Functional Programming for Logicians

Péter Mekis

Department of Logic, ELTE Budapest

Session 3: 2019 February 25

Lazy evaluation

- Boolean expressions are evaluated only until their value is set:

```
> 1==2 && head "" == 'a'
```

Lazy evaluation

- Boolean expressions are evaluated only until their value is set:

```
> 1==2 && head "" == 'a'
False
```

Lazy evaluation

- Boolean expressions are evaluated only until their value is set:

```
> 1==2 && head "" == 'a'
False
```

- The same conditions in a different order raise error:

```
> head "" == 'a' && 1 == 2
```

Lazy evaluation

- Boolean expressions are evaluated only until their value is set:

```
> 1==2 && head "" == 'a'
False
```

- The same conditions in a different order raise error:

```
> head "" == 'a' && 1 == 2
*** Exception:  Prelude.head:  empty list
```

Lazy evaluation

- Guard conditions are evaluated only up to the first one that is true:

```
howLong ::  String -> String
howLong s
   | s == ""       = "empty word"
   | tail s == "" = "single letter word"
   | otherwise     = "multiletter word"
```

Lazy evaluation

- Guard conditions are evaluated only up to the first one that is true:

```
howLong ::  String -> String
howLong s
   | s == ""       = "empty word"
   | tail s == "" = "single letter word"
   | otherwise     = "multiletter word"
> howLong ""
```

Lazy evaluation

- Guard conditions are evaluated only up to the first one that is true:

```
howLong ::  String -> String
howLong s
   | s == ""       = "empty word"
   | tail s == "" = "single letter word"
   | otherwise     = "multiletter word"
> howLong ""
empty string
```

Lazy evaluation

- Guard conditions are evaluated only up to the first one that is true:

```
howLong ::  String -> String
howLong s
   | s == ""       = "empty word"
   | tail s == "" = "single letter word"
   | otherwise     = "multiletter word"
> howLong ""
empty string
```

- The same conditions in a different order raise error for ‘""’:

```
howLong' ::  String -> String
howLong' s
   | tail s == "" = "single letter string"
   | s == ""       = "empty string"
   | otherwise     = "multiletter word"
```

Lazy evaluation

- Guard conditions are evaluated only up to the first one that is true:

```
howLong ::  String -> String
howLong s
   | s == ""       = "empty word"
   | tail s == "" = "single letter word"
   | otherwise     = "multiletter word"
> howLong ""
empty string
```

- The same conditions in a different order raise error for ‘""’:

```
howLong' ::  String -> String
howLong' s
   | tail s == "" = "single letter string"
   | s == ""      = "empty string"
   | otherwise    = "multiletter word"
> howLong' ""
```

Lazy evaluation

- Guard conditions are evaluated only up to the first one that is true:

```
howLong ::  String -> String
howLong s
  | s == ""       = "empty word"
  | tail s == "" = "single letter word"
  | otherwise     = "multiletter word"
> howLong ""
empty string
```

- The same conditions in a different order raise error for ‘""’:

```
howLong' ::  String -> String
howLong' s
  | tail s == "" = "single letter string"
  | s == ""       = "empty string"
  | otherwise     = "multiletter word"
> howLong' ""
*** Exception:  Prelude.head:  empty list
```

Lazy evaluation

- otherwise' is just an alias for 'True'. howLong'' :: String -> String

```
howLong'' s
  | otherwise    = "multiletter word"
  | s == ""      = "empty word"
  | tail s == "" = "single letter word"
```

Lazy evaluation

- `otherwise` is just an alias for `True`. `howLong'' :: String -> String`

```
howLong'' s
  | otherwise    = "multiletter word"
  | s == ""      = "empty word"
  | tail s == "" = "single letter word"
> howLong'' ""
```

Lazy evaluation

- otherwise' is just an alias for 'True'. howLong'' :: String -> String

```
howLong'' s
  | otherwise    = "multiletter word"
  | s == ""      = "empty word"
  | tail s == "" = "single letter word"
> howLong'' ""
multiletter word
```

Lazy evaluation

- An infinite list of variables:

Lazy evaluation

- An infinite list of variables:
  `data Var = V Integer`

Lazy evaluation

- An infinite list of variables:

```
data Var = V Integer
indices :: [Integer]
```

Lazy evaluation

- An infinite list of variables:

```
data Var = V Integer
indices :: [Integer]
indices = [0..]
```

Lazy evaluation

- An infinite list of variables:

```
data Var = V Integer
indices :: [Integer]
indices = [0..]
variables = map V indices
```

Lazy evaluation

- An infinite list of variables:

```
data Var = V Integer
indices :: [Integer]
indices = [0..]
variables = map V indices
```

- The first k numbers divisible by n:

Lazy evaluation

- An infinite list of variables:

```
data Var = V Integer
indices :: [Integer]
indices = [0..]
variables = map V indices
```

- The first k numbers divisible by n:

```
firstDiv :: Int -> Integer -> Integer
```

Lazy evaluation

- An infinite list of variables:

```
data Var = V Integer
indices :: [Integer]
indices = [0..]
variables = map V indices
```

- The first k numbers divisible by n:

```
firstDiv :: Int -> Integer -> Integer
firstDiv k n = take k [i | i <- [0..], i `mod` n == 0]
```

Lazy evaluation

- An infinite list of variables:

```
data Var = V Integer
indices :: [Integer]
indices = [0..]
variables = map V indices
```

- The first k numbers divisible by n:

```
firstDiv :: Int -> Integer -> Integer
firstDiv k n = take k [i | i <- [0..], i `mod` n == 0]
```

Don't use this, not very fast...