

Session: 2021 – 2025

**Submitted by:**

Abdul Mateen 2021-CS-190

**Supervised By:**

Sir Laeeq Khan Niazi

Department of Computer Science

**University of Engineering and Technology**

**Lahore Pakistan**

Contents

[Lexical Analyzer 4](#_Toc184887527)

[Parser Class 7](#_Toc184887528)

[IntermediateCodeGenerator Class 10](#_Toc184887529)

[SymbolTable Class 12](#_Toc184887530)

[1. Enum: SymbolType 12](#_Toc184887531)

[2. Class: Utils 12](#_Toc184887532)

[3. Struct: Symbol 13](#_Toc184887533)

[4. Class: SymbolTable 13](#_Toc184887534)

[TACToAssemblyConverter Class 14](#_Toc184887535)

[ThrowError class 17](#_Toc184887536)

[Static Methods: 17](#_Toc184887537)

[Error Reporting: 17](#_Toc184887538)

[Program Termination: 17](#_Toc184887539)

[Methods and Their Purpose 17](#_Toc184887540)

[unExpectedTokenError: 17](#_Toc184887541)

[expectedTokenError: 17](#_Toc184887542)

[sementicErrorVariableAlreadyDeclared: 17](#_Toc184887543)

[sementicErrorFunctionAlreadyDeclared: 18](#_Toc184887544)

[sementicErrorVarableNotDeclared: 18](#_Toc184887545)

[sementicErrorConstantAssignment: 18](#_Toc184887546)

[sementicErrorInvalidType: 18](#_Toc184887547)

[Usage 19](#_Toc184887548)

[TACToAssemblyConverter class 19](#_Toc184887549)

[Class Components 19](#_Toc184887550)

[Private Members 19](#_Toc184887551)

[Private Methods 20](#_Toc184887552)

[Public Members 21](#_Toc184887553)

[Key Features 23](#_Toc184887554)

[Extra Features Documentation 24](#_Toc184887555)

[1. Single-Line and Multi-Line Comments 24](#_Toc184887556)

[2. Additional Operators 24](#_Toc184887557)

[3. Additional Data Types 24](#_Toc184887558)

[4. Functions 25](#_Toc184887559)

[Constant Variables 25](#_Toc184887560)

[Assembly Generation Using LRU Algorithm for Register Allocation 25](#_Toc184887561)

# Lexical Analyzer

**Overview**

The lexical analyzer, also known as the scanner or lexer, is the first phase of a compiler. It is responsible for reading the source code and converting it into a sequence of tokens, which are meaningful sequences of characters. These tokens are then passed to the subsequent phases of the compiler for further processing.

**Objectives**

* **Tokenize Source Code**: Identify and classify meaningful sequences in the input source code.
* **Error Detection**: Detect invalid tokens and report lexical errors.
* **Simplification**: Reduce the complexity of the subsequent phases by abstracting the raw input into structured tokens.

**Input and Output**

* **Input**: The raw source code written in a programming language.
* **Output**: A sequence of tokens, where each token has the following attributes:
  + Token type (e.g., identifier, keyword, operator, literal)
  + Token value (the exact substring from the source code)
  + Line number (for error reporting and debugging)

**Components of the Lexical Analyzer**

**1. Token Categories**

The lexical analyzer identifies the following categories of tokens:

* **Keywords**: Reserved words in the language (e.g., if, else, while).
* **Identifiers**: Names for variables, functions, or other user-defined entities.
* **Literals**: Numeric, string, or character constants.
* **Operators**: Arithmetic, logical, relational, and assignment operators.
* **Delimiters**: Symbols used for structuring the code (e.g., ;, ,, {, }).
* **Comments**: Single-line and multi-line comments, which are ignored by the compiler.

**2. Finite Automata**

The lexical analyzer uses finite automata to recognize tokens:

* **Deterministic Finite Automaton (DFA)**: Used for token recognition.
* **Transition Table**: Maps states to transitions based on input characters.

**3. Symbol Table**

A symbol table is used to store identifiers and their associated metadata. The lexical analyzer populates this table with entries for identifiers and literals.

**4. Error Handling**

The lexical analyzer detects lexical errors such as:

* Unrecognized characters
* Unterminated strings
* Invalid numeric literals

It reports errors with line numbers and a description of the issue.

**Workflow**

1. **Input Buffering**: The source code is read into an input buffer to optimize character reading.
2. **Lexeme Recognition**: The lexer scans the input buffer to identify lexemes that match predefined patterns.
3. **Token Generation**: For each recognized lexeme, a corresponding token is generated.
4. **Error Reporting**: If an invalid lexeme is encountered, an error message is generated.
5. **Token Output**: The generated tokens are sent to the parser.

**Implementation Details**

* **Regular Expressions**: Define patterns for tokens (e.g., [a-zA-Z\_][a-zA-Z0-9\_]\* for identifiers).
* **Input Buffering**: Uses a two-buffer scheme to handle large inputs efficiently.
* **Lookahead**: Handles ambiguities in token recognition by peeking at upcoming characters.

**Example**

**Input Source Code:**

int x = 10;

if (x > 5) {

x = x + 1;

}

**Generated Tokens:**

|  |  |  |
| --- | --- | --- |
| **Token Type** | **Token Value** | **Line Number** |
| Keyword | int | 1 |
| Identifier | x | 1 |
| Operator | = | 1 |
| Literal | 10 | 1 |
| Delimiter | ; | 1 |
| Keyword | if | 2 |
| Delimiter | ( | 2 |
| Identifier | x | 2 |
| Operator | > | 2 |
| Literal | 5 | 2 |
| Delimiter | ) | 2 |
| Delimiter | { | 2 |
| Identifier | x | 3 |
| Operator | = | 3 |
| Identifier | x | 3 |
| Operator | + | 3 |
| Literal | 1 | 3 |
| Delimiter | ; | 3 |
| Delimiter | } | 4 |

**Error Examples**

**Input Source Code:**

int 123abc = 10;

**Error Report:**

|  |  |
| --- | --- |
| Line Number | Error Description |
| 1 | Invalid identifier: 123abc |

**Conclusion**

The lexical analyzer is a crucial component of the compiler, ensuring that the raw source code is converted into structured tokens. This process facilitates the functioning of subsequent compiler phases, such as parsing and semantic analysis.

# Parser Class

**Overview**

The Parser class is responsible for analyzing the syntax of a programming language. It takes the tokenized input from a lexical analyzer and validates the structure of the code according to the language's grammar. The parser typically uses recursive descent parsing techniques to break down the input into meaningful constructs and generates intermediate representations for further compilation phases.

**Constructor**

**Parser(const vector<Token>& tokens, SymbolTable symTable)**

* **Description**: Initializes the parser with a sequence of tokens and a symbol table.
* **Parameters**:
  + tokens: A vector of tokens representing the input source code.
  + symTable: A symbol table to store and retrieve information about variables, functions, and other symbols.

**Private Member Variables**

* **tokens**: A vector containing the input tokens generated by the lexical analyzer.
* **pos**: An integer tracking the current position in the token vector.
* **symTable**: A symbol table for managing identifiers and their associated metadata.
* **icg**: (Optional) Represents an intermediate code generator or data structure for building intermediate representations of the program.

**Private Member Functions**

**Core Parsing Functions**

* **parseProgram()**: Parses the entire program.
* **parseStatement()**: Parses a single statement.
* **parseDeclaration(bool isGlobal, bool isConst)**: Parses variable or constant declarations.
* **parseBlock()**: Parses a block of code enclosed by {}.

**Expression Parsing**

* **parseExpression(TokenType tokenType)**: Parses an expression starting with the given token type.
* **parseTerm(TokenType tokenType)**: Parses a term within an expression.
* **parseFactor(TokenType tokenType)**: Parses factors (basic units) of an expression.
* **parseBooleanExpression(TokenType tokenType)**: Parses a boolean expression.
* **parseBooleanTerm(TokenType tokenType)**: Parses a term in a boolean expression.
* **parseBooleanFactor(TokenType tokenType)**: Parses a factor in a boolean expression.

**Control Flow Parsing**

* **parseIfStatement()**: Parses an if statement.
* **parseWhileStatement()**: Parses a while loop.
* **parseForStatement()**: Parses a for loop.

**Function Parsing**

* **parseFunctionDeclaration(SymbolType returnType, string functionName)**: Parses a function declaration.
* **parseParameterList()**: Parses the parameter list of a function.
* **parseReturnStatement()**: Parses a return statement.

**Utility Functions**

* **parseType()**: Parses a data type.
* **expect(TokenType tokenType)**: Ensures the current token matches the expected type and advances to the next token. Throws an error if the expectation is not met.
* **expectAndReturnValue(TokenType tokenType)**: Similar to expect, but returns the value of the token.

**Miscellaneous Parsing**

* **parseCharacter()**: Parses a character literal.
* **parseString()**: Parses a string literal.
* **parseAssignment(bool isGlobal, bool isConst)**: Parses assignment statements.

**Workflow**

1. **Initialization**: The parser is initialized with a vector of tokens and a symbol table.
2. **Program Parsing**: The parseProgram() function serves as the entry point for parsing.
3. **Construct Parsing**: Different constructs (statements, expressions, loops, etc.) are parsed using specialized functions.
4. **Error Handling**: Utility functions like expect() ensure correctness by checking for specific token types and reporting syntax errors if necessary.
5. **Intermediate Code Generation**: If icg is defined, the parsed constructs are translated into intermediate code.

**Example**

**Input Tokens:**

|  |  |  |
| --- | --- | --- |
| **Token Type** | **Token Value** | **Line Number** |
| Keyword | int | 1 |
| Identifier | x | 1 |
| Operator | = | 1 |
| Literal | 10 | 1 |
| Delimiter | ; | 1 |

**Parsing Process:**

1. **Token Stream**: The tokens vector is processed sequentially.
2. **Variable Declaration**: The parseDeclaration() function identifies the int keyword and processes the declaration of x.
3. **Assignment**: The parseAssignment() function recognizes and processes the assignment operation x = 10.
4. **Output**: The parsed representation and intermediate code (if applicable).

**Error Handling**

The parser detects and reports syntax errors such as:

* Missing or unexpected tokens.
* Misplaced operators.
* Invalid constructs.

Error messages include details about the type of error, the expected token, and the line number.

**Conclusion**

The Parser class is a critical component of the compiler, responsible for syntactic analysis and validation. Its modular design allows for easy extension to support additional language features or constructs.

# IntermediateCodeGenerator Class

**Overview**

The IntermediateCodeGenerator class is a utility for generating intermediate representations of a program. It provides mechanisms for creating temporary variables, generating labels for control flow, and managing a list of instructions for further processing in a compiler.

**Constructor**

**IntermediateCodeGenerator()**

* **Description**: Initializes an instance of the IntermediateCodeGenerator class.
* **Initial Values**:
  + labelCounter: Set to 0, used to generate unique labels.
  + tempCount: Set to 0, used to generate unique temporary variables.
  + instructions: An empty vector to store intermediate code instructions.

**Public Member Functions**

**string newTemp()**

* **Description**: Generates a unique temporary variable name.
* **Returns**: A string in the format tN, where N is the current value of tempCount.
* **Behavior**:
  + Increments tempCount by 1 after each call.

**string newLabel()**

* **Description**: Generates a unique label name for control flow.
* **Returns**: A string in the format LN, where N is the current value of labelCounter.
* **Behavior**:
  + Increments labelCounter by 1 after each call.

**void addInstruction(const string &instr)**

* **Description**: Adds a new instruction to the list of intermediate code instructions.
* **Parameters**:
  + instr: A string representing the instruction to be added.

**void printInstructions()**

* **Description**: Prints all intermediate code instructions to the console.
* **Output**: Each instruction is printed on a new line.

**vector<string> getInstructions()**

* **Description**: Retrieves the list of intermediate code instructions.
* **Returns**: A vector of strings containing all the instructions added so far.

**Private Member Variables**

* **labelCounter**: An integer used to generate unique labels.
* **instructions**: A vector of strings that stores the generated intermediate code instructions.

**Workflow**

1. **Initialize**: Create an instance of IntermediateCodeGenerator.
2. **Generate Variables and Labels**:
   * Use newTemp() to generate unique temporary variable names.
   * Use newLabel() to generate unique control flow labels.
3. **Add Instructions**: Add intermediate code instructions using addInstruction().
4. **Retrieve or Print Instructions**: Use getInstructions() to retrieve the instructions or printInstructions() to display them on the console.

**Example**

**Code:**

IntermediateCodeGenerator icg;

string temp1 = icg.newTemp();

string temp2 = icg.newTemp();

string label1 = icg.newLabel();

icg.addInstruction(temp1 + " = a + b;");

icg.addInstruction(temp2 + " = c \* d;");

icg.addInstruction("goto " + label1);

icg.printInstructions();

**Output:**

t0 = a + b;

t1 = c \* d;

goto L0;

**Use Cases**

* **Temporary Variables**: Efficiently generate unique names for temporary variables during expression evaluation.
* **Control Flow Labels**: Simplify the management of control flow constructs like loops and conditional branches.
* **Instruction Management**: Maintain a clear list of intermediate code instructions for further compilation phases.

**Conclusion**

The IntermediateCodeGenerator class provides essential utilities for generating and managing intermediate representations in a compiler. Its modular design makes it suitable for various types of programming languages and compilers.

# SymbolTable Class

## 1. Enum: SymbolType

This enum defines the types of symbols supported by the compiler. Each symbol type corresponds to a specific token type in the language being compiled. It includes:

* INT: Represents an integer type (corresponding to T\_INT in TokenType).
* FLOAT: Represents a floating-point type (corresponding to T\_FLOAT in TokenType).
* CHAR: Represents a character type (corresponding to T\_CHAR in TokenType).
* STRING: Represents a string type (corresponding to T\_STRING in TokenType).
* BOOL: Represents a boolean type (corresponding to T\_BOOL in TokenType).

## 2. Class: Utils

This class provides utility functions for working with SymbolType and TokenType:

* **getTokenType(TokenType type)**:
  + Converts a TokenType to a SymbolType.
  + It takes a TokenType and returns the corresponding SymbolType based on predefined mappings.
* **getSymbolType(SymbolType type)**:
  + Converts a SymbolType to a TokenType.
  + It takes a SymbolType and returns the corresponding TokenType.
* **getSymbolTypeString(SymbolType type)**:
  + Converts a SymbolType to its corresponding string representation.
  + Returns a string such as "int", "float", "char", or "string" based on the type passed.

## 3. Struct: Symbol

This struct defines a symbol that is used in the symbol table, representing variables and functions.

* **name**: The name of the symbol (variable or function).
* **type**: The type of the symbol (e.g., INT, FLOAT, etc.).
* **isConstant**: A flag indicating whether the symbol is a constant.
* **params**: A vector of parameter types and names, used for functions to store the types and names of the parameters.

## 4. Class: SymbolTable

The SymbolTable class provides methods for declaring and retrieving symbols (variables and functions). It maintains a map of symbols (both variables and functions).

* **declareVariable(const string &name, const SymbolType type, int lineNumber, bool isConstant = false)**:
  + Declares a variable in the symbol table.
  + If the variable is already declared, a semantic error is thrown.
* **getVariableType(const string &name)**:
  + Returns the type of a variable.
  + If the variable is not declared, a semantic error is thrown.
* **isConstant(const string &name)**:
  + Checks if a variable is a constant.
  + If the variable is not declared, a semantic error is thrown.
* **isDeclared(const string &name) const**:
  + Checks if a variable or function has already been declared.
* **declareFunction(const string &name, const SymbolType returnType, const vector<pair<SymbolType, string>> &params, int lineNumber)**:
  + Declares a function in the symbol table.
  + If the function is already declared, a semantic error is thrown.
* **isFunctionDeclared(const string &name, const SymbolType returnType, const vector<pair<SymbolType, string>> &params)**:
  + Checks if a function with the specified name, return type, and parameters is already declared.
  + This function supports function overloading (with different parameter types).

**Phase of Compiler: Symbol Table Management**

This code snippet primarily deals with **symbol table management** in a compiler. The symbol table is an essential data structure used by a compiler to store information about variables, constants, functions, and their types. The compiler uses the symbol table to check for semantic errors like undeclared variables, incorrect variable types, and multiple declarations of the same variable or function.

**Explanation of the Phase:**

1. **Symbol Table Creation**:
   * The symbol table holds information about the symbols used in the source code.
   * It is typically used in the semantic analysis phase, where the compiler checks if the variables and functions are used correctly according to the rules of the language.
2. **Declaration**:
   * The declareVariable and declareFunction methods handle symbol declaration. These methods ensure that a variable or function is only declared once and checks for any semantic errors like trying to redeclare a variable or function that has already been declared.
3. **Type Checking**:
   * The getVariableType and isConstant methods help the compiler verify the type of a variable and check if it is a constant.
   * These checks are important for enforcing correct type usage and preventing errors during code execution.
4. **Handling Overloading**:
   * The isFunctionDeclared method supports checking function overloading by comparing the function’s name, return type, and parameters. This is important for languages that support multiple functions with the same name but different parameter types.
5. **Error Handling**:
   * The ThrowError class (assumed to be defined elsewhere) is used to throw semantic errors when a variable or function is not declared or is declared multiple times, ensuring that the generated code adheres to the rules of the language.

## TACToAssemblyConverter Class

This class is responsible for converting Three-Address Code (TAC) to assembly code. It processes each line of TAC and generates corresponding assembly instructions. The conversion involves handling arithmetic operations, assignments, conditional jumps, logical operations, function definitions, and function calls. It also manages register allocation using a least-recently-used (LRU) strategy.

**Private Member Variables**

* **tac**: A vector of strings storing the input TAC code.
* **assembly**: A vector of strings that holds the generated assembly code.
* **registerQueue**: A priority queue managing register usage, where the least recently used (LRU) register is prioritized.
* **registerUsage**: A map tracking the usage order of registers.
* **variableToRegister**: A map associating variables with allocated registers.
* **registerToVariable**: A map associating registers with the variables they hold.
* **usageCounter**: A counter for keeping track of register usage order.

**Private Member Functions**

* **initializeRegisters()**: Initializes the priority queue with 8 registers (R1 to R8), setting their usage to 0.
* **getRegister(string &variable)**: Allocates a register to a variable. If the variable is already mapped to a register, it returns the corresponding register. Otherwise, it pops the least recently used register from the queue and updates the mappings.
* **split(const std::string &line, char delimiter = ' ')**: A helper function that splits a string by the specified delimiter (default is space) and returns the resulting tokens as a vector of strings.
* **handleArithmetic(const std::vector<std::string> &tokens)**: Handles arithmetic operations (e.g., addition, subtraction, multiplication, division). It generates the corresponding assembly code for the operation.
* **handleAssignment(const std::vector<std::string> &tokens)**: Handles assignment operations. It generates assembly code to move values between variables and registers.
* **handleCondition(const std::vector<std::string> &tokens)**: Handles conditional jumps (e.g., if statements). It generates the corresponding assembly code based on the condition and label.
* **handleLogical(const std::vector<std::string> &tokens)**: Handles logical operations (e.g., &&, ||, !). It generates assembly code for the operation.
* **handleLabel(const std::string &label)**: Handles labels used for jumps. It generates assembly code for the label.
* **handleFunction(const std::vector<std::string> &tokens)**: Handles function definitions. It generates assembly code for the function label.
* **handleFunctionCall(const std::vector<std::string> &tokens)**: Handles function calls. It generates assembly code for pushing arguments onto the stack, calling the function, and adjusting the stack pointer.

**Public Member Functions**

* **TACToAssemblyConverter(const std::vector<std::string> &tacCode)**: Constructor that initializes the TACToAssemblyConverter with the given TAC code and calls initializeRegisters() to initialize registers.
* **convert()**: Converts the TAC code into assembly instructions. It processes each line of the TAC, splits it into tokens, and determines the type of instruction (arithmetic, assignment, conditional, etc.) to handle accordingly.
* **printAssembly()**: Prints the generated assembly code to the console.

**Usage Example**

*std::vector<std::string> tacCode = {*

*"a = b + c",*

*"if a > b goto label1",*

*"func myFunction",*

*"call myFunction x y"*

*};*

TACToAssemblyConverter converter(tacCode);

converter.convert();

converter.printAssembly();

*Output (example for the given tacCode):*

***MOV R1, b***

***MOV R2, c***

***ADD R1, R2***

***MOV a, R1***

***MOV R1, a***

***CMP R1, b***

***JG label1***

***myFunction:***

***PUSH x***

***PUSH y***

***CALL myFunction***

***ADD SP, 2***

The ThrowError class is a utility class designed to handle and report different types of syntax and semantic errors in a program. It provides static methods to print error messages to the console and terminate the program when an error is encountered. Here's a detailed explanation of its components:

# ThrowError class

## Static Methods:

* + All methods in this class are static, meaning they can be called directly using the class name without needing an instance of the class.

## Error Reporting:

* + Each method corresponds to a specific type of error and prints a descriptive error message that includes details such as the error type, variable name, line number, and additional context if necessary.

## Program Termination:

* + After reporting an error, each method terminates the program using exit(1). This ensures that once an error is detected, the program stops execution immediately.

## Methods and Their Purpose

### unExpectedTokenError:

* + Reports a syntax error when an unexpected token is encountered.
  + Parameters:
    - val: The unexpected token.
    - lineNumber: The line number where the error occurred.
  + Example output:

**Syntax error: unexpected token 'x' at line 10**

### expectedTokenError:

* + Reports a syntax error when an expected token (e.g., an assignment operator) is missing.
  + Parameters:
    - val: The expected token.
    - lineNumber: The line number where the error occurred.
  + Example output:

**Syntax error: expected assignment operator '=' at line 12**

### sementicErrorVariableAlreadyDeclared:

* + Reports a semantic error when a variable is declared multiple times.
  + Parameters:
    - val: The variable name.
    - lineNumber: The line number where the error occurred.
  + Example output:

**Semantic error: Variable 'x' is already declared. Error on line 15**

### sementicErrorFunctionAlreadyDeclared:

* + Reports a semantic error when a function is declared multiple times.
  + Parameters:
    - val: The function name.
    - lineNumber: The line number where the error occurred.
  + Example output:

**Semantic error: Function 'myFunc' is already declared. Error on line 20**

### sementicErrorVarableNotDeclared:

* + Reports a semantic error when a variable is used before being declared.
  + Parameters:
    - val: The variable name.
    - lineNumber: The line number where the error occurred.
  + Example output:

**Semantic error: Variable 'y' is not declared at line 25**

### sementicErrorConstantAssignment:

* + Reports a semantic error when an attempt is made to assign a value to a constant variable.
  + Parameters:
    - val: The constant variable name.
    - lineNumber: The line number where the error occurred.
  + Example output:

**Semantic error: Cannot assign value to constant variable 'PI' at line 30**

### sementicErrorInvalidType:

* + Reports a semantic error when a type mismatch occurs (e.g., appending a wrong type to another type).
  + Parameters:
    - val: The invalid type (or context).
    - lineNumber: The line number where the error occurred.
    - expected: The expected type.
    - actual: The actual type encountered.
  + Example output:

**Semantic error: Invalid type 'int' at line 35**

**Can not append a string in an int**

### Usage

This class is typically used in scenarios where a program (e.g., a compiler or interpreter) needs to detect and report errors during parsing or semantic analysis. Here's an example of how it might be used in a parser:

**void parseVariableDeclaration(string token, int lineNumber) {**

**if (isAlreadyDeclared(token)) {**

**ThrowError::sementicErrorVariableAlreadyDeclared(token, lineNumber);**

**}**

**// Proceed with variable declaration logic...**

**}**

# TACToAssemblyConverter class

The TACToAssemblyConverter class translates Three-Address Code (TAC), which is an intermediate representation used in compilers, into assembly code. It uses a priority queue for register allocation and provides functionality to handle different types of TAC statements like arithmetic operations, assignments, conditional jumps, function calls, and logical operations.

## Class Components

### Private Members

1. **std::vector<std::string> tac**
   * Stores the Three-Address Code lines to be converted.
2. **std::vector<std::string> assembly**
   * Stores the generated assembly code.
3. **std::priority\_queue<std::pair<int, std::string>> registerQueue**
   * Maintains a priority queue of registers based on their last usage, with least-recently-used registers having higher priority.
4. **std::unordered\_map<std::string, int> registerUsage**
   * Maps each register to its last usage count for efficient tracking.
5. **std::unordered\_map<std::string, std::string> variableToRegister**
   * Maps variables to the registers they are currently using.
6. **std::unordered\_map<std::string, std::string> registerToVariable**
   * Maps registers to the variables they currently hold.
7. **int usageCounter**
   * Global counter used to track the usage of registers.

### Private Methods

1. **void initializeRegisters()**
   * Initializes the priority queue with 8 registers (R1 to R8) and sets their initial usage to 0.
2. **std::string getRegister(string &variable)**
   * Allocates a register for a given variable:
     + Reuses the existing register if the variable is already mapped.
     + Selects the least-recently-used register if no existing mapping exists.
     + Updates mappings and usage counters.
3. **std::vector<std::string> split(const std::string &line, char delimiter = ' ')**
   * Splits a line of TAC into tokens based on a specified delimiter (default: space).
4. **void handleArithmetic(const std::vector<std::string> &tokens)**
   * Handles arithmetic TAC instructions (e.g., x = y + z).
   * Generates assembly instructions for addition, subtraction, multiplication, and division.
5. **void handleAssignment(const std::vector<std::string> &tokens)**
   * Handles assignment instructions (e.g., x = y).
   * Moves the value of the operand to a target variable using registers.
6. **void handleCondition(const std::vector<std::string> &tokens)**
   * Handles conditional jumps (e.g., if x < y goto L1).
   * Compares operands and generates appropriate jump instructions (JL, JG, JE, etc.).
7. **void handleLogical(const std::vector<std::string> &tokens)**
   * Handles logical operations like &&, ||, and !.
8. **void handleLabel(const std::string &label)**
   * Handles label definitions by adding them directly to the assembly.
9. **void handleFunction(const std::vector<std::string> &tokens)**
   * Handles function definitions (e.g., func main).
10. **void handleFunctionCall(const std::vector<std::string> &tokens)**
    * Handles function calls:
      + Pushes arguments onto the stack.
      + Calls the function and adjusts the stack pointer.

### Public Members

1. **TACToAssemblyConverter(const std::vector<std::string> &tacCode)**
   * Constructor that initializes the TAC code and registers.
2. **void convert()**
   * Core function that iterates through TAC lines, identifies the type of instruction, and delegates handling to appropriate private methods.
3. **void printAssembly()**
   * Prints the generated assembly code.

**Usage Example**

*#include <iostream>*

*#include <vector>*

*#include <string>*

*#include "TACToAssemblyConverter.h"*

*int main() {*

*std::vector<std::string> tacCode = {*

*"x = y + z",*

*"if x < 10 goto L1",*

*"L1:",*

*"func main",*

*"call foo x y",*

*"z = x \* y"*

*};*

*TACToAssemblyConverter converter(tacCode);*

*converter.convert();*

*converter.printAssembly();*

*return 0;*

*}*

**Example Output**

For the given TAC:

*x = y + z*

*if x < 10 goto L1*

*L1:*

*func main*

*call foo x y*

*z = x \* y*

*Generated assembly:*

*plaintext*

*Copy code*

*MOV R1, y*

*ADD R1, z*

*MOV x, R1*

*MOV R2, x*

*CMP R2, 10*

*JL L1*

*L1:*

*main:*

*PUSH x*

*PUSH y*

*CALL foo*

*ADD SP, 2*

*MOV R3, x*

*MUL R3, y*

*MOV z, R3*

### Key Features

1. **Efficient Register Allocation:**
   * Uses a priority queue to ensure least-recently-used registers are reused.
2. **Modular Design:**
   * Separate methods handle different TAC instruction types for better readability and extensibility.
3. **Versatile:**
   * Supports arithmetic, logical, assignment, conditional jumps, labels, and function handling.
4. **Scalable:**
   * Can be extended to support more TAC constructs and assembly instructions.

# Extra Features Documentation

## 1. Single-Line and Multi-Line Comments

* **Explanation**: The class can ignore comments in the three-address code (TAC). Single-line comments start with //, and multi-line comments are enclosed in /\* ... \*/. During the parsing phase, these lines are skipped, ensuring they do not interfere with assembly generation.
* **Implementation**: Comments are identified and ignored during the split function or the main conversion loop.
* **Example**:

tac

Copy code

// This is a single-line comment

/\* This is a

multi-line comment \*/

a = b + c;

## 2. Additional Operators

* **Explanation**: The class supports additional operators such as unary operators (-, !).
* **Implementation**: These operators are handled in the respective processing functions (e.g., handleArithmetic, handleLogical) with appropriate assembly instructions.
* **Example**:

a = b && c;

a = b || c;

## 3. Additional Data Types

* **Explanation**: New data types such as string, bool, char, and float are supported in the TAC and assembly conversion process.
  + **String**: Handled with memory allocation for strings.
  + **Bool**: Converted into integers (1 for true, 0 for false).
  + **Char**: Treated as single-byte variables.
  + **Float**: Supported using specific floating-point registers and operations.
* **Implementation**: Data type-specific handling is implemented where necessary during register allocation and instruction generation.
* **Example**:

*string name = "John";*

*bool isValid = true;*

*char grade = 'A';*

*float pi = 3.14;*

## 4. Functions

* **Explanation**: Function definitions and calls are supported. Functions can have parameters and return values. The system handles the stack for saving function arguments and managing the return address.
* **Implementation**:
  + Function definitions are marked with func name:.
  + Function calls are handled using the CALL instruction and PUSH for parameters.
* **Example**:

*func add:*

*return a + b;*

*call add, x, y;*

## Constant Variables

* **Explanation**: Constant variables are immutable and cannot be reassigned after initialization. The system ensures such variables are not modified during TAC or assembly processing.
* **Implementation**: Constants are tracked, and attempts to reassign them generate an error or warning.
* **Example**:

const pi = 3.14;

pi = 3.15; // Error

## Assembly Generation Using LRU Algorithm for Register Allocation

* **Explanation**: Registers are allocated using a **Least Recently Used (LRU)** strategy. The register that has been used the least recently is prioritized for allocation, improving efficiency and avoiding unnecessary memory spillovers.
* **Implementation**:
  + A priority queue tracks the usage of each register.
  + When a register is needed, the least recently used one is selected.
  + Registers are updated with their usage counters to reflect recent activity.
* **Example**: The getRegister function in the code manages the LRU allocation process dynamically during TAC-to-assembly translation.