Compiler for Minilang - CPS2000

Jesmar Farrugia,
Department of AI,
University of Malta
Msida, MSD 2080
jesmar.farrugia.17@um.edu.mt

I. Introduction

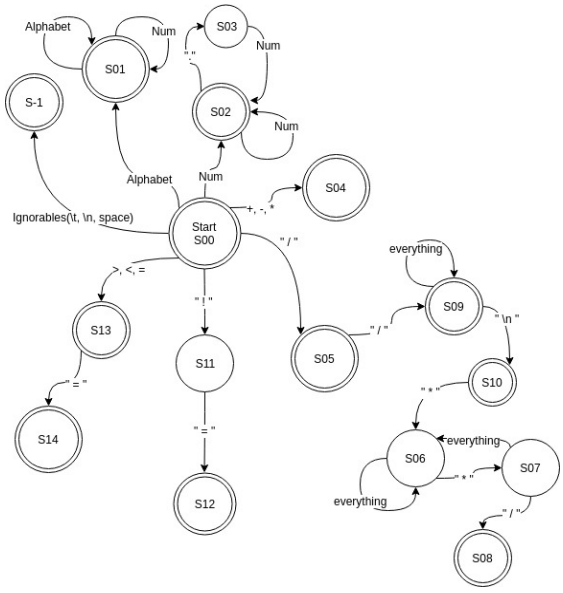
The aim of this assignment is to create the front end of a compiler. The five main tasks of this assignment are: the table driven lexer; a predictive, recursive descent parser; XML generation; semantic analysis; and interpreter execution. First a token class was made, in order to make sense of what the user will pass as code.

II. Token

For the token class, only a header file was created since only the declaration of the variables where needed, and no functions regarding the tokens were made. Token contains an enumerable that contains all token types present in the EBNF given, a variable token, to store the token type, a string that stores the actual lexeme, and a float to store the value of the lexeme or the number. It has two constructors, on that takes no parameters but generates a NONTOKEN with empty variables (used to indicate errors in lexing), and the other takes a token type, the lexeme and a float value as parameters, and generates a token respective to the parameters given.

III. Lexer

Before the lexer was coded, the table of states had to be created. The transition from one state to another was planned as such:



Not all states were included in the dfa, but the ones that weren't just lead to accepted states (like brackets for example).

Apart from the table of states, the lexer class also contains a string the holds all the data from a text file (to be compiled), a vector that will contain all of the tokens generated, a token counter (to be used by the *getNextToken()* method), and a couple of string arrays that hold specific keywords or characters (to compare with lexemes and generate the respective token).

The first function in lexer, *getRow(string)*, takes a string as a parameter, compares it to all key characters in the DFA, and returns which row the character is in the table. This is used to determine the current state whilst parsing the text file.

states(string, int) function takes the current lexeme and the current state as parameters, generates the respective token, and appends it to the vector of tokens.

This is used by the function *GenerateTokens()*, which takes no parameters. The mentioned function loops through the entire file, character by character, depending on the character and the previous state, it will parse the table, and when the table returns a -1 (default state), it will call the function *states()* passing the current lexeme and the state as parameters.

The other functions in the lexer (apart from the *peekNextToken()* and *getNextToken()*) just take a lexeme and return a token depending on the lexeme (for example *keywordHandling()* will return a token depending on which keyword the lexeme matched with).

peekNextToken() and *getNextToken()* work quite similarly. If the tokenCounter variable is smaller than the size of the vector holding all token, the it will return the current token in the index tokenCounter, else it will return a NONTOKEN, which will cause the lexer to stop. The only difference is that in *getNextToken()* the counter is increased, making it impossible for other function to access previous tokens, since the counter is a private variable.

Apart from the constructor, *peekNextToken()* and *getNextToken()*, all other functions and variables are private, as to not be changed by other functions unrelated to the lexer.

IV. Parser

All nodes (classes) that are needed for the parse tree were initialized on a single header file. This was done to avoid separate header files calling themselves along the generation of the tree (as is present in the expression node, eventually, following the EBNF, expression will call itself once it reaches the actual parameters node, for example).

Most nodes, conceptually, work similarly. Those that need to check for specific tokens will return an error message and exit, if the token doesn't match, and if the token does match and is useful then it is stored, else it is discarded. Apart from this check, almost all nodes store another node depending on their progression in the EBNF. Example taking an *if* statement:

The parser stores a pointer, which holds the address of program node. The program node contains a vector of statements, and the constructor will keep appending new statements until a NONTOKEN is reached. Once program appends a statement node, it is calling the statement node class, which will peek at the next token and determine which node to call on next, in this case it will detect an *if* token. Once if is detected, the token is popped (getNextToken() is called but not stored), and an *if* node is stored/created. The if statement node will check for specific tokens for syntax analysis (like the appropriate brackets, semicolons, and variable declarations), and if they are not detected, then an error message will ensue and the program will exit, else it will create and store an expression node. At the end it checks if the next token is an *else*, if it is then it pops the *else* token and creates and stores a block node, else it will not show an error message, since an *if* statement does not strictly need an *else*.

As mentioned before all other nodes follow the same concept, and all the nodes were constructed according to the EBNF (only printables were excluded).

V. AST XML Generation

For XML generation, the visitor design pattern [1] was implemented. This pattern allows to add methods to classes (of different types) without much altering of the classes themselves. In this case every class needed an accept method which called their respective visitor. Every visitor had a different function in representing the current node as XML, and displayed relevant token throughout. Since all nodes called other nodes recursively, the visitor was implemented the same way, for example: the visitor of the program node will first print out the open program tag (<Program>), call on the accept method of the statement node,

which will call on the respective visitor, and so on, recursively calling all the visitors of the nodes within the parse tree. Once that is done, the closing tag of program is closed (</Program>).

VI. Semantic Analysis

For semantic analysis, the visitor pattern would also be used. The difference is that the visitor class holds multiple tables which should group up variables and functions with their respective values or node or types (scope). For example, in this sample code:

```
int x = 0;
{
        bool y = true;
        int x = 5;
}
```

The variable x outside the brackets is in a different scope from the variable x inside the brackets, and if a variable is called within the brackets that is not present within the respective scope, it will go back and check if the variable was declared before. This is done to handle errors regarding variables and functions that are called, but that were not initialized or declared before hand. These scopes should also handle errors regarding incorrect assignment of types. The visitor should handle the creation of these scopes and ensure that the code currently called is semantically correct.

VII. Interpreter Execution

After the tokens are generated, the syntax is checked and the parse tree is generated, and a semantic analysis is done, the interpreter pass should grab the tree and scopes generated and compute all the different operations present in the program, and store them back in the scope, in their respective place. The interpreter basically is a simple representation of what the backend of a compiler should do.

VIII. Conclusion

The lexer goes through every character in the given code, generating token and feeding them to the parser. The parser takes the given token and generates a tree respective to the tokens received. The XML pass, the semantic, and the interpreter pass all implement a basically similar visit pattern, all having a different outcome. The XML pass makes a simple representation of the code, the semantic pass checks for errors regarding semantics, and the interpreter executes the code.

IX. Resources

[1].https://www.youtube.com/watch?v=pL4mOUDi54o