Compiler Theory and Practice

Assignment 2018 - 2019

Student Name: Valerija Holomjova

Course Number: CPS2000-SEM2-A-1819

Date: 23/05/19

Table of Contents

State of Completion

For this assignment, only Tasks 1 – 4 were completed.

Running the Code

The code was developed and run using the Clion IDE [1]. Alternatively, a bash script to run and compile the code was included in the directory. To run the code, give permission to both bash scripts using the 'chmod +x' command as shown in the screen shot below. Then run the './compile.sh' and './run' scripts in order. Any errors found in the code will be displayed in the terminal. Please note that the code will run code from the "Test.txt" and the XML structure of the code from Task 3 will be outputted to the "XMLOutput.txt" file.

```
Valerijas-MacBook-Pro:CompilersAssignment valerija$ chmod +x compile.sh
Valerijas-MacBook-Pro:CompilersAssignment valerija$ chmod +x run.sh
Valerijas-MacBook-Pro:CompilersAssignment valerija$ ./compile.sh
Valerijas-MacBook-Pro:CompilersAssignment valerija$ ./run.sh
Semantic Error - type mismatch for identifier: x
Semantic Error - type mismatch: 0
Semantic Error - identifier "error" was not declared.
Semantic Error - function "XGreaterThanY" was declared twice.
Valerijas-MacBook-Pro:CompilersAssignment valerija$
```

Figure – Compiling and Running the code in Linux Terminal

[1] https://www.jetbrains.com/clion/

Task 1 - Lexer

A lexer was implemented for the MiniLang language using the table-drive approach. The DFA diagram (Figure 1.1) below represents the MiniLang in EBNF in which a DFA transition table can be derived from (Figure 1.2) using the states from the DFA diagram and the classifier table (Figure 1.3). The lexer scans the input file character by character until a valid token is found and returns the token object to the parser through the 'get_next_token()' function. Please refer to Figure 1.4 which illustrates the possible tokens that can be retrieved from the lexer. Each token object stores a string value, float value and a token type. The transition table was implemented as a 2D array, and an 'enum' was used to hold the possible token types and classifiers. The lexer is able to detect single line comments and multi-line comments.

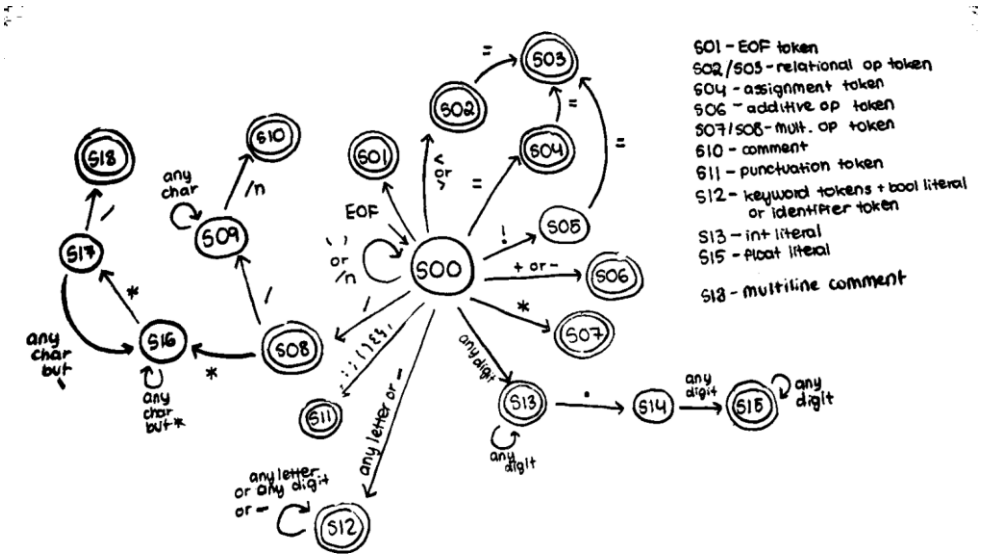

Figure 1.1 – The DFA diagram of the MiniLang EBNF

Classifiers	States																		
	S00	S01	S02	S03	S05	S06	S07	S08	S09	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19/SERR
CLASS_EOF	S01	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR
CLASS_BIG_SMALL	S02	SERR	S03	SERR	S03	SERR	SERR	SERR	S09	SERR	SERR	SERR	SERR	SERR	SERR	S16	S16	SERR	SERR
CLASS_EQUALS	S04	SERR	S03	SERR	SERR	SERR	SERR	SERR	S09	SERR	SERR	SERR	SERR	SERR	SERR	S16	S16	SERR	SERR
CLASS_EXCLAMATION	S05	SERR	SERR	SERR	SERR	SERR	SERR	SERR	S09	SERR	SERR	SERR	SERR	SERR	SERR	S16	S16	SERR	SERR
CLASS_ADD_SUB	S06	SERR	SERR	SERR	SERR	SERR	SERR	SERR	S09	SERR	SERR	SERR	SERR	SERR	SERR	S16	S16	SERR	SERR
CLASS_MULT	S07	SERR	SERR	SERR	SERR	SERR	SERR	S16	S09	SERR	SERR	SERR	SERR	SERR	SERR	S17	S16	SERR	SERR
CLASS_DIV	S08	SERR	SERR	SERR	SERR	SERR	SERR	S09	S09	SERR	SERR	SERR	SERR	SERR	SERR	S16	S18	SERR	SERR
CLASS_NEW_LINE	S00	SERR	SERR	SERR	SERR	SERR	SERR	SERR	S10	SERR	SERR	SERR	SERR	SERR	SERR	S16	S16	SERR	SERR
CLASS_PUNCT	S11	SERR	SERR	SERR	SERR	SERR	SERR	SERR	S09	SERR	SERR	SERR	SERR	SERR	SERR	S16	S16	SERR	SERR
CLASS_LETTER	S12	SERR	SERR	SERR	SERR	SERR	SERR	SERR	S09	SERR	SERR	S12	SERR	SERR	SERR	S16	S16	SERR	SERR
CLASS_DIGIT	S13	SERR	SERR	SERR	SERR	SERR	SERR	SERR	S09	SERR	SERR	S12	S13	S15	S15	S16	S16	SERR	SERR
CLASS_FULL_STOP	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	S09	SERR	SERR	SERR	S14	SERR	SERR	S16	S16	SERR	SERR
CLASS_UNDERSCORE	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	S09	SERR	SERR	S12	SERR	SERR	SERR	S16	S16	SERR	SERR
CLASS_OTHERS	SERR	SERR	SERR	SERR	SERR	SERR	SERR	SERR	S09	SERR	SERR	SERR	SERR	SERR	SERR	S16	S16	SERR	SERR

Figure 1.2 – The transition table used by the lexer

Classifier Table	
CLASS_EOF	EOF
CLASS_BIG_SMALL	< or >
CLASS_EQUALS	=
CLASS_EXCLAMATION	!
CLASS_ADD_SUB	+ or -
CLASS_MULT	*
CLASS_DIV	/
CLASS_NEW_LINE	\n
CLASS_PUNCT	: or ; or , or { or } or (or)
CLASS_LETTER	any letter
CLASS_DIGIT	any digit
CLASS_FULL_STOP	.
CLASS_UNDERSCORE	_
CLASS_OTHERS	other

Figure 1.3 – The classifier table used by the lexer

Token Table	
TOK_EOF	EOF
TOK_ERROR	ERROR
TOK_LEFT_PAR	(
TOK_RIGHT_PAR	)
TOK_LEFT_CURLY	{
TOK_RIGHT_CURLY	}
TOK_MULT_OP	* or /
TOK_ADD_OP	+ or -
TOK_REL_OP	< or > or == or != or <= or >=
TOK_COMMA	,
TOK_COLON	:
TOK_SEMICOLON	;
TOK_EQUAL	=

Token Table	
TOK_BOOL_LIT	boolean values
TOK_FLOAT_LIT	float values
TOK_INT_LIT	integer values
TOK_BOOL_TYPE	bool
TOK_INT_TYPE	int
TOK_FLOAT_TYPE	float
TOK_ID	identifer
TOK_VAR_DECLARE	var
TOK_PRINT	print
TOK_RETURN	return
TOK_FUN_DECLARE	fn
TOK_IF	if
TOK_ELSE	else
TOK_FOR	for
TOK_LOGIC	not/and/or
TOK_COMMENT	comment

Figure 1.4 – The token table used by the lexer

The lexer first stores the input program as a string variable ‘inputProgram’. It keeps track of the character position it is scanning using the ‘charIndex’ variable. The ‘get_next_token()’ function starts at the starting state of the DFA (S00). It uses a stack to keep track of the states that have been visited. In the scanning loop, the lexer scans each character from the ‘inputProgram’ and stores it in a string ‘lexeme’ until an error state is encountered. During this process, it skips any unnecessary new lines and white spaces provided that the lexeme is empty since new lines are needed to identify the end of a single line comment. To reach the next state, the classifier of the current character is obtained and used with the current state to obtain the next state that can be reached from the transition table. Since the transition function is complete, if a state cannot be reached for a given classifier, an error state will be returned.

Once an error state is reached, the function goes into the rollback loop where the lexer pops off states from the stack and characters from the lexeme until an accepting state or non-error state is reached. After this process is finished, a valid state should have been reached.

It then checks whether the current state can be a valid token by determining if it is a final state or not. If this condition is true, the lexer identifies and returns a token object using the lexeme and current state. Please note that in this function comment objects are not returned.

The 'check_next_token' is used to retrieve the next token without changing the character position of the lexer. The line number is also kept track of for error analysis.

Testing

The 'get_all_tokens()' function was implemented to check which tokens would be returned by the lexer for a given input code file. It gets the next token from a given file using the 'get_next_token()' function until an EOF token is reached. If an error is encountered the program is exited. Please note that the 'get_next_token()' does not return comment tokens but these are recognized. Below are some examples for the output displayed in the terminal by the lexer with the 'get_all_tokens()' function.

Input Code	Output from 'get_all_tokens()'
var = x; //There should be an error on line 3 var = x!;	TOK_VAR_DECLARE TOK_EQUAL TOK_ID TOK_SEMICOLON TOK_VAR_DECLARE TOK_EQUAL TOK_ID "Error - unrecognized token on line 3."

fn Square (x:float) : float {
return x[x;
}
	TOK_FUN_DECLARE
	TOK_ID
	TOK_LEFT_PAR
	TOK_ID
	TOK_COLON
	TOK_FLOAT_TYPE
	TOK_RIGHT_PAR
	TOK_COLON
	TOK_FLOAT_TYPE
	TOK_LEFT_CURLY
	TOK_RETURN
	TOK_ID
	"Error - unrecognized token on line 2."

fn Square (x:float) : float {
return x*x;
}
	TOK_FUN_DECLARE
	TOK_ID
	TOK_LEFT_PAR
	TOK_ID
	TOK_COLON
	TOK_FLOAT_TYPE
	TOK_RIGHT_PAR
	TOK_COLON
	TOK_FLOAT_TYPE
	TOK_LEFT_CURLY
	TOK_RETURN
	TOK_ID
	TOK_MULT_OP
	TOK_ID
	TOK_SEMICOLON
	TOK_RIGHT_CURLY
	TOK_EOF

Task 2 – Parser

In this task a top-down recursive descent parser was implemented to create an AST tree of the MiniLang program. The parser class stores a lexer object from which it retrieves the next valid token using the 'get_next_function'. The parser gets the next token until an EOF token is encountered. If an EOF token is not encountered, it expects a statement and calls the 'get_statement' function as a program is made up of 0 or more statements. These statements are appended into a vector of statements. In the end, the parser returns an ASTProgramNode which contains a vector of statement objects describing the structure of the code.

AST Nodes

There are three different type of ASTNode objects: the ASTStatementNode, the ASTProgramNode and the ASTExpressionNode. The ASTProgramNode as mentioned previously contains a list of statement nodes that will describe the entire program. For the other two nodes, children classes were made based on their expansion in the EBNF of MiniLang.

A statement can be expanded in 8 different ways (please refer to Figure 2.1), hence a class was created for each of the expansion methods. Since a function declaration statement could store a list of formal parameters, a formal parameters class was created to store the type and identifier of each formal parameter.

⟨*Statement*⟩ ::= ⟨*VariableDecl*⟩ ';'
| ⟨*Assignment*⟩ ';'
| ⟨*PrintStatement*⟩ ';'
| ⟨*IfStatement*⟩
| ⟨*ForStatement*⟩
| ⟨*ReturnStatement*⟩ ';'
| ⟨*FunctionDecl*⟩
| ⟨*Block*⟩

Figure 2.1 – Expansion of a Statement

An expression will ultimately result in a factor or a binary expression, hence a binary node class was created along with a class for each of the expansions of a factor (please refer to Figure 2.2). Since literals could be expanded in 3 ways (please refer to Figure 2.3), 3 different types of literal classes were also created – for an Integer literal, Float literal and Boolean literal.

⟨*Factor*⟩ ::= ⟨*Literal*⟩
| ⟨*Identifier*⟩
| ⟨*FunctionCall*⟩
| ⟨*SubExpression*⟩
| ⟨*Unary*⟩

Figure 2.2 – Expansion of a Factor

⟨*Literal*⟩ ::= ⟨*BooleanLiteral*⟩
| ⟨*IntegerLiteral*⟩
| ⟨*FloatLiteral*⟩

Figure 2.3 – Expansion of a Literal Expression

Please refer to Figure 2.4 for the inheritance diagram of all the AST objects that were created.

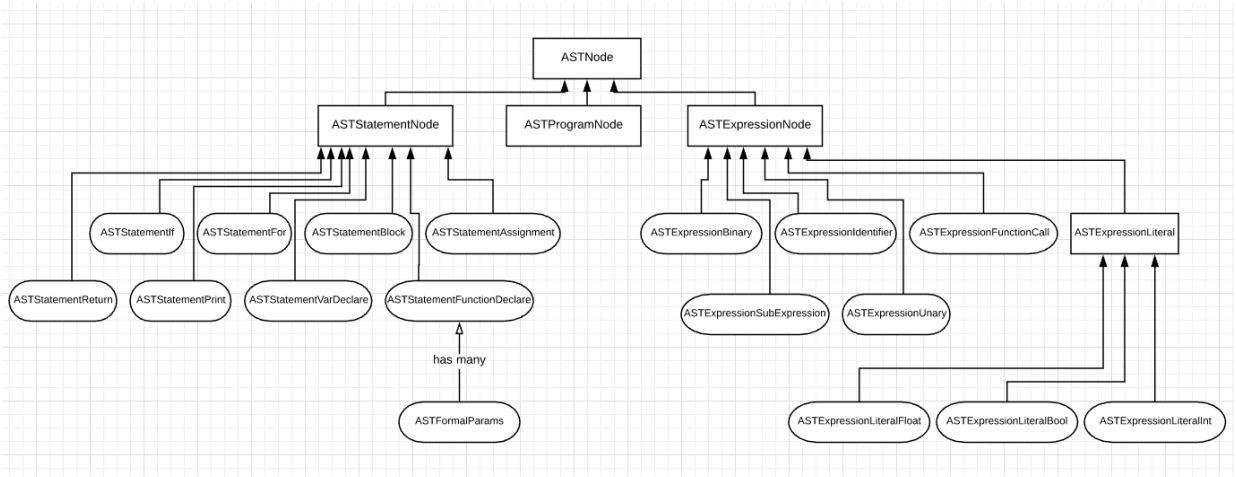

Figure 2.4 – Inheritance diagram of the AST nodes

Parsing

Similarly, to how the AST nodes were created, functions were created to return a polymorphic statement or expression node based on their expansions in EBNF. This is so that the tree could be built recursively, and objects could easily be instantiated from other expansions. I will proceed by describing an example of how a function declaration statement is parsed.

If the next token in the 'get_statement' function returns function declaration token (TOK_FUN_DECLARE), the 'parse_function_declare' function is called to return an AST function declaration node. In the 'parse_function_declare' function, the structure of the '<FunctionDecl>' in EBNF (Figure 2.5) is followed and each component is stored in the object. For instance, the function checks whether the next token is an identifier. If this condition is true, an identifier variable in the AST function declaration node is set to the tokens value. If this condition is false, an error is thrown since it is specified by the EBNF that the token following a function declaration 'fn' should be an identifier.

$$\langle FunctionDecl \rangle \quad ::= \text{`}\texttt{fn}\text{'} \ \langle Identifier \rangle \ \text{`(' [} \ \langle FormalParams \rangle \ \text{] `)' `:'} \ \langle Type \rangle \ \langle Block \rangle$$

Figure 2.5 – Expansion of Function Declaration

Next, the function checks that a left parenthesis token follows the identifier. If this condition is true, the function moves on and tries to retrieve a list of formal parameters. It does this by calling the 'get_formal_params()' function to retrieve a formal parameter object and store it in a vector list of formal parameters until a right bracket token is read by the lexer.

The 'get_formal_params()' function will create an AST formal param object based on it's expansion as well (Figure 2.6). It does this by checking that the next token is an identifier token, followed by a colon token and followed by a literal type token. If these conditions are not met an error is outputted to the console. If these conditions are met, a formal param object with an identifier and literal type is sent back to the function declaration object to be added to its vector list of formal parameters.

$$\langle FormalParams \rangle \quad ::= \langle FormalParam \rangle \ \{ \text{`,'} \ \langle FormalParam \rangle \ \}$$

Figure 2.6 – Expansion of Formal Parameters

The 'parse_function_declare()' then resumes by checking that the next token is a colon token, followed by a literal type token and then a block token. It stores the type as a string and stores the block statement as a pointer to an AST block statement object. The block statement object is obtained by calling the 'parse_block()' function which in turn holds a statement object which is obtained through the 'get_statement()' function again and this time a different statement is parsed.

In the end an entire tree is obtained from the program node object where each statement would have pointers to other statement nodes or expressions based on their EBNF expansions. The testing of the parser can be seen in the next task description (Task 3) where the XML format of the parser is outputted.

Testing

Input Code	Output from Parser	Comment

fn LoopXTimes (x : int) : int {
var x : y = 0; //This is an error.
for(var i : int = 0; i < x; i = i + 1){
//This is a test comment.
}
}
	"Error on line 2 - Expected variable type after colon in variable declaration."	In the variable declaration of x, an error is through since y is not a variable type.

fn Test (;) : int {
//This is an empty function.
}
	"Error on line 1 - Expected identifier in formal params."	In the formal parameters of a function there should be 0 or more identifiers, since ';' is not an identifier an error is thrown.

fn ReturnX (x : int) : int {
//This function should return X.
return bool;
}
	"Error on line 3 - Unrecognizable factor."	A return type expects an expression after a 'return' keyword, hence since 'bool' is not a factor but a literal type, it does not identify as an expression.

Task 3 – AST XML Generation

The 'printInfoVisitor' class is a subclass of the visitor class and contains visit functions for each AST node type so that the AST tree could be traversed and have its content could be printed into an XML file describing the structure of the code. The following link was used to help grasp a better understanding of how a visitor class is implemented[2]. The AST is traversed starting from the AST program node which was obtained from the parser. In each visit function, the variables of the node are printed and if there is a pointer to another AST node, it's 'accept()' function is called so that it's respective visit function could be called and its content could be printed as well until all the nodes contents are outputted in a file.

A variable is used to keep track of the indentation which is incremented after the starting tag of each statement or expression and then decremented when the contents are fully displayed (before the closing tag). Each AST node was given a 'accept()' function which allows the object

[2] https://cpppatterns.com/patterns/visitor.html

to accept a visitor and call the appropriate visit function on the visitor. After the entire AST tree is traversed, the 'XMLOutput.txt' file contains an XML structure of the code.

Testing

Input Code	XML Output
var ans : bool = true;	<?xml version="1.0" encoding="UTF-8"?> <ProgramNode> <VariableDeclaration> <Variable Type="TOK_BOOL_TYPE">ans</Variable> <Expression> <BoolLiteral>true</BoolLiteral> </Expression> </VariableDeclaration> </ProgramNode>

/* This is a test comment */
fn XGreaterThanY (x : int , y : int) : bool {
var ans : bool = true;
if(y > x){ans = false;}
return ans;
}
	<?xml version="1.0" encoding="UTF-8"?>
	<ProgramNode>
	<FunctionDeclaration Type="TOK_BOOL_TYPE">
	<Identifier>XGreaterThanY</Identifier>
	<FormalParameters>
	<FormalParameter Type="TOK_INT_TYPE">
	<Identifier>x</Identifier>
	</FormalParameter>
	<FormalParameter Type="TOK_INT_TYPE">
	<Identifier>y</Identifier>
	</FormalParameter>
	</FormalParameters>
	<BlockStatement>
	<VariableDeclaration>
	<Variable Type="TOK_BOOL_TYPE">ans</Variable>
	<Expression>
	<BoolLiteral>true</BoolLiteral>
	</Expression>
	</VariableDeclaration>
	<IfStatement>
	<Expression>
	<BinaryExpression Op=">">
	<Identifier>y</Identifier>
	<Identifier>x</Identifier>
	</BinaryExpression>
	</Expression>
	<BlockStatement>
	<Assignment>
	<Identifier>ans</Identifier>
	<Expression>
	<BoolLiteral>false</BoolLiteral>
	</Expression>
	</Assignment>
	</BlockStatement>
	</IfStatement>
	<ReturnStatement>
	<Expression>
	<Identifier>ans</Identifier>
	</Expression>
	</ReturnStatement>
	</BlockStatement>
	</FunctionDeclaration>
	</ProgramNode>

Task 4 – Semantic Analysis

The 'semanticAnalysisVistor' class is a subclass of the visitor class and was implemented to traverse the tree similarly as carried out in Task 3 and check the code for semantic errors.

Symbol Table

A symbol table class was used to store the function and variable declarations for each scope. Each symbol is a struct and holds an identifier, a declaration type ("function" or "variable") and

a literal type. Each symbol also has a vector of strings to store parameter types for a function declaration. The symbol table class was implemented in the form of a hash table as this is one of the most common methods where each key is an identifier for a symbol and the value is the information on the symbol. A hashing function was implemented to insert and find symbols through their identifier. The current code was made so that functions and variable declarations cannot share the same name. A 'displayInfo' function was also created so that values of a symbol could be printed out for debugging purposes.

Semantic Analysis

The semantic analyser will notify the user of double declarations, type mismatches, variables or functions that haven't been declared but called, parameter differences and scope resolution. Please note that it has only allowed the declaration of functions in the global scope. A list of symbol tables was used to keep track of the declarations that were made and could be used in to the current scope. Each time a block statement is entered a new symbol table is added to the list and each time a block statement is left, a symbol table is deleted from the top of list. Hence, the first element of the list is always the global scope. Since the AST tree is traversed in a depth first manner, implementing a list of tables in a stack method was sufficient to keep track of the declarations. The list was not implemented entirely as a stack structure so that it's elements could be easily traversed. Below is a description of the functions used to manipulate the list of symbol tables:

- The function 'addTable()' was implemented to add a new symbol table to the end of the list.
- The function 'removeLastTable()' removes a symbol form the end of the list and frees it's memory.
- The function 'getLastTable()' returns a pointer to the last symbol table of the list.
- The function 'checkInTables()' checks for declaration with a given identifier from the end of the list till the very start of the list.

The variable 'currentLiteralType' was used to hold the type of a function declaration so that the return statement expression would match. It also holds the type of an assignment. It is updated recursively in the code so when a function declaration is being check its value is updated to the functions type until the function declaration block statement has been checked (where the return expressions would be), then it returns to the previous value it was set to. The same happens for an assignment statements, when a new assignment statement is being checked, the variable is temporarily updated to the assignment type until the assignment expression is checked for matching types. If the assignment expression or return expression have unmatching literal types semantic errors are displayed to the user.

Similarly, the 'enableBinaryChecking' variable was used as a flag so that binary expressions in 'if', 'for' and 'print' statements can be checked so that their types match. When this flag is true, the 'binaryLiteralType' will hold the literal type of the first expression and ensure that the rest of the expressions match. As soon as the expression is checked, the flag variable is set to false and the literal type is reset. This was implemented as it was clashing with the 'currentLiteralType' variable.

Testing

Below are examples of how the code was tested. The semantic analyser will report multiple errors found in the code. For type mismatches – the value which was mismatched will be produced to the user in the terminal.

Input Code	Output from Parser	Comment

fn LoopXTimes (x : int) : int {
  for(var i : int = 0; i < x; i = i + 1){
  x = x+i;
  }
  return i; //This is an error.
}
	Semantic Error - identifier "i" was not declared.	Since the variable 'i' was not declared in a parent scope of the function declaration but declared in the for loop statement, it could not be returned.

/* This is a test comment */
fn XGreaterThanY (x : int , y : float) : bool {
var ans : bool = true;
if(y > x){ //There is an error here - y is a float so cant be compared to x.
ans = false;
var error : int = false; //This is an error - wrong literal type.
}
return error; //This is an error - error was declared out of scope.
}
 	
fn XGreaterThanY (x : int, y : float) : int {}
//This is an error - function already declared.
 	
fn LoopXTimes (x : int) : int {
  for(var i : int = 0; i < x; i = i + 1){
  x = x+i;
  }
  return x; //This is an error.
}
	"Semantic Error - type mismatch for identifier: x
	
	Semantic Error - type mismatch: 0
	
	Semantic Error - identifier "error" was not declared.
	
	Semantic Error - function "XGreaterThanY" was declared twice."
		The first error was found in the if statement where the comparison between 'y' and 'x' could not occur because they are different types.
		
		The second error occurred in the declaration of 'error' since a Boolean value was assigned instead of an int.
		
		The third error occurred because the 'error' value was used outside of the scope.
		
		The last error occurred because the function was declared twice.