Declaratieve Talen

Haskell 2

1 Sequences

Define a type class `Sequence a` which consists of the functions `next` and `prev` to query the next and previous element in the sequence.

```
Main> prev 't'
's'

Main> next 'z'
*** Exception: no value after 'z'

Main> next (2 :: Integer)
3

Main> next True
False

Main> next False
True
```

Define the instances of `Sequence a` for `Integer`, `Char`, and `Bool`. The instance for `Char` should only consider the small letters from a to z.
Hint: have a look in the `Data.Char` module.

Make two subclasses of this type class: `LeftBoundedSequence a` and `RightBoundedSequence a`. The former has a function `firstElem` and the latter has a function `lastElem` to query the first and the last elements in the sequence.

```
Main> firstElem :: Char
'a'

Main> lastElem :: Bool
True
```

Define instances for these two classes for `Char` and `Bool`. Note that `Integer` does not have left or right bounds, so it shouldn't be made an instance of these classes.

2 HTML

The following code defines datatypes for representing structured (X)HTML markup.

```
data Attr = Attr String String
  deriving (Eq,Show)

data HtmlElement
  = HtmlString String
  | HtmlTag String [Attr] HtmlElements
  deriving (Eq, Show)

type HtmlElements = [HtmlElement]
```

A piece of HTML code is either plain text `HtmlString` or is a tagged node `HtmlTag` with attributes. In case of a node, other elements can be nested under it. The following HTML code

```
<a href="https://www.kuleuven.be/kuleuven/">
KU Leuven
</a>
```

is represented by the following value:

```
example :: HtmlElement
example = HtmlTag "a"
                  [Attr "href" "https://www.kuleuven.be/kuleuven/"]
                  [HtmlString "KU Leuven"]
```

We can group all types that can be rendered as HTML in a type class:

```
class HTML a where
  toHtml :: a -> HtmlElement
```

- Write an `HTML` instance that creates an anchor for the following datatype

  ```
  data Link = Link
               String  -- Link target.
               String  -- Text to show.
  ```

- Encode the following unordered HTML list as an `HtmlElement`:

  ```
  <ul>
  <li>Appels</li>
  <li>Bananas</li>
  <li>Oranges</li>
  </ul>
  ```

- Write an `HTML` instance for Haskell lists using unordered HTML lists.
- Model datatypes for an address book by defining a type `AddressBook` (and as many other data types you need). You should store at least the following information about your contacts:
  - First and last name.
  - A list of email addresses.
  - For each email address you should store if it is a work or private email address.
- Define an example address book `exampleAddressBook :: AddressBook` with at least two entries.
- Define `HTML` instances for the types of your address book. This exercise can become **very long** if you only construct values of `HtmlElement` directly. Try to abstract over recurring code.

NOTE: For the last three exercises there is no coverage from our testing framework (E-Systant), since they involve a user-defined data type, not known to the system. Hence, be extra careful when defining type `AddressBook`, the example `exampleAddressBook` and the instance of `AddressBook` for class `HTML`.

3 Last Re-sort

source: XKCD, `https://xkcd.com/1185/`

In this exercise we implement two comparison based sorting algorithms: *selection sort* and *quicksort*.

Selection Sort Given a list of elements to sort, selection sort repeatedly selects a minimal element in the list, removes it from the list and adds it to a new list. Implement this algorithm in the function. Think carefully on how to handle duplicate elements. `selectionsort :: Ord a => [a] -> [a]`
Hint: have a look in `Data.List`.
Example

```
> selectionsort [2,3,10,5,-3,2]
[-3,2,2,3,5,10]
> selectionsort (reverse [1..10])
[1,2,3,4,5,6,7,8,9,10]
```

Quicksort The well-known quicksort algorithm works by selecting an arbitrary element from the input list, the *pivot*. The list is then partitioned into two halves. Elements that are less than or equal to the pivot go in the left half, the other elements in the right half. Both halves are then sorted in turn. The sorted halves are then concatenated to obtain the final, sorted, list.

The first thing to implement is the partitioning step: this is accomplished by a function `partition :: (a -> Bool) -> [a] -> ([a],[a])`, such that `partition p xs` returns a tuple `(ys,zs)` where the `ys` contains all the elements of `xs` for which `p` is true, and `zs` contains all the elements of `ys` for which `p` is false. For the purpose of the exercise, implement the partitioning step three times: once using a fold, once using `filter` and once using list comprehensions. For example:

```
> partitionFold (< 0) [2,3,4,-1,6,-20,0]
([-1,-20],[2,3,4,6,0])
> partitionFilter odd [1,2,3,4,5,6]
([1,3,5],[2,4,6])
> partitionLC (not . even) [1,2,3,4,5,6]
([1,3,5],[2,4,6])
```

Implement the quicksort algorithm. In a non-empty list, choose the first element as the pivot. Do not forget the base case(s).
For example:

```
> quicksort [2,3,10,5,-3,2]
[-3,2,2,3,5,10]
> quicksort (reverse [1..10])
[1,2,3,4,5,6,7,8,9,10]
```

4 Arithmetic Expressions

The following `Exp` datatype encodes the abstract syntax for arithmetic expressions. Note that it is recursively defined: `Exp` occurs in the right-hand side of its own definition.

```
data Exp = Const Int
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
  deriving (Show, Eq)
```

4.1 Interpreter

- Write an interpreter `eval :: Exp -> Int` that evaluates arithmetic expressions.

Examples

```
Main> eval (Add (Mul (Const 2) (Const 4)) (Const 3))
11
Main> eval (Sub (Const 42) (Mul (Const 6) (Const 7)))
0
```

4.2 Compiler

Instead of evaluating an arithmetic expression directly, we can also compile it to a program of a simple stack machine and subsequently execute the program. We represent a program as a list of instructions. The instructions `IAdd`, `ISub`, `IMul` take the two topmost elements from the stack, perform the corresponding operation, and push the result onto the stack. The `IPush` instruction pushes the given value onto the stack. A stack is modelled by a list of integers.

```
data Inst = IPush Int | IAdd | ISub | IMul
  deriving (Show, Eq)

type Prog   = [Inst]
type Stack  = [Int]
```

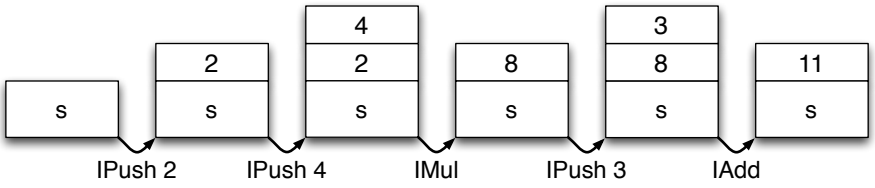
For example, the following arithmetic expression

```
Add (Mul (Const 2) (Const 4)) (Const 3)
```

is equivalent to the stack program

```
[IPush 2,IPush 4,IMul,IPush 3,IAdd]
```

The stack program leaves the result on top of the stack. The stack machine performs the following steps when executing the program on an initial stack `s`



- Write an execution function `execute :: Inst -> Stack -> Stack` that executes a single instruction.

  Since there are cases where the function can crash (e.g. a stack overflow in cases where we want to execute an `IAdd` instruction but the stack contains fewer than two elements), you can use the following exception-raising function where needed:

  ```
  runtimeError :: Stack
  runtimeError = error "Runtime error."
  ```

- Write a function `run :: Prog -> Stack -> Stack` that runs a whole program on a given initial stack.

- Write a compiler `compile :: Exp -> Prog` that compiles arithmetic expressions to stack machine programs. The compiled progam should leave the result of the computation as the top element on the stack. Make sure that your compiler uses a left-to-right evaluation order and that it produces results equivalent to the interpreter, i.e. the following identity holds

  ```
  forall (s :: Stack). run (compile e) s == (eval e) : s
  ```

Examples

```
Main> execute IAdd [4,5,6]
[9,6]

Main> execute ISub [4,5,6]
[1,6]

Main> execute (IPush 2) [4,5,6]
[2,4,5,6]

Main> run [IAdd, ISub] [4,5,6]
[-3]
```

```
Main> run [IAdd, ISub, IPush 7, IMul] [4,5,6,8]
[-21,8]

Main> run [IPush 1,IPush 2,IPush 3,IMul,ISub] []
[-5]

Main> compile (Sub (Const 1) (Mul (Const 2) (Const 3)))
[IPush 1,IPush 2,IPush 3,IMul,ISub]
```

5 Coin change

Imagine we have a cash register and an amount we are trying to reach by combining coins from this register. The coin change problem revolves around finding all possible combinations to reach a certain amount.

We can assume to have an infinite amount of each denomination available. We represent the denominations of integers (the amount of cents), and combine them in a list.

```
amountsEuro :: [Int]
amountsEuro = [1, 2, 5, 10, 20, 50, 100, 200]
```

Furthermore, we define the helper `changesEuro` which applies the (yet to be defined) `changes` function to `amountsEuro` giving a function of type `Int -> [[Int]]`.

```
changesEuro :: Int -> [[Int]]
changesEuro = changes amountsEuro
```

5.1 Calculating combinations

Define a function `changes :: [Int] -> Int ->[[Int]]` that takes a list of denominations `[Int]` and an amount to reach, and gives back a list of all (unique) combinations that add up to the given amount. You may assume the amount is not negative (≥ 0).

Hints

- How many base cases are there?
- What is the difference between the empty list `[]` and a list with an empty list as only element `[[]]`?

Examples

```
Main> changesEuro 0
[[]]
```

```
Main> changesEuro 1
[[1]]

Main> changesEuro 2
[[1,1],[2]]

Main> changesEuro 10
[[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,2],[1,1,1,1,1,1,2,2],[1,1,1,1,5],
[1,1,1,2,2,2],[1,1,1,2,5],[1,1,2,2,2,2],[1,2,2,5],[2,2,2,2,2],[5,5],[10]]
```

5.2 Order of denominations

Changing the order of the input denominations may change the *order* of the outputted combinations, but not the *amount* of combinations.

Let `amountsEuroRev` be `amountsEuro` reversed, and `changesEuroRev` be `changes` applied to `amountsEuroRev`.

```
amountsEuroRev :: [Int]
amountsEuroRev = reverse amountsEuro

changesEuroRev :: Int -> [[Int]]
changesEuroRev = changes amountsEuroRev
```

Make sure the following returns `True` for any input `i`:

```
checkReverse :: Int -> Bool
checkReverse i = (length $ changesEuro i) == (length $ changesEuroRev i)
```

Exam question

The following Word Wrap exercise is a question from a previous exam.

Some practicalities

- In the folder `TODO`, you'll find the files `MyHaskell.hs`, `MyHaskellTest.hs`, and `Testing.hs`.
  - The file `MyHaskell.hs` contains a template for your solution.
  - You can also **import extra functions and types**.
  - For each assignment, a number of functions and type classes have already been defined with corresponding type signatures. These functions and type signatures **may not be modified**. Replace all occurrences of `error "..."` with your implementation. You can add arguments in front of the equals sign, but, when possible, try to write the function *point-free*. It is of course also permitted to add extra helper functions.

  - You are allowed to use the slides of the lectures on Toledo, to use e-systant,
    hoogle (http://www.haskell.org/hoogle/ and the accompanying hackage documentation.
  - Even though you will develop a whole program in this exam, you can make most assignments separately. **Whenever you're stuck on an assignment, try the next**.
  - You can test your solution using `MyHaskellTest.hs`. Run the following command in the directory you put the three `.hs`-files in:

    ```
    runhaskell MyHaskellTest.hs
    ```

    N.B. the fact that all your tests pass doesn't mean that your program is completely correct, nor that you will get the maximum score.

- Hand in your solution (`MyHaskell.hs`) on Toledo.

Word Wrap

In this assignment we will split a text into lines with a maximal width such that the text will fit on a page with a certain width.

Take for example the text below.

> Leverage agile frameworks to provide a robust synopsis for high level overviews. Iterative approaches to corporate strategy foster collaborative thinking to further the overall value proposition. Organically grow the holistic world view of disruptive innovation via workplace diversity and empowerment.

This text, that was originally written on one line, has been split into lines by the word processor this assignment has been made with, namely LaTeX. LaTeX has a smart algorithm to split lines. It tries to keep the width of the lines as even as possible and it intelligently splits words. In this assignment we will keep it simple: we will not split words and we will use a simple *greedy* algorithm that places as many words as possible on the line until it is full, after which we go to the next line until we run out of words. When the original text contains a newline, this newline will occur at the same place in the split text (think of \\ in LaTeX).

If we run this algorithm with the text above using line width 50, we get the result below. **N.B.** the dashed first line and the bar at the end of each line were added to this figure to make it easier to see the line width. Your solution will **not** have to draw these dashes and bars.

```
----+----1----+----2----+----3----+----4----+----5|
Leverage agile frameworks to provide a robust     |
synopsis for high level overviews. Iterative      |
approaches to corporate strategy foster           |
collaborative thinking to further the overall     |
value proposition. Organically grow the holistic  |
world view of disruptive innovation via workplace |
diversity and empowerment.                        |
```

If we run the algorithm using line width 32, we get the result below:

```
----+----1----+----2----+----3--|
Leverage agile frameworks to    |
provide a robust synopsis for   |
high level overviews. Iterative |
approaches to corporate strategy|
foster collaborative thinking to|
further the overall value       |
proposition. Organically grow   |
the holistic world view of      |
disruptive innovation via       |
workplace diversity and         |
empowerment.                    |
```

Approach

The approach in this assignment is as follows: we split the original text (`String`) in a list of `LineItem`s, a new data type. We perform the splitting of lines by a number of transformations of lists of `LineItem`s Finally, we convert these lists of `LineItem`s back to a `String`.

Part 1: Line Items

Assignment 1.1: Define a data type named `LineItem`. A `LineItem` is either a space (`" "`), a newline (`"\n"`), or a word (a `String`). A word may not contain spaces or newlines, but it may contain punctuation. Multiple spaces in the input will result in multiple space `LineItem`s, the same is true for newlines. **Think carefully about which fields the constructors for a space and a newline should contain.**

See assignment 1.3 for some examples of `LineItem`s.

Assignment 1.2: Implement the functions `mkSpace :: LineItem`, `mkNewline :: LineItem`, and `mkWord :: String -> LineItem` that create the respective `LineItem`s.

Assignment 1.3: Make a `Show` instance for `LineItem`.

A space is shown as `" "`, a newline as `"\n"`, and a *word* as `"word"`. **N.B.** the double quotes are part of the `String`.

Hint: `show "foo" = "\"foo\""`.

Examples:

```
> mkSpace
" "
> mkNewline
"\n"
> mkWord "Hello"
"Hello"
> map show [mkSpace, mkNewline, mkWord "Hello"]
["\" \"","\"\\n\"","\"Hello\""]
```

Assignment 1.4: Implement the function `toLineItems :: String -> [LineItem]` which splits a `String` into a list of `LineItem`s.

Examples:

- The sentence "`See ya, John.`" consists of 5 `LineItem`s: `["See"," ","ya","," ","John."]`.
- The sentence "`Hint: read the\nassignment carefully!`" consists of 9 `LineItem`s:
  `["Hint:"," ","read"," ","the","\n","assignment"," ","carefully!"]`.
- The sentence "`! oops  \n\n `" consists of 8 `LineItem`s: `["!"," ","oops"," "," ","\n","\n"," "]`.

Assignment 1.5: Implement the function `fromLineItems :: [LineItem] -> String` which turns a list of `LineItem`s into a `String`.

Property: for each `String s`, it must be that `fromLineItems (toLineItems s) == s`.

Part 2: Word Wrap

Description of the algorithm:

- Place as many words on a line as possible until there is no more place left on the line. Start with a new line afterwards. Repeat this until there are no more words left.
- If there is a newline in the input, it should occur at the same place in the output.
- Multiple consecutive spaces will be replaced by a single space.
- Spaces at the start or end of a line are omitted.
- No line may be longer than the maximal line width, unless it contains a word longer than the line width. In this case, that word will be the only word on the line.

We split the algorithm in multiple simpler steps.

1. First we remove all spaces. Because we know there should be a space between every two words, we don't have to keep track of them explicitly. This makes the rest of the implementation easier. This also makes sure that multiple consecutive spaces will be replaced by a single space, just like required by the algorithm. Furthermore, this makes sure that no spaces will occur at the start and end of a line.
2. Next, we split the text into lines. Pay attention: these lines originate from the newlines that are already present in the text. We can then split each of those lines individually, without having to worry about preserving the newlines in the original text.
3. Afterwards, we put words longer than the maximal line width on their own lines. Later, during the actual splitting, we won't have to make sure no other words are placed on the same line before such a word.
4. Next, we will perform the actual splitting of the lines: we split a list of words into lines by putting as many words on a line as possible until there is no more place left on the line. While doing this, we remember to take the width of the space between each two words into account.
5. In the next step We restore the spaces between the words on each line.
6. Afterwards, we join the lines together by placing newlines between each two lines.

Assignment 2.1: Implement the function `removeSpaces :: [LineItem] -> [LineItem]` which removes all spaces from a list of `LineItem`s.

Examples:

```
> removeSpaces [mkWord "hello", mkSpace, mkWord "world", mkNewline, mkWord "Bye", mkSpace]
["hello","world","\n","Bye"]
> removeSpaces [mkWord "hi", mkSpace, mkSpace, mkNewline, mkSpace, mkNewline, mkSpace]
["hi","\n","\n"]
```

Assignment 2.2: Implement the function `splitInLines :: [LineItem] -> [[LineItem]]` which splits a list of `LineItem`s whenever a newline occurs. The result will no longer contain any newlines.

You may assume the input will not contain spaces.

Examples:

```
> splitInLines [mkWord "hi", mkNewline, mkWord "bye"]
[[mkWord "hi"], [mkWord "bye"]]
> splitInLines [mkNewline, mkWord "hi", mkNewline, mkNewline, mkWord "bye", mkNewline]
[[], [mkWord "hi"], [], [mkWord "bye"], []]
> splitInLines []
[[]]
> splitInLines [mkNewline]
[[], []]
> splitInLines [mkNewline, mkNewline]
[[], [], []]
> splitInLines [mkWord "foo", mkNewline]
[[mkWord "foo"], []]
```

Assignment 2.3: Implement the function `separateTooLongWords :: Int -> [LineItem] -> [[LineItem]]` which splits a list of `LineItem`s whenever a word is longer than the maximal line width. Each list of `LineItem`s represents a line, so this function splits one line into a list of lines. Whenever a word is too long, the line is split in three lines (lists): a line of the words coming before the too long word, a line with just the too long word, and a line with the words coming after the too long word. When no words come before the too long word, the first list is omitted. When no words come after the too long word, the third list is omitted. This process is repeated for each too long word.

You may assume the input will only contain words, no spaces or newlines.

Examples:

```
> separateTooLongWords 6 [mkWord "look", mkWord "a", mkWord "brontosaurus",
                          mkWord "over", mkWord "there"]
[["look","a"],["brontosaurus"],["over","there"]]
> separateTooLongWords 3 [mkWord "Yuuuuge", mkWord "amazing"]
[["Yuuuuge"],["amazing"]]
> separateTooLongWords 3 [mkWord "Banana"]
[["Banana"]]
```

```
> separateTooLongWords 100 [mkWord "Banana"]
[["Banana"]]
```

Assignment 2.4: Implement the function `wrap :: Int -> [LineItem] -> [[LineItem]]` which splits a list of `LineItem`s every time the maximal line width is reached. Each list of `LineItem`s represents a line, so this function splits one line into a list of lines. Recall the algorithm: *place as many words on a line as possible until there is no more place left on the line. Start with a new line afterwards.* Do not forget to account for the space between each two words. Also do not forget that some words may be longer than the maximal line width.

You may assume the input will only contain words, no spaces or newlines.

Examples:

```
> wrap 7 [mkWord "foo", mkWord "bar", mkWord "qu", mkWord "u", mkWord "x", mkWord "banana"]
[["foo","bar"],["qu","u","x"],["banana"]]
> wrap 4 [mkWord "gyronef"]
[["gyronef"]]
```

Assignment 2.5: Implement the function `joinLineWithSpaces :: [LineItem] -> [LineItem]` which puts a space between each two words in a list of `LineItem`s.

You may assume the input will only contain words, no spaces or newlines.

Examples:

```
> joinLineWithSpaces [mkWord "so", mkWord "much", mkWord "space"]
["so"," ","much"," ","space"]
```

Assignment 2.6: Implement the function `joinLinesWithNewlines :: [[LineItem]] -> [LineItem]` which joins the given list of lines into one list of `LineItem`s where each line is separated by a newline.

You may assume the input will only contain words and spaces, no newlines.

Examples:

```
> joinLinesWithNewlines [[mkWord "hi", mkSpace, mkWord "there"],[mkWord "bye"]]
["hi"," ","there","\n","bye"]
```

Result The above mentioned functions are combined in the function `wordWrap`. This function is given, so you do not have to write it yourself. Using this function, it is easy to test whether you have implemented the algorithm correctly by trying the examples below.

Examples:

- ```
  > wordWrap 5 "a b c d e f g"
  "a b c\nd e f\ng"
  > wordWrap 4 "a b c d e f g"
  "a b\nc d\ne f\ng"
  > wordWrap 5 "a\nb c d e f g"
  "a\nb c d\ne f g"
  ```
  ```

```
> wordWrap 4 "\n food"
"\nfood"
> wordWrap 4 "a b c delta e f g"
"a b\nc\ndelta\ne f\ng"
> wordWrap 7 "foo  bar  "
"foo bar"
> wordWrap 20 "  foo  \n  \n  bar  "
"foo\n\nbar"
```

- See the two examples from the introduction (`putStrLn $ wordWrap 50 "..."` and
  `putStrLn $ wordWrap 32 "..."`).

Hint: using the test suite you can test all the examples above at once.

Part 3: Interactive Word Wrapping

Part 3 uses IO which has not been covered yet. Therefore, it is not part of this exercise, meaning that if you finished everything of parts 1 and 2, you are done with this exercise. Part 3 has been included in case you want to experiment with IO on your own.

Assignment 3.1: Implement the function `getLines :: IO String` which reads a text (`String`) that may contain newlines. The user enters the text line by line, where each line is terminated by a press of the Enter key. This continues until the user types in `STOP` (followed by Enter).
Hint: use `getLine` to read one line. Don't forget the newline following each line.

Examples: The line following `STOP` is the result of the call to `getLines`.

```
> getLines
Hello
STOP
"Hello\n"
> getLines
Hello, there

Bye now
STOP
"Hello, there \n\nBye now\n"
```

Assignment 3.2: Implement the function `interactiveWrapper :: IO ()` which asks the user a line width and a text, after which the text will be 'wrapped' using the line width. Read the text using the `getLines` function from the previous assignment. You may assume the user enters a valid number for the line width.

Examples:

```
> main
Please enter a line width: 4
Please enter a text to wrap:
a
b c d e f g
STOP
a
b c
d e
f g
```

6 Extra: Unification

In this exercise we implement the basic unification algorithm for Prolog terms. Recall that unification is a procedure that, given two terms tells you what terms to substitute for the variables in the terms, such that the two terms become equal. Such a substitution is called a *unifier*.

For example, a unifier of the terms `f(g(X))` and `f(Y)` is substitutes `g(X)` for `Y`.

Prolog Terms First we need to introduce a data type to represent Prolog terms. Terms are either functor terms of the form `f(t1,...,tn)`, i.e. a functor `f` and zero or more arguments, or a variable `X`. To simplify things, we assume that variables are ordered, and only refer to them by their rank (an `Int`).

```
type VarId = Int
data Term
  = F String [Term]   -- f(t1,...,tn)
  | Var VarId         -- Xn
  deriving (Eq,Ord)
```

A `Show` instance for `Term` which prints the corresponding Prolog term has been defined for you. Define a function `occurs n t` which returns true if and only if `Var n` occurs in the term `t`. This function will be useful later on.
For example,

```
> F "f" []
f
> Var 0
X0
> F "f" [F "g" [], F "h" [Var 1]]
f(g,h(X1))
> F "f2" [Var 2]
f2(X2)
> occurs 0 (F "f" [Var 0])
```

```
True
> occurs 1 (F "f" [Var 1])
```

Substitutions As mentioned previously, the result of a unification of two terms is a unifier, a substitution, such that when it is applied to the two the terms they become equal. We will represent a substitution in Haskell as a list of pairs of a `VarId` and a `Term`.

Define a function `applySubst` which applies a substitution to a term. Remember that a substitution leaves the variables for which it is not defined unchanged, and a substitution applied to a functor term is that same functor term, but with the substitution applied to its arguments.
For example,

```
> applySubst [(0,F "g" [])] (Var 0)
g
> applySubst [(0,F "g" [])] (Var 1)
X1
> applySubst [(0,F "g" [Var 1])] (F "f" [Var 0])
f(g(X1))
> applySubst [(0,F "g" [Var 1])] (F "f" [F "g" [Var 1]])
f(g(X1))
> applySubst [(0,F "g" []),(1,Var 2)] (F "f" [F "g" [Var 1], Var 0])
f(g(X2),g)
```

Also write a function `conc` that concatenates two substitutions. This is not just the list concatenation, you must *also* apply the substitution on the left to every term in the substitution on the right.
For example,

```
> conc [(0,F "f" [Var 1])] [(1,F "g" [Var 0]),(2,[Var 1])]
[(0,f(X1)),(1,g(f(X1))),(2,X1)]
> conc [] [(1, F "g" [Var 2])]
[(1,g(X2))]
```

Unification Suppose we have a list of equations between terms, we want to obtain a substitution that unifies *all* those equations. This is achieved by the following algorithm:

- If the list is empty, return the empty substitution.
- If the first equation is of the form `X = X` where `X` is a variable, then try to unify the remaining equations.
- If the first equation is of the form `X = t` where `X` is a variable *and* `X` does not occur in `t`, then
  1. Substitute `t` for all occurences of `X` in the remaining equations.
  2. Try to unify the remaining equations.

3. Concatenate the substitution obtained in the previous step with the substitution `[(X,t)]`.

- If the first equation is of the form `t = X` where `X` is a variable, then swap the position of `X` and `t` and try to unify the equations again.
- If the first equation is of the form `f(t1,...,tn) = g(u1,...,um)`, where `f` = `g` *and* $n = m$, then
  1. Add all equations of the form `t1 = u1`, ..., `tn` = `un` to the list of equations.
  2. Try to unify this new set of equations.
- Otherwise, we fail to find a unifier.

In Haskell, we represent the list of equations as a list of pairs of terms. Because the unification algorithm can fail, the result is a `Maybe Substitution`.

Complete the function `unify` which implements the algorithm above.

To unify two terms, we must simply pass the appropriate list to `unify`:

```
unify1 :: Term -> Term -> Maybe Substitution
unify1 t1 t2 = unify [(t1,t2)]
```

Some examples:

```
> unify1 (Var 0) (F "f" [])
Just [(0,f)]
> unify1 (F "f" []) (Var 0)
Just [(0,f)]
> unify1 (F "f" [F "g" [Var 1]]) (F "f" [Var 0])
Just [(0,g(X1))]
> unify1 (F "f" [Var 0]) (Var 0)
Nothing
> unify1 (F "g" []) (F "f" [])
Nothing
> unify1 (F "f" [Var 0, Var1], F "f" [Var 0])
Nothing
> unify [(F "f" [Var 1],Var 0),(F "g" [Var 1],F "g" [F "a" []])]
Just [(1,a),(0,f(a))]
```