C++ Compiler Documentation

CC LAB

Session: 2021-2025

Project Supervisor

Prof Sir Laeeq Niazi

Group Member(s)

AFTAB ALI	2021-CS-166

Department of Computer Science

University of Engineering and Technology, Lahore

Pakistan

Table of Contents

Table of Contents

Compiler Development: Phase-by-Phase Breakdown

Phase 1: **Lexical Analysis (Lexer)**

Phase 2: **Syntax Analysis (Parser)**

Phase 3: **Semantic Analysis**

Phase 4: **Intermediate Code Generation**

Phase 5: **Code Generation**

Additional Features:

Compiler Development: Phase-by-Phase Breakdown

So far, the c++ compiler that have been developing goes through several key phases to transform the source code into executable code. These phases are essential for ensuring the code is processed correctly and efficiently. The phases are as follows:

- **Lexical Analysis** (Tokenizer)
- **Syntax Analysis** (Parser)
- **Semantic Analysis**
- **Intermediate Code Generation**
- **Code Generation**

Each phase handles a specific task, from breaking the code into tokens to generating target code.

Phase 1: Lexical Analysis (Lexer)

Description:
This is the first phase of the compiler. The lexical analyzer scans the source code and converts it into *tokens*. Tokens are small pieces of the program, like keywords (`int`), identifiers (`x`), numbers (`10`), or symbols (`+` , `*`). Below are the tokens handled in this version of the compiler:

```
enum TokenType {
    T_INT, T_FLOAT, T_DOUBLE, T_CHAR, T_BOOL, T_STRING,
    T_ID, T_NUM, T_IF, T_ELIF, T_ELSE, T_RETURN,
    T_ASSIGN, T_PLUS, T_MINUS, T_MUL, T_DIV,
    T_LPAREN, T_RPAREN, T_LBRACE, T_RBRACE,
    T_SEMICOLON, T_GT, T_LT, T_EQ, T_NE, T_EOF
};
```

Code Explanation:

The code below is responsible for breaking the input source code into manageable pieces called tokens. In simple terms, it reads the code character by character and groups them into meaningful units, such as numbers, keywords (like `int` , `if` , `else`), operators (like `+` , `-`), and symbols (like parentheses `(` , `)`).

Here's a breakdown of the key parts:

- The code loops through each character in the source code (`src`), checking if it is a number, a keyword, or an operator.
- If it's a number, it calls `consumeNumber()` to process it, and if it's a keyword like `int` or `if`, it assigns the appropriate token type (`T_INT` , `T_IF` , etc.).
- If the character is an operator (like `+` , `-` , or `*`), it adds the corresponding token.
- The function uses `pos` to keep track of the current position in the code, and `lineNumber` to keep track of the line for better error reporting.

This phase's goal is to produce a list of tokens that the next phase (the Syntax Analyzer) can work with, making sure the input is properly broken down for further processing.

```
vector<Token> tokenize() {
        vector<Token> tokens;
        while (pos < src.size()) {
            char current = src[pos];

            if (current == '\n') {
                lineNumber++;
                pos++;
                continue;
            }
            if (isspace(current)) {
                pos++;
                continue;
            }
            if (isdigit(current)) {
                tokens.push_back(Token{T_NUM, consumeNumber(), lineNumber});
                continue;
            }
            if (isalpha(current)) {
                string word = consumeWord();
                TokenType type = T_ID;

                if (word == "int") type = T_INT;
                else if (word == "float") type = T_FLOAT;
                else if (word == "if") type = T_IF;
                else if (word == "else") type = T_ELSE;
                else if (word == "return") type = T_RETURN;

                tokens.push_back(Token{type, word, lineNumber});
                continue;
            }

            switch (current) {
                case '=': tokens.push_back(Token{T_ASSIGN, "=", lineNumber});
break;
                case '+': tokens.push_back(Token{T_PLUS, "+", lineNumber});
break;
                case '-': tokens.push_back(Token{T_MINUS, "-", lineNumber});
break;
                case '*': tokens.push_back(Token{T_MUL, "*", lineNumber}); break;
                case '/': tokens.push_back(Token{T_DIV, "/", lineNumber}); break;
                case '(': tokens.push_back(Token{T_LPAREN, "(", lineNumber});
break;
                case ')': tokens.push_back(Token{T_RPAREN, ")", lineNumber});
break;
                case '{': tokens.push_back(Token{T_LBRACE, "{", lineNumber});
break;
                case '}': tokens.push_back(Token{T_RBRACE, "}", lineNumber});
break;
                case ';': tokens.push_back(Token{T_SEMICOLON, ";", lineNumber});
break;
                case '>': tokens.push_back(Token{T_GT, ">", lineNumber}); break;
                case '<': tokens.push_back(Token{T_LT, "<", lineNumber}); break;
                default:
                    cout << "Unexpected character: " << current << " at line " <<
lineNumber << endl;
                    exit(1);
            }
            pos++;
        }
        tokens.push_back(Token{T_EOF, "", lineNumber});
        return tokens;
    }
```

Phase 2: Syntax Analysis (Parser)

Description:
This phase checks if the sequence of tokens follows the grammar rules of the programming language. It also builds a structured representation of the code, often called an Abstract Syntax Tree (AST). During this phase, the **symbol table** is used to record details about variables (names, types, etc.).

Role of Symbol Table:
The parser uses the symbol table to store information about variables declared in the code and ensures no duplicate declarations.

Code Explanation:

This code is part of the **Syntactic Analyzer** phase, which checks whether the sequence of tokens produced by the Lexical Analyzer follows the correct syntax of the language. It focuses on ensuring the structure of the program is valid according to the rules of the language. Since the entire code is too long, we will focus on one aspect, which is assignment, for better understanding.

1. `parseProgram()`:
   - Loops through the tokens until it reaches the end of the file (`T_EOF`), calling `parseStatement()` for each statement.
2. `parseStatement()` :
   - Identifies the type of statement (declaration, assignment, if, return, or block) and calls the appropriate parsing function like `parseDeclaration()` , `parseAssignment()` , etc.
   - If the token doesn't match any known statement types, it reports a syntax error.
3. `parseAssignment()` :
   - This function handles the assignment statements.
   - First, it extracts the variable name using `expectAndReturnValue(T_ID)` to ensure that the token is an identifier (the variable being assigned).
   - Then, it checks if the variable has been declared in the symbol table by calling `symTable.getVariableType(varName)` . This ensures the variable exists in the current scope.
   - Next, it expects and processes the assignment operator (`T_ASSIGN`), and parses the right-hand side expression using `parseExpression()` .
   - Finally, it expects a semicolon (`T_SEMICOLON`) to complete the assignment statement.

This ensures the syntax of the program is correct and prepares it for the next phase.

```
void parseProgram() {
        while (tokens[pos].type != T_EOF) {
            parseStatement();
        }
    }

void parseStatement() {
        if (tokens[pos].type == T_INT || tokens[pos].type == T_FLOAT) {
            parseDeclaration(tokens[pos].type);
        } else if (tokens[pos].type == T_ID) {
            parseAssignment();
        } else if (tokens[pos].type == T_IF) {
            parseIfStatement();
        } else if (tokens[pos].type == T_RETURN) {
            parseReturnStatement();
        } else if (tokens[pos].type == T_LBRACE) {
            parseBlock();
        } else {
            cout << "Syntax error: unexpected token '" << tokens[pos].value << "'
at line " << tokens[pos].lineNumber << endl;
            exit(1);
        }
    }

void parseAssignment() {
        string varName = expectAndReturnValue(T_ID);
        symTable.getVariableType(varName);    // Ensure the variable is declared
in the symbol table.
        expect(T_ASSIGN);
        string expr = parseExpression();
        icg.addInstruction(varName + " = " + expr);  // Generate intermediate
code for the assignment.
        expect(T_SEMICOLON);
    }
```

Phase 3: Semantic Analysis

Description:
 Semantic analysis ensures that the program is meaningful. For example, it checks if variables are used without being declared or if types are mismatched (e.g., assigning a string to an integer variable).

Example Checks:

- Ensure variables are declared before use.
- Type-checking (e.g., integers cannot be added to strings).

Code Snippet for Semantic Checks:

```
void declareVariable(const string &name, const string &type) {
        if (symbolTable.find(name) != symbolTable.end()) {
            throw runtime_error("Semantic error: Variable '" + name + "' is
already declared.");
        }
        symbolTable[name] = type;
    }

    string getVariableType(const string &name) {
        if (symbolTable.find(name) == symbolTable.end()) {
            throw runtime_error("Semantic error: Variable '" + name + "' is not
declared.");
        }
        return symbolTable[name];
    }
```

Phase 4: Intermediate Code Generation

Description:
In this phase, the compiler generates intermediate instructions that are simple and independent of the machine architecture. These instructions are easier to translate into assembly code. The intermediate code uses three-address code, where each instruction typically involves three operands: a destination, a source, and a result. This makes the code more flexible and machine-independent.

Example Code:

The intermediate code generation happens simultaneously with parsing. Since the entire code is too long, we will focus on one aspect, which is if else block, for better understanding. Below, intermediate code is generated for `if` statements during parsing.

1. First, the `if` condition is parsed, and an intermediate instruction is created to evaluate it.
2. Labels for the "true" and "false" branches are generated, along with jump instructions based on the condition.
3. Inside the "if" block, the statement is parsed, and any necessary code is generated.
4. If there's an `else` or `else if`, corresponding labels and jump instructions are created.
5. This captures the branching logic in intermediate code, making it easier to translate into assembly later

```
void parseIfStatement() {
        expect(T_IF);                    // Expect and consume the 'if' keyword.
        expect(T_LPAREN);                // Expect and consume the opening
parenthesis for the condition.
        string cond = parseExpression(); // Parse the condition expression inside
the parentheses.
        expect(T_RPAREN);                // Expect and consume the closing
parenthesis.

        string temp = icg.newTemp();     // Generate a new temporary variable for
the condition result.
```

```
        icg.addInstruction(temp + " = " + cond); // Generate intermediate code
for evaluating the condition.

        string trueLabel = icg.generateLabel();   // Generate a label for the
"true" branch.
        string falseLabel = icg.generateLabel();  // Generate a label for the
"false" branch.
        string endLabel = "";                     // End label for all branches,
generated only if needed.

        icg.addInstruction("if " + temp + " goto " + trueLabel);  // Branch to
trueLabel if condition is true.
        icg.addInstruction("goto " + falseLabel);                // Otherwise,
branch to falseLabel.

        // True branch
        icg.addInstruction(trueLabel + ":");
        parseStatement();    // Parse the statement inside the "if" block.

        // Handle optional "else" or "else if"
        while (tokens[pos].type == T_ELSE) {
            endLabel = icg.generateLabel();       // Generate an end label for
after the "else" or "else if" block.
            icg.addInstruction("goto " + endLabel); // Jump to endLabel after the
previous block.

            icg.addInstruction(falseLabel + ":");   // Label for the "else" or
"else if" block.
            expect(T_ELSE);                          // Consume the "else" keyword.

            // Check if it's an "else if"
            if (tokens[pos].type == T_IF) {
                expect(T_IF);                     // Consume the 'if' keyword.
                expect(T_LPAREN);                 // Expect and consume the
opening parenthesis.
                string elifCond = parseExpression(); // Parse the "else if"
condition.
                expect(T_RPAREN);                 // Expect and consume the
closing parenthesis.

                string elifTemp = icg.newTemp();  // Generate a temporary
variable for the "else if" condition.
                icg.addInstruction(elifTemp + " = " + elifCond);

                trueLabel = icg.generateLabel();  // Generate a new label for the
"else if" true branch.
                falseLabel = icg.generateLabel(); // Generate a new label for the
next block.

                icg.addInstruction("if " + elifTemp + " goto " + trueLabel);
                icg.addInstruction("goto " + falseLabel);

                icg.addInstruction(trueLabel + ":");
                parseStatement();                // Parse the statement inside
the "else if" block.
            } else {
                parseStatement();                // Parse the statement inside
the plain "else" block.
                break;                           // Break the loop as there are
no further "else if" branches.
            }
        }

        if (!endLabel.empty()) {
            icg.addInstruction(endLabel + ":"); // Add the end label after the
entire chain.
        } else {
            icg.addInstruction(falseLabel + ":"); // Label for the false branch
if no "else" or "else if".
        }
    }
```

Generated Intermediate Code:

For verification, let's run the code on this c++ code snippet:

```
int x;
x = 10;
int y = 20;
int sum;
sum = x + y * 3;
if(5 > 3){
    x = 20;
}
```

Below is the intermediate code generated by the compiler for the above code:

```
x = 10
y = 20
t0 = y * 3
t1 = x + t0
sum = t1
t2 = 5 > 3
t3 = t2
if t3 goto L1
goto L2
L1:
x = 20
L2:
```

Phase 5: Code Generation

Description:
The final phase converts intermediate instructions into assembly code. Assembly code is machine-readable and can be executed by the processor. In this case, **RISC-V** assembly language is used for the conversion. RISC-V is a popular open-source instruction set architecture (ISA) that is simple and flexible, making it ideal for educational purposes and custom processor designs. The intermediate instructions generated earlier are mapped to corresponding RISC-V instructions, ensuring that the generated code can be efficiently executed on a RISC-V processor.

Code Explanation:

Below code generates RISC-V of any arithmetic operation involving only two operands, which is ensured by using three address code in earlier phase. This function processes binary operations (like addition, subtraction, multiplication, or division) in a line of intermediate code. It first extracts the left-hand side (lhs) and right-hand side (rhs) of the operation. The operation is identified based on the operator in the expression (e.g., `+`, `-`, `*`, or `/`), and a corresponding RISC-V instruction (e.g., `add`, `sub`, `mul`, `div`) is selected using a map.

The function then allocates registers for the variables involved in the operation, checking if any operand is a constant (digit). If it is, it loads the constant into a register. Finally, the appropriate assembly instruction is generated and added to the instruction list.

```
void processBinaryOperation(const string &line) {
        static const unordered_map<string, string> operationMap = {
            {"+", "add"},
            {"-", "sub"},
            {"*", "mul"},
            {"/", "div"}
        };

        auto eqPos = line.find("=");
        string tVar = line.substr(0, eqPos);
        string expr = line.substr(eqPos + 1);

        string op;
        if (expr.find("+") != string::npos) op = "+";
        else if (expr.find("-") != string::npos) op = "-";
        else if (expr.find("*") != string::npos) op = "*";
        else if (expr.find("/") != string::npos) op = "/";

        auto opPos = expr.find(op);
        string lhs = expr.substr(0, opPos);
        string rhs = expr.substr(opPos + 1);
        string operation = operationMap.at(op);

        tVar = trim(tVar);
        lhs = trim(lhs);
        rhs = trim(rhs);

        string targetRegister = allocateRegister(tVar);
        string lhsRegister = isdigit(lhs[0]) ? "t" + to_string(registerCounter++)
: allocateRegister(lhs);
        string rhsRegister = isdigit(rhs[0]) ? "t" + to_string(registerCounter++)
: allocateRegister(rhs);

        if (isdigit(lhs[0])) addInstruction("li " + lhsRegister + ", " + lhs);
        if (isdigit(rhs[0])) addInstruction("li " + rhsRegister + ", " + rhs);

        addInstruction(operation + " " + targetRegister + ", " + lhsRegister + ",
" + rhsRegister);
    }
```

Generated Assembly Code:

For verification, let's run the whole code on the previously generated intermediate code:

```
x = 10
y = 20
t0 = y * 3
t1 = x + t0
sum = t1
t2 = 5 > 3
t3 = t2
if t3 goto L1
goto L2
L1:
x = 20
L2:
```

Below is the assembly code generated by the compiler for the above code:

```
li t0, 10
li t1, 20
li t3, 3
mul t2, t1, t3
add t4, t0, t2
mv t5, t4
sgt t8, t6, t7
mv t9, t8
bnez t2, L1
j L2
L1:
li t0, 20
L2:
```

Additional Features:

Below are the additional features I added:

1. Supporting more variable types for declaration.
2. Allowing declaration and initialization in the same line.
3. Handling multiple `if` statements with `else if` and `else`, ending with an appropriate block.
4. Handling **while** loop.
5. Generating RISC-V assembly code from intermediate code.
6. Optimizaion Constant Folding &DeadCode Elimination