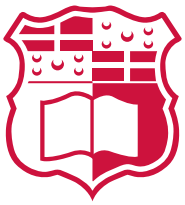

L-Università ta' Malta

CPS2000 - Assignment
MiniLang Compiler

Daniel Magro

Bachelor of Science in
Information Technology (Honours)
(Artificial Intelligence)

May 2018

Contents

Task 1

Table-Driven Lexer

The Lexer takes the input program which is to be compiled, and converts it into a sequence of **Tokens**, which are then handled by the **Parser** in the next task.

The micro-syntax of MiniLang was first encoded into a Deterministic Finite State Automata (DFSA). Having a deterministic encoding is very advantageous as this provides no ambiguity whatsoever when deciding how to deal with a particular character when reading input. Furthermore, when stopping on any state, given it is a final state and no more of the input can be accepted, that state will only return one possible token (with the exception of state 15, which can return both identifiers and keywords, this is explained further below).

The Table-Driven Lexer was first drawn out as a DFSA, as can be seen in Figure 1.1.
The c++ program will simply start at S00 (state 0), and depending on the input, character by character, will follow the transitions from one state to the next.

The DFSA shown in Figure 1.1 was then converted into a transition table, which can be seen in Table 1.1. The use of this table is very simple, and can always be done in constant time. For example, if the program is currently on the start state, S00, then given an exclamation mark, !, then it will simply look at row 'S00', under column '!' (column 2) and read "2", which means go to state 2. If an equals, =, is then read, then from row 'S02', under col-

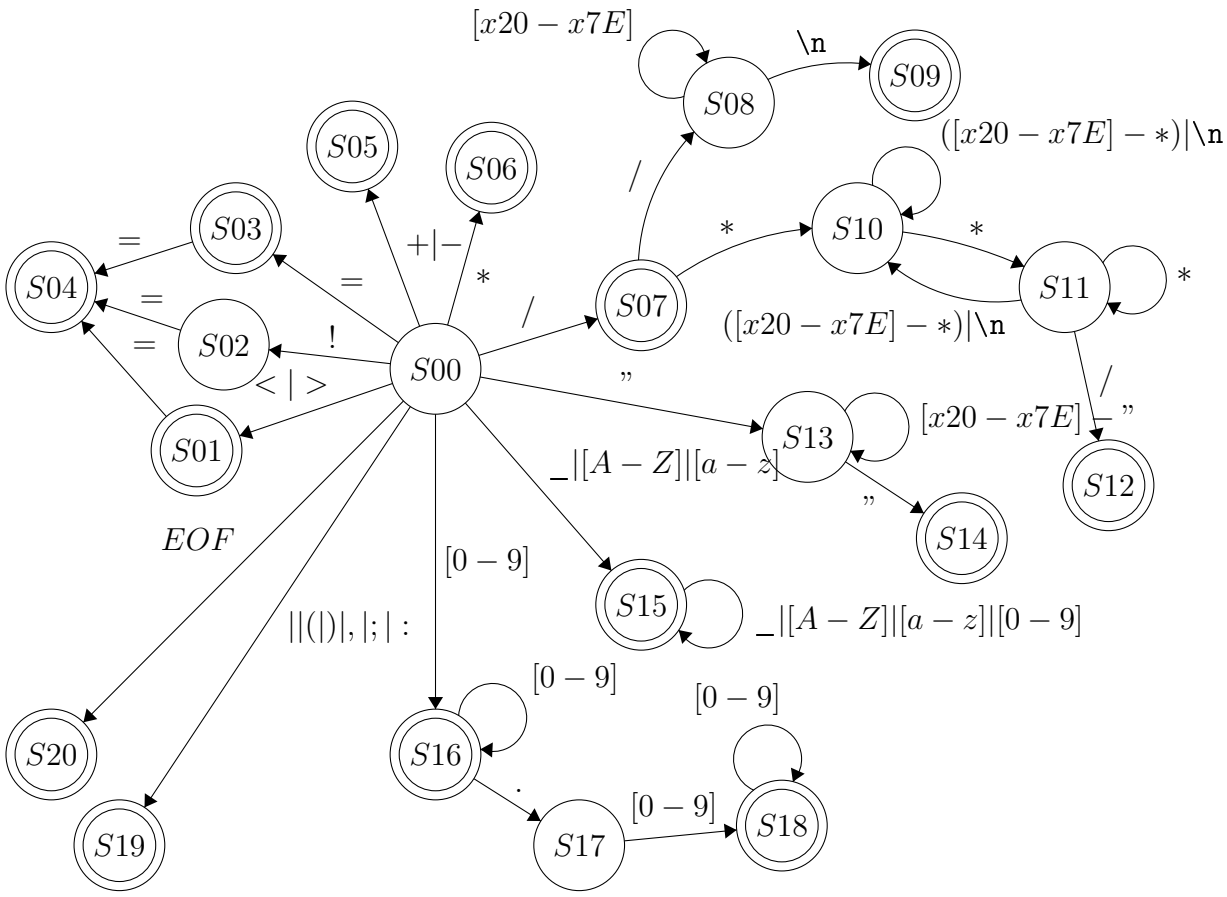

Figure 1.1: DFSA Lexer

inputs →	**0**	**1**	**2**	**3**	**4**	**5**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**6**
states ↓	<|>	=	!	+|-	*	/	\n	"	_|[A-Z]|[a-z]	[0-9]	.	{|}|(|)|,|;|:	EOF	[\x20-\x7E]
S00	1	3	2	2	6	7	0	13	15	16	-1	19	20	-1
S01	-1	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S02	-1	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S03	-1	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S04	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S05	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S06	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S07	-1	-1	-1	-1	10	8	-1	-1	-1	-1	-1	-1	-1	-1
S08	8	8	8	8	8	8	9	8	8	8	8	8	-1	8
S09	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S10	10	10	10	10	11	10	10	10	10	10	10	10	-1	10
S11	10	10	10	10	11	12	10	10	10	10	10	10	-1	10
S12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S13	13	13	13	13	13	13	13	14	13	13	13	13	-1	13
S14	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S15	-1	-1	-1	-1	-1	-1	-1	-1	15	15	-1	-1	-1	-1
S16	-1	-1	-1	-1	-1	-1	-1	-1	-1	16	17	-1	-1	-1
S17	-1	-1	-1	-1	-1	-1	-1	-1	-1	18	-1	-1	-1	-1
S18	-1	-1	-1	-1	-1	-1	-1	-1	-1	18	-1	-1	-1	-1
S19	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S20	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Table 1.1: Transition Table

umn '=' (column 1) it will read "4", which means go to state 4. Any input read after state 4 will always return "-1", which indicates that there are no outgoing transitions, this is fine however since S04 is a final state, and will return the RelationalOperator Token.

As mentioned earlier, each final state returns one particular token, as shown below in Table 1.2.

The implementation in c++ was done as follows:

Tokens.h was defined in c++. This class defines the *TokenIdentifier* enum, which contains all the types of tokens which are recognised by the compiler.

State	Token	Values
S00	TOK_ERROR	N/A
S01	TOK_RelationalOp	<|>
S02	TOK_ERROR	N/A
S03	TOK_Equals	=
S04	TOK_RelationalOp	<=|>=|!=|==
S05	TOK_AdditiveOp	+|-
S06	TOK_MultiplicativeOp	*
S07	TOK_MultiplicativeOp	/
S08	TOK_ERROR	N/A
S09	TOK_Comment	// Inline Comment
S10	TOK_ERROR	N/A
S11	TOK_ERROR	N/A
S12	TOK_Comment	/* Block Comment */
S13	TOK_ERROR	N/A
S14	TOK_StringLiteral	"Text"
S15	TOK_Identifier	Name of a variable starting with _||[A-Z]||[a-z] and continuing with _||[A-Z]||[a-z]||[0-9]
S15	TOK_Keyword	Same as TOK_Identifier, however must match a keyword (eg: def, if, else, while etc)
S16	TOK_IntegerLiteral	[0-9]+
S17	TOK_ERROR	N/A
S18	TOK_RealLiteral	[0-9]+ . [0-9]+
S19	TOK_Punctuation	{|}|(|)|,|;|:
S20	TOK_EOF	EOF

Table 1.2: State-Token Table

Secondly, it defines a *Token* struct, which defines that a Token object must contain a TokenIdentifier field (from the values defined in the enum) and a string representing the lexeme.

States.h first defines the number of states in the dfsa, which is **21**, and the size of the alphabet, or rather how many classifications of input characters there are, which is **14**.
Then, a 2D array of integers (where each integer refers to a state number), was declared and named *transitionFunction*, with the 'number of states' rows and the 'number of input classifications' columns. Each element was then initialised according to the State Transition Table shown above in Table 1.1.
A struct for **State** was also created, which represents the stateID, a boolean representing whether a state is final or not, and a TokenIdentifier showing what Token is returned by each state.
A static array of type State was declared, which will store each state in the DFSA.
A function *initialiseDFSAStates* is also declared, which initialises each state's ID, finalState flag and default token to be returned.

Lexer.h first defines a set of strings, keywords which the lexer should recognise when reading through the input program.
Lexer.h also defines a group of variables which it requires to function, these are:

- A vector of characters representing the entire input program file, *program*
- An integer storing up till what character of the input file the lexer has tokenised, *inputProgramIndex*
- An integer storing what line the program is currently reading, *lineNumber*, this is used for error reporting to the user rather than functionality of the program
- A set of strings, where each string is a keyword recognised by the compiler, *keywords*

A number of functions were also declared, these being:

- readProgram, which, given the path of the input MiniLang code to be compiled, reads the entire input file character by character, and stores it inside the vector of characters *program*, until the end is reached.

When the end of the file is reached, an EOF symbol is pushed to the end of the vector.

- isKeyword, which receives a string of characters as an argument, and returns whether it is a keyword or not. This method simply checks whether the passed string is a member of the *keywords* set or not.
- nextState, which receives the current state of the DFSA and the next character in the input program as parameters, and returns an int representing what state the DFSA should go to next. This method simply accesses the transitionFunction 2D array defined in States.h, and returns the next state the lexer should go to.
- colAlphabet, which given a character, returns what column of the transition table it corresponds to. This function basically enumerates a given character input so that it can be used as an index for the 2D array transitionFunction. Each character is assigned its number in the same way the heading of Table 1.1 is labelled (eg. '!' $\Rightarrow$ 2, ';' $\Rightarrow$ 12).
- **getNextTok** is the main component of the lexer. It first declares the currentState and sets it to 0, the lexeme as an empty string, a stack of visited states, with state 0 at the bottom.
  Then, the following is done either until for the current character on the current state there is no valid outgoing transition, or the end of the file has been reached.
  The currentChar is set to the character in the *program* vector of chars at postion *inputProgramIndex*, and *inputProgramIndex* is incremented.
  It is then checked whether the current character is a white space or a tab, in which case it is ignored and the next iteration is performed (unless the lexer is on state 13, meaning a string literal was being read, in which case whitespaces are kept).
  Then it is checked whether the character is a new-line, in which case the *lineNumber* is incremented, and the next iteration is performed, unless
  - A string literal was being read, in which case it is desirable to retain new lines
  - An inline comment was being read, in which case it is necessary to retain new lines to determine when a comment has terminated

  If this point of execution has been reached, the *currentChar* is added to the end of *lexeme*
  The *currentState* is now set to the return of the nextState function

given the currentState and the currentChar.
Then it is checked whether the lexer is in state 15, and whether the current lexeme is a keyword, in which case the respective Token is returned.
It is then checked whether the currentState is a final state, in which case the visitedStates stack is emptied, since in case of a rollback, rollback only needs to be done to the last final state. (This is a small optimisation).
The currentState is added to the top of the visiedStates stack.

When the loop exits, another loop checks whether the currentState is a final state and that the stack is not empty.
While the above condition is true,

- the non-final state at the top of the stack is popped
- the last character in the lexeme is removed
- the inputProgramIndex is decremented
- the currentState is set to the state at the top of the visitedStates stack

Finally, a condition checks if after the last while loop, the current state is final, if so, a Token is returned with the TokenIdentifier associated with the curentState (as specified in States.h - Table 1.2) and the lexeme.
If the currentState is not final, then a lexical error is reported at *lineNumber*, and the program exits.

- **getNextToken** simply calls *getNextTok*, and checks whether the Token is a Comment, if so, it calls getNextTok until a Token which is not a comment is returned, and it returns that.
- **peekNextToken** was implemented since the Parser is an LL(1) parser, and thus lookahead is required. This function accomplishes that by saving the current states of *inputProgramIndex* and *lineNumber*, calling getNextToken, and restoring the values of inputProgramIndex and lineNumber to what they were, so the next time getNextToken is called again, it will return the same Token since it will start from just before that token is produced.

  For example, given the input "var x : real = 2.4;", the getNextTok method does as follows:

- the first character, 'v' is consumed, added to the *lexeme* string, and the inputProgramIndex is incremented. The new currentState is calculated using the nextState function with parameters state 0 + char 'v', resulting in state 15. Since the current state is 15, it is checked whether the lexeme is a keyword, however "v" is not a keyword, so the next iteration is executed.
- 'a' is consumed, and similarly to the 'v', it is added to the lexeme and the inputProgramIndex is incremented. The nextState is calculated, and the answer is again 15. The lexeme is checked to be a keyword however "va" is not a keyword.
- 'r' is consumed and added to the lexeme. The nextState function again returns 15. The lexeme this time is "var", which is a keyword, thus the Token: TOK_Keyword, "var" is returned.
- 'x' is consumed and added to the new lexeme. The nextState function returns 15. The lexeme "x" is not a keyword.
- ':' is consumed and added to the lexeme. The nextState function now returns -1, as there is no transition for ':' from state 15. Therefore, the loop exits. The next loop then pops the -1 state from the top of the stack, removes the : from the end of the lexeme, decrements the inputProgramIndex, and sets the currentState to the top of the visited states stack, in this case 15. The next iteration does not execute, as the condition is false as State 15 is final. Thus a token is returned with TokenIdentifier as the default for State 15 (i.e. TOK_Identifier) and lexeme "x".
- ':' is consumed again since the index was decremented. This time the nextState function returns state 19, as state 0 does have a transition for ':'. In the next iteration, 'r' will be consumed, however state 19 has no transition for 'r', so it will be removed, and the Token TOK_Punctuation will be returned, with lexeme ":".
- the process for "real", "=" and "2.4" is very similar to what is explained above.

Task 2

Hand-Crafted Recursive Descent Parser

The Parser is able to construct an Abstract Syntax Tree, while retrieving Tokens from the Lexer, through the getNextToken() function.

First, **ASTNode.h** was created, which serves as a parent class for all AST Nodes that will be created during the parser's runtime. It is very useful to have this superclass as all the defied AST Nodes will inherit from this class, and thus when an instance of, say, ASTReturn or ASTBinaryExpr is created, it will be of type ASTNode.

The same concept was applied to **ASTExpressionNode.h** and **ASTStatementNode.h**. Both classes inherit from ASTNode.h, and every expression node inherits from ASTExpressionNode and every statement from ASTStatementNode. This was very useful, for example, when storing a vector of *ASTStatementNode*s in ASTBlock, as there might be ASTReturn, ASTWhile, ASTIf statements etc, but can be stored as StatementNodes thanks to this inheritance.

An ASTNode class was defined for almost each rule in the EBNF. Each class defines the fields necessary for each Node, a constructor and a destructor, and an *accept* method, which is made use of by the visitors (of the Visitor Design Pattern).
For example, ASTBinaryExpr has a pointer to two *ASTExpressionNode*s, and to an *ASTOperator* node.

Parser.h defines all the variables and functions that are needed for the Parser to work correctly.
The only two variables needed are *currentToken*, which stores the current token, and *ast*, which is a vector of ASTNodes.
The functions used by the Parser are:

- A constructor for the parser which initialises *ast* to an empty vector, and a destructor which deletes every element of the vector and the vector itself
- A *parse* method, which sets currentToken to the next token from the lexer (using getNextToken). Then calls parseStatement(), and pushes the returned ASTStatementNode to the end of *ast*. All of this repeats until the next token is the EOF token (TOK_EOF).
- The **parseStatement** method mentioned earlier, as the name implies, returns parses the current statement and returns it. The method checks that the identifier of the current token is TOK_Keyword, and whether the lexeme is var, set, print, if, while, return, def or "{" (in which case the identifier will be TOK_Punctuation). Depending on which keyword is matched, the relevant parse method is called, as explained in the next bullet.
- A parseXXXStatement function exists for every statement recognised by the compiler (i.e. that is in the EBNF). Each method is called when the respective token is met in parseStatement(), so for example if the current token is a "def" keyword, then parseFunctionDeclarationStatement() is called.
  So for example, parseAssignmentStatement() does the following:
  - Declares a pointer to an ASTIdentifier node and another pointer to an ASTExpressionNode.
  - Then updates the currentToken with getNextToken, and sets identifier to the return of *parseIdentifier()*.
  - The currentToken is again updated, and it is checked whether it is a TOK_EQUALS. If not, an error is displayed and the compiler exits.
  - The currentToken is again updated, and the expression is set to the return of *parseExpression*.

  - The currentToken is updated, and it is checked whether the current token is a semicolon ';'. If not, an error is shown and the compiler exits.
  - The method finally returns a new ASTAssignment node, with the pointers to the identifier and the expression as arguments to the constructor.

- A parseXXX function for every expression recognised by the compiler.
  - parseExpression() parses one an Expression, which can be a SimpleExpression, or multiple simple expressions joined together by relational operators.
  - parseSimpleExpression() parses one Simple Expression, which can be made up of a term, or multiple terms joined together by additive operators.
  - parseTerm() parses one Term, which can be a factor, or multiple factors joined together by multiplicative operators.
  - parseFactor() parses one factor, be it a literal, an identifier, functioncall, sub expression or a unary expression.

The general concept of this Parser is that during the execution of a particular parseStatement(), multiple other parse methods are executed, and for each of those parse method an ASTNode is created. Once these ASTNodes are created, the ASTNode of the parseStatement can be initialised using its constructor and the pointers to the sub ASTNodes.
For example the parseIfStatement() during its execution calls parseExpression for the boolean expression inside the if, and calls parseBlock() for the if (and possibly else) blocks.

Task 3

Generating XML of AST

The ASTtoXMLVisitor 'visits' every ASTnode inside the Abstract Syntax Tree, and outputs an XML representation of the contents of every node that is visited.

Visitor.h defines a list of virtual (abstract) visit functions that must be defined in every Visitor class.

ASTtoXMLVisitor.h inherits from **Visitor.h**, and thus it must have a definition for every virtual function specified in Visitor.h.
Apart from all the visitors, a string *indentation* is declared, which will store how many ''s there are behind the current XML line.
The contents of *indentation* are modified by the *indent* and *outdent* methods. indent() appends a '' to the end of the string, and outdent() removes a from the string.
The constructor simply initialises the *indentation* to an empty string.
A visitAST method was also declared, which given a pointer to the AST (vector of pointers to ASTNodes) outputs the XML.

ASTtoXMLVisitor.cpp contains the definitions for all the methods specified in the .h file.
In brief, each method does the following:

- Indents according to the current indentation level in the *indentation* string and outputs the opening XML Tag for the current ASTNode being visited (so for ASTBinaryExpr, the visitor will output <Binary-

Expression>).

- Calls the indent() function, which increases the indent for the next XML Tags to be printed.
- Now the visitor will print the XML tags for the attributes of the current node. This is achieved by calling the visitor of all the subASTNodes of the current ASTNode.
  In the case of terminal ASTNodes, such as ASTLiteral Nodes, and ASTOperator or ASTType Nodes, the visitor will simply print the value in between two tags, at the current indentation level. For example, for Operator *, the visitor will print <Operator>'*'</Operator>
- When the attributes of a particular node have been printed, the outdent() function is called, which reduces the indentation.
- Finally, the closing XML Tag is printed at the current indentation level, so again for ASTBinaryExpr, the visitor will output </BinaryExpression>.

Thus, the general XML Representation for all ASTNodes is as follows:

```
<ASTNode>
    <subASTNode>
    <subASTNode>
    ⋮
</ASTNode>
```

Each of the subASTNodes may once again be expanded with a similar hierarchy, for example, the XML representation of the expression 5*2+4 would look as follows:

```
<BinaryExpression>
    <Operator>'+'</Operator>
    <BinaryExpression>
        <Operator>'*'</Operator>
        <IntegerLiteral>5</IntegerLiteral>
        <IntegerLiteral>2</IntegerLiteral>
    </BinaryExpression>
    <IntegerLiteral>4</IntegerLiteral>
</BinaryExpression>
```

Task 4

Semantic Analysis Pass

The Semantic Analysis Pass traverses the AST using the visitor pattern, similarly to the ASTtoXML visitor, however performs type checking on the program. This pass detects semantic errors, which may include assigning variables values of different types, trying to carry out a calculation on two variables of different types, or calling a function which does not return the correct type or which has not yet been declared

The first step was the creation of **VariableProperties.h**, which is a class that stores all of a variable's information that is required during both the Semantic Analysis Pass and the Interpreter Execution Pass. In this section, only the attributes relating to the Semantic Analysis Pass will be discussed. The Variable Properties stores the following information about every variable:

- valueType, which stores whether a variable, or the return type of a function, is of type Bool, Int, Real or String.
- variableType, which stores whether a variable is indeed a variable or a function.
- parameterTypes, which is a vector of types, that defines the method signature of a function.

The **Symbol Table** was then defined. The Symbol Table's role is to store what variable identifiers are in the current scope, and the VariableProperties for each of those identifiers.
The SymbolTable class contains a multimap of strings to VariableProperties.

The multimap data structure was chosen over a map since a certain scope might contain more than one variable with the same identifier (for example a variable x, and 5 different functions named x with different method signatures (i.e. different parameter type sequences)). One multimap is used for every scope of the program, this makes it possible, for one, to differentiate between global variables and variables in the current scope.
Apart from the multimap, the SymbolTable class defines a number of functions which make it much easier for the Semantic Analysis visitors to interact with the Symbol Table. These are:

- add, which given a variable's identifier and variableProperties will add the pair to the multimap representing the scope's symbol table.
- isVariableInSymbolTable, which given the identifier of a variable returns whether that variable is in the symbol table or not.
- isFunctionInSymbolTable, which given the identifier and variableProperties of a function returns whether that function is in the symbol table or not.
- typeOfVariable, which given a variable's identifier, returns its data type.
- areFunctionIdandParamsInSymbolTable, which given a function's identifier, and parameter types signature, returns whether or not the function is in the symbol table.
- typeOfFunction, which given a function's identifier, and parameter types signature, returns that function's return data type.

These functions are all implemented using an iterator which goes through every element in the map. The functions like typeOfVariable() first assert that the variable is in the symbol table before looking for it, via isVariableInSymbolTable().

The SemanticAnalysis class is the class that defines the process for performing semantic analysis on the Abstract Syntax Tree. To do this, a number of global (within the class) variables were declared, these being:

- A vector of pointers to Symbol Tables, *symbolTableStack*. A symbolTable is declared for each stack, this makes it possible for a certain scope to 'know' about variables or functions which are defined in an outer scope.

- A variable of type 'Type', typeOfCurrentVariable. This variable stores the type of a particular node or variable which has just been visited, so that it can be accessed from the visitor that called that particular node's visitor in the first place.
- A multimap of parameter identifiers to their type, which is used for passing function parameters from one visitor to another.
- A boolean value, hasReturnStatement, which is used to determine whether the last visited block has a return statement or not.
- An integer value, currentSymbolTableScopeIndex, which is used as an index when looking for a variable, always starting from the inner-most scope, and moving outwards towards the global scope.

A few functions were also declared, these being the constructor for the SemanticAnalysis class, a visitor for the whole AST (visitAST - which takes a pointer to the AST as a parameter), and visitors for all the ASTNodes, which were declared virtual in Visitor.h

The constructor first pushes a new symbol table to the *symbolTableStack*, one representing the global scope.
Then the visitAST method loops through every statement node inside the AST, and calls its visitor.
Once every ASTStatementNode has been visited, the destructor can be called and the global symbol table is popped, and the user is informed that no errors were found during semantic analysis.

All the other visitors make sure that each statement in the AST is not violating any of the rules that semantic analysis should catch or detect. Some of these are:

- If a variable is being declared in any scope, another variable with the same identifier cannot be present in the same scope.
- If a function is being declared in any scope, another function with the same identifier AND same method signature (i.e. same ordered parameter data types) cannot be present in the same scope.
  (Please note that having, for example, int x and string x will result in an error in my implementation - this was done intentionally as most C-like languages will not allow this either).
- A variable of type X should be assigned values of type X, be it as a

direct assignment or assigning to the return of a function.

- The condition inside If and While statements should be of type Bool.
- Every function should have a return statement matching the return type of the function.
- Binary Expressions should be performed on variables of the same data type.
- Certain Binary Operators can only be performed on certain Data Types (For example *string + string* is invalid).
- Certain Unary Operators can only be performed on certain data types (for example *NOT string* is invalid).

As mentioned above, each visitor makes certain the the above rules are not being broken by the AST Node they are visiting.

Given an ASTVariableDeclaration Node for the statement "var x:int = 5 * 2 + 4;", its visitor will:

- Check if a variable named "x" is already declared within the current scope, using the isVariableInSymbolTable method.
- Call the expression's visitor, so that the data type of the expression is stored inside the global variable *typeOfCurrentVariable*.
- Checks that the data type in the declaration is the same as that of the expression.

- When determining the data type of the expression, the visitor for the expression is called, in this case it is the ASTBinaryExpr visitor.
- The visitor for each simple expression is called recursively, in this case on 5 and 2+4. The type of each simple expression is stored, and compared to make sure that the operation is being carried out on 2 identical data types.
- Furthermore, it is checked that the operator can in fact be applied to the data type of the operands. Depending on the result of the operation, the *typeOfCurrentVariable* global variable is updated. This was necessary because for example performing == on two variables of type Int, results in a Bool value.

Task 5

Interpreter Execution Pass

The Interpreter Execution Pass traverses the AST using the visitor pattern, similarly to the SemanticAnalysis visitor, however executes the AST Statements. This pass 'runs' the program that the user has programmed.

The Interpreter Execution Pass again makes use of the Symbol Table class, and thus the map of strings (identifiers) to variable properties.

The Variable Properties class was modified to include more attributes which are needed by the interpreter, these are:

- boolValue, intValue, realValue and stringValue, four variable each of their respective type, that will store the variables value. It isn't the most elegant solution to have a variable for each data type given one variable can only be of one type and will only ever use one of them, but it is rather easy to implement, works perfectly for what is required, and is easy to understand.
- A pointer to an ASTFormalParameters Node, which is used to store the parameters of a function.
- A pointer to an ASTBlock Node, which is used to store the block of code of a function.

The SymbolTable class was also modified to include more functions which were required by the Interpreter Execution Pass, these are:

- setBoolValueOfVariable, setIntValueOfVariable, setRealValueOfVariable

and setStringValueOfVariable, which given the variable's identifier and value in the respective data type, look for the identifier in the Symbol Table (multimap) and set the value of the variable to the passed value.

- getVariablePropertiesOfVariable, which given a variable's identifier, returns the VariableProperties of that variable. (This method asserts that the variable is in the Symbol Table)
- isFunctionIdentifierInSymbolTable which returns whether at least one function with the passed identifier exists in the symbol table.
- getVariablePropertiesOfFunction which returns a vector of VariableProperties of each function found in the symbol table with the specified identifier.

Similarly to what was done in visitAST in the SemanticAnalysis class, in the Interpreter Execution pass, the constructor pushes an empty symbol table for the global scope in the *symbolTableStack* vector.
The visitAST method then iteratively visits each AST Statement Node.
Finally the destructor pops the Symbol Table pushed by the constructor (which, as mentioned, represents the global scope).

All the other statement node visitors execute the statement that they represent, while the expression node visitors evaluate the expression that they represent.

Given an ASTVariableDeclaration Node for the statement "var x:int = 5 * 2 + 4;", its visitor will:

- call the ASTExpression's (that is a sub node of the ASTVariableDeclaration node) visitor.
  - The ASTExpression node is an ASTBinaryExpr node, so it is visited by that visitor.
  - First the visitor will visit the **second** simple expression, so that its value will be pushed to the end of its respective data type's global vector. In this case the second simple expression is 2+4, so the ASTBinaryExpr visitor is called recursively.
  - Next, the visitor is called for the **first** simple expression, so that it is at the top of its respective data type's (which is the same

as the second one - proven by the semantic analysis pass) global vector.

- Depending on the Operator of the ASTBinaryExpr (in this case ***** multiplication)
  - *op1* is set to the the value at the end of the global vector of values of whichever data type the two simple expressions are.
  - the global vector of values from which Op1 was set is popped, so that the evaluation of simpleExpression2 is now at the end of the vector.
  - *op2* is set to the value at the end of the global vector of values of whichever data type the two simple expressions are.
  - the global vector of values from which Op2 was set is popped, so that the whatever value was at the end of the vector before execution of this visitor is left on top for when control is returned.
  - the resulting value is pushed to the end of its respective vector of values. For most operators, this is the same as the data type of the operands, however in the case of boolean operators, such as ==, the resulting value is pushed to the vector of boolValues.

- An instance of VariableProperties is initialised with the type of variable of the ASTType sub node inside ASTVariableDeclaration, and VariableType::VARIABLE.
- The value of the VariableProperties is set, again depending on the type of the variable inside ASTType. The value at the end of the respective vector is stored inside the VariableProperties instance and popped from the vector.
- The pair made up of the variable's identifier (stored inside the ASTIdentifier subnode) and the VariableProperties instance initialised above is added to the current scope's symbol table.

Task 6

The REPL - Read Execute Print Loop

The REPL allows the user to enter single statements into a command line-like interface which will be parsed, semantically analysed and executed. The REPL also allows loading of MiniLang program files (#load), and displaying the global scope's symbol table (#st).

On launch of the Main method, the user is asked whether he would simply like to execute the contents of the *input* file by pressing **i**, or whether they would like to launch MiniLangI, by pressing **m**.

The MiniLangI method first asks the user to input either **#st**, **#load**, **#exit** or any MiniLangI statement.

If #st is entered, An iterator goes through the global scope's symbol table, and outputs every map pair, i.e. every identifier paired with its data type, whether it is a Variable or a Function, and a list of its parameters' data types if it is a function.
The Variable Properties were printed by simply implementing a toString method in the VariableProperties class.

If #load is entered, The user is asked for the name of the file that they would like to be loaded. The input file must be located inside the **MiniLangPrograms** folder. Once The user has entered the file name:

- The Lexer's vector of chars which might be storing any previous MiniLang code is cleared, and the inputProgramIndex and lineNumber are reset. The Lexer's readProgram method is called, which converts the file that the user has asked to be executed into a vector of chars which will later be traversed by getNextToken().
- The parser's AST is also cleared, so that any previously parsed ASTNodes will not be re-executed by the semantic analysis and interpreter execution passes. Then the parser is called to parse the input file and generate a new AST.
- Semantic Analysis is performed on the AST generated by the parser to check for any semantic errors and update the symbol table.
- Finally the Interpreter Execution Pass is performed on the AST to execute the input program and update the symbol table.

If #exit is entered, MiniLangI simply exits, deletes the parser, semanticAnalysis and interpreterExecution pointers, which in turn call their respective destructors and no memory is leaked.

If a MiniLang statement is entered, the process is very similar to what is followed by the #load command. The only difference is that the vector of chars, instead of being initialised by the Lexer's *readProgram* method, is initialised by the Lexer's *setProgramCode*, which simply takes the string (MiniLang statement) rather than a file name.
The rest of the process is identical, the inputProgramIndex and lineNumber are reset, the AST is cleared, the parser is called, followed by semantic analysis and finally interpreter execution.

In all cases, from one command to other, the semantic analysis's and interpreter's symbol tables are retained, so that two input files or input statements (or a mix of both) can refer to variables or functions from a previous command.

Task 7

Testing

The first tests that were tried out on the Compiler were those provided in the Assignment Specification. Both ran successfully and returned the expected outputs.

The program shown in Figure 7.1, factorial, computes the factorial of any given integer recursively. This program demonstrates that function declarations, if-else statements, return statements, binary expressions, recursive function calls and print statements do function correctly.

This program was then loaded from the REPL interface, as shown in Figure 7.2. This demonstrated that the REPL was capable of loading a program from a specified file. Furthermore, *print fact(5);* was then run, which also showed that the function declaration that was done in the loaded file, was still in the symbol table, and the REPL was able to execute a statement which called that function.

Another program, that shown in Figure 7.3, functions, has two function declarations, named identically and returning the same data type, where both return whether the passed values are equal or not. One however accepts two integers, while the other accepts two strings.
What this program is testing is whether the compiler can handle functions with identical identifiers and return types, but different input parameter types (different method signatures). Not only that but the test demonstrates that it is capable of choosing the correct function definition from the symbol table when given a function call according to the types of the parameters.

```
1  def fact (x : int) : int {
2      if (x == 0) {
3          return 1;
4      } else {
5          return x * fact(x - 1);
6      }
7  }
8  print fact(10);
```

Figure 7.1: Factorial - Recursion Testing

```
Would you like to compile the code inside 'input', or Launch MiniLagI?
Press 'i' and enter for the input file or 'm' and enter for MiniLangI
m
Please enter '#st', '#load', '#exit', or a MiniLang statement
MLi> #load
Please enter the file name you wish to load. The file must be located ins
factorial
No Errors found during Semantic Analysis
3628800
MLi> print fact(5);
No Errors found during Semantic Analysis
120
MLi> print fact(0);
No Errors found during Semantic Analysis
1
MLi> #exit
Exiting MiniLangI
```

Figure 7.2: Factorial - REPL Testing

```
1  def areEqual (x : int, y : int) : bool {
2      print "ints";
3      if (x == y) {
4          return true;
5      } else {
6          return false;
7      }
8  }
9  def areEqual (x : string, y : string) : bool {
10     print "strings";
11     if (x == y) {
12         return true;
13     } else {
14         return false;
15     }
16 }
17 print areEqual(5, 5);
18 print areEqual(5, 15);
19 print areEqual("hello", "hello");
20 print areEqual("hello", "bye");
```

Figure 7.3: Functions - Function Declaration and Calling Testing

A final program, that shown in Figure 7.5, iterativefibonacci, defines a function *fibonacci*, which takes an integer, representing the index of a term in the Fibonacci sequence, and returns the value of that term in the sequence.
The function was defined iteratively, which served to test the While statement, and since MiniLang doesn't support *if, else if, else*-like statements, nested *if-else*s were used, which demonstrated again the functionality of if statements. This program also showed that block work correctly, and that multiple return statements in one function declaration are also supported correctly.

If the user would like to see the Symbol Table of an input program, they can run *#st*, after loading a program via *#load* or inputting any number of MiniLang statements directly to the REPL interface. The symbol table generated from running the example code from page 2 of the assignment specification is shown in Figure 7.7.

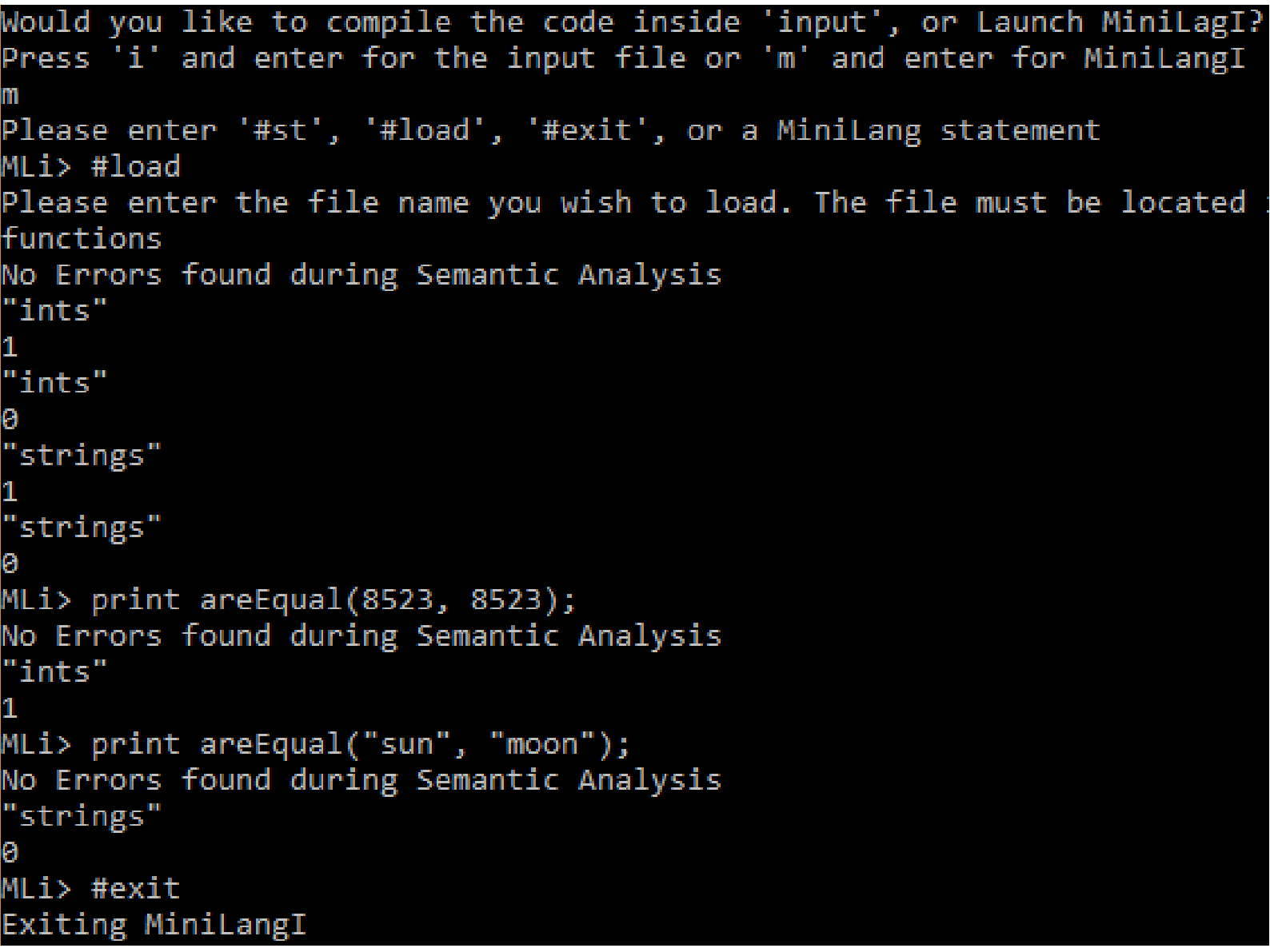

Figure 7.4: Functions - REPL Testing

```
1  def fibonacci (n : int) : int {
2      var counter : int = 0;
3      var t1 : int = 0;
4      var t2 : int = 1;
5      var fibonacci : int = 0;
6      if (n == 1) {
7          return 0;
8      } else {
9          if (n ==2) {
10             return 1;
11         } else {
12             while(counter < n-2) {
13                 set fibonacci = t1 + t2;
14                 set t1 = t2;
15                 set t2 = fibonacci;
16                 set counter = counter + 1;
17             }
18             return fibonacci;
19         }
20     }
21 }
22 print fibonacci(1);
23 print fibonacci(5);
24 print fibonacci(15);
```

Figure 7.5: IterativeFibonacci - Nested if-else statements, While loop and Multiple Return statements in a block Testing

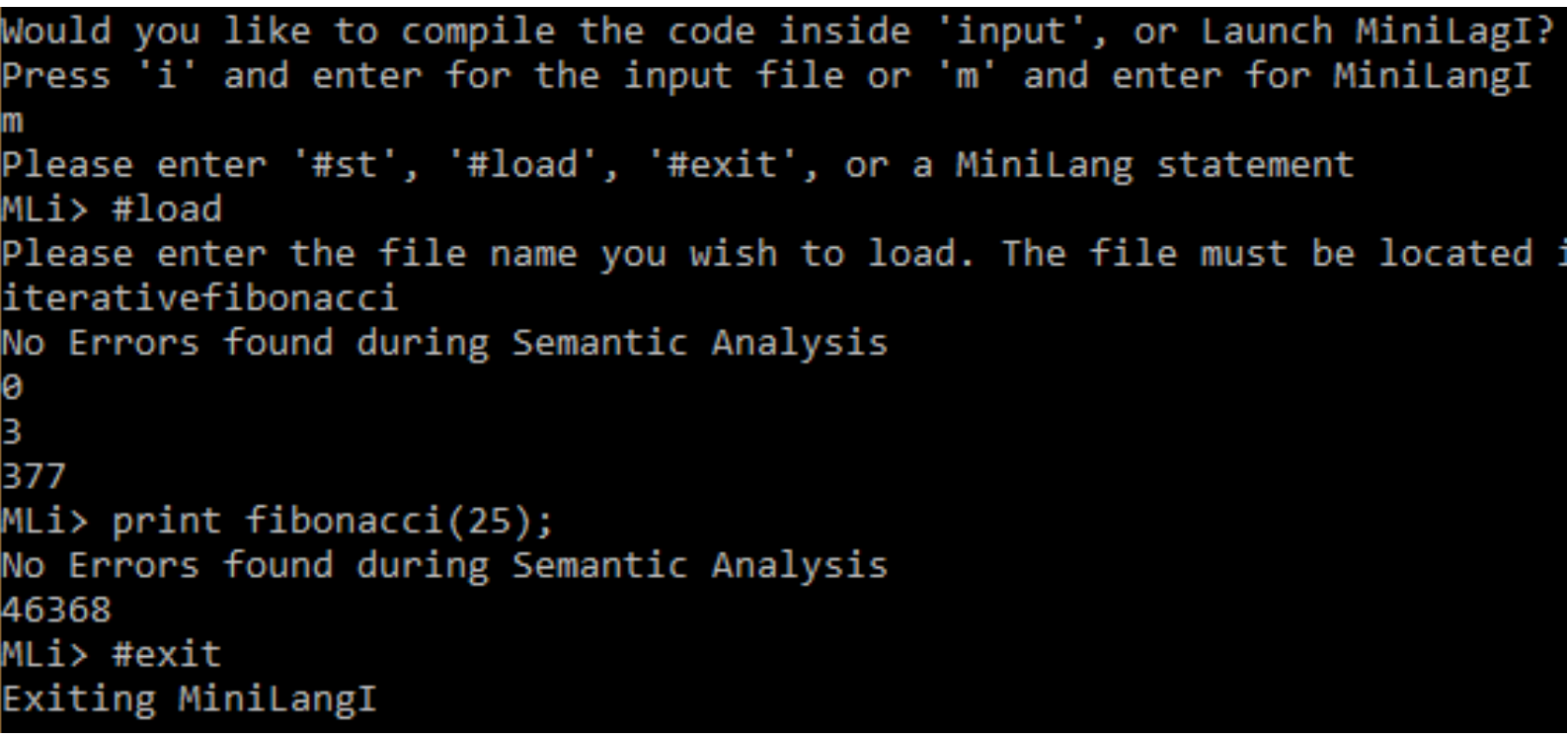

Figure 7.6: IterativeFibonacci - REPL Testing

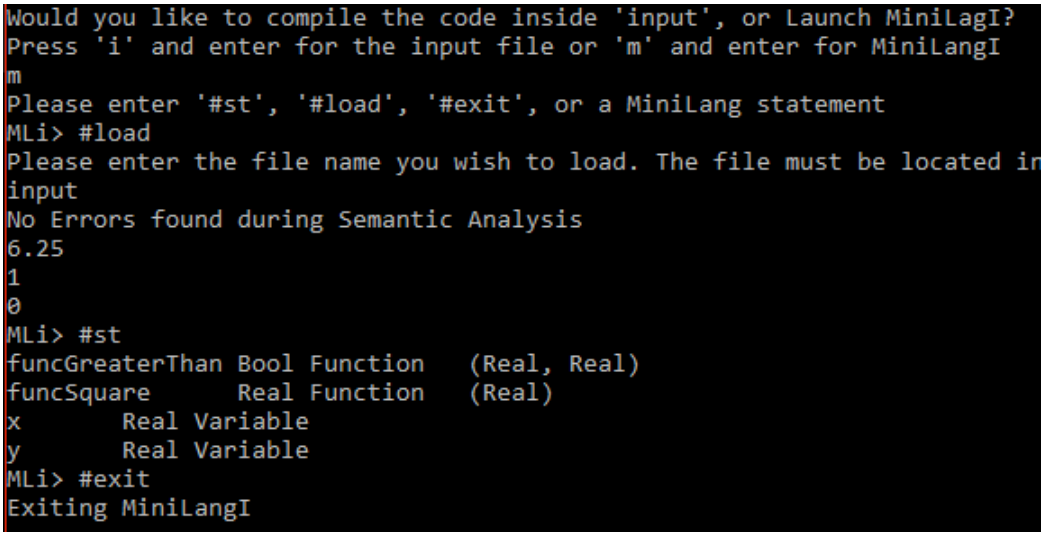

Figure 7.7: Symbol Table from REPL for code from page 2 assignment specification

Finally, if the user would like to see the XML representation of an input program's AST, they need to place the desired code in the *input* file, and when running the program, choose *i*. The XML is not printed in the REPL to avoid unnecessary clutter. A sample XML representation is included in Figure 7.8, which is for the AST of the code snippet provided on page 2 of the assignment specification.

```
<AbstractSyntaxTree>
        <FunctionDeclaration>
                <Identifier>funcSquare</Identifier>
                <FormalParameters>
                        <FormalParameter>
                                <Identifier>x</Identifier>
                                <Type>Real</Type>
                        </FormalParameter>
                </FormalParameters>
                <Block>
                        <Return>
                                <BinaryExpression>
                                        <Operator>'*'</Operator>
                                        <Identifier>x</Identifier>
                                        <Identifier>x</Identifier>
                                </BinaryExpression>
                        </Return>
                </Block>
                <Type>Real</Type>
        </FunctionDeclaration>
        <FunctionDeclaration>
                <Identifier>funcGreaterThan</Identifier>
                <FormalParameters>
                        <FormalParameter>
                                <Identifier>x</Identifier>
                                <Type>Real</Type>
                        </FormalParameter>
                        <FormalParameter>
                                <Identifier>y</Identifier>
                                <Type>Real</Type>
                        </FormalParameter>
                </FormalParameters>
                <Block>
                        <VariableDeclaration>
                                <Identifier>ans</Identifier>
                                <Type>Bool</Type>
                                <BooleanLiteral>1</BooleanLiteral>
                        </VariableDeclaration>
                        <If>
                                <BinaryExpression>
                                        <Operator>'>'</Operator>
                                        <Identifier>y</Identifier>
                                        <Identifier>x</Identifier>
                                </BinaryExpression>
```

Figure 7.8: XML Representation snippet