COM2040 – Coursework - 6529925

This document serves to support the work produced throughout the COM2040 Coursework. It will consist of a small paragraph describing my approach to solving each subsection and where to find it in the .zip file provided.

CPP

The C++ files are found under cpp in the archive provided.

1.1 Class Design

In the DynamicArray header file, parts a through f have been implemented. I have split the class into private and public and included the appropriate functions and variables in each.

1.2 Implementation

In the DynamicArray c++ file I have implemented all the functions and given the class functionality. I have implemented parts a through e.

I will explain how I implemented the prepend function:

In the class instance I have a size and a current_size. If the current array doesn't have any spare places it creates a temporary array that is one bigger than its current size and copies it all over, leaving one space at the beginning for the new element to be prepended. If the Array has a spare space it moves all the elements along one, using a temporary array, and then adds the new element to the beginning. This is slightly redundant though, as in my constructor the size and current_size are set to the size passed in as a parameter.

1.3 Testing

This is in main.cpp and is self-explanatory. In my program, parts a through e appear in order. The coursework handout didn't say anything about using c++ test cases so I haven't implemented this but this could easily be done.

```
5, 4, 3, 2, 1, 52, 77, 53, 81, 74,
Length:  10
Printing Double Type in Scientific Notation:
-1.222330e+02, 9.333333e+00, 8.987654e+00, 1.234560e-01, 3.333333e+00, 1.000000e+00, 0.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00,
Length:  10
```

Haskell

The Haskell files are found under haskell in the archive provided.

2.1 Basic RPN

This part includes the step and rpn function defined in rpn.hs. I debated changing the type of the rpn function from [String] to String and having:

```
rpn = head . foldl' step [] . words
```

But decided against this as it wasn't in the question although would make it much faster to use. I did use String in the recursive RPN function despite the question saying [String] because I felt it made the program easier to use and wanted to implement it at some point.

The head operator takes the leftmost item in a list, this allows the returned data to be Int and not [Int]. The foldl function allows the program to collapse the user-provided expression in on itself. While reading the documentation for foldl I found a similar function called foldl' which "is the more efficient way to arrive at that result" so I used this. Although, full disclosure, I'm not quite sure why it's more efficient.

Regarding the step function: I have had a few versions, including the one below:

```
step :: [Int] -> String -> [Int]
step [] _ = []
step [x] _ = [x]
step (y:x:ns) o
        | o == "+" = (x+y):ns
        | o == "-" = (x-y):ns
        | o == "*" = (x*y):ns
        | o == "/" = (x `div` y):ns
        | otherwise = (read o :: Int) : y:x:ns
```

But settled on the one shown in my solution as it uses less lines and does the same job; prettier. Its idea is that it uses the first two elements of the stack and then combines them based on the operator provided and returns the stack. Unless the string doesn't match any of its known operators, in which case it adds this number to the beginning of the stack.

To solve 2.1c I simply found the documentation for the fold function, found how it worked and created my own custom_fold function.

```
custom_fold f z [] = z
custom_fold f z ( x : xs ) = custom_fold f ( f z x ) xs
```

2.2 Polish Notation Evaluator

I decided to implement it much like RPN but in reverse.

```
pn :: [ String ] -> Int
pn = head . foldr pn_step []


pn_step "+" ( x1 : x2 : xs ) = x1 + x2 : xs
pn_step "-" ( x1 : x2 : xs ) = x1 - x2 : xs
pn_step "*" ( x1 : x2 : xs ) = x1 * x2 : xs
pn_step num xs = read num : xs
```

It works identically to the rpn but uses foldr and using x1:x2 -> x1 *operator* x2.

2.3 Errors

In this question I first defined a custom data structure as outlined in the handout. This will be used to display to the user a status message along with the returned answer.

Next, I defined another step function, called step_error, that is more or less a duplicate of the normal step function with the addition of a few lines:

```
step3  _ "+" = Nothing
step3  _ "-" = Nothing
step3  _ "*" = Nothing
step3  xs num = Just (read num : xs)
```

The lines above contain a wildcard, "_" which match anything that is passed in. This means that if it reaches this point it doesn't match the lines above and must be an error; therefore, it produces "Nothing". I also needed to add "Just" to the function as I am using Maybe.

I then created the rpn function, "rpn_error", which takes in a string array, calls another function responsible for producing an output and then passes this to the data structure where it is displayed. It works by passing the entered string array, along with another empty array (the stack) to the rpn_state function. I will breakdown each line now:

```
rpn_state [x] [] = Success x
```

This line looks to see if there is a single integer in the left stack, if so it displays this to the user with the Success message.

```
rpn_state stack [] = Incomplete stack
```

This would match if there was a stack which had more than one integer remaining, but no operators left to process. An example of an input that would trigger this would be “rpn_error [“1”,”2”]”.

```
rpn_state stack input@ ( x : xs ) =
  case step3  stack x of
    Nothing -> Stuck stack input
    Just stack -> rpn_state stack xs
```

This is the last state the function could be in, and is where all the processing is done. It takes the current stack and the remaining input and processes it. It uses the “@” operator which is another way of saying “as” and splits the input into two parts. The first part being the leftmost entry in the array and the second part being the remaining data; only the leftmost part is used in this call.

It then enters the case statement which selects the relevant operation based on the result of the step_error call. If the step_error call returns “Nothing”, which would happen in an error scenario, the state function would produce a ‘Stuck’ Error. If the step_error call returns “Just” then it worked as intended and it then calls itself again; a recursive loop.

2.4 Caller-Defined Extensions

2.4a Implementation of RPN4

I first wrote the exts function but as this is first on the document, I’ll explain this first.

I defined another step function, step4.

```
step4 stack user_defined_string =
 if (user_defined_string `elem` ["sum", "prod", "fact", "fib"]) then
  case exts user_defined_string of
  Just user_defined_function -> user_defined_function stack
 else
  step stack user_defined_string
```

This function works by checking to see if the parameter for the user_defined_string is a known extension function. If it isn’t then it proceeds as normal to the step function. If it is then the case statement will aim to workout what the function call is. It returns “Just user_defined_function” which I then use to call this operator on the stack; “user_defined_function stack”.

2.4b Implementation of exts

```
exts :: String -> Maybe ([ Int ] -> [ Int ])
exts "sum" = Just (summation)
exts "prod" = Just (product_function)
exts "fib" = Just (fib_function)
exts "fact" = Just (fact_function)
```

This function works by taking a string and matching it with its associated function.

I then defined all the functions in separate places:

```
summation :: [Int] -> [Int]
summation stack = [sum stack]

product_function :: [Int] -> [Int]
product_function stack = [product stack]

fib_function :: [Int] -> [Int]
fib_function = map fib

fact_function :: [Int] -> [Int]
fact_function = map fac
```

These use the following functions:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fac :: Int -> Int
fac n = foldl(*) 1 [1..n]
```

An example of the function working is shown below:

```
*Main> rpn4 exts ["1","2","3","4","sum"]
10
```

Haskell Testing

2.1.ai

```
*Main> step [3,4] "+"
[7]
```

2.1.aii

```
*Main> step [3,4] "-"
[1]
```

2.1.aiii

```
*Main> step [3,4] "*"
[12]
```

2.1.aiv

```
*Main> step [] "1"
[1]
```

2.1.bi

```
*Main> rpn ["3","4","+"]
7
```

2.1.bii

```
*Main> rpn ["21","5","1","1","+","+","*","3","*","2","2","+","-"]
437
```

2.1.ci

```
*Main> rpnRec ["3","4","+"]
7
```

2.1.cii

```
*Main> rpnRec ["21","5","1","1","+","+","*","3","*","2","2","+","-"]
437
```

2.2i

```
*Main> pn ["+", "3", "4"]
7
```

2.2ii

```
*Main> pn ["-", "*", "*", "+", "21", "5", "+", "1", "1", "3", "+", "2", "2"]
152
```

2.2iii

```
*Main> pn ["-","+","2","2","*","3","*","+","+","1","1","5","21"]
-437
```

Prolog

The Prolog files are found under prolog in the archive provided.

3.1 Recursive Predicates

I started by defining a dictionary, as shown below.

```
equivalent(hello,bonjour).
```

I then created a language switching function that split the list of words into a leftmost and then a list of words to the right of it. Much like the head/tail in Haskell. This then converted the first word and made a recursive call on the remaining in the list. After all the words have been translated (or an attempt has been made to translate them) the result is displayed.

“lang_switch([hello, cat,goodbye, car], F).”

3.2 Shortest Path

I solved this problem using Dijkstra’s Algorithm. I started by defining all the edges on the graph and assigning them with a weight as shown in the coursework handout:

```
edge(1, 2, 4).
edge(1, 3, 10).
edge(1, 5, 2).
edge(2, 5, 1).
edge(2, 6, 1).
edge(2, 4, 6).
edge(3, 5, 2).
edge(3, 4, 1).
edge(4, 5, 8).
```

The following lines link the edges together using ‘connected’. I have duplicated the line and swapped the edges to show that the connection is bidirectional.

```
connected(X,Y,L) :- edge(Y,X,L).
connected(X,Y,L) :- edge(X,Y,L).
```

The path function calls the traverse function and then reverses it as otherwise it would appear in the wrong order.

```
path(X,Y,Path,Len) :- traverse(X,Y,[X],Q,Len), reverse(Q,Path).
```

The traverse function is used to actually find the shortest path between two points. It works by first checking to see if they are directly linked. If they are then it will return the edge and its weight. If not it will then check to see if they are equal using the “\==” operator. This will only allow the execution of this call to continue if theyre not equal. It will then check to see if the current node has already been visited using the member function. The “\+” operator will only allow execution of this call to continue if it isn’t currently a member. This prevents cycles/loops. It will then make a recursive call with the new node as the start node and the target node as it was. It also adds the new start node to the members list to keep track of those visited. As it recurses it adds together the lengths of each

```
traverse(X,Y,P,[Y|P],L) :- connected(X,Y,L).
traverse(X,Y,Visited,Path,L) :- connected(X,Z,N), Z \== Y,
\+member(Z,Visited),traverse(Z,Y,[Z|Visited],Path,L1), L is
N+L1.
```

edge used and returns this to the user.

This function is the one that is called by the user and works by getting a list of all the paths and the lengths and finding the one of minimal length. This is then returned to the user.

```
find(X,Y,Path,Length) :-
   setof([P,L],path(X,Y,P,L),Set),
   Set = [_|_],
   minimal(Set,[Path,Length]).
```

I also use minimum finding functions that I found on the prolog documentation website. Before submitting I remembered that it also needed to display all paths so I then added this to the code as shown below:

```
path(X,Y,Path,Len) :- traverse(X,Y,[X],Q,Len),
            reverse(Q,Path),
            print(Path),
            writef("\n").
```

3.3a Constraint Solving

Prolog works by assigning values to variables in a way that will allows the clause to succeed. Once a variable has been assigned a value it cannot be reassigned by subsequent code. This means it must backtrack if it realises its current variable values cant allow the rest of the clause to be satisfied. You will see in the tree.pdf that the calls backtrack if that decision branch cant be satisfied.

```
prefix(P,L) :- append(P,_,L).

suffix(S,L) :- append(_,S,L).

sublist(SubL,L) :- suffix(S,L),prefix(SubL,S).
```

The sublist function works off of a call that will look something similar to the one below:

```
Call:sublist([house(blue, _12772, _12774), house(_12784,
_12786, snail)], [house(blue, spanish, jaguar), house(black,
french, snail), house(brown, japanese, _12710)])
```

This looks quite complex at first but is basically passing in two house arrays. One is the blue house (with unknown family and pet) and its neighbouring house, the one containing the snail. And the other is the street so far, as we think it is. This call then calls suffix which uses the append function to return the possible suffixes, starting with the full list and, with the help from backtracking, dropping one from the start until it returns just []. The prefix function is called next, which works in the exact opposite way as suffix. It starts with [] and adds an element with each backtrack call until it is a list of all elements. A table below will show this in action with the array [a,b,c,d].

Backtrack iteration	**Suffix**	**Prefix**
First iteration	[a,b,c,d]	[]
First backtrack call	[b,c,d]	[a]
Second backtrack call	[c,d]	[a,b]
Third backtrack call	[d]	[a,b,c]

These two functions work brilliantly together when trying to find all the ways a list can be split, much like this problem required when finding which order the houses could be in based on the rules given to allow the clause to succeed.

This is how I implemented the sublist function, which returns a different sublist with each backtrack call. The returned lists are the prefixes of the suffixes of the list provided.

The rest of the code is outlining the rules we know about the street and working out which family owns the rabbit. The member function is responsible for adding the elements to the lists by way of unification. The street is defined with _House1 etc. The underscore before the variable name means its an anonymous variable which can be replaced by way of unification using the member function. This means that as the member function checks to see if it is in the list, if there is an anonymous variable it will add it in its place.

The insertion of (_, Spanish, jaguar) works in a similar way. It tries to find where there is an empty slot in the list for a nationality and pet; in this case it unifies the empty place with the blue house as the black is taken with French. This is lucky as this allows the clause to succeed when it goes to the sublist section but if in this case it added it to the brown house by mistake it wold proceed to the sublist section and fail. This would cause the program to backtrack and put the Spanish,jaguar with the blue house.

```
rabbit(N) :-
        Street = [_House1,_House2,_House3],
        member(house(blue,_,_), Street),
        member(house(black,_,_), Street),
        member(house(brown,_,_), Street),
        member(house(black,french,_), Street),
        member(house(_,spanish,jaguar), Street),
        sublist([house(blue,_,_),house(_,_,snail)], Street),
        sublist([house(_,_,snail),house(_,japanese,_)], Street),
        member(house(_,N,rabbit),Street).
```

As you can see, the street information is completely full:

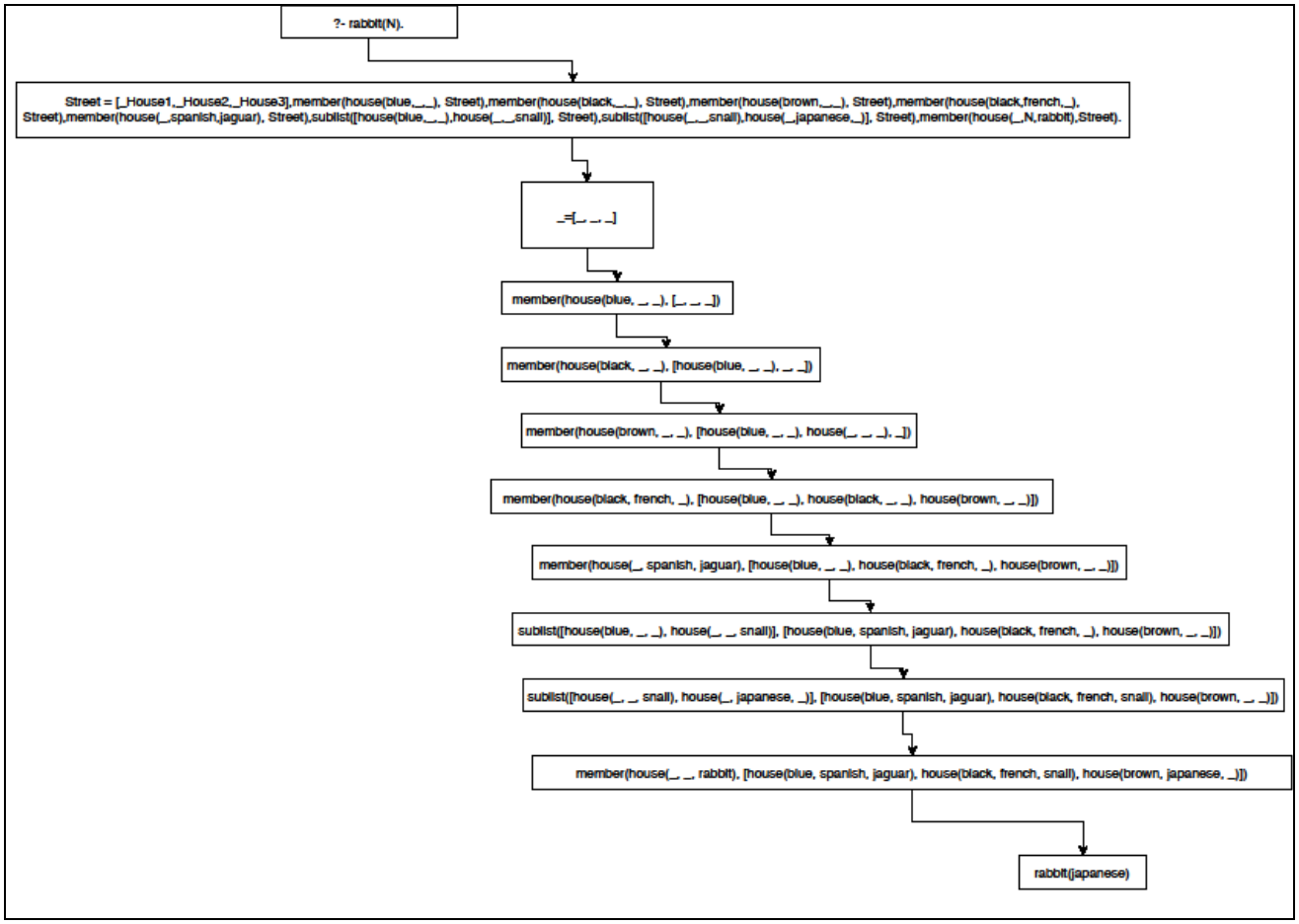
```
Exit:lists:member(house(brown, japanese, rabbit), [house(blue, spanish, jaguar), house(black, french, snail), house(brown, japanese, rabbit)])
```

The program can now return the owner of the rabbit to the user.

3.3b Constraint Solving – Tree

The decision tree shows the calls that my program takes when trying to find a solution to the clause.

This can be seen in full at "/prolog/tree.pdf"



As the order in which members are added to the street doesn't require any backtracking it flows down to the right with each call. If there was a backtrack it would have a second branch off of the call in which it would try again with different returned values from the call.