<https://github.com/fakubwoy/C-Compiler-proto->

**Lexical Analysis Phase**

**Purpose of Lexical Analysis**

Lexical analysis is the first phase of a compiler, where the raw source code (a sequence of characters) is transformed into a sequence of **tokens**. Each token represents a basic unit of meaning in the programming language, such as a variable name, an operator (+, -), or a number (5, 10). These tokens act as building blocks for the subsequent parsing phase.

**What is a Token?**

A token consists of:

1. **Type** – The category the token belongs to, such as an INTEGER, PLUS (for +), or IDENTIFIER (for variable names).
2. **Value** – The actual string or character representation from the source code, like "5", "x", or "+".

Each token is an instance of the Token struct defined in the code:

struct Token {  
 TokenType type;  
 std::string value;  
 Token(TokenType t, const std::string& v) : type(t), value(v) {}  
};

* **TokenType** is an enumeration that defines the categories of tokens, like:

enum class TokenType {  
 INTEGER,IDENTIFIER,PLUS,MINUS,MULTIPLY,DIVIDE,LPAREN,RPAREN,SEMICOLON,ASSIGN,EOF\_TOKEN  
};

**Lexer Class: The Core of Lexical Analysis**

**Class Structure:**

The **Lexer** class is responsible for reading through the source code and producing tokens. It operates character by character, identifying sequences of characters that match predefined patterns (such as numbers or identifiers).

class Lexer {  
private:  
 std::string input; //The pre-processed code  
 size\_t position; //Current position in input string  
public:  
 explicit Lexer(const std::string& source) : input(source), position(0) {}  
 Token getNextToken();

};

* **input**: This string contains the entire source code that the lexer will process.
* **position**: This keeps track of the current character being processed within the input string.

**Detailed Breakdown of Lexical Analysis with Lexer**

**1. Skipping Whitespace**

Before attempting to read meaningful tokens, the lexer first skips over any irrelevant characters, like spaces or tabs:

while (position < input.length() && std::isspace(input[position])) {  
 position++;}

* **Purpose:** Whitespace is not important in this phase and is ignored, except when it acts as a separator between tokens.
* **Example:** For the source code x = 5;, the space between x and = will be ignored.

**2. Checking for End of Input**

The lexer checks if it has reached the end of the input string. If it has, it returns an EOF\_TOKEN, signaling the end of the source code.

if (position >= input.length()) {  
 return Token(TokenType::EOF\_TOKEN, "");}

* **Purpose:** This ensures the lexer knows when to stop processing the input. It generates a special EOF\_TOKEN to indicate the end.

**3. Lexing Identifiers and Keywords**

Next, the lexer checks if the current character is an alphabet letter (which could indicate an identifier or keyword):

if (std::isalpha(input[position])) {  
 std::string identifier;  
 while (position < input.length() && (std::isalnum(input[position]) || input[position] == '\_')) {  
 identifier += input[position++];}  
 return Token(TokenType::IDENTIFIER, identifier);}

* **How It Works:**
  + If the current character is alphabetic (std::isalpha), it starts reading the entire sequence of alphanumeric characters (this could form a variable like x, or keywords like return).
  + The loop continues until it encounters a character that is not valid in an identifier (e.g., whitespace, operator, etc.).
  + Once the identifier is captured, it creates an IDENTIFIER token with the name of the variable or keyword.
* **Example:**
  + For x = 5;, this block will generate the token (IDENTIFIER, "x").
  + Similarly, for the line return z;, it will generate the token (IDENTIFIER, "return").
  + NOTE: The tokens generated via the tokenizer aren’t stored using the actual token type but rather the index of the token type in the enum class.

Therefore, the actual tokenization of the statements :

x = 5; will be token (1,x),

return z; will be token (1,return).

**4. Lexing Integers**

If the current character is a digit, the lexer treats it as the start of an integer:

if (std::isdigit(input[position])) {  
 std::string num;  
 while (position < input.length() && std::isdigit(input[position])) {  
 num += input[position++];}  
 return Token(TokenType::INTEGER, num);}

* **How It Works:**
  + If a digit (std::isdigit) is encountered, the lexer collects the entire sequence of digits, constructing a number.
  + Once the number is complete, it generates an INTEGER token with the value of the number.
* **Example:** For x = 5;, this block will generate the token (INTEGER, "5").

**5. Lexing Operators and Symbols**

The lexer also identifies single-character operators and symbols like +, -, \*, =, (, ), etc.:

switch (input[position]) {  
 case '+': position++; return Token(TokenType::PLUS, "+");  
 case '-': position++; return Token(TokenType::MINUS, "-");  
 case '\*': position++; return Token(TokenType::MULTIPLY, "\*");  
 case '/': position++; return Token(TokenType::DIVIDE, "/");  
 case '(': position++; return Token(TokenType::LPAREN, "(");  
 case ')': position++; return Token(TokenType::RPAREN, ")");  
 case ';': position++; return Token(TokenType::SEMICOLON, ";");  
 case '=': position++; return Token(TokenType::ASSIGN, "=");  
 default:  
 throw std::runtime\_error("Invalid character encountered: " + std::string(1, input[position]));}}};

* **How It Works:**
  + Depending on the current character, the lexer produces the corresponding token for operators and punctuation.
  + If an unrecognized character is found (like & or ^, which are not defined in this language), the lexer throws an error.
* **Example:** For x = 5;, this part will generate tokens for the =, and ; symbols: (ASSIGN, "=") and (SEMICOLON, ";").

**6. Handling Errors**

If the lexer encounters an invalid character, it throws an exception:

throw std::runtime\_error("Invalid character encountered: " + std::string(1, input[position]));}}};

* **Purpose:** Lexers must handle malformed input. If an unexpected or unsupported character is found, this ensures the lexer halts and informs the user of the issue.

**How Lexical Analysis Fits into Compilation**

* **Primary Role:** Lexical analysis is the **first phase** in the compilation process. It reads the raw source code and converts it into **tokens**, which serve as the input to the **parser** in the next phase.
* **Relation to Parsing:** Without the lexer breaking the code into manageable tokens, the parser would have difficulty understanding the structure of the code. Tokens are the "words" that the parser uses to form "sentences" (statements and expressions).

**Example of Lexical Analysis in Action**

Given this source code:

cpp

Copy code

x = 5;

y = 10;

z = (x + y) \* 2;

return z;

The lexer would produce the following tokens:

(IDENTIFIER, "x") (ASSIGN, "=") (INTEGER, "5") (SEMICOLON, ";")

(IDENTIFIER, "y") (ASSIGN, "=") (INTEGER, "10") (SEMICOLON, ";")

(IDENTIFIER, "z") (ASSIGN, "=") (LPAREN, "(") (IDENTIFIER, "x") (PLUS, "+") (IDENTIFIER, "y") (RPAREN, ")") (MULTIPLY, "\*") (INTEGER, "2") (SEMICOLON, ";")

(IDENTIFIER, "return") (IDENTIFIER, "z") (SEMICOLON, ";")

**Summary of the Lexical Analysis Phase**

* **Goal:** To convert raw source code into **tokens** that will be consumed by the parser.
* **Implementation in this Code:**
  + The **Lexer** class processes the input string, identifying keywords, variable names, numbers, and operators.
  + It generates tokens that are used by the next phase (parsing) to construct the program's structure.

**Syntax Analysis Phase**

**Purpose of Syntax Analysis**

The primary goal of syntax analysis is to check whether the tokens follow the grammatical rules of the programming language. It translates the flat sequence of tokens into a hierarchical structure, the **Abstract Syntax Tree (AST)**, which expresses the program's structure.

The AST is a tree where:

* **Nodes** represent language constructs, such as expressions, statements, and operations.
* **Edges** represent the relationships between those constructs, like which operands are involved in an operation.

**Parser Class: The Heart of Syntax Analysis**

In the code, the **Parser** class is responsible for building the AST by parsing the tokens produced by the lexer. It implements recursive descent parsing, which uses methods that recursively call each other to break down complex expressions into simpler components.

Let's break down the **Parser** class and its core methods.

**1. Class Structure**

class Parser {  
private:  
 Lexer lexer;  
 Token currentToken;  
   
 void eat(TokenType tokenType);  
 std::unique\_ptr<ASTNode> factor();  
 std::unique\_ptr<ASTNode> term();  
 std::unique\_ptr<ASTNode> expr();  
 std::unique\_ptr<ASTNode> assignment();  
 std::unique\_ptr<ASTNode> statement();  
public:  
 explicit Parser(const std::string& input);  
 std::vector<std::unique\_ptr<ASTNode>> parse();  
};

* **lexer**: The lexer instance that produces tokens from the source code.
* **currentToken**: The token currently being processed.
* **Recursive Functions**: Methods like expr(), term(), factor() break down expressions recursively.

**2. How the Parser Works**

void eat(TokenType tokenType) {  
 if (currentToken.type == tokenType) {  
 currentToken = lexer.getNextToken();  
 } else {  
 throw std::runtime\_error("Unexpected token: expected " + std::to\_string(static\_cast<int>(tokenType)) +  
 ", got " + std::to\_string(static\_cast<int>(currentToken.type)));}}

* **Purpose:** The eat() function consumes the current token if it matches the expected token type, and then advances to the next token by calling lexer.getNextToken().
* **Role in Parsing:** This is a key function for progressing through the token stream. If a mismatch occurs (e.g., expecting a semicolon but finding an identifier), the parser throws an error, signaling a syntax error.
* **Example:** When parsing x = 5;, after reading the IDENTIFIER token (x), the parser expects an ASSIGN token (=), which will be eaten.

**Recursive Descent Parsing**

The parser uses recursive descent to parse expressions and statements. It breaks down the code into manageable pieces using recursive functions like factor(), term(), expr(), and statement().

**Basic Structure of Recursive Descent Parsing**

Each recursive method typically handles a subset of grammar rules. For example, the parser has distinct functions for:

* **Parsing Numbers or Variables (factor())**
* **Handling Multiplicative Operations (term())**
* **Handling Additive Operations (expr())**

**3. Parsing Expressions**

In the code, an expression could involve operations like addition, subtraction, multiplication, or division. Let's walk through how expressions are parsed and turned into an AST.

**factor(): Parsing Numbers or Variables**

std::unique\_ptr<ASTNode> factor() {  
 if (currentToken.type == TokenType::INTEGER) {  
 int value = std::stoi(currentToken.value);  
 eat(TokenType::INTEGER);  
 return std::make\_unique<IntegerNode>(value);  
 } else if (currentToken.type == TokenType::IDENTIFIER) {  
 std::string name = currentToken.value;  
 eat(TokenType::IDENTIFIER);  
 return std::make\_unique<VariableNode>(name);  
 } else if (currentToken.type == TokenType::LPAREN) {  
 eat(TokenType::LPAREN);  
 auto node = expr();  
 eat(TokenType::RPAREN);  
 return node;}  
 throw std::runtime\_error("Invalid factor");}

* **Purpose:** This function parses basic components of expressions, such as numbers (INTEGER tokens) or variables (IDENTIFIER tokens).
  + If an INTEGER token is encountered, it creates an IntegerNode.
  + If an IDENTIFIER token is encountered, it creates a VariableNode.
  + If it sees a left parenthesis ((), it recursively parses the expression inside the parentheses by calling expr().
* **Role in AST:** factor() forms the leaves of the AST, where IntegerNode and VariableNode represent constants and variable references, respectively.

**Example:**

For x = 5;, factor() will first process 5 as an INTEGER, creating an IntegerNode with value 5.

**term(): Handling Multiplication and Division**

std::unique\_ptr<ASTNode> term() {  
 auto node = factor();  
 while (currentToken.type == TokenType::MULTIPLY || currentToken.type == TokenType::DIVIDE) {  
 Token op = currentToken;  
 if (op.type == TokenType::MULTIPLY) {  
 eat(TokenType::MULTIPLY);  
 } else if (op.type == TokenType::DIVIDE) {  
 eat(TokenType::DIVIDE);}  
 node = std::make\_unique<BinaryOpNode>(std::move(node), factor(), op.type);}  
 return node;}

* **Purpose:** This method handles **multiplication** and **division** operations, which have higher precedence than addition and subtraction.
  + It first calls factor() to parse a number or variable.
  + If it encounters a MULTIPLY (\*) or DIVIDE (/) token, it recursively builds a BinaryOpNode for the operation and its operands.
* **Role in AST:** It creates nodes for binary operations (multiplication and division) and attaches IntegerNode or VariableNode as children. The resulting node represents a binary operation in the AST.

**Example:**

For z = (x + y) \* 2;, the multiplication operation (\* 2) is handled by term(). It constructs a BinaryOpNode with the left-hand side being the result of (x + y) and the right-hand side being 2.

**expr(): Handling Addition and Subtraction**

std::unique\_ptr<ASTNode> expr() {  
 auto node = term();  
 while (currentToken.type == TokenType::PLUS || currentToken.type == TokenType::MINUS) {  
 Token op = currentToken;  
 if (op.type == TokenType::PLUS) {  
 eat(TokenType::PLUS);  
 } else if (op.type == TokenType::MINUS) {  
 eat(TokenType::MINUS);}  
 node = std::make\_unique<BinaryOpNode>(std::move(node), term(), op.type);}  
 return node;}

* **Purpose:** This method handles **addition** and **subtraction**, which have lower precedence than multiplication and division.
  + It calls term() to parse the higher-precedence operations first.
  + If it encounters a PLUS (+) or MINUS (-) token, it creates a BinaryOpNode that combines the current node with the next term (the right operand).
* **Role in AST:** It builds the tree for binary addition and subtraction operations, where the left and right subtrees represent the operands.

**Example:**

For z = (x + y) \* 2;, the addition (x + y) is handled by expr(), creating a BinaryOpNode for the + operator.

**4. Parsing Assignments**

**assignment(): Handling Variable Assignments**

std::unique\_ptr<ASTNode> assignment() {  
 if (currentToken.type == TokenType::IDENTIFIER) {  
 std::string varName = currentToken.value;  
 eat(TokenType::IDENTIFIER);  
 eat(TokenType::ASSIGN);  
 auto exprNode = expr();  
 return std::make\_unique<AssignmentNode>(varName, std::move(exprNode));}  
 return expr();}

* **Purpose:** This method handles assignment statements (e.g., x = 5;).
  + It expects an identifier (a variable name), followed by an assignment operator (=), and then an expression on the right-hand side.
  + It creates an AssignmentNode, which represents the assignment operation, with the left-hand side being the variable and the right-hand side being the result of the expression.
* **Role in AST:** The AssignmentNode becomes a parent node in the AST, representing an assignment statement. The left child is the variable, and the right child is the expression being assigned to the variable.

**Example:**

For x = 5;, assignment() will create an AssignmentNode where the left side is a VariableNode (x) and the right side is an IntegerNode (5).

**5. Parsing Statements**

**statement(): Handling Full Statements**

std::unique\_ptr<ASTNode> statement() {  
 if (currentToken.type == TokenType::IDENTIFIER && currentToken.value == "return") {  
 eat(TokenType::IDENTIFIER);  
 auto returnExpr = expr();  
 eat(TokenType::SEMICOLON);  
 return std::make\_unique<ReturnNode>(std::move(returnExpr));  
 } else {  
 auto node = assignment();  
 eat(TokenType::SEMICOLON);  
 return node;}}

* **Purpose:** This method parses both assignment and return statements.
  + If the statement begins with return, it creates a ReturnNode that contains the expression to return.
  + Otherwise, it treats the statement as an assignment.
* **Role in AST:** It handles top-level statements in the program, producing nodes like ReturnNode or AssignmentNode.

**Example:**

For return z;, statement() creates a ReturnNode whose child is a VariableNode representing z.

**6. Building the Entire AST: parse()**

std::vector<std::unique\_ptr<ASTNode>> parse() {  
 std::vector<std::unique\_ptr<ASTNode>> statements;  
 while (currentToken.type != TokenType::EOF\_TOKEN) {  
 statements.push\_back(statement());}  
 return statements;}};

* **Purpose:** The parse() method orchestrates the entire parsing process. It repeatedly calls statement() until it reaches the end of the input, building a list of AST nodes (one per statement).
* **Role in AST:** It returns a full list of AST nodes representing the entire program.

**Example:**

For the input code:

x = 5;

y = 10;

z = (x + y) \* 2;

return z;

The parser produces an AST like this:

AssignmentNode: x = 5

AssignmentNode: y = 10

AssignmentNode: z = (x + y) \* 2

ReturnNode: return z

**How Syntax Analysis Fits into Compilation**

* **Primary Role:** Syntax analysis checks whether the code follows the syntactic rules of the language and builds an **AST** that represents the program's structure.
* **Relation to Other Phases:**
  + **Lexical Analysis** provides tokens for the parser to consume.
  + **Semantic Analysis** uses the AST to check for semantic correctness, like variable declarations.

**Summary of the Syntax Analysis Phase**

* **Goal:** To convert the token stream into an **Abstract Syntax Tree (AST)**, which represents the syntactic structure of the program.
* **Implementation in this Code:**
  + The **Parser** class breaks down expressions and statements using recursive descent parsing, producing nodes like AssignmentNode, BinaryOpNode, and ReturnNode.
  + The resulting AST serves as the input for semantic analysis and code generation.

**Semantic Analysis Phase**

**Purpose of Semantic Analysis**

While the parser ensures the program's syntax is correct, the **semantic analyzer** ensures the **meaning** of the program makes sense. It validates:

* Whether variables are defined before they are used.
* Type correctness in expressions and assignments.
* That operations are performed on valid data types (e.g., you can’t divide a string by a number).

The output of this phase is usually an AST with semantic checks completed, making it ready for code generation.

**Semantic Analysis in this Code**

In the code, the **SemanticAnalyzer** class is responsible for validating the semantic rules of the language. This class traverses the AST generated by the parser and checks for any semantic errors, such as the use of undefined variables.

**Class Structure:**

class SemanticAnalyzer {  
private:  
 std::unordered\_map<std::string, bool> variables;  
public:  
 void analyze(const ASTNode\* node);  
};

**1. variables Map**

The variables map keeps track of variables that have been declared or assigned a value:

std::unordered\_map<std::string, bool> variables;

* **Purpose:** This map acts as a symbol table, which stores information about each variable encountered in the program. The key is the variable name (a std::string), and the value is a boolean indicating whether the variable has been assigned a value (i.e., declared).
* **Example:** If the program has the statement x = 5;, the symbol table will store {"x": true}, indicating that x is now a valid variable.

**2. analyze() Method**

The analyze() method is the heart of semantic analysis. It performs a **recursive traversal** of the AST, checking for semantic errors.

void analyze(const ASTNode\* node) {  
 if (node->type == NodeType::INTEGER) {  
 } else if (node->type == NodeType::VARIABLE) {  
 const auto\* varNode = static\_cast<const VariableNode\*>(node);  
 if (variables.find(varNode->name) == variables.end()) {  
 throw std::runtime\_error("Undefined variable: " + varNode->name);}  
 } else if (node->type == NodeType::BINARY\_OP) {  
 const auto\* binOp = static\_cast<const BinaryOpNode\*>(node);  
 analyze(binOp->left.get());  
 analyze(binOp->right.get());  
 } else if (node->type == NodeType::ASSIGNMENT) {  
 const auto\* assignNode = static\_cast<const AssignmentNode\*>(node);  
 variables[assignNode->variable] = true;  
 analyze(assignNode->expression.get());  
 } else if (node->type == NodeType::RETURN) {  
 const auto\* returnNode = static\_cast<const ReturnNode\*>(node);  
 analyze(returnNode->expression.get());}}};

**3. Detailed Breakdown of analyze()**

**Integer Nodes (NodeType::INTEGER)**

if (node->type == NodeType::INTEGER) {  
 // Integer nodes are always valid  
}

* **Purpose:** Integer constants (e.g., 5, 10) are always valid semantically, so no further checks are needed.
* **Role in Semantic Analysis:** These nodes represent constants in the AST, and no extra semantic checking is necessary.

**Variable Nodes (NodeType::VARIABLE)**

else if (node->type == NodeType::VARIABLE) {  
 const auto\* varNode = static\_cast<const VariableNode\*>(node);  
 if (variables.find(varNode->name) == variables.end()) {  
 throw std::runtime\_error("Undefined variable: " + varNode->name);}  
}

* **Purpose:** This section checks if the variable being used has been declared earlier in the program.
  + It looks up the variable name in the variables map.
  + If the variable is not found in the symbol table (i.e., variables.find(varNode->name) == variables.end()), it throws an error for using an **undefined variable**.
* **Role in Semantic Analysis:** Ensures that all variables used in expressions are properly declared.

**Example:**

For the code:

z = (x + y) \* 2;

If neither x nor y has been assigned a value earlier in the code, the semantic analyzer will raise an error, e.g., "Undefined variable: x".

**Binary Operation Nodes (NodeType::BINARY\_OP)**

else if (node->type == NodeType::BINARY\_OP) {  
 const auto\* binOp = static\_cast<const BinaryOpNode\*>(node);  
 analyze(binOp->left.get());  
 analyze(binOp->right.get());  
}

* **Purpose:** This section recursively checks the left and right operands of binary operations (e.g., +, -, \*, /).
  + It calls analyze() on both the left and right operands, ensuring that they are semantically correct (e.g., they involve valid variables or constants).
* **Role in Semantic Analysis:** Ensures that both operands of a binary operation are valid expressions.

**Example:**

For the expression (x + y) \* 2, it checks:

1. Whether x and y are valid variables.
2. Whether 2 is a valid constant.

**Assignment Nodes (NodeType::ASSIGNMENT)**

else if (node->type == NodeType::ASSIGNMENT) {  
 const auto\* assignNode = static\_cast<const AssignmentNode\*>(node);  
 variables[assignNode->variable] = true;  
 analyze(assignNode->expression.get());  
}

* **Purpose:** This section handles **variable assignments**, ensuring that the right-hand side of the assignment is valid.
  + It adds the variable on the left-hand side to the variables map, marking it as defined.
  + It recursively calls analyze() on the expression on the right-hand side to ensure its validity.
* **Role in Semantic Analysis:** Tracks variable declarations and ensures the right-hand side of an assignment is a valid expression.

**Example:**

For the assignment z = (x + y) \* 2;:

1. It marks z as a declared variable.
2. It recursively checks the expression (x + y) \* 2.

**Return Nodes (NodeType::RETURN)**

else if (node->type == NodeType::RETURN) {  
 const auto\* returnNode = static\_cast<const ReturnNode\*>(node);  
 analyze(returnNode->expression.get());}}};

* **Purpose:** This section validates **return statements**, ensuring that the expression being returned is valid.
  + It recursively calls analyze() on the expression to ensure it is semantically correct.
* **Role in Semantic Analysis:** Checks that the return value is valid and involves only declared variables or constants.

**Example:**

For return z;, the analyzer ensures that z is a valid, declared variable.

**How Semantic Analysis Fits into Compilation**

* **Primary Role:** Semantic analysis ensures that the program **makes sense** logically and complies with the rules of the programming language. It catches errors that aren't apparent in the syntax alone, such as using undeclared variables.
* **Relation to Other Phases:**
  + **Syntax Analysis** ensures that the structure of the code is valid, but it does not check whether the code is meaningful.
  + **Semantic Analysis** ensures the correctness of variable usage, operations, and expressions. Once this is done, the AST is ready for **Intermediate Code Generation**.

**Example of Semantic Analysis in Action**

Given the following code:

x = 5;

y = 10;

z = (x + y) \* 2;

return z;

Here’s what the SemanticAnalyzer would do:

1. **x = 5;**
   * It marks x as declared and assigns the value 5.
2. **y = 10;**
   * It marks y as declared and assigns the value 10.
3. **z = (x + y) \* 2;**
   * It checks whether x and y have been declared (they have).
   * It assigns the result of (x + y) \* 2 to z.
4. **return z;**
   * It checks that z has been declared (it has) and returns its value.

**Summary of the Semantic Analysis Phase**

* **Goal:** To ensure that the program’s **meaning** is correct, ensuring that all variables are declared before use, and that operations are performed on valid expressions.
* **Implementation in this Code:**
  + The **SemanticAnalyzer** class recursively traverses the AST, checking that all variables are declared and that expressions are semantically valid.
  + The **variables** map acts as a symbol table, tracking declared variables.

**Intermediate Code Generation Phase**

**Purpose of Intermediate Code Generation**

The goal of this phase is to convert the AST into an **intermediate form** that is easier to optimize and generate machine code from. The intermediate form typically abstracts away from the complexities of hardware-specific machine code, making it a crucial step for efficient optimization and code generation.

In this case, the intermediate form is **Three-Address Code (TAC)**, which consists of simple instructions where each instruction typically involves a maximum of three operands.

**Three-Address Code (TAC)**

TAC is a form of intermediate representation where each instruction has the general form:

arduino

Copy code

result = operand1 operator operand2

* **Example:** For the expression x = (a + b) \* 2;, a typical TAC might look like:

t1 = a + b

t2 = t1 \* 2

x = t2

**Intermediate Code Generation in this Code**

In the code, the **IntermediateCodeGenerator** class is responsible for generating TAC from the AST produced by the parser. It recursively traverses the AST and generates TAC instructions.

**Class Structure:**

class IntermediateCodeGenerator {  
private:  
 int tempCounter = 0;  
 std::string generateTemp();  
public:  
 std::vector<std::string> generateTAC(const ASTNode\* node);  
private:  
 std::string generateTACHelper(const ASTNode\* node, std::vector<std::string>& code);  
};

* **tempCounter**: Keeps track of temporary variables used in TAC.
* **generateTemp()**: Generates a new temporary variable (t1, t2, etc.) to store intermediate results.
* **generateTAC()**: The main public method that generates the complete TAC from the AST.
* **generateTACHelper()**: The recursive function that generates TAC instructions for each node in the AST.

**1. Temporary Variables and TAC Structure**

**Generating Temporary Variables (generateTemp())**

std::string generateTemp() {  
 return "t" + std::to\_string(++tempCounter);}

* **Purpose:** Each intermediate result of an expression or operation is stored in a **temporary variable** (e.g., t1, t2).
  + Temporary variables are crucial because they hold intermediate computations before they are assigned to actual variables.
  + Each call to generateTemp() generates a new temporary variable (e.g., t1, t2, t3), ensuring no name conflicts.

**Example:**

For the expression (x + y) \* 2, the code generator might first compute t1 = x + y and then compute t2 = t1 \* 2. Both intermediate results (t1 and t2) are stored in temporary variables.

**2. generateTAC() – Generating the Three-Address Code**

This is the main public function that takes an AST node as input and produces the corresponding TAC code as a vector of strings.

std::vector<std::string> generateTAC(const ASTNode\* node) {  
 std::vector<std::string> code;  
 std::string result = generateTACHelper(node, code);  
 if (!result.empty()) {  
 code.push\_back(result);}  
 return code;}

* **Purpose:** This method generates the complete TAC for a given AST. It calls generateTACHelper() to recursively traverse the AST and build the TAC.
* **Returns:** A list of TAC instructions in the form of strings, where each string represents one three-address instruction.

**3. generateTACHelper() – Recursively Generating TAC**

The core of the intermediate code generation is handled by this recursive function, which walks the AST and generates TAC for each node type.

std::string generateTACHelper(const ASTNode\* node, std::vector<std::string>& code) {  
 if (node->type == NodeType::INTEGER) {  
 const auto\* intNode = static\_cast<const IntegerNode\*>(node);  
 return std::to\_string(intNode->value);  
 } else if (node->type == NodeType::VARIABLE) {  
 const auto\* varNode = static\_cast<const VariableNode\*>(node);  
 return varNode->name;  
 } else if (node->type == NodeType::BINARY\_OP) {  
 const auto\* binOpNode = static\_cast<const BinaryOpNode\*>(node);  
 std::string leftTemp = generateTACHelper(binOpNode->left.get(), code);  
 std::string rightTemp = generateTACHelper(binOpNode->right.get(), code);  
 std::string op;  
 switch (binOpNode->op) {  
 case TokenType::PLUS: op = "+"; break;  
 case TokenType::MINUS: op = "-"; break;  
 case TokenType::MULTIPLY: op = "\*"; break;  
 case TokenType::DIVIDE: op = "/"; break;  
 default: throw std::runtime\_error("Unknown operator");}  
 std::string result = generateTemp();  
 code.push\_back(result + " = " + leftTemp + " " + op + " " + rightTemp);  
 return result;  
 } else if (node->type == NodeType::ASSIGNMENT) {  
 const auto\* assignNode = static\_cast<const AssignmentNode\*>(node);  
 std::string exprTemp = generateTACHelper(assignNode->expression.get(), code);  
 code.push\_back(assignNode->variable + " = " + exprTemp);  
 return "";  
 } else if (node->type == NodeType::RETURN) {  
 const auto\* returnNode = static\_cast<const ReturnNode\*>(node);  
 std::string exprTemp = generateTACHelper(returnNode->expression.get(), code);  
 code.push\_back("return " + exprTemp);  
 return "";}  
 throw std::runtime\_error("Unknown node type");}};

**Detailed Breakdown of generateTACHelper()**

**Handling Integer Nodes (NodeType::INTEGER)**

if (node->type == NodeType::INTEGER) {  
 const auto\* intNode = static\_cast<const IntegerNode\*>(node);  
 return std::to\_string(intNode->value);  
}

* **Purpose:** For **integer constants**, the value is simply returned as a string (e.g., 5, 10). Since these are constants, they do not generate new TAC instructions.
* **Role in TAC:** Integers act as operands for TAC instructions.

**Example:**

For the integer 5, the function would return the string "5" to be used in a TAC expression.

**Handling Variable Nodes (NodeType::VARIABLE)**

else if (node->type == NodeType::VARIABLE) {  
 const auto\* varNode = static\_cast<const VariableNode\*>(node);  
 return varNode->name;  
}

* **Purpose:** For **variables**, their name (e.g., x, y, z) is returned. These variables are used as operands in TAC.
* **Role in TAC:** Variables act as operands in three-address instructions.

**Example:**

For the variable x, this function would return the string "x", which would be used in TAC instructions like t1 = x + y.

**Handling Binary Operations (NodeType::BINARY\_OP)**

else if (node->type == NodeType::BINARY\_OP) {  
 const auto\* binOpNode = static\_cast<const BinaryOpNode\*>(node);  
 std::string leftTemp = generateTACHelper(binOpNode->left.get(), code);  
 std::string rightTemp = generateTACHelper(binOpNode->right.get(), code);  
 std::string op;  
 switch (binOpNode->op) {  
 case TokenType::PLUS: op = "+"; break;  
 case TokenType::MINUS: op = "-"; break;  
 case TokenType::MULTIPLY: op = "\*"; break;  
 case TokenType::DIVIDE: op = "/"; break;  
 default: throw std::runtime\_error("Unknown operator");}  
 std::string result = generateTemp();  
 code.push\_back(result + " = " + leftTemp + " " + op + " " + rightTemp);  
 return result;  
}

* **Purpose:** For **binary operations** (like addition, subtraction, multiplication, or division), the function:
  1. Recursively generates TAC for the left and right operands.
  2. Chooses the correct operator (+, -, \*, /).
  3. Allocates a temporary variable to store the result of the operation.
  4. Generates a TAC instruction in the form result = operand1 operator operand2 and stores it in the code vector.
  5. Returns the temporary variable that stores the result of the operation.
* **Role in TAC:** Binary operations are a core part of expressions, and this method ensures they are converted into simple TAC instructions.

**Example:**

For the expression x + y, the following TAC might be generated:

t1 = x + y

Where t1 is a temporary variable holding the result of x + y.

**Handling Assignments (NodeType::ASSIGNMENT)**

else if (node->type == NodeType::ASSIGNMENT) {  
 const auto\* assignNode = static\_cast<const AssignmentNode\*>(node);  
 std::string exprTemp = generateTACHelper(assignNode->expression.get(), code);  
 code.push\_back(assignNode->variable + " = " + exprTemp);  
 return "";  
}

* **Purpose:** For **assignment statements** (e.g., x = 5), the function:
  1. Recursively generates TAC for the right-hand side expression.
  2. Generates an assignment instruction in the form variable = expression.
  3. Appends the instruction to the code vector.
* **Role in TAC:** This handles assignments by generating a simple TAC instruction for assigning the result of an expression to a variable.

**Example:**

For z = (x + y) \* 2;, the following TAC might be generated:

t1 = x + y

t2 = t1 \* 2

z = t2

**Handling Return Statements (NodeType::RETURN)**

else if (node->type == NodeType::RETURN) {  
 const auto\* returnNode = static\_cast<const ReturnNode\*>(node);  
 std::string exprTemp = generateTACHelper(returnNode->expression.get(), code);  
 code.push\_back("return " + exprTemp);  
 return "";}

* **Purpose:** For **return statements** (e.g., return z;), the function:
  1. Recursively generates TAC for the expression being returned.
  2. Generates a return instruction in the form return expression.
  3. Appends the return instruction to the code vector.
* **Role in TAC:** This ensures that the return statement is translated into the appropriate TAC instruction.

**Example:**

For return z;, the following TAC might be generated:

return z

**How Intermediate Code Generation Fits into Compilation**

* **Primary Role:** Intermediate code generation translates the high-level AST into a lower-level, simpler intermediate form like **Three-Address Code (TAC)**, which is easier to optimize and translate into machine code.
* **Relation to Other Phases:**
  + **Syntax and Semantic Analysis** ensure the correctness of the program, but the AST is not suitable for direct translation into machine code.
  + **Intermediate Code Generation** simplifies the program by converting it into a form that is easy to optimize.

**Summary of the Intermediate Code Generation Phase**

* **Goal:** To convert the AST into an **intermediate form** like **Three-Address Code (TAC)**, which is easier to optimize and translate into machine code.
* **Implementation in this Code:**
  + The **IntermediateCodeGenerator** class recursively traverses the AST and generates TAC instructions.
  + Temporary variables are used to store intermediate results of expressions.
  + The output is a list of TAC instructions that represent the program in a simple, three-operand form.

**Code Optimization Phase**

**Purpose of Code Optimization**

The goal of code optimization is to improve the intermediate representation of the program (in this case, TAC) by:

* Reducing redundant instructions.
* Eliminating unnecessary calculations.
* Improving execution efficiency without changing the semantics of the program.

Optimizations can range from simple transformations (like **constant folding**) to more complex optimizations (like **dead code elimination**, **loop unrolling**, or **inlining**). In the code, **constant folding** is the primary optimization being applied.

**Code Optimization in this Code**

The **CodeOptimizer** class is responsible for performing constant folding and other simple optimizations on the TAC.

**Class Structure:**

class CodeOptimizer {  
public:  
 std::vector<std::string> optimize(const std::vector<std::string>& tac);  
private:  
 std::string foldConstants(const std::string& instruction, std::unordered\_map<std::string, std::string>& constantMap);  
 bool isInteger(const std::string& s);  
};

* **optimize()**: The main function that takes the TAC as input and applies optimizations to it.
* **foldConstants()**: A helper function that performs constant folding—this detects constant expressions and simplifies them.
* **isInteger()**: A utility function that checks whether a string represents an integer value.

**1. optimize() – Main Optimization Function**

This function applies optimization to the list of TAC instructions.

std::vector<std::string> optimize(const std::vector<std::string>& tac) {  
 std::vector<std::string> optimizedCode;  
 std::unordered\_map<std::string, std::string> constantMap;  
 for (const auto& instruction : tac) {  
 std::string optimizedInstruction = foldConstants(instruction, constantMap);  
 if (!optimizedInstruction.empty()) {  
 optimizedCode.push\_back(optimizedInstruction);}}  
 return optimizedCode;}

**How It Works:**

* **Purpose:** It iterates over each TAC instruction, applying optimizations using the foldConstants() function.
  + The function keeps track of constants that are found during optimization using a constantMap, which maps variables to their constant values.
* **Returns:** A vector of optimized TAC instructions.

**Key Components:**

1. **tac**: The input list of three-address code instructions that need optimization.
2. **constantMap**: This map stores variables that have been identified as constants during optimization. It allows for optimizations like replacing variables with their constant values.

**Example:**

For the unoptimized TAC:

t1 = 3 + 2

t2 = t1 \* 4

This could be optimized to:

t1 = 5

t2 = t1 \* 4

**2. Constant Folding with foldConstants()**

Constant folding is a common optimization where constant expressions are evaluated at compile time rather than runtime. For example, an expression like 3 + 2 would be computed at compile time and replaced with 5.

std::string foldConstants(const std::string& instruction, std::unordered\_map<std::string, std::string>& constantMap) {  
 std::istringstream iss(instruction);  
 std::string result, eq, op1, op, op2;  
 iss >> result >> eq;  
 if (eq == "=") {  
 iss >> op1;  
 if (iss >> op >> op2) {  
 op1 = constantMap.count(op1) ? constantMap[op1] : op1;  
 op2 = constantMap.count(op2) ? constantMap[op2] : op2;  
 if (isInteger(op1) && isInteger(op2)) {  
 int val1 = std::stoi(op1);  
 int val2 = std::stoi(op2);  
 int res;  
 if (op == "+") res = val1 + val2;  
 else if (op == "-") res = val1 - val2;  
 else if (op == "\*") res = val1 \* val2;  
 else if (op == "/") res = val1 / val2;  
 else return instruction;  
 constantMap[result] = std::to\_string(res);  
 return result + " = " + std::to\_string(res);}  
 } else {  
 if (isInteger(op1)) {  
 constantMap[result] = op1;  
 } else if (constantMap.count(op1)) {  
 return result + " = " + constantMap[op1];}}  
 } else if (result == "return") {  
 iss >> op1;  
 if (constantMap.count(op1)) {  
 return "return " + constantMap[op1];}}  
 return instruction;}

**How It Works:**

* **Purpose:** The foldConstants() function evaluates constant expressions at compile time, simplifying instructions where possible.
  + It checks if both operands in a binary operation are constants (integers).
  + If both operands are constants, the operation is evaluated immediately, and the result is stored in the constantMap for future use.
  + If a variable is known to hold a constant value (from constantMap), it replaces the variable with the constant.
* **Returns:** The optimized instruction, either with constants folded or unchanged if no optimization was possible.

**Steps in foldConstants():**

1. **Parse the Instruction:** It breaks the instruction into components, such as the result variable, the operator, and the operands.
2. **Check for Constant Expressions:**
   * If both operands are constants (e.g., 3 + 2), it computes the result at compile time.
   * If one or both operands are variables but their values are known to be constants, it substitutes the constants.
3. **Return the Optimized Instruction:** The result of the constant folding is returned, replacing complex expressions with simplified constants when possible.

**Example of Constant Folding:**

Consider the TAC instruction:

t1 = 3 + 2

* The operands 3 and 2 are both constants, so foldConstants() computes the result (5) and returns:

t1 = 5

If another instruction uses t1, such as:

t2 = t1 \* 4

* The optimizer would recognize that t1 is a constant (5), so it simplifies the instruction to:

t2 = 5 \* 4

* This would then be further simplified to:

t2 = 20

**3. isInteger() – Utility Function**

This utility function checks if a string represents a valid integer value.

bool isInteger(const std::string& s) {  
 return !s.empty() && std::find\_if(s.begin(), s.end(), [](unsigned char c) { return !std::isdigit(c); }) == s.end();}};

* **Purpose:** It ensures that only valid integer strings (like "3", "42") are considered for constant folding.
* **How It Works:** It uses the std::isdigit() function to check if all characters in the string are digits.

**How Code Optimization Fits into Compilation**

* **Primary Role:** Code optimization improves the efficiency of the generated TAC by reducing unnecessary operations and computations. This makes the program run faster or use fewer resources.
* **Relation to Other Phases:**
  + **Intermediate Code Generation** creates a representation of the program (TAC) that is easy to optimize.
  + **Code Optimization** simplifies the TAC, making the code more efficient, which will result in better performance once the machine code is generated.

**Example of Code Optimization**

Given the following unoptimized TAC:

t1 = 3 + 2

t2 = t1 \* 4

x = t2

After constant folding, the optimized TAC would look like this:

t1 = 5

t2 = 5 \* 4

t2 = 20

x = 20

This greatly simplifies the original code, reducing unnecessary computations.

**Summary of the Code Optimization Phase**

* **Goal:** To improve the intermediate code (TAC) by simplifying constant expressions and eliminating redundant instructions.
* **Implementation in this Code:**
  + The **CodeOptimizer** class applies constant folding, evaluating constant expressions at compile time.
  + The **constantMap** is used to track which variables hold constant values and substitute them where possible.

**Assembly Code Generation Phase**

**Purpose of Assembly Code Generation**

The goal of assembly code generation is to convert the intermediate representation of the program (such as TAC) into machine-specific assembly code. This phase ensures that the instructions are properly structured to run on actual hardware or a target architecture.

**Assembly Code Generation in this Code**

In the code, the **AssemblyCodeGenerator** class is responsible for converting the optimized TAC into assembly code.

**Class Structure:**

class AssemblyCodeGenerator {  
public:  
 std::vector<std::string> generateAssembly(const std::vector<std::string>& tac);  
};

* **generateAssembly()**: This is the main function responsible for taking the TAC as input and generating corresponding assembly instructions.

**How Assembly Code Generation Works**

Let's walk through the code to see how the assembly instructions are generated from TAC.

**1. The generateAssembly() Function**

This function converts TAC into assembly language, which consists of a series of low-level instructions that directly correspond to machine operations.

std::vector<std::string> generateAssembly(const std::vector<std::string>& tac) {  
assembly.clear();  
variableOffsets.clear();  
allocatedVariables.clear();  
nextOffset = 4;  
assembly.push\_back("section .text");  
assembly.push\_back("global \_start");  
assembly.push\_back("\_start:");  
assembly.push\_back(" push ebp");  
assembly.push\_back(" mov ebp, esp");  
for (const auto& instruction : tac) {  
 std::istringstream iss(instruction);  
 std::string result, eq, op1, op, op2;  
 iss >> result >> eq;  
 if (eq == "=" && !isTemporary(result)) {  
 getVariableOffset(result);}  
 if (iss >> op1) {  
 if (!isTemporary(op1) && !isInteger(op1)) {  
 getVariableOffset(op1);}  
 if (iss >> op >> op2) {  
 if (!isTemporary(op2) && !isInteger(op2)) {  
 getVariableOffset(op2);}}}}  
int stackSize = nextOffset - 4;  
assembly.push\_back(" sub esp, " + std::to\_string(stackSize));  
std::unordered\_set<std::string> usedVariables;  
std::string lastResult;  
for (const auto& instruction : tac) {  
 std::istringstream iss(instruction);  
 std::string result, eq, op1, op, op2;  
 iss >> result >> eq;  
 if (result == "return") {  
 iss >> op1;  
 if (isInteger(op1)) {  
 assembly.push\_back(" mov eax, " + op1);  
 } else if (allocatedVariables.find(op1) != allocatedVariables.end()) {  
 assembly.push\_back(" mov eax, " + getVariableOffset(op1));}  
 break;  
 } else if (eq == "=") {  
 iss >> op1;  
 if (iss >> op >> op2) {  
 if (op == "+") {  
 if (isInteger(op1) && isInteger(op2)) {  
 int value = std::stoi(op1) + std::stoi(op2);  
 assembly.push\_back(" mov eax, " + std::to\_string(value));  
 } else {  
 if (!isInteger(op1)) {  
 assembly.push\_back(" mov eax, " + getVariableOffset(op1));  
 } else {  
 assembly.push\_back(" mov eax, " + op1);}  
 if (isInteger(op2)) {  
 assembly.push\_back(" add eax, " + op2);  
 } else {  
 assembly.push\_back(" add eax, " + getVariableOffset(op2));}}  
 } else if (op == "-") {  
 if (isInteger(op1) && isInteger(op2)) {  
 int value = std::stoi(op1) - std::stoi(op2);  
 assembly.push\_back(" mov eax, " + std::to\_string(value));  
 } else {  
 if (!isInteger(op1)) {  
 assembly.push\_back(" mov eax, " + getVariableOffset(op1));  
 } else {  
 assembly.push\_back(" mov eax, " + op1);}  
 if (isInteger(op2)) {  
 assembly.push\_back(" sub eax, " + op2);  
 } else {  
 assembly.push\_back(" sub eax, " + getVariableOffset(op2));}}  
 } else if (op == "\*") {  
 if (isInteger(op1) && isInteger(op2)) {  
 int value = std::stoi(op1) \* std::stoi(op2);  
 assembly.push\_back(" mov eax, " + std::to\_string(value));  
 } else {  
 if (!isInteger(op1)) {  
 assembly.push\_back(" mov eax, " + getVariableOffset(op1));  
 } else {  
 assembly.push\_back(" mov eax, " + op1);}  
 if (isInteger(op2)) {  
 assembly.push\_back(" imul eax, " + op2);  
 } else {  
 assembly.push\_back(" imul eax, " + getVariableOffset(op2));}}  
 } else if (op == "/") {  
 if (isInteger(op1) && isInteger(op2)) {  
 int value = std::stoi(op1) / std::stoi(op2);  
 assembly.push\_back(" mov eax, " + std::to\_string(value));  
 } else {  
 if (!isInteger(op1)) {  
 assembly.push\_back(" mov eax, " + getVariableOffset(op1));  
 } else {  
 assembly.push\_back(" mov eax, " + op1);}  
 if (isInteger(op2)) {  
 assembly.push\_back(" mov ebx, " + op2);  
 } else {  
 assembly.push\_back(" mov ebx, " + getVariableOffset(op2));}  
 assembly.push\_back(" xor edx, edx");  
 assembly.push\_back(" div ebx");}}  
 } else {  
 if (isInteger(op1)) {  
 assembly.push\_back(" mov eax, " + op1);  
 } else {  
 assembly.push\_back(" mov eax, " + getVariableOffset(op1));}}  
 if (!isTemporary(result)) {  
 usedVariables.insert(result);  
 assembly.push\_back(" mov " + getVariableOffset(result) + ", eax");}  
 lastResult = result;}}  
 assembly.push\_back(" mov esp, ebp");  
 assembly.push\_back(" pop ebp");  
 assembly.push\_back(" ret");  
return assembly;}

**How It Works:**

* **Purpose:** The generateAssembly() function iterates through each TAC instruction and converts it into corresponding assembly instructions.
* **Assembly Instructions:** It uses simple assembly operations such as MOV, ADD, SUB, MUL, DIV, and RET.
  + **MOV:** Move data from one location to another.
  + **ADD/SUB/MUL/DIV:** Perform arithmetic operations.
  + **RET:** Return from a function.
* **Key Concepts:**
  + **Registers:** The assembly instructions often operate on CPU registers (e.g., R0, R1). In this case, R0 is used as a temporary register for storing intermediate results.
  + **Memory Addressing:** Variables are mapped to memory locations or registers in the generated assembly code.

**Example of Assembly Code Generation**

Let's break down how specific TAC instructions are translated into assembly instructions:

**1. Simple Assignment**

Consider the TAC:

x = 5

This is a simple assignment, where x is assigned the value 5. The corresponding assembly code would be:

MOV x, 5

This means that the value 5 is moved into the memory or register associated with x.

**2. Binary Operation**

Consider the TAC:

t1 = x + y

This is a binary operation, where the sum of x and y is stored in the temporary variable t1. The corresponding assembly code would be:

MOV R0, x ; Load the value of x into register R0

ADD R0, y ; Add the value of y to the value in R0

MOV t1, R0 ; Store the result in t1

* The first instruction moves the value of x into the register R0.
* The second instruction adds the value of y to R0.
* Finally, the result is moved from R0 to t1.

**3. Return Statement**

Consider the TAC:

return z

This is a return statement, where the function returns the value of z. The corresponding assembly code would be:

MOV R0, z ; Load the value of z into register R0

RET ; Return from the function

* The value of z is loaded into register R0.
* The RET instruction causes the function to return, with R0 holding the return value.

**How Assembly Code Generation Fits into Compilation**

* **Primary Role:** Assembly code generation translates the intermediate representation (TAC) into machine-specific assembly instructions. These assembly instructions can be assembled into machine code that the target processor can execute.
* **Relation to Other Phases:**
  + **Intermediate Code Generation** provides a simplified representation of the program that is easier to translate into assembly code.
  + **Code Optimization** improves the efficiency of the TAC, making the generated assembly code more efficient.
  + **Assembly Code Generation** is the final step before machine code, converting TAC into low-level instructions that correspond directly to machine operations.

**Registers and Memory in Assembly Code**

Assembly code generation relies heavily on **registers** (small, fast memory locations in the CPU) to store temporary values. In the code:

* **R0** is used as a temporary register to hold intermediate results.
* Variables and temporary results (like t1, t2, etc.) are mapped to memory locations or registers.

Efficient use of registers is important for performance, as accessing registers is faster than accessing main memory. The assembly code generator tries to ensure that intermediate results are kept in registers as much as possible to avoid expensive memory accesses.

**Instruction Types in Assembly**

The assembly instructions generated by the code can be categorized into several types:

1. **Data Movement Instructions:**
   * **MOV**: This instruction moves data from one location to another (e.g., from memory to a register, or between registers).
   * Example: MOV R0, x moves the value of x into the register R0.
2. **Arithmetic Instructions:**
   * **ADD, SUB, MUL, DIV**: These instructions perform arithmetic operations on registers or memory locations.
   * Example: ADD R0, y adds the value of y to the value in register R0.
3. **Control Flow Instructions:**
   * **RET**: This instruction returns from a function, signaling the end of the function’s execution.
   * Example: RET causes the function to return to the calling code.

**Summary of the Assembly Code Generation Phase**

* **Goal:** To convert the intermediate representation (TAC) into **assembly language instructions** that can be assembled into machine code for execution on the target platform.
* **Implementation in this Code:**
  + The **AssemblyCodeGenerator** class converts each TAC instruction into the appropriate assembly instructions.
  + Arithmetic operations, assignments, and return statements are translated into their low-level assembly equivalents.

**Machine Code Generation Phase (Not handled by this Code)**

**Purpose of Machine Code Generation**

The goal of machine code generation is to convert the low-level **assembly code** into **machine code**, which is a sequence of binary instructions that can be executed directly by the CPU of the target machine. Each assembly instruction is mapped to an equivalent machine code instruction based on the architecture (x86, ARM, etc.).

Machine code is made up of:

* **Operation Codes (Opcodes)**: The binary encoding of operations like add, subtract, move, etc.
* **Operands**: These are the binary encodings of the registers or memory addresses involved in the operation.

**Machine Code Generation**

In a typical compiler, this phase would be handled by an **Assembler**, which translates the assembly language instructions into machine-specific opcodes and operands. Let's break down how this process works, even though the details of binary encoding may not be explicitly handled in the code, which stops at the assembly generation phase.

**Steps in Machine Code Generation**

**1. Translating Assembly Instructions to Machine Code**

Each **assembly instruction** corresponds to a **machine instruction**. For example:

* **MOV R0, x** (move the value of x into register R0) might be translated into machine code as:

0x1234 (where 0x12 is the opcode for MOV and 0x34 represents register R0 and the memory location of x)

* **ADD R0, y** (add the value of y to R0) might translate into:

0x5678 (where 0x56 is the opcode for ADD and 0x78 represents register R0 and the memory location of y)

**2. Opcode Mapping**

Each assembly instruction (like **MOV**, **ADD**, **RET**) has a corresponding **opcode** in machine language:

* **MOV** might have an opcode like 0x12
* **ADD** might have an opcode like 0x56
* **RET** might have an opcode like 0xFF

The assembler looks up the opcode for each assembly instruction and converts it into the corresponding binary sequence.

**3. Operand Encoding**

The operands (registers and memory locations) are also encoded in binary:

* **Registers**: Each register (e.g., R0, R1) is assigned a unique binary code.
  + **R0** might be encoded as 00
  + **R1** might be encoded as 01
* **Memory Locations**: Memory addresses are translated into binary. For example, if a variable x is stored at memory address 0x20, the assembler encodes that address in the machine code.

**4. Instruction Formatting**

Each machine code instruction is typically made up of:

* **Opcode**: A binary code representing the operation (e.g., MOV, ADD).
* **Operands**: Binary encodings of registers or memory locations.
* **Instruction Length**: Some architectures use fixed-length instructions, while others may use variable-length instructions depending on the operation and operands.

**Example of Machine Code Generation**

Let's walk through a couple of examples to see how assembly instructions are translated into machine code.

**1. Simple Assignment (MOV)**

Given the assembly instruction:

MOV R0, x ; Move the value of x into register R0

The machine code might look like:

00010000 00100000

Here:

* 0001 is the **opcode** for the MOV instruction.
* 0000 represents **R0** (register 0).
* 00100000 represents the memory location of x (e.g., 0x20).

**2. Addition (ADD)**

Given the assembly instruction:

ADD R0, y ; Add the value of y to R0

The machine code might look like:

01010100 00110000

Here:

* 0101 is the **opcode** for the ADD instruction.
* 0100 represents **R0** (register 0).
* 00110000 represents the memory location of y (e.g., 0x30).

**3. Return Statement (RET)**

Given the assembly instruction:

RET ; Return from the function

The machine code might look like:

Copy code

11111111

Here:

* 11111111 is the **opcode** for the RET instruction, which signals the end of a function and returns to the calling function.

**Assembler and Linking**

After generating machine code, the next steps involve:

1. **Assembling**: The assembler takes the assembly code generated by the compiler and converts it into binary machine code, which is stored in an object file (.obj or .o file).
2. **Linking**: The **linker** combines the object files generated from different modules or libraries and resolves any external references (like function calls to libraries). The final result is an **executable file** that can be run by the operating system.

**How Machine Code Generation Fits into Compilation**

* **Primary Role:** Machine code generation is the final phase in the compilation process where the assembly code is translated into binary machine code that the target CPU can execute.
* **Relation to Other Phases:**
  + **Assembly Code Generation** produces the low-level assembly instructions that are closely tied to machine operations.
  + **Machine Code Generation** converts those assembly instructions into binary machine code that is specific to the target architecture.

**Full Compilation Flow**

**Source Code (High-Level)**

//comment

x = 5;

y = 10;

z = x + y;

return z;

**Preprocessed Code:**

x = 5;

y = 10;

z = (x + y) \* 2;

return z;

**Tokens:**

(1, x) (9, =) (0, 5) (8, ;) (1, y) (9, =) (0, 10) (8, ;) (1, z) (9, =) (6, () (1, x) (2, +) (1, y) (7, )) (4, \*) (0, 2)

(8, ;) (1, return) (1, z) (8, ;)

**AST:**

Assignment:

Variable: x

Expression: Integer: 5

Assignment:

Variable: y

Expression: Integer: 10

Assignment:

Variable: z

Expression: BinaryOp: \*

Left: BinaryOp: +

Left: Variable: x

Right: Variable: y

Right: Integer: 2

Return:

Expression: Variable: z

**Semantic analysis completed**

**Three-Address Code:**

x = 5

y = 10

t1 = x + y

t2 = t1 \* 2

z = t2

return z

**Optimized Three-Address Code:**

x = 5

y = 10

t1 = 15

t2 = 30

z = 30

return z

**Assembly Code:**

section .text

global \_start

\_start:

push ebp

mov ebp, esp

sub esp, 12

mov eax, 5

mov dword [ebp - 4], eax

mov eax, 10

mov dword [ebp - 8], eax

mov eax, 15

mov eax, 30

mov eax, 30

mov dword [ebp - 12], eax

mov esp, ebp

pop ebp

ret