

CS2006 Haskell 2 Practical Group Report

Tutor: Gregor Heywood

Group Number: 8A

Matriculation ID's: 200014517

200027532

200019811

Overview

In this practical, we were asked to create a Haskell program which would run a small scripting language, with functionality for basic expressions, variables, inputs (through files or user input), outputs and loops. Our program supports all of the basic functionalities and the following additional requirements:

- **Easy** Add some appropriate tests using Quickcheck.
- **Easy** Support additional functions, such as abs, mod or power
- **Easy** Extend the parser to support negative literals
- **Easy** Support floating point numbers as well as integers, including conversion operators
- **Medium** Use a better representation for variables rather than lists of pairs (e.g. a binary tree).
- **Medium** Add a command to read and process input files containing a list of commands, rather than reading commands from the console
- **Medium** Implement better treatment of errors: instead of using Maybe and Nothing to indicate errors, look at the Either type and consider how to use this to represent specific errors
- **Medium** Add a command for making choices (i.e. an if...then...else construct)
- **Hard** Use Haskell line to add command history and completion (including tab completion of variable names) to the REPL
- **Hard** Add a command for simple repetition. This should allow looping over a *block* of commands separated by a semicolon. e.g., you could support: repeat 10 { x = x + 1; print x }
- **Hard** Add more expressive looping commands, such as for or while loops
- **Hard** Allow defining and calling functions

Design

Generally, we designed this language to have very simple and forgiving syntax, with a weak type system and very small command set.

While the specification stated that only the Expr and REPL files needed to be changed, we ended up changing every other file as well as adding a file for QuickCheck and a binary tree implementation.

The main loop of the scripting language is:

- Get command string from user
- Parse the string with pCommand to convert it to a command data type
- Process that command
- Repeat

To get the command string from the user, we use Haskeline. We decided to do this as it allows for implementation of command history and tab completion.

We extended the Command type that was given in the supplied code to represent any command that the user can give, and did the same with Expr. In addition to this, we created a Value type, which can represent the value of any variable or literal that can be used in a script.

All of the commands that can be given by the user are:

- Set (an assignment)
- Print
- File (run a file at a given file path)
- Block (a block of commands)
- If-else
- Repeat (repeat a command a given number of times)
- For loop
- While loop
- Quit
- Execute a function
- No command (if the user gives an empty string, get another input)

All of the expressions that can be evaluated are:

- Add (can be used to add two numbers or concatenate two strings)
- Subtract
- Multiply
- Divide
- Absolute value
- Power
- Modulus
- ToString
- ToInt
- A value
- A variable name
- User input
- Boolean expressions (==,!=,<=,>=,<,>)
- A function definition

And all of the values that can be represented are

- Integer
- Floating point number
- String
- Error
- Boolean
- Function

Minimum Requirements

In order to implement basic arithmetic, we extended the parser and Expr to allow for reading of arithmetic expressions which are then converted to the corresponding Expr. We then extended the definition of eval to allow for these to be evaluated. For example, an input string “1+2” would be converted to ADD (IntVal 1) (IntVal 2), which, when evaluated, gives Just (IntVal 3). For ToString(x), eval returns the Just (StrVal (show (eval(x)))), and for toInt(x), eval returns the Just (IntVal (digitsToInt x)). Process was also extended to allow for commands to be executed. To perform an assignment, the vars in state are updated so that they include a name-value pair to represent that variable. To perform a print, the result of evaluating a given expression is printed to the console.

Basic Requirements

A quit command to exit the system

When “quit” or “exit” is received as input, exitSuccess is used to close the system. This is used as it allows for exiting the program from anywhere.

Tracking the system state (LState), which initially includes the values of the variables in scope

LState is still used to track the system state, and is updated when a new variable is added, and when a variable is reassigned. The LState only needs to track the system state as functions are first-class, so are represented as variables. The variables are represented using a binary tree instead of a list to improve performance (more details on function and binary tree implementations below).

A parser. It currently only supports a subset of the commands and expressions. It also only supports single digit numbers, and does not support whitespace. Extend it so that it supports multi-digit numbers and whitespace around operators.

Some details on the parser are given above. The parser function pCommand is used to parse user input as a command, and pExpr is used in conjunction with pFactor and pTerm to parse expressions with a correct order of precedence. We also updated Parsing.hs to allow for a more complete parser. For example, when parsing string literals between “”, we use our printable function to parse any character that is printable (alphanumeric, punctuation etc). This works because before a string is parsed, all “ characters are replaced with ‘\0’, which is not printable. We used this approach because the current parsing.hs would make it far more complicated to parse this great range of characters into a string. To parse multi-digit numbers and variable names, we used the “many” function to allow for many characters to be returned. To include white space, the space function is used.

Support for strings, including parsing and relevant operators, in particular:

- Concatenating strings
- Converting between strings and integers

String literals are parsed using the method described above. Concatenation is performed using the '+' operator. We chose to use this because it results in a simpler syntax and parser, as the parser only has a single way to parse a '+' character as opposed to using '++'. Evaluating this adds very little complexity as eval already checks the type of both of the values that are being added, so all that is needed to implement this is another case for adding two strings. Converting between strings and integers is performed using the method described above.

A command for reading user input as strings

If a variable is assigned to input, when evaluating the value of that variable, haskeline is used to accept input from the user, and this input is stored as a string.

Additional Requirements

We have implemented all of the Additional requirements with the exception of using StateT to represent system state.

Easy Support additional functions, such as abs, mod or power

We have added support for abs, mod and power. These were implemented in a very similar fashion to previous arithmetic expressions. The syntax for abs x is |x|. We chose this as it is the same as the syntax for absolute value in mathematics. The operator used for mod is '%'. This is because this is the same symbol used in other languages such as Java, and we were making an effort to make the syntax as simple and easy to understand as possible. The symbol for power is '^'. We used the same reasoning as mod when deciding the syntax for this operation.

Easy Extend the parser to support negative literals

All that this involves, is when parsing a number, parse the first character as '-' OR as a numeric character, then parse many numeric characters.

Easy Support floating point numbers as well as integers, including conversion operators

We extended the Value type to be able to represent a float, extended eval to be able to perform operations on floats (and where applicable, perform operations on ints and floats), and extended the parser so that it parses a number, a '.' and then more numeric characters. This string is then converted to a float.

Medium Use a better representation for variables rather than lists of pairs (e.g. a binary tree)

The file BinaryTree.hs includes a simple implementation of a binary tree that is used to implement this. The only functions that are implemented for this tree are insert (add an element to the tree) and getElem (returns an element from the tree) as these are the only functions that are needed to represent the variables in a script. getElem works because for a name-value pair that represents a variable, Eq and ordering is defined by the name of the variable. So given this function a name with an empty variable will return a name with the correct value from the tree.

Medium Add a command to read and process input files containing a list of commands, rather than reading commands from the console

If “file” followed by a file path is given, this file is read and executed. To read and execute the file, all of the contents are read, and split into lines. Each of these lines is then parsed as a command. Process has been modified so that it can accept a list of commands, which allows for all of these to be processed sequentially once they have been parsed. File Paths can also be given as a command line argument. If this happens, the system will quit once the file has terminated. This allows for executing scripts in a similar fashion to executables.

Medium Implement better treatment of errors: instead of using Maybe and Nothing to indicate errors, look at the Either type and consider how to use this to represent specific errors

While we did not use Either to represent errors, we did implement a better treatment of errors. We did this by including an “Error” type for values. This means that if an evaluation of an expression or command is invalid, instead of trying to process this, the system will print out an appropriate error message.

Medium Add a command for making choices (i.e. an if...then...else construct)

We have implemented an if-else construct. In order to do this, we have also implemented a boolean value type as well as boolean expressions. This involved implementing `==`, `!=`, `<=`, `>=`, `<` and `>` operators. These are used between two expressions to be used as the condition for if statements, for loops and while loops. If this condition is true, execute the first command, otherwise execute the command after else. If else is not given, and the condition is false, no command is executed. Because an If statement is a command, this allows for else if statements.

Hard Add a command for simple repetition. This should allow looping over a *block* of commands separated by a semicolon. e.g., you could support: `repeat 10 { x = x + 1; print x }`

When a repeat statement is given, it is converted to a for loop, where the target variable is “\0\0”, so that the user will not have a variable with the same name. Blocks of commands are read as a string between curly brackets, which is then split on ‘;’ and each command is parsed and executed when the block command is processed.

The split command used is a modified version of the lines command from prelude, found at:

<https://www.haskell.org/onlinereport/standard-prelude.html>

Hard Add more expressive looping commands, such as for or while loops

For loops:

Using boolean expressions from above, for loops are given a variable to check, a condition for that variable, and an operation to perform on that variable to move it closer to the condition. Until that condition is met, a given command is repeated. This command can be a block of commands or a single command

While loops:

Again using boolean expressions, a while loop is given a boolean expression and a command to repeat until that condition is no longer met.

Hard Allow defining and calling functions

We have a very simple function system in our language. They are first class and are assigned as values of variables. They can accept arguments that do not have a type system for the arguments, so each argument can be any valid value. These arguments will be stored in the main vars tree as variables, as there is no implementation of scope for functions. This means functions do not need a return value, and can just assign a variable a value which can then be accessed by the rest of the script once the function has executed.

Testing

For testing we implemented a few QuickCheck tests as well as made use of StacsCheck. Below is a list of manual tests as well. See figure 1 for the QuickCheck tests

Case No	What case is Being Tested	Pre Conditions	Expected Outcome	Actual Outcome
1	Quit and exit command works	NA	The program terminates	The expected outcome (figure 2)
2	Simple variable assignment (Strings and Integers)	NA	The variables assign with no parse errors or other errors displaying	The expected outcome (figure 3)
3	Print command	Variables have been established to print.	The variables and literals print correctly	The expected outcome (figure 4)
4	Printing string and integers in the same statement, showing automatic int to string and concatenation	NA	They print correctly	The expected outcome (figure 4)
5	Whitespace being ignored.	NA	The commands performs the same thing no matter the white space	The expected outcome (figure 5)
6	Evaluation with brackets	NA	The number correctly evaluates given the brackets	The expected outcome (figure 6)

7	User input	NA	The input is stored in the variable correctly	The expected outcome (figure 7)
8	Reading from a file	A file exists to be read from	The commands in the file are executed	The expected outcome (figure 8)
9	ToInt and ToString commands	NA	The evaluation is made correctly and so is the concatenation.	The expected outcome (figure 9)
10	Float and Negative literals are implemented.	NA	The evaluations are correct.	The expected outcome (figure 10)
11	Additional operators (abs, mod, power) work with integer and float	NA	The evaluations are correct.	The expected outcome (figure 11)
12	If statements	NA	The program prints the correct message	The expected outcome (figure 12)
13	Repeat	NA	The program repeats the block of code	The expected outcome (figure 13)
14	For loops	NA	The program repeats the block of code	The expected outcome (figure 13)
15	While loops	NA	The program repeats the block of code	The expected outcome (figure 13)
16	Function calling	A function is created to use.	The program executes the block of code with the given arguments	The expected outcome (figure 14)

The stacscheck test results can be seen in figure 15. They are 5 simple tests to test a few basic functionalities of the program.

QuickCheck was used to test some of the basic arithmetic expressions in the language.

Evaluation

Our submission completes the basic requirements as well as most of the advanced requirements. The program is very robust, it has decent error checking for a simple language. The syntax itself is quite intuitive and works quite well. It isn't whitespace dependent and does automatic int to string conversions when performing prints. The typing is quite loose and robust in general. There is a slight problem with functions, in which any errors that occur within a function are only caught when the function is executed. Also functions do not allow for argument names to have numbers in them i.e. num1. Additionally, Block statements cannot be nested due to the way that they are parsed, similarly, for loops cannot be used within Block statements. If we had more time, we would have added more tests with quickcheck.

Conclusion

We completed all of the basic and most of the advanced requirements to a good degree. We even used initiative and found ways to complete certain aspects of the practical not suggested in the specification (i.e. stacscheck & error type implementation). We found it quite difficult to get this done on time considering one of our team was ill with COVID and we were not granted an extension. Another member of our team also caught COVID making the workload even harder. Nevertheless we were still able to achieve this much.


```
*QuickCheck> runTests
=== prop_addToTree from QuickCheck.hs:19 ===
+++ OK, passed 100 tests.

=== prop_ADDInts from QuickCheck.hs:23 ===
+++ OK, passed 100 tests.

=== prop_ADDFloats from QuickCheck.hs:26 ===
+++ OK, passed 100 tests.

=== prop_ADDStrings from QuickCheck.hs:29 ===
+++ OK, passed 100 tests.

=== prop_SubIntvals from QuickCheck.hs:32 ===
+++ OK, passed 100 tests.

=== prop_SubFloats from QuickCheck.hs:35 ===
+++ OK, passed 100 tests.

=== prop_MultInt from QuickCheck.hs:38 ===
+++ OK, passed 100 tests.

=== prop_MultFloat from QuickCheck.hs:41 ===
+++ OK, passed 100 tests.

True
```

Figure 1: QuickCheck returning all true.

```

PS C:\Users\ruars\OneDrive\Documents\GitHub\haskell2> cabal run
Up to date
>quit
PS C:\Users\ruars\OneDrive\Documents\GitHub\haskell2> cabal run
Up to date
>exit
PS C:\Users\ruars\OneDrive\Documents\GitHub\haskell2> 

```

Figure 2: The program terminating from both the quit and exit commands

```

PS C:\Users\ruars\OneDrive\Documents\GitHub\haskell2> cabal run
Up to date
>x = "Hello World"
>y = 42
>z = true
>

```

Figure 3: The variables assign with no parse errors or other errors displaying

```

>x = "Hello World"
>y = 42
>print x
Hello World
>print y
42
>print ("Hello World!")
Hello World!

>print (x + ". World Hello! ")
Hello World. World Hello!

```

Figure 4: Variables and literals printing correctly and are concatenated in the print

```
>x = 1 + 3
>y=1+3
>z = 1+ 3
>print x
4
>printy
4
>print (z)
4
>
```

Figure 5: The program outputs the same thing no matter the white space


```
>print(2*(1+3))
8
```

Figure 6: The number correctly evaluates given the brackets

```
>x = input
42
>y = input
Hello world
>printx
42
>printy
Hello world
> print(x + y)
42Hello world
>
```

Figure 7: The input is stored in the variable correctly and printed correctly

```
PS C:\Users\ruars\OneDrive\Documents\GitHub\haskell2> ./dist-newstyle/build/x86_64-windows/ghc-8.10.7/haskell2-0.1.0.0/x/haskell2/build/haskell2/haskell2 Tests/TestFiles/TestBasic
hello, world!
```

Tests > TestFiles >  TestBasic

```
1 print("hello, world!")
```

Figure 8: The output of the commands in the file and the file being executed

```
>x = 20 + toInt("22")
>print x
42
```

```
>y = "The meaning of life is " + toString(42)
>print y
The meaning of life is 42
```

Figure 9: The program calculates a number after converting it to an int and turns an integer to string

```
>x = -2
>y = 2.5
>print x+y
0.5
```

Figure 10: The program calculates with negative numbers and floating point numbers

```
>x = |-2|
>y = 10%3
>z = 2^3
>print(toString(x) + " " + toString(y) + " " + toString(z))
2 1 8
```

Figure 11: The program calculates with abs, mod and power modifier

```
>x = 3
>if(x==3){print"yes"}else{print"no"}
yes
>x = 1
>if(x==3){print"yes"}else{print"no"}
no
>
```

Figure 12: The program prints the correct message based off the if statement

```
>repeat 5{print "hellow world"}
hellow world
hellow world
hellow world
hellow world
hellow world
```

```
>i = 0
>for(i; i >= 4; i=i+1){print(i)}
0
1
2
3
```

```
>i = 0
>while(i<10){i=i+1}
>print i
10
```

Figure 13: The program repeats the block of code correctly for a repeat, for and while loop

```
>funcAdd = args(a,b){number = a+b; print(number);}
>funcAdd args(1,2)
3
>x = args(a,b,c){printa;printb;printc}
>x args("Hello World","2", "3")
Hello World
2
3
```

Figure 14: The program executing the block of code with the given arguments

```
rdn1@pc8-002-1:~/Documents/year2/cs2006/practicals/Haskell2Testing/haskell2/app $ stacscheck /cs/home/rdn1/Documents/year2/cs2006/practicals/Haskell2Testing/haskell2/Tests
Testing CS2006 Haskell2
- Looking for submission in a directory called 'app': Already in it!
* BUILD TEST - build-clean : pass
* BUILD TEST - public/build : pass
* COMPARISON TEST - public/Test01_Print/progRun-expected.out : pass
* COMPARISON TEST - public/Test02_If_Statements/progRun-expected.out : pass
* COMPARISON TEST - public/Test03_For_Loops/progRun-expected.out : pass
* COMPARISON TEST - public/Test04_Functions/progRun-expected.out : pass
* COMPARISON TEST - public/Test05_Repeat/progRun-expected.out : pass
7 out of 7 tests passed
```

Figure 15: Stacscheck showing all tests run correctly