G6021: Comparative Programming

Exercise Sheet 9

A: Logic Programming

We will implement a small fragment of a logic programming language. First we explain the language. Programs are collections of:

- facts: things that are known to be true,
- and rules: ways to generate new facts.

In addition, we have a query which is a question about the facts that we have (the result of a query will be either True or False). We write facts and rules as shown in the following example:


```
likePasta.
likeRissoto.
likePizza.
likeItalianFood :- likePizza, likePasta, likeRissoto.
```


The first three lines are facts, the fourth is a rule. We read the rule as: someone likes Italian food if they like pizza, pasta and risotto. (i.e., we read `:-` as "if", and comma as "and".)

A query is a question about the facts. For example: `?likeRisotto` is true, because it is a fact. When we write queries, we put a question mark at the start. A more complicated query is `?likeRisotto,likePasta`. This is again true, because both facts are true, however, `?likeHaggis` is false, because there is no fact about this in our program.

An even more complicated query: `?likeItalianFood` is true. In this case, we have use the rule, and now we have a new query: `?likePizza, likePasta, likeRisotto`, and now we need to check each of these, etc. Computation is searching: either the fact is there, or we have to apply a rule and we get some new facts to check. When we have no more facts to check, then the query is true. If we cannot find a fact, then it is false.

Facts and rules are in fact the same, it's just that there is nothing else to check for a fact. Facts and rules are called Clauses (specifically Horn Clauses). In a Clause such as `likeItalianFood :- likePizza, likePasta, likeRisotto`, we call `likeItalianFood` the head of the Clause, and the list: `likePizza, likePasta, likeRisotto` is called the tail of the Clause. Note that for a fact, the tail is empty.

We can represent Clauses as a pair: the head and the tail of the Clause. Here is one way to represent the above example in Haskell:

```
("likeItalianFood",["likeRissoto","likePasta","likePizza"])
("likeRissoto",[])
("likePasta",[])
("likePizza",[])
```

Using this representation, we can write some functions to implement this form of computation.

1. Write a function to look-up a query in the list of clauses. If it is there, return the pair `(True, t)` where `t` are the new queries (the tail of the Clause, or the empty list if it was a fact). If the goal is not in the list of clauses, then return `(False,[])`.

Answer:

```
lookup h [] = (False,[])
lookup h ((p,t):r) = if h==p then (True,t) else lookup h r
```

2. Write a function called `solution` that takes a list of queries and a list of clauses, and returns `True` if the queries are satisfied by the program clauses, `False` otherwise.

   Answer:

```
solution [] prog = True
solution (h:t) prog =
 let (a,b) = (lookup h prog) in
   if a then solution (b++t) prog else False
```

3. Test the above example with:

```
solution ["likeItalianFood"]
[ ("likeItalianFood",["likeRissoto","likePasta","likePizza"]),
  ("likeRissoto",[]), ("likePasta",[]), ("likePizza",[])]
```

   Make up some other example programs and queries to test your functions.

B: Input/Output

There are a number of problems with user interaction in Haskell programs:

- Reading from the keyboard and writing to the screen are actions with side effects,
- Side effects change the state of the environment the function runs in and cannot be undone,
- Pure functional programming has no side effects.

However, there are some solutions to this, that allow impure functions to be introduced in a controlled way (through types). Without going into the details, the following examples illustrate how things work.

The standard library provides a number of actions, including:

- `getChar :: IO Char` reads a char from the keyboard, echoes it to the screen and returns the read value. `getChar` has type `IO Char`, which means that the return type is a `Char`, but some side effects might have taken place.
- `putChar :: Char -> IO ()` writes a char to the screen and returns `()`. (`()` is analogous to `void` in Java.)
- `return :: a -> IO a` returns its argument with no interaction.

A sequence of actions can be combined into a single composite action using the keyword `do`, and we use the operator `<-` to bind the result of these kinds of functions:

```
getTwo :: IO (Char, Char)
getTwo = do x <- getChar
            y <- getChar
            return (x,y)
```

Here is another example, that reads a String from the Keyboard:

```
getChars     :: IO String
getChars     =  do c <- getChar
                   if c == '\n'
                       then return ""
                       else do l <- getChars
                               return (c:l)
```

Finally, here is an example where we write a String to the Screen:

```
putString :: String -> IO ()
putString [] = return ()
putString (x:xs) = do putChar x
                      putStr xs
putStringLn :: String -> IO ()
putStringLn xs = do putString xs
                    putChar '\n'
```

1. Test the above functions to make sure you understand how these work.
2. Write a function `len :: IO ()` that will prompt the user to enter a string, will read the string, and then calculate and print the length of the string.

Answer:

```
len :: IO ()
len = do putStr "Enter a string: "
         xs <- getLine
         putStr "The string has "
         putStr (show (length xs))
         putStrLn " characters"
```