Declaration on Plagiarism

Assignment Submission Form

This form must be filled in and completed by the student(s) submitting an assignment

Name(s): Ethan Sharkey

Programme: Computer Applications and Software Engineering (CASE3)

Module Code: CA341

Assignment Title: Comparing Logic Programming and Functional Programming

Submission Date: 15/12/19

Module Coordinator: David Sinclair

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

I/We have read and understood the referencing guidelines found at

http://www.dcu.ie/info/regulations/plagiarism.shtml ,
https://www4.dcu.ie/students/az/plagiarism

and/or recommended in the assignment guidelines.

Name(s): Ethan Sharkey_____Date: 15/12/19_____

Comparing Logic Programming and Functional Programming

Name: Ethan Sharkey
Student No.: 17355756

Functional Programming:

For the functional programming aspect of this assignment, I chose to do it in Haskell as I currently have a module based purely on the Haskell language. It is also the only functional language I have experience with. My objective was to be able to find a longest common prefix between a list of words. This was completed which is evident below with test cases.

```
ethan@Jessica-HP-Pavilion-x360:~/Documents/ThirdYear/CA341/LogicProgrammingVSFunctionalProgramming$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Prelude> :load longestCommonPrefix.hs
[1 of 1] Compiling Main             ( longestCommonPrefix.hs, interpreted )
Ok, modules loaded: Main.
*Main> longestPrefix(["interview", "interrupt", "integrate", "intermediate"])
"inte"
*Main> longestPrefix(["hello", "hell"])
"hell"
*Main> longestPrefix([""])
""
*Main> longestPrefix(["hi", "hill", "hilux"])
"hi"
*Main> longestPrefix(["hx", "hill", "hilux"])
"h"
*Main>
```

I first began by trying to compare each word letter by letter in the list. It did not take me long to find out Haskell has a built-in function called “foldl”, that assists me in doing this this. Doing more research on foldl, I found a specific “foldl”. I would need this form of “foldl” to complete this program the way I wanted when I began. This form of “foldl” was called “foldl1”. It is also built-in to Haskell.

What I began with:

longestCommonPrefix.hs

```haskell
longestPrefix :: (Eq a) => [[a]] -> [a]
longestPrefix = foldl1 longestCommonPrefix


longestCommonPrefix :: (Eq e) => [e] -> [e]

longestCommonPrefix [] = []

longestCommonPrefix (x : xs : xss)
    | x == xs = x : longestCommonPrefix xss
    | otherwise = []
```

After my research on “foldl1”, the program above is what concluded from it however, it did not compile. I soon realised that “foldl1” requires two lists. The purpose of this is so it can compare all the letters in the first and second word. Then the result of their prefix gets compared to the third word and the result from that computation gets compared to the fourth word and so on. This continues until it reaches the end of the list of words. Adding another list also required me to add another base case as either of the two lists could be empty.

The final implementation of my program in Haskell:

```haskell
--["interview", "interrupt", "integrate", "intermediate"]

longestPrefix :: (Eq words) => [[words]] -> [words]
longestPrefix = foldl1 longestCommonPrefix


longestCommonPrefix :: (Eq letter) => [letter] -> [letter] -> [letter]

longestCommonPrefix _ [] = []
longestCommonPrefix [] _ = []

longestCommonPrefix (x : xs) (y : ys)
    | x == y = x : longestCommonPrefix xs ys
    | otherwise = []
```

Above is my final implementation of my program in Haskell. This is the program used to test my test cases which can be found above.

Logic Programming:

With the logic programming aspect of this assignment, I completed it in Prolog. The reason I chose Prolog for the logic programming part of this assignment, was due to the fact that it is the only logic programming language I have had previous experience with. The objective was the same as the functional programming objective. It was completed with test cases proving its completion below.

```
?- [longestCommonPrefix].
true.

?- longestCommonPrefix(Prefix, ["interview", "interrupt", "integrate", "intermediate"]).
Prefix = inte.

?- longestCommonPrefix(Prefix, ["hello", "hell"]).
Prefix = hell.

?- longestCommonPrefix(Prefix, [""]).
Prefix = ''.

?- longestCommonPrefix(Prefix, ["hi", "hilux", "hill"]).
Prefix = hi.
```

When I first began working on this assignment, through little research and what I remember about Prolog, I quickly got a program to give me an answer. However, there were problems with the answer. The code below would give me one letter of the prefix at a time until it returned false. There is also an example of this below.

First Basic Program to Find the longest Common Prefix:

```
13  :- set_prolog_flag(double_quotes, chars).  % "abc" = [a,b,c]
14
15  prefix_of(Prefix, List) :-
16      append(Prefix, _, List).
17
18  commonprefix(Prefix, Lists) :-
19      maplist(prefix_of(Prefix), Lists).
```

```
?- [longestCommonPrefix].
true.

?- commonprefix(Prefix, ["interview", "interrupt", "integrate", "intermediate"]).
Prefix = [] ;
Prefix = [i] ;
Prefix = [i, n] ;
Prefix = [i, n, t] ;
Prefix = [i, n, t, e] ;
false.
```

The code above shows that I used the built in functions, append and maplist. Append adds the letter that is involved in the prefix to a list. This is evident above where the code is executed. The other function maplist, takes the prefix of each word. It checks if its able to be applied to other words in the list. This is how the program knows to stop after it finds the letter “e” which is present at the end of the longest prefix between the words given in this example above. From this point on in the assignment, between research and me working on it, I began to attempt to get it to just return the prefix which is found in the final list above.

Through further research, I learned of the built-in “findall” function. This creates, along with the “maplist” function, all possible prefixes of the words and puts them into a list. Using another built-in function, “last”, I was able to return the largest common prefix as it would be the last element in the list made by the “findall” function. However, this was returning one list, and I wanted it to return a string, so I went back and wrote a toString function. I did this using the built-in function, “atom_chars”. This converts a list of atoms to a string. The end result is found at the beginning as it was used to complete the test cases.

Final state of the prolog code:

```
15
16  :- set_prolog_flag(double_quotes, chars).  % "abc" = [a,b,c]
17
18  prefix_of(Prefix, List) :-
19     append(Prefix, _, List).
20
21  toString(Str,Longest):-                 %%Takes a list of atoms and converts it to a String
22      atom_chars(Str,Longest).            %%   using inbuilt atom_chars function
23
24  longestCommonprefix(Prefix, List) :-
25     findall(Letters, maplist(prefix_of(Letters),List), ListOfAllPrefixes),
26     last(ListOfAllPrefixes, LongestPrefix),
27     toString(Prefix, LongestPrefix).
28
```

Efficiency of the Two Programmes:

Haskell:

```
ethan@Jessica-HP-Pavilion-x360:~/Documents/ThirdYear/CA341/LogicProgrammingVSFunctionalProgramming$ runhaskell longestCommonPrefix.hs
inte
ethan@Jessica-HP-Pavilion-x360:~/Documents/ThirdYear/CA341/LogicProgrammingVSFunctionalProgramming$ ghc longestCommonPrefix +RTS -sstderr
[1 of 1] Compiling Main             ( longestCommonPrefix.hs, longestCommonPrefix.o )
Linking longestCommonPrefix ...
      99,638,008 bytes allocated in the heap
      40,930,728 bytes copied during GC
       8,277,056 bytes maximum residency (6 sample(s))
       2,129,632 bytes maximum slop
              22 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0        75 colls,     0 par    0.038s   0.038s     0.0005s    0.0040s
  Gen  1         6 colls,     0 par    0.065s   0.065s     0.0109s    0.0179s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.001s  (  0.001s elapsed)
  MUT     time    0.052s  (  0.569s elapsed)
  GC      time    0.103s  (  0.103s elapsed)
  EXIT    time    0.011s  (  0.011s elapsed)
  Total   time    0.188s  (  0.683s elapsed)

  Alloc rate    1,915,621,676 bytes per MUT second

  Productivity  44.6% of total user, 84.8% of total elapsed

gc_alloc_block_sync: 0
whitehole_spin: 0
gen[0].sync: 0
gen[1].sync: 0
ethan@Jessica-HP-Pavilion-x360:~/Documents/ThirdYear/CA341/LogicProgrammingVSFunctionalProgramming$
```

Prolog:

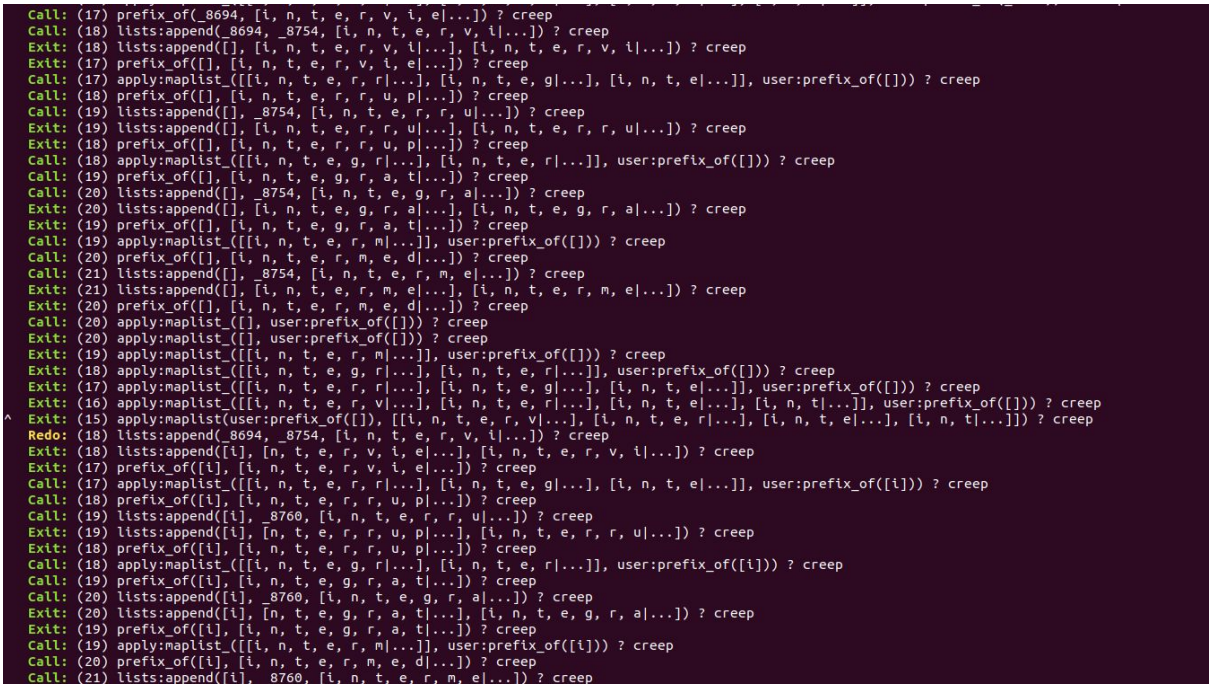
```
?- time(longestCommonprefix(Prefix, ["interview", "interrupt", "integrate", "intermediate"])).
% 166 inferences, 0.000 CPU in 0.000 seconds (95% CPU, 1643792 Lips)
Prefix = inte.

?- profile(longestCommonprefix(Prefix, ["interview", "interrupt", "integrate", "intermediate"])).
============================================================================
Total time: 0.000 seconds
============================================================================
Predicate                         Box Entries =    Calls+Redos     Time
============================================================================
$sig_atomic/1                               1 =        1+0         0.0%
atom_chars/2                                1 =        1+0         0.0%
$add_findall_bag/1                          5 =        5+0         0.0%
$new_findall_bag/0                          1 =        1+0         0.0%
$collect_findall_bag/2                      1 =        1+0         0.0%
$destroy_findall_bag/0                      1 =        1+0         0.0%
apply:maplist_/2                            1 =        1+0         0.0%
maplist/2                                   1 =        1+0         0.0%
lists:last_/3                               1 =        1+0         0.0%
append/3                                   32 =       22+10        0.0%
last/2                                      1 =        1+0         0.0%
longestCommonprefix/2                       2 =        1+1         0.0%
toString/2                                  1 =        1+0         0.0%
prefix_of/2                                22 =       22+0         0.0%
Prefix = inte.
```

Both programs were tested for efficiency on the same machine and with the same test case to get results as accurate as possible for comparisons. They were both tested with the list ["interview", "interrupt", "integrate", "intermediate"].

In the first image above, the efficiency of the longest common prefix program in haskell is evident. There are two important times to look at here, the MUT time and GC time. The MUT time signifies how long it takes the program to execute. The GC time signifies how long is spent in the garbage collector. The longer the time spent in the garbage collector, the less efficient the program is. The MUT time for this program with the test case specified above is 0.052s (0.569s elapsed). The GC time for the same test case is 0.103s.

In the second image above, the efficiency of the longest common prefix program in Prolog is visible. It takes 166 interferences until the program finishes executing using the test case specified above. These interferences represent how many calls it takes for the program to execute. Using the trace keyword in Prolog, I was able to follow all of the calls. Some of the calls can be seen in the image below.



Overall, efficiency wise, the programs are not that great. They're not terrible in the sense of wasting time or memory but they are not great. Improvements can definitely be made in various areas to improve efficiency throughout the two programs.

Comparison:

Functional programming is a programming paradigm that is based upon lambda calculus. It treats computation as the evaluation of mathematical functions. It avoids changing-state and mutable data. In functional programming, the returned value of a function depends only on the arguments that are passed to the function.

Logic programming is another programming paradigm, however, it is based on predicate calculus, whereas functional programming is not. It is also based on formal logic. A program written in any logical programming language e.g. Prolog, results in being a set of sentences in logical form. It expresses facts and rules about some problem domain.

An example of some facts written in prolog, a logical programming language would look like this:

```
1   parent(pam, bob).
2   parent(tom, bob).
3   parent(tom, liz).
4   parent(bob, ann).
5   parent(bob, pat).
6   parent(pat, jim).
```

An example of some rules written in the same language:

```
8   ancestor(X, Y) :-
9       parent(X, Y).
10
11  ancestor(X, Y) :-
12      parent(X, Z), ancestor(Z, Y).
13
14
```

An example of a query in Prolog based off the examples above:

```
16  ancestor(tom, bob).
17
```

Both of the paradigms of programming are declarative, however, this is where the similarities end between the two. The two languages have two differences that are most evident when you first get introduced to them.The first difference between the two programming paradigms is that functional programming uses mathematical expressions to solve problems. Opposite to functional programming, logic programming uses logic expressions to solve problems. The second difference which is evident between these two programming paradigms is their output values and their computations of retrieving said values. Functional programming uses functions to output such values whereas logic programming uses predicates to output either True or False.

In my functional implementation of my program to find the longest common prefix, It uses recursion and the “foldl1” built-in function to compare every word letter by letter until it reaches the end of the list and outputs the longest common prefix. This is different to how my logical implementation of this problem finds the resulting answer. Although it does use recursion, it uses it differently compared to my other implementation. It uses recursion to find every prefix between the words in the given list. It then returns a list of all the common prefixes. Within this list is more lists of the letters used to make up the prefixes that were found. I get the longest prefix by using the built-in “last” function. I then convert that list to a string using the “atom_chars” built-in function.

Although they both use recursion to find the longest common prefix, the two programs go in very different ways in implementing recursion throughout their individual programs.

In conclusion, throughout this assignment, I have discovered that I preferred implementing the functional aspect over the logical programming part of this assignment. I believe that was due to the fact Haskell is fresher in my mind compared to prolog. I also discovered through my research on these languages used within this assignment, the many reasons and appropriate times as to when you would

implement functional programming. That is also the case for learning when to implement logic programming. Overall, I have learned plenty about functional programming, logic programming and recursion throughout this assignment that will undoubtedly benefit me in the future.

References:

Understanding the function findall - found here:
https://www.swi-prolog.org/pldoc/man?predicate=findall/3

Understanding foldl and foldl1 - found here:
https://ca320.computing.dcu.ie/T03_Functional.pdf
https://stackoverflow.com/questions/49123178/haskell-foldl1-how-does-it-work

How to convert list of characters to a string in Prolog - found here:
https://www.swi-prolog.org/pldoc/man?predicate=atom_chars/2

Understanding how to use maplist function in prolog - found here:
https://www.swi-prolog.org/pldoc/man?predicate=maplist/2

Functional and logic programming paradigm - found here:
https://www.computing.dcu.ie/~davids/courses/CA341/CA341_Logic_Programming_Paradigm_2p.pdf
https://www.computing.dcu.ie/~davids/courses/CA341/CA341_Functional_Programming_Paradigm_2p.pdf