L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Computer Information Systems

Compiler Theory and Practice - TinyLang

Edward Thomas Sciberras (274402L)
B.Sc. (Hons) Information Technology (AI)

Study-unit: **Compiler Theory and Practice**
Code: **CPS2000**
Supervisor: **Mr Sandro Spina**

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as “the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

We, the undersigned, declare that the assignment submitted is my / our* work, except where acknowledged and referenced.

We understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected and will be given zero marks.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Edward Thomas Sciberras 

Student Name Signature

CPS2000 Compiler Theory and Practice - TinyLang

Course Code Title of work submitted

18/06/2022

Date

Contents

Task 1 – Lexer

The job of a table-driven lexer is to tokenise the program that is being inputted to it and check and report any lexical errors. The Deterministic Finite Automaton (DFA) represents the EBNF given for the TinyLang programming language. The DFA could be and has been translated into a classifier table, which in this case would be a 2D array in the code. Enums were created for the token types which were split into twenty-nine categories and for each state (0 – 18 and error state). Enums were used to facilitate the usage of words instead of meaningless numbers that can get confusing easily. In addition, a class full of classifiers with an associated number to them was developed to compare items with more ease throughout the development. These helper classes can be found in the appendix.

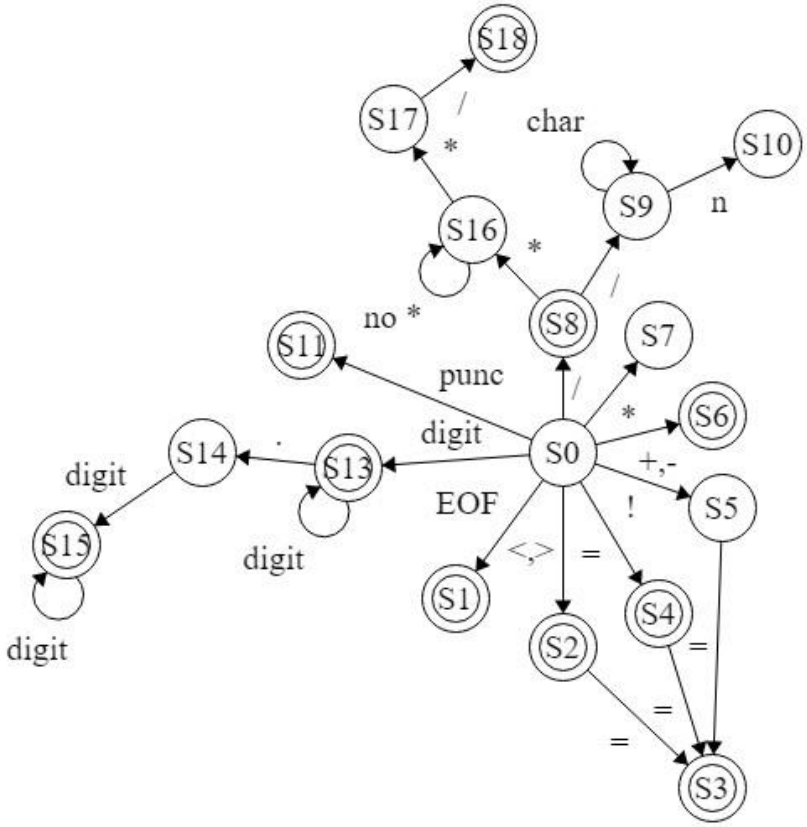

Figure 1 - Representation of DFA used in graph

The lexer scans through every character of the inputted program until a token that is in the token type list is found. The method for getting a token is called upon by the parser to keep on getting a token until an error is found and reported or the end of the file has been reached. The lexer uses a stack to hold the data of all the states in the program that have been visited. Once the lexer starts scanning each character of the program, it will inevitably

reach a stage when an error state is encountered. When this scenario is reached, the algorithm will go back into a rollback loop where the lexer essentially, removes states from the stack of states until a final state and a non-error stage is the current state. Once this is accomplished, then a valid state should be reached, however, the program will then determine if it a final state or not. If the state is a final state, then a valid token will be returned to the parser that is calling it. It is important to note that the lexer will ignore all comments, being single line comments or multi line comments.

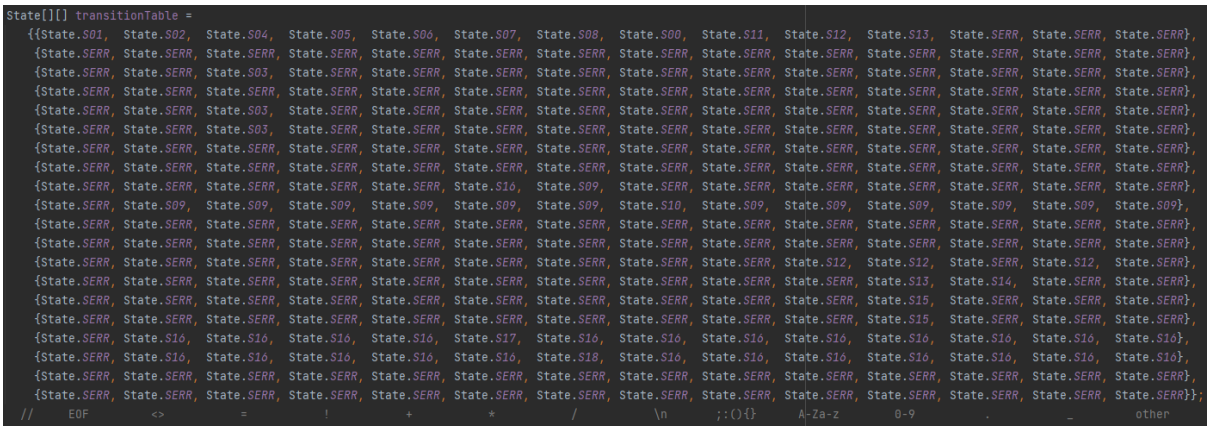

Figure 2 - Representation of transition table as 2D array

Testing the Lexer

A method was also developed, mostly for testing purposes which would see the return of all tokens into one array list, that can then be printed out to check what the lexer is returning and take a closer look at how it works. The output of this method will be shown with the program that was inputted below.

Inputted Program	Output of method

// this is a comment
fn XGreaterY_2(x:float, y:float) -> bool {
return x > y;
}
	TOK_Function
	TOK_Identifier
	TOK_LeftBracket
	TOK_Identifier
	TOK_Colon
	TOK_FloatKeyword
	TOK_Comma
	TOK_Identifier
	TOK_Colon
	TOK_FloatKeyword
	TOK_RightBracket
	TOK_Addition
	TOK_Relation
	TOK_BooleanKeyword
	TOK_LeftCurly
	TOK_Return
	TOK_Identifier

	TOK_Relation TOK_Identifier TOK_Semicolon TOK_RightCurly TOK_EOF

// this is a comment
fn XGreaterY_2(x:float, y:float) -> bool {
return x > y;
 	
(no ending bracket)
	TOK_Function
	TOK_Identifier
	TOK_LeftBracket
	TOK_Identifier
	TOK_Colon
	TOK_FloatKeyword
	TOK_Comma
	TOK_Identifier
	TOK_Colon
	TOK_FloatKeyword
	TOK_RightBracket
	TOK_Addition
	TOK_Relation
	TOK_BooleanKeyword
	TOK_LeftCurly
	TOK_Return
	TOK_Identifier
	TOK_Relation
	TOK_Identifier
	TOK_Semicolon
	TOK_EOF
	Given statement token not recoginised

// this is still a comment
fn XGreaterY_2(x:float, y[float) -> bool {
return x > y;
}
 	
(invalid character)
	Error at line: 2

Task 2 – Parser

An AST tree of the TinyLang program was created using a top-down recursive parser. The parser class keeps a lexer object and uses the method to obtain the next token to acquire the next valid token from it. Until an End-of-File (EOF) token is detected, the parser obtains the next token. Since a program is made up of 0 or more statements, if an EOF token is not found, it anticipates a statement and runs the function to retrieve the next statement. The statements that are received will then be added to an arraylist of statements. Finally, the parser produces an ASTProgramNode that has an arraylist of statement objects that describe the code's structure.

A statement node could have been expanded upon into eight different classes or objects. Thus, a different class for each was made. For instance, a 'for' statement would hold the data for the expression, the if block and the else block if there is one present. The following list includes all the statements that were included from the EBNF:

- Assignment
- Block
- For
- Function Declaration
- If
- Print
- Return
- Variable Declaration

An expression is either a factor or a literal, thus a class was made for each of them. Factors could be split up into five sub-factors, these being literals, identifiers, function calls, sub-expressions and unary. The literals can then be further divided into boolean, integer and float literals.

Using functions, polymorphism and inheritance, functions were created for each type of possible statement or expression in the EBNF. This was done so that the whole parsing could be performed using recursion, which makes the code more elegant and easier to read.

An example of how the parser works will start of with the lexer and if the next token received returns the enum that represents an 'if' statement, then the function for parsing 'if' statements is called upon.

```
private ASTStatementNode parseIfStatement(){
    currToken = lexer.getNextToken();
    if(currToken.type != TokenType.TOK_LeftBracket){
        System.out.println("Left bracket expected after 'if' in if statement");
        System.exit( status: 1);
    }

    ASTExpressionNode expressionNode = getExpression();

    currToken = lexer.getNextToken();
    if(currToken.type != TokenType.TOK_RightBracket){
        System.out.println("Right bracket expected after 'if' in if statement");
        System.exit( status: 1);
    }

    ASTStatementNode ifBlock = parseBlock();
    if(lexer.checkNextToken() == TokenType.TOK_Else){
        currToken = lexer.getNextToken();
        ASTStatementNode elseBlock = parseBlock();
        return new ASTStatementIf(expressionNode, ifBlock, elseBlock);
    }
    return new ASTStatementIf(expressionNode, ifBlock);
}
```

Figure 3 - Code to parse an 'if' statement

As seen in the code above, the algorithm will get the next token and check if it is a left bracket immediately as it was the ‘if’ statement represents in the EBNF.

$$\langle IfStatement \rangle \quad ::= \text{‘if’ ‘(’ } \langle Expression \rangle \text{ ‘)’ } \langle Block \rangle \text{ [‘else’ } \langle Block \rangle \text{]}$$

Figure 4 - EBNF of 'if' statement

Then the next token would have to be the expression that would be tested in the ‘if’ statement, and once that part is complete and no errors were found it would check that the next token is a right bracket. Once again, if no errors were found in the previous pieces of code, then the block statement that will be executed if the expression is found to be true would be parsed. Moreover, if there is an else token found it would also be added to the class of ‘if’ statements. In this case, the class for ‘if’ statements has two constructors, one with an ‘else’ statement and one without. This is to accommodate both instances of the statements.

```
public ASTStatementIf(ASTExpressionNode expression, ASTStatementNode ifBlock){
    this.expression = expression;
    this.ifBlock = ifBlock;
    this.elseBlock = null;
}

public ASTStatementIf(ASTExpressionNode expression, ASTStatementNode ifBlock, ASTStatementNode elseBlock){
    this.expression = expression;
    this.ifBlock = ifBlock;
    this.elseBlock = elseBlock;
```

Figure 5 - Constructors for the 'if' statement

Testing the Parser

Inputted Program	Output

// this is a comment
fn XGreaterY_2(x:float, y:float) -> bool {
return x > y;
}
	No error

fn XGreaterY(x:float, y:float) -> bool {
// error
var x:z = 0;
}
	Equals expected after identifier in assignment.

Task 3 – XML Generator

A visitor class is a subclass which holds visit functions for each node type so that the AST tree can be printed and transformed into an XML file. The XML file will represent the structure of the code as well as the contents of the variables and expressions inside it. The traversal starts from the top-most 'program' node which has the contents of the whole program and was obtained from the parser. Each statement in the program node will be looped and gone through using the 'accept()' function that is called and the specific function to the node that called it will be printed into the contents of the XML file. If there is another node in a specific statement, then that node will call upon its own method that will be finished and then returned to the parent node. This way the whole program will be done and written to the file.

```
@Override
public void visit(ASTStatementVarDeclare node) {
    writer.println(getIndents() + "<VariableDeclare>");
    indent++;
    writer.println(getIndents() + "<Variable Type='" + node.type + "'>" + node.identifier + "</Variable>");
    writer.println(getIndents() + "<Expression>");
    indent++;
    node.expression.accept( visitor: this);
    indent--;
    writer.println(getIndents() + "</Expression>");
    indent--;
    writer.println(getIndents() + "</VariableDeclare>");
}
```

Figure 6 - Writing XML of a variable declaration.

Testing the XML Generator

Inputted Program	Output

fn Sq(x:float) -> float {
return x*x;
}
	

fn XGreaterY(x:float , y:float) -> bool {
let ans:bool = true;
if (y > x) { ans = false; }
return ans;
}
	

Task 4 – Semantic Analysis

The semantic analyser class is also a visitor class that was implemented to travel through the tree similarly to how the XML performs the traversal also. However, instead of being converted to XML, this visitor class will be checking for semantic errors.

A symbol table class was developed to store the declarations of functions and variables in each scope that there is in the code of the inputted program. Each symbol in the program holds the information of its identifier, the declaration type to see if it was a function or a variable, and the literal type. Each symbol also has a data field for parameter types for each function declaration. This data is stored in an arraylist of strings. A hashing function was also implemented to find and add symbols using their identifier easily throughout the application.

```
public Symbol search(String identifier){
    index = hashFunction(identifier);
    Symbol start = head[index];

    if(start == null){
        return null;
    }

    while(start != null){
        if(start.identifier.equals(identifier)){
            return start;
        }
        start = start.next;
    }
    return null;
}
```

Figure 7 - Code of hashing function to find a symbol using its identifier.

The task of the semantic analyser is to let the developer know that there are mismatching types, declarations of the same variable or method call on methods or variables that have not been declared yet and for allowing uses within different scopes. An arraylist of symbol tables is used to keep track of all the declarations that were made in each scope to see which declarations can be used in which scope. Every time a block statement is found, then a new symbol table is added to the list. When a block statement is then done from the parser it is then removed from the arraylist.

A variable called ‘currLiteralType’ was created to represent the type of declaration of a function to ensure that the return statement will also return the same type. It will be updated using recursion so when a new declaration is being checked the values will update accordingly, and once it is done with the new declaration it will be set to the previous one again. In the case that there is a difference between types, then the user will be notified of a semantic error.

In addition, a variable called ‘binaryCheck’ was used as a boolean value to check that in certain statements like ‘if’ the types used would be the correct ones. In the cases that this flag is true, then another variable ‘literalType’ would be set to the literal type of the first expression and would ensure that all other expressions match. The variable will then be set to false and the literal type variable will be emptied so that they can be used again on different occasions throughout the program.

Unfortunately, a few errors came up when developing the code that could not be fixed. With this being said, the semantic analyser does not work exactly as intended. However, in theory, it does abide by the expectations of the language.

Appendix

Token Types

```
public enum TokenType {
   TOK_EOF,
   TOK_Error,
   TOK_LeftBracket,
   TOK_RightBracket,
   TOK_LeftCurly,
   TOK_RightCurly,
   TOK_Multiplication,
   TOK_Addition,
   TOK_Relation,
   TOK_Comma,
   TOK_Colon,
   TOK_Semicolon,
   TOK_Equal,
   TOK_BooleanValue,
   TOK_FloatValue,
   TOK_IntegerValue,
   TOK_BooleanKeyword,
   TOK_FloatKeyword,
   TOK_IntegerKeyWord,
   TOK_Identifier,
   TOK_Let,
   TOK_Print,
   TOK_Return,
   TOK_Function,
   TOK_If,
   TOK_Else,
   TOK_For,
   TOK_Logic,
   TOK_Comment
}
```

Classifier

```
public class Classifier{
   public static final int EOF = 0;
   public static final int COMPARISON = 1;
   public static final int EQUALS = 2;
   public static final int EXCLAMATION = 3;
   public static final int ADD_AND_SUB = 4;
   public static final int ASTERISK = 5;
   public static final int SLASH = 6;
```

```
    public static final int NEW_LINE = 7;
    public static final int PUNCTUATION = 8;
    public static final int LETTER = 9;
    public static final int DIGIT = 10;
    public static final int DOT = 11;
    public static final int UNDERSCORE = 12;
    public static final int OTHER = 13;
}
```