Tracy Tonnu
Cody Thompson
Russel Koh

Top-Down Syntax Analyzer: Stack Implementation

May 15, 2020

1. Problem Statement

The purpose of the assignment was to reimplement the syntax analyzer segment of the compiler using a top-down approach with a stack. The syntax analyzer works with the lexical analyzer to determine if the input statements are valid syntactically based on the provided grammars. A sequence of characters is initially read in via a text file or console and analyzed by the lexer to determine if the initial input is valid. Once the lexer has validated the text input, the statements are passed to the parser where it is broken up and pushed onto a vector stack with a '$' appended at the end of the statement to signify the end of the statement. Within the parser class, various functions are called to validate the statement based on productions rules of griven grammars. If the input statement is found to be valid, the token and productions will be printed to the screen.

2. How to Use Your Program

Before the syntax analyzer can interpret the input statement, the lexical analyzer is called first to evaluate the text input. A Lexer object is instantiated from the main program file, with a text filed passed through as a parameter. If the Lexer object is able to successfully open the text file, the Lexer object runs the necessary functions so that the syntax analyzer portion of the compiler can run successfully.

In order to run the syntax analyzer portion of the compiler, a Parser object is instantiated and accepts the Lexer object as a parameter in the constructor. Once the Parser object has been successfully created, the Parser adds a '$' to the end of the statement to signify the ending case of the statement. A '$' is also pushed onto the stack to be compared with the end of the statement.

The Parser continues to get lexemes from the passed Lexer object and checks if the lexeme is a separator. If the Parser hits a separator, it will break out. Within the Parser's constructor, it calls upon the member function SyntaxAnalysis to complete the syntactical analysis of the statement. No other explicit calls are needed within the main to perform the analysis.

3. Design of Your Program

FIRST(**α**):

FIRST(S) = { type, i }
FIRST(D) = { type }
FIRST(A) = { i }
FIRST(E) = { (, i, num }
FIRST(E') = { +, -, **ε** }
FIRST(T) = FIRST(E) = { (, i, num }
FIRST(T') = { *, /, **ε** }
FIRST(F) = FIRST(E) = { (, i, num }

FOLLOW(**α**):

FOLLOW(S) = { $ }
FOLLOW(D) = { $ }
FOLLOW(A) = { $ }
FOLLOW(E) = {), $ }
FOLLOW(E') = {), $ }
FOLLOW(T) = { +, -,), $ }
FOLLOW(T') = { +, -,), $ }
FOLLOW(F) { *, /, +, -,), $ }

The syntactical analyzer consists of a Parser class with a constructor and functions to perform the analysis on the passed statements from the Lexer object. The Parser class contains one constructor and nine member functions, and two private member variables.

The Parser class utilizes several standard libraries to run: string, vector, and iostream. These libraries are included in the Lexer class and inherited by the Parser class. These libraries provide a way to store and print the tokens and lexemes of the statements being passed and analyzed by the Parser.

CONSTRUCTOR for the Parser Class:

Parser(Lexer lex)

When called, the Parser object is instantiated with a Lexer object passed through as a parameter. The '$' symbol is pushed onto the end of the statement to signify the end of the statement and the tokens and lexemes are paired and stored as statements in a vector to be analyzed by the various grammar checking functions found in the Parser class. The constructor also prints the resulting analysis of the passed statements.

FUNCTIONS for the Parser Class:

SyntaxAnalysis( )

The SyntaxAnalysis (SA) function calls on the DeclarationRule and AssignRule functions to verify the statement's grammar. The SA function takes no parameters but calls upon the private variable of the Parser that stores the statement to be analyzed. When calling the Rule functions, the SA will return True if the statement is validated by either rule, otherwise it will return False.

DeclarationRule( )

The DeclarationRule (DR) function is used to determine if the statement stored by the Parser is a valid declarative based on the given productions of the grammar provided. The function looks for an identifier token in the pair, and if one is found, DR will print the lexeme and the production rule that corresponds to it.

AssignRule( )

The AssignRule (AR) function's goal is to determine if the statement is a valid assignment based on the given grammar productions. The AR calls upon the ExpressionRule (ER) function to analyze the statement being passed. The AR looks for an identifier token as well as an assignment operator (=). If AR finds one, the program attempts to determine if the statement is an expression and will return True if the statement is a valid assignment.

ExpressionRule( )

The ExpressionRule (ER) function determines if the statement being analyzed is an expression based on the given productions. The ER calls upon the TermRule (TR) and ExpPrime (E') functions to further analyze the expression. If these two functions return True, then the statement is a valid expression and the ER will also return True.

ExpPrime( )

The ExpPrime (E'), a production created to remove left recursion from the expression production, handles the end cases for the expression rule based on the given grammar. The function looks for certain operators before entering the main body of the function to run any type of analysis on the statement. Within E', it calls upon the TermRule (TR) and recursively calls upon itself until it finds an end statement symbol.

TermRule( )

The TermRule (TR) function calls upon the FactorRule (FR) and TermPrime (T') functions to analyze the statement stored by the Parser. There are no other conditions checking within the function before calling upon FR and T'. If both of the functions return True, the statement being analyzed is a valid term and TR will return True.

TermPrime( )

TermPrime (T') removes left recursion from the TR function and handles end cases for the term rule based on given productions. It looks for operations before calling upon the FR and recursively calling itself to search for end statement symbols. If the function successfully steps through the statement and finds an end case, T' will return True and validate the stored statement.

FactorRule( )

The FactorRule (FR) looks for the open parenthesis before stepping through the stored statement. If an open parenthesis is found, the FR calls upon the ER and looks for the closing parenthesis. If the open parenthesis is not found, the FR will look at the token associated with the first member of the pair to check if it is an identifier. If an identifier is not found, the FR looks for an integer token. If either of these conditions are met, the FR will return True and a valid factor has been identified.

4. **Any Limitations**

None

5. **Any Shortcomings**

None