Programming in Haskell: Lecture 6

S P Suresh

August 26, 2019

Lists

- To describe a collection of values

Lists

- To describe a collection of values
- `[1,2,3,1]` is a list of **`Int`**

Lists

- To describe a collection of values
- `[1,2,3,1]` is a list of **`Int`**
- **`[True,False,True]`** is a list of **`Bool`**

Lists

- To describe a collection of values
- `[1,2,3,1]` is a list of `Int`
- `[True,False,True]` is a list of `Bool`
- Elements of a list must be of a uniform type

Lists

- To describe a collection of values
- `[1,2,3,1]` is a list of `Int`
- `[True,False,True]` is a list of `Bool`
- Elements of a list must be of a uniform type
- Cannot write `[1,2,True]` or `[3,'a']`

Lists

- List with values of type `T` has type `[T]`

Lists

- List with values of type `T` has type `[T]`
- `[1,2,3,1] :: [Int]`

Lists

- List with values of type `T` has type `[T]`
- `[1,2,3,1] :: [Int]`
- `[True,False,True] :: [Bool]`

Lists

- List with values of type `T` has type `[T]`
- `[1,2,3,1] :: [Int]`
- `[True,False,True] :: [Bool]`
- `[]` denotes the empty list, for all types

Lists

- List with values of type `T` has type `[T]`
- `[1,2,3,1] :: [Int]`
- `[True,False,True] :: [Bool]`
- `[]` denotes the empty list, for all types
- Lists can be nested

Lists

- List with values of type `T` has type `[T]`
- `[1,2,3,1] :: [Int]`
- `[True,False,True] :: [Bool]`
- `[]` denotes the empty list, for all types
- Lists can be nested
- `[[3,2], [], [7,7,7]] :: [[Int]]`

Internal representation

- To build a list, add one element at a time to the front (left)

Internal representation

- To build a list, add one element at a time to the front (left)
- Operator to append an element is :

Internal representation

- To build a list, add one element at a time to the front (left)
- Operator to append an element is `:`
- `1:[2,3] ---> [1,2,3]`

Internal representation

- To build a list, add one element at a time to the front (left)
- Operator to append an element is `:`
- `1:[2,3] ---> [1,2,3]`
- All Haskell lists are built this way, starting with `[]`

Internal representation

- To build a list, add one element at a time to the front (left)
- Operator to append an element is `:`
- `1:[2,3] ---> [1,2,3]`
- All Haskell lists are built this way, starting with `[]`
- `[1,2,3]` is actually `1:(2:(3:[]))`

Internal representation

- To build a list, add one element at a time to the front (left)
- Operator to append an element is `:`
- `1:[2,3] ---> [1,2,3]`
- All Haskell lists are built this way, starting with `[]`
- `[1,2,3]` is actually `1:(2:(3:[]))`
- `:` is right associative, so `1:2:3:[]` is `1:(2:(3:[]))`

Internal representation

- To build a list, add one element at a time to the front (left)
- Operator to append an element is `:`
- `1:[2,3] ---> [1,2,3]`
- All Haskell lists are built this way, starting with `[]`
- `[1,2,3]` is actually `1:(2:(3:[]))`
- `:` is right associative, so `1:2:3:[]` is `1:(2:(3:[]))`
- `1:[2,3] == 1:2:3:[]`, `1:2:[3] == [1,2,3]`, ...all return **`True`**

Decomposing lists

- Functions `head` and `tail`

Decomposing lists

- Functions `head` and `tail`
- `head (x:xs) ---> x`

Decomposing lists

- Functions `head` and `tail`
- `head (x:xs) ---> x`
- `tail (x:xs) ---> xs`

Decomposing lists

- Functions `head` and `tail`
- `head (x:xs) ---> x`
- `tail (x:xs) ---> xs`
- Both undefined for `[]`

Decomposing lists

- Functions `head` and `tail`
- `head (x:xs) ---> x`
- `tail (x:xs) ---> xs`
- Both undefined for `[]`
- Note: `head` returns a value, `tail` returns a list

Decomposing lists

- Functions `head` and `tail`
- `head (x:xs) ---> x`
- `tail (x:xs) ---> xs`
- Both undefined for `[]`
- Note: `head` returns a value, `tail` returns a list
- `null l` is `True` exactly when `l` is `[]`

Defining functions on lists

- Recall inductive definition of numeric functions

Defining functions on lists

- Recall inductive definition of numeric functions
- Base case is `f 0`

Defining functions on lists

- Recall inductive definition of numeric functions
- Base case is `f 0`
- Define `f n` in terms of `n` and `f (n-1)`

Defining functions on lists

- Recall inductive definition of numeric functions
- Base case is `f 0`
- Define `f n` in terms of `n` and `f (n-1)`
- For lists, induction on list structure

Defining functions on lists

- Recall inductive definition of numeric functions
- Base case is `f 0`
- Define `f n` in terms of `n` and `f (n-1)`
- For lists, induction on list structure
- Base case is the empty list

Defining functions on lists

- Recall inductive definition of numeric functions
- Base case is `f 0`
- Define `f n` in terms of `n` and `f (n-1)`
- For lists, induction on list structure
- Base case is the empty list
- For a non-empty list `l`

Defining functions on lists

- Recall inductive definition of numeric functions
- Base case is `f 0`
- Define `f n` in terms of `n` and `f (n-1)`
- For lists, induction on list structure
- Base case is the empty list
- For a non-empty list `l`
  - define `f l` in terms of `head l` and `f (tail l)`

Examples

- Increment every element in an integer list

```
addOne :: [Integer] -> [Integer]
addOne l = if null l then [] else head l + 1 : addOne (tail l)
```

Examples

- Increment every element in an integer list

```
addOne :: [Integer] -> [Integer]
addOne l = if null l then [] else head l + 1 : addOne (tail l)
```

- Compute the length of a list

```
myLength :: [Integer] -> Integer
myLength l = if null l then 0 else 1 + myLength (tail l)
```

Pattern matching

- `[]` is the pattern that matches the empty list

Pattern matching

- `[]` is the pattern that matches the empty list
- A nonempty list decomposes uniquely as `x:xs`

Pattern matching

- `[]` is the pattern that matches the empty list
- A nonempty list decomposes uniquely as `x:xs`
- Pattern matching implicitly separates head and tail

Pattern matching

- `[]` is the pattern that matches the empty list
- A nonempty list decomposes uniquely as `x:xs`
- Pattern matching implicitly separates head and tail
- Empty list will not match this pattern `x:xs`

Pattern matching

- `[]` is the pattern that matches the empty list
- A nonempty list decomposes uniquely as `x:xs`
- Pattern matching implicitly separates head and tail
- Empty list will not match this pattern `x:xs`
- We should the pattern with parentheses: `(x:xs)`

```
myLength :: [Integer] -> Integer
myLength []         = 0
myLength (x:xs)     = 1 + myLength xs
```

Pattern matching

- `[]` is the pattern that matches the empty list
- A nonempty list decomposes uniquely as `x:xs`
- Pattern matching implicitly separates head and tail
- Empty list will not match this pattern `x:xs`
- We should the pattern with parentheses: `(x:xs)`

```
myLength :: [Integer] -> Integer
myLength []        = 0
myLength (x:xs)    = 1 + myLength xs
```

- Built-in function **`length`**

Examples

- `addAtEnd x l` adds `x` at the end of `l`

```
addAtEnd :: Int -> [Int] -> [Int]
addAtEnd x []         = [x]
addAtEnd x (y:ys)     = y:addAtEnd x ys
```

Examples

- `addAtEnd x l` adds `x` at the end of `l`

```
addAtEnd :: Int -> [Int] -> [Int]
addAtEnd x []        = [x]
addAtEnd x (y:ys)    = y:addAtEnd x ys
```

- `attach l1 l2` attaches `l2` to the end of `l1`

```
attach :: [Int] -> [Int] -> [Int]
attach l1 []        = l1
attach l1 (y:ys)    = attach (addAtEnd l1 y) ys
```

Examples

- `attach l1 l2` requires more than `length l1 * length l2` steps

Examples

- `attach l1 l2` requires more than `length l1 * length l2` steps
- Smarter version recurses on the first list:

```
attach :: [Int] -> [Int] -> [Int]
attach [] l2        = l2
attach (x:xs) l2    = x:attach xs l2
```

Examples

- `attach l1 l2` requires more than `length l1 * length l2` steps
- Smarter version recurses on the first list:

```
attach :: [Int] -> [Int] -> [Int]
attach [] l2          = l2
attach (x:xs) l2      = x:attach xs l2
```

- This takes around `length l1` steps

Examples

- `attach l1 l2` requires more than `length l1 * length l2` steps
- Smarter version recurses on the first list:

```
attach :: [Int] -> [Int] -> [Int]
attach [] l2        = l2
attach (x:xs) l2    = x:attach xs l2
```

- This takes around `length l1` steps
- Built-in function `++`

Examples

- `attach l1 l2` requires more than `length l1 * length l2` steps
- Smarter version recurses on the first list:

```
attach :: [Int] -> [Int] -> [Int]
attach [] l2        = l2
attach (x:xs) l2    = x:attach xs l2
```

- This takes around `length l1` steps
- Built-in function `++`
- `[3,2,4] ++ [5,7,6]` is `[3,2,4,5,7,6]`

Example: `valueAtPosition`

- Positions in a list `l` range from `0` to `length l - 1`

Example: `valueAtPosition`

- Positions in a list `l` range from `0` to `length l - 1`
- `valueAtPosition n l` returns the value at position `n` of list `l`

```
valueAtPosition :: Int -> [Int] -> Int
valueAtPosition 0 (x:xs) = x
valueAtPosition n (x:xs) = valueAtPosition (n-1) xs
```

Example: `valueAtPosition`

- Positions in a list `l` range from `0` to `length l - 1`
- `valueAtPosition n l` returns the value at position `n` of list `l`

```
valueAtPosition :: Int -> [Int] -> Int
valueAtPosition 0 (x:xs) = x
valueAtPosition n (x:xs) = valueAtPosition (n-1) xs
```

- What happens if the list is empty?

Example: `valueAtPosition`

- Positions in a list `l` range from `0` to `length l - 1`
- `valueAtPosition n l` returns the value at position `n` of list `l`

```
valueAtPosition :: Int -> [Int] -> Int
valueAtPosition 0 (x:xs) = x
valueAtPosition n (x:xs) = valueAtPosition (n-1) xs
```

- What happens if the list is empty?
- What if `n >= length l`?

Example: `valueAtPosition`

- Positions in a list `l` range from `0` to `length l - 1`
- `valueAtPosition n l` returns the value at position `n` of list `l`

```
valueAtPosition :: Int -> [Int] -> Int
valueAtPosition 0 (x:xs) = x
valueAtPosition n (x:xs) = valueAtPosition (n-1) xs
```

- What happens if the list is empty?
- What if `n >= length l`?
- What if `n < 0`?

Example: `valueAtPosition`

- Handling the problem cases:

```
valueAtPosition n l
    | null l            = error "Empty list"
    | n < 0             = error "Negative index"
    | n >= length l     = error "Index too large"
    | otherwise         = f n l
      where f n (x:xs) = if n == 0 then x else f (n-1) xs
```

Example: `valueAtPosition`

- Handling the problem cases:

```
valueAtPosition n l
    | null l             = error "Empty list"
    | n < 0              = error "Negative index"
    | n >= length l      = error "Index too large"
    | otherwise          = f n l
      where f n (x:xs) = if n == 0 then x else f (n-1) xs
```

- `f n l` will be called only when `l` is non-empty and `0 <= n <= length l - 1`

Example: `valueAtPosition`

- Handling the problem cases:

```
valueAtPosition n l
    | null l            = error "Empty list"
    | n < 0             = error "Negative index"
    | n >= length l     = error "Index too large"
    | otherwise         = f n l
      where f n (x:xs) = if n == 0 then x else f (n-1) xs
```

- `f n l` will be called only when `l` is non-empty and `0 <= n <= length l - 1`
- No error in recursive calls of `f`

Example: `valueAtPosition`

- Handling the problem cases:

```
valueAtPosition n l
    | null l          = error "Empty list"
    | n < 0           = error "Negative index"
    | n >= length l   = error "Index too large"
    | otherwise       = f n l
      where f n (x:xs) = if n == 0 then x else f (n-1) xs
```

- `f n l` will be called only when `l` is non-empty and `0 <= n <= length l - 1`
- No error in recursive calls of `f`
- `error` prints an error message and aborts (matches any type)

List notation

- `valueAtPosition` is equivalent to the built-in operator `!!`

List notation

- `valueAtPosition` is equivalent to the built-in operator `!!`
- Positions in any list are numbered from `0` to `length l - 1`

List notation

- `valueAtPosition` is equivalent to the built-in operator `!!`
- Positions in any list are numbered from `0` to `length l - 1`
- `l!!j` is the value at position `j` of the list

List notation

- `valueAtPosition` is equivalent to the built-in operator `!!`
- Positions in any list are numbered from `0` to `length l - 1`
- `l!!j` is the value at position `j` of the list
- Accessing position `j` takes time proportional to `j`

List notation

- `valueAtPosition` is equivalent to the built-in operator `!!`
- Positions in any list are numbered from `0` to `length l - 1`
- `l!!j` is the value at position `j` of the list
- Accessing position `j` takes time proportional to `j`
- Need to "peel off" applications of the `:` operator

List notation

- `valueAtPosition` is equivalent to the built-in operator `!!`
- Positions in any list are numbered from `0` to `length l - 1`
- `l!!j` is the value at position `j` of the list
- Accessing position `j` takes time proportional to `j`
- Need to "peel off" applications of the `:` operator
- Arrays, in other languages, allow constant-time access to any position

List notation

- `[m..n] ---> [m, m+1, ..., n]`

List notation

- `[m..n] ---> [m, m+1, ..., n]`
- Returns empty list if `m < n`

List notation

- `[m..n] ---> [m, m+1, ..., n]`
- Returns empty list if `m < n`
  - `[1..7] ---> [1,2,3,4,5,6,7]`

List notation

- `[m..n] ---> [m, m+1, ..., n]`
- Returns empty list if `m < n`
    - `[1..7] ---> [1,2,3,4,5,6,7]`
    - `[3..3] ---> [3]`

List notation

- `[m..n] ---> [m, m+1, ..., n]`
- Returns empty list if `m < n`
    - `[1..7] ---> [1,2,3,4,5,6,7]`
    - `[3..3] ---> [3]`
    - `[5..4] ---> []`

List notation

- `[m..n] ---> [m, m+1, ..., n]`
- Returns empty list if `m < n`
  - `[1..7] ---> [1,2,3,4,5,6,7]`
  - `[3..3] ---> [3]`
  - `[5..4] ---> []`
- Can skip values (arithmetic progression)

List notation

- `[m..n] ---> [m, m+1, ..., n]`
- Returns empty list if `m < n`
    - `[1..7] ---> [1,2,3,4,5,6,7]`
    - `[3..3] ---> [3]`
    - `[5..4] ---> []`
- Can skip values (arithmetic progression)
    - `[1,3..8] ---> [1,3,5,7]`

List notation

- `[m..n] ---> [m, m+1, ..., n]`
- Returns empty list if `m < n`
    - `[1..7] ---> [1,2,3,4,5,6,7]`
    - `[3..3] ---> [3]`
    - `[5..4] ---> []`
- Can skip values (arithmetic progression)
    - `[1,3..8] ---> [1,3,5,7]`
    - `[2,5..19] ---> [2,5,8,11,14,17]`

List notation

- `[m..n] ---> [m, m+1, ..., n]`
- Returns empty list if `m < n`
    - `[1..7] ---> [1,2,3,4,5,6,7]`
    - `[3..3] ---> [3]`
    - `[5..4] ---> []`
- Can skip values (arithmetic progression)
    - `[1,3..8] ---> [1,3,5,7]`
    - `[2,5..19] ---> [2,5,8,11,14,17]`
- Can have descending sequences

List notation

- `[m..n] ---> [m, m+1, ..., n]`
- Returns empty list if `m < n`
    - `[1..7] ---> [1,2,3,4,5,6,7]`
    - `[3..3] ---> [3]`
    - `[5..4] ---> []`
- Can skip values (arithmetic progression)
    - `[1,3..8] ---> [1,3,5,7]`
    - `[2,5..19] ---> [2,5,8,11,14,17]`
- Can have descending sequences
    - `[8,7..5] ---> [8,7,6,5]`

List notation

- `[m..n] ---> [m, m+1, ..., n]`
- Returns empty list if `m < n`
    - `[1..7] ---> [1,2,3,4,5,6,7]`
    - `[3..3] ---> [3]`
    - `[5..4] ---> []`
- Can skip values (arithmetic progression)
    - `[1,3..8] ---> [1,3,5,7]`
    - `[2,5..19] ---> [2,5,8,11,14,17]`
- Can have descending sequences
    - `[8,7..5] ---> [8,7,6,5]`
    - `[12,8..(-9)] ---> [12,8,4,0,-4,-8]`

Reversing a list

- Remove the head

Reversing a list

- Remove the head
- Recursively reverse the tail

Reversing a list

- Remove the head
- Recursively reverse the tail
- Add head at end

```
myReverse :: [Int] -> [Int]
myReverse []     = []
myReverse (x:xs) = myReverse xs ++ [x]
```

Reversing a list

- Remove the head
- Recursively reverse the tail
- Add head at end

```
myReverse :: [Int] -> [Int]
myReverse []     = []
myReverse (x:xs) = myReverse xs ++ [x]
```

- Number of steps is proportional to n^2, where n is the length

Reversing a list

- Remove the head
- Recursively reverse the tail
- Add head at end

```
myReverse :: [Int] -> [Int]
myReverse []     = []
myReverse (x:xs) = myReverse xs ++ [x]
```

- Number of steps is proportional to n^2, where n is the length
- Built-in function **`reverse`** is smarter

Built-in functions on lists

```
head (x:xs)     = x

tail (x:xs)     = xs

length []       = 0
length (x:xs)   = 1 + length xs

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

Built-in functions on lists

- `init` returns all but the last element of a list

Built-in functions on lists

- `init` returns all but the last element of a list
- `last` returns the last element of a list

Built-in functions on lists

- `init` returns all but the last element of a list
- `last` returns the last element of a list
- Undefined for the empty list

Built-in functions on lists

- `init` returns all but the last element of a list
- `last` returns the last element of a list
- Undefined for the empty list
- Possible implementations:

```
init [x]       = []
init (x:xs)    = x:init xs

last [x]       = x
last (x:xs)    = last xs
```

Built-in functions on lists

- **`take`** `n l` returns the first `n` elements of `l`

Built-in functions on lists

- `take n l` returns the first `n` elements of `l`
- `drop n l` returns all but the first `n` elements of `l`

Built-in functions on lists

- `take n l` returns the first `n` elements of `l`
- `drop n l` returns all but the first `n` elements of `l`
- `take n l ++ drop n l == l`

```
take _ []                 = []
take n (x:xs) | n <= 0    = []
              | otherwise = x:take (n-1) xs

drop _ []                 = []
drop n (x:xs) | n <= 0    = x:xs
              | otherwise = drop (n-1) xs
```

Built-in function: `reverse`

- The built-in `reverse` takes time proportional to n, the length of the list

Built-in function: `reverse`

- The built-in `reverse` takes time proportional to n, the length of the list
- **Strategy**: Repeatedly extract head and place it in front of an accumulator list

Built-in function: `reverse`

- The built-in `reverse` takes time proportional to n, the length of the list
- **Strategy**: Repeatedly extract head and place it in front of an accumulator list
- The list is automatically reversed

```
reverse l                = revInto [] l
  where
    revInto a []         = a
    revInto a (x:xs) = revInto (x:a) xs
```