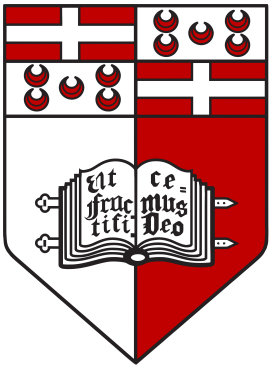

CPS2000 - Compiler Theory and Practice
-
Course Assignment 2016/2017

Full name: Miguel Dingli
I.D: 49997M
Course: B.Sc. (Hons) in Computing Science

Table of Contents

Task 1 – Table-Driven Lexer

1.1 – DFA

This section will start out by presenting a DFA based on the given EBNF. This DFA will later be encoded as a table of states, since the lexer will use a *table-driven* approach. The DFA is presented below in Figure 1.

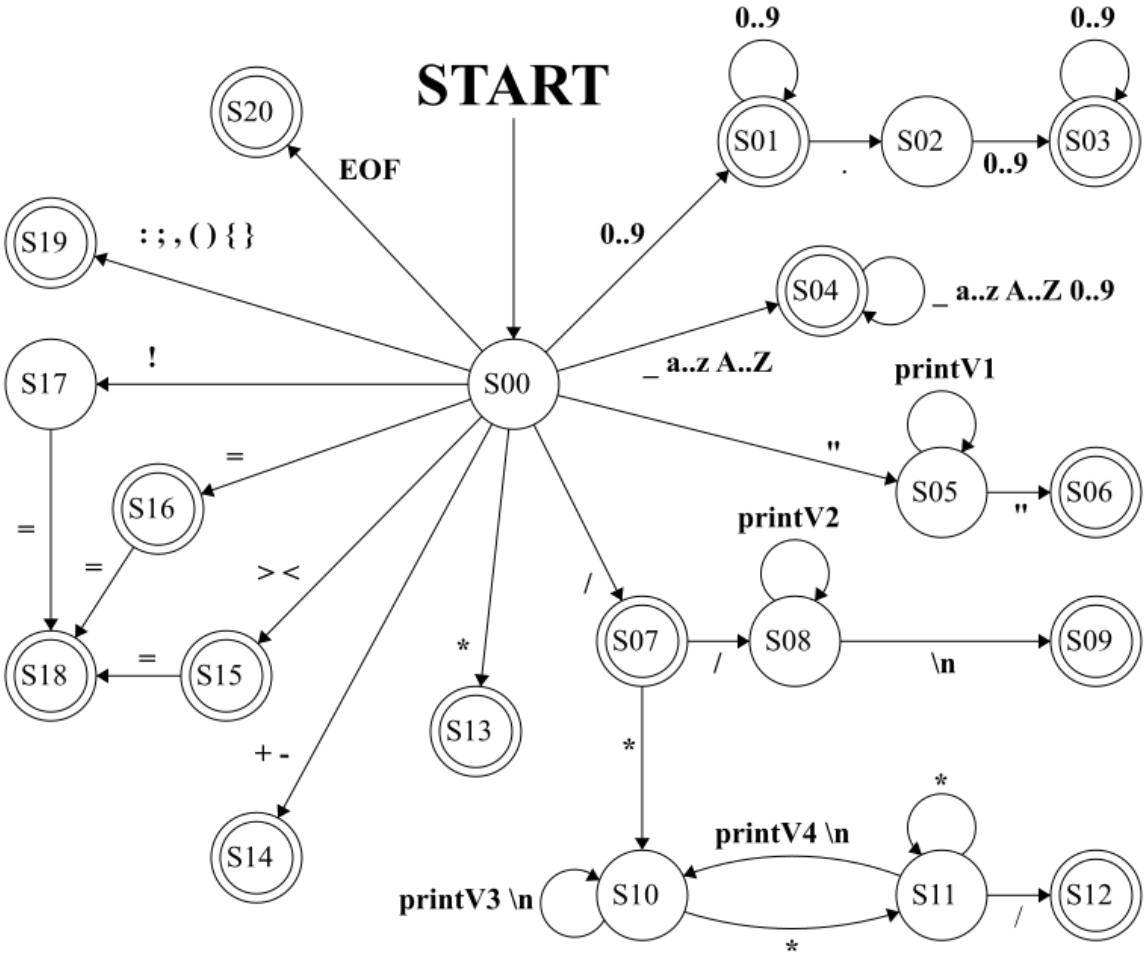

Figure 1 – DFA for MiniLang

A table (Table 1) to briefly introduce these states is presented overleaf. Note that throughout the DFA, four types of printable character sets are considered; **printV1**, **printV2**, **printV3**, and **printV4**. These are based on the definition of *Printable* in the EBNF but differ slightly based on their purposes which are defined below:

- **printV1** is used for string literals and is similar to *Printable* but excludes inverted commas, meaning that it is assumed that strings cannot include inverted commas.
- **printV2** is used for single-line comments and matches exactly with *Printable*.
- **printV3** is used for multi-line comments and is similar to *Printable* but excludes the asterisk symbol to eliminate any ambiguities when searching for the comment-closing asterisk followed by forward slash.
- **printV4** is also used for multi-line comments but excludes both the asterisk symbol and the forward slash, again, to eliminate any ambiguities when searching for the comment-closing forward slash.

State	Description
S00	Start state for the lexing process.
S01-S03	Group of states for integer literals and real literals. For integers, only up to state **S01**.
S04	State for identifiers, starting with underscore or letter, followed by underscores, letters, and/or digits. Through further processing, identifiers can be identified to be keywords.
S05-S06	Group of states for string literals which start and end with inverted commas and use printV1.
S07-S12	Group of states for single and multi-line comments and for the division symbol. For the division symbol, traversal ends at **S07**. **S07** to **S09** are used for single-line comments ending with a newline (\n) symbol. **S07** and **S10** to **S12** are used for multi-line comments.
S13	State used for product (*) multiplicative operator.
S14	State used for plus (+) and minus (-) additive operators.
S15-S18	Group of states used for relational operators and the assignment equals (=) symbol. For '>' or '<', **S15** is used as final state. For assignment symbol (=), **S16** is the final state. Finally, **S18** as a final state means a two-character relational operators ending with equals.
S19	State used for punctuation symbols including colon, semicolon, comma, and parentheses.
S20	State used exclusively for the End-Of-File (EOF) symbol.

Table 1 – Table of States

1.2 – Lexer Implementation

The actual implementation of the Lexer will now be discussed, with reference to the DFA (Figure 1).

1.2.1 – The States Enumeration

The states **S00** to **S20** were listed in an enum (Listing 1). Throughout the program, states are treated as integers from 0 (S00) up to 20 (S20). NUMBER_OF_STATES by default stores the number of states in the enum.

```
enum State {
    S00,
    S01,
    ...
    S19,
    S20,
    NUMBER_OF_STATES
};
```

Listing 1 – States Enumeration

1.2.2 – The Categories Enumeration

Another enumeration **Category** was defined (Listing 2). This enumeration defines a set of categories to significantly simplify the table in the table-driven approach in that characters belonging to the same category are grouped. Similar to the previous enum, a NUMBER_OF_CATEGORIES entry was included.

```
enum Category {
    CT_Digit,        // digit
    CT_DecimalPt,    // decimal point
    ...
    CT_EOF,          // end-of-file symbol
    CT_Other,        // other
    NUMBER_OF_CATEGORIES
};
```

Listing 2 – Categories Enumeration

A **charCat** function converts a character into a category. Note: the notions of *punctuation* and *printable* categories are flexible. Character belonging to a more specific category will not be categorised into a more general one. For example, although a digit is a printable character, it is categorized as a digit.

1.2.3 – The Token Types Enumeration

Another enumeration **TkType** was defined (Listing 3). This defines a set of token *types* that the lexer uses to produce tokens from when going through a given stream of characters.

```
enum TkType {
     TK_Bool,
     TK_Integer,
     ...
     TK_Comment,
     TK_EOF,
     NUMBER_OF_TOKEN_TYPES
};
```

Listing 3 – Token Types Enumeration

Among the types of tokens:

- **Literals** (*boolean*, *integer*, *real*, and *string*) are represented individually;
- **Identifiers** are grouped under a general *identifier* token type;
- **Operators** are grouped as *multiplicative*, *additive*, and *relational*;
- **Punctuation** symbols are defined individually;
- **Keywords** are defined individually;
- **Types** (*real*, *int*, *bool*, *string*) are grouped under a general *type* token type;
- **Comments** are grouped under a general *comment* token type;
- **The end-of-file symbol** is given a token type of its own.

Note: although a token type for comments was included, tokens of this type will not be returned by the lexer. It merely identifies that it has formed a comment token and instead produces and returns the next token.

1.2.4 – The Token Class

The actual structure of tokens is defined in the **Token** class, which is presented in Figure 2. Each token is at least defined by a token type and lexeme (the actual characters that make up the token). A token may additionally store a numeric value in the case of integers and reals, instead of just the lexeme. Note that in the case of integers, *getIntegerValue()* rounds the double to an integer.

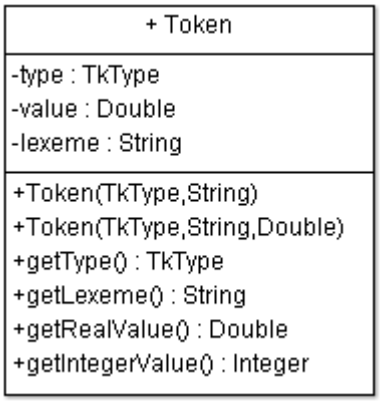

Figure 2 – Token Class

1.2.5 – The DFA Class

For the actual table in the table-driven functionality, a **DFA** class was created and is presented in Figure 3. The DFA class will not store any details such as the current state, and hence all members were made stateic.

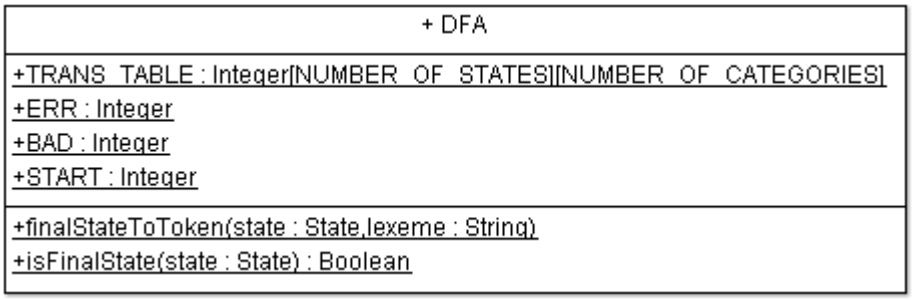

Figure 3 – DFA Class

The constants **ERR** and **BAD** extend the number of states in the DFA by two. These denote the error state, used when an undefined transition occurs in the DFA. and the bad state, which serves as the base case state in the "state stack" that will be discussed in the next section. **START** is simply set to **S00** from the states enum. The two-dimensional array **TRANS_TABLE** encodes the transition function of the DFA into a table (Listing 4).

```
const int DFA::transitionTable[NUMBER_OF_STATES][NUMBER_OF_CATEGORIES] = {  /*
     Dgit DecP Lett Prnt Star Fwsl PlMn GrLs Excl Eqls UndS Quot Pnct NewL EofF Othr  */
    {S01, ERR, S04, ERR, S13, S07, S14, S15, S17, S16, S04, S05, S19, ERR, S20, ERR}, //00
    {S01, S02, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR}, //01
    {S03, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR}, //02
    {S03, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR}, //03
    {S04, ERR, S04, ERR, ERR, ERR, ERR, ERR, ERR, ERR, S04, ERR, ERR, ERR, ERR, ERR}, //04
    {S05, S05, S05, S05, S05, S05, S05, S05, S05, S05, S05, S06, S05, ERR, ERR, ERR}, //05
    ...
    {ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR}, //19
    {ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR}  //20
};
```

Listing 4 – The Transition Table

In the table, a transition is encoded by considering a row as the current state and a column as the category of the character under consideration. The value at the particular column and row is the next state to be transitioned to. Example (below): if a digit is read from state **S01**, the transition loops the traversal to **S01**.

```
     Dgit DecP Lett Prnt Star Fwsl PlMn GrLs Excl Eqls UndS Quot Pnct NewL EofF Othr  */
    {S01, ERR, S04, ERR, S13, S07, S14, S15, S17, S16, S04, S05, S19, ERR, S20, ERR}, //00
    {S01, S02, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR, ERR}, //01
```

By definition of the error state, any undefined transition is set to ERR. One might make the observation that the rows for final states with no outward transitions have ERR for any category, since no further transition is possible. Although a row of ERR values can be seen as a space waster, these are still important since they let the lexer know that an unexpected character was read, meaning that a token is likely to have ended.

The function **finalStateToToken** expects a final state and a lexeme that was formed while transitioning.

- For final state **S04** which caters for identifiers, the lexeme is compared to all possible keywords. If the lexeme matches with no keyword, it is assumed to be an identifier.
- For final state **S19** which caters for punctuation, the lexeme is expected to be of length *1*.
- For the remaining final states, a token with a token type based on the final state is returned.

1.2.6 – The Lexer Class

The final class in this section is the actual **Lexer** class, presented in Figure 4.

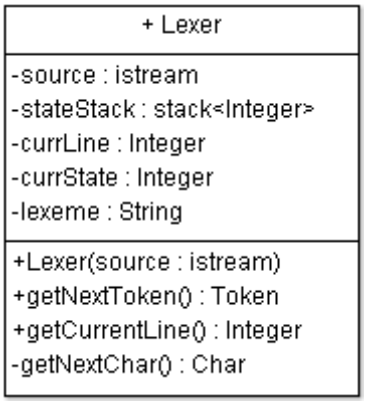

Figure 4 – Lexer Class

The lexer expects an input stream with the source code. **stateStack** stores states in the order that they are encountered, **currLine** stores the current source line (for error messages), **currState** stores the current state in the traversal, and **lexeme** stores the sequence of characters accumulated in the token-forming process.

The main use of **getNextChar()**is to return the next character, but it also increments the current line and returns EOF for every call to it beyond the end of file. For the EOF token to be detected, an extra character has to be read to cause a transition to ERR. This is made possible since the function returns multiple EOF symbols.

The **getNextToken()** function contains the core of the Lexer functionality and is discussed below in point form:

1. At the start, the current state is the start state, the lexeme is initialized as a blank string, and the state stack is initialized with the BAD state, which is the only time that this state is pushed onto the stack.

2. A loop now clears all whitespace that prefix the characters of interest. After the loop, the stream is rolled back a character so that the extra non-whitespace character read will be un-read.

3. Another loop now models the DFA transitioning and runs until the ERR state is detected, indicating an invalid transition. In each iteration, the next character from the stream is appended to the lexeme and the new state is pushed onto the state stack. If at any point a final state is encountered, the state stack is cleared since previous states are not needed anymore. Note: transition to ERR happens in two cases:
    a. Lexeme so far makes sense as a token, but an extra character was read; or
    b. Lexeme so far does not yet make sense as a token, meaning that a syntax error is present.

4. Once the above loop terminates, the last state transitioned to is the ERR state. Another loop now pops states from the state stack until the latest final state pushed onto the stack or the BAD state is found. The popping of states can be seen as traversing the DFA in reverse. For each state popped, the lexeme is shortened by one character to counteract the reading of a single extra character.

5. At this point, the current state is either a final state or the BAD state. In the final part of the function, if the current state is a final state, then the token to be returned is generated. However, if this token is a comment token, getNextToken() is called again to skip the comment token. Otherwise, if the current state is not a final state, a syntax error exception is thrown.

Task 2 – Recursive-Descent Parser

2.1 – UML Diagram of AST Classes

Moving on to the Parser, this section will start out by presenting a UML diagram (Figure 5) which contains the various classes that will be used to come up with an abstract syntax tree as a result of the parsing.

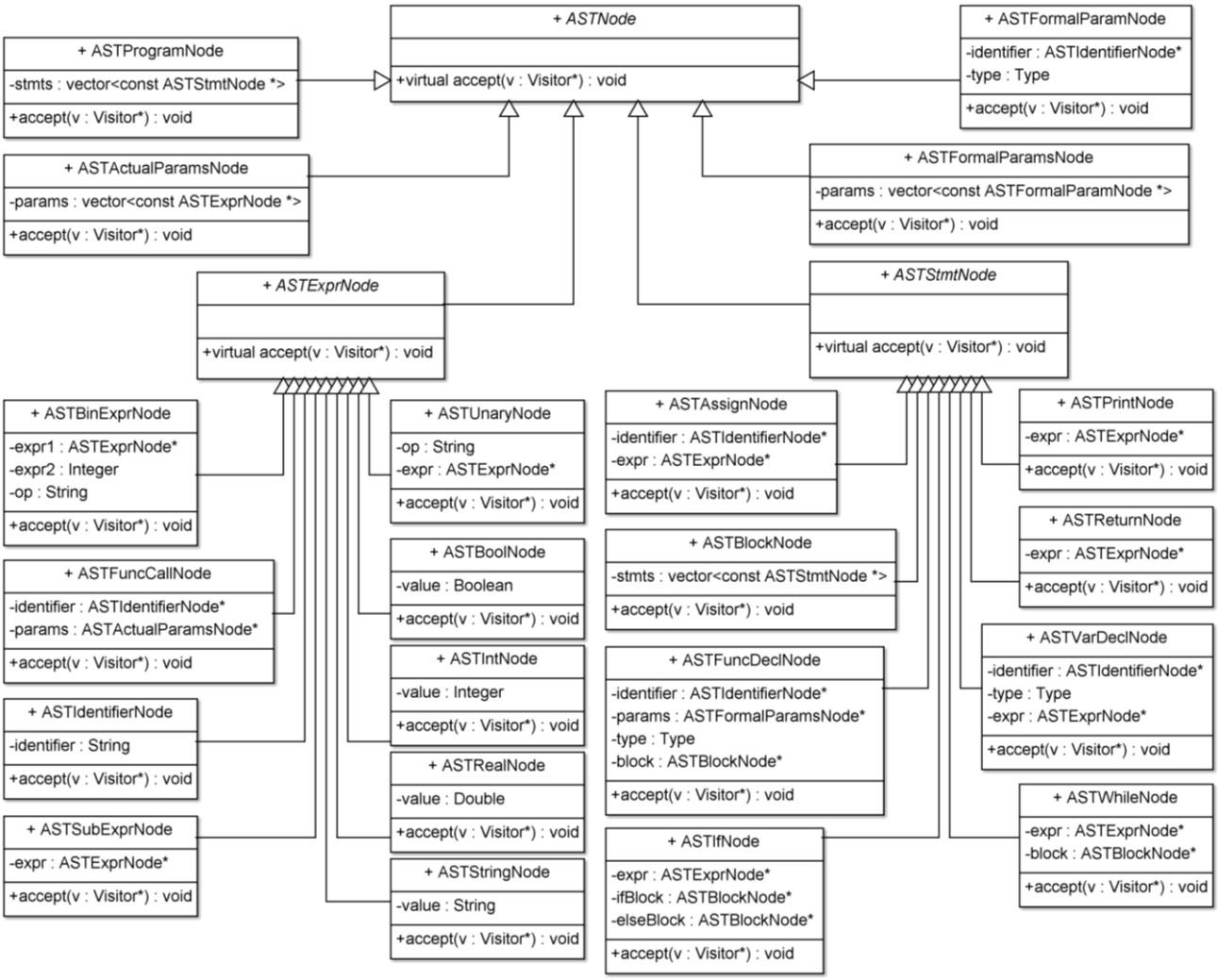

Figure 5 – UML Diagram of AST Classes

An important note is that the UML diagram above excludes constructors, getters, and destructors and instead focuses on the private member variables of each class. The constructor for each class accepts values for all the variables and simply sets these values to the variables, a getter exists for each private member variable, and destructors perform delete on all member variables that are (or consist of) pointers to AST objects. Each class includes an **accept** function which will essentially be used in the coming tasks for the visitor pattern.

The most important aspect of the AST classes, as far as the Parser is concerned, is their private member variables since these will store the information that collectively makes up the final abstract syntax tree. The member variables of all non-abstract classes will now be discussed (overleaf), starting from the expressions.

- **ASTBinExprNode** represents a binary expression and consists of two expression nodes that represent the left-hand-side and right-hand-side of the expression, and an operator as a string.
- **ASTFuncCallNode** represents a function call and consists of an identifier for the function name and a pointer to the actual parameters node which represents zero or more actual parameters.
- **ASTIdentifierNode** represents an identifier and simply consists of a string which stores the identifier.
- **ASTSubExprNode** represents a sub-expression and simply consists of the expression which would be found in parentheses. The parentheses are not needed as they are implied by the class of the node.
- **ASTUnaryNode** represents a unary operation and consists of the operator as a string, which can be either a minus or the *not* keyword, and an expression on which the operator is being applied.
- **ASTBoolNode**, **ASTIntNode**, **ASTRealNode**, and **ASTStringNode** represent boolean, integer, real, and string literals and simply store a value of their respective type (i.e. bool, int, double, and string).

The nodes that represent statements will now be discussed:

- **ASTAssignNode** represents a variable assignment and consists of an identifier for the variable name and the expression, the result of which is being set to the variable.
- **ASTBlockNode** represents a block and consists of a collection of statements that make up the block.
- **ASTFuncDeclNode** represents a function declaration and consists of an identifier for the function name, a pointer to a formal parameters node representing zero or more parameters, the return type of the function, and a pointer to a block representing the statements that make up the function body.
- **ASTIfNode** represents an if-else statement and consists of the expression that is being evaluated as the if condition, a block representing the *if* body, and another block representing the *else* body. Since the *else* block is essentially optional, this is set as a null pointer in cases where it is not needed.
- **ASTPrintNode** represents a print and consists of an expression, the result of which is being printed.
- **ASTReturnNode** represents a function return and consists of the expression which is being returned.
- **ASTVarDeclNode** represents a variable declaration and consists of an identifier for the variable name, the variable's type, and an expression, the result of which is being set as the variable's initial value.
- **ASTWhileNode** represents a while loop and consists of an expression to be evaluated as the loop condition and a block representing the body of the loop.

The remaining classes that are immediate subclasses of the ASTNode class will now be discussed:

- **ASTProgramNode** represents a whole program and consists of a collection of statements which represents the statements that make up the program.
- **ASTActualParamsNode** represents actual parameters that are passed as arguments to a function and consists of a collection of zero or more expressions, the results of which will be passed as arguments.
- **ASTFormalParamsNode** represents formal parameters of a function and consists of a collection of zero or more pointers to individual formal parameters.
- **ASTFormalParamNode** represents an individual formal parameter of a function and consists of an identifier for the parameter name and a type variable representing the parameter's type.

These classes were essentially modelled on the EBNF non-terminals that consist of further non-terminals in their right-hand-sides. Terminals included in the right-hand-sides of these non-terminals are implicitly encoded in the definition of the classes, meaning that the terminals do not need to be included explicitly. Some classes used **Type**. This comes from an enumeration that will be discussed in the start of the next section.

2.2 – Parser Implementation

2.2.1 – The Types Enumeration

An enumeration for types (Listing 5) defines the four data types allowed in *MiniLang*.

```
enum Type {
    Real,
    Bool,
    Int,
    String,
    NUMBER_OF_TYPES
};
```

Listing 5 – Types Enumeration

A simple function **strToType(...)** to convert a type from string to a value from the above enumeration was also defined. It attempts to match the string argument with one of the four possible types in string form and returns the result as a value from the enum. An exception is thrown if the argument is unrecognized as a type.

2.2.2 – The Parser Class

The **Parser** class, presented in Figure 6 in a reduced form, consists of many similar functions each serving the purpose of parsing a particular type of expression, statement, etc, and producing a node for the abstract syntax tree. Since an ASTProgramNode represents whole program, the function **parse()** which starts the parsing process returns a pointer to an ASTProgramNode, which will be the root of the whole abstract syntax tree.

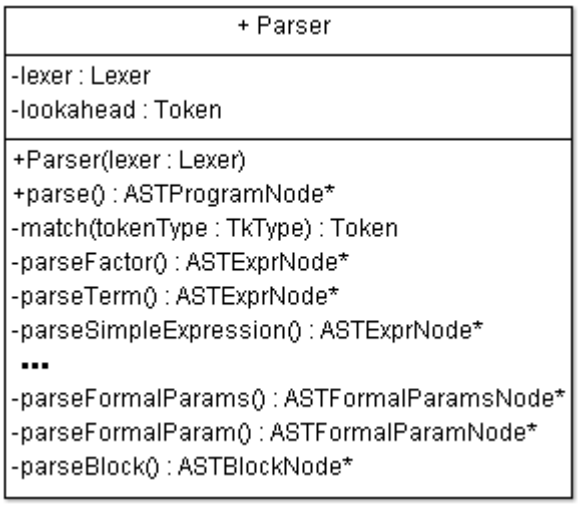

Figure 6 – Parser Class

The parser will use the lexer, passed to the constructor, for a stream of valid tokens using the **getNextToken** lexer function. The constructor initializes the **lookahead** variable to the first token. The aim of the lookahead is to store the latest token obtained which can then be analysed by the **match** function.

The **match** function is a simple but important function which, given a token type, checks that the token type of the token held by **lookahead** matches with the token type argument. In each of the remaining functions, the **match** function will be called as needed to unambiguously check that a sequence of tokens matches the make-up of a particular expected non-terminal.

If the types of the argument and the lookahead token match, lookahead is set to the next token and the previous token held by the lookahead is returned. The previous token is returned after it has been matched so that if it stores values that will be passed to AST node constructors, these can be extracted. On the other hand, if the types do not match, a syntax error is thrown since the type of a token was not as expected.

2.2.2.1 – The Parse Functions

As stated previously, the Parser class contains many similar functions each serving the purpose of parsing a particular type of statement, expression, etc. Instead of decribing each function individually, some general points will be discussed.

In general, the parse functions will make use of a mix of calls to the *match* function, and calls to other parse functions. A call to the *match* function generally means that a particular token is expected at a particular point in the parsing process, while calls to other parse functions means that the next sequence of tokens expected should follow a specific sequence depending on the parse function called.

As an example, consider the parse function for the variable assignment statement (Listing 6). For the *set* keyword, the equals symbol, and the semicolon, a simple match is used to check that these elements are present. In the case of the identifier, since the identifier will be needed for the ASTAssignNode, the match's return value is stored. For the expression that is being set as the new value in the assignment, the parse function for an expression is called, and the result is stored so that it is included in the ASTAssignNode.

```
ASTAssignNode *Parser::parseAssignment(void) {

    match(TK_KW_Set);
    Token id = match(TK_Identifier);
    match(TK_Equals);
    ASTExprNode *expr = parseExpression();
    match(TK_Semicolon);
    return new ASTAssignNode(id.getLexeme(), expr);
}
```

Listing 6 – Parse Function for Return Statement

In some cases, the parse function called depends on the sequence of tokens that follows, rather than the other way round. Such a case is when parsing a statement, which be a variable declaration, if statement, etc. To avoid having to read a set of tokens to identify the type of statement, identification is based on the first token, which is essentially unique for all possible types of statments, e.g. an assignment must start with keyword *set*.

Another important point of discussion is operator precedence in expressions. In the parse functions, this is implicitly handled by three functions *parseExpression()*, *parseSimpleExpression()*, and *parseTerm()*, having a common structure. Consider the following expression consisting of one of each of the three operator types.

$$1 > 2 + 3 * 4$$

By precedence, the operators should be evaluated from right-to-left. Although *parseExpression()* is called first when an expression is encountered, operator precedence is successfully achieved. In the expression, (1) and $(2 + 3 * 4)$ are parsed as simple expressions, (2) and $(3 * 4)$ are parsed as terms, and (3) and (4) are parsed as factors. The same with the operators reversed would still be parsed correctly, with respect to precedence.

Moving on, for function calls and identifiers, since both of these start with an identifier, these were joined into a *parseFunctionCallOrIdentifier()* function which continues parsing if it finds an open parentheses after the identifier, meaning that the start of a function call was found, rather than simply an identifier.

2.3 – End of Section

In this section, the parsing stage was discussed. By the end of a successful parse of a program, in cooperation with the lexer, the parser produces an abstract syntax tree which describes the structure of the program. In the coming tasks, the generated AST will be traversed using a Visitor design pattern. The function that was not given importance was the **accept** function which is present in all AST classes. Each of these functions is identical in that they allow an AST class to accept a visitor.

Task 3 – Generate XML of AST

3.1 – Introducing the Visitor Class

The Visitor design pattern will be used to perform operations on the elements of the AST produced by the parser. For the purpose of having an interface on which visitors can be based, a Visitor class (Figure 7) was designed, containing **visit** functions for every type of AST node. When implemented by subclasses, each **visit** function will perform operations based on the type of node parameter that it expects.

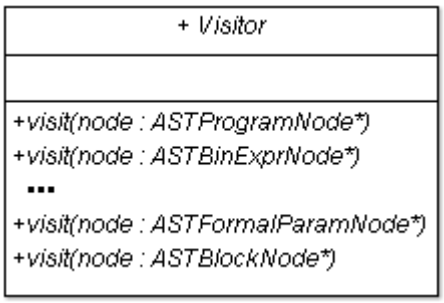

Figure 7 – Visitor Class

Note that the **accept** functions in the AST node classes expect an object of type *Visitor*. This means that any subclass of the above interface can be used with the accept function in each of the AST node classes.

3.2 – The PrintXmlVisitor Visitor Class

For the purpose of outputting a properly-indented XML representation of the AST that was generated by the parser, a **PrintXmlVisitor** visitor class (Figure 8) was created, which implements all the visit functions.

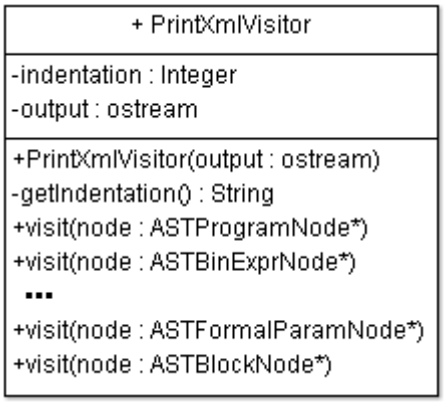

Figure 8 – PrintXmlVisitor Class

The **constructor** for this class takes a reference to an output stream which it uses to output the XML generated, instead of outputing to standard output. The **getIndentation** function simply returns a string consisting of a sequence of tabs that is obtained at the start of a function and is prefixed to all the outputs made by that function. The number of tabs is defined by the **indentation** variable, which is incremented or decremented when entering or exiting an XML element, respectively.

Since the root of an AST is an **ASTProgramNode**, the *visit* function that is expected to run first is the one expecting such a node, following a calling of the program node's *accept* function. From that point onwards, the visit functions will call other *accept* functions which will in turn call further *visit* functions until the AST is fully traversed and operated on.

Throughout the PrintXmlVisitor, some functions add multi-line elements to the XML, while others add single-line elements. Additionally, elements may have attributes. Consider the below listings.

Listing 7 shows an example of a multi-line XML element with attributes. The Unary element includes the unary operator as an attribute of the element and contains the expression on which the operator will be applied. Note how the indentation is incremented just before the expression's *Accept* is called and decremented after.

```
void PrintXmlVisitor::visit(const ASTUnaryNode *node) {

    const std::string tabs = getIndentation();
    output << tabs << "<Unary op=\"" << node->getOp() << "\">" << endl;

    indentation++;
    node->getExpr()->Accept(this);
    indentation--;

    output << tabs << "</Unary>" << endl;
}
```

Listing 7 – Multi-Line XML Element

Listing 8 shows an example of a single-line XML element. Since the integer value will only contain a single value at most, i.e. the integer itself, the integer is placed in the same line as the opening and closing XML tags.

```
void PrintXmlVisitor::visit(const ASTIntNode *node) {

    const std::string tabs = getIndentation();
    output << tabs << "<Integer>" << node->getValue() << "</Integer>" << endl;
}
```

Listing 8 – Single-Line XML Element

On another note, consider Listing 9 below. Note how besides the Assignment element, an extra Expression element is also included around the expression. The main reason why this was done was because the expression itself can be an identifier, and since the assignment element already includes an identifier, if an extra expression element was not included, it would be harder to distinguish between the two.

```
void PrintXmlVisitor::visit(const ASTAssignNode *node) {

    const std::string tabs = getIndentation();
    output << tabs << "<Assignment>" << endl;

    indentation++;
    node->getIdentifier()->Accept(this);
    output << tabs << "\t" << "<Expression>" << endl;  // added for distinction
    indentation++;
    node->getExpr()->Accept(this);
    indentation--;
    output << tabs << "\t" << "</Expression>" << endl; // added for distinction
    indentation--;

    output << tabs << "</Assignment>" << endl;
}
```

Listing 9 – Extra Expression Element

3.3 – End of Section

When control eventually returns to the initial visit call (i.e. that of **ASTProgramNode**) and the closing tag is output, the XML is complete. In the next section, a visitor class for the semantic analysis pass will be discussed.

Task 4 – Semantic Analysis Pass

4.1 – Introduction

For this section, another visitor class was implemented to traverse the AST and perform type and scope checks. Before this is presented, another class that is used in the semantic analysis process will be presented.

4.2 – The SymbolTable Class

The **SymbolTable** class (Figure 9) keeps track of all of the currently active scopes, and the identifiers that were defined in each one. It is treated as a stack where a push indicates a entry into a new scope, a pop indicates a scope exit, and provides a means to insert and/or modify identifiers and their associated type into scopes.

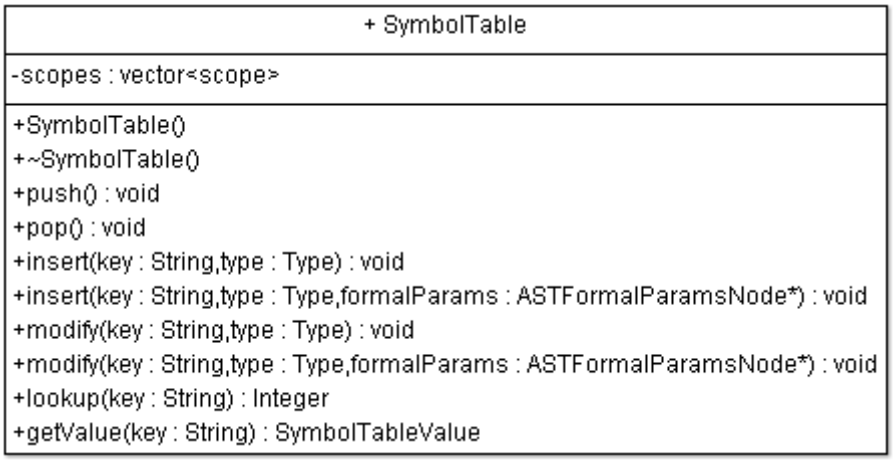

Figure 9 – Semantic Analysis SymbolTable Class

The collection of active scopes is represented by a vector of *scopes* which is manipulated as a stack. A **scope** is represented by a map with string keys to represent identifiers, and **SymbolTableValue** values. The struct SymbolTableValue stores the type associated with the identifier. In the case that the identifier identifies a function, the type represents the return type and the formal parameters are the function's parameters.

A call to **push** adds a new empty map (i.e. scope), whereas a call to **pop** removes the last scope added. Calling **insert** allows for the adding of a map entry to the innermost scope. It creates a SymbolTableValue and adds this to the latest map with the string as the key. If no parameters pointer is provided, this is set as a *nullpointer*. Note that the modification of entries using **modify** will only be used in the R.E.P.L section.

The **lookup** function traverses the scopes vector in reverse, i.e. starting from the innermost scope, to find the latest entry having the specified identifier, and counts the scopes traversed until the identifier is found. **INNERMOST** and **NOTFOUND** are integers that indicate if the identifier was not found or is in the innermost scope. The **getValue** function returns the SymbolTableValue associated with that identifier. Since **lookup** is meant to be called beforehand, an error is thrown if the identifier is non-existent. **getValue** itself performs a lookup and if the identifier is found, the scope count is used as an index to the identifier's scope.

4.3 – The SemAnalysisVisitor Visitor Class

The new **SemAnalysisVisitor** visitor class (Figure 10) is a subclass of the class **Visitor**.

Considering the private members of the class, **ST** is an instance of the SymbolTable class that will be used throughout the class. **FL** is a list of types which will store the list of active function scopes, described by their return type, so that the type of return statement expressions can be matched to the latest function entered.

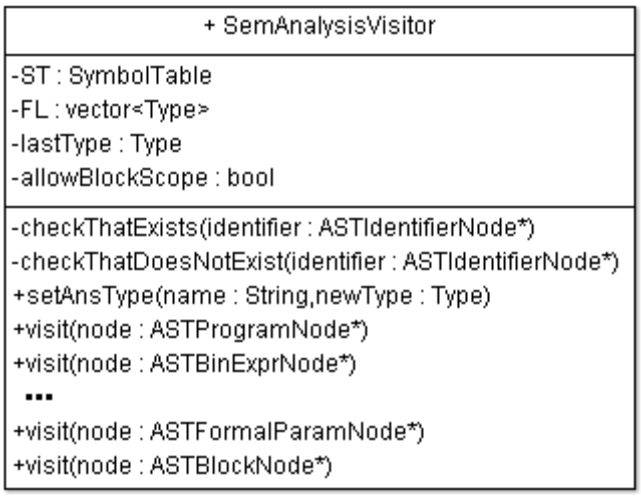

Figure 10 – SemAnalysisVisitor Class

The **lastType** store the resultant type of the last expression visited, meaning that its value is set exactly before returning from a visit to an expression, and usually checked immediately after. This will help to confirm that the resultant type of a particular expression was as expected.

The **allowBlockScope** is set to false whenever the scope of the next ASTBlockNode to be visited needs to be disabled. This is intended to be used when visiting a function declaration since the function's formal parameters need to be in the same scope as the function body block. Hence, the scope created by the function body's ASTBlockNode is disabled and the scope is instead pushed/popped in the function declaration visitor.

The **checkThatExists** and **checkThatDoesNotExist** are simply helper functions which, given an identifier, perform a symbol table lookup to check the existence of the identifier in the symbol table. The *checkThatExists* search all the scopes for the identifier and hence throws an error if the lookup results in **NOTFOUND** whilst *checkThatDoesNotExist* is meant to check that the identifier is not a redefinition of an identifier within the innermost scope, and hence throws an error if the lookup results in **INNERMOST**.

Besides the overridden visit functions, there is a **setAnsType** function. Since this will only be used in the *Real-Eval-Print Loop*, it will be discussed in section 6.

4.3.1 – Some Important Notes

In this subsection, some additional points about the implementation will be presented:

- For convenience to the user, some non-trivial forms of binary expressions are supported:
  - Addition between strings and any other value of any type produces a new string with the other value's string form appended or prepended to the string;
  - Additive or multiplicative operations between an integer and a real produces a real result. This possibility will be further discussed in the next section;
- The ASTIdentifierNode visit function is only visited when the identifier identifies an *existing* variable. Identifiers for new variables/functions and in function calls are handled in more specific visit functions.
- The type of actual parameters in function calls are checked using the formal parameters entry in a SymbolTableValue, while a function return type is checked by checking the last type entered in **FL**.
- The visit function for ASTReturnNode handles cases where a return is found outside of any function.

4.4 – End of Section

When control eventually returns to the initial visit call (i.e. of **ASTProgramNode**), this means that the semantic analysis pass was successful. In the next section, a visitor class for the execution pass will be discussed.

Task 5 – Interpreter Execution Pass

5.1 – Introduction

For this section, yet another visitor class was implemented, but this time to traverse the AST and execute the program. A slightly changed symbol table class was created for this purpose. This class highly resembles and causes duplication of code which could have been highly reduced by using inheritance and/or templates.

5.2 – The SymbolTable and Value Classes

The new **SymbolTable** class, in comparison to that of Figure 9, only adds one new function. This function, **printValues**, will not be discussed in this section since it will only be used in the R.E.P.L, i.e. in the next section.

The purpose of the functions in the new symbol table is identical to those of the previous symbol table. However, a new **SymbolTableValue** is used, which instead of storing a type, stores a generic value (using a new **Value** class) which can be of any of the four allowed types. This is because we are now more interested in the actual value associated with a particular identifier, rather than its type. Additionally, the **modify** functions will not only be used in the R.E.P.L, since they are needed in the case of a variable assignment.

The **Value** class (Figure 12) makes use of a union called **AnyValue** which stores a double, a boolean, an integer, a pointer to a string, or a pointer to an ASTFuncDeclNode. The first four options are used for the value of a variable when an identifier is identifying a variable, while the pointer to an ASTFuncDeclNode is used to keep a pointer to a function when an identifier is identifying a function, so that its block is accessible if it is called.

This class provides functions to *set* the value and to *get* it in a specific type. There is also a getter for the type. In the case of a function declaration, the NUMBER_OF_TYPES value is returned, which is admittingly not good practice, but allows **getType** to return a **Type**. An important note is that an error is thrown if a get for a type that does not match with **valueType** is made. Due to the semantic analysis, it is generally assumed that the types will match. Finally, the **getNumericVal** allows for operations between numbers without actually having to know if the number is a real or an integer. If the expected result is an integer, the result is then rounded up.

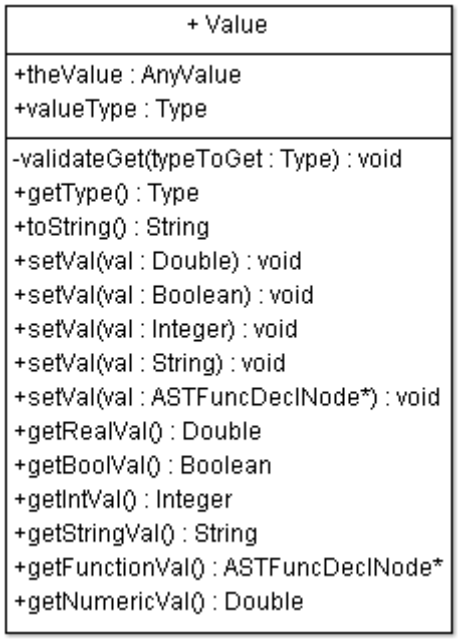

Figure 11 – Value Class

5.3 – The InterpreterExecVisitor Visitor Class

The new **InterpreterExecVisitor** visitor class (Figure 12) is a subclass of the class **Visitor**.

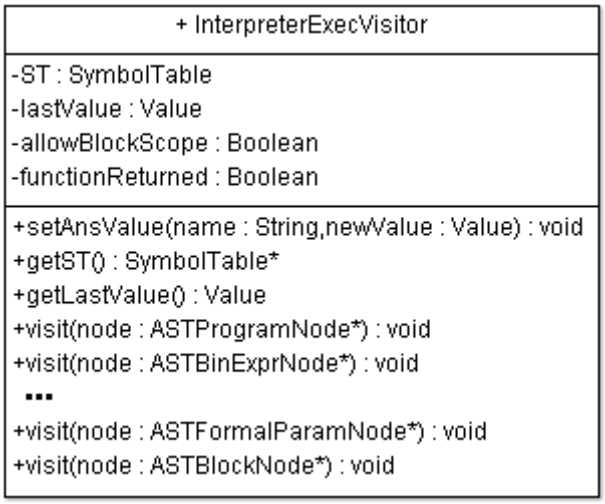

Figure 12 – InterpreterExecVisitor Class

Considering the private members, **ST** is an instance of the SymbolTable class that will be used throughout the class, **lastValue** is similar to **lastType** of the semantic analysis visitor class but stores the last value instead of the last type, **allowBlockScope** is identical to that of the semantic analysis visitor, and **functionReturned** is used to check if a function returned so that a traversal through a function body can terminate earlier. The three non-visit functions will only be used in the R.E.P.L and will be discussed in section 6.

5.3.1 – Some Important Notes

In this subsection, some additional points about the implementation will be presented:

- Overall, the visit functions are similar to the ones in the semantic analysis but do not perform explicit type and scope checks. Instead, they store the latest value after visiting expressions and manipulate the symbol table as needed, among other functionality;
- As stated in the previous section, some non-trivial forms of binary expressions are supported:
  - Addition between strings and any other value was made possible by the **toString** method in the Value class which returns a string representation of the value, irrespective of type.
  - Additive or multiplicative operations between an integer and a real produces a real result. This was made possible by the **getNumericVal** method in the Value class.
- For binary expressions, the C++ operation performed is based on the operator.
- The function node pointer stored in the symbol table makes it possible to perform function calls. When a function identifier is found, the list of formal parameters is traversed and the values of the actual parameters are assigned to the formal parameter identifiers. Next, the statements in the function's block are traversed and visited. After each visit, if **functionReturned** is true, the traversal stops. Since the return statement is the last visited statement, at such a point, **lastValue** holds the returned value.
- Print statements implicitly include a newline so that each print produces output on a separate line.

5.4 – End of Section

When control eventually returns to the initial visit call (i.e. of **ASTProgramNode**), this means that the execution of the program was successful. In the next section, the implementation of the R.E.P.L will be discussed.

Task 6 – The REPL

6.1 – Introduction

In this section, the implementation of the Read-Eval-Print Loop will be discussed. For this loop, a second main method was implemented wich, when executed, will immediately start the interactive mode. For each statement entered, the R.E.P.L will execute the lexer, parser, semantic analysis, and interpterer. The printing of the XML was not seen as important for the R.E.P.L since the output would have to change frequently.

6.2 – Pseudocode

Listing 10 below presents pseudocode of the implemented R.E.P.L. Note how an *infinite* loop encloses most code. This constitutes the loop aspect of the Read-Eval-Print Loop. In each iteration of the loop, the input is read and goes through some checks to detect if it is a `QUIT` input, `#st`, `#load` with a file, or simply a statement. These types of input will now be discussed.

```
ANS = 0
while (true) {
    Read INPUT;
    if (INPUT is "QUIT") {
        Exit the REPL;
    } else if (INPUT is "#st") {
        Print the symbol table;
    } else {
        if (INPUT is "#load \"<FILE>\"") {
            Set SOURCE to file <FILE>;
        } else {
            Append newline to INPUT;
            Set SOURCE to string INPUT;
        }
        try {
            Run lexer and parser;
            Perform semantic analysis;
            Perform interpreter execution pass;
            Save TREE instance;
            Set type and value of ANS;
        } catch LexerError {
            Print error message;
        } catch ParserError {
            Print error message;
        } catch SemAnalysisError {
            Print error message;
            Delete TREE;
        } catch InterpreterError {
            Print error message;
            Delete TREE;
        }
    }
}
```

Listing 10 – R.E.P.L Pseudocode

6.3.1 – "QUIT" and "#st" Inputs

First of all, a **"QUIT"** input allows the user to exit the R.E.P.L. Secondly, if the input matches with **"#st"**, then this means that the values of the symbol table will be printed. For this, the **getST** function from the InterpreterExecVisitor class (Figure 12) is used to obtain a pointer to the symbol table, for which the **printValues** function (mentioned in subsection 5.2) is called to display the values.

6.3.2 – Running the Lexer, Parser, Semantic Analysis, and Interpreter

6.3.2.1 – Two Types of Sources

If the input is neither "QUIT" nor "#st", this means that the source for the lexer, parser, and so on will be set. If the input is of the form **`#load "<FILE>"`**, then the source is set as a file stream, where <FILE> is assumed to indicate the path to a file containing MiniLang code. Otherwise, it is assumed that the input itself contains the statement or expression that will be evaluated, and so the source is set as the input itself. Note that in this case, a newline character is appended so that this can indicate the end of the line.

6.3.2.2 – Four Stages

From this point on, the lexer and parser are run, followed by the semantic analysis of the input and an interpreter execution pass. Note that these are contained within a *try* block. This is so that if an error occurs in any of the four stages, it can be handled. If the interpreter successfully executes the source, the tree instance is saved. Since a new AST will be created in each iteration of the R.E.P.L, it is important to save this tree or otherwise all associated AST nodes that will be used in the future will be destroyed. If a semantic error or interpreter error occur, the tree is deleted.

6.3.2.3 – The ANS Value

Note that after saving the tree, the type and value of a certain *ANS* value are set. Throughout the R.E.P.L, the *ANS* value stores the value of the last result computed during the interpreter execution pass. It is given an initial type and value of an integer zero. To obtain this value and its type, the **getLastValue** function (Figure 12) is used. The type and value are then passed to **setAnsType** (Figure 10) and **setAnsValue** (Figure 12) so that the type and value of this variable are registered in both the semantic analysis and interpreter symbol tables.

6.3.2.4 – Expressions as Statements

Unfortunately, the ability to input an expression as a statement, such as "24 + 12;" as shown in the assignment sheet, are not accepted by the R.E.P.L. This could have been implemented by deviating a bit from the given EBNF and introduce an "expression statement" having the form shown in Figure 13. Despite this missing feature, it was observed that similar functionality can be achieved by simply inserting the expression inside a print statement, for example: $print(24 + 12);$. This evaluates the expression and stores the result in ANS.

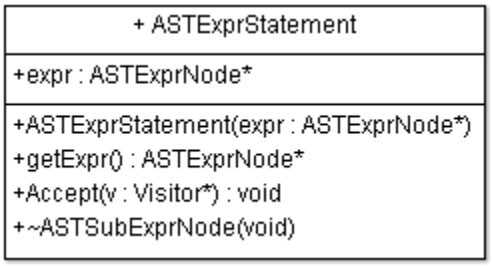

Figure 13 – ASTExprStatement

6.4 – End of Section

Two important points about this section will be pointed out. Firstly, `QUIT`, `#st`, and `#load` statements are not recognized when found inside a script that is obtained from a file and run by using the `#load` command.

Secondly, a bug was found where if an error occurs during the semantic analysis of a function body in a function declaration, the function identifier is still kept in the semantic analysis symbol table if it is in the global scope. If the function is eventually called, this causes a FatalSymbolTableError since the function's identifier is not found in the interpreter's symbol table. This could have been solved by checking for errors while going through a function declaration and removing the function's identifier from the symbol table if an error occurs. This is not the case with variable declarations since the identifier is inserted after the expression is evaluated.

Testing

7.1 – Introduction

In this section, tests will first be performed by giving files to the program and observing the outputs. Afterwards, the R.E.P.L will be used to perform further tests, including invalid inputs.

7.2 – Tests using an Input File

For the input file tests, a MiniLang source script was inserted into a file called **File1.prog** to test most aspects of the program. The following (Listing 11) is the contents of the file:

```
// Testing variable declaration
var a : real = 123.456;
var b : bool = true;
var c : int = 123456;
var d : string = "Text";

// Testing function declaration and formal parameters
def func(arg1 : int, arg2 : string) : bool {

    // Testing assignment
    set a = a / 2;          // Testing division operator with real
    set b = not b;          // Testing unary not
    set c = -c;             // Testing unary minus with integer
    set d = d + d;          // Testing string addition

    // Testing print
    print("a = " + a);      // Testing string addition with real
    print("b = " + b);      // Testing string addition with bool
    print("c = " + c);      // Testing string addition with int
    print("d = " + d);      // Testing string addition with string

    print("arg1 = " + arg1);
    print("arg2 = " + arg2);
    return (true and (false and true)) or true; // and, or, return, sub-expression
}

/* This is for the testing
 * of multi-line comments
 */
var i : int = 0;
while ( (i >= 0) and (i < 2) ) {    // Testing while loop and relational operators
    if ( i == 0 ) {                 // Testing if statement and equality operator
        print("i is 0");
    } else {                        // Testing else statement
        print("i is not 0");
    }
    var retVal : bool = func(i, (""+i)); // Testing function call and actual param
    print("Return value: " + retVal);
    print("");
    set i = i + 1;
}

// Testing block
{
    print("This is a print statement in a block.");
}

// Testing binary expressions and operator precedence
print("val1 = " + (1 > 2 + 3 - 4 * 5));
print("val2 = " + (1 * 2 - 3 + 4 > 5));
```

Listing 11 – File1.prog

Note that comments in the MiniLang program indicate the most important aspects being tested at each stage. Running the program with the **File1.prog** file as an input produces the following output (Listing 12):

```
Lexer and Parser were successful.
XML print was successful.
Semantic Analysis was successful.
Any output during execution is shown below...
-------------------------------------------
i is 0
a = 61.728000
b = false
c = -123456
d = TextText
arg1 = 0
arg2 = 0
Return value: true

i is not 0
a = 30.864000
b = true
c = 123456
d = TextTextTextText
arg1 = 1
arg2 = 1
Return value: true

This is a print statement in a block.
val1 = true
val2 = false
-------------------------------------------
Interpreter Execution Pass was successful.
```

Listing 12 – Output for File1.prog

This output indicates that, first of all, the lexer, parser, XML printer, semantic analysis, and interpreter execution pass were all successful. It also provides the input produced by *print* statements during execution.

The two main chunks of output correspond to the two iterations of the while loop. The first line of each of the two chunks shows that the *if* statement worked since the correct output was printed out. The second (**a**) shows that the division by 2 worked since the value is halved each time. The third (**b**) and fourth (**c**) lines of each of the two chunks show that the unary operations worked since the values return to their original. The fifth line (**d**) shows that the string concatenation worked. The sixth and seventh lines show that the arguments were successfully passed to the function. Finally, the return value matches with the expected *true* return value.

Finally, in the last two lines of the output, the operations resulted in true and false. These can be confirmed:

$$1 > 2 + 3 - 4 * 5 \;=\; 1 > \big(2 + \big(3 - (4 * 5)\big)\big) \;=\; 1 > \big(2 + (3 - 20)\big) \;= 1 > -15 \;= \boldsymbol{true}$$

$$1 * 2 - 3 + 4 > 5 \;=\; \Big(\big((1 * 2) - 3\big) + 4\Big) > 5 \;=\; \big((2 - 3) + 4\big) > 5 \;= 3 > 5 \;= \boldsymbol{false}$$

One can also appreciate that the fact that the program actually executed means that the lexer, parser, XML printer and semantic analysis were all successful, and that all keywords were recognized and type and scope checks were all successful. Since the resultant XML is a very long file, this will instead be included outside of this file as **File1XMLResult.xml**. In the next sub-section, tests will be performed using the R.E.P.L.

7.3 – Tests using R.E.P.L

In this section, tests will be performed using the R.E.P.L. In the cases where the testing targets errors, only normal errors will be tested. Non-normal (i.e. "Fatal") errors are errors that are expected to never occur.

7.3.1 – Lexer and Parser Errors

In this subsection, the inputs given to the R.E.P.L will be expected to generate specific errors. All of the possible lexer and parser errors will be tested. Consider the inputs and corresponding outputs presented in Listing 13.

```
MLi> var x : int = 123 ^ 456;
Lexer error :: syntax error at line 1.

MLi> var x : int : 123;
Parser error :: syntax error at line 1 due to unexpected token ":".

MLi> var x : int = 123+;
Parser error :: syntax error at line 1 due to invalid or missing factor.

MLi> var x : int = +123;
Parser error :: syntax error at line 1 due to invalid unary operator.

MLi> variable x : int = 123;
Parser error :: syntax error at line 1 due to invalid start of statement.
```

Listing 13 – Lexer and Parser Errors

The above five input-output pairs will be discussed below:

- The first input generated a lexer syntax error due to the caret '^' which in some languages is recognized as an exponentiation operator, but which is not supported by MiniLang.
- The second input generates a parser syntax error since in a variable declaration, the equals symbol is expected after the variable's type, so the statement should have been $var\ x : int = 123;$
- The third input generates a parser syntax error since the addition operator found after a valid expression is expected to form part of a binary expression. However, the right-hand-side is missing.
- The fourth input generates a parser syntax error since the addition operator cannot be used as a unary operator. The only supported unary operators are the minus '-' operator and the *not* operator.
- The final input generates a parser syntax error since "variable" is recognized as an identifier rather than the "var" variable declaration keyword, and no possible statement starts with an identifier.

7.3.2 – Semantic Analysis and Interpreter Errors

In this subsection, the inputs given to the R.E.P.L will be expected to generate semantic analysis errors. Semantic analysis errors are caused by the incorrect use of identifiers, functions, and types in general. Consider the inputs and corresponding outputs presented in Listing 14.

```
MLi> print(123 and true);
Semantic analysis error :: type mismatch; expected "bool" but found "int" in
"and" operation.

MLi> print(123 + true);
Semantic analysis error :: type mismatch; expected "numeric type" but found "Bool
or String" in "+" operation.

MLi> print(123 >= true);
Semantic analysis error :: type mismatch; expected "numeric type" but found "Bool
or String" in ">=" operation.
```

```
MLi> print(not 123);
Semantic analysis error :: type mismatch; expected "bool" but found "int" in
unary "not" operation.

MLi> print(-true);
Semantic analysis error :: type mismatch; expected "numeric type" but found
"bool" in unary "-" operation.
```

Listing 14 – Semantic Analysis Errors due to Simple Operations

Listing 14 presents five invalid operations. Each error includes the expected and found types and the situation in which the error occurred. This helps with finding the error since the line number is not given in semantic analysis errors. Consider the inputs and outputs presented in Listing 15.

```
MLi> def func1(arg : int) : int { return "123"; }
Semantic analysis error :: type mismatch; expected "int" but found "string" in
function return statement.

MLi> def func2(arg : int) : int { return 123; }
Variable ANS : real = 0.000000

MLi> print(func2(123, 456));
Semantic analysis error :: func2 given incorrect number of arguments.

MLi> print(func2("123"));
Semantic analysis error :: type mismatch; expected "int" but found "string" in
argument arg to function func2.
```

Listing 15 – Semantic Analysis Errors due to Functions

Listing 15 presents three invalid operations. In the first input, the function has an invalid return type. The second input defines a valid function to be used in the next two inputs. The first call to *func2* gives an incorrect number of arguments while the second gives an argument of an invalid type. Consider Listing 16.

```
MLi> var x1 : int = "123";
Semantic analysis error :: type mismatch; expected "int" but found "string" in
declaration of variable x1.

MLi> var x2 : int = 123;
Variable ANS : int = 123

MLi> set x2 = "123";
Semantic analysis error :: type mismatch; expected "int" but found "string" in
assignment of variable x2.
```

Listing 16 – Semantic Analysis Errors due to Variables

Listing 16 presents two invalid operations. In the first input, the variable is given an invalid initial value. The second input defines a valid variable to be used in the final input, which invalidly tries to assign a string to the variable of integer type. Consider Listing 17 for further tests.

```
MLi> if (123) { print("This should not work."); }
Semantic analysis error :: type mismatch; expected "bool" but found "int" in if
statement condition.

MLi> while (123) { print("This should not work."); }
Semantic analysis error :: type mismatch; expected "bool" but found "int" in
while loop condition.
```

Listing 17 – Semantic Analysis Errors due to Conditions

Listing 17 presents two cases; an if statement and while loop, where integers are used as the conditions. This of course causes a type mismatch since a condition is supposed to be a boolean. Consider Listing 18 overleaf.

```
MLi> var x1 : int = x2;
Semantic analysis error :: identifier "x2" does not exist.

MLi> var x1 : int = 123;
Variable ANS : int = 123

MLi> var x1 : int = 456;
Semantic analysis error :: duplicate declaration of "x1".

MLi> print(x1("arg"));
Semantic analysis error :: identifier x1 is not a function.

MLi> def func() : int { return 123; }
Variable ANS : int = 123

MLi> print(func);
Semantic analysis error :: function func is used as a variable.

MLi> return 123;
Semantic analysis error :: attempted to return outside of a function body.
```

Listing 18 – Semantic Analysis Errors due to Miscellaneous Cases

In Listing 18, a further five invalid inputs are presented. The first input tries to make use of an inexistent identifier. After `x1` is properly declared in the second input, the third input tries to re-declare variable `x1`, which causes an error. Next, `x1` is incorrectly used as a function; a variable cannot take arguments. Now, a function is correctly defined, but is then used incorrectly as a variable; a function call must use parentheses. Finally, a return statement is used outside of a function, which causes an error.

The final error to be discussed in this section is a function-related error that is missed during semantic analysis but detected during the interpreter execution pass when the function is called. Consider Listing 19 below.

```
MLi>def func() : int { }
Variable ANS : real = 0.000000

MLi>print(func());
Interpreter error :: function func did not return any value.
```

Listing 19 – Interpreter Execution Pass Error

In Listing 19, a function is first defined and the function declaration is accepted. However, one can observe that no return statement was provided in the function. In fact, when it is now used, an error indicates that the function did not return any value.

7.3.3 – Load, Symbol Table, and Quit Tests

Listing 20 below shows an example usage of the `#load` statement to run a set of statements stored in **File2.prog**. The symbol table is then output using `#st`, and the `QUIT` is used to exit the program.

```
MLi> #load "test/File2.prog"
Variable ANS : int = 123

MLi> #st
Variable ANS : int = 123
Function function(argument : real) : bool
Variable integer : int = 123

MLi> QUIT
Process finished with exit code 0
```

Listing 20 – Load, Symbol Table, and Quit Tests