Computer Science 312

Haskell Strings and Tuples

One More List Operation: Insertion Sort

```
insert :: Ord a => a -> [a] ->[a]
insert item [] = [item]
insert item (x:xs)
    | item <= x = item : x : xs
    | otherwise = x : insert item xs
```

- If the list is empty, wrap a list around the item
- Else if the item <= the head of the list, add the item and the head to the tail
- Else add the head to the result of inserting the item into the tail

One More List Operation: Insertion Sort

```
insert :: Ord a => a -> [a] ->[a]
insert item [] = [item]
insert item (x:xs)
    | item <= x = item : x : xs
    | otherwise = x : insert item xs
```

Sort the list’s tail, and then insert the list’s head into the result

```
insertionSort :: Ord a => [a] -> [a]
insertionSort [] = []
insertionSort (x:xs) = insert x (insertionSort xs)
```

String Literals and The String Type

Strings are just lists of characters

```
Prelude> "Hi there!"
"Hi there!"

Prelude> :type "Hi there!"       -- A list of characters
"Hi there!" :: [Char]

Prelude> putStrLn "Hi there!"    -- Like Python’s print
Hi there!

Prelude> putStrLn "Hi\nthere!"
Hi
there!
```

String Operations

- All of the list operations (because strings are just lists of characters)
- The functions in **`Data.String`**
- Character-based functions in **`Data.Char`**

Data.String

```
Prelude> import Data.String

Prelude Data.String> words "Hi there, Ken"         -- split
["Hi","there,","Ken"]

Prelude Data.String> unwords ["Hi","there,","Ken"] -- join
"Hi there, Ken"

Prelude Data.String> lines "Hi\nthere, Ken"
["Hi","there, Ken"]
```

Operations for search, substring, etc., are in **`Data.List`**

Data.List

```
Prelude> import Data.List

Prelude Data.List> isPrefixOf "Ken" "Ken Lambert"
True

Prelude Data.List> isSuffixOf "bert" "Ken Lambert"
True

Prelude Data.List> isInfixOf "Lamb" "Ken Lambert"
True

Prelude Data.List Data.List> take 3 "Ken Lambert"
"Ken"

Prelude Data.List Data.List> drop 4 "Ken Lambert"
"Lambert"
```

Convert a String to Uppercase

```
import qualified Data.Char (toUpper)

toUpper :: String -> String
toUpper [] = []
toUpper (x:xs) = Data.Char.toUpper x : toUpper xs
```

You can use either **`[]`** or **`""`** for the empty string

`String` is a synonym for **`[Char]`**

The **show** Function

```
Prelude> :type show
show :: Show a => a -> String

Prelude> show 34
"34"

Prelude> show 3.14
"3.14"
```

Like **`str`** in Python, wraps quotes around its argument

Number the Lines of Text

```
numberLines :: String -> String
```

1. Extract a list of lines from the string

2. Prepend a line number to each line in this list

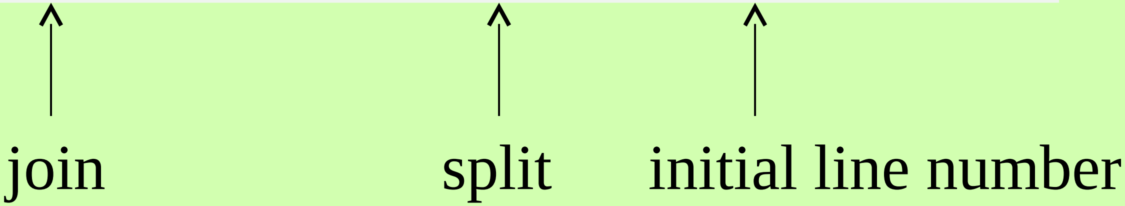
3. Join the lines together into a string

Define a helper function for step #2

Number the Lines of Text

```
import Data.String (lines, unlines)

numberLines :: String -> String
numberLines text = unlines (helper (lines text) 1)
```



join

split

initial line number

Number the Lines of Text

```
import Data.String (lines, unlines))

numberLines :: String -> String
numberLines text = unlines (helper (lines text) 1) where
    helper :: [String] -> Int -> [String]
    helper [] _ = []
    helper (x:xs) count =
        (show count ++ " " ++ x) : helper xs (count + 1)
```

The _ symbol matches any pattern, with no binding

Temporary Variables and **let**

```
import Data.String (lines, unlines)

numberLines :: String -> String
numberLines text = unlines (helper (lines text) 1) where
    helper :: [String] -> Int -> [String]
    helper [] _ = []
    helper (x:xs) count =
        (show count ++ " " ++ x) : helper xs (count + 1)
```

```
import Data.String (lines, unlines))

numberLines :: String -> String
numberLines text = unlines (helper (lines text) 1) where
    helper :: [String] -> Int -> [String]
    helper [] _ = []
    helper (x:xs) count =
        let thisLine = show count ++ " " ++ x
            restOfLines = helper xs (count + 1)
        in
            thisLine : restOfLines
```

Syntax of `let` / `in`

```
let
    <variable-1> = <expression-1>
    .
    .
    <variable-n> = <expression-n>
in
    <expression>
```

Creates a local context for variables with temporary bindings

Lists vs Tuples

A list is a sequence of items of the same type

```
Prelude> numbers = [100, 34, 67]

Prelude> :type numbers
numbers :: Num a => [a]
```

A tuple is a sequence of items of any types

```
Prelude> studentInfo = ("Stanley", 19, 3.56)
Prelude> :type studentInfo
studentInfo :: (Num b, Fractional c) => ([Char], b, c)]
```

Pattern Matching with Tutples

```
Prelude> studentInfo = ("Stanley", 19, 3.56)
Prelude> :type studentInfo
studentInfo :: (Num b, Fractional c) => ([Char], b, c)]
```

Extract components with a pattern:

```
Prelude> (name, age, gpa) = studentInfo

Prelude> name
"Stanley"

Prelude> age
19

Prelude> gpa
3.56
```

Association Lists

- Like a Python dictionary, associates a set of keys with values
- The key/value pairs are tuples within a list

```
Prelude> students = [("Stanley", 3.56), ("Ann", 4.0),
                     ("Bill", 2.95)]

Prelude> :type students
students :: Fractional b => [([Char], b)]

Prelude> (name, gpa) = head students

Prelude> name
"Stanley"

Prelude> gpa
3.56
```

Built-in Functions to Build A-Lists

`zip` – turns a list of keys and a list of values into an association list

`unzip` – turns an association list into a tuple of a list of keys and a list of values

```
Prelude> :type zip
zip :: [a] -> [b] -> [(a, b)]

Prelude> :type unzip
unzip :: [(a, b)] -> ([a], [b])
```

Defining and Using `zip`

```
zip :: [a] -> [b] -> [(a, b)]
zip [] [] = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

```
Prelude> students = zip ["Stan", "Ann", "Bill"]
                        [3.56, 4.00, 2.95]

Prelude> students
[("Stan",3.56),("Ann",4.0),("Bill",2.95)]
```

Built-in Function to Access Items

`lookup` – returns the value, of the form **`Just <value>`**, at a given key, if present, or **`Nothing`** otherwise

```
Prelude> students
[("Stan",3.56),("Ann",4.0),("Bill",2.95)]

Prelude> lookup "Bill" students
Just 2.95

Prelude> lookup "Ken" students
Nothing

Prelude> :type lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Maybe, or Maybe Not

`Maybe` is a ***union type***, meaning that it can be either of two values:

1. **`Nothing`**, or
2. **`Just a`**, where **`a`** is a value of any type

```
data Maybe a  = Nothing | Just a
                deriving (Eq, Show)
```

```
lookup:: Eq a => a -> [(a, b)] -> Maybe b
lookup key aList = helper aList where
    helper :: [(a, b)] -> Maybe b
    helper [] = Nothing
    helper ((k, v):restOfPairs)
       | key == k = Just v
       | otherwise = helper restOfPairs
```

Why is there anything at all, rather than nothing?

```
Prelude> let name1 = Just "Ken"   -- Construct optional values

Prelude> let name2 = Nothing

Prelude> :type name1              -- Examine their types
name1 :: Maybe [Char]

Prelude> :type name2
name2 :: Maybe a

Prelude> let Just name = name1    -- Extract the data

Prelude> name
"Ken"
```

Using **maybe** with **Maybe** values

Syntax:

maybe <a default value> <a function> <a Maybe value>

```
Prelude> :type maybe
maybe :: b -> (a -> b) -> Maybe a -> b

Prelude> maybe "" id name1     -- Return the data or a default
"Ken"

Prelude> maybe "" id name2
""
```

If the third argument is **Nothing**, the default value is returned.

Otherwise, the function is applied to the data within the **Maybe** value, and its result is returned.

id is a built-in identity function

From the **`Doctor`** Program

Python version:

```python
def changePerson(sentence):
    newlist = map(lambda word: replacements.get(word, word),
                  sentence.split())
    return " ".join(newlist)
```

Haskell version:

```haskell
changePerson :: String -> String
changePerson sentence = unwords (map myLookup (words sentence))
    where
    myLookup :: String -> String
    myLookup word =
        maybe word id (lookup (makeUppercase word) replacements)
```

For next time

Introduction to higher-order functions