Declaration on Plagiarism

Assignment Submission Form

Name(s): Killian Connolly
Programme: Computer Applications and Software Engineering
Module Code: CA341
Assignment Title: Comparing Functional and Logic Programming
Submission Date: 16th December 2019
Module Coordinator: David Sinclair

Name(s): ___KillIan Connolly____________ Date: _24/11/2019________

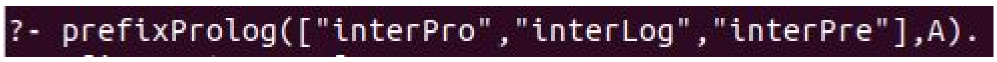
Introduction

I have chosen to do my functional style program in Haskell and my logic style program in Prolog.

Logic Programming

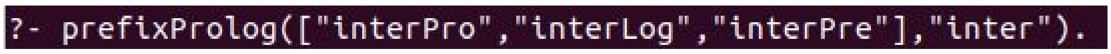
Logic programming style paradigm consists of logic expressions indicating rules and facts involved in your program. Theses facts and rules are called predicates and consists of clauses, which are the individual definition of the fact. First-order logic consists of these facts and relations relating to certain objects. This paradigm also incorporates Horn Clauses, which is a clause that has a positive literal or value. All these concepts is associated to Prolog which is leading programming language in logic programming paradigm.

My logic program starts of by setting prolog flags. Meaning, anything contained in double quotes, is converted from ASCII code to the letters themselves. The program is used by calling on the “prefixProlog” function giving the arguments of a list of strings and a literal.



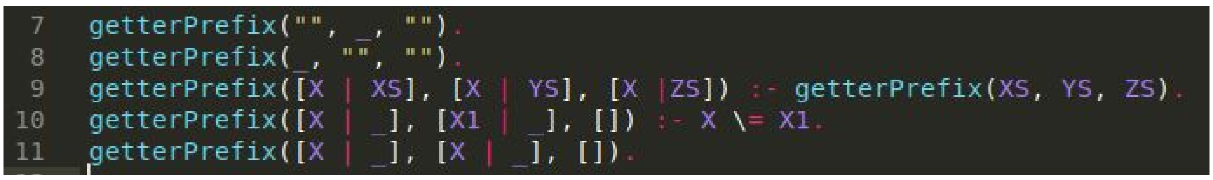
```
?- prefixProlog(["interPro","interLog","interPre"],A).
```

Or



```
?- prefixProlog(["interPro","interLog","interPre"],"inter").
```

The concept for my program gets the first two elements of the list and passes the those elements onto another clause along with the prefix called “getterPrefix”. Once the function is complete, the rest of the elements is called.

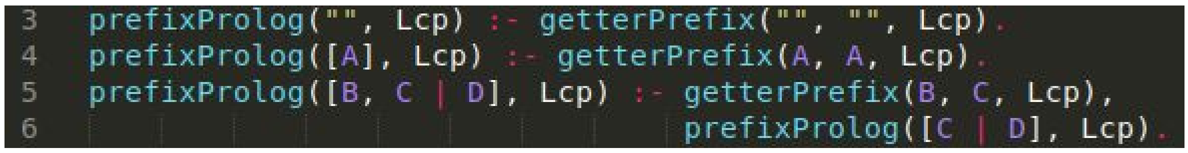


```
7   getterPrefix("", _, "").
8   getterPrefix(_, "", "").
9   getterPrefix([X | XS], [X | YS], [X |ZS]) :- getterPrefix(XS, YS, ZS).
10  getterPrefix([X | _], [X1 | _], []) :- X \= X1.
11  getterPrefix([X | _], [X | _], []).
```

getterPrefix

Line nine is the main rule, it takes each letter off the two elements and calls recurse until a base case is hit. The base cases include line seven and eight which is if one of the words is shorter than the other. Line ten, which is if the two letters are not the same. Finally line eleven, which is for when the longest common prefix is not assigned yet. Once one of the base cases are hit, the clause is backtracked and the appropriate letters are added to the front of the longest common prefix literal. During the next iteration if a letter are not the same as the longest common prefix literal, the program backtracks until the last time it was true, takes off a letter and tries recurring again.

Once my program comes to an end the base cases are used within the “prefixProlog” clause.



```
3   prefixProlog("", Lcp) :- getterPrefix("", "", Lcp).
4   prefixProlog([A], Lcp) :- getterPrefix(A, A, Lcp).
5   prefixProlog([B, C | D], Lcp) :- getterPrefix(B, C, Lcp),
6                                     prefixProlog([C | D], Lcp).
```

prefixProlog

Line three is only implemented if the clause was given an empty array. Line four is the base case for the whole clause once the last element of the array is hit, give “getterPrefix” two of the same element. This is enforced because without it, if the clause was given an array of one element the clause would return false.

Finally, when the remaining iteration of the last element is complete within the “getterPrefix”, line six is then backtracked and the program is complete.

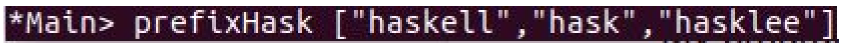
Functional Programming

Functional programming style consists of functions that can pass around values and those functions can be values themselves. These functions are generally deterministic, hence, making them referentially transparent. Meaning that the given input is never changed throughout the program.

In functional programming commands or updating variables assignments is not common. In order to combat this, lazy evaluations, first-class functions and higher order functions are utilised which is treating functions as values.

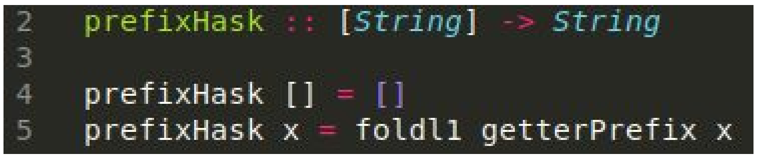
My program doesn’t have some traits of functional programming like generic abstraction types or pattern matching since I didn't need them. However, the powerful concepts higher order functions, first class functions and recursion are utilized.

My functional program is initialized by calling upon the prefixHask function as shown:



```
*Main> prefixHask ["haskell","hask","hasklee"]
```

My Functional program takes advantage of a higher-order function within the haskell library, foldl1, when passing the list onto the “getterPrefix” function.



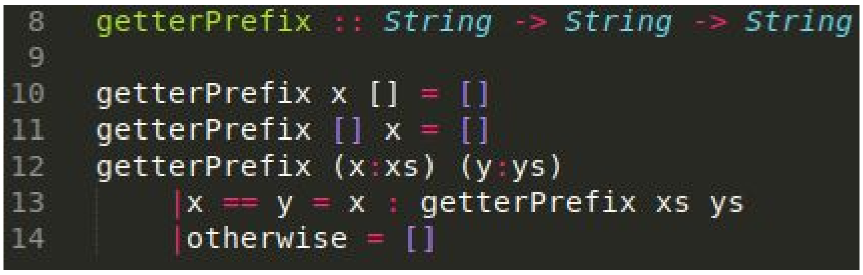
```haskell
prefixHask :: [String] -> String

prefixHask [] = []
prefixHask x = foldl1 getterPrefix x
```

prefixHask

This function takes two elements from the second parameter, respectively a list of strings, “x”, and passes it to the first parameter which is a function. This parameter can be arithmetic operations, however in my case, it is the “getterPrefix” function. The output of that operation is executed recursively with the next element of list. This function also catches the unlikely occurrence that if an empty list was given.

Function “getterPrefix”, obtains the longest common prefix of the two elements.



```haskell
8   getterPrefix :: String -> String -> String
9
10  getterPrefix x [] = []
11  getterPrefix [] x = []
12  getterPrefix (x:xs) (y:ys)
13      |x == y = x : getterPrefix xs ys
14      |otherwise = []
```

getterPrefix

The two elements are split at the head on line twelve, “x” and “y”. At line thirteen, the heads of each letter of the two elements is compared to see if they are identical, given that they are, one of the letters is then added to a string and the rest of the words, ”xs” and “ys” are then recurred through the same function. Once the letters are not the same, the base case is initialised and the function ends returning the string of the longest common prefix of the two words. Then the foldl1 is called again on the next argument using the output it obtained from the first iteration. The foldl1 function gathers all the results from each iteration and adds it to a list. Once the function has reached the end of the initial list, the last element is outputted onto the screen, resulting in the longest common prefix.

Parametric Polymorphism, a key concept in functional programming, could also be applied to my program, nonetheless strings was our specified data type. Additionally, strings of integers can operate within my program.

Efficiency

Measuring the efficiency of code is tough to do, there is a lot of contributing factors that apply to efficiency such as computation time, memory/cost, storage, operations. I am just going to explain the execution time of both of my implementations along with the complexity of the program.

Logic Program Implementation

Prologs memory management consists of global stack, local stack and its trail. The big O notation of my logic implementation would be O(n^2). Giving my program one hundred words, runtime is almost instant since CPU usage is or almost one hundred percent. Pure logic programming is relatively quick while an imperative implementation is considered slow. My program also shows to have over a half a million logical inferences per second when running my code with one hundred words. Backtracking is a powerful concept in logic programming, however, it comes with a disadvantage. It is quite expensive. My program uses quite a lot of backtracking, nevertheless, cuts can be implemented to reduce this computationally expensive concept.

Functional Program Implementation

Haskell uses automated dynamic memory allocation, hence, requiring a garbage collector. The big O notation of my functional implementation would also be O(n^2). One of my operations would be for iterating through the line, using the higher-order function foldl1. The second operation is the function that compares two elements. The big O notation for this would be O(n*m), however, since for every n operations(elements in the list), you, at most, do m operations (length of the word). Making the big O notation n squared. Tested with approximately 100 words, my program runs in 0.01 seconds and using around 120 thousand bytes. Also CPU usage is or almost one hundred percent.

Comparison

My implementations of both functional and logic programming paradigms are essentially done using the same methods. Firstly, obtaining the first two elements of the list, comparing to find the longest common prefix, and finally, iterating through the list constantly checking the longest common prefix.

One of the principal differences between these two paradigms is functional programming offers functions while logic provide predicates. Both have their similarities, nonetheless, the difference is predicates express relationships with object and functions can manipulate objects given an object.

My functional program is shorter than my logical because of the build in higher order function, foldl1. In my logic program, I had to implement that function into the main predicate, “prefixProlog”. Even though both languages contain built in functions, only my functional program utilises one of them.

Functional programming supports recursion, while logic supports recursion and backtracking. Backtracking is a key concept in Prolog and my implementation. In my logic program, once the current longest common prefix is incorrect, backtracking occurs until the last time the prefix was changed. Once it gets to this point, it backtracks once more where another letter is removed from the back. Finally that prefix is checked with the current word. In my functional program, my string is creating and appended onto the result of the recursive function and so on. No backtracking is used. Backtracking is used to obtain the output for my logic program.

Aforementioned, the efficiency of backtracking is quite a disadvantage, this may be why the more efficient and faster programming paradigm is my functional program. Since logic goes through every possibility the local and global stacks will start to get more use causing the cost of the program to increase.

Both functional and logic programming paradigms make use of stacks for storage. Prolog uses three stacks, local, global and trail. For Haskell, since all data types are immutable this creates a lot of memory garbage, especially when recurring. Within the garbage collection there is a stack. Hence, Haskell cost can be quite high. Both implementations memory consumptions are equally expensive.

Pure logic programming is referentially transparent along with functional programming. This means that an expression can be replaced with the result.

Prolog data is immutable by design, similarly with Haskell. However within Haskell there are packages that you can import to make data mutable but this is not an aspect of functional programming. Both of my implementations uses the data type, list, which are immutable in both of these paradigms along with every other data type. Meaning once a value is bound to a name you cannot change it.

For my logic program, I had set the flags so my program understands what is inside of the double quotation marks. This is because the program will automatically assume you would like the program to comprehend ASCII. This is Prolog working logically rather than literally.

Within my functional program, having a declaration in my program is not needed, however is it most commonly used. Since the requirements asked for String the declarations parameters require such data types. Haskell has this option along with polymorphic parameters. On the other hand within Prolog, each argument is an arbitrary Prolog term.

In conclusion, while functional programming provide many concepts such as functions, parametric polymorphism, data abstractions and lazy evaluation, logic programming provide very powerful notations, such as backtracking.

Reference:

https://wiki.haskell.org/Haskell

https://www.swi-prolog.org/