MiniLang Compiler

Dylan Frendo (138497M)
CPS 2000 - Compiler Theory and Practice

Department of Computer Science
University of Malta

May 18, 2017

Contents

1 Lexer

The lexer will be implemented using the table-driven approach which encodes the DFA transition function of the MiniLang micro-syntax. The lexer will be implemented in such a way that lexical errors found in the input program will be detected and showing the line number of the error.

1.1 Design

Before starting to implement the lexer, the finite-state automata for MiniLang was designed. The finite-state automata can be seen in Figure 1. As it can be seen in Figure 1, there are in total 20 states and the starting state is S00. The finite-state automata was designed to keep a low amount of states as possible without interfering with the logic and the cleanliness of the design. The lexer will continue reading until the EOF is found. When the EOF is found, the lexer stops are the lexical analysis is completed and the parser takes over.

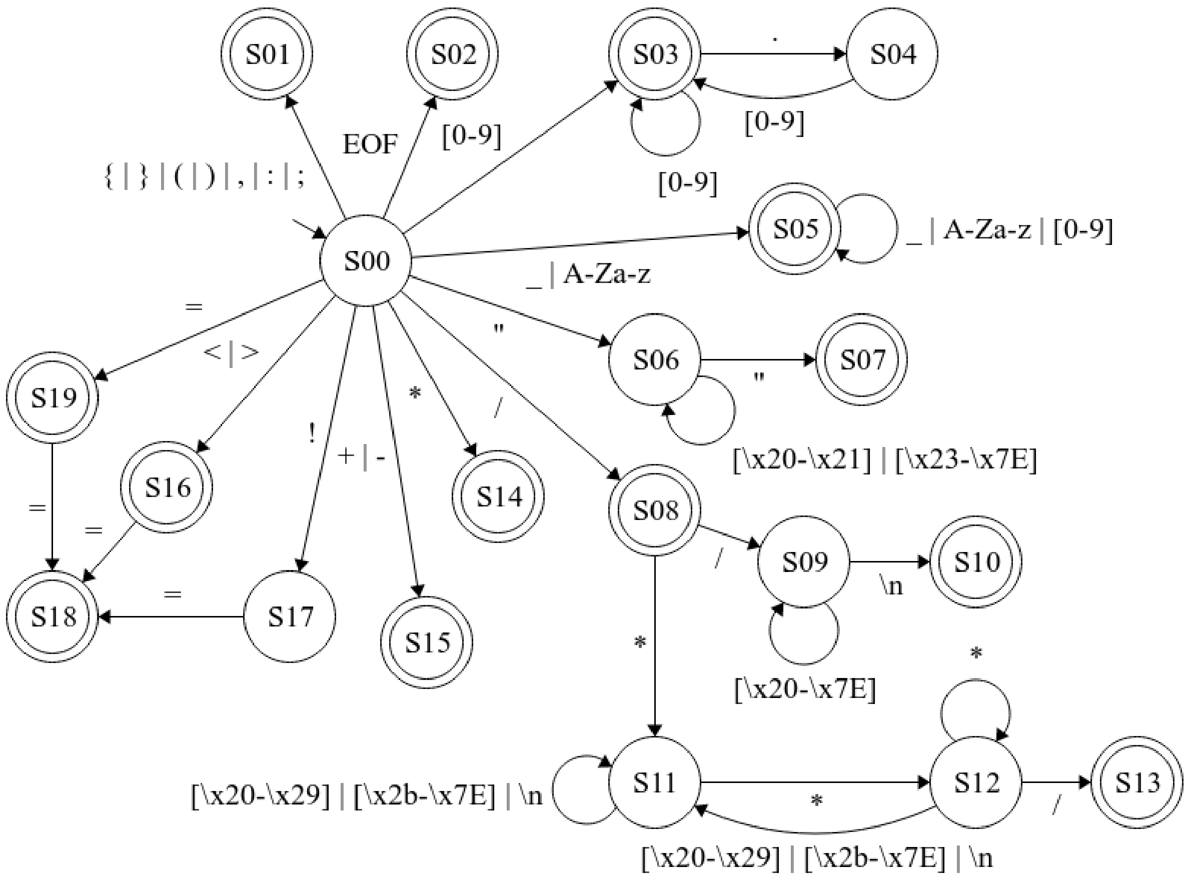


Figure 1: Finite-State Automata for MiniLang.

The Classifier Table of the lexer can be found in Table 1. There are in total 16 classifiers. The number of the classifier is used for the Transition Table for MiniLang. The amount of

Classifier Name	Number	Value
Punctuation	0	{ } ( ) , : ;
End of File	1	EOF
Number Literal	2	0-9
Letter	3	A-Z a-z
Underscore	4	_
Double Quotation Marks	5	”
Printable	6	\x20 - \x7E
Multiplier	7	*
Division	8	/
Operators	9	+ -
New Line	10	\n
Relational Operators	11	><
Exclamation Mark	12	!
Equals	13	=
Full Stop	14	.
Other	15	Any remaining characters

Table 1: The Classifier Table for MiniLang.

classifiers is depended on the finite-state automata. The classifier New Line was added for single line and multi line comments, other expressions will discard the new line as it will be explained further on.

Combining both the finite-state automata and the Classifier Table, the Transition Table can be obtained. The Transition Table can be seen in HTML format in the folder tests-documentation/TransitionTable.html found in the zip file. The transition table was implemented as a 2-D array. The lexer will start by reading the input of a program character by character and move to different states. White spaces and new lines will only be discarded if the lexeme contains no characters. This is done since new line are needed to show an end of a single line command and they cannot be simply discarded from the program completely. The next state depends on the classification of the current character. This is repeated until a S_ERR is found and a lexeme/word is formed. The program will rollback states until an accepting state is found, decreasing the lexeme will each rollback. If an accepting state is not found, that indicates that the syntax of the inputted program is not correct. If the accepting state is found, the lexeme is turned to a token. There are in total 32 Tokens for MiniLang and they can be seen in Table 2. The tokens were fit into an enum and a class container holding the token, name and value will be used. The amount of tokens could be reduced but decided not to avoid pattern matching in future sections.

Token	Value	State Created
TOK_EOF	EOF	S02
TOK_LeftCurlyBracket	{	S01
TOK_RightCurlyBracket	}	S01
TOK_LeftParenthesis	(	S01
TOK_RightParenthesis	)	S01
TOK_Comma	,	S01
TOK_Colon	:	S01
TOK_SemiColon	;	S01
TOK_IntegerLiteral	Integer Values	S03
TOK_RealLiteral	Real Values	S03
TOK_RealType	real	S05
TOK_IntType	int	S05
TOK_BoolType	bool	S05
TOK_StringType	string	S05
TOK_BooleanLiteral	true/false	S05
TOK_Logic	and/or/not	S05
TOK_Set	set	S05
TOK_Var	var	S05
TOK_Print	print	S05
TOK_Return	return	S05
TOK_If	if	S05
TOK_Else	else	S05
TOK_While	while	S05
TOK_Def	def	S05
TOK_Identifier	Name of a variable	S05
TOK_Printable	Arrays of characters containing \x20 - \x7E	S07
TOK_MultiplicativeOperator	*/ ’/’	S08, S14
TOK_Comment	Single or multi-comments	S10, S13
TOK_AdditiveOperator	+/-	S15
TOK_RelationalOperator	</>/!=/<=/>=	S16, S18
TOK_Equals	=	S19
TOK_Error	Any errors found	State not found

Table 2: Tokens used for MiniLang.

2 Parser

A top down hand-crafted recursive descent parser will be implemented for MiniLang. After the lexical analysis, the lexer will pass the tokens via getNextToken() function found in the lexer. The parser will report any syntax errors in the input program and will produce an abstract syntax tree if no errors were encountered.

2.1 Design

The parser will continue to call the getNextToken() function until the token indicating the EOF is found. A valid program contains 0 or many statements thus the parser will parse statements until the EOF is found. The parser follows the EBNF given for MiniLang almost completely. Since the parser is implemented recursively, with every function call an AST node is return. The type of the node depends on the current tokens to be executed and all nodes used can be seen in Subsection 2.2. The parser can detect if an unexpected token is found and if so the parser will output the error and stoping the execution afterwards. The parser also has a functionality given by the lexer to preview the next token instead of consuming the token. This is needed for EBNF expressions that requires 0 or many other EBNF expressions or optional expressions. The following shows an example (Expression parsing) on how the parser works:

The parseExpression() method is called everytime there is <Expression>in the EBNF. parseExpression() will call parseSimpleExpression(). parseSimpleExpression() will call parseTerm() and parseTerm() will call parseFactor(). parseFactor() has 5 options to choice from. The nextToken is called to check which option is next. These can be

- parseLiteral() will check if the current token is either a BooleanLiteral, IntegerLiteral, RealLiteral or a stringLiteral. Since the literals are already handled by the lexer, the type of the token is only checked and an AST Literal Node is returned depending on the literal.

- To check if the next token is an identifier, the type of the token is checked since the identifier is already handled by the lexer. An AST Identifier node is returned.

- After checking the identifier, parseFunctionCall() is called to check if the identifier belongs to a function. The next token is previewed to check if it is a left parenthesis token. If it is, that indicated that the identifier belongs to a function call. The parseActualParams() is called to handle the parameters passed recursively. parseActualParams() will call parseExpression() restarting this whole process. parseActualParams() can throw an exception if no comma token is found between expression or after a comma token there is the right parenthesis token. After finishing parsing the parameters, parseFunctionCall() will check if there is a right parenthesis. If there is not, an exception is thrown. An AST function call node is returned.

- If a left parenthesis is found, that indicates that it is a subexpression. parseExpression() is called and after returning the expression, the next token is checked to see if it a right parenthesis. An AST expression node is returned and the type of the expression node depends on the return of the parseExpression() method.

- If the current token contains the operators '-' or 'not', it indicates that it is an Unary. parseExpression() is called to handle the expression of the unary. An AST Unary is returned.

If the token does not satistfy any options of the parseFactor() that indicates that the program syntax is incorrect and thus throwing an exception. parseTerm() previews the next token to see if it a multiplication operator. If it is, the operator token is taken and the parseTerm is re-called. Since there are now 2 expression node, AST Binary Expression Node is returned by parseTerm(). If there is no operator, the return of parseFactor() node is returned. This is repeated for parseSimpleExpression() and parseExpression(). parseExpression() will then return the AST Expression Node of the parsed expression ending the parsing of an expression.

2.2 Abstract Syntax Tree

The abstract syntax tree of the input program is created by the parser. The AST will use the Visitor design pattern so that visitor classes can be created. There are 3 main nodes in order to create the AST for MiniLang. These are ASTNode, ASTStatementNode and ASTExpressionNode.

2.2.1 AST Node

The ASTNode is the main node that contains the 0 to many statements of the input program. The parser will start to evaluate the 0 or many statements of the input program. The statements will be pushed into a vector and the vector containing the statements will be passed to the AST Node. Thus AST Node is the main containing of the input program. The class hierarchy of AST Node is seen in Figure 2.

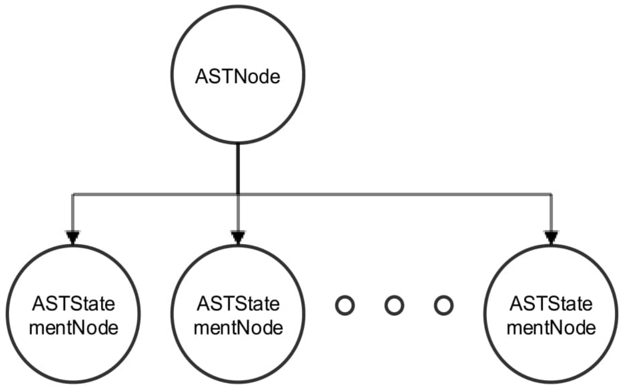


Figure 2: Class Hierarchy for AST Node.

2.2.2 AST Statement Node

Figure 3 shows the class hierarchy of the statement Node. The class AST Statement Node itself is an abstract class and thus a statement node can be one of the children which are shown in the figure.

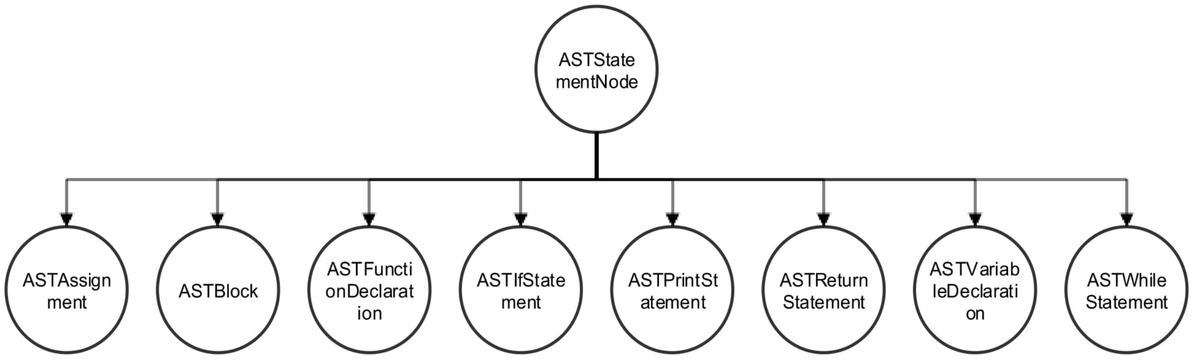


Figure 3: Class Hierarchy for AST Statement Node.

2.2.3 AST Expression Node

Figure 4 shows the class hierarchy of the expression Node. The class AST Expression Node itself is an abstract class and thus a statement node can be one of the children which are shown in the figure. It is to node that ASTBinaryExprNode is used to hold all operators used for MiniLang. Also ASTLiteralNode is an abstract class as well.

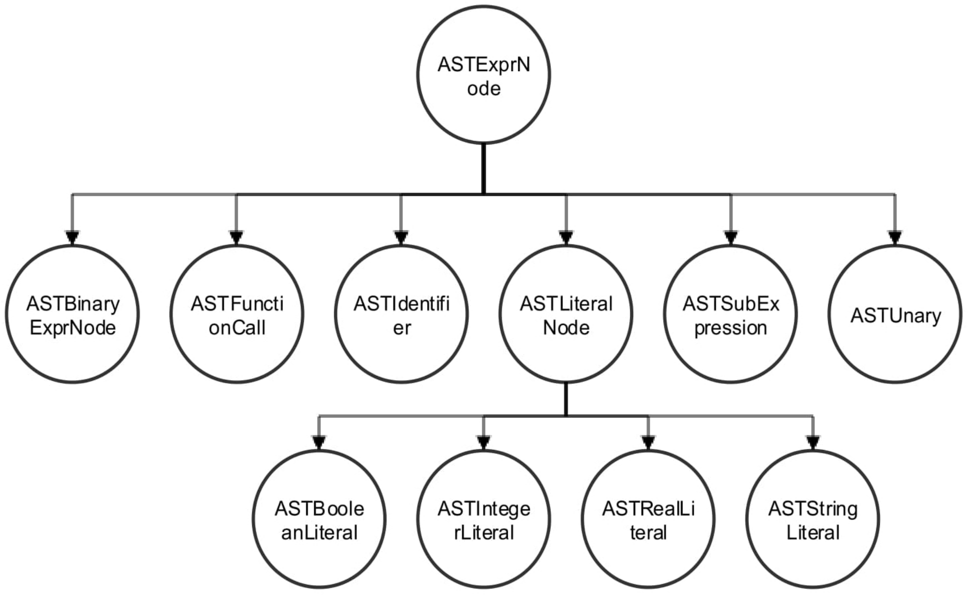


Figure 4: Class Hierarchy for AST Expression Node.

2.3 XML of Abstract Syntax Tree

A visitor class was used to output a properly indented XML representation of the AST produced by the parser. A global variable was held to know the current indentation. The indentation is incremented when a block is entered or there is a binary expression node. It is decremented when the block exits or the binary expression node exits. Some operators cannot be printed normally, for example < will result in an error and thus special XML characters are used.

3 Semantic Analysis

Another visitor class was used to do the semantic analysis pass of MiniLang. A class Scope is used to hold all identifiers found in a scope. Identifiers need to be unique in the scope and thus map is used with the name of the identifiers as keys. The value of the map is another class created called SymbolTable where it holds the return Token type of the identifier and if the identifier is a function, it holds the parameters. Whenever a block is entered, a scope is pushed into a stack. If the block entered is that of a function declaration, the parameters are pushed as identifiers. When there is a variable or function declaration, the identifier is only compared with the current Scope to check for duplicated. When there is an assignment, all the scopes are checked and the most recent declared identifier is used. Some semantic rules regarding declaration:

- Identifiers are unique to their current scope.
- Functions can be declared inside other functions similar to scala.

To check if variables are assigned to appropiately typed expressions, a global Token declared variable (lastToken) is used that holds the type of the Token of the last used expression. For example if the expression node is of ASTIntegerLiteral, the lastToken is set to IntType. If the expression contains several operators indicating it is an ASTBinaryExprNode, the left node is first evaluate and lastToken is held as a local variable. Similaraly the right node is evaluated and lastToken is also held as a local variable. The 2 local variables containing the tokens type and the operator are sent to a function that determines if the two tokens are assigned properly to the passed operator. If the operator does not handle the token type, an error is indicated. If the types do not match another error is indicated. Some semantic rules regarding expressions:

- If/While will only accepts true/false expressions similar to java.
- Check if the operators are accepted given the variables types. This can be seen in Table 3. The operator + can accept 2 strings and it indicates string concatenation. The division by 0 is allowed by the semantic analysis but not allowed in the Interpreter. The Interpreter will stop the execution if division by 0 is encountered.
- If the expression type is integer and real type is needed, the integer is automatically turned to real. This is to avoid to use number.0 to declare real type variables. Integer will not accept any real values on the otherhand.

	Int	Real	Bool	String
*	Y	Y	N	N
/	Y	Y	N	N
and	N	N	Y	N
+	Y	Y	N	Y
-	Y	Y	N	N
or	N	N	Y	N
not	N	N	Y	N
<	Y	Y	N	N
>	Y	Y	N	N
==	Y	Y	Y	Y
!=	Y	Y	Y	Y
<=	Y	Y	N	N
>=	Y	Y	N	N

Table 3: Operators type acceptance where Y indicates accepted, N indicates not accepted.

Functions are also checked to see if they all have a return regardless of the path taken and if the return type is correct. The semantic analysis was made to handle all the paths if/else statements in that block to have a return or a global return (not inside an if/else block) inside the function itself. Multiple if/else inside each other are also allowed and checked accordingly. This was done by having a struct called ReturnCheckForFunctionDeclaration. This struct has 6 variables. isFunctionDeclarationBlock boolean is made to check if the next block that will be traverse is that of a function. If it is, the actual params need to be added to the new scope when parsing the statements of the function. The actual params can be found in the functionDeclaration pointer that contains the information of the function declaration. The variable numberOfIfEncountered contains the number of if statements encountered. The variable numberOfReturnsEncountered contains the number of returns encountered. These two variables are needed to check if all the if/else statements are covered by a return. These variables are discarded if the boolean isReturnFoundGlobal is true since this indicates that a return statement is found outside the if/else statements indicating that the function guarantees a return. For the return statement to indicate that the return is not inside an if/else statement, the variable currentBlockIndex is used. This variable indicates the current depth of if/else statements were 0 indicates it is not inside any if/else statements. By this method, the semantic analysis guarantees that every function will have a return type and whenever a return is found, the type is checked. Also the return cannot be used if it is not currently inside a function. In short the semantic rules are:

- The function calls arguments and the function declaration arguments need to be matched to check if the type are correct.
- All possible paths the program can take in a function will end up to a return. For example if there is an if/else statement and only if has a return, the function is not accepted. Multiple if/else statements inside each other return check is handled as well.

- The return type is checked that it matches with the function declaration type.
- Return cannot be used outside a function.
- Recusive functions are supported. Measurements were taken so that functions identifiers are not added to the symbol table if an error is encountered in the function declaration block.

4 Interpreter Execution Pass

Another visitor class was implemented that traverse the AST tree and executed the program. The implementation resembles a lot to the one that semantic analysis uses. The interpreter was also implemented to have scopes and all the scopes of the program are kept in a stack. Like in the semantic analysis, the identifiers are kept in the scope they are declared. If a variable is re-declared and changed in another scope, the previous declared variable will not be changed. The identifers and the functions were kept in different maps. The reason behind this is that the evaluation of the identifiers will be changed constantly and thus to not introduce any unwanted errors, the list of the functions created are kept seperately.

To evaluate an expression, an instance of class Evaluation was kept globally in the interpreter similar to lastToken in semantic. Evaluation contains all the types of the program (int, real, bool, string) and contains an enum of these type to indicate the last set type of the evaluation. With this class, the interpreter can know the evaluation by getting the value indicated by the evaluation's last set type. The return will be assumed to be right since the type checking was done by the semantic analysis. The evaluations will only change when an expression is evaluated. Identifiers will all have their unique Evaluation since identifiers evaluations can be used further on in the program. The interpreter was programmed to not de-allocate evaluation memory that is used by the identifiers, but de-allocate memory that will not be used for example anonymous expressions evaluation. Also if the current evaluation is integer but real is required, the integer is accepted by both the semantic analysis and the interpreter. If the current evaluation is real and integer is required, it is not accepted. This was done to avoid typing number.0 to declare a real number.

When there is a function call, the parameters passed in the function call need to be matched with the identifiers of the function in the new scope of the function block. The interpreter accomplish this task by setting a global boolean variable to true, indicating that the next block that will be executed is that of a function. The parameters that are passed in the function call are linked globally so that Block method can access them. The Block method will then link the parameters passed with the identifiers of the function declared and add them to the new function scope. The interpreter will stop executing the block when the first return is found.

5 Read Execute Print Loop

The REPL will start to read statements when the compiled code is executed. When implementing the REPL, some of the previous implements needed to be adjusted.

The lexer was adjusted to end the tokenization with both EOF and the end of a string indicated by the character '\0' since MiniLangI can both accept files as input or commands from the terminal. Another change made to the lexer was that syntax errors did not stop the program like previously but threw an exception with the type of the error. The exception is caught by MiniLangI are displayed to the user. The user can then enter other commands afterwards. The lexer was also made to reset after executing all the commands passed. This was done so that one instance of the lexer is enough.

The parser and AST Tree structure hierarchy were also changed. A new statement node was added with the other statements. The new statement node, called ASTExprStatement expects an expression followed by a semi-colon after the expression as seen in Equation 1. The statement was added since the REPL can now accepts an expression as well with the other statements. The parser was also adjusted to check for expression statements when checking the next statement. It is to note that expressions still require a semi-colon to be accepted. An example of the new statement is "34 + 46;".

$$< ExpressionStatement > \quad ::= \quad < Expression >';' \tag{1}$$

The visitors classes of the semantic analysis and interpreter were also changed. The Scope containing the symbol table used in the main node, AST Node, was changed to be global and publically available in both semantic analysis and interpreter visitors. This change was required since the symbol table in the scope is almost updated with every statement passed and the previous identifiers are still valid. Thus the symbol will remain without being de-allocated until the deconstructor is called. With this changed, the previous statements do not require to be executed again, the new statements only are executed. After the execution of the visitors classes on the new statements, the new statements are added to the previous statements forming an AST Node containing all the statements passed. This AST Node is then executed by the XML parser to re-generate the XML for the whole program. The special variable "ans" that holds the last execution of either variable declaration or assignment was put in the symbol table when it was being declared in both semantic and interpreter. If the user tries to re-declare the variable, he will be unsuccessful. The variable "ans" was made to only be shown if there was a variable declaration, assignment statement and an expression statement in which all of them are not inside a block statement.

MiniLangI was also programmed to accept special commands starting with #. These commands are the following:

- *#help* will display all the commands available.
- *#load* will load a file and be executed with the MiniLangI syntax. The file location needs to be in quotation marks and the extension of the file needs to be .gulp.

- *#st* will print the current variables and function delclared found in the symbol table. The special variable ”ans” is not printed.
- *#quit* will quit MiniLangI.

Also MiniLangI was also implemented to accept multiple command statement like a function declaration. MiniLangI will enable multi-line commands when there is a left curly bracket and will stop when the right curly bracket is inserted. Multiple curly bracket are supported.

6 General Comments

The compilation of the program can be done by cmake since the compilation specific was not specified. To check for memory leaks, valgrind was used and every leak found was fixed. Bugs encountered were also fixed. This does not guarantee the program is bug and leak free.

7 Testing

The compiler will be tested by showing the inputted program and comparing the expected result with the actual result. The XML representing the AST structure will not be included in the documentation but can be seen in the test folder with the source code.

7.1 Program 1

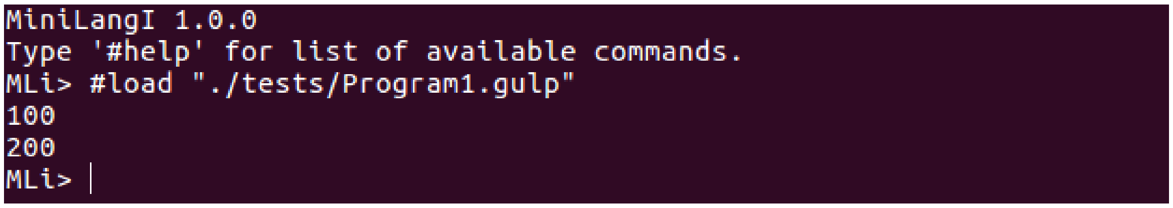
The first program to be tested can be seen in List 1. This program aims to test a function declaration with multiple return type in which the first return traverse should stop the function call. It is testing an if statement, variables declaration and assignments, function declaration and return of a function, the print statement, function call parameters check, single line comment and multiple line comment. It is also testing that a real variable can also take int as a value. This was done for a quality of life so that a real can also be declared without using number.0.

```
/**
 * Test 1;
 * Function that returns the maximum out of 2 numbers.
 */
def max(x : real, y : real) : real {
    if (x > y) {
        return x;
    }
    return y;
}

// X is int but function accepts real. Semantic analysis and interpreter
// should accept this.
var x : int = 23;
// Real but expression is 0, should be accepted.
var y : real = 100;
// Should print 100;
```

```
print max(x, y);

set x = 200;
// Should print 200
print max(x, y);
```

Listing 1: Program 1 source code. Returns the maximum number.

The program AST in XML format can be found in the folder tests-and-documentation/ Program1.xml in the zip file. The compiler is expected to output 2 times indicating the maximum value of the 2 variables. Thus the program should output 100 and afterwards 200. The actual output is as the expected output as seen also in Figure 5.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> #load "./tests/Program1.gulp"
100
200
MLi> |
```

Figure 5: Program 1 output. Returns the maximum number.

7.2 Program 2

The source code to be tested can be seen in List 2. The program is aimed to test some of the possible expression evaluations with operators. It is a reminder that the operators follow BODMAS. The expected output of the program should be ”This is a string test.”, 10, 11, 17, 66.5, 401, false and true. This output of the program was exactly that as can be seen in Figure 6. The program AST in XML format can be found in the folder tests-and-documentation/Program2.xml in the zip file.

```
/*
 * Test 2.
 * The program will be tested for expression validation.
 */
// This should concatenate the strings.
print "This is a " + "string test.";
// This should print 10.
print (3 + 2) * (6 - 4);
// This should print 11.
print 3 + 2 * 6 - 4;
// This should print 17.
print (6 + 5) * 3 - 4 * 4;
// This should print 66.5
print 6.5 + 30 * 2;
// This should print 401.
print 100.25 / 0.25;
// This should print false.
print 6 * 4 + 5 < 6 + 4 * 5;
// This should print true.
```

```
print (true and false) or (not false);
```
Listing 2: Program 2 source code. Expressions evaluations.

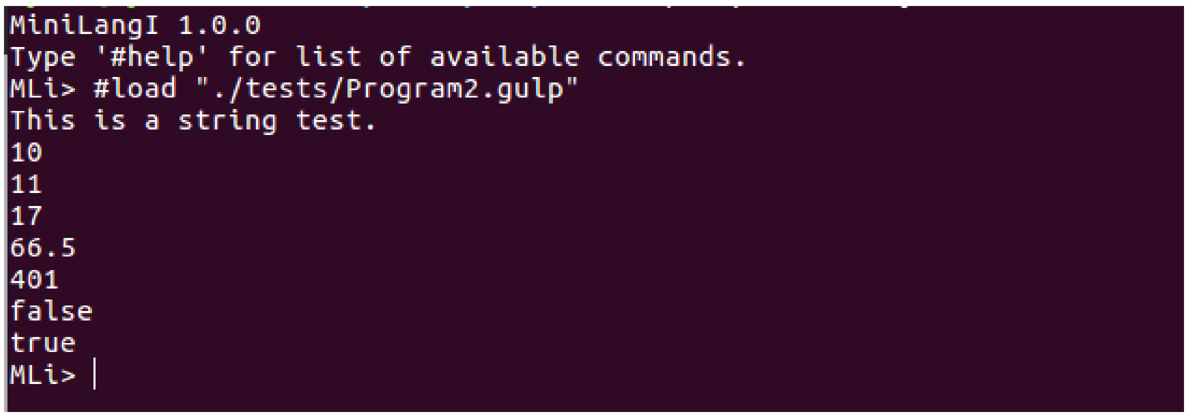

```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> #load "./tests/Program2.gulp"
This is a string test.
10
11
17
66.5
401
false
true
MLi> |
```

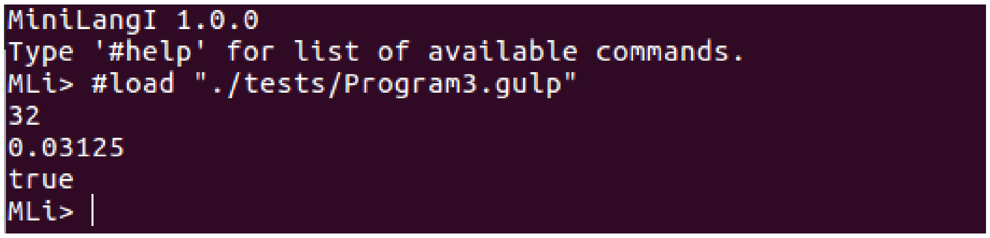
Figure 6: Program 2 output. Expressions evaluations.

7.3 Program 3

The source code to be tested can be seen in List 3. The program tests a function declaration that is used multiple times, while and if/else statements and variable and assignment statements. The expected output of the program is 32, 0.03125 and true. The actual output is the same of the expected output as can be seen in Figure 7. The program AST in XML format can be found in the folder tests-and-documentation/Program3.xml in the zip file.

```
/*
 * Test 3.
 * Function definition for Power.
 */
def funcPow(x : real, n : int) : real {
    // Declare y and set it to 1.0
    var y : real = 1.0;

    if (n > 0) {
        while (n > 0) {
            // Assignment y = y * x;
            set y = y * x;
            // Assignment n = n - 1;
            set n = n - 1;
        }
    } else {
        while (n < 0) {
            //Assignment y = y / x;
            set y = y / x;
            // Assignment n = n + 1;
            set n = n + 1;
        }
    }
    // return y as the result;
    return y;
}

var temp : real = 2;
var temp2 : int = 5;
// Should print 32
print funcPow(temp, temp2);

set temp2 = −5;
// Should print 0.03125
print funcPow(temp, temp2);

set temp2 = 10;
// Should print true.
print funcPow(temp, temp2) > − 34;
```

Listing 3: Program 3 source code. Power function.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> #load "./tests/Program3.gulp"
32
0.03125
true
MLi> |
```

Figure 7: Program 3 output. Power function.

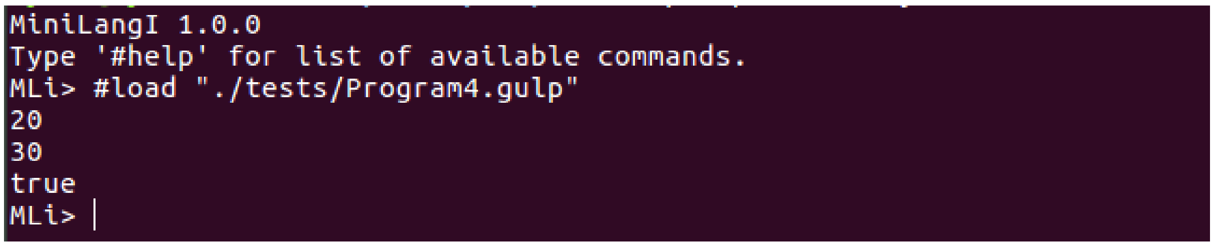
7.4 Program 4

The source code to be tested can be seen in List 4. The program tests several functions declarations, a function declaration inside a function declaration and some if statements, variable declarations and prints statements. The expected output is 20, 30 and true. The actual output matches the expected output as seen in Figure 8. The program AST in XML format can be found in the folder tests-and-documentation/Program4.xml in the zip file.

```
/*
 * Test 4.
 * Testing multiple function declarations.
 */
def max(x : int, y : int) : int {
    if (x > y) {
        return x;
    }
    return y;
}

def min(x : int, y : int) : int {
    def minHelper(x : int, y : int) : int {
        if (x < y) {
                return x;
```

```
            }
            return y;
    }
    if (x == minHelper(x, y)) {
        return x;
    }
    return y;
}

var x : int = 20;
var y : int = 30;
// Should print x (20);
print min(x, y);
// Should print y (30);
print max(x, y);
// Should print true
print min(x, y) < max(x, y);
```

Listing 4: Program 4 source code. Multiple functions declarations.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> #load "./tests/Program4.gulp"
20
30
true
MLi> |
```

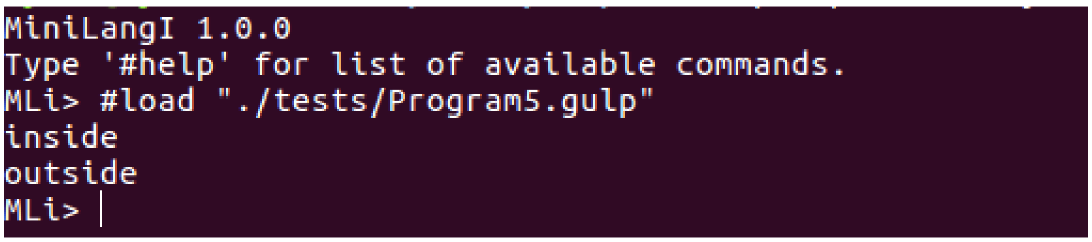
Figure 8: Program 4 output. Multiple functions declarations.

7.5 Program 5

The source code to be tested can be seen in List 5. The program aims to test the declaration of a new scope by initialising a block statement. The same variable is re-declared in another scope and thus it should not affect the previous variable. The expected result in ”inside” and ”outside”. The actual output matches the expected output as seen in Figure 9. The program AST in XML format can be found in the folder tests-and-documentation/Program5.xml in the zip file.

```
/*
 * Test 5.
 * Multiple variable declaration in different blocks and Scopes.
 */
var x : string = ”outside”;
{
    var x : string = ”inside”;
    print x;
}
print x;
```

Listing 5: Program 5 source code. Scope testings and multiple variable declarations.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> #load "./tests/Program5.gulp"
inside
outside
MLi> |
```

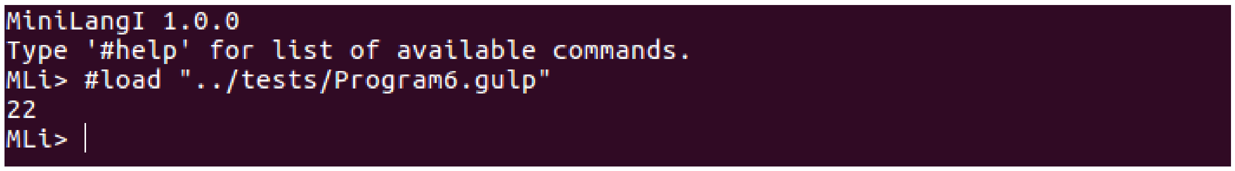
Figure 9: Program 5 output. Scope testings and multiple variable declarations.

7.6 Program 6

The source code to be tested can be seen in List 6. The program aims is to test recursive function. These functions were implemented to act like normal functions. Measurements were taken to ensure that the function identifier is deleted if an error is encountered in the function block. The expected output is 22. The actual output can be seen in Figure 10. The program AST in XML format can be found in the folder tests-and-documentation/Program6.xml in the zip file.

```
/*
 * Test 6.
 * Recursive Function.
 */
def addition(a : int, b : int) : int {
    if (b == 0) {
        return a;
    }
    return addition(a + 1, b - 1);
}

print addition(10, 12);
```

Listing 6: Program 6 source code. Recursive functions.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> #load "../tests/Program6.gulp"
22
MLi> |
```

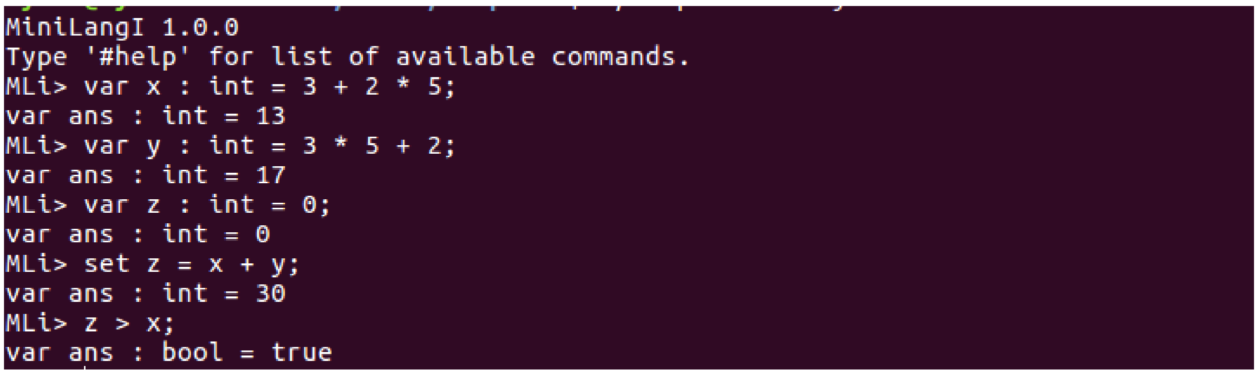
Figure 10: Program 6 output. Recursive Function.

7.7 Testing with REPL

The REPL with be testing for functionality it offers. It will also be used to test different errors that are given by the lexer, parser, semantic analysis, interpreter and the REPL itself.

7.7.1 REPL Testing

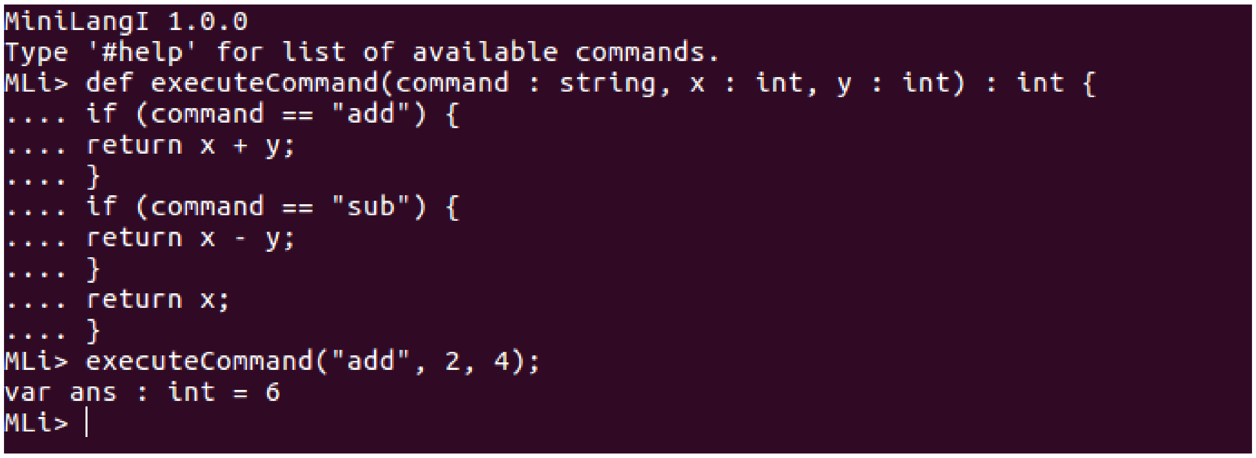
The REPL is expected to run statements like in the previous section. The following will show same statements that are ran by the REPL.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> var x : int = 3 + 2 * 5;
var ans : int = 13
MLi> var y : int = 3 * 5 + 2;
var ans : int = 17
MLi> var z : int = 0;
var ans : int = 0
MLi> set z = x + y;
var ans : int = 30
MLi> z > x;
var ans : bool = true
```

Figure 11: REPL Test 1. Variables declarations and assignments.

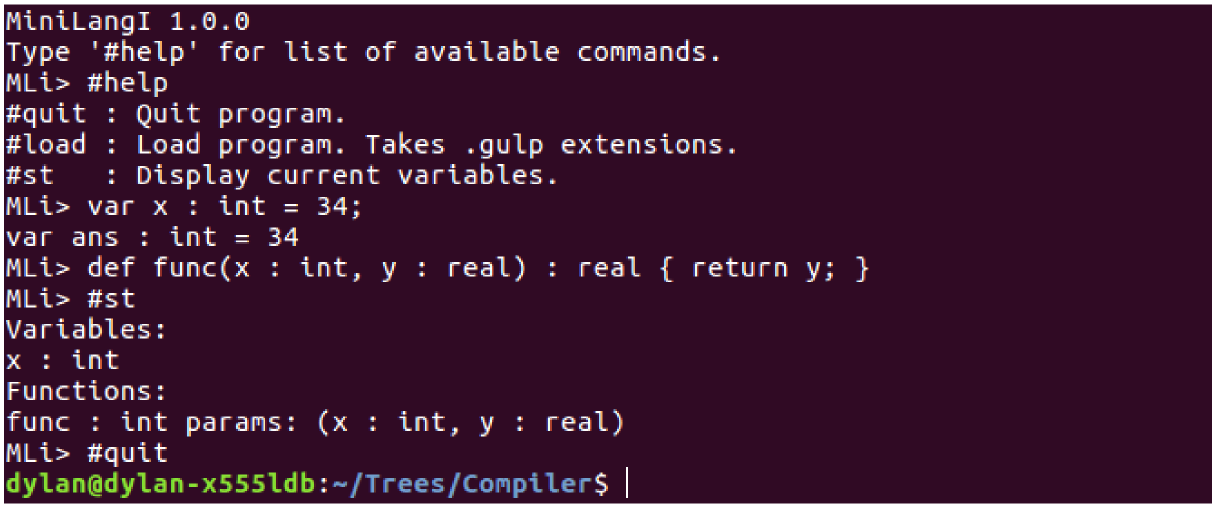
Figure 11 shows same variable declarations and and an assignment. As can be seen, the special variable ”ans” is working as intended by displaying the most recently evaluated expression. Also previous declared variables can be used at any time in the program. The newly added expression statement is also working and the annoynous function evaluated is given to ”ans” as well.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> def executeCommand(command : string, x : int, y : int) : int {
.... if (command == "add") {
.... return x + y;
.... }
.... if (command == "sub") {
.... return x - y;
.... }
.... return x;
.... }
MLi> executeCommand("add", 2, 4);
var ans : int = 6
MLi> |
```

Figure 12: REPL Test 2. Multi-line command.

Figure 12 shows a multi-line command that is allowed by the REPL. A function is created that will string compare the passed string with random commands and then return according to the chosen command. The executed command in this case is an addition. As can be seen, multiple curly brackets are also supported.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> #help
#quit : Quit program.
#load : Load program. Takes .gulp extensions.
#st   : Display current variables.
MLi> var x : int = 34;
var ans : int = 34
MLi> def func(x : int, y : real) : real { return y; }
MLi> #st
Variables:
x : int
Functions:
func : int params: (x : int, y : real)
MLi> #quit
dylan@dylan-x555ldb:~/Trees/Compiler$ |
```

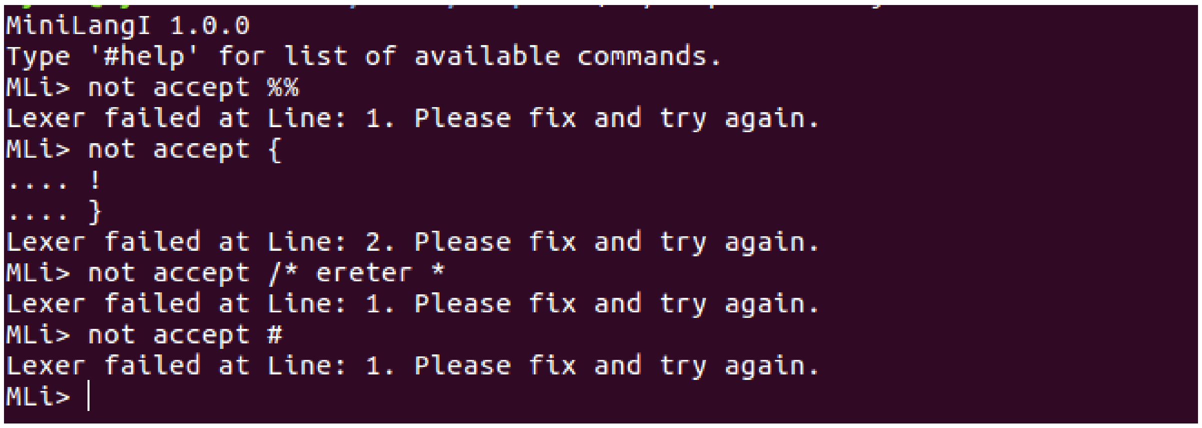
Figure 13: REPL Test 3. Special Commands.

Figure 13 shows the special commands available by the REPL. As can be seen, the #*st* command will print both variables and functions that are currently found in the main Scope of the program. The #*load* command can be seen used in the about screenshots when loading the program used for testing.

7.7.2 Error Handling

The following section will be used to show some of the error handling done.

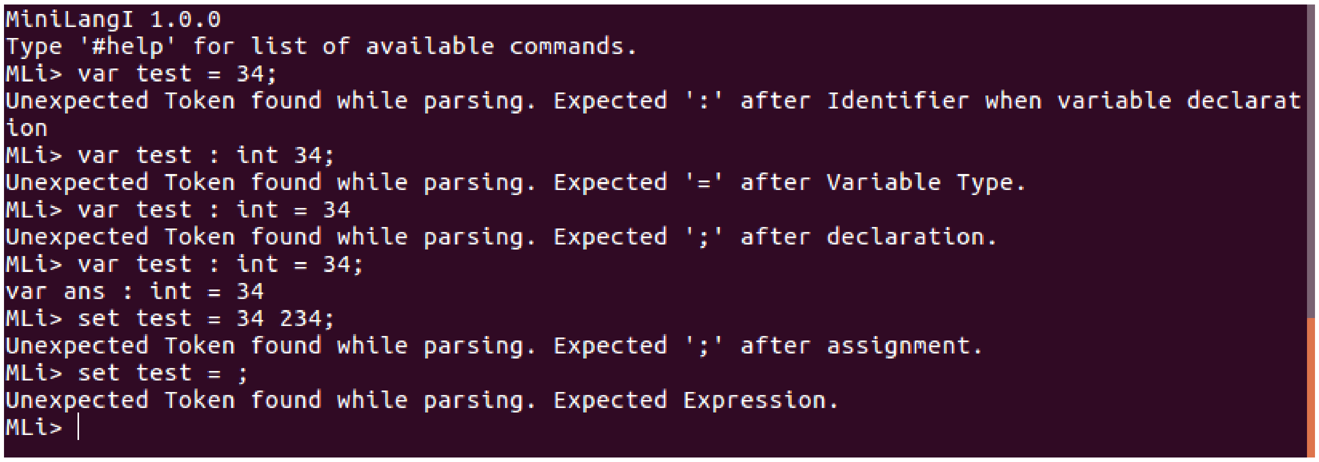
Figure 14 shows some of the lexical errors that the lexer can encounter. As it can be seen, the lexer is able to detect the line number where the lexical error occured. Also if an error is encountered, the REPL will continue to await new commands.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> not accept %%
Lexer failed at Line: 1. Please fix and try again.
MLi> not accept {
.... !
.... }
Lexer failed at Line: 2. Please fix and try again.
MLi> not accept /* ereter *
Lexer failed at Line: 1. Please fix and try again.
MLi> not accept #
Lexer failed at Line: 1. Please fix and try again.
MLi> |
```

Figure 14: Error Handling Test 1. Lexer.

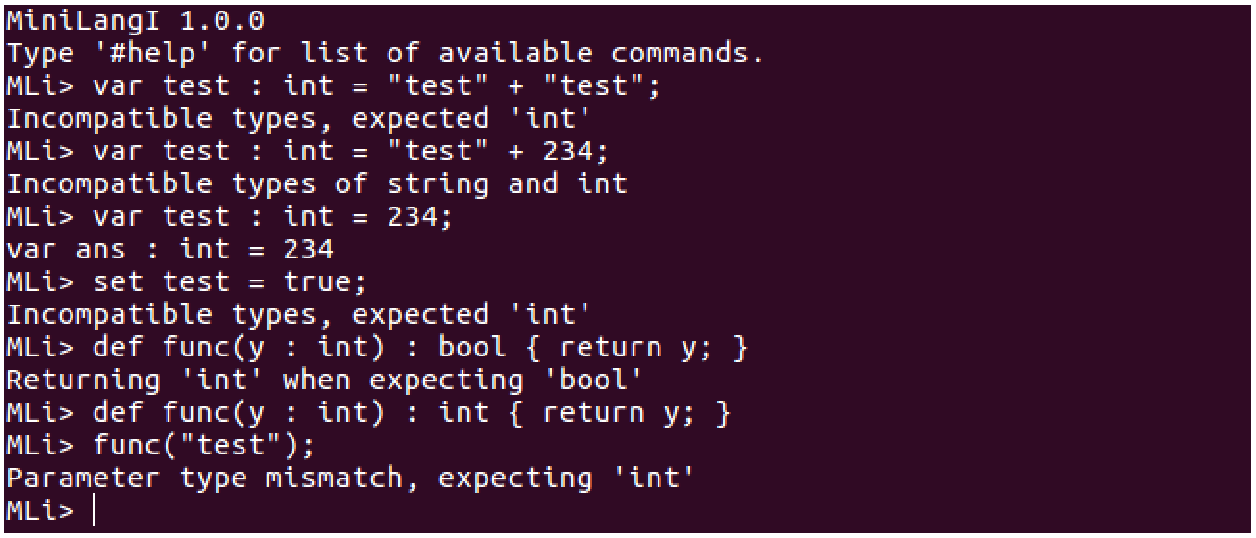
Figure 15 shows some of the errors that can be encountered by the parser. There are many possible errors that can be encountered by the parse and they will not be listed all in this documentation.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> var test = 34;
Unexpected Token found while parsing. Expected ':' after Identifier when variable declarat
ion
MLi> var test : int 34;
Unexpected Token found while parsing. Expected '=' after Variable Type.
MLi> var test : int = 34
Unexpected Token found while parsing. Expected ';' after declaration.
MLi> var test : int = 34;
var ans : int = 34
MLi> set test = 34 234;
Unexpected Token found while parsing. Expected ';' after assignment.
MLi> set test = ;
Unexpected Token found while parsing. Expected Expression.
MLi> |
```

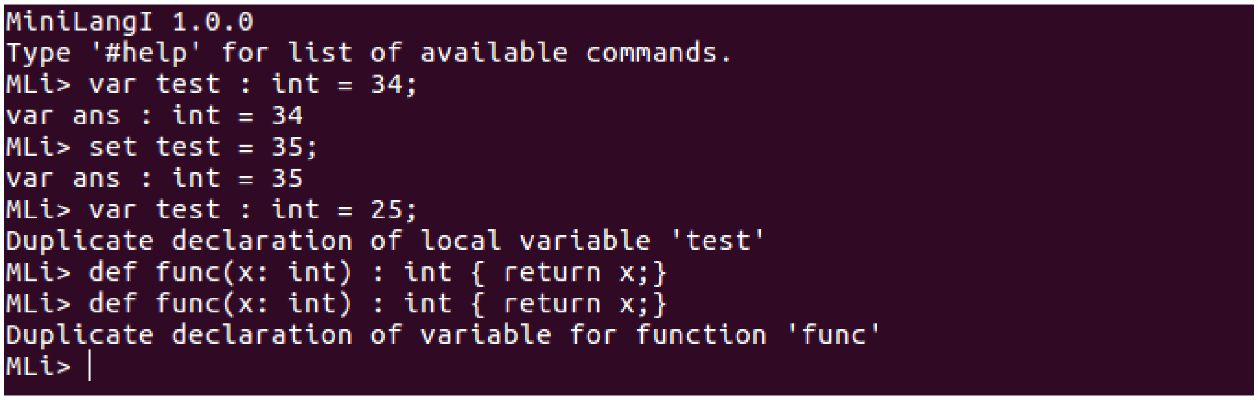
Figure 15: Error Handling Test 2. Parser.

Figures 16, 17 and 18 show some of the semantic errors that can be encountered when compiling the program. In Figure 16, the program is testing that the expressions of variable declaration are passed correctly according to the type of the variable declared. Also it is testing that the expressions are of the same type. The program is also testing the return type of the function is correct. Also it is testing that the parameters of the function are passed correctly. In Figure 17, the program is testing the duplication of both variables and functions. If there are multiple variables or function with the same names, they are not accepted. Figure 18 shows the checking of the function that every path will result in a return. In this case, the second if statement does not complete the path since after the if statement there are no more returns and there is no global return. Because of this the function is not accepted.



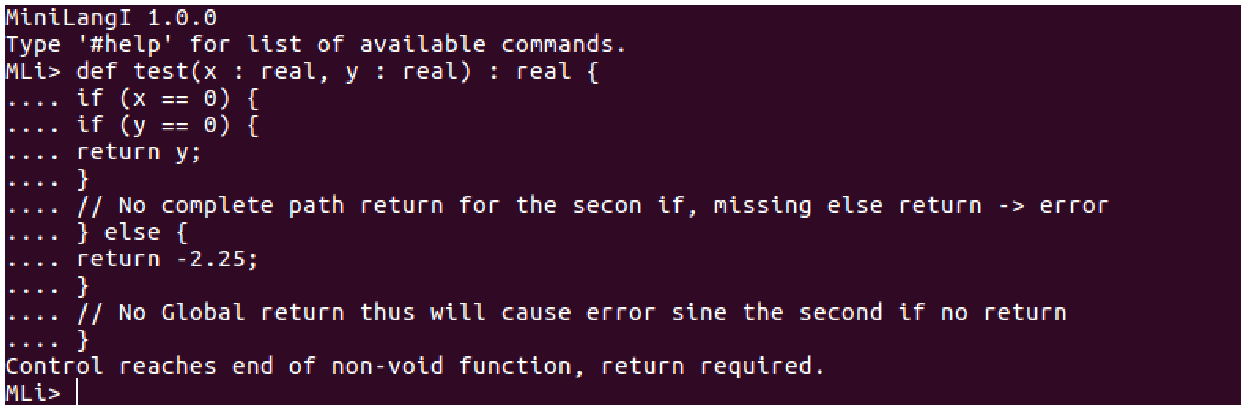
```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> var test : int = "test" + "test";
Incompatible types, expected 'int'
MLi> var test : int = "test" + 234;
Incompatible types of string and int
MLi> var test : int = 234;
var ans : int = 234
MLi> set test = true;
Incompatible types, expected 'int'
MLi> def func(y : int) : bool { return y; }
Returning 'int' when expecting 'bool'
MLi> def func(y : int) : int { return y; }
MLi> func("test");
Parameter type mismatch, expecting 'int'
MLi> 
```

Figure 16: Error Handling Test 3. Semantic Analysis.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> var test : int = 34;
var ans : int = 34
MLi> set test = 35;
var ans : int = 35
MLi> var test : int = 25;
Duplicate declaration of local variable 'test'
MLi> def func(x: int) : int { return x;}
MLi> def func(x: int) : int { return x;}
Duplicate declaration of variable for function 'func'
MLi> 
```

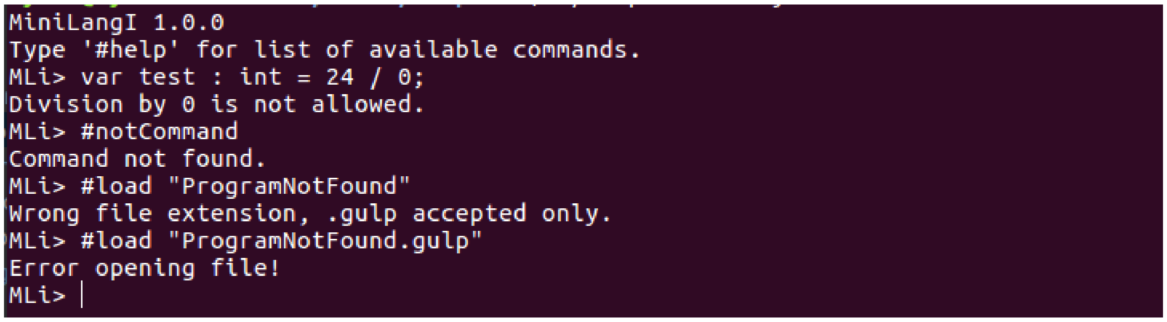
Figure 17: Error Handling Test 4. Semantic Analysis.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> def test(x : real, y : real) : real {
.... if (x == 0) {
.... if (y == 0) {
.... return y;
.... }
.... // No complete path return for the secon if, missing else return -> error
.... } else {
.... return -2.25;
.... }
.... // No Global return thus will cause error sine the second if no return
.... }
Control reaches end of non-void function, return required.
MLi> 
```

Figure 18: Error Handling Test 5. Semantic Analysis.

Figure 19 shows an error that can be thrown by the interpreter which is division by 0. Also it shows some commands that can fail when executing special commands with MiniLangI.



```
MiniLangI 1.0.0
Type '#help' for list of available commands.
MLi> var test : int = 24 / 0;
Division by 0 is not allowed.
MLi> #notCommand
Command not found.
MLi> #load "ProgramNotFound"
Wrong file extension, .gulp accepted only.
MLi> #load "ProgramNotFound.gulp"
Error opening file!
MLi>
```

Figure 19: Error Handling Test 6. Interpreter and REPL.