Zolon

a simple interpreted functional programming language

Petar Peychev

27 April 2019

Project report, submitted as part of the Computer Science Programming SOFT10101 module, School of Science and Technology, Nottingham Trent University

Overview

My project consists of language design and an interpreted implementation of a simple functional programming language. The name is a semi-anagram of the given name of Alonzo Church, inventor of the λ-calculus. In this report, I will go through the namespace structure, folder structure, build process, grammar, read-eval-print loop (REPL) logic, lexical analysis pass, syntactical analysis pass, evaluation pass and the implementation of the Abstract Syntax Tree (AST). I will also provide examples along the way and acceptance tests for the whole project at the end.

1. Namespace Structure

In order to prevent naming collisions and pollution of the global namespace, the project defines namespaces for the major components of the language implementation. The list of namespaces with classes, which belong to them is as follows:

cli (command-line interfaces to the interpreter):
- InteractiveInterface (the REPL interface to the interpreter)
- FileInterface (the file interpreter interface) * to be implemented

lexical_analysis (lexical analysis of the source string and generation of lexemes):
- Scanner (main class, responsible for the lexical analysis pass)
- Token (data representation of lexemes with line, type and source information)

syntactical_analysis (syntactical analysis of the tokens and building of an AST):
- Parser (main class, responsible for the syntactic analysis pass)

syntax_tree (node classes for the AST, which contain evaluation logic):
- ArithmeticOperation (node representing an arithmetic expression)
- Binding (node representing a binding statement)
- BooleanLiteral (node representing a boolean literal value)
- Expression (abstract base class for all expressions)
- FunctionApplication (node representing function application) * to be implemented
- Function (node representing lambda function definition) * to be implemented
- InvalidBinding (node representing an invalid binding statement)
- InvalidExpression (node representing an invalid expression)
- LogicalOperation (node representing a logical expression)
- NumberLiteral (node representing a literal number value)
- RelationalOperation (node representing a relational expression)
- Statement (root node for the statement AST)

evaluation (logic related to the evaluation stage):
- Environment (representation of the evaluation environment)
- Value (data representation of a runtime value, which expressions evaluate to)

exceptions (user-defined exceptions):
- NotImplementedException (self-explanatory)

2. Folder Structure

The folder structure for the interpreter strictly follows from the namespace structure specified above. In the root directory “interpreter/” are located the entry-point file “zl.cpp” and the interpreter pipeline implementation “Interpreter.cpp”.

3. Build Process

As I have made the project using a text editor on GNU/Linux and command-line tools for building (GNU make & GCC) and debugging (GDB), the process I have used is Linux-specific.

I’ve written a makefile in the root “interpreter/” directory, which includes two different build recipes:

```
$ make zl (build the project into an executable named “zl” and clear excess files)
$ make zl-d (build a temporary “zl-d” executable and run the GDB debugger on it)
```

I’m including in the project the “zl” linux binary. Running the REPL should be as simple as navigating to the “interpreter/” directory and running “./zl”. (or alternatively building from the makefile using “make zl” then “./zl”)

If there are any issues with this, I’m sure I could figure out a way to resolve them.

4. Grammar

For the language design, I have written a grammar specification in a variant of Backus-Naur Form. **Nonterminals** in this format are represented by lowercase identifiers, **terminals** are capitalised identifiers, **derivation rules** are signified by an equals sign, a **choice** is represented by a vertical bar, **repetition** is signified by an asterisk (regular expression operator) and symbols can be parenthesized. The grammar is as follows:

```
statement = binding
          | import
          | expression

binding = IDENTIFIER BIND expression ;

import = IMPORT PATH ;

expression = logical ;

logical = equality (( AND | OR ) equality )* ;

equality = comparison (( EQUALS | NEQUALS ) comparison )* ;
```

comparison = addition ((LESS | GREATER | LEQUALS | GEQUALS) addition)* ;

addition = multiplication ((PLUS | MINUS) multiplication)* ;

multiplication = unary ((ASTERISK | FSLASH) unary)* ;

unary = (NOT | MINUS) unary
        | primary ;

primary = NUMBER
          | BTRUE
          | BFALSE
          | function
          | function-application
          | LPAREN expression RPAREN;

function = IDENTIFIER
           | BSLASH IDENTIFIER MAP LBRACKET function-body RBRACKET ;

function-body = expression
                | BAR subdomain-list expression ;

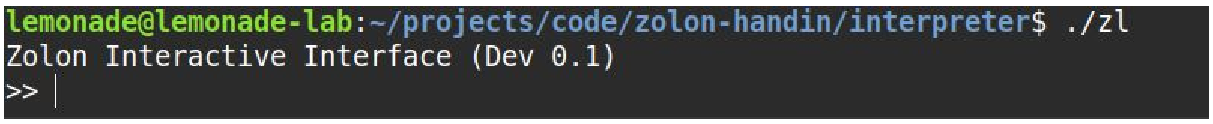
subdomain-list = expression COLON expression BAR
                 | subdomain-list expression COLON expression BAR ;

function-application = function
                       | function LPAREN expression RPAREN ;

* Greyed-out production rules are not yet implemented in the interpreter

5. REPL (read-eval-print loop)

Running the executable with no arguments starts the “Zolon Interactive Interface” or REPL.



```
lemonade@lemonade-lab:~/projects/code/zolon-handin/interpreter$ ./zl
Zolon Interactive Interface (Dev 0.1)
>> |
```

The logic for this part of the interpreter is located in “/cli/InteractiveInterface.cpp”. It initialises an environment, instantiates an interpreter, reads potentially multi-line statements as input and handles the execution of the reserved commands **debug** , **env** and **quit**.

The **debug** command toggles the displaying of verbose debug information about the interpreting process. (a list of generated tokens by the scanner and fully-parenthesized representation of the syntax tree generated by the parser)

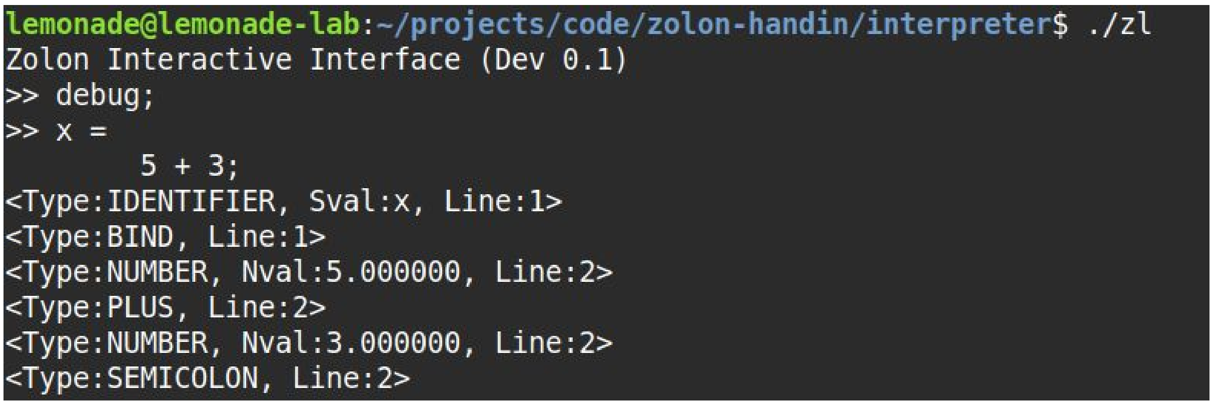
The **env** command displays a list of the currently bound values in the environment frame.

The **quit** command displays the environment bindings as well as the full list of valid statements entered during the interactive session, then exits the REPL.

Any other statement is sent to the interpreter and evaluated.

6. Lexical Analysis

The lexical analysis pass of the interpreter consists of scanning the input string (in the case of Zolon, a single statement) and splitting it into valid lexemes, generating errors in case the analysis detects problems. The class representation of a lexeme is in the Token class, while the logic for the analysis is in the Scanner class. I will use the interactive interface's debug mode to show how the scanner works. When a statement, such as "x = 5 + 3;" is scanned, the result is a list of Token objects with information about their types, line numbers and literal values:



```
lemonade@lemonade-lab:~/projects/code/zolon-handin/interpreter$ ./zl
Zolon Interactive Interface (Dev 0.1)
>> debug;
>> x =
       5 + 3;
<Type:IDENTIFIER, Sval:x, Line:1>
<Type:BIND, Line:1>
<Type:NUMBER, Nval:5.000000, Line:2>
<Type:PLUS, Line:2>
<Type:NUMBER, Nval:3.000000, Line:2>
<Type:SEMICOLON, Line:2>
```

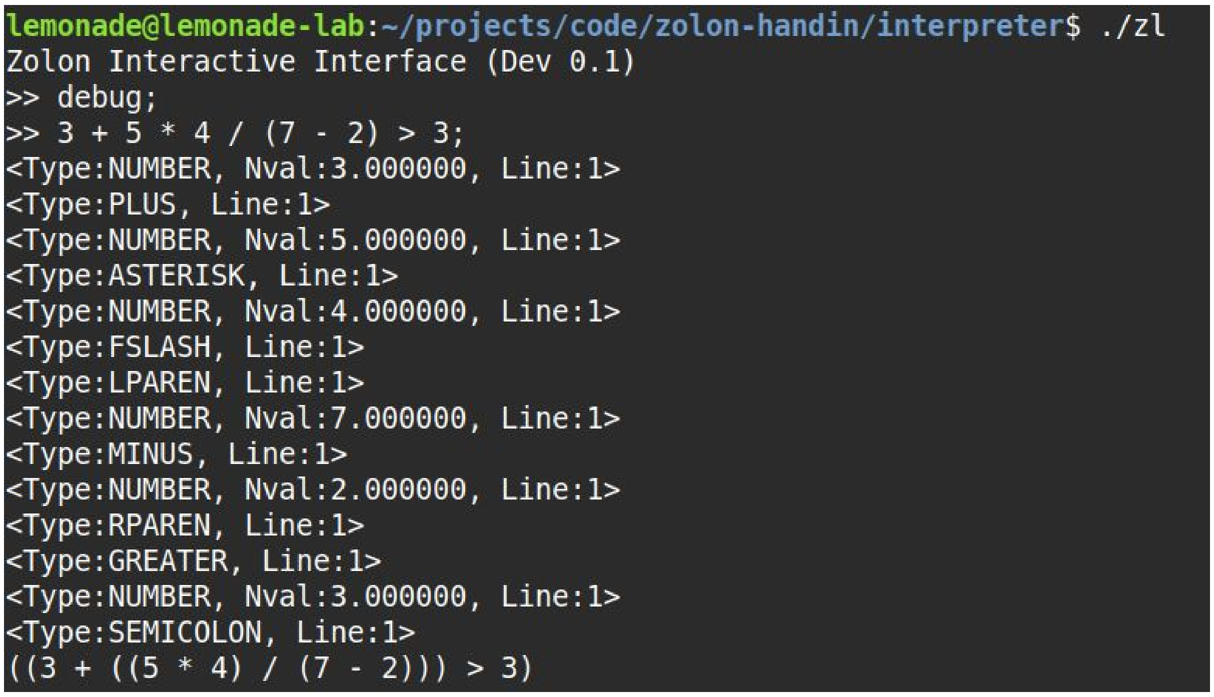
The full list of terminals in the grammar (lexeme types) is as follows:

PLUS, ASTERISK, FSLASH, BSLASH, LPAREN, RPAREN, COMMA, CARET, COLON, BAR, LBRACKET, RBRACKET, SEMICOLON, NEQUALS, MINUS, MAP, BIND, EQUALS, LEQUALS, LESS, GEQUALS, GREATER, IDENTIFIER, NUMBER, PATH, BTRUE, BFALSE, IMPORT, NOT, AND, OR

7. Syntactical Analysis

The syntactical analysis pass of the interpreter consists of parsing the token vector generated by the scanner according to the grammar previously defined in BNF and generating an abstract syntax tree annotated with information about the syntactic and semantic structure of the statement. Logic for this is located in the Parser class. It's organised as a set of mutually recursive methods, which parse the tree from the top down. This method of parsing is called Recursive Descent.

In order to demonstrate the function of the parser, I will again make use of the debug function of the Interactive Interface:

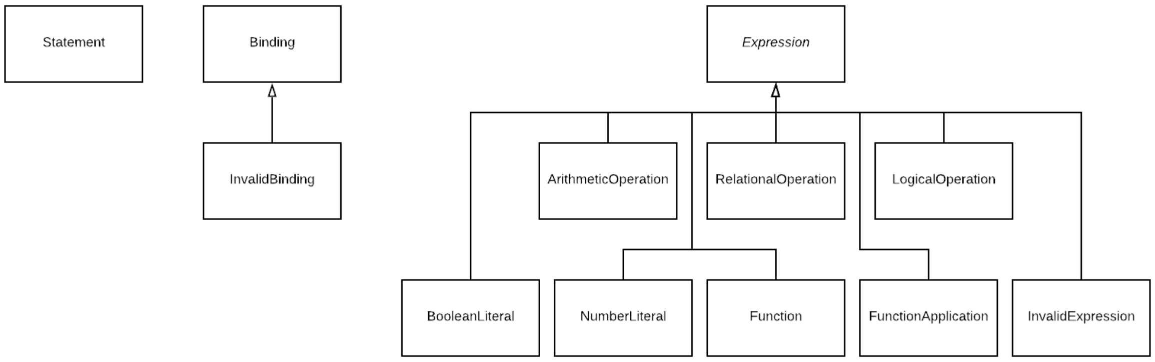


```
lemonade@lemonade-lab:~/projects/code/zolon-handin/interpreter$ ./zl
Zolon Interactive Interface (Dev 0.1)
>> debug;
>> 3 + 5 * 4 / (7 - 2) > 3;
<Type:NUMBER, Nval:3.000000, Line:1>
<Type:PLUS, Line:1>
<Type:NUMBER, Nval:5.000000, Line:1>
<Type:ASTERISK, Line:1>
<Type:NUMBER, Nval:4.000000, Line:1>
<Type:FSLASH, Line:1>
<Type:LPAREN, Line:1>
<Type:NUMBER, Nval:7.000000, Line:1>
<Type:MINUS, Line:1>
<Type:NUMBER, Nval:2.000000, Line:1>
<Type:RPAREN, Line:1>
<Type:GREATER, Line:1>
<Type:NUMBER, Nval:3.000000, Line:1>
<Type:SEMICOLON, Line:1>
((3 + ((5 * 4) / (7 - 2))) > 3)
```

Here, I have inputted a complex expression, containing many operators of different precedences and the interface has generated a fully-parenthesized representation of the tree, which the parser builds.

8. AST (Abstract Syntax Tree)

The syntax tree, which is built during the syntactical analysis pass is implemented as the following class hierarchy of nodes to allow for polymorphism in the rest of the code.



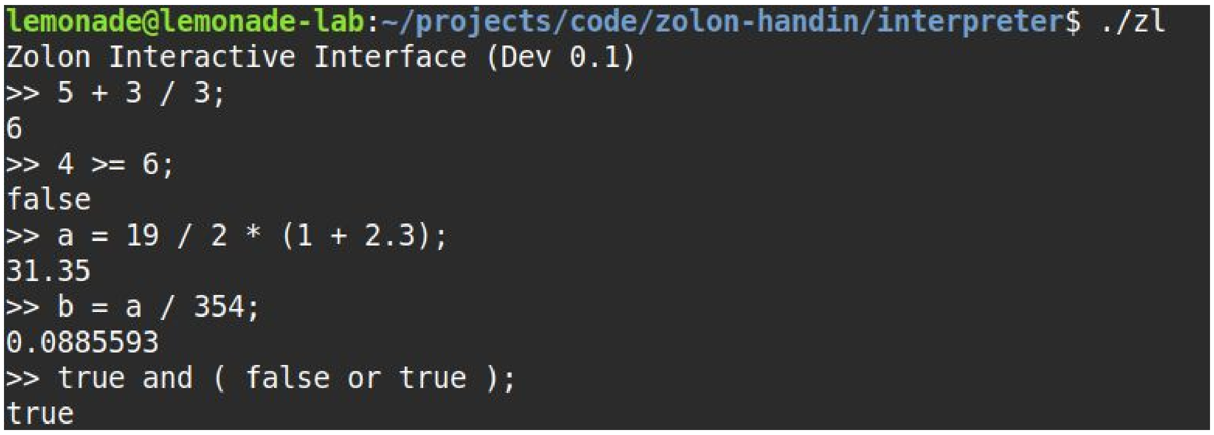
9. Evaluation

The ideal design pattern for evaluating an object-oriented syntax tree like the one in this project would be the Visitor pattern. This would improve ease of modification greatly, reduce the amount of logic in the tree node classes and adhere to the open/closed object-oriented principle. However, due to time concerns, I have implemented the Gang of Four (GaF) interpreter pattern instead. This consists of implementing a specific evaluation function for each type of node in the tree, which in turn calls the evaluation function of its children.

When evaluating expressions, a Value object is produced, which represents a runtime value in the Zolon. Expression statements are evaluated simply by displaying the final Value to the user.

In order to evaluate binding statements, an Environment class is defined, which carries a map of the bound names and values in the current frame and a pointer to the parent environment. (the environment hierarchy is not currently used, but will be in the future to implement function scope). Binding statements are evaluated by adding new bound name to the current Environment an displaying its value to the user.

The final output of the evaluation can be viewed in the Interactive Interface even without debug mode. Here are a few examples:



```
lemonade@lemonade-lab:~/projects/code/zolon-handin/interpreter$ ./zl
Zolon Interactive Interface (Dev 0.1)
>> 5 + 3 / 3;
6
>> 4 >= 6;
false
>> a = 19 / 2 * (1 + 2.3);
31.35
>> b = a / 354;
0.0885593
>> true and ( false or true );
true
```

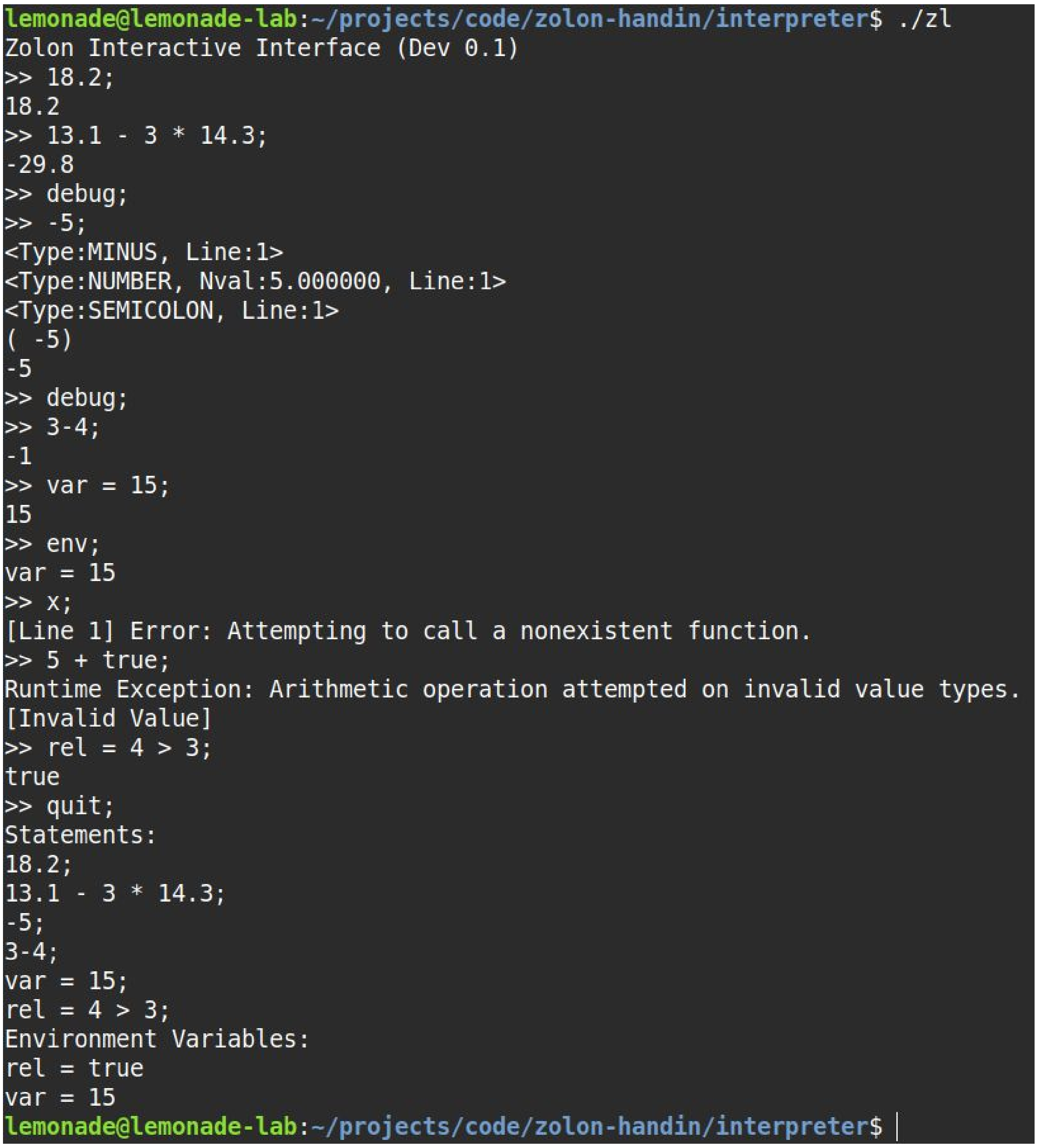
Where did the Semantic pass go?

Since the current typesystem is quite simple, I have opted to do dynamic typechecking in the evaluation functions. When the language's semantics become more complex, it will make sense to extract out the typechecking logic into its own interpreter pass along with extra semantic analysis functionality, such as scope resolution. This would help reduce eventual complexity.

10. Acceptance Tests

Test ID	Actions to undertake	Expected results
1	Run the executable with no arguments.	The Interactive Interface starts up.
2	Input a number or boolean literal into the REPL, followed by a semicolon.	The same literal is echoed back.
3	Input a valid arithmetic expression into the REPL, followed by a semicolon.	The value of the expression is displayed.
4	Input “debug;” into the REPL, followed by any other statement.	Detailed debug information about the tokens and syntax tree is displayed as well as the normal evaluation.
5	Input “debug;” into the REPL, followed by any other statement.	Only the evaluation is displayed.
6	Input a valid binding statement into the REPL, then input “env;”.	The bound value is displayed in the list of environment variables.
7	Input a previously unused identifier into the REPL, followed by a semicolon.	“Attempting to call a nonexistent function.” error is displayed.
8	Input an arithmetic operation statement into the REPL on values of type Boolean and Number.	“Runtime Exception: Arithmetic operation attempted on invalid value types.” is displayed, followed by “[Invalid Value]”.
9	Input a binding statement to a relational operation on two numbers into the REPL.	Either “true” or “false” is bound to the identifier and displayed.
10	Input “quit;” into the REPL.	Lists of the valid statements and environment variables are displayed before exiting the Interactive Interface.

Results:



```
lemonade@lemonade-lab:~/projects/code/zolon-handin/interpreter$ ./zl
Zolon Interactive Interface (Dev 0.1)
>> 18.2;
18.2
>> 13.1 - 3 * 14.3;
-29.8
>> debug;
>> -5;
<Type:MINUS, Line:1>
<Type:NUMBER, Nval:5.000000, Line:1>
<Type:SEMICOLON, Line:1>
( -5)
-5
>> debug;
>> 3-4;
-1
>> var = 15;
15
>> env;
var = 15
>> x;
[Line 1] Error: Attempting to call a nonexistent function.
>> 5 + true;
Runtime Exception: Arithmetic operation attempted on invalid value types.
[Invalid Value]
>> rel = 4 > 3;
true
>> quit;
Statements:
18.2;
13.1 - 3 * 14.3;
-5;
3-4;
var = 15;
rel = 4 > 3;
Environment Variables:
rel = true
var = 15
lemonade@lemonade-lab:~/projects/code/zolon-handin/interpreter$ 
```