Programming in Haskell: Lecture 8

S P Suresh

September 4, 2019

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
```

Built-in functions: `take` *and* `drop`

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

Built-in functions: `take` *and* `drop`

- `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: **take** *and* **drop**

- `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: `splitAt`

- `splitAt n l = (take n l, drop n l)`

Built-in function: `splitAt`

- `splitAt n l = (take n l, drop n l)`
- Can be defined directly:

```
splitAt _ []          = ([], [])
splitAt n (x:xs)
    | n < = 0         = ([], x:xs)
    | otherwise       = (x:fst (splitAt (n-1) xs),
                         snd (splitAt (n-1) xs))
```

Built-in function: `splitAt`

- `splitAt n l = (take n l, drop n l)`
- Can be defined directly:

```
splitAt _ []          = ([], [])
splitAt n (x:xs)
    | n < = 0         = ([], x:xs)
    | otherwise       = (x:fst (splitAt (n-1) xs),
                          snd (splitAt (n-1) xs))
```

- Two recursive calls to `splitAt (n-1)`

Built-in function: **splitAt**

- `splitAt n l = (take n l, drop n l)`
- Can be defined directly:

```
splitAt _ []       = ([], [])
splitAt n (x:xs)
    | n < = 0      = ([], x:xs)
    | otherwise    = (x:fst (splitAt (n-1) xs),
                         snd (splitAt (n-1) xs))
```

- Two recursive calls to `splitAt (n-1)`
- Very inefficient – time proportional to 2^n

Built-in function: **splitAt**

- Much better version:

```
splitAt _ []              = ([], [])
splitAt n (x:xs)
    | n < = 0             = ([], x:xs)
    | otherwise           = (x:ys, zs)
        where (ys, zs) = splitAt (n-1) xs
```

Built-in function: **splitAt**

- Much better version:

```
splitAt _ []          = ([], [])
splitAt n (x:xs)
    | n < = 0         = ([], x:xs)
    | otherwise       = (x:ys, zs)
      where (ys, zs) = splitAt (n-1) xs
```

- Only one recursive call to `splitAt (n-1)`

Built-in function: `splitAt`

- Much better version:

```
splitAt _ []            = ([], [])
splitAt n (x:xs)
    | n < = 0           = ([], x:xs)
    | otherwise         = (x:ys, zs)
        where (ys, zs) = splitAt (n-1) xs
```

- Only one recursive call to `splitAt (n-1)`
- Running time is proportional to n

Built-in function: `splitAt`

- Much better version:

```
splitAt _ []            = ([], [])
splitAt n (x:xs)
    | n < = 0           = ([], x:xs)
    | otherwise         = (x:ys, zs)
       where (ys, zs) = splitAt (n-1) xs
```

- Only one recursive call to `splitAt (n-1)`
- Running time is proportional to n
- Local definitions helps avoid repeated computation of same value

The datatype **Char**

- Values are written with single quotes

The datatype **`Char`**

- Values are written with single quotes
- `'a'`, `'3'`, `'%'`, `'#'`, …

The datatype `Char`

- Values are written with single quotes
- `'a'`, `'3'`, `'%'`, `'#'`, …
- Character symbols stored in a table (e.g. ASCII, Unicode)

The datatype `Char`

- Values are written with single quotes
- `'a'`, `'3'`, `'%'`, `'#'`, ...
- Character symbols stored in a table (e.g. ASCII, Unicode)
- Functions `ord` and `chr` connect characters and table

The datatype **`Char`**

- Values are written with single quotes
- `'a'`, `'3'`, `'%'`, `'#'`, ...
- Character symbols stored in a table (e.g. ASCII, Unicode)
- Functions **`ord`** and **`chr`** connect characters and table
- Inverses of each other: `c == chr (ord c)`, `j == ord (chr j)`

The datatype `Char`

- Values are written with single quotes
- `'a'`, `'3'`, `'%'`, `'#'`, …
- Character symbols stored in a table (e.g. ASCII, Unicode)
- Functions `ord` and `chr` connect characters and table
- Inverses of each other: `c == chr (ord c)`, `j == ord (chr j)`
- Note: import `Data.Char` to use `ord` and `chr`

Example: `toUpper`

- Convert lowercase letters to uppercase

Example: `toUpper`

- Convert lowercase letters to uppercase
- Brute-force, enumerate all cases:

```
toUpper 'a' = 'A'
toUpper 'b' = 'B'
toUpper 'c' = 'C'
           ...
           ...
toUpper 'x' = 'X'
toUpper 'y' = 'Y'
toUpper 'z' = 'Z'
```

Example: `toUpper`

- `'a'`, ..., `'z'` have contiguous `ord` values

Example: `toUpper`

- `'a'`, ..., `'z'` have contiguous `ord` values
- Same with `'A'`, ..., `'Z'` and `'0'`, ..., `'9'`

Example: `toUpper`

- `'a'`, ..., `'z'` have contiguous `ord` values
- Same with `'A'`, ..., `'Z'` and `'0'`, ..., `'9'`
- Can compare two characters to see which one appears earlier in the table

Example: `toUpper`

- `'a'`, …, `'z'` have contiguous `ord` values
- Same with `'A'`, …, `'Z'` and `'0'`, …, `'9'`
- Can compare two characters to see which one appears earlier in the table
- Smarter solution for `toUpper`:

```
toUpper :: Char -> Char
toUpper c
    | ('a' <= c && c <= 'z')
              = chr (ord c + (ord 'A' - ord 'a'))
    | otherwise = c
```

Built-in functions on `Char`

- Character classification: `isSpace`, `isUpper`, `isLower`, `isDigit`, `isAlpha`, `isAlphaNum`

Built-in functions on `Char`

- Character classification: `isSpace`, `isUpper`, `isLower`, `isDigit`, `isAlpha`, `isAlphaNum`
- Case conversion: `toLower`, `toUpper`

Built-in functions on `Char`

- Character classification: `isSpace`, `isUpper`, `isLower`, `isDigit`, `isAlpha`, `isAlphaNum`
- Case conversion: `toLower`, `toUpper`
- Single digit characters: `digitToInt`, `intToDigit`

Built-in functions on `Char`

- Character classification: `isSpace`, `isUpper`, `isLower`, `isDigit`, `isAlpha`, `isAlphaNum`
- Case conversion: `toLower`, `toUpper`
- Single digit characters: `digitToInt`, `intToDigit`
- Numeric representation: `ord`, `chr`

Strings

- A string is a sequence of characters

Strings

- A string is a sequence of characters
- In Haskell, `String` is a synonym for `[Char]`

Strings

- A string is a sequence of characters
- In Haskell, `String` is a synonym for `[Char]`
- Type synonyms are defined using the `type` keyword

```
type String = [Char]
```

Strings

- A string is a sequence of characters
- In Haskell, `String` is a synonym for `[Char]`
- Type synonyms are defined using the `type` keyword

```
type String = [Char]
```

- Special syntax for strings

Strings

- A string is a sequence of characters
- In Haskell, `String` is a synonym for `[Char]`
- Type synonyms are defined using the `type` keyword

```
type String = [Char]
```

- Special syntax for strings
  - `"hello"` is syntactic sugar for `['h','e','l','l','o']`

Strings

- A string is a sequence of characters
- In Haskell, **`String`** is a synonym for **`[Char]`**
- Type synonyms are defined using the **`type`** keyword

```
type String = [Char]
```

- Special syntax for strings
  - `"hello"` is syntactic sugar for `['h','e','l','l','o']`
  - The empty string, denoted, `""`, is just `[]`

Strings

- A string is a sequence of characters
- In Haskell, `String` is a synonym for `[Char]`
- Type synonyms are defined using the `type` keyword

```
type String = [Char]
```

- Special syntax for strings
  - `"hello"` is syntactic sugar for `['h','e','l','l','o']`
  - The empty string, denoted, `""`, is just `[]`
  - Recall: `[]` is the empty list of all types

Strings

- A string is a sequence of characters
- In Haskell, `String` is a synonym for `[Char]`
- Type synonyms are defined using the `type` keyword

```
type String = [Char]
```

- Special syntax for strings
  - `"hello"` is syntactic sugar for `['h','e','l','l','o']`
  - The empty string, denoted, `""`, is just `[]`
  - Recall: `[]` is the empty list of all types
- Usual list functions like `length`, `reverse`, …can be used on `String`

Example: occurs

- Search for a character in a string

Example: `occurs`

- Search for a character in a string
- `occurs c s` returns **`True`** exactly when `c` occurs in string

```
occurs :: Char -> String -> Bool
occurs _ ""      = False
occurs c (a:as) = c == a || occurs c as
```

Example: occurs

- Search for a character in a string
- `occurs c s` returns **`True`** exactly when `c` occurs in string

```
occurs :: Char -> String -> Bool
occurs _ ""     = False
occurs c (a:as) = c == a || occurs c as
```

- Just a version of the general function **`elem`** on lists

Example: capitalize

- Convert all lowercase letters in a string to uppercase

```
capitalize :: String -> String
capitalize ""     = ""
capitalize (a:as) = toUpper a : capitalize as
```

Example: `capitalize`

- Convert all lowercase letters in a string to uppercase

```
capitalize :: String -> String
capitalize ""     = ""
capitalize (a:as) = toUpper a : capitalize as
```

- Apply the same function (`toUpper`) to every element in the list

Example: `capitalize`

- Convert all lowercase letters in a string to uppercase

```
capitalize :: String -> String
capitalize ""     = ""
capitalize (a:as) = toUpper a : capitalize as
```

- Apply the same function (**`toUpper`**) to every element in the list
- We will revisit this pattern later

Example: `position`

- `position c s` : first position in `s` where `c` occurs

Example: `position`

- `position c s` : first position in `s` where `c` occurs
- Return `length s` if no occurrence of `c` in `s`

Example: `position`

- `position c s` : first position in `s` where `c` occurs
- Return `length s` if no occurrence of `c` in `s`
- `position 'a' "battle axe" = 1`

Example: `position`

- `position c s` : first position in `s` where `c` occurs
- Return **`length`** `s` if no occurrence of `c` in `s`
- `position 'a' "battle axe" = 1`
- `position 'd' "battle axe" = 10`

Example: `position`

- `position c s` : first position in `s` where `c` occurs
- Return `length s` if no occurrence of `c` in `s`
- `position 'a' "battle axe" = 1`
- `position 'd' "battle axe" = 10`
- Simple recursive program

```
position :: Char -> String -> Int
position c ""    = 0
position c (d:ds)
    | c == d    = 0
    | otherwise = 1 + (position c ds)
```

Maybe

- `position c s == length s` indicates that `c` does not occur in `s`

`Maybe`

- `position c s == length s` indicates that `c` does not occur in `s`
- Need a more direct way to indicate non-occurrence

Maybe

- `position c s == length s` indicates that `c` does not occur in `s`
- Need a more direct way to indicate non-occurrence
- Use the type `Maybe Int`

`Maybe`

- `position c s == length s` indicates that `c` does not occur in `s`
- Need a more direct way to indicate non-occurrence
- Use the type `Maybe Int`
- For any type `t`, `Maybe t` is also type

`Maybe`

- `position c s == length s` indicates that `c` does not occur in `s`
- Need a more direct way to indicate non-occurrence
- Use the type `Maybe Int`
- For any type `t`, `Maybe t` is also type
- Values of type `Maybe t`:

`Maybe`

- `position c s == length s` indicates that `c` does not occur in `s`
- Need a more direct way to indicate non-occurrence
- Use the type `Maybe Int`
- For any type `t`, `Maybe t` is also type
- Values of type `Maybe t`:
  - `Nothing`

`Maybe`

- `position c s == length s` indicates that `c` does not occur in `s`
- Need a more direct way to indicate non-occurrence
- Use the type `Maybe Int`
- For any type `t`, `Maybe t` is also type
- Values of type `Maybe t`:
  - `Nothing`
  - `Just x` for all `x` of type `t`

Example: a better `position`

- Return `Nothing` if `c` does not occur in `s`

```
position :: Char -> String -> Maybe Int
position c ""   = Nothing
position c (d:ds)
    | c == d    = Just 0
    | otherwise = case position ds of
                    Nothing -> Nothing
                    Just x  -> Just (x+1)
```

Example: Counting words

- `wordc` : count the number of words in a string

Example: Counting words

- `wordc` : count the number of words in a string
- Words separated by white space: `' '`, `'\t'`, `'\n'` &c.

Example: Counting words

- `wordc` : count the number of words in a string
- Words separated by white space: `' '`, `'\t'`, `'\n'` &c.
- Maybe we can count the number of white spaces in the string:

```
wordc :: String -> Int
wordc ""   = 0
wordc (d:ds)
    | isSpace d = 1 + wordc ds
    | otherwise = wordc ds
```

Example: Counting words

- `wordc` : count the number of words in a string
- Words separated by white space: `' '`, `'\t'`, `'\n'` &c.
- Maybe we can count the number of white spaces in the string:

```
wordc :: String -> Int
wordc ""    = 0
wordc (d:ds)
    | isSpace d = 1 + wordc ds
    | otherwise = wordc ds
```

- Not correct: `wordc "abc     d"` will return `5`

Example: Correct `wordc`

- A word starts when previous character is a space and the current one is not

Example: Correct `wordc`

- A word starts when previous character is a space and the current one is not
- Add a space at the very beginning to apply same logic to first word

```
wordc :: String -> Int
wordc s           = go (' ':s)
go [c]            = 0
go (c:d:ds)
    | isSpace c && not (isSpace d)
                  = 1 + go (d:ds)
    | otherwise   = go (d:ds)
```

Tuples

- Keep multiple types of data together

Tuples

- Keep multiple types of data together
- Student info: `("Suresh", 3170, "01/01/2000")`

Tuples

- Keep multiple types of data together
- Student info: `("Suresh", 3170, "01/01/2000")`
- List of marks in a course

Tuples

- Keep multiple types of data together
- Student info: `("Suresh", 3170, "01/01/2000")`
- List of marks in a course
- `[("Ashvini", 85), ("Bharani", 90), ("Krittika", 87)]`

Tuples

- Keep multiple types of data together
- Student info: `("Suresh", 3170, "01/01/2000")`
- List of marks in a course
- `[("Ashvini", 85), ("Bharani", 90), ("Krittika", 87)]`
- `(3, -21) :: (Int, Int)`

Tuples

- Keep multiple types of data together
- Student info: `("Suresh", 3170, "01/01/2000")`
- List of marks in a course
- `[("Ashvini", 85), ("Bharani", 90), ("Krittika", 87)]`
- `(3, -21) :: (Int, Int)`
- `(13, True, 97) :: (Int, Bool, Int)`

Tuples

- Keep multiple types of data together
- Student info: `("Suresh", 3170, "01/01/2000")`
- List of marks in a course
- `[("Ashvini", 85), ("Bharani", 90), ("Krittika", 87)]`
- `(3, -21) :: (Int, Int)`
- `(13, True, 97) :: (Int, Bool, Int)`
- `([1,2], "abcd") :: ([Int], String)`

Example: Marks list

- A mark list is a list of pairs

Example: Marks list

- A mark list is a list of pairs
- Each pair consists of the student name and marks

Example: Marks list

- A mark list is a list of pairs
- Each pair consists of the student name and marks
- **`lookup`** finds the marks obtained by a student:

```
type Marklist = [(String, Int)]
lookup :: String -> Marklist -> Maybe Int
lookup n []       = Nothing
lookup n (name,marks):ml
    | n == name = Just marks
    | otherwise = lookup n ml
```