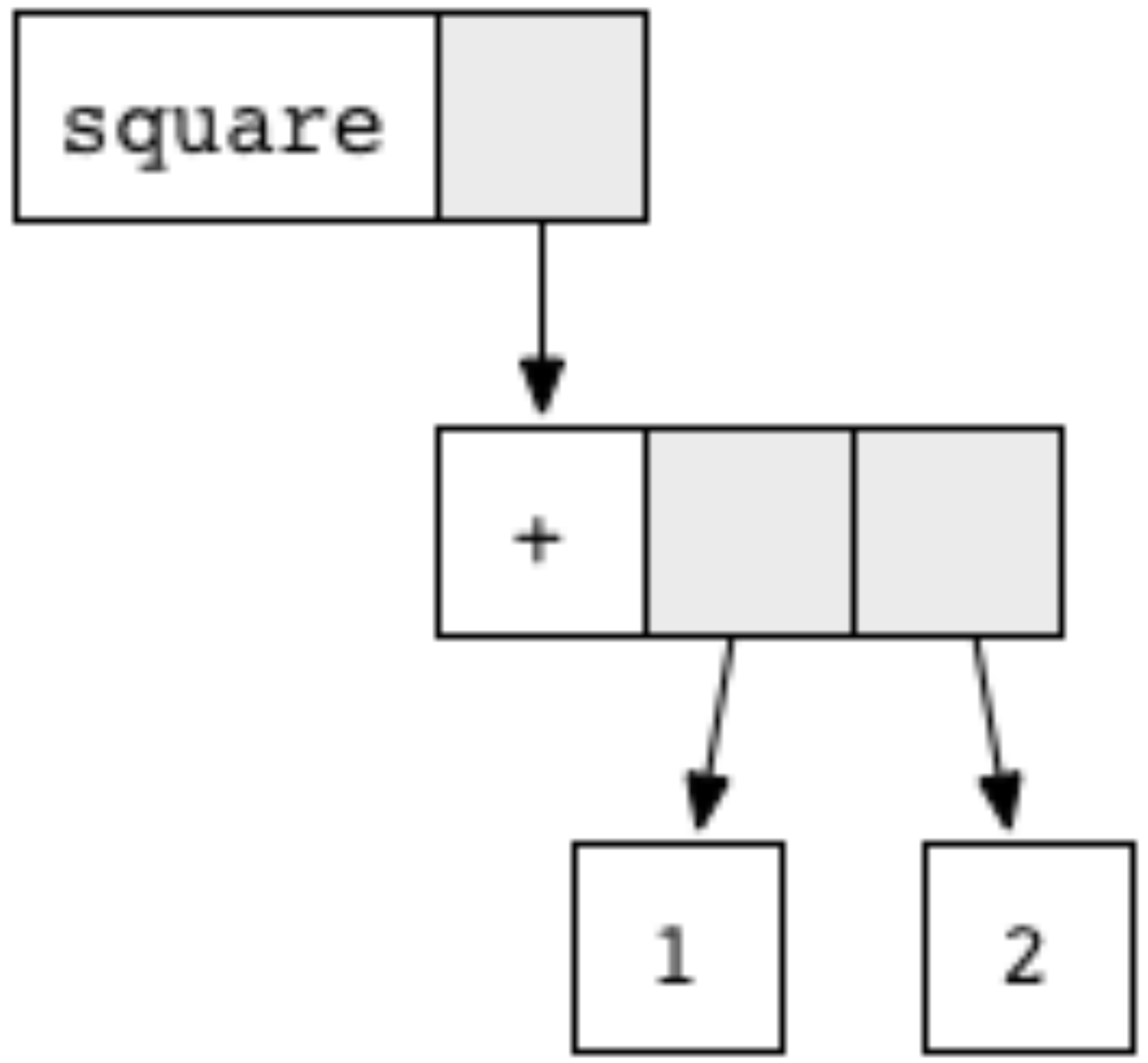
```
square (1+2)
⇒ (1+2) * (1+2)
⇒ 3 * (1+2)
⇒ 3 * 3
⇒ 9
```

Inside-out (innermost)

```
square (1+2)
⇒ square 3
⇒ 3 * 3
⇒ 9
```

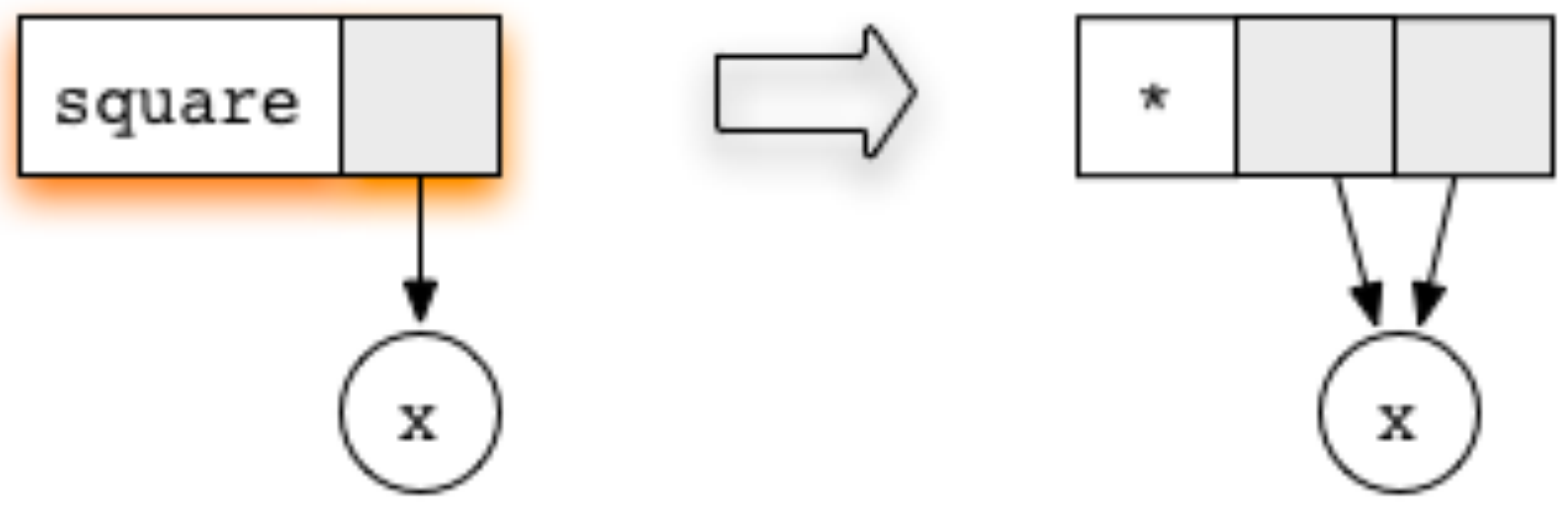
Graph Reduction (Optional)

```
square (1+2)
```

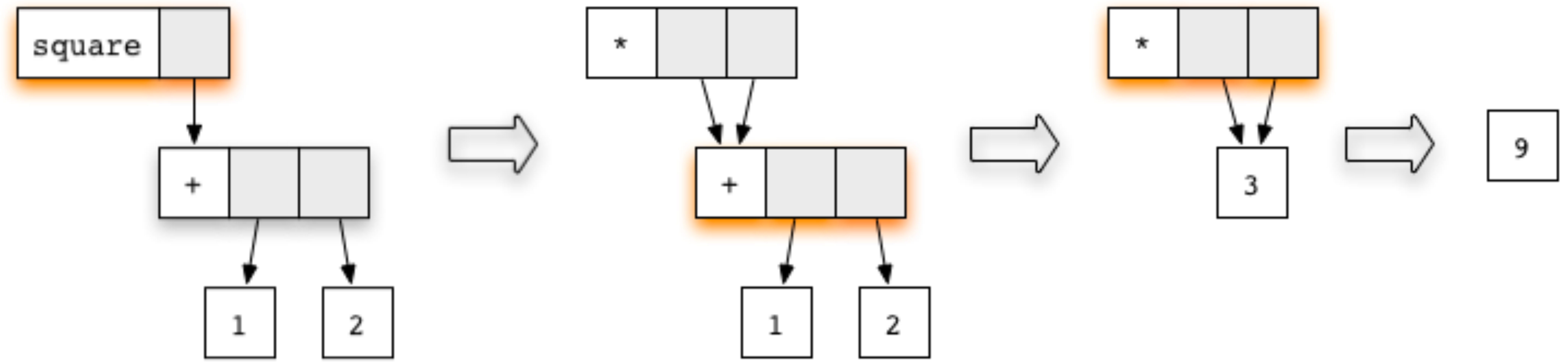


Graph Reduction (Optional)

Unlike tree representation,
the graph can share an expression



Graph Reduction (Optional)



Efficiency scales modularly

```
prefix :: Eq a ⇒ [a] → [a] → Bool
prefix xs ys = and (zipWith (==) xs ys)
```

```
prefix "Haskell" "eager"
```

Efficiency scales modularly

```
prefix :: Eq a ⇒ [a] → [a] → Bool
prefix xs ys = and (zipWith (==) xs ys)
```

```
prefix "Haskell" "eager"
⇒  and (zipWith (==) "Haskell "eager")
```

Efficiency scales modularly

```haskell
prefix :: Eq a ⇒ [a] → [a] → Bool
prefix xs ys = and (zipWith (==) xs ys)
```

```haskell
prefix "Haskell" "eager"
⇒  and (zipWith (==) "Haskell "eager")
⇒  and ('H' == 'e' : zipWith (==) "askell" "ager")
```

Efficiency scales modularly

```
prefix :: Eq a => [a] -> [a] -> Bool
prefix xs ys = and (zipWith (==) xs ys)
```

```
prefix "Haskell" "eager"
=>  and (zipWith (==) "Haskell "eager")
=>  and ('H' == 'e' : zipWith (==) "askell" "ager")
=>  'H' == 'e' && and (zipWith (==) "askell" "ager")
```

Efficiency scales modularly

```
prefix :: Eq a ⇒ [a] → [a] → Bool
prefix xs ys = and (zipWith (==) xs ys)
```

```
prefix "Haskell" "eager"
⇒  and (zipWith (==) "Haskell "eager")
⇒  and ('H' == 'e' : zipWith (==) "askell" "ager")
⇒  'H' == 'e' && and (zipWith (==) "askell" "ager")
⇒  False && and (zipWith (==) "askell" "ager")
```

Efficiency scales modularly

```
prefix :: Eq a ⇒ [a] → [a] → Bool
prefix xs ys = and (zipWith (==) xs ys)
```

```
prefix "Haskell" "eager"
⇒  and (zipWith (==) "Haskell "eager")
⇒  and ('H' == 'e' : zipWith (==) "askell" "ager")
⇒  'H' == 'e' && and (zipWith (==) "askell" "ager")
⇒  False && and (zipWith (==) "askell" "ager")
⇒  False
```

demand-driven evaluation