COMP3259: Principles of Programming Languages
Final Project

3035435668

A1.
Adding Unary and Binary operators:

The Unary operator can be pattern matched as **(Unary a op) environment memory**, where a is an expression and op is the unary operator. This expression is first evaluated to get the value of the expression and the updated state. It is then checked if this value is not an exception. Once checked, it is passed to a helper function that applies the specified unary operator to the correct evaluated value.

The Binary operator can be pattern matched as **(Binary a b op) environment memory**, where a and b are expressions and op is a binary operator. Expressions a and b are evaluated respectively, in order to accomplish left to right program evaluation. The updated state is passed to both the expressions appropriately. Only if both the expressions evaluate successfully, do we pass the expressions to a helper function that applies the correct binary function to the two values obtained from the evaluated value.

A2.
Local Variable Declarations and Conditionals:

The Local Variable Declaration can be pattern matched as **(Let x a b) environment memory**, where x is the Var and a and b are expressions. To declare a variable, expression a is evaluated and a value and updated memory is obtained from this evaluation. The evaluated value is then added to the environment for the function with the variable name x. This updated environment and updated memory are then used to evaluate expression b.

The Conditional can be pattern matched as **(If a b c) environment memory**, where all the parameters are expressions. The first expression is evaluated and an evaluated value and updated memory is obtained. The obtained value must be a VBool, if it is not then an error is thrown. If the value VBool is true then expression b is evaluated with the updated memory. Else c is evaluated with the updated memory.

A3.
Clone Expression:

The Clone can be pattern matched as **(Clone a) environment memory**, where a is the expression that the user wants cloned. We first evaluate the expression a and obtain a value and an updated memory. Since we can only clone Objects, we check if the value of the cloned

expression is an object reference. If it is not then we throw an error. Once its type is checked, then we access the memory with the Object Reference value. We then add this object to the memory and pass the length of the memory before adding the object as the object reference. In haskell, values do not contain state and are immutable. Thus, when we access the object from the reference, a new object is created and we can simply add it to the list.

A4.
The final part of the code uses a monadic approach to code the evaluator. This monadic approach not only uses the state monad but also transforms it with the MaybeT to allow for cleaner code and improved error handling.

The project is using the Monads taken from the Control library, namely it is using **Control.Monad.State**, **Control.Monad.Trans** and **Control.Monad.Trans.Maybe**

The monad defined is as **MaybeT (State Mem) Value**. MaybeT is a state transformer whose first parameter is another monad and the second parameter is the type to apply maybe to. Having used the monad transformer adds the power of Maybe to the State Monad.

Some auxiliary functions are defined as helpers. The auxiliary functions use functions described in the state monad. Namely they use, **get** and **put** which gets the current state and puts to the state respectively. Since these functions are defined for the State monad and not for the transformed MaybeT (State a) monad, they have to be lifted to the correct context. The **lift** function helps with this lifting of state.

The helper functions added are discussed in the paragraph.
newMemory is a function that takes as input an object to add to memory and returns the Object reference to the added object.
readMemory is a function that takes as its input an object reference value which it then uses to access memory and return the object at that memory index.
cloneMemory is a function that takes as its input an object reference. It then accesses the memory at that index to obtain the object. Furthermore, it updates the state with a clone of that object and returns the new index of the cloned object.
callMethod is a function that calls a method of a function. It takes as input an object reference and a label for the method to call. It then indexes the memory at that location to get the object. One the object is queried, it does a lookup on all the labels and if a label with the same name as that given as input is found then it calls that method with the evaluator.
updateObject is a function that takes an object reference, a method label to update and the closure to update the method to. It does an index lookup on the memory to obtain an object. Once the object is queried, a lookup is done on its labels. If a label as the supplied label is found then it is replaced with the closure passed as the method parameter.

Using the two monads in conjunction allows for succinctness and increased readability.

A5.
For this question only two constructs were modified in the source language. All the other Expressions were directly translated to their SigmaTerm counterparts. The two expressions modified were the lambda and the apply function.

The lambda expression allows for first class functions in the sigma calculus based language. A lambda function is interpreted as an object with two methods in lambda calculus. The first is the args method and the second is the val method. The args method is used to store the arguments of the lambda expression and is updated before the function is called. The args contain the logic for the function.

Discussing the lambda expression, we see that it is trivial to define the arg function. The function that stores the logic for the expressions is more exciting. We notice that when the logic is passed to the expression, it contains a variable v. This function must be called with the value of the variable defined. In our language. We must substitute the value of this variable with a call to the variable's arg method. This is handled by a helper substitute function.

The Apply expression calls a given lambda function. The pattern matching for the function is (**Apply b a) tc** where a and b are expressions and tc is storing the class definitions. The b expression represents the object that is to be called. As we are going to be updating the arg of this function before calling this function, we clone the expression and then translate it to convert it into a SimaTerm. The expression a denotes the variable that we would like to pass to the given expression. This variable is just simply translated to a SigmaTerm. The arg of the cloned object is then updated with the evaluated value for expression a. Finally the value method of the class is called.

A6.
In order to allow for classes in our sigma language, the language added the SNew and the Class expression type to the language. Both of these expressions take the help of the **classGen** function. The classGen function creates an object that changes all the original methods of an object into pre-methods. That is, the methods are updated to lambda functions. The second function is to create a new method that instantiates the object. This is done by converting each method into applying the lambda function to the object ref with a variable.

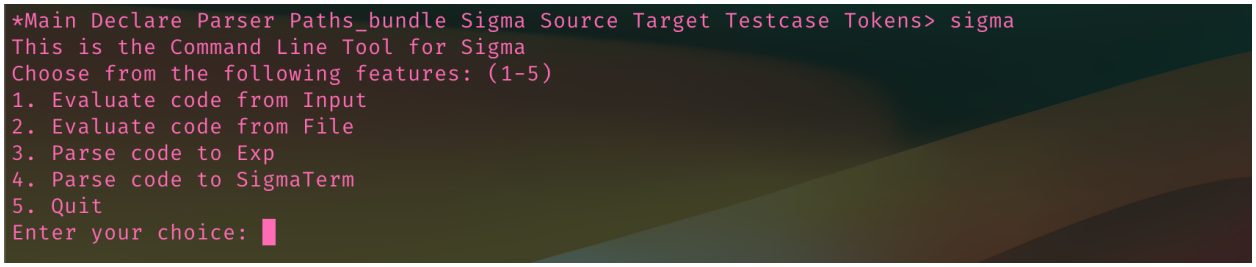
A7.
To allow for single class inheritance we modify the classGen function. If the classGen function is taking a variable as its input then that means that it is asking to inherit from the variables methods. There we first do a lookup of this variable in the class definition. If the variable exists in the class definition and is a correct class object, then it's methods are appended to the existing objects methods. This method was chosen instead of only keeping unique methods as a lookup function would only get the first value of the defined label. Therefore, if two labels are defined it will get the subclass defined label. Another reason for this decision is that running through method defined for both the lists and then choosing the unique method is $O(n^2)$. While a single lookup runs in $O(n)$.

In order to query to obtain a class definition from the list of class definitions, the class definition must be updated when a new class is discovered. Explicitly, when a variable is being created and that variable is a class, then we updated the class definitions.

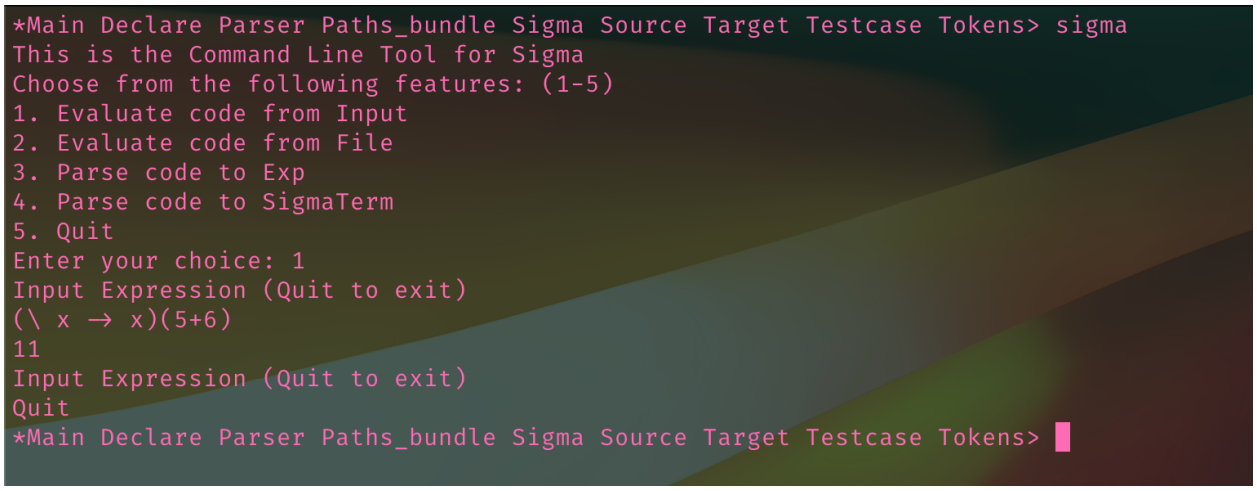
In order to allow overriding of methods to work, the super keyword had to be properly handled when a subclass uses the super method in the object context. A naive method is currently used in the interest of time. When a class is defined with an expression which is either a variable or a class definition then it is added to the class definitions as super. Later encountering a variable, we check if the variable is super, if it is super and it is a class, then we translate the class, else we change its type to a SigmaVar.

In order to allow for multiple objects to inherit and call super. Instead of appending super, one suggestion is to append super followed by the class name calling super. In order for this to work, we would have to substitute the super variable in all the classes with the super followed by the class name as well.

A8.
In addition to improved error handling and method overriding, a command line interface is also built. This command line interface is called Sigma and allows the user to 4 features at present. This can be seen from the image below



The first feature allows you to type in an expression and evaluate it. This is seen below



The second allows one to evaluate an expression from a file as shown below

```
*Main Declare Parser Paths_bundle Sigma Source Target Testcase Tokens> sigma
This is the Command Line Tool for Sigma
Choose from the following features: (1-5)
1. Evaluate code from Input
2. Evaluate code from File
3. Parse code to Exp
4. Parse code to SigmaTerm
5. Quit
Enter your choice: 2
Input File (Quit to exit)
test/testcases/gcell.obj
7
Input Expression (Quit to exit)
Quit
*Main Declare Parser Paths_bundle Sigma Source Target Testcase Tokens>
```

The third parses code to an expression

```
*Main Declare Parser Paths_bundle Sigma Source Target Testcase Tokens> sigma
This is the Command Line Tool for Sigma
Choose from the following features: (1-5)
1. Evaluate code from Input
2. Evaluate code from File
3. Parse code to Exp
4. Parse code to SigmaTerm
5. Quit
Enter your choice: 3
Input Expression to parse (Quit to exit)
(\ x → x)(5+6)
(\ x→x)((5+6))
Input Expression (Quit to exit)
```

The final option parses the code to a SigmaTerm

```
*Main Declare Parser Paths_bundle Sigma Source Target Testcase Tokens> sigma
This is the Command Line Tool for Sigma
Choose from the following features: (1-5)
1. Evaluate code from Input
2. Evaluate code from File
3. Parse code to Exp
4. Parse code to SigmaTerm
5. Quit
Enter your choice: 4
Input Expression to parse (Quit to exit)
(\ x → x)(5+6)
(clone([arg={x} x.arg,val={x} x.arg]).arg↞{this} (5+6)).val
Input Expression (Quit to exit)
Quit
*Main Declare Parser Paths_bundle Sigma Source Target Testcase Tokens>
```