CS119 – Module 9: Lists

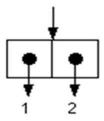
Purpose: The list is the basic data structure in Haskell and in computer science in general. Therefore, we will work on a larger project that centers on this important data structure.

Knowledge: This module will help you become familiar with the following content knowledge:

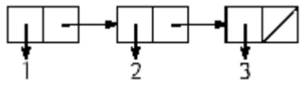
- Manipulation of lists

Activity: With your group perform the following tasks and answer the questions. You will need to copy the lab9 directory and work in `Example9.hs`. You will be reporting your answers back to the class in 1 hour.

1. There is an operator in Haskell called "cons" and denoted by a colon ":" that creates a pair of values in computer memory. Visually you can think of the pair `(1:2)` as

   We can create a list made up of these pairs which will look like:

   Explain why the expression `(1:(2:(3:[])))` represents the list above.

   A shorthand notation for the list `(1:(2:(3:[])))` is `[1,2,3]`.

2. An important feature of Haskell is that we can use a cons pair as a pattern when defining a function. We can define a couple of functions directly into REPL to test them out. Type in REPL:

   ```
   let f (x:xs) = x
   let g (x:xs) = xs
   ```

   Predict the values of the following expressions and test them out to verify:

   ```
   f [1,2,3]
   g [1,2,3]
   4: (g [1,2,3])
   (f [1,2,3]) : (g (g [1,2,3]))
   ```

3. The type signature for our list `[1,2,3]` would be `[Int]` but we often want to write functions which will work on lists of any type. We can do this by using the signature `[a]` (or use any other type variable like `b`, `c`, etc).

   Take a look at this function:

   ```
   len :: [a] -> Int
   len [] = 0
   len (x:xs) = len xs + 1
   ```

   Predict the result of the following and test them out to verify:

   ```
   len [2,4,6]
   len []
   ```

4. The pattern `[]` indicates an empty list and the pattern `(x:xs)` indicates a list with the first value being denoted by the variable `x` and everything but the first element denoted by the variable `xs`.

   What pattern would you use for a list containing one element?

   Complete the function below which returns the last element in a non-empty list.

   ```
   lastThing :: [a] -> a
   lastThing ________= ____________________ -- the case for a list with one element
   lastThing (x:xs) = ____________________ -- the case for a list with more than one element
   ```

   Use example in REPL:
   ```
   > lastThing [1,2,3,4]
   4
   ```

5. Suppose we have a list of pairs like: `[(1,"one"), (2,"two"),(3,"three"),(4,"four")]`. We could view this as an "association list", where we are associating a String with each Integer value. A useful function, therefore, would be a lookup function with which we give an Integer and it returns the associated String.

   Since we know that we are dealing with a list of pairs, we can use the pattern `((x,y): xs)` to indicate that we have a non-empty list whose first value is the pair `(x,y)` , with `x` and `y` as variables for the first and second entry in the pair, and the rest of the list is denoted by the variable `xs`. In the example above, `x` would have the value 1, `y` would have the value "one", and `xs` would be the list `[(2,"two"),(3,"three"),(4,"four")]`.

Complete the function `lookItUp` which returns the empty string if the value is not found, and returns the second item in the pair if the value is found:

```
lookItUp :: Integer -> [(Integer,String)] -> String
lookItUp n [] = ""
looiItUp n ((x,y) : xs) =
       if n == x then
            ___________________
       else
             ___________________
```

Use example in REPL:

```
> let a1 = [(1,"one"), (2,"two"),(3,"three"),(4,"four")]
> lookItUp 3 a1
"three"
> lookItUp 5 a1
""
```

6. Suppose we have a list of Strings and we know all the String values represent numbers like ”12” or ”-5”. Also suppose the we want to sum up the numbers that these String values represent. One way to do this would be to use the `map` function which is the list equivalent of `every` that we used with sentences. It applies a function to each element of the list.

   The function `read` can be used to take a String and convert it to an Integer:

```
stringToInt :: String -> Integer
stringToInt s = read s

> stringToInt "12"
 12
> stringToInt "-5"
-5
```

   Try using `map` to change all the Strings to their Integer values:

```
map stringToInt ["12","3","-5"]
```

7. We could use the `foldl` function, which is the list equivalent of the `accumulate` function that we used with sentences, to add up all the Integer values in a list:

   ```
   > foldl (+) 0 [12,3,-5]
   10
   ```

   Combine `foldl` and `map` to define a function which sums the Strings in a list:

   ```
   sumStr :: [String] -> Integer
   sumStr x = ____________________
   ```

   Use example REPL:
   ```
   > sumStr ["12", "3", "-5"]
   10
   ```

8. What if we never use stringToInt again and just want to define it on the fly for our one time use. Haskell allows us to define ”anonymous functions” which are functions that we use once and don’t name. For example,

   ```
   \x -> x + 1
   ```

   is an anonymous function which takes a parameter `x` and returns `x + 1`.

   We could use this in a map, for example:

   ```
   > map (\x->x+1) [1,2,3]
   [2,3,4]
   ```

   Rewrite `sumStr` so that it uses an anonymous function instead of using `stringToInt`.

Complete the following assignments to be submitted for grading. Each should be done individually but you can consult with a classmate to discuss your strategies or if you get an error message that you do not understand.

Complete the following assignments by writing functions in `Movies.hs` file in the `lab9` directory. You are going to complete the code for a Movie Query System in which a user can get information from movies like title, director, actor, and year made. To access the information we will allow users to be able to ask questions (in regular English sentences!) to the system and it will look up the answer. Such a feature is called a *natural language query system*.

The movie database will be stored as a list of movie records and we will need a way of searching through this list in various ways. We will also need a way of programming the search using patterns on the user's queries. We will start by defining the types for building a movie record:

```
type Title = String
type Director = String
type Year = Int
type Actor = String
type Actors = [Actor]

type Movie = (Title, Director, Year, Actors)
```

Note that we have a list of actors and that the movie record is a *tuple* containing the information in a given order. Our movie database will be a list of type `Movie`. Here is an example database with just two movie records:

```
movieDB :: [Movie]
movieDB = [("Amarcord",
           "Federico Fellini",
           1974,
           ["Magali Noel", "Bruno Zanin", "Pupella Maggio", "Armando Drancia"]),
          ("The Godfather",
           "Francis Ford Coppola",
           1972,
           ["Marlon Brando", "Al Pacino","James Caan", "Robert Duvall","Diane Keaton"])
           ]
```

In order to use abstraction and not have to remember the order of our tuples, we will define functions to select information from a movie record:

```
movieTitle :: Movie -> Title
movieTitle (t,_,_,_) = t

movieDirector :: Movie -> Director
movieDirector (_,d,_,_) = d

movieYearMade :: Movie -> Year
movieYearMade (_,_,y,_) = y

movieActors :: Movie -> Actors
movieActors (_,_,_,a) = a
```

Assignment 1:
We can use the `filter` higher order function to search for movies with certain properties. For example, the following function will keep only those movies that were made in 1974:

```
moviesMadeIn1974 :: [Movie] -> [Movie]
moviesMadeIn1974 movieDB = filter f movieDB where
 f movie = (movieYearMade movie) == 1974
```

Write functions which will search for movies in a given year, by a given director, or containing a given actor:

```
moviesMadeInYear :: Year -> [Movie] -> [Movie]
moviesDirectedBy :: Director -> [Movie] -> [Movie]
moviesWithActor :: Actor -> [Movie] -> [Movie]
```

For `moviesWithActor`, you will want to check if a certain `String` is contained in the list of actors. The function `elem :: a -> [a] -> Bool` will do this.

Criteria for Success: You can test these by:

```
> moviesMadeInYear 1974 movieDB
> moviesDirectedBy "Francis Ford Coppola" movieDB
> moviesWithActor "Al Pacino" movieDB
```

Assignment 2:
Rather than getting a list of entire movie records from our searches, we would prefer to get just the list of titles. Write the function `titlesOfMoviesSatisfying` which will return a list of titles satisfying a given predicate.

```
titlesOfMoviesSatisfying :: (Movie->Bool) -> [Movie] -> [Title]
```

For example, the following will return the titles of movies made in 1974:

```
> let f m = (movieYearMade m) == 1974
> titlesOfMoviesSatisfying f movieDB
```

Hint: Use the higher order function `map`.

Criteria for Success: You get a list of just the titles rather than the entire movie records.

Assignment 3:
We might want some attribute other than the title when we are searching. Write the function `moviesSatisfying` which returns a list of any movie attribute like title, director, actors, which satisfy the given predicate.

```
moviesSatisfying :: (Movie->Bool) -> (Movie->a)-> [Movie] -> [a]
```

For example, the following will return the directors of movies made in 1974:

```
> let f m = (movieYearMade m) == 1974
> moviesSatisfying f movieDirector movieDB
```

Important: I provided a "stub" (a function which has not been completely written) for this function already to allow the given code to compile without errors. You will want to replace the body of the provided function with your own for this assignment.

Criteria for Success: You should be able to print the title, director, actors from movies made in 1974 just by using this function with different parameters.

Now that we have a way to search through the database, we need to work on the natural language interface. We will start with a function called `queryLoop` which repeatedly reads and responds to the user's questions. To accomplish this, we will create a list of *pattern/action pairs* in which the pattern describes the type of question the user is asking (like "Who is the director of ..." or "What movies were made between _ and _") and the action will be a function for performing the search. I have written the `queryLoop` function for you already. It uses some code that we have not learned yet (but will!). Basically it inputs a line of text repeatedly and calls the function `answerByPattern` to process the user's query.

All the real work is handled by `answerByPattern`. This function takes as arguments the string that the user entered as well as the list of pattern/action pairs. This function will try to match the query with each of the patterns in the list of pattern/action pairs. If it finds a match, it then executes the action associated with that pattern, using the words in the query that fill in the blanks in the pattern as arguments. For example, if the pattern is "Who is the director of ..." and the query is "Who is the director of The Godfather", then the pattern and query match and the associated action function will be executed with the string "The Godfather" sent in as an argument. This function has also been provided but you will need to supply the `matches` and `substInMatch` functions which we will describe shortly.

Let's look at the *pattern/action pairs*. The patterns will contain *wild cards* which will stand for parts of the pattern which will be matched in the query. The wild card "..." will stand for a blank that will be filled in by one or more words matching the rest of the query. The pattern will therefore be a string which contains wild cards. The action will be a function using the `moviesSatisfying` function that you have written.

In Haskell, we can define an un-named or anonymous function by using something like `\x->x+1`. The `\x` indicates that the function takes one argument, x, and then the body of the function appears after the `->`. So

```
moviesSatisfying f movieTitle movieDB where
          f m = (movieYearMade m) == 1974
```

could be rewritten by substituting an anonymous function in place of `f`:

```
moviesSatisfying (\m->(movieYearMade m) == 1974) movieTitle movieDB
```

We will use anonymous functions to define the actions. These functions will be of type `[String] -> [String]`. The functions take a list of strings which are the parts of the query matching the wildcards, and return a list of strings giving the possible multiple answers. The type of the *pattern/action pair* will therefore be `(String, [String] -> [String])`.

Here is an example of a pattern/action pair for looking for the director of a movie with a given title:

```
("Who is the director of ...",
  (\[title] -> moviesSatisfying (\m->title == (movieTitle m))
                                movieDirector
                                moveDB))
```

So if we match that pattern, we have a title matching the "..." wild card. This title is the argument to an anonymous function which performs the `moviesSatisfying` search using the filter `(\m-> title == (movieTitle m))`, to find all movies, `m`, with that given title, and then maps the function `movieDirector` to extract the director's name from that movie.

To make the `answerByPattern` function work we need to write `matches` and `substInMatch`. The function `matches` is of type `[String]->[String]->Bool`. It takes a list of words making up the pattern and a list of words making up the query and returns whether or not the query matches the pattern. Clearly a word in the pattern will have to exactly match the corresponding word in the query. The wild card "..." will match all the remaining words in the query. Here is the code for matches:

```
matches :: [String] -> [String] -> Bool
matches [][] = True
matches _ [] = False
matches [] _ = False
matches ("..." : ps) q = True
matches (p : ps) (q : qs) = p == q && matches ps qs
```

Assignment 4:
The function `substInMatch` is similar to `matches` in that it takes two lists of the pattern and the query. However, it is only called when we already know that the two lists match. It returns a list of substitutions for the wild cards in the pattern that will make it match the query. For example,

```
> substInMatch ["foo","..."] ["foo","bar","baz"]
[["bar","baz"]]
```

I stub has been provided for this function. Complete the stub so this function works for patterns containing the wildcard "...". That will mean that the function will return a list containing a single list. We will add additional wild cards later so that we may have multiple items in the list later on.

```
substInMatch :: [String]->[String]->[[String]]
```

Hint: Use pattern matching like the function `matches`. Remember that since we already know that the pattern and query match for this function, you don't need the pattern for when they don't match.

Criteria for Success: Your function should work for the example above.
You should also now be able to test out the query system with our one pattern in the `patternActionList`:

```
> queryLoop
Who is the director of The Godfather

Francis Ford Coppola
```

Assignment 5:
Add another pattern of the form "Who acted in ..." to the `patternActionList` with its corresponding action.

Hint: Since the function `movieActors` returns a `[String]` you need to use the function `listToString` to convert this to a single `String`.
Therefore use `(listToString.movieActors)` as the selector. The "." here is function composition. It performs `movieActors` and then performs `listToString` on the result.

Criteria for Success: Enter `queryLoop` and then ask a query that matches this new pattern.

Now that our program recognizes simple patterns, we can start adding more complicated ones. The next pattern is typified by the following queries:

```
What movies were made in 1974
What movie was made in 1972
```

The pattern for this can be written as

```
"What ( movie movies ) ( was were ) made in _"
```

We have extended our pattern language in two ways. The "_" wild card matches exactly one word and it need not occur at the end of the pattern as is required for "...". The "( )" wild card provides a list of words that may match. So `( movie movies )` can match either the word "movie" or the word "movies".

Here is `matches` extended for the list wild card:

```
matches :: [String] -> [String] -> Bool
matches [][] = True
matches _ [] = False
matches [] _ = False
matches ("..." : ps) q = True
matches ("(" : ps) (q: qs) = elem q (makeList ps) && matches (restPattern ps) qs
matches (p : ps) (q : qs) = p == q && matches ps qs
```

The case `("(" : ps) (q: qs)` occurs when we are start a list. We use the function `makeList` to get the list of items up to the closing ")" and check to see if the first word in the query is contained in that list. The function `restPattern` returns the rest of the pattern after the ")" so that we can continue checking for a match after the list.

Assignment 6:
Extend `matches` to include the "_" wild card. Remember that this wild card matches a single word and that there can be more than one "_" in a pattern and need not occur at the end of the pattern.

Hint: Make sure that you add the new pattern in a correct location in the definition. The patterns are checked in the order that they appear and that is important.

Criteria for Success: You may test your modified function as follows. The function `words` breaks up the string into a list of words for us.

```
> matches (words "a _ c _ e") (words "a b c d e")
> matches (words "a _ c _ e") (words "a b c d f")
```

Assignment 7:
Extend `substInMatch` to include both the "( )" and the "_" wild cards. The function returns a list of substitutions, one for each wild card.

Criteria for Success: You may test your modified function as follows.

```
> substInMatch (words "a _ c _ e") (words "a b c d e")
> substInMatch (words "a ( b c ) e") (words "a b e")
```

We can test both our extensions with the new *pattern/action pair*:

```
("What ( movie movies ) ( was were ) made in _",
  (\[noun,verb,year] -> moviesSatisfying (\m->(stringToNum year) == (movieYearMade m))
                                         movieTitle
                                         movieDB))
```

Note that the action function takes a list of three values for the three substitutions into the wild cards. It only uses the last substitution value, however. The function `stringToNum` converts the year from a `String` to an `Int`.

Assignment 8:
Add a *pattern/action pair* for the pattern:

```
("What ( movie movies ) ( was were ) made between _ and _"
```

Criteria for Success: Test this in the query loop with a query that matches this pattern.

Assignment 9:
Add a *pattern/action pair* for the pattern:

```
("What ( movie movies ) ( was were ) made ( before after since ) _"
```

Criteria for Success: Test this in the query loop with a query that matches this pattern.

Assignment 10:
Modify the pattern/action pairs so that it is case insensitive and little words like ”a” and ”the” are ignored. That would mean that the titles ”The godfather” and ”Godfather” and ”A GODFATHER” would all match.

To accomplish this, you will want to write a function `modifyTitle` and then change the function for moviesSatisfying in the pattern/action pairs that compares the title that was entered with the title in the movie database to use the modified titles. Perform the following steps:

1. Write `modifyTitle` which will first convert all the characters to lower case This can be done by using the `toLower` function which takes a character and returns the lower case character. Obviously this must be done for all the characters in the title.
2. Then `modifyTitle` will break the title into a list of words using the `words` function and filter out all the little words.
3. Perform an equality test on the two modified titles within `moviesSatisfying` for the pattern/action pairs.

Criteria for Success: For the ”Who is the director ...” pattern the queries ”Who is the director of The Godfather”, ”Who is the director of Godfather”, ”Who is the director of godfather”, and ”Who is the director of A GODFATHER” all provide the correct answer.

Submit your `Movies.hs` file in Canvas for grading.