MOD08: Functional Programming

Marco Gerards

29-04-2019

Organisation of the course

Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Advanced type classes
	- Parser combinators and ParSec (application)
4	- Code generation (application)
	- TBD
5-7	- Project

Organisation of the course

Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Advanced type classes
	- Parser combinators and ParSec (application)
4	- Code generation (application)
	- TBD
5-7	- Project

This lecture: learning goals

Focus on *deeper* understanding of:

- Evaluation and execution mechanisms
  - Substitution
  - Lazy evaluation
  - Function application and currying
- Concepts and importance of typing
- Basic concepts of functional programming
  - Recursion
  - List comprehension
  - Higher-order functions (`map`, `foldl`, `foldr`, ...)

This lecture: learning goals

Focus on *deeper* understanding of:

- Evaluation and execution mechanisms
  - Substitution
  - Lazy evaluation
  - Function application and currying
- Concepts and importance of typing
- Basic concepts of functional programming
  - Recursion
  - List comprehension
  - Higher-order functions (`map`, `foldl`, `foldr`, ...)

This prepares you for the next weeks

- Solve non-trivial programming problems in FP
  - Blocks 2+3: parsing

This lecture: plan for today

- More background and examples:
  - Currying (+ function application)
  - List comprehension
  - Fold

This lecture: plan for today

- More background and examples:
  - Currying (+ function application)
  - List comprehension
  - Fold

- Quiz: 10(+2) questions
  - VoxVote; as before
  - Work in pairs
  - End scores are shown
    - ...but you may vote anonymously
  - For each question:
    - One minute: reading question; think about it for yourself
    - Two minutes: discussion and voting
    - Class discussion
    - Explanation and background

Currying and function application

```
f :: a -> b -> c -> d
```

Currying and function application

```
f :: a -> b -> c -> d
```

- Interpretation:
  - `f` has arguments of type `a`,`b`,`c`
  - `f` produces a value of type `d`

Currying and function application

```
f :: a -> b -> c -> d
```

- Interpretation:
  - `f` has arguments of type `a`,`b`,`c`
  - `f` produces a value of type `d`
- Currying:
  - `f` has an argument of type `a`
  - `f` produces a function of type `b -> c -> d`

Currying and function application

```
f :: a -> b -> c -> d
```

- Interpretation:
  - `f` has arguments of type `a`,`b`,`c`
  - `f` produces a value of type `d`
- Currying:
  - `f` has an argument of type `a`
  - `f` produces a function of type `b -> c -> d`
- Function application
  - `f x :: b -> c -> d`

Currying and function application

```
f :: a -> b -> c -> d
```

- Interpretation:
  - `f` has arguments of type `a`,`b`,`c`
  - `f` produces a value of type `d`
- Currying:
  - `f` has an argument of type `a`
  - `f` produces a function of type `b -> c -> d`
- Function application
  - `f x :: b -> c -> d`
- Arguments in function definition

```
g :: a -> b -> c -> d
g = f

h :: t -> a -> b -> c -> d
h x = f
```

Example: intersperse

```
Prelude> :m Data.List
Prelude Data.List> :t intersperse
intersperse ::  a -> [a] -> [a] Prelude Data.List>
intersperse ',' "abcde" "a,b,c,d,e"
```

- Base case: intersperse of an empty list?

  ```
  intersperse c [] = ...
  ```

Example: intersperse

```
Prelude> :m Data.List
Prelude Data.List> :t intersperse
intersperse ::  a -> [a] -> [a] Prelude Data.List>
intersperse ',' "abcde" "a,b,c,d,e"
```

- Base case: intersperse of an empty list?
  `intersperse c [] = []`
- Base case: list with one element?
  `intersperse c [x] = ...`

Example: intersperse

```
Prelude> :m Data.List
Prelude Data.List> :t intersperse
intersperse ::  a -> [a] -> [a] Prelude Data.List>
intersperse ',' "abcde" "a,b,c,d,e"
```

- Base case: intersperse of an empty list?
  `intersperse c [] = []`
- Base case: list with one element?
  `intersperse c [x] = [x]`

Example: intersperse

```
Prelude> :m Data.List
Prelude Data.List> :t intersperse
intersperse ::  a -> [a] -> [a] Prelude Data.List>
intersperse ',' "abcde" "a,b,c,d,e"
```

- Base case: intersperse of an empty list?
  `intersperse c [] = []`
- Base case: list with one element?
  `intersperse c [x] = [x]`
- Recursive case: intersperse of
  `intersperse c (x:xs) = ...  intersperse c xs ...`

Example: intersperse

```
Prelude> :m Data.List
Prelude Data.List> :t intersperse
intersperse ::  a -> [a] -> [a] Prelude Data.List>
intersperse ',' "abcde" "a,b,c,d,e"
```

- Base case: intersperse of an empty list?
  `intersperse c [] = []`
- Base case: list with one element?
  `intersperse c [x] = [x]`
- Recursive case: intersperse of
  `intersperse c (x:xs) = x : c : intersperse c xs`

Fibonacci using lazy evaluation

```
fibSeq :: [Integer]
fibSeq = 0 : 1 : zipWith (+) fibSeq (tail fibSeq)
```

- Assume `fibSeq` gives all fibonacci numbers:

`fibSeq`	0	1	1	2	3	5	8	13	...
`tail fibSeq`	1	1	2	3	5	8	13	...	
`zipWith (+) ...`	1	2	3	5	8	13	20	...	

Fibonacci using lazy evaluation

```
fibSeq :: [Integer]
fibSeq = 0 : 1 : zipWith (+) fibSeq (tail fibSeq)
```

- Assume `fibSeq` gives all fibonacci numbers:

`fibSeq`	0	1	1	2	3	5	8	13	...
`tail fibSeq`	1	1	2	3	5	8	13	...	
`zipWith (+) ...`	1	2	3	5	8	13	20	...	

- Does the function work for the first two values?

Fibonacci using lazy evaluation

```
fibSeq :: [Integer]
fibSeq = 0 : 1 : zipWith (+) fibSeq (tail fibSeq)
```

- Assume `fibSeq` gives all fibonacci numbers:

`fibSeq`	0	1	1	2	3	5	8	13	...
`tail fibSeq`	1	1	2	3	5	8	13	...	
`zipWith (+) ...`	1	2	3	5	8	13	20	...	

- Does the function work for the first two values?
- Can it calculate the third value? And the fourth?

Fibonacci using lazy evaluation

```
fibSeq :: [Integer]
fibSeq = 0 : 1 : zipWith (+) fibSeq (tail fibSeq)
```

- Assume `fibSeq` gives all fibonacci numbers:

`fibSeq`	0	1	1	2	3	5	8	13	...
`tail fibSeq`	1	1	2	3	5	8	13	...	
`zipWith (+) ...`	1	2	3	5	8	13	20	...	

- Does the function work for the first two values?
- Can it calculate the third value? And the fourth?
- If the first n values are available, can we calculate value $n+1$?

Fibonacci using lazy evaluation

```
fibSeq :: [Integer]
fibSeq = 0 : 1 : zipWith (+) fibSeq (tail fibSeq)
```

- Assume `fibSeq` gives all fibonacci numbers:

`fibSeq`	0	1	1	2	3	5	8	13	...
`tail fibSeq`	1	1	2	3	5	8	13	...	
`zipWith (+) ...`	1	2	3	5	8	13	20	...	

- Does the function work for the first two values?
- Can it calculate the third value? And the fourth?
- If the first n values are available, can we calculate value $n+1$?
- Try at home: use substitution

Quiz

Question 1

```
max' x y | x < y      = y
         | otherwise = x
```

`otherwise` is not a reserved keyword in Haskell, but defined as a function. What is a valid definition for `otherwise`?

A `otherwise x = x`

B `otherwise = (==)`

C `otherwise = True`

D `otherwise = False`

Answer 1: otherwise

```
max' x y | x < y     = y
         | otherwise = x
```

`otherwise` is not a reserved keyword in Haskell, but defined as a function. What is a valid definition for `otherwise`?

Type of the expression `x < y`?

Answer 1: otherwise

```
max' x y | x < y     = y
         | otherwise = x
```

`otherwise` is not a reserved keyword in Haskell, but defined as a function. What is a valid definition for `otherwise`?

Type of the expression `x < y`?

```
(<) x y :: Bool
```

Answer 1: otherwise

```
max' x y | x < y     = y
         | otherwise = x
```

`otherwise` is not a reserved keyword in Haskell, but defined as a function. What is a valid definition for `otherwise`?

Type of the expression `x < y`?

```
(<) x y :: Bool
```

What kind of expressions are used in guards?

Answer 1: otherwise

```
max' x y | x < y     = y
         | otherwise = x
```

`otherwise` is not a reserved keyword in Haskell, but defined as a function. What is a valid definition for `otherwise`?

Type of the expression `x < y`?

`(<) x y :: Bool`

What kind of expressions are used in guards?

```
*Main> :t otherwise
otherwise ::  Bool
```

Question 2

Consider the following Haskell function:

```
f (x,y) = (2*x, 0)
g (x,y) = (1,   10*x)

q = b
  where (x, y) = f (a,b)
        (a, b) = g (x,y)
```

What does q evaluate to?

A The compiler does not accept this code.

B The code hangs during evaluation

C 0

D 20

Answer 2: Lazy evaluation and substitution (1/2)

```
f (x,y) = (2*x, 0)
g (x,y) = (1,   10*x)

q = b
  where (x, y) = f (a,b)
        (a, b) = g (x,y)
```

Answer 2: Lazy evaluation and substitution (1/2)

```
f (x,y) = (2*x, 0)
g (x,y) = (1,   10*x)

q = b
  where (x, y) = f (a,b)
        (a, b) = g (x,y)
```

Equivalent:

```
q = b
  where (x, y) = (2*a, 0)
        (a, b) = (1,   10*x)
```

Answer 2: Lazy evaluation and substitution (2/2)

```
q = b
  where (x, y) = (2*a, 0)
        (a, b) = (1,   10*x)
```

Answer 2: Lazy evaluation and substitution (2/2)

```
q = b
  where (x, y) = (2*a, 0)
        (a, b) = (1,   10*x)
```

Equivalent:

```
q = b
  where x = 2*a
        y = 0
        a = 1
        b = 10*x
```

Lazy evaluation example

```
x = ([1..], 2)

y = snd x
```

Question 3

What is the type of the following function?

```
f q = map (2:) q
```

A `f :: Num a => a -> a`

B `f :: Num a => [a] -> [a]`

C `f :: Num a => [[a]] -> [[a]]`

D `f :: Num a => a -> [a] -> [a]`

Answer 3: types

What is the type of the following function?

```
f q = map (2:) q
```

Answer 3: types

What is the type of the following function?

```
f q = map (2:) q
```

```
(2:)  ::
```

Answer 3: types

What is the type of the following function?

```
f q = map (2:) q
```

```
(2:)  ::  ?
```

Answer 3: types

What is the type of the following function?

```
f q = map (2:) q
```

```
(2:)  ::   Num c =>  [c] -> [c]
```

Answer 3: types

What is the type of the following function?

```
f q = map (2:) q
```

```
(2:)  ::   Num c =>  [c] -> [c]
```

```
map ::
```

Answer 3: types

What is the type of the following function?

```
f q = map (2:) q
```

```
(2:)  ::   Num c =>  [c] -> [c]

map ::  ?
```

Answer 3: types

What is the type of the following function?

```
f q = map (2:) q
```

```
(2:)  ::   Num c =>  [c] -> [c]

map ::   ( a -> b ) -> [a] -> [b]
```

Answer 3: types

What is the type of the following function?

```
f q = map (2:) q
```

```
(2:)  ::   Num c =>  [c] -> [c]

map ::   ( a -> b ) -> [a] -> [b]
```

Answer 3: types

What is the type of the following function?

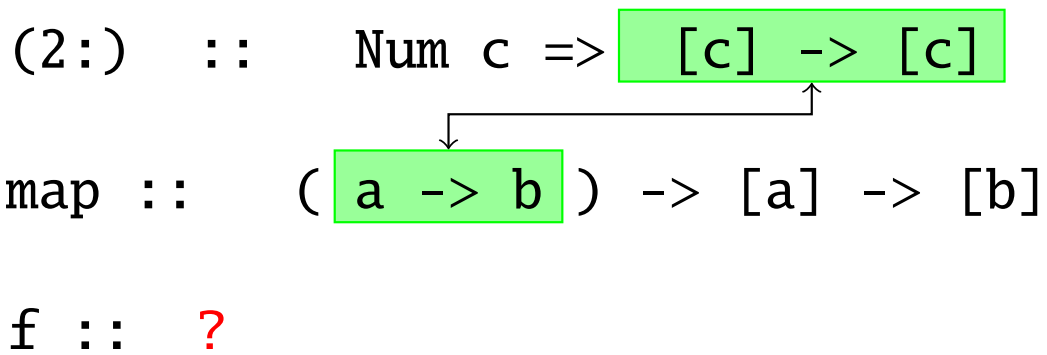
```
f q = map (2:) q
```

```
(2:)  ::   Num c =>  [c] -> [c]

map ::   ( a -> b ) -> [a] -> [b]

f ::
```

Answer 3: types

What is the type of the following function?

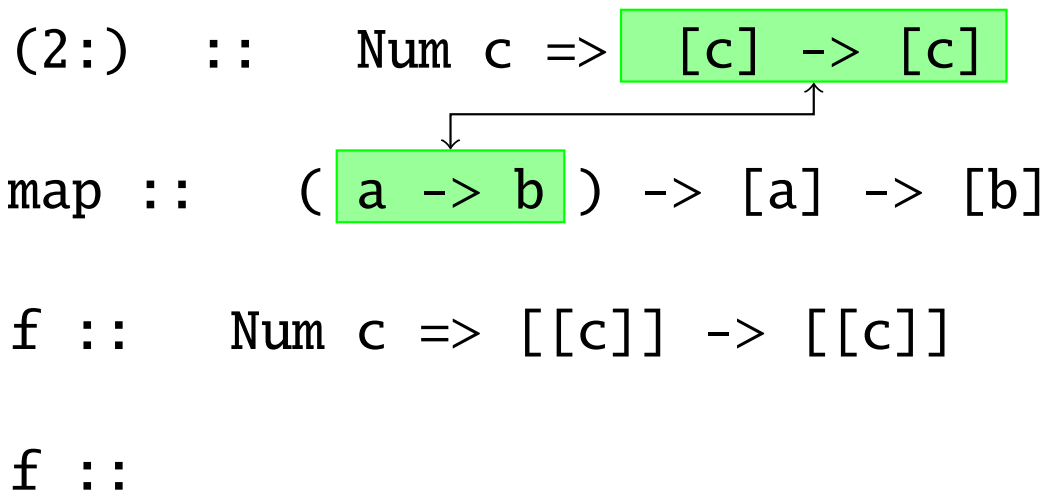
```
f q = map (2:) q
```



Answer 3: types

What is the type of the following function?

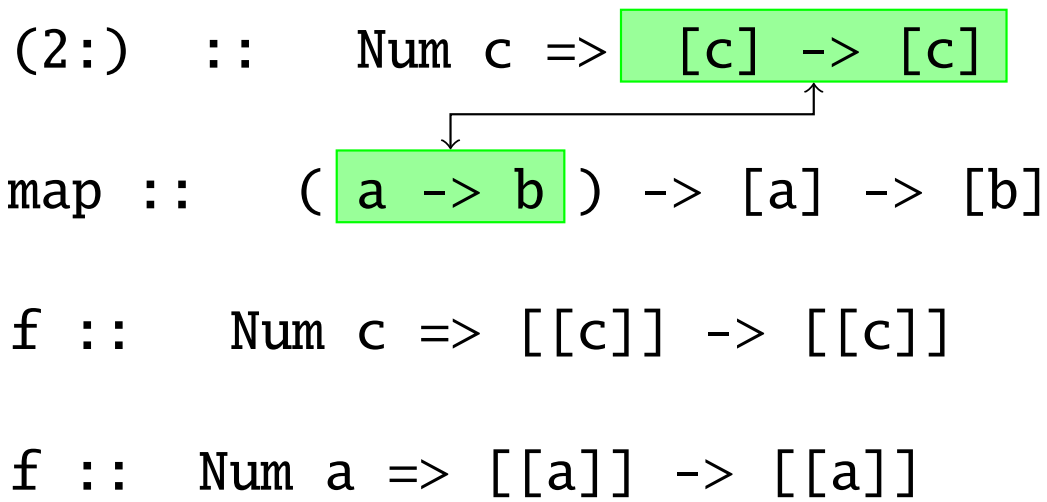
```
f q = map (2:) q
```



Answer 3: types

What is the type of the following function?

```
f q = map (2:) q
```



Question 4

The higher order function `flip` has the type

```
flip :: (a -> b -> c) -> b -> a -> c
```

What is the type of the following function?

```
g = flip (++)
```

A `g :: ([a] -> [b] -> [c]) -> [b] -> [a] -> [c]`

B `g :: b -> a -> c`

C `g :: [b] -> [a] -> [c]`

D `g :: [a] -> [a] -> [a]`

Answer 4: types

The higher order function `flip` has the type

```
flip :: (a -> b -> c) -> b -> a -> c
```

What is the type of the following function?

```
g = flip (++)
```

Answer 4: types

The higher order function `flip` has the type

```
flip :: (a -> b -> c) -> b -> a -> c
```

What is the type of the following function?

```
g = flip (++)
```

```
flip :: (a -> b -> c) -> b -> a -> c
```

```
(++) :: [a] -> [a] -> [a]
```

Answer 4: types

The higher order function `flip` has the type

```
flip :: (a -> b -> c) -> b -> a -> c
```

What is the type of the following function?

```
g = flip (++)
```

```
flip :: (a -> b -> c) -> b -> a -> c
```

```
(++) :: [a] -> [a] -> [a]
```

```
g :: [a] -> [a] -> [a]
g = flip (++)
```

Question 5

What is a valid type for the following function?

```
h []  = []:[]
h [x] = [x,x]
h xs  = xs
```

A `h :: [Int] -> [[Int]]`

B `h :: [[Int]] -> [Int]`

C `h :: [Int] -> [Int]`

D `h :: [[Int]] -> [[Int]]`

Type checking anology: Zebra riddle

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.

...

15 The Norwegian lives next to the blue house.

Questions: Who drinks water? Who owns the zebra?

Answer 5: types

What is a valid type for the following function?

```
h []  = [][]
h [x] = [x,x]
h xs  = xs
```

Answer 5: types

What is a valid type for the following function?

```
h []  = []:[]
h [x] = [x,x]
h xs  = xs
```

Function that receives one argument:

```
h ::  a -> b
```

Answer 5: types

What is a valid type for the following function?

```
h []  = []:[]
h [x] = [x,x]
h xs  = xs
```

Function that receives one argument:

```
h ::  a -> b
```

- What do we know about `a` and `b`?

Answer 5: types

What is a valid type for the following function?

```
h []  = []:[]
h [x] = [x,x]
h xs  = xs
```

Function that receives one argument:

```
h ::  a -> b
```

- What do we know about `a` and `b`?
- `h xs = xs` implies: `a == b`

Answer 5: types

What is a valid type for the following function?

```
h []  = []:[]
h [x] = [x,x]
h xs  = xs
```

Function that receives one argument:

```
h ::  a -> b
```

- What do we know about `a` and `b`?
- `h xs = xs` implies: `a == b`
- `h [] = [[]]` implies: `b` is a list of lists

Answer 5: types

What is a valid type for the following function?

```
h []  = []:[]
h [x] = [x,x]
h xs  = xs
```

Function that receives one argument:

```
h ::  a -> b
```

- What do we know about `a` and `b`?
- `h xs = xs` implies: `a == b`
- `h [] = [[]]` implies: `b` is a list of lists
- `h :: [[a]] -> [[a]]`

Question 6

What is the type of the function `f` defined below?

```
f = tail [head, head]
```

A `f :: [[a] -> a]`

B `f :: [a -> a]`

C `f :: [a] -> a`

D `f :: a -> a`

Answer 6

What is the type of the function `f` defined below?

```
f = tail [head, head]
```

Answer 6

What is the type of the function `f` defined below?

```
f = tail [head, head]
```

- Type signatures of `head` and `tail`:

```
head :: [a] -> a
tail :: [a] -> [a]
```

Answer 6

What is the type of the function `f` defined below?

```
f = tail [head, head]
```

- Type signatures of `head` and `tail`:

```
head :: [a] -> a
tail :: [a] -> [a]
```

- Type of `[head]`

```
[head] :: [[a] -> a]
```

Answer 6

What is the type of the function `f` defined below?

```
f = tail [head, head]
```

- Type signatures of `head` and `tail`:

```
head :: [a] -> a
tail :: [a] -> [a]
```

- Type of `[head]`

```
[head] :: [[a] -> a]
```

- Type of `[head, head]`

```
[head, head] :: [[a] -> a]
```

Answer 6

What is the type of the function `f` defined below?

```
f = tail [head, head]
```

- Type signatures of `head` and `tail`:

```
head :: [a] -> a
tail :: [a] -> [a]
```

- Type of `[head]`

```
[head] :: [[a] -> a]
```

- Type of `[head, head]`

```
[head, head] :: [[a] -> a]
```

- Type of `tail [head, head]`

```
tail [head, head] :: [[a] -> a]
```

Question 7

Consider the following type signatures:

```
f :: Int -> Int -> Char
g :: Int -> Char
```

Which definition of g corresponds to the given type signature?

A `g x = f x`

B `g x = f`

C `g = \x -> f`

D `g x = f x x`

Answer 7: Currying and function application

Consider the following type signatures:

```
f :: Int -> Int -> Char
g :: Int -> Char
```

Which definition of g corresponds to the given type signature?

1. `g x = f x`

2. `g x = f`

3. `g   = \x -> f`

4. `g x = f x x`

Answer 7: Currying and function application

Consider the following type signatures:

```
f :: Int -> Int -> Char
g :: Int -> Char
```

Which definition of g corresponds to the given type signature?

```
   g :: Int -> Int -> Char
1. g x = f x

2. g x = f

3. g   = \x -> f

4. g x = f x x
```

Answer 7: Currying and function application

Consider the following type signatures:

```
f :: Int -> Int -> Char
g :: Int -> Char
```

Which definition of g corresponds to the given type signature?

```
   g :: Int -> Int -> Char
1. g x = f x

   g :: a -> Int -> Int -> Char
2. g x = f

3. g   = \x -> f

4. g x = f x x
```

Answer 7: Currying and function application

Consider the following type signatures:

```
f :: Int -> Int -> Char
g :: Int -> Char
```

Which definition of g corresponds to the given type signature?

```
   g :: Int -> Int -> Char
1. g x = f x

   g :: a -> Int -> Int -> Char
2. g x = f

   g :: a -> Int -> Int -> Char
3. g   = \x -> f


4. g x = f x x
```

Answer 7: Currying and function application

Consider the following type signatures:

```
f :: Int -> Int -> Char
g :: Int -> Char
```

Which definition of g corresponds to the given type signature?

```
   g :: Int -> Int -> Char
1. g x = f x

   g :: a -> Int -> Int -> Char
2. g x = f

   g :: a -> Int -> Int -> Char
3. g   = \x -> f

   g :: Int -> Char
4. g x = f x x
```

Question 8

Consider the following (correct) Haskell code:

```
vlen :: [Double] -> Double
vlen vs = sqrt $ sum $ map (^2) vs
```

Is the following alternative definition also correct?

```
vlen' :: [Double] -> Double
vlen' = sqrt $ sum $ map (^2)
```

A Yes, due to currying

B Yes, since this is the point-free style

C No, now `map` receives only one argument (not two)

D No, now `vlen'` does not receive an argument

Answer 8: right associative function application

```
vlen' = sqrt $ sum $ map (^2)
```

Answer 8: right associative function application

```
vlen' = sqrt $ sum $ map (^2)
```

Shorthand notation for:

```
vlen' = sqrt (sum (map (^2)))
```

Answer 8: right associative function application

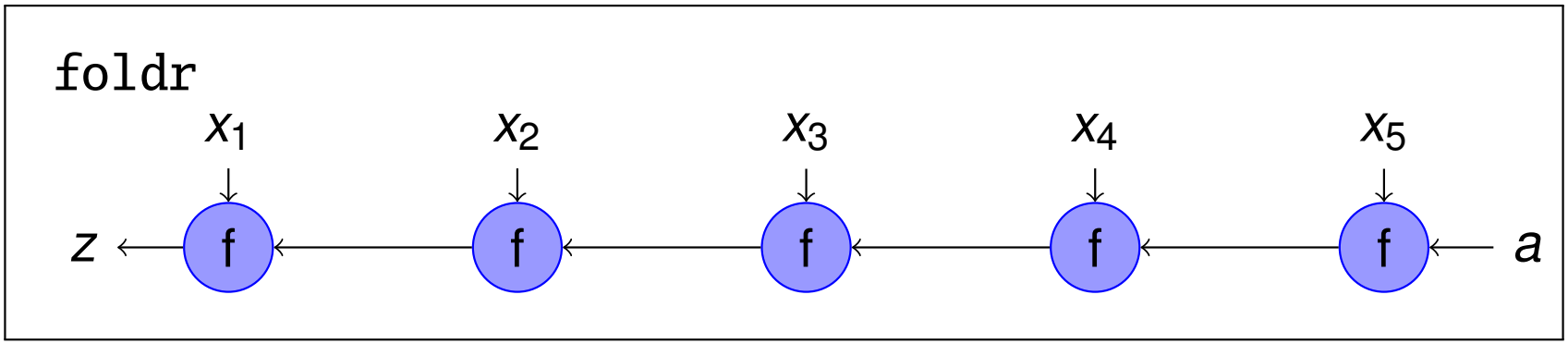
```
vlen' = sqrt $ sum $ map (^2)
```

Shorthand notation for:

```
vlen' = sqrt (sum (map (^2)))
```

```
Prelude> :t map (^2)
map (^2) ::  Num b => [b] -> [b]
```

Question 9



What is the result of the following expression?

```
foldr min 0 [4,1,99]
```

A `0`

B `4`

C `1`

D `99`

Answer 9: foldr

What is the result of the following expression?

```
foldr min 0 [4,1,99]
```

Answer 9: foldr

What is the result of the following expression?

```
foldr min 0 [4,1,99]
```

Right associative definition:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x [x0,x1,..,xn] = x0 # (x1 # (.. (xn # x)))
```

Answer 9: foldr

What is the result of the following expression?

```
foldr min 0 [4,1,99]
```

Right associative definition:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x [x0,x1,..,xn] = x0 # (x1 # (.. (xn # x)))
```

Substitute the values:

```
foldr min 0 [4,1,99] = 4 `min` (1 `min` (99 `min` 0))
```

Question 10

What is the result of the following expression?

```
foldr (:) [] [1,2,3]
```

A `[]`

B `[[1],[2],[3]]`

C `[1,2,3]`

D `[3,2,1]`

Answer 10: another interpretation of foldr

What is the result of the following expression?

```
foldr (:) [] [1,2,3]
```

Answer 10: another interpretation of foldr

What is the result of the following expression?

```
foldr (:) [] [1,2,3]
```

- Recall:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x [x0,x1,..,xn] = x0 # (x1 # (.. (xn # x)))
```

Answer 10: another interpretation of foldr

What is the result of the following expression?

```
foldr (:) [] [1,2,3]
```

- Recall:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x [x0,x1,..,xn] = x0 # (x1 # (.. (xn # x)))
```

- This can be written as:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x (x0:(x1:.. (xn:[]))) = x0#(x1#(.. (xn#x)))
```

Answer 10: another interpretation of foldr

What is the result of the following expression?

```
foldr (:) [] [1,2,3]
```

- Recall:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x [x0,x1,..,xn] = x0 # (x1 # (.. (xn # x)))
```

- This can be written as:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x (x0:(x1:.. (xn:[]))) = x0#(x1#(.. (xn#x)))
```

- `foldr` rewrites the list equation:
  - Replaces `:` by the reduction function `(#)`
  - Replaces `[]` by the starting value

Answer 10: another interpretation of foldr

What is the result of the following expression?

```
foldr (:) [] [1,2,3]
```

- Recall:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x [x0,x1,..,xn] = x0 # (x1 # (.. (xn # x)))
```

- This can be written as:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x (x0:(x1:.. (xn:[]))) = x0#(x1#(.. (xn#x)))
```

- `foldr` rewrites the list equation:
  - Replaces `:` by the reduction function `(#)`
  - Replaces `[]` by the starting value

- `foldr (:) [] [1,2,3] == [1,2,3]`

Question 11

Consider the following Haskell expression

```
falses = False : falses
x      = foldr (&&) True falses
```

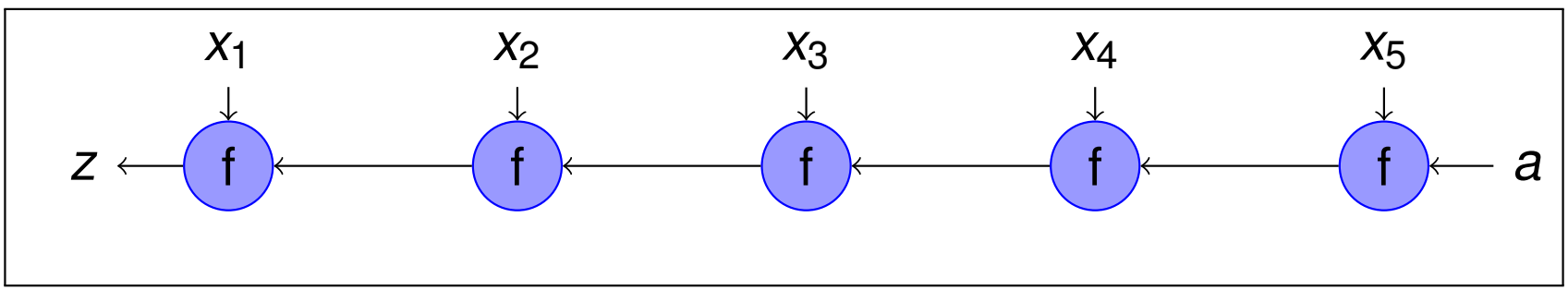
What do we know about the evaluation of `x`?

A It results in `False`, because of lazy evaluation

B It results in `False`, the compiler infers this using induction

C It "*hangs*", since `falses` has infinite length

D It "*hangs*", since `foldr` starts at the end of the list



Answer 11: lazy foldr

Consider the following Haskell expression

```
falses = False : falses
x      = foldr (&&) True falses
```

What do we know about the evaluation of `x`?

- Recall the equation:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x [x0,x1,..,xn] = x0 # (x1 # (.. (xn # x)))
```

- `x = False && (False && (False ( .. )))`

How about foldl?

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (#) x [x0,x1,..,xn] = ((x # x0) # x1)#..) # xn
```

- Can we use the following?

  ```
  foldl (:) [] [1,2,3]
  ```

How about foldl?

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (#) x [x0,x1,..,xn] = ((x # x0) # x1)#..) # xn
```

- Can we use the following?

  ```
  foldl (:) [] [1,2,3]
  ```

- No, since:

  ```
  foldl (:) [] [1,2,3] == (([] : 1) : 2) : 3
  ```

How about foldl?

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (#) x [x0,x1,..,xn] = ((x # x0) # x1)#..) # xn
```

- Can we use the following?

  ```
  foldl (:) [] [1,2,3]
  ```

- No, since:

  ```
  foldl (:) [] [1,2,3] == (([] : 1) : 2) : 3
  ```

- Can we use the following?

  ```
  falses = False : falses
  x      = foldl (&&) True falses
  ```

How about foldl?

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (#) x [x0,x1,..,xn] = ((x # x0) # x1)#..) # xn
```

- Can we use the following?

  ```
  foldl (:) [] [1,2,3]
  ```

- No, since:

  ```
  foldl (:) [] [1,2,3] == (([] : 1) : 2) : 3
  ```

- Can we use the following?

  ```
  falses = False : falses
  x      = foldl (&&) True falses
  ```

- No, since:

  ```
  x = (((True && False) && False) && ...) && False
  ```

Question 12

What result is produced by the following Haskell expression?

```
[ (y,x) | x <- "ab", y <- [1,2] ]
```

A `[(1,'a'),(2,'b')]`

B `[(1,"ab"),(2,"ab")]`

C `[(1,'a'),(2,'a'),(1,'b'),(2,'b')]`

D `[(1,'a'),(1,'b'),(2,'a'),(2,'b')]`

Answer 12: list comprehension

What result is produced by the following Haskell expression?

```
[ (y,x) | x <- "ab", y <- [1,2] ]
```

- Generators are processed in order

Answer 12: list comprehension

What result is produced by the following Haskell expression?

```
[ (y,x) | x <- "ab", y <- [1,2] ]
```

- Generators are processed in order
- For each `x`, generate all `ys`

Answer 12: list comprehension

What result is produced by the following Haskell expression?

```
[ (y,x) | x <- "ab", y <- [1,2] ]
```

- Generators are processed in order
- For each `x`, generate all `ys`
- Then use this to determine (y, x)

Answer 12: list comprehension

What result is produced by the following Haskell expression?

```
[ (y,x) | x <- "ab", y <- [1,2] ]
```

- Generators are processed in order
- For each `x`, generate all `ys`
- Then use this to determine (y, x)
- Result: `[(1,'a'),(2,'a'),(1,'b'),(2,'b')]`

Self study

- Block 1: “Introduction to FP” & “Higher-order functions”
  - Learn you a Haskell (Lipovaca): Chapters 1–6
  - Programming in Haskell (Hutton): Chapters 1–7

Self study

- Block 1: “Introduction to FP” & “Higher-order functions”
  - Learn you a Haskell (Lipovaca): Chapters 1–6
  - Programming in Haskell (Hutton): Chapters 1–7
- Lecture Wednesday (02-05): “Types and Type Classes”
  - Learn you a Haskell (Lipovaca): Chapter 8
  - Programming in Haskell (Hutton): Chapter 8, Chapter 12.1

Self study

- Block 1: “Introduction to FP” & “Higher-order functions”
  - Learn you a Haskell (Lipovaca): Chapters 1–6
  - Programming in Haskell (Hutton): Chapters 1–7
- Lecture Wednesday (02-05): “Types and Type Classes”
  - Learn you a Haskell (Lipovaca): Chapter 8
  - Programming in Haskell (Hutton): Chapter 8, Chapter 12.1
- Lecture Friday (04-05): “Parsing in Haskell”
  - Knowledge on parsing is assumed (top-down, (E)BNF, recursive descent)
  - MOD08 students: part of compiler construction
  - Other students: self-study; suggestions:
    - “CS143 Notes: Parsing”, David L. Dill (pages 1–25, see Canvas)
    - Compiler construction video lectures (Blackboard)