CPS2000 - Compiler Theory and Practice - Minilang Assignment 2019

Aidan Cauchi

Faculty of ICT
University of Malta

11/05/2019

Contents

Chapter 1

Introduction

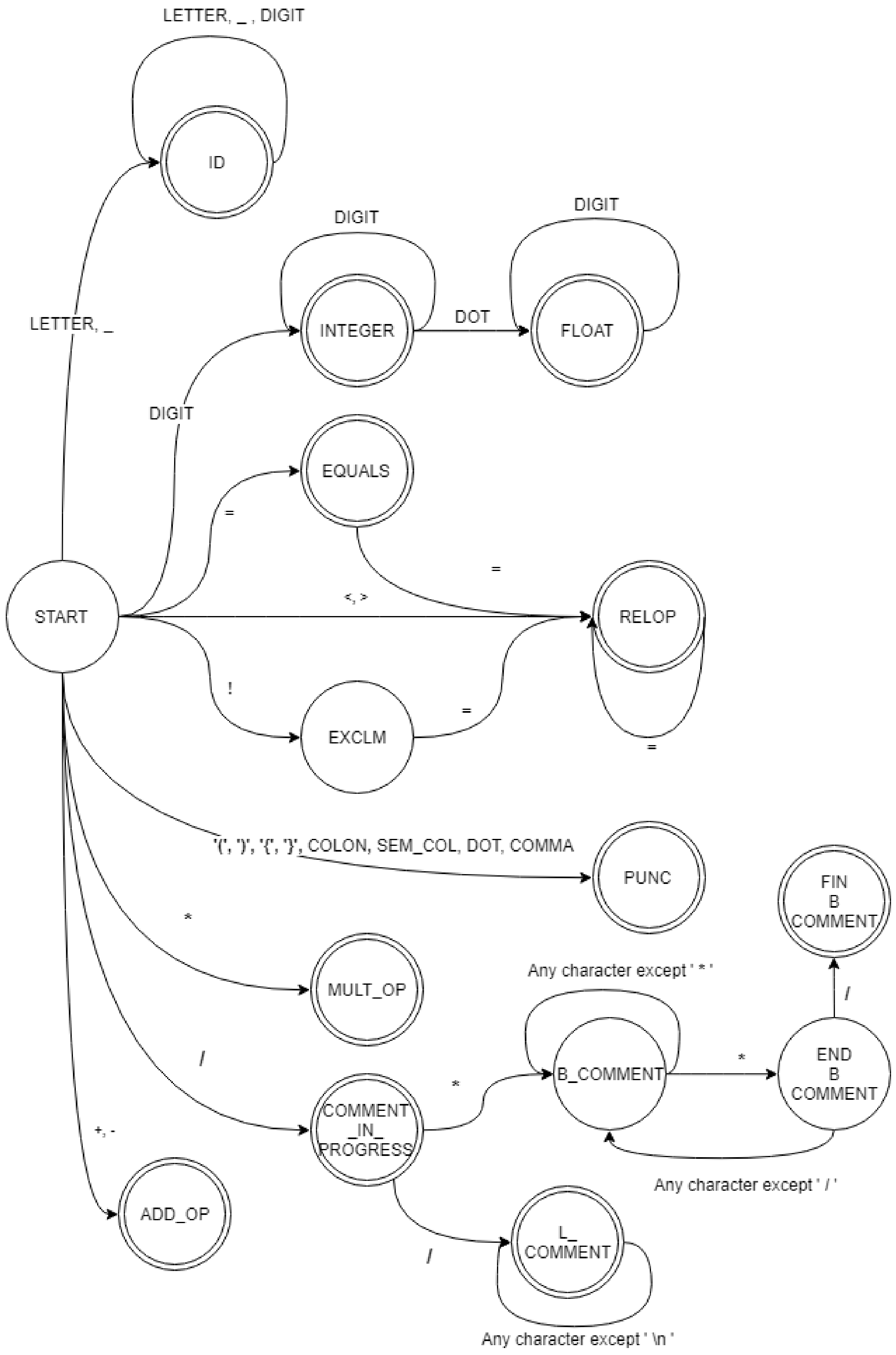
This report contains a high level description of Aidan Cauchi's solution to the Minilang assignment for CPS2000.

Chapter 2

Lexer

2.1 DFA

This next page contains the DFA that the table in the lexer represents. If a transition doesn't exist, it can be assumed that the next state would therefore be the ERROR state.


LETTER, _ , DIGIT
ID
DIGIT
DIGIT
INTEGER
DOT
FLOAT
LETTER, _
DIGIT
EQUALS
=
=
START
<, >
RELOP
!
=
EXCLM
=
'(', ')', '{', '}', COLON, SEM_COL, DOT, COMMA
PUNC
FIN B COMMENT
*
MULT_OP
Any character except ' * '
/
/
*
B_COMMENT
*
END B COMMENT
+, -
COMMENT _IN_ PROGRESS
Any character except ' / '
ADD_OP
/
L_ COMMENT
Any character except ' \n '

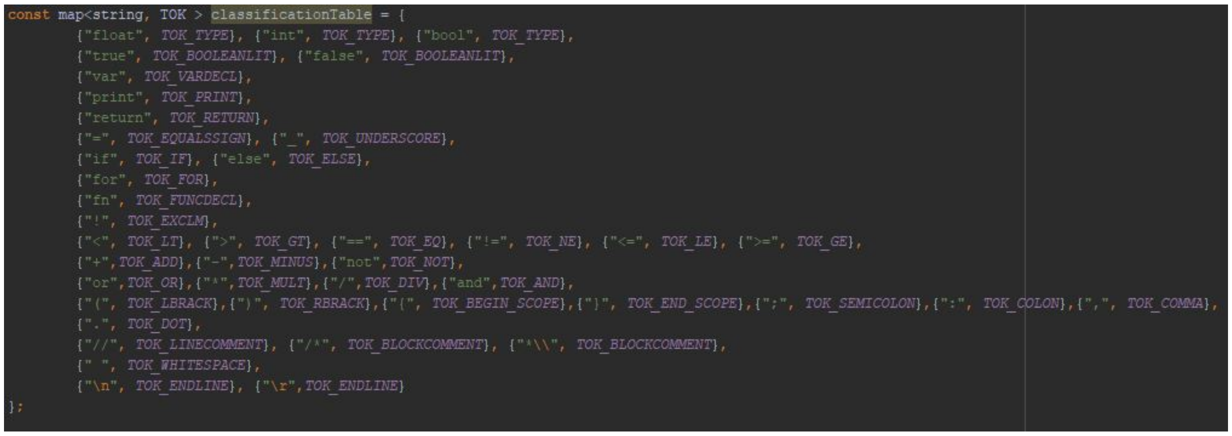
2.2 Overview

The lexer starts working after the *parseString()* method is called. In here, the lexer loops through the entire string (program), tokenising words which follow the pattern described by the DFA. All valid tokens are then put into a vector of tokens representing the entire tokenised program.

The way the lexer recognizes tokens is through the *nextWord()* method. The method implements the algorithm found in the class notes. The first part of the algorithm goes through the program character by character, identifies what character that is, and does the necessary transition from the current state to the next one while simultaneously keeping track of the entire string so far(lexeme). The states traversed are put on a stack until an ERROR state is found.

Once an ERROR state is encountered, the program then enters the roll back loop, where it keeps popping out non-final states while truncating the lexeme until a final state is encountered. Once this happens, the final state is checked to see what token it represents and a token is thus generated.

In an effort to reduce the overall size of the table describing the DFA for the sake of maintainability, certain measures were taken. Notice how the DFA does not represent keywords. If this were the case the number of states needed would be much bigger. So to prevent this, the lexer treats a keyword as an **Identifier**. After going through the process above, a keyword is then checked against the **Classification Table** containing the keywords and their TOK counterpart (TOK is the enum to identify token types). If a match is found, the lexeme becomes identified as a keyword. Otherwise it is treated as an **Identifier**. This process is almost replicated with other tokens such as Punctuation, Relation Operators, Additive Operators etc, where the result from the DFA is input again in the **Classification Table** to get the absolute final token. The table can be seen by Figure 2.1.



```
const map<string, TOK > classificationTable = {
        {"float", TOK_TYPE}, {"int", TOK_TYPE}, {"bool", TOK_TYPE},
        {"true", TOK_BOOLEANLIT}, {"false", TOK_BOOLEANLIT},
        {"var", TOK_VARDECL},
        {"print", TOK_PRINT},
        {"return", TOK_RETURN},
        {"=", TOK_EQUALSSIGN}, {"_", TOK_UNDERSCORE},
        {"if", TOK_IF}, {"else", TOK_ELSE},
        {"for", TOK_FOR},
        {"fn", TOK_FUNCDECL},
        {"!", TOK_EXCLM},
        {"<", TOK_LT}, {">", TOK_GT}, {"==", TOK_EQ}, {"!=", TOK_NE}, {"<=", TOK_LE}, {">=", TOK_GE},
        {"+",TOK_ADD},{"-",TOK_MINUS},{"not",TOK_NOT},
        {"or",TOK_OR},{"*",TOK_MULT},{"/",TOK_DIV},{"and",TOK_AND},
        {"(", TOK_LBRACK},{")", TOK_RBRACK},{"{", TOK_BEGIN_SCOPE},{"}", TOK_END_SCOPE},{";", TOK_SEMICOLON},{":", TOK_COLON},{",", TOK_COMMA},
        {".", TOK_DOT},
        {"//", TOK_LINECOMMENT}, {"/*", TOK_BLOCKCOMMENT}, {"*\\", TOK_BLOCKCOMMENT},
        {" ", TOK_WHITESPACE},
        {"\n", TOK_ENDLINE}, {"\r",TOK_ENDLINE}
};
```

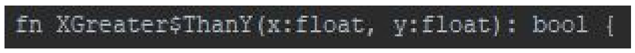
Figure 2.1: The Map representing the Classification Table

2.3 Tokens

Tokens in the lexer take the form of **(<Token type >, <Value >, <Line number >)**. Furthermore, almost every terminal symbol in the lexer has its own token type. It is noted that this is may not the most efficient way to store tokens as it leads to a lot of useless information. Example: An *if* token will be stored as **(TOK_IF , if , 1)**. One other way this could have been done is to store all keywords as TOK_KEYWORD and then use the value to denote what kind of keyword it is. Example: **(TOK_KEYWORD , 1 , 1)** where 1 denotes an IF statement, a 2 denotes a FOR statement etc. Another important thing to note is that the lexer only tokenises terminal symbols. The parser then groups the tokens into higher non-terminal symbols when doing its top-down check. Finally the line number is used for the parser to be able to report exactly where any errors are.

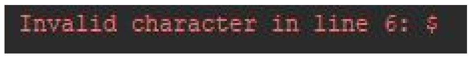
2.4 Error Handling

The Lexer is able to detect unknown characters and report them to the user. If the character is not able to be indentified from the Classification Table a **TOK_LEXERROR** is generated which is then handled by the *reportLexerError()* method.



```
fn XGreater$ThanY(x:float, y:float): bool {
```

Figure 2.2: Function declaration with invalid character



```
Invalid character in line 6: $
```

Figure 2.3: Error of invalid character

2.5 Limitations

It is important to note that the DFA is not the most optimal for the language. Certain errors in the DFA add the need of extra code to handle these exceptions (as is the case with block comments which can be seen in the first part of *nextWord()*. This in turn leads to less maintainable code and to bad programming practices in general. This can mostly be blamed due to poor planning and lack of knowledge while implementing the lexer. Furthermore, certain lexical errors are shown when outputting the tokens. In reality, tokens won't be shown to the user so errors need to be detected in another manner.

Chapter 3

Parser

3.1 Overview

The Parser communicates with the Lexer through the *getNextToken()* method whose function is self-explanatory. Furthermore, the Parser takes a top-down predictive approach to recognize the syntax of the program.

A Parse run starts with the *parse()* method in the Parser. In order to choose what to do next, the Parser closely follows the rules in the assignment specification. Therefore, it first assumes a 'program'. Since a program is made up of multiple 'statement' expressions, the Parser keeps calling the *parseStatement()* method until a TOK_EOF is found. Inside the *parseStatement()*, the Parser decides what type of statement to parse depending on the next token. For example, if the next token is TOK_VARDECL then the *parseVarDeclaration()* method is called. Else if it is a TOK_IF, then the *parseIfStatement()* method is called and so on.

In order to promote the reuse of code, the same methods can be used in different contexts. Let's take the two methods mentioned above to help with the explanation. The *parseVarDeclaration()* calls the *parseExpression()* to understand what to assign to the newly created variable. On the otherhand, the *parseIfStatement()* calls the *parseExpression()* to get the details of what the condition consists of. The process of reusing such methods follows closely the specification of the language. All the methods follow the same principle outlined in this paragraph so there is no need to go over every method here.

Another imporant note is that the Parser may also communicate with the Lexer through the *peekNext-Token()* which returns the next token without removing it from the front of the vector. This allows the Parser to take decisions where certain elements don't necessarily have to be there. The *parseIfStatement()* makes use of this method in order to check whether to parse an 'else' statement or not. Thus it can decide what to do without consuming the next token which makes it simpler and simple is nice.

3.2 Abstract Syntax Tree

Every method in the Parser returns an ASTNode that is added in the final Abstract Syntax Tree describing the program. The ASTfiles folder in the directory contains all the types of nodes which can be found in the AST. Through the use of polymorphism, certain ASTNodes could be logically grouped together to organise code better. This can be seen by the *ASTStatementNode* and the *ASTExpressionNode*. These nodes do not get actually represented in the AST but their use is to group other ASTNodes to allow polymorphism to occur and reduce errors. A successful representation of a full AST can be seen by Figure 4.2 which is handled by the XMLVisitor class in the next Chapter. Figure 3.2 shows the AST Equivalent of a program containing the function listed in Figure 3.1.

```
fn XGreaterThanY(x:float, y:float): bool {
    var ans:bool = true;
    if (y>x) { ans=false; }
    return ans;
}
```

Figure 3.1: Function to represent

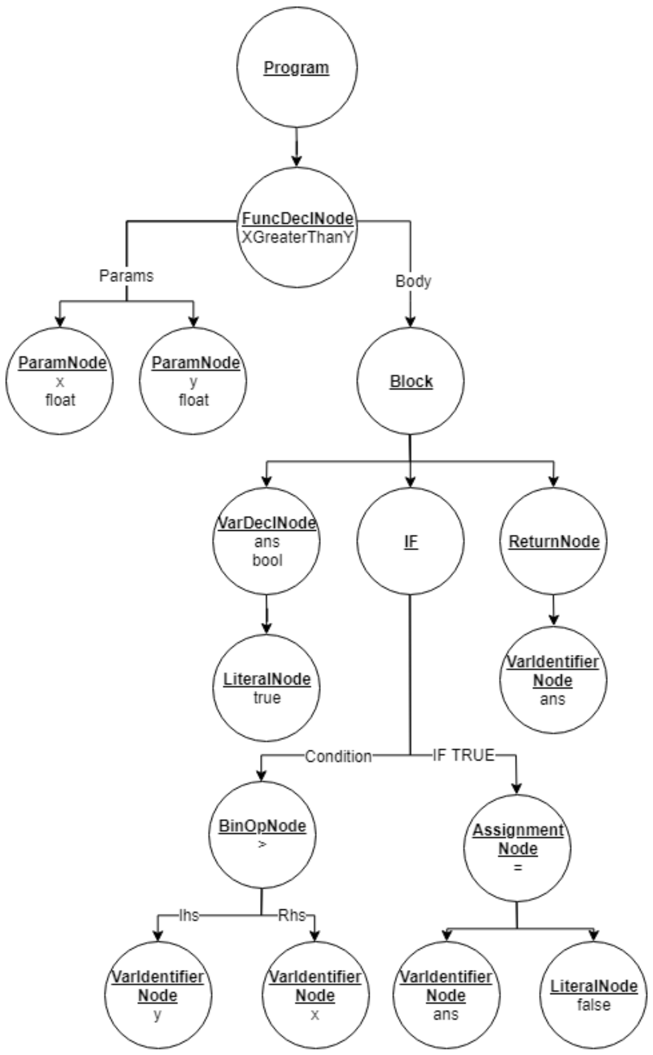
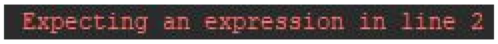


Figure 3.2: AST Of Function XGreaterThanY

3.3 Error Handling

The *Error()* method is called when the Parser encounters a Token that it is not expecting. It outputs the type of error and exits the program. As a result of this design choice, implementation is simpler but this means that it can only detect one error at a time. The user will have to fix multiple errors one by one.

If the code in Figure 3.1 is modified such that *var ans:bool = ;*, it can be seen by Figure 3.3 that the Parser does indeed complain.



```
Expecting an expression in line 2
```

Figure 3.3: Parse Error expecting an Expression to perform variable assignment

Chapter 4

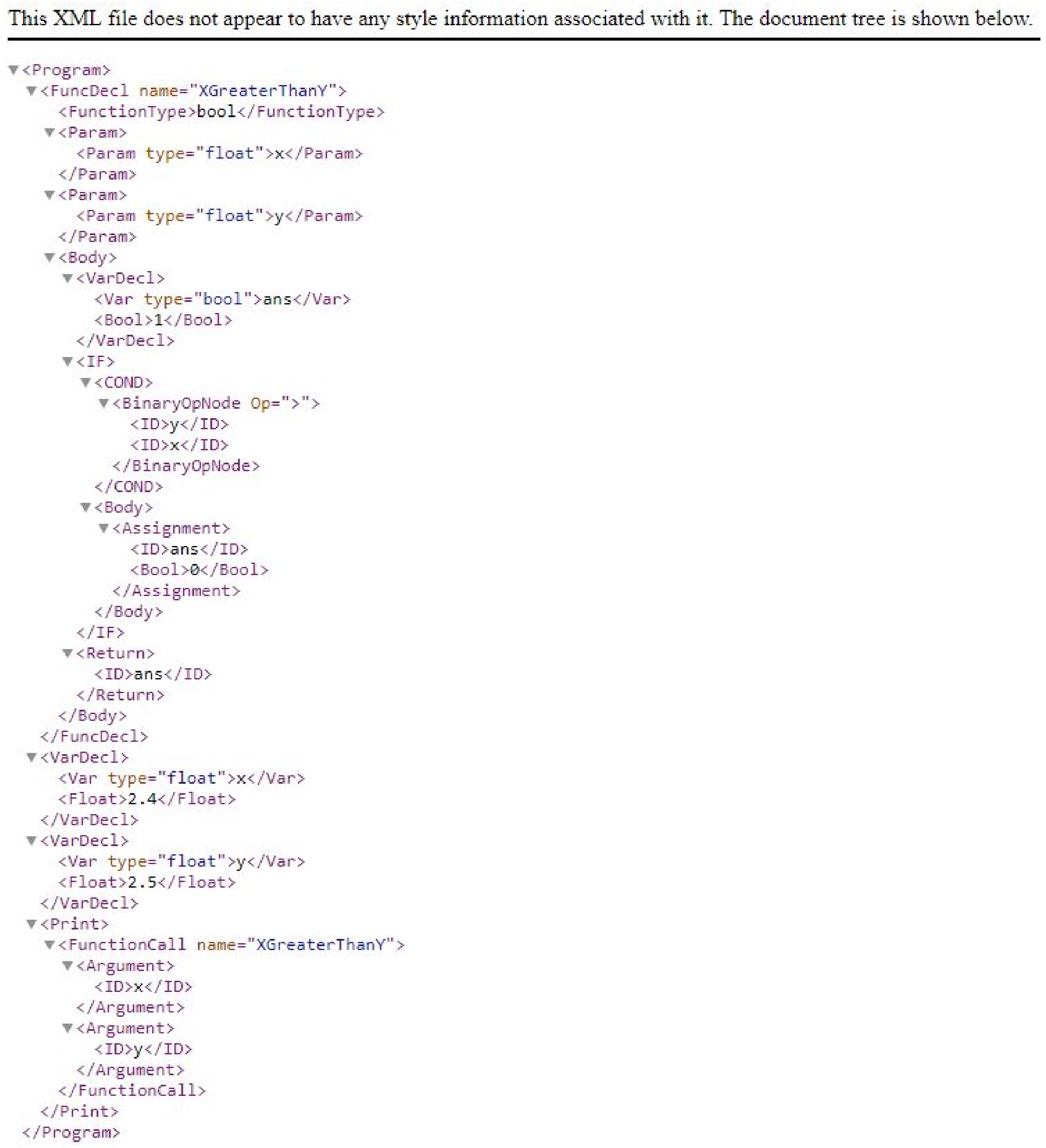
XML Generation

4.1 Overview

The XML generation phase is carried out by the XMLVisitor class which implements the Visitor design pattern. As a result of this design pattern, the *visit()* method in the XMLVisitor class makes use of **method overloading** to output the XML depending on the type of ASTNode supplied. Furthermore, every ASTNode has an *Accept()* method which takes any type of Visitor to do something on the node. Lastly, the XMLVisitor keeps track of the number of indentations required to output a properly indented XML representation of the input program. Figure 4.1 is the test program input to the compiler while Figure 4.2 is the resulting AST displayed in a web browser to show that it is a complete XML file.

```
fn XGreaterThanY(x:float, y:float): bool {
    var ans:bool = true;
    if (y>x) { ans=false; }
    return ans;
}

var x:float = 2.4;
var y:float = 2.5;
print XGreaterThanY(x, y);
```

Figure 4.1: Short input program


This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<Program>
  ▼<FuncDecl name="XGreaterThanY">
     <FunctionType>bool</FunctionType>
    ▼<Param>
       <Param type="float">x</Param>
     </Param>
    ▼<Param>
       <Param type="float">y</Param>
     </Param>
    ▼<Body>
      ▼<VarDecl>
         <Var type="bool">ans</Var>
         <Bool>1</Bool>
       </VarDecl>
      ▼<IF>
        ▼<COND>
          ▼<BinaryOpNode Op=">">
             <ID>y</ID>
             <ID>x</ID>
           </BinaryOpNode>
         </COND>
        ▼<Body>
          ▼<Assignment>
             <ID>ans</ID>
             <Bool>0</Bool>
           </Assignment>
         </Body>
       </IF>
      ▼<Return>
         <ID>ans</ID>
       </Return>
     </Body>
   </FuncDecl>
  ▼<VarDecl>
     <Var type="float">x</Var>
     <Float>2.4</Float>
   </VarDecl>
  ▼<VarDecl>
     <Var type="float">y</Var>
     <Float>2.5</Float>
   </VarDecl>
  ▼<Print>
    ▼<FunctionCall name="XGreaterThanY">
      ▼<Argument>
         <ID>x</ID>
       </Argument>
      ▼<Argument>
         <ID>y</ID>
       </Argument>
     </FunctionCall>
   </Print>
 </Program>
```

Figure 4.2: XML Representation of the AST

Chapter 5

Semantic Analysis

5.1 Overview

Semantic analysis is concerned with the semantics of the program, as is suggested by its name. As a result, actions in this phase mostly consist of ensuring that variables are not declared multiple times in the same scope, and that variables are assigned the correct data types to ensure consistency and prevent errors.

Semantic Analysis is handled by the SemanticAnalysisVisitor class. As with all visitor design pattern approaches, this class has a method that handles any type of node in the AST.

5.2 Scopes

The SemanticAnalysisVisitor handles the scopes by using a vector (acting like a stack) of maps. Each map takes a string as the name/key and returns a *tableEntry*. A *tableEntry* is just a struct which contains the type of the variable/function, an enum which denotes whether it is a function or a variable and a pointer to a vector of ASTParameterNodes if it is a function.

Each map in the stack denotes a new scope. Thus, the top most scope is the current scope the program is in. Keeping this in mind, the SemanticAnalysisVisitor class has a number of functions which are used to carry out the analysis:

- STPush(): Pushes a new empty map/scope on the stack.
- STInsert(name, dType, entryType, params): Enters a new variable/function in the current scope.
- STLookup(name): Returns the tableEntry of a variable/function if it exists, nullptr otherwise. This checks all the scopes in the stack.
- STPop(): Pops and deletes the current scope from the top of stack.

5.3 Type Checking

Since all the Visitor *visit()* methods have a void return type, there needed to be some way for the methods to communicate with each other without returning any values. As a result, the use of a stack was favoured. Certain methods would push their type on the stack while other methods would pop to perform type checking. An example of this can be seen with variable declaration: The *visit(ASTVarDeclNode)* is called and the type of the variable is saved. Then the respective method for the result is called say *visit(ASTLiteralNode)*. This method would just get the type of the literal, and push it on the stack. Once the function returns, all the *visit(ASTVarDeclNode)* has to do is simply pop from the stack and compare with the saved type.

To perform type checking, the class makes use of the following methods:

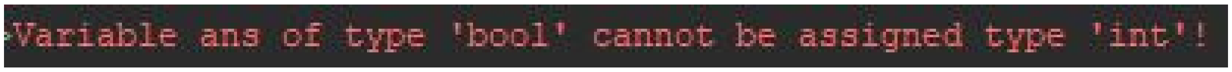
- dtStackAdd(dataType): Pushes a data type on the stack.
- dtStackPop(): Pops a data type from the stack for type checking.

5.4 Semantic Analysis in Action

In order to show it working, the following program in Figure 5.1 will be given as input, which has 3 semantic errors

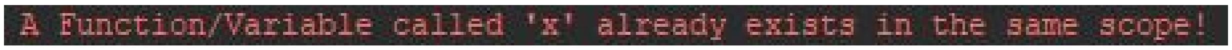
```
fn XGreaterThanY(x:float, y:float): bool {
    var ans:bool = 34; //Semantic error
    if (y>x) { ans=false; }
    return ans;
}

var x:float = 2.4;
var x:float = 2.5; //Semantic error
print XGreaterY(x, y); //Semantic error
```

Figure 5.1: Short input program to test semantic analysis



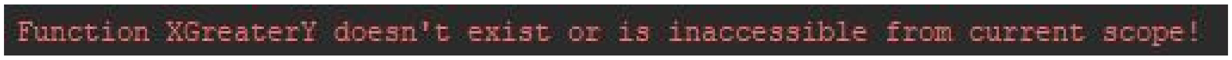
```
Variable ans of type 'bool' cannot be assigned type 'int'!
```

Figure 5.2: Semantic Error 1



```
A Function/Variable called 'x' already exists in the same scope!
```

Figure 5.3: Semantic Error 2



```
Function XGreaterY doesn't exist or is inaccessible from current scope!
```

Figure 5.4: Semantic Error 3

5.5 Limitations and Things To Note

The biggest known limitation is the fact that the errors don't show the line numbers due to them not being stored in the ASTNodes. This could have easily been avoided with proper planning but due to time constraints, it was decided to be left as is. Another thing to note is that functions and variables cannot have the same name. Another limitation is that one function must and needs to only have one return statement, otherwise data types on the stack won't line up and errors will be caused. An interesting thing to note is the fact that the specification of the language does not technically forbid function declarations inside other functions, and neither does this implementation.

Chapter 6

Interpreter

6.1 Overview

This section of the report highlights the actual execution of code written for the Minilang compiler.

The InterpreterVisitor works almost identically to the SemanticAnalysisVisitor. It makes use of a stack of maps denoting the scopes, and a stack where the result of a calculation is stored for use by other functions. So instead of comparing types, the actual values are now used without any need for type checking.

The *tableEntry* is augmented with more variables, namely the current value of a variable and a new pointer to the body of a function, which is used whenever a function is called. This ensures that on a function call the correct scopes get created and the function is able to run multiple times.

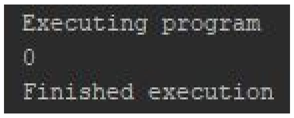
An important detail to note is that under the hood, everything is represented by a float, since the only types that the language can handle are integer, float and boolean. As for the boolean type, the C style approach is taken where **0** denotes a boolean **false** while anything else denotes a boolean **true**.

6.2 Examples

This section contains some examples that were input to the compiler and their results.

```
fn XGreaterThanY(x:float, y:float): bool {
    var ans:bool = true;
    if (y>x) { ans=false; }
    return ans;
}


var x:float = 2.4;
var y:float = 2.5;
print XGreaterThanY(x, y); //false
```

Figure 6.1: Input program to test Interpreter

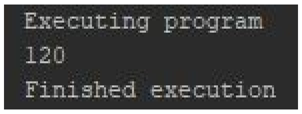


```
Executing program
0
Finished execution
```

Figure 6.2: Result for program 1

```
fn factorial(x:int):int {
    var tempVar:int = x;
    if (x!=1) {
        tempVar = factorial(x-1);
    }
    return x*tempVar;
}

var x:int = 5;
print factorial(x); //120
```

Figure 6.3: Input program 2 to test Interpreter

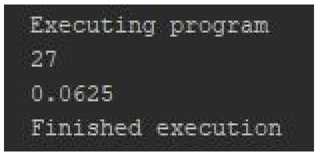


```
Executing program
120
Finished execution
```

Figure 6.4: Result for program 2

```
//Function definition for Power
fn Pow(x:float, n:int) : float
{
    var y : float = 1.0;
    if(n>0)
    {
        for(; n>0 ; n=n-1)
        {
            y = y * x;
        }
    }
    else
    {
        for(; n<0; n=n+1)
        {
            y = y / x;
        }
    }
    return y;
}

print Pow(3.0,3); //27
print Pow(4.0,-2); //0.0625
```

Figure 6.5: Input program 3 to test Interpreter



```
Executing program
27
0.0625
Finished execution
```

Figure 6.6: Result for program 3

Chapter 7

Conclusion

This assignment highlighted the many challenges and obstacles present in designing a compiler. Since all the sections build on top one another, it also highlighted how important careful planning is. If a mistake was made early in the design of the Lexer and found later on, it might require the programmer to change the behaviour of other components entirely.

The Lexer showed how a table driven approach can be used to build it. Furthermore, the Parser highlighted how building the AST correctly is extremely important especially later on, as everything else after the parser depends on it. The sections afterwards showed how useful the Visitor Design pattern is when dealing with such a structure and also how convenient it is, making it possible to separate code depending on the type of AST Node being processed.

To conclude, the assignment was quite challenging but also very rewarding.