A Scripting Language

Overview

This practical required us to implement a small scripting language, consisting of various programming constructs such as expressions, variables, basic input/output and loops. This would allow the user to create a simple program simply by entering commands in the command line.

Link to repository: https://kb267.hg.cs.st-andrews.ac.uk/ScriptLanguage

Summary of Functionality

The following functionalities have been implemented in our program:

- Basic Requirements:
  - Allows user to declare and initialise variables
  - Tracks the state of the system which consists of variables and values
  - Has simple built-in commands that can be entered to perform calculations with numbers
  - Supports operations on strings such as concatenating and conversion between strings and integers
  - An extended parser which supports the commands and expressions, as well as allowing multi-digit numbers and whitespaces
  - A command to output variable values and a command to read user input
  - A 'quit' command to exit the system
- Additional Requirements:
  - Includes several Quickcheck tests to be used in testing alongside manual tests to ensure the proper functioning of the program
  - Supports a broader range of functions that perform a variety of calculations
  - Supports the use of negative numbers
  - Supports additional data types such as floating-point numbers and Boolean
  - Uses a binary tree to represent variables instead of a list of pairs
  - Includes a command to read files containing code which can be executed to produce an output
  - Allows for better treatment of errors
  - Allows user to write conditionals using an if…then…else construct which can also be nested
  - Allows user to repeatedly execute sections of code by writing loops over a block of commands

- The program can save command history and completion in a file using Haskeline

Known Problems

Repetition of Code causing slow running in QuickCheck Property Tests

Throughout the eval-related property tests in `Tests.hs` there is heavy code repetition in the generation of generic Trees and values done at the beginning of each test. Other than adding bulk to the code, this also means that each test must generate a new set of values which causes the tests to take additional time to run.

We investigated methods to reduce this, aiming to try and use the same arbitrary datasets in each test however we did not have time to fully implement a solution, so the problem still exists. The tests do however, still work correctly.

Design

Performing all Numerical Operations on Floats

We chose to perform all operations on floating point numbers instead of including cases for integers. This is because integers can be easily represented as floating point numbers (with a decimal value of 0), whereas converting floating point numbers to integers often results in a loss of data.

As a result, we could use one set of cases for both Floats and Ints without losing any data which is why we chose this option

Taking all Numerical Inputs as Floats

In order to allow the easy passing of inputs to operations, we decided to convert all numerical input to floating point numbers.

This is because of the decisions discussed above that resulted in our evaluation commands expecting Float inputs rather than Integers.

The Command Precedence System

The precedence order of commands in our language is as follows (from highest to lowest):

- Higher level commands such as repeat
- Values / Strings / brackets / variable names
- Mathematical operators (multiplication, division, power, etc)/ Boolean evaluators
- Basic commands which must come before values
- Assignment operators (=) / operations on variables (print, remove)/ if then else / reading user input
- Semicolon

We chose this order of operations as it mimicked the behaviour of other programming languages we are used to coding in and so we felt it would be most intuitive for other users too.

An example of how this affects how statements are processed can be seen below:

```
Print (pi 3 + 4 * 5 / 6)
```

This line of code will be evaluated in the following steps:

- `Print (pi 3 + 4 * 5 / 6)`
- `Print (pi 3 + 20 / 6)`
- `Print (pi 3 + 3.33)`
- `Print (pi (6.33)`
- `Print (19.88)`
- The value `19.88` is printed into CLI

Using `StateT` to update system state

`StateT` has been implemented as the state tracker for the language to help implement the variable name auto complete function.

There also is a temporary state variable which is used when each set of commands is processed. The reason behind this decision is to have an atomic transaction between the memory and the language to prevent malicious and broken syntax from affecting the memory.

For example, if when a set of commands is processed and for some reason the process function cannot process the given command, the whole set is discarded, and the main memory is not affected.

Using Haskeline instead of the basic input taking system

We implemented Haskeline for our input taking system as it provided the infrastructure to implement autocomplete and command history functionality used in the program.

It also allowed for easier customisation of the interface provided to the user.

Processing multiple commands

In our program, multiple command inputs are processed in 2 steps. First, the parser breaks the input down into single commands by splitting on any ';' characters.

Then, the '`processList`' function iterates through the generated list of individual commands and runs them one-by-one, making changes to a temporary memory after each which is returned when all of the commands have been run.

We felt this was the most intuitive option as it meant any sequence of commands containing an error would not modify the memory at all, instead of having partial modification of the memory from commands ran prior to the error.

Parsing a file

To process a file the following approach was taken. First, the file is read using the '`readFile`' function then each line is stored in a list using the 'lines' function.

Then, we decided to concatenate all the lines together as a single string, delimiting each line of code by a ';' to allow for the sequence of commands to be processed by the existing multiple command infrastructure discussed above.

An important note as a result is that when writing program in a file intended for the scripting language it is advised not to use ';' at the end of lines and it would not be able to parse the code provided.

The parser is designed to use ';' to differentiate between commands on a single line and newline characters to differentiate between different lines of commands.

For example, the following code would result in a `bad parse` error:

```
X=20;

Print(x)
```

Looping over a set of commands

To loop over a set of commands for a given amount of repeats the following approach was taken. First, the parser looks for the keyword repeat followed by a natural number (as the program cannot repeat negative number of times) and '`{`'.

It then looks for a set of commands followed by '`}`'.

If these conditions are met, it then passes the set of commands and the repeat counts to a function named '`pRepeat`'. Then the function recursively appends the list of commands given to it to the end of the previous list and returns the resulting list.

Similar to above, this approach was taken to take advantage of the existing infrastructure for processing multiple commands.

This could result in a higher space complexity than iterating over the same list multiple times, however we felt this approach took good advantage of our program's existing abilities and still worked efficiently.

Using Trees to Store Variables

We decided to go with the suggestion of a binary search tree to represent variables instead of the list of pairs. Since this reduces the number of comparisons needed to find variables, when storing large numbers of variables, it would reduce the time taken to access their values which is why we decided to use it.

Although many examples of binary trees use a node's value to decide its placement in the tree, this would not work for our program as variable values have to be searched for using their name.

To solve this, we decided to use the variable names as keys and store values alongside them in each node. Through String comparisons this allowed for the creation of a properly formatted Binary Tree that would map variable names to values.

Testing

QuickCheck Testing

For the QuickCheck testing, we developed a generator that would create a tree of arbitrary length with random Int values stored in it. This allowed for running of property tests with an appropriately random dataset to ensure functionality over a far greater range of cases.

QuickCheck tests were focused on cases of the eval function. For each, the result of running the operation directly on the values and the eval case were compared for consistency.

These were the focus specifically because the generator allowed for them to be run on a large variety of data, giving greater confidence they were robust in cases of valid data.

Due to the format of these tests, running them also covers testing the following:

- searching the Tree for existing variable values as treeSearch is used throughout
- the show cases of Outcome as they are used in several of the tests (such as prop_eval_to_string)
- many of the instance cases of Outcome (e.g. Num Outcome) as they are used in operations such as adding two Outcomes which appear in the tests

If any of these sections of functionality weren't implemented correctly, it would cause related tests to fail.

Manual Testing

The manual tests were designed to cover areas left after the QuickCheck tests, this includes invalid cases for the eval function, testing variable storage works correctly and many other cases where user input is involved that couldn't be as easily simulated or the test mapped less neatly to a property test.

The details of the manual tests were as follows: (Will be filled in)

Test	**Reason**	**Process**	**Expected Result**	**Result**

Storing a variable	Checking variables are stored correctly and can be retrieved	Storing 3 in x and then running print x to confirm it was stored	3 is printed	3 was printed as expected.
Updating a variable	Checking variables are updated correctly	Storing 3 in x and then storing 5 in x and printing to confirm.	5 is printed	5 was printed as expected.
Deleting a variable	Checking variable deletion works correctly	Storing 3 in x and then running rem x.	X is removed	X was removed as expected
Reassigning a deleted variable	Checking a previously deleted variable can be reused for different values	Running the above test then assigning 5 to x and printing to test.	5 is printed	5 was printed as expected.
Attempting to store an error in a variable	Checking error handling takes place before attempting to assign values to variables	Attempting to store 'ln "test"' in x and printing to check.	An error is printed and no value is found at x.	The appropriate errors for both parts of the test are printed.
Incorrect typed evals	Checking error handling prevents the program from crashing on incorrect inputs	Attempting to run 'print 3 + "test"'	An error is printed.	The appropriate error is printed and the program keeps running.
Inputting invalid commands	Checking error handling prevents the program from crashing on incorrect inputs	Attempting to run 'test'.	An error is printed.	The appropriate error is printed and the program keeps running.
Deleting an unassigned variable	Checking the program does not crash if attempting to	Attempting to run 'rem x' when no value is assigned to x.	An error is printed.	The appropriate error is printed and

	delete a non-existent variable			the program keeps running.
Printing an unassigned variable	Checking the program does not crash if attempting to print a non-existent variable	Attempting to run 'print x' when no value is assigned to x.	An error is printed.	The appropriate error is printed and the program keeps running.
Auto-completing a variable assignment	Checking the auto-complete works for writing a variable assignment	Running 'test = "hello"' then typing 'test = "h' and pressing tab. (After clearing history)	The rest of the statement is filled in correctly.	The rest of the statement auto-completes to 'test = "hello"'
FileReader operating	Checking running a given file containing code and operating on the file functions correct	Running read filename.txt	The program reads the commands in the file and processes the operations	The program reads the file does the operations and proceeds to take further user input
FileReader recognising a bad parse	Checking the error handling for a file containing unparsable / wrong code works correctly	Running read errorfile.txt	The program reads the commands and outputs bad parse then proceeds to take user input	The program reads the commands and outputs bad parse then proceeds to take user input
FileReader error handling	The user could input the wrong file name or directory, and this should be handled carefully	Running read wrongfilename.txt	The program outputs an error saying cannot find file and bad parse and proceeds to take user input	The program outputs the given error and asks user for new input

Notable Challenges and Solutions

Parsing and Processing multiple operations at a time

The main challenge for this section was in the decision-making process on whether to separate the commands when user input is taken and to parse and process individual commands or to parse the whole input and then break down into individual commands and then process each command.

The flow for the first approach would be the following:

User input -> split commands by ';' -> iterate over the list of commands -> take each command and pass into parser -> parse command -> process command -> repeat until list is empty

The flow for the second approach would be the following:

User input -> pass into parser -> parse set of commands -> return list of commands -> iterate over list of commands -> process individual command -> repeat until list is empty

The second was taken as its more abstract nature allowed for the '`processList`' function to be reused in other aspects of the program (such as running the commands parsed from a file).

Processing operations on the created Outcome datatype

As required by the specification, the program had to have a Datatype handling results of evaluated commands. Initially, this cause issued when evaluating individual terms in commands since it could not be used with many of Haskell's predefined operators (such as +, -, etc).

When tackling this issue, we came up with 3 different approaches which were used throughout our code as appropriate.

The first was to create a function which converted all instances of Outcome to their inbuilt counterparts. An example would be the following:

```
EvalOutcome (Flo n1) = n1

EvalOutcome (eval vars x) + EvalOutcome (eval vars x)
```

This approach was taken for some of the evaluating functions such as Boolean operands it would have been messy to implement it everywhere.

The second approach was to have separate functions take in Outcome types and process the operations appropriately. An example would be the following:

```
AddExpFunc (Flo n1) (Flo n2) = n1 + n2
```

This was used in methods such as `flosEval` where the type of the input Outcomes had to be validated before running the command.

The third approach was overriding the Numerical and Fraction operations for the Outcome Datatype, to allow for commands such as

```
(Integ 3) + (Integ 2)
```

To be ran directly to get a result of

```
Integ 5
```

This was useful for cases such as the following snippet as it allowed for them to be evaluated without issues:

```
eval vars x + eval vars y
```

Parsing a String

The challenge when parsing a String was that the provided parser's character set did not allow for parsing of non-alphanumerical characters. However, we still wished to use it as it was useful when handling input as it could compare inputs to the String given before.

To solve this, we created a parser function which would handle both alphanumerical characters and other characters such as dots, hyphens, sharps, etc. The issue with this approach was that it also then failed to detect Strings not within the modified character set.

In our final solution, we used the 'sat' function to evaluate input characters which allowed it to accept as many characters as possible provided they were not ' " '.

Haskeline auto complete

When implementing the auto-complete functionality, we had to use a lot of trial and error due to a lack of similar examples being found in research.

Through this we discovered that if the program state was tracked by StateT then it would allow for the completion function to access memory when comparing the user input to stored variables.

Initially, we had planned to implement this by passing the State to the Haskeline setting function after each input however because our program separated the REPL loop from the base Haskeline call this was unable to be implemented.

As a result, we implemented `StateT` into the completion function to allow for this functionality to work correctly.

Reading commands from a file

For reading commands from a file, the main challenge was maintaining the state of all the variables that were in the file using `stateT`.

Originally, the approach was to have a separate function with its own loop to read each line of code and execute it. But this made this unnecessarily complicated and resulted in errors as there were several type mismatches with InputT(`StateT` [String] IO), resulting in the state not being maintained.

The simple solution was to only have a separate function for the file handler which read each line and returned it to the main loop. And then processing the code in the original loop in the same way. Therefore, variables were able to be stored in `StateT` as the processing is done in the original context. This also meant that there weren't any type mismatches.

Generating Trees for QuickChecks

For property tests, QuickCheck provides generators for types such as Int and String to create random test data. When running the eval tests, we decided to create a custom generator for an arbitrary length Tree (of Integers) to provide similarly random test data.

The existing generators were able to be used to generate Keys and Values, however the issue was if the keys were fully random then all tests would fail as the search relies on the Tree's structure being correct (with random keys, it would be possible for a key of greater value to be stored as the left child of a node instead of the right child).

To solve this, a custom generator method was created which calculates appropriate keys from a base value instead of randomly generating them. The calculation used is based upon the length of the tree so the difference between key values is large enough that the tree can be correctly structured.

Testing Consistency in Results that could return NaN

When testing the eval cases with QuickCheck, their validity was confirmed by comparing the result to the result of running the intended operation outside of eval. This became an issue in cases such as a division by 0 which returns NaN.

In Haskell, comparing two NaN values will always return false so another helper method had to be created for tests that involved NaN values so they could be compared appropriately.

Adding Error Handling to Float-based Eval Cases

During initial testing, we discovered that eval cases based around Floats (i.e. that mapped values to floats with `showFlo` before running a calculation like log or factorial) were not appropriately error handled if a non-numerical input was passed.

Although this could be solved by adding an Either case to each calculation, we found this made the eval function extremely bulky and was very repetitive.

To solve this, the flosEval and floEval methods (for 1 and 2 arguments respectively) were created to integrate error handling into the calculations in a way that did not increase the complexity or length of eval by as significant a margin.

Error Handling resulting in stored Errors

Originally when error handling was added to the Set case of `process` in `REPL.hs`, the errors were correctly handled, however the method itself had no functionality to prevent the updating of state in an error case.

This led to scenarios where commands like `x = 1 + "test"` would cause the value of x to be set to `"Type incompatible"` instead of skipping assignment. The solution to this was creating the `safeEval` method which returned an Either value so in the case of an error, instead of updating the state it would be maintained and the error would be printed instead of stored.

Summary of Provenance & Breakdown of Functionality

The following code was written directly by us:

- Most of REPL.hs:
  - processList: processes each command in the list of commands which are separated by ';'
  - slice: used to take the variable name when auto complete is used which is then used as part of the state to store its value
  - hlSettings: used to store history commands which also takes a autocomplete function to automatically complete variable names
  - fildHandler: reads the provided file and formats the code into a list of commands
  - concater: turns the list of code into a single string separated by ';'
- Most of Expr.hs:
  - Instance of Num Outcome which overrides the instance for number operations for the outcome type
  - Instance of Fractional Outcome which overrides the instance of fractional operations for the outcome type
  - `showFlo`: takes an outcome variable and returns the float value of it
  - showInteg: takes an outcome variable and returns the int value of it
  - showBool: takes an outcome variable and returns the int value of it
  - pRepeat: takes a number and a list of commands to provide the loop functionality
  - pAbove: a parser for a list of commands which allows for the implementation of loops
  - toExpr: takes a string and returns an expression

  - pAbover: adds commands to the end of each other, so the command after ‘;’ gets added to the list of commands before ‘;’
  - pSub: applies input precedency for operations with ‘-’ which was made to oppose the inside out evaluation of expressions
  - pDriver: does the same as pSub but for operations with ‘/’
  - var: variable parser which identifies the command line input with the identifier parse and returns a variable
  - neg: a negation parser which is used as a helper tool for variable negation
  - nVar: negative variable parser which is used to match negative variables
  - val: value parser which is used to match any entry numbers so that both floats and ints get accepted as values
  - value: a value parser which matches either a value or a variable
- Some of Parsing.hs:
  - float: a negative and positive float parser
  - numb: a specific parser which matches both an int and a float
  - flt: a positive float parser
- All of Outcome.hs:
  - ‘ErrorCase’ type
  - ‘Outcome’ data which contain all the data types
  - An instance of Outcome which overrides the show method for the outcome data type
- All of Trees.hs:
  - ‘Name’ type
  - ‘Tree’ data structure
  - treeSearch: searches the tree checking if a variable exists and if not, then an error is given
  - varLookUp: searches the tree and returns an appropriate either case
  - treeAdd: adds a node to the tree or updates existing one
  - treeRemove: removes a node from the tree, updating any children accordingly
  - removeNode: handles the children during the removal of a node
- All of Tests.hs:
  - Arbitrary Tree of Int Outcomes generator
  - Arbitrary Int Outcome generator
  - treeKeys: iterates through a Tree and adds its keys to a list. Created separate to showTree so that it returns [String] instead of [[Char]]
  - generateRandomKey: takes a Tree and picks a random key from it
  - genTest: helper method for testing the arbitrary tree generator. Returns the tree and a random key, value pair.
  - Property tests: Tests various cases of eval

  - arbitraryLengthTree: defines how to generate an arbitrary length Tree for the generator
  - check: runs the property tests

The following code was built upon the provided source files:

- Some parts of REPL.hs:
  - dropVar: which returns a new set of variables with the given name removed
  - process: processes the command retuned by the expression parsers
- Some parts of Expr.hs:
  - Expr data which contains the operations and variables
  - Command data which contains the commands
  - Eval: evaluates the operations on variables
  - pCommand: a command parser which takes an input from the command line
  - pExpr: an expression parser which parses a term first
  - pFactor: a factor parser which proceeds to evaluate the given expressions for the given operations
  - pTerm: a term parser which takes a factor and tries to match one of the following commands to apply the operation on

The following code was built upon or inspired by externally sourced files:

- Some parts of REPL.hs:
  - findCompletion: searches the variable list in `StateT` and checks if the current input in CLI is prefix of any of the variables in there which was inspired from [1] and [2]
  - findSubstring: finds the position of a '=' so that variable declarations can be identified which was inspired by the following article at [3]
  - loop: contains the main read, evaluate and print loop which was inspired by the following article at [4]

Conclusion

We managed to implement a small scripting language, consisting of expressions, variables, basic input/output and loops. Then, testing the program by both manual tests and implementing QuickCheck tests and running them.

Overall, we view this practical submission as a success, due to the lack of errors being outputted, the results of our tests, and the completion of all steps.

Given more time we would have liked to implement for and while loops. Also, we would have liked to find a way to implement Sin, Cos and Tan functions. Since the inbuilt haskell functions round the results causing inaccuracies and we did not have time to explore an alternative.

Additionally, we would have liked to reduce the repetition in the QuickCheck property tests by using another testing framework alongside it – in our research, we identified Tasty ([5]) as a good potential option for this but again did not have time to implement it.

References

[1] amindfv (2011). Haskell (haskeline) word completion. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/6147201/haskell-haskeline-word-completion [Accessed 24 Mar. 2022].

[2] reddit. (2013). r/haskell – haskeline woes. [online] Available at: https://www.reddit.com/r/haskell/comments/1os0yq/haskeline_woes/ [Accessed 24 Mar. 2022].

[3] Programming-idioms.org. (2022). Find substring position, in Haskell. [online] Available at: https://programming-idioms.org/idiom/62/find-substring-position/968/haskell [Accessed 24 Mar. 2022].

[4] reddit. (2013). r/haskell - Haskeline woes. [online] Available at: https://www.reddit.com/r/haskell/comments/1os0yq/haskeline_woes/ [Accessed 24 Mar. 2022]

[5] hackage.haskell.org (2021). tasty: Modern and extensible testing framework [online] Available at: https://hackage.haskell.org/package/tasty [Accessed 4 Apr. 2022]