Lab Manuals

Compiler Construction

****

Implementing tokenizer and parser of the compiler in C++ with help of STL

**Designed by**

Laeeq Khan Niazi

Lecturer Computer Science Department,

University of Engineering and Technology, Lahore.

**Department of Computer Science**

**University of Engineering and Technology, Lahore.**

**Lab Activity.**

[**https://fileadmin.cs.lth.se/cs/Education/EDAN65/2020/lectures/L03.pdf**](https://fileadmin.cs.lth.se/cs/Education/EDAN65/2020/lectures/L03.pdf)

Visit the link and understand the fundamental concept of CFG and on slide/page 50 try to understand the Java code. After getting enough understanding try to write a short code in your C++ to implement few parts of CFG.

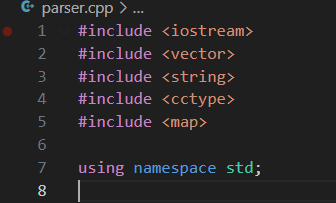
**Complete Guide to implement a parser**

**Step1:**

Create a clean folder with file name like compiler.cpp or parser.cpp

**Step 2:**

Import following libraries



**Step 3:**

**Explanation**

Defines an enumeration TokenType, representing different types of tokens that the lexer can recognize, such as keywords (T\_IF, T\_ELSE), operators (T\_PLUS, T\_MINUS), symbols (T\_LPAREN, T\_RPAREN), and T\_EOF to mark the end of the input.

**How to advance it?**

1. Add more token types for additional language constructs (e.g., T\_FOR, T\_WHILE, T\_EQ for ==, T\_LE for <=, T\_AND for &&).
2. Extend the enum to handle floating-point numbers (T\_FLOAT), string literals (T\_STRING), and other advanced types

A computer screen shot of a black screen

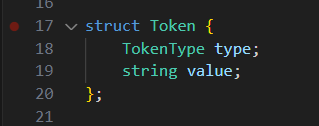
Description automatically generated

**Step 4:**

Defines a Token structure that stores the type of token (type) and its corresponding value (value), which is the literal string from the source code.

**How to Advance it?**

1. Add a line number and column number to the Token struct for improved error reporting and debugging.
2. Store additional metadata, such as the data type for T\_ID tokens (e.g., int, float).



**Step 5:**

Create Lexer class in C++

A screen shot of a computer

Description automatically generated

In the class make two sections private and public.

A screenshot of a computer program

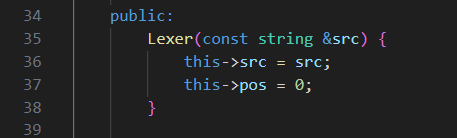
Description automatically generated

Create two data members src and pos to hold the code and position of the pointer

A black background with yellow text

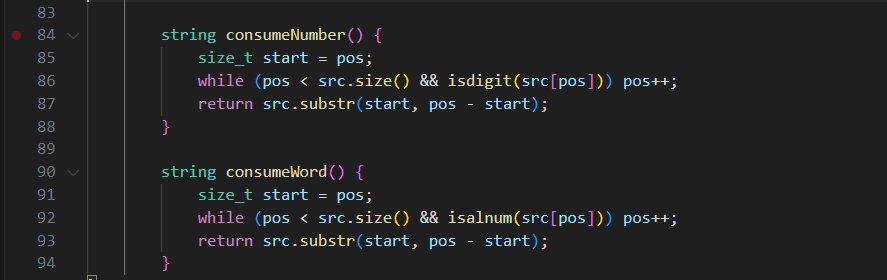
Description automatically generated

In public part make the constructor of the class



Write two functions consumer Number and consume Word. In the Lexer class, the functions consumeNumber() and consumeWord() are responsible for extracting numbers and words (identifiers or keywords) from the source code string (src).

1. In consumeNumber() function the loop continues advancing the pos pointer as long as the current character is a digit.
2. In consumeWord() function, the loop continues advancing pos as long as the current character is alphanumeric (checked using isalnum(src[pos])). Alphanumeric characters include letters (a-z, A-Z) and digits (0-9).



**Write a function name Tokenizer**

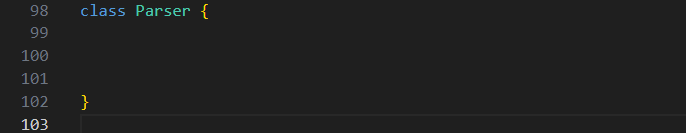
 A screen shot of a computer program

Description automatically generated

**Algorithm of tokenzier**

* Initialize an empty list of tokens
* Iterate over each character
* Skip whitespace (isspace(current))
* Handle numbers (isdigit(current) it calls consumeNumber() to extract full number)
* Handle words (keywords or identifiers) (isalpha(current) it calls consumeWord() to extract whole word)
* Handle symbols and operators. If we have symbol like +,\*,/,=,) directly make token and add to the token vector/list.
* When the end of the input is reached, the function adds an EOF token to signify that no more tokens are left.

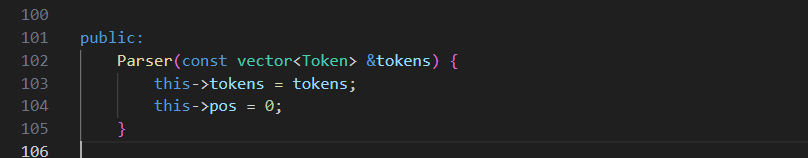
Make one more class named parser



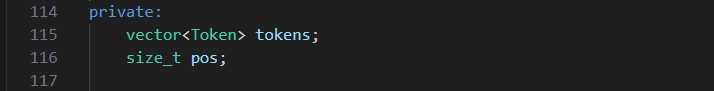
Define two sections public and private



In public section define the constructor of the parser class to take the vector of tokens and assign pos variable with zero.



In private section make two data structures vector and pos.



In public section of the class define one more function parseProgram() it will loop through the tokens vector until it encounter the T\_EOF token.

A screen shot of a computer

Description automatically generated

Write the functions in parser private section to parse different parts of the code like

Parsing statements, parse blocks, parse variable declarations, parse assignments, parse if statements, parse return statements parse expressions etc.

Starting with parseStatement() function.



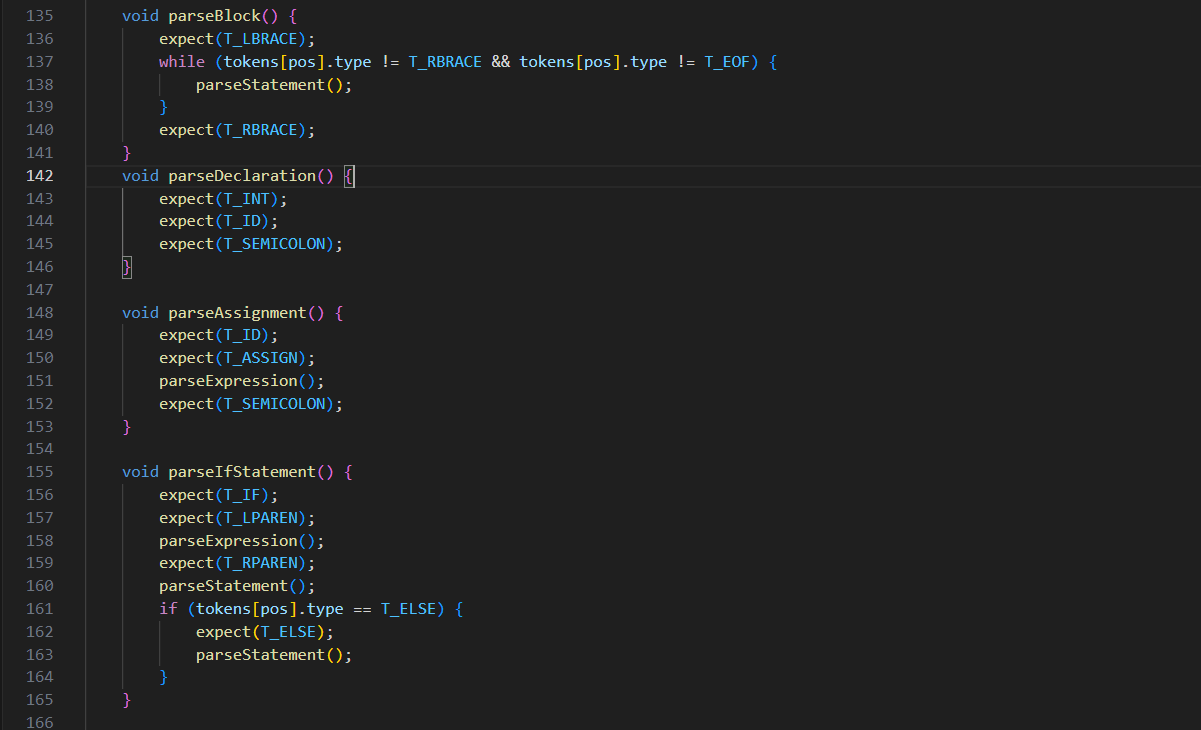
**How the code is working?**

This is very important function call other parsing functions on the basis of token encountered.

This function determines the type of statement (e.g., declaration, assignment, if, return, block) and calls the corresponding parsing function. It checks the current token and calls the appropriate function based on the token type.

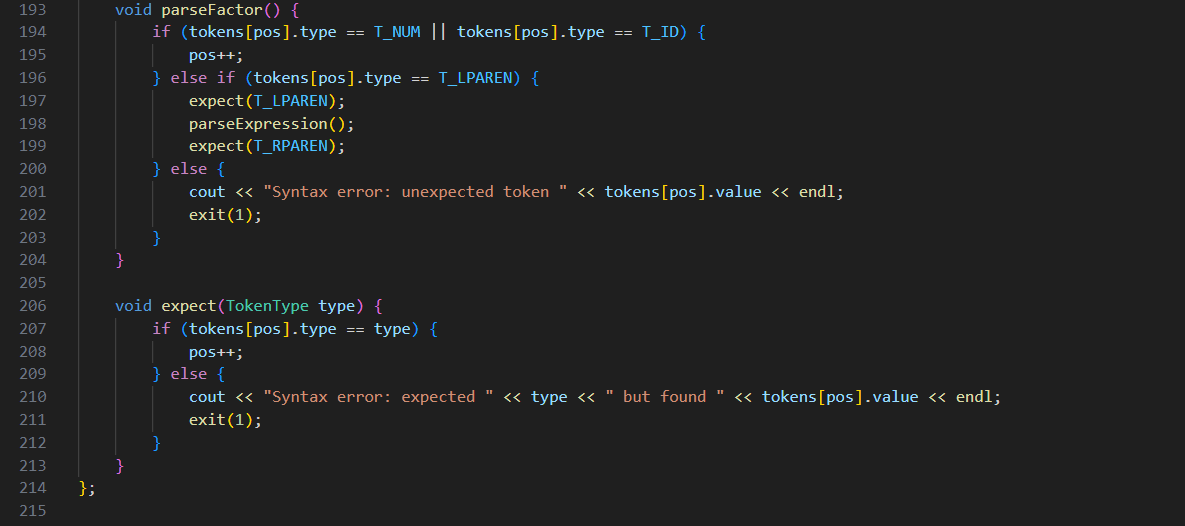
1. If the token is T\_INT, it calls **parseDeclaration**() to handle variable declarations.
2. If the token is T\_ID, it calls **parseAssignment**() to handle assignments.
3. If the token is T\_IF, it calls **parseIfStatement**() to handle if conditions.
4. If the token is T\_RETURN, it calls **parseReturnStatement**() for return statements.
5. If the token is { (i.e., T\_LBRACE), it calls **parseBlock**() to handle blocks of code.
6. Otherwise, it throws a syntax error if none of the expected tokens are found.

Different functions to handle the tokens



A screenshot of a computer program

Description automatically generated



**How are the above parse functions working? Let’s see the explanation.**

1. **parseBlock()**

This function handles blocks of code enclosed in {}. It expects an opening brace { (T\_LBRACE) and a closing brace } (T\_RBRACE). It checks for an opening brace, then repeatedly parses statements within the block until it finds the closing brace or end of the file. It expects a closing brace to ensure the block is properly terminated.

1. **parseDeclaration()**

This function parses a variable declaration statement, such as **int a;** It expects the token T\_INT (for the int keyword). It then expects an identifier (T\_ID), which is the variable name. Finally, it expects a semicolon (T\_SEMICOLON) to mark the end of the declaration.

1. **parseAssignment()**

This function handles assignment statements like **a = 5;** It expects the token T\_ID for the variable being assigned a value. It then expects the = token (T\_ASSIGN).

After that, it parses the expression on the right-hand side of the assignment using parseExpression(). Finally, it expects a semicolon (T\_SEMICOLON) to mark the end of the assignment statement.

1. **parseIfStatement()**

This function parses if statements, including optional else clauses. It expects the if keyword (T\_IF). It expects a left parenthesis ( (T\_LPAREN), followed by an expression representing the condition, and then a right parenthesis ) (T\_RPAREN). After the condition, it expects a statement (which could be a block or a single statement) to execute if the condition is true. If the next token is else (T\_ELSE), it also parses the statement that follows the else keyword.

1. **parseReturnStatement()**

This function parses return statements, like **return 5;** It expects the return keyword (T\_RETURN). Then it parses the expression after the return. Finally, it expects a semicolon (T\_SEMICOLON) to end the return statement.

1. **parseExpression()**

This function parses expressions that involve addition, subtraction, or relational operators (like >). For example, expressions like **a + 10** or **b > 5**. It first parses a term (a piece of an expression) using parseTerm(). It checks for + (T\_PLUS) or - (T\_MINUS) operators and continues parsing additional terms if these operators are found. It also checks for the relational operator > (T\_GT) and recursively parses the next expression after >.

1. **parseTerm()**

This function handles multiplication and division within an expression, like **a \* 5**. It parses a factor (a basic component of an expression) using parseFactor(). It checks for \* (T\_MUL) or / (T\_DIV) operators and continues parsing additional factors if these operators are found.

1. **parseFactor()**

This function parses the smallest units of an expression, such as numbers, variable identifiers, or parenthesized expressions. If the current token is a number (T\_NUM) or an identifier (T\_ID), it consumes the token and moves on. If the current token is a left parenthesis ( (T\_LPAREN), it recursively parses an expression inside the parentheses, and then expects a closing parenthesis ) (T\_RPAREN). If none of these are found, it throws a syntax error.

1. **expect(TokenType type)**

This function is used to check that the current token is of the expected type. If the token is as expected, it moves to the next token. If it is not, it throws a syntax error.

**Finally**

Write the main function, store your code in the double quotes and pass the code to tokenizer to get tokens and then pass the tokens to the parser to check the syntax of your code.

A computer screen shot of a black background

Description automatically generated