COMP3259 PROJECT REPORT

Sigma Calculus Interpreter using Haskell

3035445974

In this report, I would describe and analyze the implementations of each of the implemented partial/complete functions, any helper functions, and/or any other code that I had to write for all three parts of the project. I will talk about each part in sequence starting from Question 1 and finally moving onto Question 8 where the actual interesting additional features have been implemented and also where, under the heading of **“Command Line Interpreter”**, the details have been provided on how to run the interpreter.

Compulsory parts *(part 1 and 2)*

Question 1:

The cases for unary and binary operators were straightforward, a simple extension to what we have been writing throughout this semester in lectures and in tutorials. I implemented two helper functions:

```
unary :: UnaryOp -> Value -> Value
```

The unary function just takes one of the two unary operators and a value and returns the resulting value after applying the unary operator to it. One extra feature added to this function that makes it standout from what we have been doing during the semester is the addition of the case where we check for the Type Error and returns a comprehensive error message, making our code both user-friendly and robust.

```
binary :: BinaryOp -> Value -> Value -> Value
```

Like the case of the unary function, the binary function takes one of the many binary operators and two values and returns the result after applying binary operator to the values. Again, what makes this function unique are descriptive error messages for the cases when one of two possible errors occur namely **DivisionByZero** and **Type Error** (when operand and operation types do not match). This again allows our code to be more user-friendly and allows it to be robust once we add the additional feature of **StatefulChecked** monads.

Now once we have these functions at hand, the rest of the implementation for the two cases is a direct consequence of what we did during the lecture 12.

For the unary case, we evaluate the given **SigmaTerm a** under the given memory and environment to obtain a new memory **mem’** and Value **v** and finally return **Just (unary op v, mem’).**

For the binary case, we evaluate first **SigmaTerm a** under the given memory **mem** and given environment to get a Value **v1** and the updated memory **mem’**, then we propagate the **mem’** and evaluate the second **SigmaTerm b** under the new memory **mem’**, obtaining another Value **v2** and an updated memory **mem’’**. Finally, we return **Just (binary op v1 v2, mem’’).**

Finally, for the pretty printer the implementation is straightforward, just showing the operands together with the operators with appropriate whitespace characters and brackets, where necessary.

Question 2:

Again, the implementations for local variable declarations, together with conditionals were direct extension of the code we wrote in lectures and tutorials.

For the variable declarations case:

```
evaluateS (Let x a b) e mem =  --q2
  let Just (ans, mem') = evaluateS a e mem
      newEnv = (x, ans) : e
      in evaluateS b newEnv mem'
```

I simply, first evaluate **a** under the current environment and memory to get a value and an updated memory **(ans,mem')**, then we added the tuple **(x, ans)** to the environment e and finally we evaluate **b** under the new environment and the memory **mem'**.

For the conditionals,

```
evaluateS (If a b c) e mem =
  case evaluateS a e mem of
    Just (VBool test, mem') -> evaluateS (if test then b else c) e mem'
    _                       -> error ("Type error: The expression:\n "  ++
                                      show a ++ "\n does not evaluate to a boolean!")
```

First evaluate **a** and make sure that it evaluates to a **VBool** value, otherwise throw a type error. Once evaluated to a **VBool** and updated memory **(VBool test, mem')**, depending on the value of test, we would evaluate **b** (true) or **c** (false) under the given environment and new memory **mem'**.

The pretty printer again is straightforward, with trivial implementations. Just **show** all the **SigmaTerm** and add whitespaces and brackets where appropriate. For example, **If (*show* x1) then *show* x2 else *show* x3.**

```
  show (If x1 x2 x3)    = "if (" ++ show x1 ++ ") then " ++ show x2 ++ " else " ++ show x3
  show (Let x e1 body) = "var " ++ show x ++ " = " ++ show e1 ++ "; " ++ show body
```

Question 3:

Now the Clone case was a bit complicated but following the instructions given in project instructions, the task became self-explanatory.

```
evaluateS (Clone a) e mem =
  case evaluateS a e mem of
    Just (ObjRef i, mem') -> -- created an object
      let newObject = access i mem' -- retrieve an object
      in  Just (ObjRef (length mem'), mem' ++ [newObject])
    _ -> error ("Type error: The expression:\n "  ++
                     show a ++ "\n does not evaluate to an object!")
```

In this case, we first evaluate expression **a**, and check whether **a** is actually an Object i.e. it returns an ObjRef (object reference). If it does not, we give a user-friendly and comprehensive error message, while if it does, we access the object from the updated memory **mem'** using the returned **ObjRef**. And finally, we return the **ObjRef** of the new object (the clone which is stored at the end of the memory), and the updated memory **mem'** with the new clone object added.

The pretty printer was again straightforward for this case, using **"clone (show c)".**

```
show (Clone c)        = "clone(" ++ show c ++ ")"
```

Question 4:

For this question I used **StatefulMonad**, same as the one used in lecture 12, however, in Question 8 as I will show that I have used a more complex Monad as one of the added features which combines this Stateful Monad with the Checked Monad used in Lecture 11.

Now the implementation is somewhat straightforward once you have a strong understanding and grasp of the concepts, functions and syntax involved in using Monads. This refactored **evaluateMonad** function performs exact same actions as the **evaluateS** but uses a tailored Monad syntax. What it simply does is, it completely encapsulates the memory management by automatically propagating memory at every step with the help of usual bind and return functions *(see lecture 12).*

For instance, the case for **Binary** is written as:

```
evaluateMonad (Binary op a b) e = do
  v1 <- evaluateMonad a e
  v2 <- evaluateMonad b e
  return (binary op v1 v2)
```

This case kind of explains the complete logic behind this function. For instance, in **evaluateS** we had to get the updated memory at each call of **evaluateS** and propagate it to the next step. However, with **StatefulMonad**, all we have to do is just store the resulting **Value** after evaluating an expression and the memory propagation is done automatically under the hood. Finally, since, **binary op v1 v2** returns a **Value,** we use the usual **return** function to transform the **Value** into a **StatefulMonad Value**.

With similar reasoning, other cases can be understood. For instance, for the Clone case:

```
evaluateMonad (Clone a) e = do
  objRef <- evaluateMonad a e
  case objRef of
    ObjRef i -> do
                obj <- readMemory i
                newMemory obj
    _        -> error ("Type error: The expression:\n " ++
                     show a ++ "\n does not evaluate to an object!")
```

First evaluate **a** under the environment (and memory, implicitly) and check whether the result is an **ObjRef**, if not throw a type error. If no error, simply, read memory at index **i** (using **readMemory** function) and extend the memory by adding the new clone object to the memory by using **newMemory** helper function. Notice, how the code is simplified with the use of **StatefulMonad,** as it automatically propagates memory.

Lastly, some helper functions for **evaluateMonad** had to be implemented for the cases where the memory needs to be extended, read or updated (by updating existing objects). Functions help to encapsulates memory management even in the situations when we have to manipulate or access memory *(referred to lecture 12 for help)*.

To read from the memory (replica of the access function), simply return a **Stateful Object** by taking an object reference **i** and returning **ST (\mem -> (access i mem, mem))**:

```
readMemory :: Int -> Stateful Object
readMemory i = ST (\mem-> (access i mem, mem))
```

To extend memory (when a new object needs to be added):

```
newMemory :: Object -> Stateful Value
newMemory obj = ST (\mem -> (ObjRef (length mem), mem ++ [obj]))
```

To update memory (replica of the provided **replace** function):

```
updateMemory :: Int -> Label -> MethodClosure -> Stateful Value
updateMemory i l closure = ST(\mem -> let (before, o : after) = splitAt i mem in
                                      (ObjRef i, before ++ [[if (l == n) then (n, closure)
                                                   else (n,c) | (n,c) <- o]] ++ after))
```

Question 5:

The **translate** function except for a few cases, was direct translation of cases from **Exp** to **SigmaTerm**. For **translate** function we use a helper function **transHelp** which does all the work and takes one extra integer argument.

Now, for **SLit**, **SVar** and **SBool** cases, the translation is straightforward translation of the Exp constructs to SigmaTerm constructs. For the cases **SUnary**, **SBin**, **SIf**, **SClone**, **SCall** and **SUpdate** the cases just translate each Exp constructor to corresponding constructor used in **SigmaTerm** and recursively calls **transHelp** on any **Exp**. For example, see recursive calls in case of **Exp e1** and **e2** below:

```
transHelp (SBin op e1 e2) tc xs = Binary op (transHelp e1 tc xs) (transHelp e2 tc xs)
```

For **SLet**, we first check whether **e1** evaluates to **Class**, if it does, we update **tc** (TClass) and call **transHelp** on **e2** under the new **tc'.** If it does not, then we just directly translate it into **SigmaTerm** using recursive calls for **transHelp** on **e1** and **e2**.

```
transHelp (SLet v e1 e2) tc xs =
  case e1 of
    Class _ _ -> let tc' = (v, e1) : tc in transHelp e2 tc' xs
    _         -> Let v (transHelp e1 tc xs) (transHelp e2 tc xs)
```

For **Lam,** the implementation was direct translation of the definition (pseudo-code) given in the project description (used the **substitue1** auxiliary function to perform the substitution explained in the project description). Similarly, a direct application of the pseudo-code given in the project description resulted in the code for the remaining two cases namely **Apply** and **SNew**. Finally, for **SObject**, I used list comprehension to loop through each tuple, **(l, SMethod v exp),** 1 by 1 and calling **transHelp** on the method's body **exp** for each tuple.

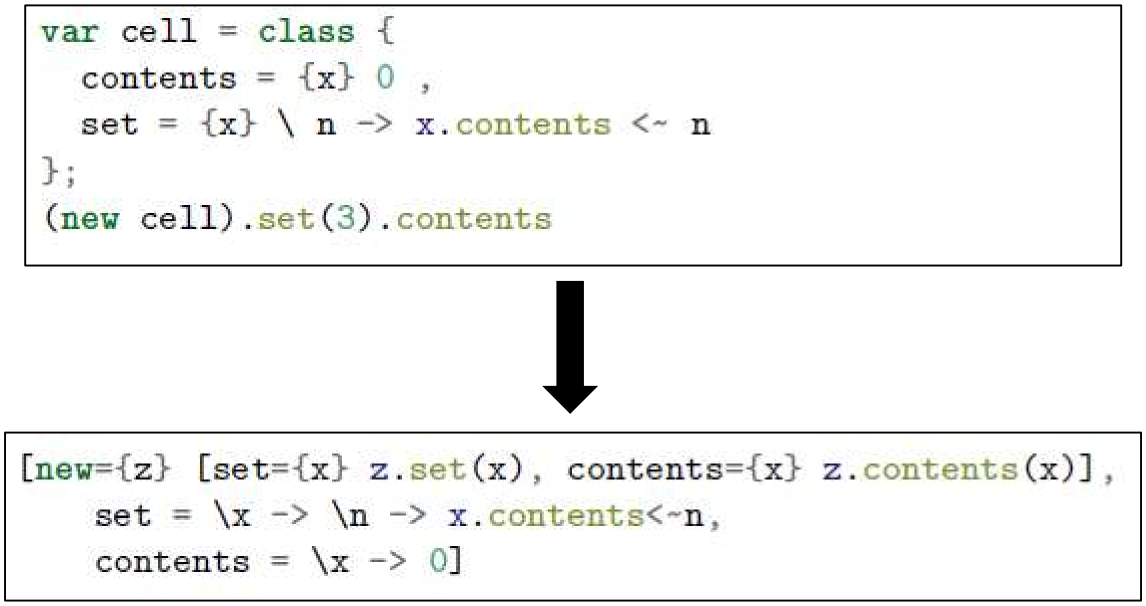
Lastly, we used an extra argument of **Int** for **transHelp**, just to make sure that no two references to the objects overlapped.

The pretty printer part was straightforward application of what we had written for **SigmaTerm** in previous questions and the cases that we were already provided in **Declare.hs** – that came with the project bundle.

Question 6:

As explained in the project description file, the main aim behind this part is to use a definition of the class to instantiate new objects or instances of the class. The strategy that we use aim to convert a class into an object-like structure so that instantiation of objects is possible in our language using the **new** operation.

We first build a **new object** that contains all the class's methods (in lambda form) as well as a method labelled **“new”.** The body of the **new** method contains another object that contains all the methods originally contained within the class with each method’s body using reference **‘z’** for the outer object (the one that contains **new**) to call the methods that are again applied with self-reference **{x}** as an argument. As mentioned in the project description file, here the reference **‘z’** references the outer object (the one that contains the **new** method). Basically, the class cell in the expression is transformed as below:



```
var cell = class {
  contents = {x} 0 ,
  set = {x} \ n -> x.contents <~ n
};
(new cell).set(3).contents
```

```
[new={z} [set={x} z.set(x), contents={x} z.contents(x)],
    set = \x -> \n -> x.contents<~n,
    contents = \x -> 0]
```

Lastly, list comprehensions were used to iterate through the list of methods in the class and transform the class into an object required so that new objects can be instantiated from the class using the **new** operation.

Question 7:

In order to incorporate single inheritance in our design, we first check whether the **new** operation was called on a class or a variable that references a class. If it is indeed a variable, we lookup the corresponding expression from the **TClass tc** and recursively apply **classGen**.

After obtaining a **Class Exp** in the form **(Class cMethods exp),** we perform case analysis on the **exp** which represents whether the fetched class is a top class or a child class. If it is a **Top** class, we follow the methodology described in **Question 6.** However, if it is a **Class** or a **SVar,** in this case our target class extends this class. We apply the **ClassGen** to this class (parent) and obtain **SObject cMethsTemp** an object-like structure for the parent class.

Now, we remove the method labelled **new** from the **SObject cMethsTemp,** and obtain a new list of (label, method) named **c2**. Then we let **cNew** equal the target class's methods (represented by **cMethods**) with the map function applied to each method.

The map function applies the **doItSuper** function to each method in **cMethods. doItSuper** method deals with the case when there is a reference to the **"super"** within the method body. First it performs case analysis on the method body to check whether it has a reference to **"super"**. If it has a reference to **"super"**, it replaces that part of the body (where the reference to super is and the method is called or applied) with the method found in the parent class (which is in the object form). If no reference to super is found, the (label, method) is returned unchanged.

```
doItSuper :: (Label, SMethod) -> [(Label, SMethod)] -> (Label, SMethod)
```

(A practical and common assumption has been made for Question7, that we can only use super.<method> as the first statement in the method as is the case with some other programming languages like Java).

Next, we let **listD** equal the methods within the parent class that are not overridden within the child class (only the inherited methods) using **checkClass** function for filtering. This function acts as a helper function for filtering out all the methods from the parent class that have been overridden.

```
checkClass ::  [(Label, SMethod)] -> (Label, SMethod) -> Bool
```

Finally, we concatenate **cNew** with **listD,** to get **newL,** and then except a little alteration, we use the same approach as used for the Top case in **Question 6** to process the class and transform into an object used for instantiating new instances / objects of the class.

Additional features:

Now that we are done with the explanation and analysis of the compulsory features, we will now turn our attention towards the additional features that the interpreter has – making the interpreter more robust, programmer-friendly and efficient.

I have been able to successfully implement the following three additional features:

1. Improved error handling with **StatefulChecked Monad**
2. **Command Line Interpreter**
3. **Evaluation By Need**

We will introduce and talk about each one of them one by one starting with the **StatefulChecked Monad**.

StatefulChecked Monad *(implementation src/TargetMonads.hs)*

Disclaimer: *Please note the name for this function and the function used for Q4 are the same but have been implemented in different files with different functionalities.*

Implementing this feature was not as difficult as it was to understand the functioning of as well as gaining an in-depth knowledge of Monads. For that I referred to the paper written by **Philip Wadler, University of Glasgow**, titled "*Monads for functional programming*". Afterwards, I spent multiple hours understanding the Haskell's Monad and Show class through the help of Lecture 11 and 12, provided Haskell documentation and online discussion forums.

Finally, I was able to create suitable datatype and monad instance:

```haskell
instance Monad (StatefulChecked) where
  -- return :: a -> StatefulChecked a
  return val = ST (\m -> Good (val, m))
  -- (>>=) :: StatefulChecked a -> (a -> StatefulChecked b) -> StatefulChecked b
  (ST c) >>= f =
    ST (\m ->
        case c m of
            Good x    ->
              let (val, m') = x in
                let ST f' = f val in
                    f' m'
            Error msg -> (\m -> Error msg) m
      )

data StatefulChecked a = ST (Mem -> Checked (a, Mem))
```

The monad named **StatefulChecked** is very much similar to the **StatefulMonad** that I implemented in Question 4 as well as to the one used in Lecture 12 except that instead of having **Mem -> (a, Mem)**, the monad takes a memory **Mem** and returns a Checked tuple namely **Checked (a, Mem)**, where Checked is the same datatype defined in Lecture 11.

For this **StatefulChecked Monad**, the **return** function just takes an element **val** and converts it into **StatefulChecked val**, basically combining what Stateful Monad and Checked Monad's return functions do.

Moreover, the **bind** function takes a **StatefulChecked a** and a function **f** *(a -> StatefulChecked b)*, and returns a **StatefulChecked b**. Given memory **m**, it does case analysis on **c** under **m**, if an error results, it propagates the error otherwise it computes **val**, and the new memory **m'** from the result of the case analysis on **c**. Then it applies the function **f** to **val** and returns its result together with the new memory **m'**.

Once, we have defined the datatype of **StatefulChecked** and defined the corresponding **return** and **bind** functions, I defined a few more auxiliary functions to help with the implementation of the main evaluation function.

First of all, I defined **chekedReturn,** that simply takes a **Checked Value** and returns **StatefulChecked Value.** For instance, if the **Checked Value** is of the form **Good val** then it outputs **ST (\m -> Good (val, m))** where **m** is the encapsulated memory. Otherwise, if **Checked Value** is of the form **Error msg** then it simply propagates the error message by outputting **ST (\m -> Error msg)**.

```
checkedReturn :: Checked Value -> StatefulChecked Value
```

Moreover, we had to convert our **binary** and **unary** functions (previously defined in Question 1) to **binaryChecked** and **unaryChecked** respectively. Instead of returning a **Value**, the checked functions return a **Checked Value** and instead of throwing an exception using **error** keyword in Haskell, these functions make use of the Checked datatype and instead returns **Error msg**. Again, these functions include comprehensive error checking, and provides thoughtful and meaningful error messages, if the error is encountered – making the interpreter programmer-friendly.

```
unaryChecked :: UnaryOp -> Value -> Checked Value
binaryChecked :: BinaryOp -> Value -> Value -> Checked Value
```

The two other helper functions used with **evaluateMonad** function in the file **TargetMonads.hs** are **condChecked** and **objectChecked**. **condChecked** basically takes a **Value**, and checks whether the Value is of **VBool cond** kind. If it is, it returns the **Good cond**, otherwise it returns **Error msg**. This function is used for the conditional case, when the first expression must evaluate to a **VBool**. Moreover, the function **objectChecked**, takes a **Value**, and checks to see whether the **Value** is of kind **ObjRef i.** If it is, it returns **Good i** otherwise returns an **Error msg**. This function is used in the **Call**, **Clone** and **Update** cases to check whether a value evaluates to an **ObjRef** or not.

```
condChecked :: Value -> Checked Bool
objectChecked :: Value -> Checked Int
```

Finally, the last three helper functions are simply the ones used in Question 4 with **StatefulChecked** return type instead of simply **Stateful**. All the functions perform the same tasks as before but instead of returning something of the form:

ST (\mem -> (XXX, mem), they return **ST (\mem -> Good (XXX, mem)**, where **mem** stands for memory.

Now comes the main **evaluateMonad** function where the memory is encapsulated and is automatically propagated together with the error. Lets explain it on a case-by-case basis:

For the case **SigmaVar v**, we simply lookup **v** in the environment, if found, we apply **return** to the found **Value**, to transform it into **StatefulChecked Value**. Otherwise, we apply **checkedReturn** to **Error msg**, to propagate the error message**.**

The case for **Object o** is straightforward, just extend the memory by adding the Object in the memory using **newMemory** function.

For the **Call a l** case, we first evaluate **a** and get a **Value.** We then check whether the **Value** is an **ObjRef** using **objectChecked** function, and if not an **ObjRef** we **checkedReturn (Error msg),** otherwise we read memory to get the corresponding object **ms** to the **ObjRef,** using **readMemory** function. Lastly, we **lookup l** in the object **ms** and call the function, if found, otherwise, we propagate the error using **checkedReturn (Error msg).**

For the case of **Let x a b,** we first evaluate **a** and get a Value **v.** Then we store **(x, v)** in the environment and finally evaluate **b** under the new updated environment. As you can see, the memory and any errors are automatically propagated and are dealt with, implicitly, using monads.

For the case of **Clone a,** we first evaluate **a** under the environment **e** and check the result using **objectChecked** function. If an **ObjRef** is returned, we **readMemory** to get the corresponding object **obj.** Finally, we extend the memory by calling **newMemory**. If object reference is not returned, we propagate **Error msg** by using **checkedReturn (Error msg).**

For the case of **Lit a** and **Boolean a**, we just apply **return** to the corresponding **Value(s).**

For the case of **Update a l (Method v m),** we first evaluate **a** under the current environment **e**, and check whether the Value obtained is an **ObjRef.** If it is, we update memory using **updateMemory** function otherwise we propagate **Error msg** using **checkedReturn (Error msg).**

For the **Binary op a b** and **Unary op a** case, we first evaluate the given expressions one by one to get the corresponding values and finally apply **checkedReturn (binaryChecked op v1 v2)** or **checkedReturn (unaryChecked op v)** to turn the **Checked Value** returned by **binaryChecked** and **unaryChecked** to **StatefulChecked Value.**

Lastly for the **If a b c** case, we first evaluate **a** and check whether it results in a **Checked cond'** form with the help of **condChecked** function. If it does, then we evaluate **b** or **c** depending on the value of **cond'** (true or false respectively). Otherwise, if it does not evaluate to a **VBool** we propagate **Error msg** using **checkedReturn (Error msg).**

After completing this **evaluateMonad** function, I finally linked this function to the command line interpreter by defining a tailored execute function. For that I was required to define **Show** instance for **StatefulChecked:**

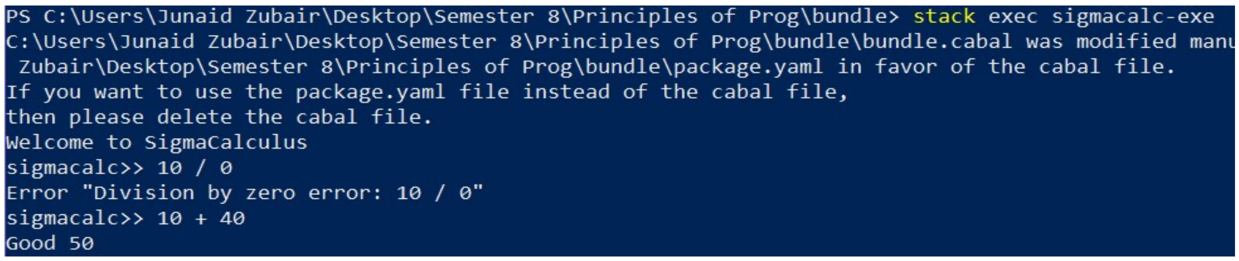
```haskell
instance Show a => Show (StatefulChecked a) where

   show (ST f) =
     case (f []) of
       -- x -> show x
       Good (a, mem) -> show a
       Error msg     -> msg
```

Which just applies the **f** in **ST f,** to empty memory to get a corresponding **Checked (a, mem)**. If **Good (a, mem)** is returned we just **show a** otherwise we just output the **msg** String in **Error msg.**

Command Line Interpreter *(Implementation: app/Main.hs)*

Implementation of this additional feature was inspired from the interpreter for our **SINH language** – first used in tutorial5 and then used extensively in assignment3 for debugging and testing my implementations. Motivated by this, I decided that this is an important feature that needs to be implemented to make our Sigma Calculus more user-friendly by making it easy to run, test and debug programs written in the language.

The source code for the interpreter is present in app/Main.hs file and the command line interpreter could be started using the command: ***stack exec sigmacalc-exe***


```
PS C:\Users\Junaid Zubair\Desktop\Semester 8\Principles of Prog\bundle> stack exec sigmacalc-exe
C:\Users\Junaid Zubair\Desktop\Semester 8\Principles of Prog\bundle\bundle.cabal was modified manu
 Zubair\Desktop\Semester 8\Principles of Prog\bundle\package.yaml in favor of the cabal file.
If you want to use the package.yaml file instead of the cabal file,
then please delete the cabal file.
Welcome to SigmaCalculus
sigmacalc>> 10 / 0
Error "Division by zero error: 10 / 0"
sigmacalc>> 10 + 40
Good 50
```

To run a file written in our Sigma Calculus language without entering the shell interpreter, the command *stack exec sigmacalc-exe <filename>* could be used. See below for examples*. Moreover, shortcut ways for exiting the interpreter, executing a file from the interpreter, and writing multi-line expressions within the interpreter using the newly added feature in the Haskell library (*repline*) have also been implemented that not only makes our interpreter easy to use, robust, and user-friendly but also gives it a professional outlook.

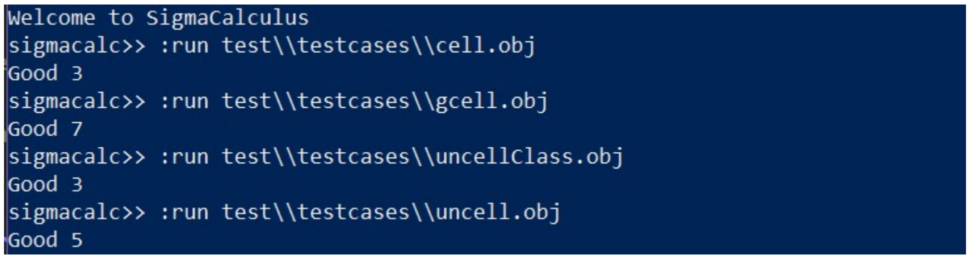
The interpreter uses the **evaluateMonad** function in TargetMonads.hs as its underlying evaluator.

Other Standout Features of Interpreter

To get the most out of this interpreter, three unique, yet very convenient commands have been made available:

1. ***:run <filename>***

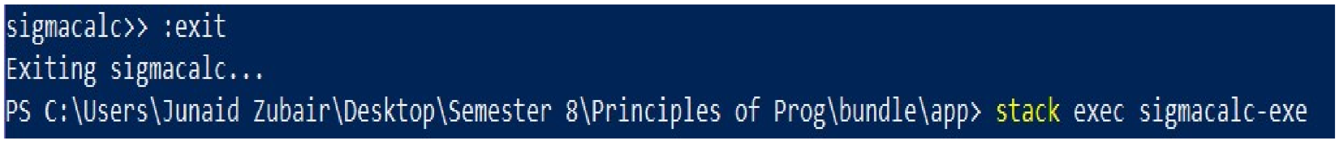
   This command executes the file with name filename containing the expression of our Sigma Calculus language.



```
Welcome to SigmaCalculus
sigmacalc>> :run test\\testcases\\cell.obj
Good 3
sigmacalc>> :run test\\testcases\\gcell.obj
Good 7
sigmacalc>> :run test\\testcases\\uncellClass.obj
Good 3
sigmacalc>> :run test\\testcases\\uncell.obj
Good 5
```

2. **:exit**

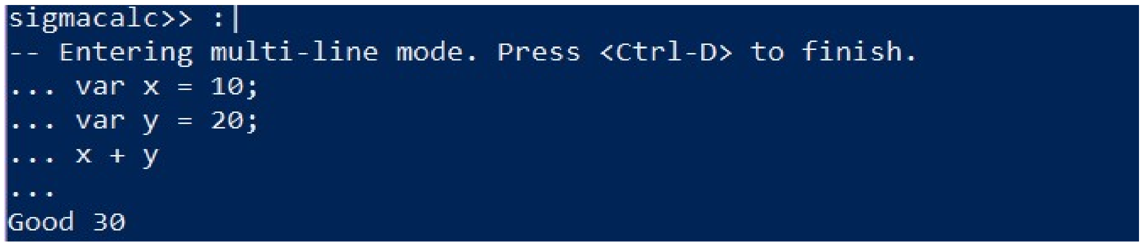
   This command allows the user to simply exit the interpreter without having to close the command line.



```
sigmacalc>> :exit
Exiting sigmacalc...
PS C:\Users\Junaid Zubair\Desktop\Semester 8\Principles of Prog\bundle\app> stack exec sigmacalc-exe
```

3. **:|**

   (explanation: write your code/expression on multiple lines, once done press enter and type Ctrl-D to execute) This command takes the user into a multi-line mode where a user can write code on multiple lines and execute it rather than writing only single line expressions which clutters the code and hence, makes it difficult for the programmer to even test a code spanning only a few lines.



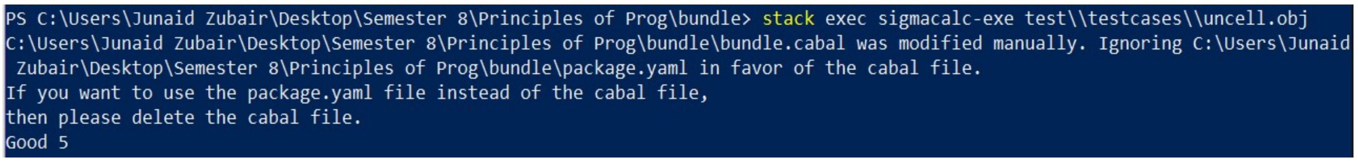
```
sigmacalc>> :|
-- Entering multi-line mode. Press <Ctrl-D> to finish.
... var x = 10;
... var y = 20;
... x + y
...
Good 30
```

Auto Completer

Auto completion functionality have also been introduced into our interpreter to further enhance the user-friendliness and one's programming experience. For instance, typing only ***:e*** or ***:r***, would automatically complete the whole command to ***:exit*** or ***:run*** (respectively) upon entering the tab key on your keyboard. Similar, functionality have been introduced for filenames. For example, writing ***:run t,*** and pressing tab would complete ***:run test\\*** (depending on the files and directories in your current folder). Similarly, ***stack exec sigmacalc-exe test\\testcases\\u*** becomes ***stack exec sigmacalc-exe test\\testcases\\uncell.obj*** upon entering tab.

Summary of available commands (run commands in bundle directory):

- *stack build*
  - For building the whole project stack
- *stack exec sigmacalc-exe*
  - Command for entering the command line interpreter
- *:exit*
  - Command for exiting the command line interpreter
- *:run <filename>*
  - Command for executing the file named filename containing the Sigma Calculus expression within the interpreter. For example, ***:run test\\testcases\\uncell.obj***
- :|
  - Command for entering the multiline mode and writing expressions/code on multiple lines within the interpreter. Once done writing the expression, type **Ctrl-D** to exit the mode and subsequently execute the expression.
- *stack exec sigmacalc-exe <filename>*
  - Command for executing the file filename containing the Sigma Calculus expression without entering the command line interpreter. For instance, ***stack exec sigmacalc-exe test\\testcases\\uncell.obj*** *



```
PS C:\Users\Junaid Zubair\Desktop\Semester 8\Principles of Prog\bundle> stack exec sigmacalc-exe test\\testcases\\uncell.obj
C:\Users\Junaid Zubair\Desktop\Semester 8\Principles of Prog\bundle\bundle.cabal was modified manually. Ignoring C:\Users\Junaid
 Zubair\Desktop\Semester 8\Principles of Prog\bundle\package.yaml in favor of the cabal file.
If you want to use the package.yaml file instead of the cabal file,
then please delete the cabal file.
Good 5
```

Evaluation by Need *(Implementation: src/Target.hs ~ evaluateByNeed)*

What is the need of a programming language, if it cannot provide intelligent and faster execution in today's tech-centered world? Partly inspired by this idea, and partly inspired by assignment3, I decided to introduce an evaluator that performs evaluation based on need. When performing an evaluation of an expression using our previous evaluators, every time a variable declaration is encountered, we evaluate the expression, obtain a value and store it in the environment. Even if that newly declared variable is never used in the program.

For instance, see the expression *var x = long computation; 10*, without **evaluateByNeed**, our evaluator would compute the long computation first, would update the environment only to find out the result is 10.

This brings undesired inefficiency in computation. To get rid of this, we have introduced an **evaluateByNeed** that only evaluates an expression of a variable when actually needed. For instance, in the previous example our new function would save the variable together with a closure containing "long computation" in the environment and would simply return 10 – never having to perform the *long computation*.

Now to implement this we simply added a **ClosureX** constructor in the Value datatype (similar to what was used in assignment 3), of the form **ClosureX SigmaTerm Env [[(Label, MethodClosure)]] MethodClosure)]]).** Now at every declaration of a variable, instead of evaluating the expression, we directly construct a closure (ClosureX) and store it directly in the environment together with the variable name.

Apart from propagating the updated environment at each step, the rest of the cases, except for the cases of **SigmaVar** and **Let,** are the same as **evaluateS**.

For the **Let x a b** case, we simply transform **a** into a **ClosureX** and store it in the environment and evaluate **b** under the new environment. For the **SigmaVar** case, a helper function namely **updateEnv** was used, that takes a new environment with the closure replaced by a Value (using **updateContext**), the environment that was returned after evaluating the closure and the index of the variable whose closure was evaluated. It returns the updated environment with the closure replaced by the computed Value *(as was done in assignment 3).* And then carries on with the straightforward computation.

For testing / running the **evaluateByNeed**, a testing function has been implemented in **src/Testcase.hs** namely **testEvalByNeed** which takes a **SigmaTerm** expression in String format and outputs the result. One can simply compile Testcase.hs using *ghci Testcase.hs* and then test the function using the command:

- *testEvalByNeed <string representation of a sigma expression>*

Lastly, a testEval function for testing **evaluateS** has also been provided in the same file and can be called like testEvalByNeed:

- *testEval <string representation of a sigma expression>*

Can also use execute function in Target.hs for testing **evaluateS**.

--