CS2006 Haskell 2

Scripting Language - Group Report

190001741, 200007779, 190022658

Overview

The objective of this project is to implement a small scripting language, as laid out in the project specifications. These were implemented:

Basic Requirements

- [x] Quit command
- [x] Tracking system State
- [x] Extend parser to support multi-digit numbers and whitespace around operators
- [x] String concatenation
- [x] Implement conversion between string to int functionality
- [x] Implement command to read user input as string functionality

Easy Requirements

- [x] Implement Quickcheck
- [x] Implement functionality for abs, mod, and power
- [x] Extend parser to support negative numbers
- [x] Support floats

Medium Requirements

- [x] Implement binary search tree
- [x] Add command to read and process input files with commands
- [x] Implement better treatment of errors using Either type
- [x] Implement functionality for if...then...else

Hard Requirements

- [x] Add Haskeline for command history
  - [x] Implement tab completion for commands and variable names
- [x] Add command for simple repetition
- [x] Implement functionality for loop constructs
- [x] Allow defining and calling functions

Team Organisation

In terms of team organisation, we had a Microsoft Teams group where we discussed and picked out new tasks and targets, which were then added to the README to keep track of progress. Otherwise we would assign tasks to ourselves and complete them, and intermediate progress reports could be added to the chat to further break down the work if needed, or avoid two people doing the same thing. The provided submission is a Git repository, so precise breakdown of lines of code written can be found by running `git log` or `git blame` (note though that commit identities have not been anonymised -- they contain names and email addresses instead of matriculation numbers).

Instructions

Instructions for running the language's repl can be found in `README.md`.

Design

Haskeline

Haskeline is used for input and output in the program. This means that control keys like backspace work correctly (unlike in standard Haskell `IO` in GHCi), and the up and down arrows can be used to browse the history.
To implement auto completion, we need a completion function which takes the current word to the left of the cursor and returns a completion list. In our case we have a word list containing all predefined keywords in the scripting language, then use "isPrefixOf" to find a list of words that has the argument as their prefix.
To implement auto completion of variables, we used StateT to contain State type, so that the completion function could get the state using "get", and retrieve word list from it. The word list stores predefined keywords as well as variable names the user set. Each time the user sets a new variable, the word list will be updated.
We changed the design and used StateT to wrap repl, in each loop we retrieve the state and process the command, then we replace it with the processed new state.

Parsing

We extended the parser from *Parsing.hs* and made *LangParser.hs* to parse the language commands and expressions. Here, there are Parser functions for Statements - Import, Print, While, Conditional, Assignment, Quit, Function call, Function definition and Expressions - Numeric, String, and Boolean. There are two kinds of function calls, one is FunCall that returns a value, and the other is VoidFunCall that does not return a value. This was a design choice that had to be made because the function blocks execute in REPL.hs and therefore they cannot be used as "expressions" that can be evaluated (i.e. a function that returns a value can not directly be added to a number). They are similar to the input command in this aspect. Functions that don't return any value can be called as a command. This was another reason for the design choice. This disables the user to call functions that don't return a value while assignment/print.

User Input

Since the input functionality needs to use IO, this was implemented separately than all the other expressions (which were implemented in the eval function, Expr.hs). Since this was implemented in REPL.hs, eval functions don't apply to it and therefore you cannot do string operations on them directly for example

```
> print(input ++ " i)
Parse error
```

You can however do this

```
> a = input
Input > John
> print a ++ " Doe"
"John Doe"
```

Or this

```
> print input
Input > John Doe
John Doe
```

Errors

The evaluation error is defined as shown below:
data EvalError = ExprErr ExprName ErrMsg
Where ExprName and ErrMsg are strings, ExprName represents the name of the expression that causes the error when evaluating, and ErrMsg represents the error message.
We use "Either EvalError Value" to replace Maybe Value, so that we could not only return the expression value if evaluation succeeds, but also return the error information if the evaluation fails. When the error information is returned, we use "outputStrLn" to print "Error on <ExprName>: <ErrMsg>".

State

The state had to store all the data necessary to reuse them. We ended up having:
1. **BTree** for variables and their values

2. **List of Tuples** for functions, their arguments, and their body
3. **List** of words for autocompletion
4. **List** of commands to implement import in a neat way

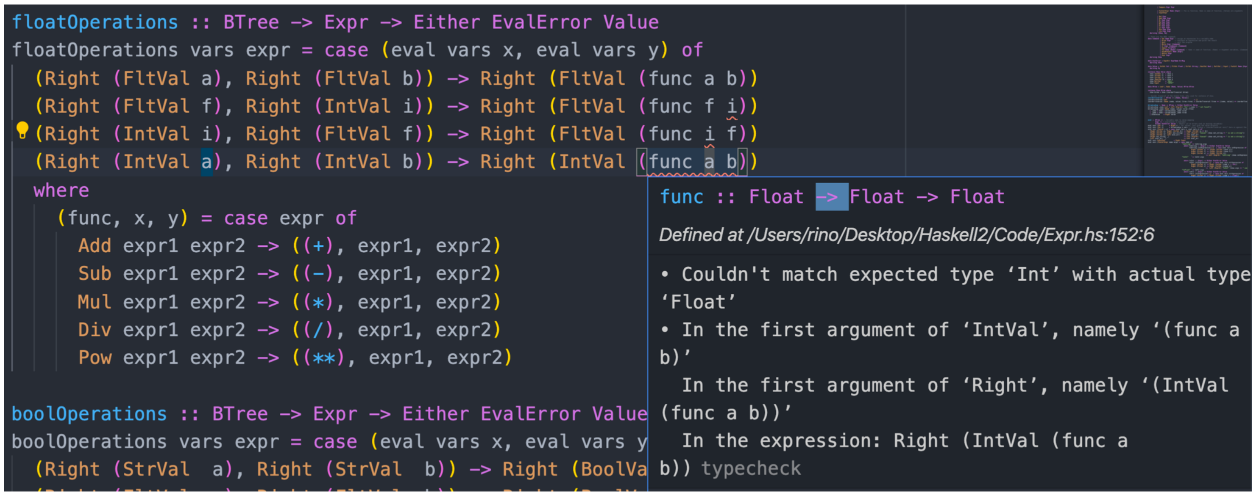
The initial state contains the list of words which contain all the keywords, it also contains some premade functions (you can find out which ones by pressing tab in the repl).

Process

Process originally returned "IO ()" and then "InputT IO ()" but to make it compatible with our needs we converted it to return "InputT IO (State)". This is a design choice that makes it easier to run blocks of code. This subsequently helped in the import command. We finally changed to using StateT, this was a lot of change in the design, but it was worth it. The uses are described in the Haskeline section.

floatOperations and boolOperations

In principle, built-in comparison operators and arithmetic operators in haskell work on multiple types such as floats, integers, boolean values and strings. So when we were trying to apply the same operators on different types we may encounter many repeated codes. We first came up with the solution as shown below:

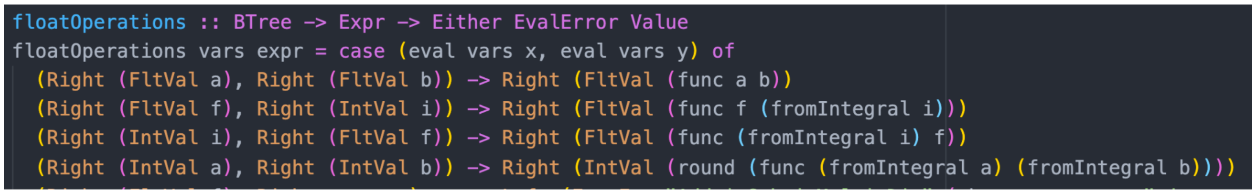


```
floatOperations :: BTree -> Expr -> Either EvalError Value
floatOperations vars expr = case (eval vars x, eval vars y) of
  (Right (FltVal a), Right (FltVal b)) -> Right (FltVal (func a b))
  (Right (FltVal f), Right (IntVal i)) -> Right (FltVal (func f i))
  (Right (IntVal i), Right (FltVal f)) -> Right (FltVal (func i f))
  (Right (IntVal a), Right (IntVal b)) -> Right (IntVal (func a b))
  where
    (func, x, y) = case expr of
      Add expr1 expr2 -> ((+), expr1, expr2)
      Sub expr1 expr2 -> ((-), expr1, expr2)
      Mul expr1 expr2 -> ((*), expr1, expr2)
      Div expr1 expr2 -> ((/), expr1, expr2)
      Pow expr1 expr2 -> ((**), expr1, expr2)

boolOperations :: BTree -> Expr -> Either EvalError Value
boolOperations vars expr = case (eval vars x, eval vars y
  (Right (StrVal  a), Right (StrVal  b)) -> Right (BoolVa
```

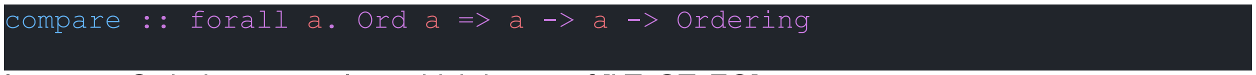
```
func :: Float -> Float -> Float
Defined at /Users/rino/Desktop/Haskell2/Code/Expr.hs:152:6
• Couldn't match expected type ‘Int’ with actual type
‘Float’
• In the first argument of ‘IntVal’, namely ‘(func a
b)’
  In the first argument of ‘Right’, namely ‘(IntVal
(func a b))’
  In the expression: Right (IntVal (func a
b)) typecheck
```

However, the compiler complains because the type of “func” is “Float -> Float -> Float”. We thought the type of it should be the same as, for example, the type of (+), which is “Num a => a -> a -> a”, so it should work on integers and floats at the same time. But the func’s arguments and return value are limited to Float as the first two arguments func received are of type Float. After we did some research on the Internet, we realized in Haskell there is something called rank-n types. We didn’t learn much about it but at least we know that func is rank 1, and once it receives arguments of a specific type then its parameters’ are limited to that type. We bypass this issue by converting all integers into floating numbers using “fromIntegral”, then perform the arithmetic operations. If the two arguments are both integers, we will convert the floating number result back to integer(using “round”) and return, as shown below:



```haskell
floatOperations :: BTree -> Expr -> Either EvalError Value
floatOperations vars expr = case (eval vars x, eval vars y) of
  (Right (FltVal a), Right (FltVal b)) -> Right (FltVal (func a b))
  (Right (FltVal f), Right (IntVal i)) -> Right (FltVal (func f (fromIntegral i)))
  (Right (IntVal i), Right (FltVal f)) -> Right (FltVal (func (fromIntegral i) f))
  (Right (IntVal a), Right (IntVal b)) -> Right (IntVal (round (func (fromIntegral a) (fromIntegral b))))
```

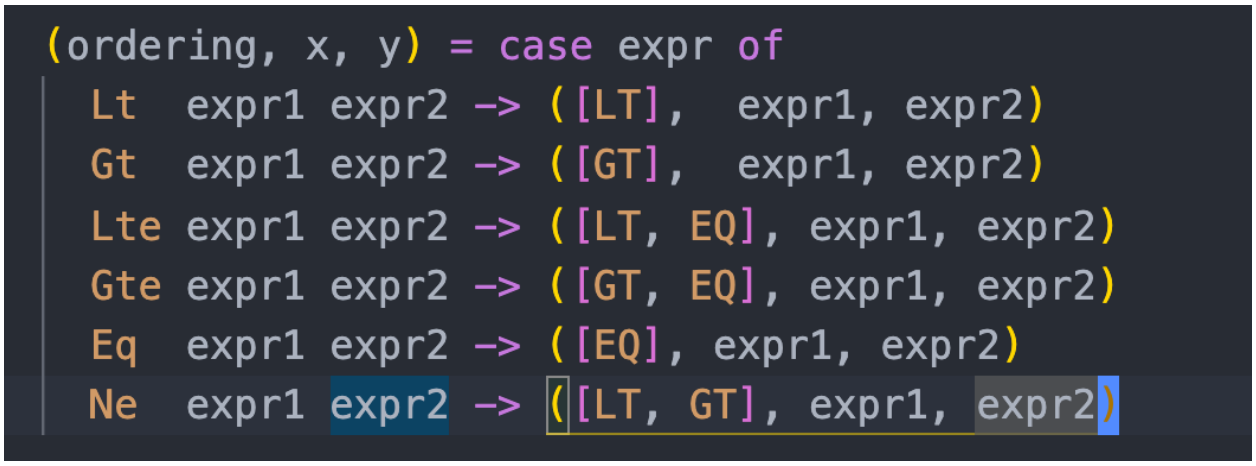
When trying to apply comparison operators on string, float, integer and boolean values, we realized we cannot convert string into numbers and do the comparison. So we end up using a built-in function supporting these different type values all at the same time:



```haskell
compare :: forall a. Ord a => a -> a -> Ordering
```

It returns Ordering type value, which is one of [LT, GT, EQ].

We used the following mapping to determine if the comparison evaluates True or False:



```haskell
(ordering, x, y) = case expr of
  Lt  expr1 expr2 -> ([LT],   expr1, expr2)
  Gt  expr1 expr2 -> ([GT],   expr1, expr2)
  Lte expr1 expr2 -> ([LT, EQ], expr1, expr2)
  Gte expr1 expr2 -> ([GT, EQ], expr1, expr2)
  Eq  expr1 expr2 -> ([EQ], expr1, expr2)
  Ne  expr1 expr2 -> ([LT, GT], expr1, expr2)
```

So if the result of “compare” is an element of “ordering”, we know that the comparison returns True, otherwise it returns False.

Function Declaration

Since functions need to be stored, we started off by adding a list of tuples that would store function data in state. Then we implemented the parser for it which was divided into three parts-

1. fun name -> "fun" is keyword and name is an identifier
2. (comma separated args) -> arguments are identifiers for variables
3. {commands} -> this is a command block, surrounded by curly braces and contains multiple commands

The implementation for it is done in REPL.hs since the commands run there. It is simply parsed and stored.

Function Call

The parser was divided into two parts -

1. name -> name is an identifier
2. (comma separated expressions) -> expressions are the Expr data structure constructors

There are two types of function calls as explained in the Parsing section. They both run almost the same way, first we check if the function name exists in the state's function attribute. Then we check if the number of arguments and the number of expressions match. Then we check if the function contains a return statement. It does not matter where the return statement is, although the lines after it are redundant. After the checks, the function is given a new scope which is a copy of the state at the point of calling the function. Then the arguments are added to it, and then the block of code gets executed using different functions for VoidFunCall and FunCall.

Scope

A scope is only given to functions here and not to all kinds of blocks (while if). The scope here is a copy of the state when a function's block statement is reached. The execution is properly described in the Function section.

Conditional Statements / While Loop

While statements are similar to the ones in Java, there is a keyword, then a boolean expression and finally a block of statements.

If statements are also similar, although there are no else if statements, and the else statements are compulsory. Else if can be emulated by nesting inside the else block.

Note that the variables assigned inside the block will exist outside too.

Testing

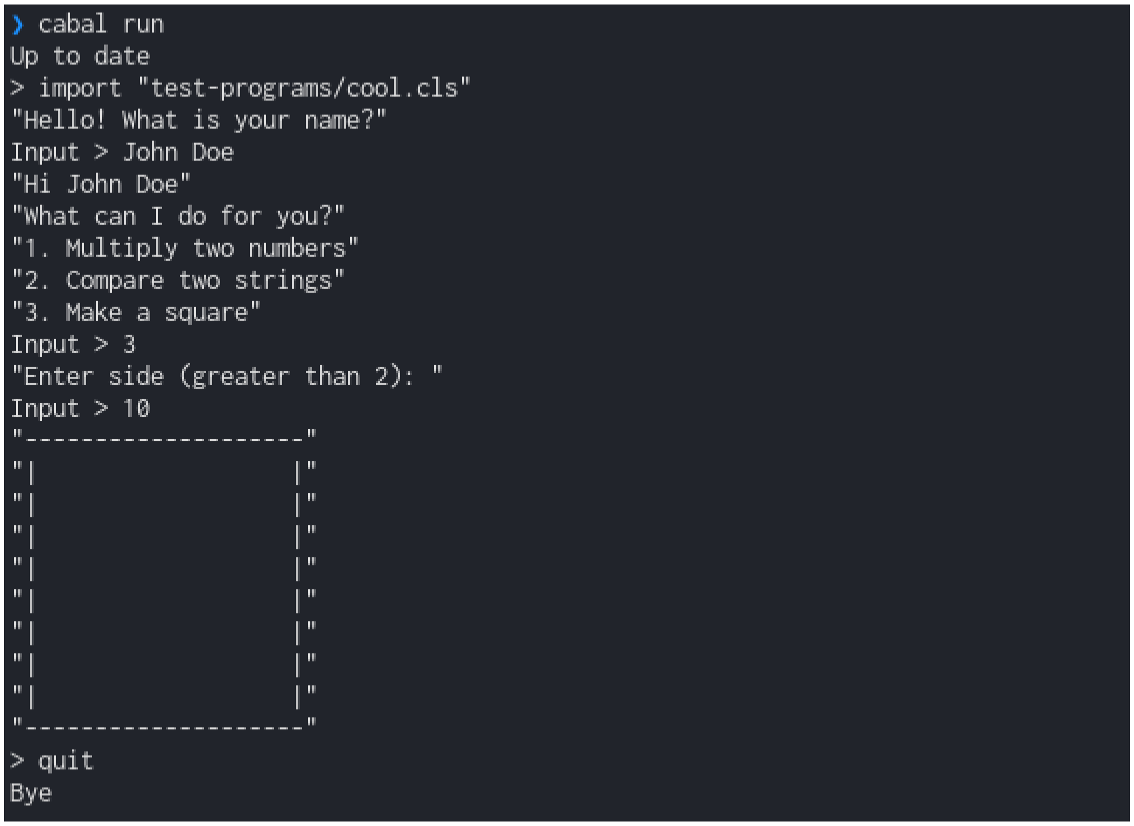
QuickCheck

Quickcheck property tests were used to test individual aspects of the parser and eval implementation. To test that the parser appropriately parses expressions and all the cases it's meant to deal with, the tests were quite unitary in nature. The tests begin from the bottom up, testing that appropriate types are parsed, such as Ints, Floats, and Strings. Quickcheck provides a random variety of values for a given type, thus ensuring that the parser is put through all valid forms of the input type. Of course, it also means that we encounter specific cases which we could choose to consider or ignore. For example, in Haskell, exponential notation is a valid form for floats, e.g. -1e0.5. The prop_parseFloat() test caught this and allowed us to decide whether or not we wanted to implement this form into the parser. In the end, we did not. Operators were also tested (where a generator was made that spits out a random operator to use as a character in a string) alongside ignoring whitespace (which also used a generator). Additional operators were also given their own tests, as some of them only function in specific cases, such as Abs which is a unary operator.

The eval function was also tested to make sure that the correct operations are performed on the values provided. Namely, the conversion of types should be handled correctly for binary arithmetic operators (where conversion is possible). For example, when adding an integer and a float, the result should have the type of a float.

Test Programs

There are test programs in '/test-programs'. Run / use them by importing them into the repl.



```
❯ cabal run
Up to date
> import "test-programs/cool.cls"
"Hello! What is your name?"
Input > John Doe
"Hi John Doe"
"What can I do for you?"
"1. Multiply two numbers"
"2. Compare two strings"
"3. Make a square"
Input > 3
"Enter side (greater than 2): "
Input > 10
"--------------------"
"|                  |"
"|                  |"
"|                  |"
"|                  |"
"|                  |"
"|                  |"
"|                  |"
"|                  |"
"--------------------"
> quit
Bye
```

Note that you can use the functions that were in the file.

Evaluation and Known Bugs

With more time, we would have enabled multi line parsing, and therefore multi line commands. The program fully meets the specification, including all easy, medium, hard and very hard requirements.

There are no bugs that we noticed at the time of submitting.

Conclusion

The language REPL fully meets all the requirements. It is well documented, so anyone can make small scripts using the language.