Programming Paradigms Assignment

Report

The word ‘paradigm’ can also be termed as method to solve some problem or do some task. A programming paradigm is an approach to solve problem using some programming language, or a method to solve a problem using tools and techniques that are available following some approach (GeeksforGeeks, n.d.). The four most common paradigms are; Object Oriented, Procedural, Functional, and Logical. Each differ vastly, however Object Oriented and Procedural share some similarities as imperative paradigms, and functional and logical programming are both known as declarative. This report will explore each of these paradigms in detail, to explain what defines them, their relative advantages and disadvantages, and how these affect the appropriateness of using the paradigm to solve the given problem. The task is to implement a simple but crucial text editor system as part of a major contract with a security company to provide fully cloud-based services to a private operator in private equity establishment. Due to security concerns and for a risk of compromising data, the client avoids using the publicly available commercial software. The text editor will be used to record transactions keys of VIP clients from the transactions contained within a multi-platform application controlled from high-end business servers. Given this, it is crucial to consider the ability of each paradigm to fulfil the basic functionality of a text editor as well as other relevant factors such as security and the cost, speed, and ease of development.

Object-oriented programming (OOP) is a programming language model in which programs are organized around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behaviour (Rouse, 2019). The three major features of OOP are encapsulation, inheritance, and polymorphism. OOP is arguably the most popular paradigm since the five most in-demand languages of 2018 (Java, Python, JavaScript, C++, and C#) (Misirlakis, 2017) all have object-oriented features. SmallTalk is an example of a purely object-oriented language. It is an integrated programming language, development tool, and runtime environment, and everything in it is an object, which is essentially an independent chunk of code that manages a specific piece of data (Kb.iu.edu, 2018). A major advantage to OOP is improved software-development productivity (Resources.saylor.org, n.d.) due to its modular nature. This makes OOP an excellent paradigm for development as part of a team since subsections of the code can be allocated to different team members for efficient development. However, this advantage cannot apply to the scenario since there will be one developer only. Nevertheless, the concept of objects can also result in improved software maintainability since only specific parts of the code need to be edited even if significant changes are required. While this advantage will not immediately apply to this scenario, it may become relevant in the future if the company decide to alter or add to the functionality of the basic text editor. It is also extensible, as objects can be extended to include new attributes and behaviours, they can also be reused within and across applications. Reuse enables both faster and cheaper development since OOP languages generally come with rich libraries of objects, and any code developed during

projects can be reusable in future projects (Resources.saylor.org, n.d.). Faster development and lower costs will always be a relevant advantage regardless of the scenario, and the reusability may be useful in future for making a similar tool to serve a different purpose. Furthermore, as each object can be assigned their own access modifiers meaning they can be made private, protected, or public, this may improve the security of the code and program. On the other hand, a well-known disadvantage of OOP is the steep learning curve, the thought process involved in this paradigm may not be natural for some people and can take some time to get used to it (Resources.saylor.org, n.d.). As the developer of this project, I can confess that I have struggled with OOP in the past so picking up an unfamiliar language of this type may be something to avoid. Furthermore, programs created with OOP languages are typically larger and slower (Resources.saylor.org, n.d.). They typically require more lines of code than procedural programs, for example, which may over-complicate something that could be a relatively simple task and require more instructions to be executed, slowing down a basic program. While the speed may not make a noticeable difference for a small and simple program such as a text editor, the longer and more complex code required may slow down development significantly. To consider SmallTalk specifically as the purest object-oriented language, a benefit is that it is non-commercial. This is advantageous in this scenario because programming efforts and time are not in fact dependent on the continued success of a commercial firm, and the rareness of bugs and infelicities in its hidden source (Maartensz.org, n.d.). It is also open source, and whilst this can be advantageous in some cases, it may be detrimental in this scenario since development of the source is "up to the public", the development may be interfered with by all manner of social processes having little to do with developing source (Maartensz.org, n.d.). Overall, OOP is definitely a paradigm worth considering for this project, but it is not the ideal solution.

Procedural programming is a programming paradigm that uses a linear or top-down approach, which relies on procedures or subroutines to perform computations (Techopedia.com, n.d.). Procedural programming was derived from structured programming and is also known as imperative programming. The word ‘imperative’ is used to describe this paradigm because the code should instruct the computer exactly what to do in logical steps. C and Pascal are well-known and popular examples of procedural languages. An obvious advantage of procedural programming is that in many cases, procedural languages are easier to learn and use than others. This is largely due to ability to jump right into coding a program without the need to create any objects or classes (Eliason, 2013). As well as this, the top-down structure allows one to easily change and develop code as it is being written rather than requiring significant planning in advance. Furthermore, there are many books and references available on well-tried and tested algorithms (Teach-ict.com, n.d.) so expertise is not required. Ease of learning and use is a definite advantage in any situation, especially if the programmer has no previous experience as is the case in this scenario. Whilst it may mean that an inexperienced developer can create something impressive and useful in a relatively short amount of time, there are many more factors needed to consider the paradigm’s appropriateness to this task. For example, a major disadvantage of procedural programming is the inability to reuse code throughout the program - having to

rewrite the same type of code many times throughout a program can add to the development cost and time of a project (Eliason, 2013). This may be an issue for the task at hand since some functions of a text editor may be fairly similar and therefore their code may be reusable in a different paradigm. This may mean that the basic program would require a lengthy code written in a procedural language, it could also result in a slower running time but that may not be noticeable in this fairly small-scale case. Furthermore, another disadvantage of procedural programming is that it struggles to handle situations in which a number of possible actions may lead to the desired result (Reference.com, n.d.). In a classic example of an advanced text editor, this would be a problem since a variety of click and keyboard combinations could potentially result in the same action. This would clearly make procedural programming inappropriate for text editors in general, however it may be of little significance in this simplified example of a very basic text editor. Lastly, the data used in procedural languages are exposed to the whole program so there is no security for the data (Dhananjaiyan.blogspot.com, 2011). This is a very concerning drawback of this paradigm since the purpose of creating a new text editor for the company rather than using an existing one is security. Overall, it seems clear that the disadvantages of this paradigm far outweigh the advantages, especially in this scenario. Therefore, a paradigm which offers more flexibility, reusability, and security would be far more appropriate to fulfil this task.

Functional programming is the process of building software by composing pure functions, avoiding shared state, mutable data, and side-effects. Functional programming is declarative rather than imperative, and application state flows through pure functions (Elliot, 2017). Its main focus is on “what to solve” in contrast to an imperative style where the main focus is “how to solve”. It uses expressions instead of statements, an expression is evaluated to produce a value whereas a statement is executed to assign variables (GeeksforGeeks, n.d.). It is based on Lambda Calculus which is a type of formal system from mathematical logic used in computer science for function definition, application and recursion (Techopedia.com, n.d.). JavaScript is the most popular example of a language with functional features, and Haskell is an entirely functional language. Functional programming is famous for its high-level abstractions that hide a large number of details of such routine operations like iterating. This makes the code shorter and, as a consequence, guarantees a smaller number of errors that can be tolerated (Korkishko, 2018). This is a significant advantage over OOP because it would allow for relatively short and simple code for the basic program. Furthermore, in functional programming, there is a smaller number of language primitives - well-known classes are not used. Instead of creating a unique description of an object with operations in the form of methods, in functional programming, there are several basic language primitives that are well optimized inside (Korkishko, 2018). This may mean that new functional languages are easier to learn since there is simply less to have to learn. As well as this, because pure functions don’t change any states and are entirely dependent on the input, they are simple to understand. The return value given by such functions is the same as the output produced by them, and the arguments and return type of pure functions are given out by their function signature (Bhadwal, 2019). Moreover, functional programming supports the concept of ‘lazy evaluation’, which means that the value is evaluated and stored only when it is required (Bhadwal, 2019). The power of lazy

evaluation will make the program run faster because it only provides what we really required for the queries result (packtpub.com, n.d.). This may mean that a simple text editor created with a functional language may be the most efficient solution. However, there are also some drawbacks to functional programming. For example, since there's no state and no update of variables is allowed, loss of performance will take place. The problem occurs when we deal with a large data structure and it needs to perform a duplication of any data even though it only changes a small part of the data (packtpub.com, n.d.). But, this should not be an applicable problem to our scenario since our simple text editor will not be dealing with a large data structure as it only needs to record names and numbers. On the other hand, while the steep learning curve of OOP seemed like a cause for concern, it may be even more problematic with functional programming. It has a much steeper learning curve than OOP because the broad popularity of OOP has allowed the language and learning materials of OOP to become more conversational, whereas the language of functional programming tends to be much more academic and formal. Functional concepts are frequently written about using idioms and notations from lambda calculus, algebras, and category theory, all of which requires a prior knowledge foundation in those domains to be understood (Klimenko, 2016). Although it may be more challenging than initially anticipated, overall it seems that functional programming is a very appropriate solution to this scenario so it would not be worth sacrificing appropriateness and efficiency over ease. This means that a functional programming language will be strongly considered to complete this task. Considering Haskell specifically, an important benefit is performance and its fast prototyping capabilities. The huge difference with other fast prototyping languages such as Ruby and Python is that a Haskell prototype can become the real app and stand the test of time due to its conciseness and correctness (Contorer, 2013). This makes Haskell rather appropriate for this scenario since the program will be used to store important client information over many years.

Logical programming is a computer programming paradigm in which program statements express facts and rules about problems within a system of formal logic. Rules are written as logical clauses with a head and a body and facts are expressed similar to rules, but without a body (Computerhope.com, 2018). Logical languages are relatively unpopular and unknown, Prolog may be the most familiar example and this is very specific to Artificial Intelligence, implying that it is not widely applicable. There are some advantages to this paradigm. Logic programming can be used to express knowledge in a way that does not depend on the implementation, making programs more flexible, compressed and understandable. It enables knowledge to be separated from use, i.e. the machine architecture can be changed without changing programs or their underlying code (Doc.ic.ac.uk, 2006). However, as mentioned previously, this simply cannot be applied to the scenario since logical programming is so niche to artificial intelligence and machine learning. Since logical programming and functional programming are both declarative paradigms, they share many similarities and advantages. However, the differences between these two paradigms make functional programming far more suitable. The difference between logical and functional languages are illustrated in the blocks used in each, functional languages often use functions which return a value whereas logic programming often uses predicates which gives true or false as a result rather than values (Bouiti, 2014). Functional languages are used in many

areas such as industry compilers and software development, and also academically especially in the mathematical discipline. Comparing to functional programming, logic programming is not widely used except in small applications such as database management systems, expert systems and NLP. It is clear that no further information is needed to rule this paradigm out as an option for solving this task as it is simply inapplicable and inappropriate.

After considering all of the information, it is clear that there are only two plausible paradigms that can be used to solve this problem; object-oriented and functional. Procedural programming raised numerous causes for concern such as its lack of flexibility, reusability, and security. Logical programming was a clearly impractical and inappropriate solution since its applications beyond artificial intelligence are very limited. Object-oriented is a fairly attractive solution since it offers productivity, maintainability, flexibility, and reusability. However, its relative complexity may be somewhat "overkill" for this simple program since it may result in an extensive piece of code that may be quite difficult to develop and understand. As SmallTalk is the purest language of this paradigm, it was important to consider its benefits and drawbacks too. The fact that it is open source and fairly poorly documented may mean that it is both difficult to learn and use and is inappropriate for the needs of the company. The final paradigm to be considered is Functional. The principal fact that everything is done using pure functions makes it appropriate for a text editor since operations are carried out from specific actions. Furthermore, this style usually results in a shorter piece of code and leaves less room for errors which would be suited and expected for a simple program. As with all paradigms, however, there were also some disadvantages to consider. For example, it may be more difficult to learn due to its more formal and less readily available documentation. However, this should be considered on a language-level instead and after brief research, it appeared that the pure functional language (Haskell) is better documented than the OOP alternative, SmallTalk. Besides this, there are far more important factors that should be used to make an informed decision rather than the ease of development. Another drawback that was discovered was the loss in performance with large data. However, as mentioned previously, this should be irrelevant in this scenario since the datatypes should remain small. In conclusion, it seems that functional programming is the most ideal solution for the task at hand since the associated drawbacks are largely inapplicable in this scenario. Furthermore, as Haskell seems to be a relatively consistent and reliable language, it is an appropriate choice.

References:

Resources.saylor.org. (n.d.). Advantages and Disadvantages of Object-Oriented Programming (OOP). [online] Available at: https://resources.saylor.org/wwwresources/archived/site/wp-content/uploads/2013/02/CS101-2.1.2-AdvantagesDisadvantagesOfOOP-FINAL.pdf [Accessed 17 Apr. 2019].

Maartensz.org. (n.d.). Advantages and disadvantages of Squeak and Smalltalk. [online] Available at: http://www.maartensz.org/computing/squeak/Helps/Background/Remarks/Advantages%20and%20disadvantages.htm [Accessed 20 Apr. 2019].

Bhadwal, A. (2019). Functional Programming: Concepts, Advantages, Disadvantages, Applications. [online] Hackr.io Blog. Available at: https://hackr.io/blog/functional-programming-concepts-advantages-disadvantages-applications [Accessed 18 Apr. 2019].

Bouiti, S. (2014). The difference and the similarity of Functional and Logic Programming Languages. [online] Academia. Available at: https://www.academia.edu/11202753/The_difference_and_the_similarity_of_Functional_and_Logic_Programming_Languages [Accessed 18 Apr. 2019].

Contorer, A. (2013). In Praise of Haskell. [online] Dr. Dobb's. Available at: http://www.drdobbs.com/architecture-and-design/in-praise-of-haskell/240163246 [Accessed 18 Apr. 2019].

Dhananjaiyan.blogspot.com. (2011). Disadvantages of Procedural languages. [online] Available at: http://dhananjaiyan.blogspot.com/2011/07/what-are-disadvantages-of-procedural.html [Accessed 20 Apr. 2019].

Eliason, K. (2013). Difference Between Object Oriented and Procedural Programming • Page 2 of 3. [online] NeONBRAND. Available at: https://neonbrand.com/website-design/procedural-programming-vs-object-oriented-programming-a-review/2/ [Accessed 12 Apr. 2019].

Elliot, E. (2017). Master the JavaScript Interview: What is Functional Programming?. [online] Medium. Available at: https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0 [Accessed 18 Apr. 2019].

GeeksforGeeks. (n.d.). Functional Programming Paradigm - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/functional-programming-paradigm/ [Accessed 18 Apr. 2019].

GeeksforGeeks. (n.d.). Introduction of Programming Paradigms. [online] Available at: https://www.geeksforgeeks.org/introduction-of-programming-paradigms/ [Accessed 20 Apr. 2019].

Klimenko, R. (2016). What are the pros and cons of functional programming vs object-oriented programming?. [online] Trello.com. Available at: https://trello.com/c/GEVPl6hL/74-what-are-the-pros-and-cons-of-functional-programming-vs-object-oriented-programming [Accessed 18 Apr. 2019].

Korkishko, I. (2018). Pros and cons of functional programming. [online] ITNEXT. Available at: https://itnext.io/pros-and-cons-of-functional-programming-32cdf527e1c2 [Accessed 18 Apr. 2019].

Doc.ic.ac.uk. (2006). Logic Programming - Summary. [online] Available at: http://www.doc.ic.ac.uk/~cclw05/topics1/summary.html [Accessed 18 Apr. 2019].

Misirlakis, S. (2017). The 7 Most In-Demand Programming Languages of 2018 - Coding Dojo Blog. [online] Coding Dojo Blog. Available at: https://www.codingdojo.com/blog/7-most-in-demand-programming-languages-of-2018 [Accessed 17 Apr. 2019].

Rouse, M. (2019). What is object-oriented programming (OOP)? - Definition from WhatIs.com. [online] SearchMicroservices. Available at: https://searchmicroservices.techtarget.com/definition/object-oriented-programming-OOP [Accessed 15 Apr. 2019].

Teach-ict.com. (n.d.). Teach-ICT A Level Computing OCR exam board - Pros and Cons of Procedural Languages. [online] Available at: https://www.teach-ict.com/as_as_computing/ocr/H447/F453/3_3_6/types_language/miniweb/pg3.htm [Accessed 12 Apr. 2019].

packtpub.com. (n.d.). The advantages and disadvantages of functional programming. [online] Available at: https://subscription.packtpub.com/book/application_development/9781785282225/1/ch01lvl1sec13/the-advantages-and-disadvantages-of-functional-programming [Accessed 18 Apr. 2019].

Reference.com (n.d.). What Are the Disadvantages of Procedural Programming? [online] Available at: https://www.reference.com/technology/disadvantages-procedural-programming-eccac5f2db0faebd [Accessed 20 Apr. 2019].

Techopedia.com. (n.d.). What is Lambda Calculus? - Definition from Techopedia. [online] Available at: https://www.techopedia.com/definition/16402/lambda-calculus [Accessed 18 Apr. 2019].

Computerhope.com. (2018). What is Logic Programming? [online] Available at: https://www.computerhope.com/jargon/l/logic-programming.htm [Accessed 18 Apr. 2019].

Techopedia.com. (n.d.). What is Procedural Programming? - Definition from Techopedia. [online] Available at: https://www.techopedia.com/definition/21481/procedural-programming [Accessed 12 Apr. 2019].

Kb.iu.edu. (2018). What is Smalltalk? [online] Available at: https://kb.iu.edu/d/aeya [Accessed 17 Apr. 2019].

ADT

NAME

Text Editor – an ADT representing a basic Text Editor

SETS

T the set of text { (L, R, S, Cl) }

L the set of characters in the left string { [C] }

Ri the set of characters in the right string { [C] }

S the set of selected characters { [C] }

Cl the set of characters copied to the clipboard { [C] }

C the set of characters { a..z U A..Z U 0..9}

SYNTAX

CreateText:	⊥	→	T
DestroyText:	T	→	⊥
InitialiseText:	T	→	T
ShowText:	T	→	T
MoveLeft:	T	→	T
MoveRight:	T	→	T
MoveToStart:	T	→	T
MoveToEnd:	T	→	T
SelectPreviousChar:	T	→	T
SelectNextChar	T	→	T
MoveBeforeWord:	T	→	T
MoveAfterWord:	T	→	T
SelectPreviousWord:	T	→	T
SelectNextWord:	T	→	T
SelectToLeft:	T	→	T
SelectToRight:	T	→	T

SelectAllText:	T	→	T
InsertText:	T x C	→	T
Copy:	T	→	T
Paste:	T	→	T
DeletePreviousChar:	T	→	T
DeleteNextChar:	T	→	T
DeleteAllLeft:	T	→	T
DeleteAllRight:	T	→	T
Save:	T	→	T
Load:	T	→	T
Undo:	T	→	T
Redo:	T	→	T

SEMANTICS

Pre-Create( ) :: true
Post- Create (r) :: r = ([ ], [ ], [ ], [ ])

Pre-Destroy (t) :: true
Post-Destroy (t; r) :: r = ⊥

Pre-Init (t) :: true
Post-Init (t; r) :: r = (l, [ ], [ ], [ ])

Pre-ShowText (t) :: true
Post-ShowText (t; r) :: r = (l, ri, s, [ ])

Pre-MoveLeft (t, l) :: true
Post-MoveLeft ((l, ri, _, _); r) :: r = (init(l), (last (l) + (ri)), [ ], [ ])

Pre-MoveRight (t, ri) :: true
Post-MoveRight ((l, ri, _, _); r) :: r = ((l) + head(ri)), tail(ri), [ ], [ ])

Pre-MoveToStart (t, l) :: true
Post-MoveToStart ((l, ri, _, _); r) :: r = ([ ], l + ri, [ ], [ ])

Pre-MoveToEnd (t, ri) :: true
Post-MoveToEnd ((l, ri, _, _); r) :: r = (l + ri, [ ], [ ], [ ])

Pre-SelectPreviousChar (t, l) :: true
Post-SelectPreviousChar ((l, ri, s, _); r) :: r = (init (l), (last(l) + ri), (last (l) + s, [ ])

Pre-SelectNextChar (t, ri) :: true
Post-SelectNextChar ((l, ri, s, _); r) :: r = ((l) + head(ri), tail(ri), (s + head(ri)), [ ])

Pre-MoveBeforeWord (t, l) :: true
Post-MoveBeforeWord ((l, ri, _, _); r) :: r = Find (" ", last (l))
Where
Find (" ", last (l)) = (l , ri, [ ], [ ])
Find (c, last (l)) = MoveBeforeWord (MoveLeft)
Pre-MoveAfterWord (t, r) :: true
Post-MoveAfterWord ((l, ri, _, _); r):: r = Find (" ", head (ri))
Where
Find (" ", head (ri)) = ((l) + head(ri)), tail(ri), [ ], [ ])
Find (c, head (ri)) = MoveAfterWord (MoveRight)

Pre-SelectPreviousWord (t, l) :: true
Post-SelectPreviousWord ((l, ri, s, _); r) :: r = Find (" ", last (l))
Where
Find (" ", last (l)) = (l, ri, l, [ ])
Find (c, last (l)) = SelectPreviousWord (SelectPreviousChar)

Pre-SelectNextWord (t, r) :: true
Post-SelectNextWord ((l, ri, s, _); r) :: r = Find (" ", head (ri))
Where
Find (" ", head (ri)) = (l, ri, ri, [ ])
Find (c, head (ri)) = SelectNextWord (SelectNextChar)

Pre-SelectToLeft (t, l) :: true
Post-SelectToLeft ((l, ri, s, _); r) :: r = ([ ], (l + ri), l, [ ])

Pre-SelectToRight (t, r) :: true
Post-SelectToRight ((l, ri, s, _); r) :: r = ((l + ri), [ ], ri, [ ])

Pre-SelectAllText (t) :: true
Post-SelectAllText ((l, ri, s, _); r) :: r = (l, ri, (l + ri), [ ])

Pre-InsertText (t, c) :: true
Post-InsertText ((l, ri, _, _); r) :: r = (l + c, ri, [ ], [ ])

Pre-Copy (t) :: true
Post-Copy ((l, ri, s, cl); r) :: r = (l, ri, s, s)

Pre-Paste (t, cl) :: true
Post-Paste ((l, ri, s, cl); r) :: r = (l + cl, ri, [ ], cl)

Pre-DeletePreviousChar (t, l) :: true

Post-DeletePreviousChar ((l, ri, _, _); r) :: r = (init(l), ri, [ ], [ ])

Pre-DeleteNextChar (t, ri) :: true
Post-DeleteNextChar ((l, ri, _, _); r) :: r = (l, tail(ri), [ ], [ ])

Pre-DeleteAllLeft (t, l) :: true
Post-DeleteAllLeft ((l, ri, _, _); r) :: r = ([ ], ri, [ ], [ ])

Pre-DeleteAllRight (t, ri) :: true
Post-DeleteAllRight ((l, ri, _, _); r) :: r = (l, [ ], [ ], [ ])

Pre-Save
Post-Save

Pre-Load
Post-Load

Pre-Undo
Post-Undo

Pre-Redo
Post-Redo

AUTHOR

Gabriella Di Gregorio, 15624188@students.lincoln.ac.uk, April 2019

Haskell Code

```
--CMP2092M Programming Paradigms Assignment by Gabriella Di Gregorio 15624188, April 2019

--In Haskell, lists are a homogenous data structure. It stores several elements of the same
type. (Learnyouahaskell.com, 2011)
--Data.List must be imported in order to perform operations on lists.
--System.IO is the standard IO library and this must also be used to input and output to the
console as well as other files.
import Data.List
import System.IO

--The code operates on one variable only which will be the text/sentence input into the
editor
--This is initiated as four strings, one for the left of the cursor, one for the right, one
for selected text, and one for text that has been copied.
data Text = Text {left :: String, right :: String, selection :: String, clipboard :: String}
deriving (Show)

--This function simply sets the four strings to be empty to begin with. These can be changed
and operated upon.
create :: Text
create = Text [] [] [] []

--This function creates a more readable output to the console to make it clearer what each of
the other functions do.
--It prints the contents of the left string, a cursor (|), and the contents of the right hand
side.
display :: Text -> IO()
```

```
display t = putStrLn ((left t) ++ "|" ++ (right t))

--This is another function for the inteface, but this one clearly shows the user what is
currently selected.
--I decided to have two different functions for this since it looked either untidy or unclear
if this function was used every time, even when no selection has been made.
displaySelection :: Text -> IO()
displaySelection t = putStrLn ((left t) ++ "|" ++ (right t) ++ "\t\tYou have selected: " ++
(selection t))

--This function moves the cursor one to the left, so it will appear in front of the previous
character
--This behaves the same as pressing the left arrow key would on popular text editors
--Firstly it checks if you are able to move left, because if you are already at the start of
the sentence, this means the left string is currently empty so no more moves can be made to
the left.
--If you are already leftmost of the sentence, it will simply display the same again rather
than throwing an error.
--However, you are able to keep moving to the left until you reach the start, so more of the
text will be put into the right string as you move along.
--This is done by keeping all of the left except the last character in the left string, and
then concatenating the very last character of the left with right in the right string.
moveLeft :: Text -> Text
moveLeft t =
 if (left t) == []
  then t
 else (Text (init (left t)) ([last (left t)] ++ (right t)) [] [])

--This function behaves very similarly to the previous one, except this one is for moving one
character to the right.
--So this time, there must be contents in the right string in order to be able to move to the
right. This is why it first checks if the right string is empty.
--Again, if this string is empty it will simply display the same again rather than throwing
an error.
--You can keep moving to the right until the end of the text (when all of the text will be in
the left string) as one character will be added to the left string with each right move.
--This is done by combining the contents of the left with the first character of the right in
the left string, and putting the remainder of the right string (all but the first character)
into the right string.
moveRight :: Text -> Text
moveRight t =
 if (right t) == []
  then t
 else (Text ((left t) ++ [(head (right t))]) (tail (right t)) [] [])

--The next function allows the user to move the cursor to the start of the line of text.
--This is done by placing all of the text into the right string and emptying the left.
--However, if the left string is already empty then the cursor is already at the start so
this move cannot be made. If this is the case, the text is displayed in its current state
rather than throwing an error.
moveStart :: Text -> Text
moveStart t =
 if (left t) == []
  then t
 else (Text [] ((left t) ++ (right t)) [] [])

--Similar to the function above, this one moves the cursor to the other end of the sentence.
--This is done by combining the contents of the left and right into the left string and
emptying the right string.
--If the right string is already empty then the cursor is already at the end so this function
does not need to do anything, it simply prevents an error.
moveEnd :: Text -> Text
moveEnd t =
 if (right t) == []
  then t
 else (Text ((left t) ++ (right t)) [] [] [])
```

```
--This function selects all of the text left of the cursor, which means anything in the left
string.
--Whether the cursor is currently at the end of the sentence, in the middle, or after the
first word, anywhere it is will select what is left of it.
--This is done by copying the contents of the left string into the selection string.
--However, if the left string is empty then the cursor is at the start of the text so there
is nothing in the left string. This makes the function unable to operate so an error check
takes place.
selectLeft :: Text -> Text
selectLeft t =
 if (left t) == []
  then t
 else (Text [] ((left t) ++ (right t)) (left t) [])

--This function is the same idea as the previous one, except it selects everything to the
right of the cursor/in the right string
--If this function is displayed, the contents of the right string would be shown in the
selection string.
--This time, if the right string is empty then there will be nothing to select, so the if
statement prevents an error by returning the text in its current state.
selectRight :: Text -> Text
selectRight t =
 if (right t) == []
  then t
 else (Text ((left t) ++ (right t)) [] (right t) [])

--The copy and paste functions make use of the fourth string. I called it 'clipboard' as this
is what it is refered to by a very well-known text-editor, Microsoft Word
--A copy can only occur if something has been selected so this is checked in the if
statement. It sees whether the selection string is empty and if it is it simply returns the
text instead of allowing an error.
--If there is something in the selection string, this can be copied by also placing its
contents into the clipboard string.
copy :: Text -> Text
copy t =
 if (selection t) == []
  then t
 else (Text (left t) (right t) (selection t) (selection t))

--For a paste to work, it relies on the copy funtion being used first.
--So, it checks if the clipboard string is empty. If it is then nothing has been copied yet,
but instead of throwing an error it just returns the text.
--If there is something inside the clipboard string, then this can be pasted by copying its
contents into the left string (before the cursor)
paste :: Text -> Text
paste t =
 if (clipboard t) == []
  then t
 else (Text ((left t) ++ (clipboard t)) (right t) [] (clipboard t))

--This is a very simple function since it simply concatenates the contents of the left and
right strings into the selection string, so all of the text will be selected.
selectAllText :: Text -> Text
selectAllText t = Text (left t) (right t) ((left t) ++ (right t)) []

--This function puts one character to the left only into the selection string.
--Similar to the other left functions, it must see if the left string is currently empty
because if it is then there are no characters on the left to be selected. This would return
an error without this check in place.
--This is done fairly similarly to the moveLeft function in the way that all but the last
character of the left is kept in the left string, then the very last character of the left is
merged with the right into the right string.
--A selection is made by putting the very last character of the left into the selection
string, as well as any other selection previously made as several characters may be selected
one after another.
selectLeftLetter :: Text -> Text
selectLeftLetter t =
 if (left t) == []
```

```
  then t
 else (Text (init (left t)) ([last (left t)] ++ (right t)) ([last (left t)] ++ (selection t))
[])

--This function combines the logic of the function above with that of the moveRight function.
It selects a single character to the right of the cursor.
--As with the other right functions, it must check whether the right string is empty so it
can avoid an error if it is. If it is empty then no characters would be on the right to
select.
--Like moveRight, it puts the left and the very first character of the right into the left
string, and keeps the rest of the right in the right string.
--The first character of the right is stored in the selection after any others that have
already been selected.
selectRightLetter :: Text -> Text
selectRightLetter t =
 if (right t) == []
  then t
 else (Text ((left t) ++ [(head (right t))]) (tail (right t)) ((selection t) ++ [(head (right
t))]) [])

--This function jumps the cursor to the beginning of a word on the left
--Like the rest of the functions, the left must not be empty for this to be able to happen.
--Another if-else statement is needed if the left is not empty in order to check whether the
last character in the left is a blank space (" ") instead of a character as this would
indicate the start/end of a word.
--If it is already " " then it can be returned since it is already in between words
--However, this is a recursive function since it will keep looping through, moving one to the
left until a blank space has been reached.
moveFrontOfWordLeft :: Text -> Text
moveFrontOfWordLeft t =
 if (left t) == []
  then t
 else
  if ([last (left t)]) == " "
   then t
  else moveFrontOfWordLeft (moveLeft t)

--This is a similar idea to the previous function, but moves to the beginning of a word on
the right
--The usual check for the empty right string is made
--This time, the next check is to see whether the first character on the right is an empty
space (" ") to indicate the start/end of a word
--If it is, it is not quite as simple as the previous function since another moveRight needs
to be made to move the cursor to the start of the word on the right rather the end of the
word on the left
--This time, the recursion loops through moving one to the right until the " " is found
moveFrontOfWordRight :: Text -> Text
moveFrontOfWordRight t =
 if (right t) == []
  then t
 else
  if ([(head (right t))]) == " "
   then (moveRight t)
  else moveFrontOfWordRight (moveRight t)

--This function is almost identical to the moveFrontOfWordLeft function
--However, this time instead of looping through moveLeft, it repeats selectLeftLetter until
all characters up to a " " have been selected.
--The operation of this function was the main purpose of adding previously selected
characters into the selection in selectLeftLetter so that it could be used to select a word
rather than emptying after each selected character.
selectWordLeft :: Text -> Text
selectWordLeft t =
 if (left t) == []
  then t
 else
  if ([last (left t)]) == " "
   then t
```

```
  else selectWordLeft (selectLeftLetter t)

--This function is almost identical to moveFrontOfWordRight with the same logic as the
function above.
--For this one to work, the function is repeated, looping through selectRightLetter until a "
" is reached.
selectWordRight :: Text -> Text
selectWordRight t =
 if (right t) == []
  then t
 else
  if ([(head (right t))]) == " "
   then t
  else selectWordRight (selectRightLetter t)

--This function deletes all of the contents of the left string (deletes all text left of the
cursor)
--It does this by setting the left string to empty ([])
deleteLeft :: Text -> Text
deleteLeft t = Text [] (right t) [] []

--This one is similar, but deletes all right of the cursor by emptying the right string.
deleteRight :: Text -> Text
deleteRight t = Text (left t) [] [] []

--This function deletes a single character to the left of the cursor, like the backspace
button would
--In order for this to work, there must be something on the left to delete so the usual error
prevention has been made
--It operates by keeping only the first part of the left (except the very last character) in
the left string, and the right in the right string as usual
deleteLeftLetter :: Text -> Text
deleteLeftLetter t =
 if (left t) == []
  then t
 else (Text (init (left t)) (right t) [] [])

--This is very similar to the function above, but it deletes one character to the right of
the cursor just like the Delete key would
--This time, the left remains in the left string, and all of the right string except the very
first character remains in the right string
deleteRightLetter :: Text -> Text
deleteRightLetter t =
 if (right t) == []
  then t
 else (Text (left t) (tail (right t)) [] [])

--This function allows the user to input some text (of any length) where the cursor currently
is.
--Since it takes more information, we need to pass in String
--It works by adding an insertion string to the left string
--It is set in the main function that insertion is the user's input by using getLine.
insertInput :: Text -> String -> Text
insertInput t insertion = Text ((left t) ++ (insertion)) (right t) [] []

--This function saves the text (the contents of the left and right strings) into a txt file.
--Since it must take the name of the text file and output, it must take String and IO() as
well as the usual Text
--When this function is called in the main or the console, it must be given a file name (in
the form of a string) and the text that you would like to save.
--It combines the left and right strings so the whole sentence is saved regardless of cursor
location.
save :: String -> Text -> IO()
save nameOfFile t = writeFile nameOfFile ((left t) ++ (right t))


--While a main function is not needed for the program to work, this will display all the
functionality I have implemented in a clear and nicely displayed order and output.
```

```
--The first part of the video will show main called in GHCi alongside this code, so you can
pause the video here to see what each line/function does.
--However, I will also prove that the main is not needed by demonstrating a few functions on
a newly input text.
main :: IO()
main = do
--Put contents into the left string:
 let t = Text "Gabriella Di Gregorio 15624188" "" "" ""
 display t
--Save the text to a .txt file:
 save "text.txt" t
 putStrLn "Your file has been saved!"
--Move the cursor to the start of the sentence:
 let h = moveStart t
 display h
--Select the single character to the right of the cursor:
 let srl = selectRightLetter h
 displaySelection srl
--Move the cursor to the end of the sentence:
 let e = moveEnd srl
 display e
--Move the cursor one character to the left:
 let l = moveLeft e
 display l
--Move the cursor another character to the left:
 let ll = moveLeft l
 display ll
--Select the single character on the left-hand side of the cursor:
 let slc = selectLeftLetter ll
 displaySelection slc
--Copy the current selection:
 let c = copy slc
--Paste the selected character left of the cursor:
 let p = paste c
 display p
--Delete one character to the left of the cursor (the one that had just been pasted):
 let dlc = deleteLeftLetter p
 display dlc
--Move the cursor one character to the right:
 let r = moveRight dlc
 display r
--Select all text on the right-hand side of the cursor:
 let sr = selectRight r
 displaySelection sr
--Move in front of the word to the left:
 let lw = moveFrontOfWordLeft sr
 display lw
--Select everything to the left of the cursor:
 let sl = selectLeft lw
 displaySelection sl
--Move to the front of the next word on the right:
 let rw = moveFrontOfWordRight sl
 display rw
--Delete everything on the left-hand side of the cursor:
 let dl = deleteLeft rw
 display dl
--Prompts the user to input text then inserts it to the left of the cursor:
 putStr "Please type what you would like to add: "
 insertion <- getLine
 let i = insertInput dl insertion
 display i
--Select the next word on the right:
 let srw = selectWordRight i
 displaySelection srw
--Select all text, regardless of cursor position:
 let sa = selectAllText srw
 displaySelection sa
--Delete one character to the right of the cursor:
```

```
 let drl = deleteRightLetter sa
 display drl
--Delete everything on the right-hand side of the cursor:
 let dr = deleteRight drl
 display dr
--Select the next word on the left of th cursor:
 let slw = selectWordLeft dr
 displaySelection slw

--References:
--Learnyouahaskell.com. (2011). Starting Out - Learn You a Haskell for Great Good!. [online]
--Available at: http://learnyouahaskell.com/starting-out [Accessed 24 Apr. 2019].
--    This online book was also used to help me learn the basics of Haskell and write my
code.
```

Video Demonstration

https://youtu.be/lLIxf02fpoU