Tracy Tonnu
Cody Thompson
Russel Koh

Syntax Analyzer

April 16, 2020

1. Problem Statement

The purpose of the assignment was to create the syntax analyzer segment of a compiler. Working with the lexical analyzer created during the first assignment, the syntax analyzer is to determine if the input statement is valid syntactically based on the provided grammars. A sequence of characters is originally read in via text file or console and analyzed by the lexer to determine if the input text is valid. Once the lexer has validated the input, it is passed to the parser, where the statement is pushed and stored in a vector with '$' appended to be used as an end case. The parser then calls on various functions to validate the statement based on production rules of given grammars. If the input is valid, the token and productions will be printed to the screen.

2. How to Use Your Program

Before the syntax analyzer can be run, the lexical analyzer must be run to evaluate the inputted text. A Lexer object is instantiated, with a text file passed through as a parameter. If the text file is successfully open, the Lexer object performs the necessary functions so that the syntax analyzer portion of the program may be run successfully.

In order to access the syntax analyzer, a Parser object is instantiated, which accepts a Lexer object as a parameter. Once the program has initialized a Parser object, successfully stores the tokens and lexemes from the Lexer object into a vector in token-lexeme pairs, and appends an end statement symbol ($) to the statements pushed onto the vector, the Parser will begin the grammatical analyses of the stored statements.

Within the Parser object's program code, it calls upon the member function within the class to perform the syntactical analysis. No other explicit calls are needed within the main.

Tracy Tonnu
Cody Thompson
Russel Koh

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 syntactical analysis on the passed statements from the Lexer object. The Parser class contains one constructor and nine member functions, and one private member attribute.

The Parser class utilizes several standard libraries to run, string, vector, and iostream, included in the Lexer class. These libraries provide a simple way to store and print the tokens and lexemes of the statements being passed through to the Parser object.

Tracy Tonnu
Cody Thompson
Russel Koh

CONSTRUCTORS for the Parser Class:

Parser(Lexer lex)

When called, the Parser object is instantiated with a Lexer object passed through as a parameter. The tokens and lexemes obtained from the Lexer object are paired and stored as statements in a bector to be analyzed by the various grammar checking functions provided by the Parser class. When the statements are pushed onto the stack, a end statement signifier, $, is placed at the end to alert the program when the end of the statement has been reached. The constructor also prints the resulting analysis of the passed statements.

FUNCTIONS for the Parser Class:

SyntaxAnalysis(std::vector<std::pair<std::string, std::string>>statement)

The behavior of the SyntaxAnalysis (SA) function is to call on the DeclarationRule and AssignRule functions to check the statement's grammar. The SA function takes one parameter, a vector, which consists of pairs of lexemes and tokens created within the constructor as statements. When calling the Rule functions, the SA will pass the statement vector through to the Rule functions and return True if the statement is validated by either rule, and false otherwise.

DeclarationRule(std::vector<std::pair<std::string, std::string>>statement)

The DeclarationRule (DR) function is used to determine if the passed statement 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 and the production rule that corresponds to it.

AssignRule(std::vector<std::pair<std::string, std::string>>statement)

The AssignRule (AR) function's primary goal is to determine if the passed statement is a valid assignment based on the grammar productions provided. Within the AR, it calls upon the ExpressionRule (ER) function to further analyze any statement being passed. Similarly to the DR, the AR looks for an identifier token but goes on to look for the assignment operator (=). If one is present, the program tries to determine if the statement is an expression and will return true if the statement is a valid assignment.

Tracy Tonnu
Cody Thompson
Russel Koh

ExpressionRule(std::vector<std::pair<std::string, std::string>>statement)

The ExpressionRule (ER) function is used to determine if the statement being analyzed is an expression given the declared productions. Within the ER, it 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 return true.

ExpPrime(std::vector<std::pair<std::string, std::string>>statement)

The ExpPrime (E'), which was created to remove left recursion from the expression production rule, handles end cases for the expression rule based on given productions. The function looks for certain operators (+, -) before entering into the main body of the function to run further analysis. Within E', it calls upon the TermRule (TR) and upon itself until it finds an end statement symbol ($,)).

TermRule(std::vector<std::pair<std::string, std::string>>statement)

The TermRule (TR) function calls upon the FactorRule (FR) and TermPrime (T') functions to analyze the passed statement to determine if it is a valid term. There are no other conditions checked within the function before calling upon FR and T'. If both these functions return true, then the analyzed statement is a valid term and TR will also return true.

TermPrime(std::vector<std::pair<std::string, std::string>>statement)

The TermPrime (T') function removes left recursion from the TR function and handles end cases for the term rule based on given productions. It begins by looking for operations (*, /) before calling upon the FR and itself (T') to search for end statement symbols (), $). If the function successfully steps through the statement and finds the end cases, T' will return true and validate the passed statement.

FactorRule(std::vector<std::pair<std::string, std::string>>statement)

The FactorRule (FR) looks for the open parenthesis (() before stepping through the passed statement. If it is found, the FR calls upon the ER and looks for the closing parenthesis ()). If the open parenthesis is not located in the statement, the FR will look to see if the token associated with the first member of the pair is an identifier. If an identifier is not found, the FR looks for an integer token. If any of these conditions are met, the FR returns true and a valid factor has been identified.

Tracy Tonnu
Cody Thompson
Russel Koh

4. **Any Limitations**

   None

5. **Any Shortcomings**

   None