

6/26/2025

# COSC 44283

## Theory of Compilers

Assignment- MiniLang- Mini Compiler  
Development Project

1- Lexical Analysis Implementation



PS/2020/188 - WIJAYASEKARA N.

## Introduction

The MiniLang project involves the design and implementation of a compiler for a simplified programming language. The compiler simulates real-world compilation phases: lexical analysis, syntax analysis, semantic analysis, and intermediate code generation. This document explains each component and provides example input/output to demonstrate compiler functionality.

## Project Objective

The main objectives of this project are:

- To design a custom programming language called MiniLang with essential programming features.
- To implement a lexical analyzer using regular expressions to tokenize source code.
- To develop a syntax analyzer using recursive descent parsing to validate language structure.
- To perform semantic analysis such as type checking and undeclared variable detection.
- To generate intermediate 3-address code representing the logic of MiniLang programs.
- To gain hands-on experience in compiler construction and understanding compiler phases.

## Tools and Languages

- Programming Language: Java
- Tools: Java Regex for tokenization, Recursive Descent Parser for syntax analysis
- IDE: IntelliJ IDEA
- Operating System: Windows

**All documents :** <https://github.com/NeluniWijayasekara/MiniLang--Mini-Compiler>

## MiniLang Grammar Description

**MiniLang** is a simple high-level programming language designed to support basic constructs of imperative programming. It includes variable declarations, assignments, arithmetic operations, conditional statements, loops, and output printing.

## Lexical Analysis Implementation

### Overview

The Lexical Analysis phase is the first step in the MiniLang compiler. Its purpose is to read MiniLang source code and break it into a sequence of tokens, such as keywords, identifiers, numbers, operators, and punctuation. This phase is crucial for enabling correct parsing and further analysis.

The lexical analyzer reads the source code and breaks it into tokens: keywords, identifiers, numbers, operators, and punctuation.

**Token Types:** Defines all possible token types recognized by the lexer.

```
enum TokenType {  
    INT, IF, ELSE, WHILE, PRINT, INPUT,  
    ID, NUMBER,  
    PLUS, MINUS, TIMES, DIVIDE,  
    ASSIGN, GT, LT,  
    LPAREN, RPAREN, LBRACE, RBRACE, SEMICOLON,  
    COMMENT, WHITESPACE,  
    UNKNOWN  
}
```

### Token Class

```
static class Token {  
    TokenType type;  
    String value;  
    int line;  
  
    Token(TokenType type, String value, int line) {  
        this.type = type;  
        this.value = value;  
        this.line = line;  
    }  
  
    public String toString() {  
        return type + "(" + value + ") at line " + line;  
    }  
}
```

## Source code of the compiler

### JAVA Code:

MiniLangLexer.java

```
import java.util.*;
import java.util.regex.*;
import java.io.*;

public class MiniLangLexer {
    // Define token types for MiniLang language
    enum TokenType {
        INT, IF, ELSE, WHILE, PRINT, INPUT,    // Keywords
        ID, NUMBER,                             // Identifiers and literals
        PLUS, MINUS, TIMES, DIVIDE,             // Arithmetic operators
        ASSIGN, GT, LT,                         // Assignment and comparison
        LPAREN, RPAREN, LBRACE, RBRACE, SEMICOLON, // Delimiters
        COMMENT, WHITESPACE,                   // Ignored elements

        UNKNOWN                                // Error tokens
    }

    // Token class to store token information
    static class Token {
        TokenType type;    // Token category
        String value;      // Actual string value
        int line;         // Source line number

        Token(TokenType type, String value, int line) {
            this.type = type;
            this.value = value;
            this.line = line;
        }

        public String toString() {
            return type + "(" + value + ") at line " + line;
        }
    }

    // Map token types to their regex patterns
```

```

private static final Map<TokenType, String> tokenPatterns = new LinkedHashMap<>();
static {
    // Define regex patterns for each token type
    tokenPatterns.put(TokenType.WHITESPACE, "\\s+");    // Whitespace
    tokenPatterns.put(TokenType.COMMENT, "//.*");        // Single-line comments
    tokenPatterns.put(TokenType.INT, "\\bint\\b");      // 'int' keyword
    tokenPatterns.put(TokenType.IF, "\\bif\\b");        // 'if' keyword
    tokenPatterns.put(TokenType.ELSE, "\\belse\\b");    // 'else' keyword
    tokenPatterns.put(TokenType.WHILE, "\\bwhile\\b");  // 'while' keyword
    tokenPatterns.put(TokenType.PRINT, "\\bprint\\b");  // 'print' keyword
    tokenPatterns.put(TokenType.INPUT, "\\binput\\b");  // 'input' keyword
    tokenPatterns.put(TokenType.NUMBER, "\\b\\d+\\b");   // Integer literals
    tokenPatterns.put(TokenType.ID, "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b"); // Identifiers
    tokenPatterns.put(TokenType.PLUS, "\\+");           // Addition operator
    tokenPatterns.put(TokenType.MINUS, "-");            // Subtraction operator
    tokenPatterns.put(TokenType.TIMES, "\\*");          // Multiplication operator
    tokenPatterns.put(TokenType.DIVIDE, "/");           // Division operator
    tokenPatterns.put(TokenType.ASSIGN, "=");           // Assignment operator
    tokenPatterns.put(TokenType.GT, ">");               // Greater than
    tokenPatterns.put(TokenType.LT, "<");               // Less than
    tokenPatterns.put(TokenType.LPAREN, "\\(");         // Left parenthesis
    tokenPatterns.put(TokenType.RPAREN, "\\)");         // Right parenthesis
    tokenPatterns.put(TokenType.LBRACE, "\\{");         // Left brace
    tokenPatterns.put(TokenType.RBRACE, "\\}");         // Right brace
    tokenPatterns.put(TokenType.SEMICOLON, ";");        // Semicolon
}

private static final Pattern masterPattern;
private static final List<TokenType> masterTypes = new ArrayList<>();

static {
    StringBuilder masterPat = new StringBuilder();
    for (Map.Entry<TokenType, String> entry : tokenPatterns.entrySet()) {
        if (masterPat.length() > 0) masterPat.append("|");
        masterPat.append("(").append(entry.getValue()).append(")");
        masterTypes.add(entry.getKey());
    }
    masterPattern = Pattern.compile(masterPat.toString());
}

// Lexical analysis method
public static List<Token> lex(String input) {

```

```

List<Token> tokens = new ArrayList<>();
String[] lines = input.split("\n");
boolean errorFound = false;

for (int lineNum = 0; lineNum < lines.length; lineNum++) {
    String line = lines[lineNum];
    int pos = 0;

    // Process entire line
    while (pos < line.length()) {
        Matcher matcher = masterPattern.matcher(line);
        matcher.region(pos, line.length());
        if (matcher.lookingAt()) {
            for (int i = 1; i <= matcher.groupCount(); i++) {
                if (matcher.group(i) != null) {
                    TokenType type = masterTypes.get(i - 1);
                    String value = matcher.group(i);
                    if (type == TokenType.WHITESPACE || type == TokenType.COMMENT) {
                        // skip
                    } else {
                        tokens.add(new Token(type, value, lineNum + 1));
                    }
                    pos = matcher.end();
                    break;
                }
            }
        } else {
            System.err.println("Lexical Error: Unrecognized character " +
                line.charAt(pos) + " at line " + (lineNum + 1));
            errorFound = true;
            pos++;
        }
    }
}

if (errorFound) {
    System.err.println("Lexical analysis completed with errors.");
}

return tokens;
}

```

```

// Main method for interactive testing

```

```

public static void main(String[] args) throws IOException {
    System.out.println("Enter your MiniLang code (finish input with an empty line):");
    Scanner scanner = new Scanner(System.in);
    StringBuilder inputBuilder = new StringBuilder();
    String line;
    while (true) {
        line = scanner.nextLine();
        if (line.trim().isEmpty()) break;
        inputBuilder.append(line).append("\n");
    }
    String input = inputBuilder.toString();
    if (input.trim().isEmpty()) {
        System.out.println("No input provided. Exiting.");
        return;
    }
    // Perform lexical analysis
    List<Token> tokens = lex(input);

    // Print all tokens
    System.out.println("\nTokens:");
    for (Token token : tokens) {
        System.out.println(token);
    }
}
}

```

### Input:

```

"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Fi
Enter your MiniLang code (finish input with an empty line):
int a;
input(a);
a = a + 1;
print(a);

```

### Output

```
Tokens:
INT(int) at line 1
ID(a) at line 1
SEMICOLON(;) at line 1
INPUT(input) at line 2
LPAREN(() at line 2
ID(a) at line 2
RPAREN()) at line 2
SEMICOLON(;) at line 2
ID(a) at line 3
ASSIGN(=) at line 3
ID(a) at line 3
PLUS(+) at line 3
NUMBER(1) at line 3
SEMICOLON(;) at line 3
PRINT(print) at line 4
LPAREN(() at line 4
ID(a) at line 4
RPAREN()) at line 4
SEMICOLON(;) at line 4
```

**When error occur**



```

"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\Jet
Enter your MiniLang code (finish input with an empty line):
int 2abc;          // Invalid identifier: starts with digit
a = 5$;           // Invalid character: '$'
print(@a);        // Invalid character: '@'
while (a < 10) {
    b = a + #1;    // Invalid character: '#'
}

Lexical Error: Unrecognized character '2' at line 1
Lexical Error: Unrecognized character '$' at line 2
Lexical Error: Unrecognized character '@' at line 3
Lexical Error: Unrecognized character '#' at line 5
Lexical analysis completed with errors.

```

## Syntax Analysis Implementation

### Overview

The Syntax Analysis phase (parsing) checks whether the sequence of tokens produced by the lexical analyzer conforms to the MiniLang grammar. It validates the structure of the source code, detects syntax errors, and ensures that statements and expressions are used correctly.

**Error Handling Utilities :** *All syntax errors are reported with clear messages and line numbers. Parsing stops immediately upon error detection.*

```

// Unexpected token error
private void handleUnexpectedToken(MiniLangLexer.Token t) {
    System.out.println("Syntax error at line " + t.line +
        ": Unexpected token " + t.type);
    System.exit(1);
}

// Missing relational operator
private void reportMissingRelOp() {
    System.out.println("Syntax error at line " +
        (peek() != null ? peek().line : "EOF") +
        ": Expected relational operator");
}

```

```

    System.exit(1);
}

// Invalid factor in expression
private void reportInvalidFactor() {
    System.out.println("Syntax error at line " +
        (peek() != null ? peek().line : "EOF") +
        ": Expected ID, NUMBER, or '('");
    System.exit(1);
}

```

### MiniLangParser.java

```

import java.util.List;
import java.util.Scanner;

public class MiniLangParser {
    // List of tokens generated by the lexer
    private List<MiniLangLexer.Token> tokens;
    // Current position in the token list
    private int current = 0;

    // Constructor initializes with tokens from lexer
    public MiniLangParser(List<MiniLangLexer.Token> tokens) {
        this.tokens = tokens;
    }

    // ===== HELPER METHODS =====

    // Peek at current token without consuming it
    private MiniLangLexer.Token peek() {
        return current < tokens.size() ? tokens.get(current) : null;
    }

    // Move to next token
    private void advance() {
        if (current < tokens.size()) current++;
    }

    // Check if current token matches expected type, consume if matches
    private boolean match(MiniLangLexer.TokenType expected) {

```

```

    if (peek() != null && peek().type == expected) {
        advance();
        return true;
    }
    return false;
}

// Enforce that current token MUST be of expected type
private void expect(MiniLangLexer.TokenType expected) {
    if (!match(expected)) {
        MiniLangLexer.Token t = peek();
        // Error handling: unexpected or missing token
        System.out.println("Syntax error at line " + (t != null ? t.line : "EOF") +
            ": Expected " + expected + ", found " + (t != null ? t.type : "EOF"));
        System.exit(1);
    }
}

// ===== MAIN PARSING METHODS =====

// Entry point: Parse entire program
public void parseProgram() {
    while (peek() != null) {
        parseStmt();
    }
}

// Dispatch to appropriate statement parser
private void parseStmt() {
    MiniLangLexer.Token t = peek();
    if (t == null) return;
    switch (t.type) {
        case INT:  parseDeclStmt(); break;
        case ID:   parseAssignStmt(); break;
        case IF:   parseIfStmt(); break;
        case WHILE: parseWhileStmt(); break;
        case PRINT: parsePrintStmt(); break;
        case INPUT: parseInputStmt(); break;
        default:   handleUnexpectedToken(t); // Error handling: unexpected token
    }
}

```

```
// DECLARATION STATEMENT: int <id>;
private void parseDeclStmt() {
    expect(MiniLangLexer.TokenType.INT);
    expect(MiniLangLexer.TokenType.ID);
    expect(MiniLangLexer.TokenType.SEMICOLON);
}
```

```
// ASSIGNMENT STATEMENT: <id> = <expr>;
private void parseAssignStmt() {
    expect(MiniLangLexer.TokenType.ID);
    expect(MiniLangLexer.TokenType.ASSIGN);
    parseExpr();
    expect(MiniLangLexer.TokenType.SEMICOLON);
}
```

```
// IF STATEMENT: if (cond) block [else block]
private void parseIfStmt() {
    expect(MiniLangLexer.TokenType.IF);
    expect(MiniLangLexer.TokenType.LPAREN);
    parseCond();
    expect(MiniLangLexer.TokenType.RPAREN);
    parseBlock();
    // Optional else clause
    if (match(MiniLangLexer.TokenType.ELSE)) {
        parseBlock();
    }
}
```

```
// WHILE LOOP: while (cond) block
private void parseWhileStmt() {
    expect(MiniLangLexer.TokenType.WHILE);
    expect(MiniLangLexer.TokenType.LPAREN);
    parseCond();
    expect(MiniLangLexer.TokenType.RPAREN);
    parseBlock();
}
```

```
// PRINT STATEMENT: print(id);
private void parsePrintStmt() {
    expect(MiniLangLexer.TokenType.PRINT);
}
```

```

    expect(MiniLangLexer.TokenType.LPAREN);
    expect(MiniLangLexer.TokenType.ID);
    expect(MiniLangLexer.TokenType.RPAREN);
    expect(MiniLangLexer.TokenType.SEMICOLON);
}

// INPUT STATEMENT: input(id);
private void parseInputStmt() {
    expect(MiniLangLexer.TokenType.INPUT);
    expect(MiniLangLexer.TokenType.LPAREN);
    expect(MiniLangLexer.TokenType.ID);
    expect(MiniLangLexer.TokenType.RPAREN);
    expect(MiniLangLexer.TokenType.SEMICOLON);
}

// BLOCK: { stmt* } or single statement
private void parseBlock() {
    if (match(MiniLangLexer.TokenType.LBRACE)) {
        // Parse multiple statements until closing brace
        while (!match(MiniLangLexer.TokenType.RBRACE)) {
            parseStmt();
        }
    } else {
        // Single statement without braces
        parseStmt();
    }
}

// CONDITION: expr relop expr
private void parseCond() {
    parseExpr();
    // Check for relational operator (>, <, =)
    if (peek() != null && isRelOp(peek().type)) {
        advance();
        parseExpr();
    } else {
        reportMissingRelOp(); // Error handling: missing relational operator
    }
}

// EXPRESSION PARSING (with operator precedence)

```

```

private void parseExpr() {
    parseTerm();
    while (peek() != null && isAddOp(peek().type)) {
        advance();
        parseTerm();
    }
}

private void parseTerm() {
    parseFactor();
    while (peek() != null && isMulOp(peek().type)) {
        advance();
        parseFactor();
    }
}

private void parseFactor() {
    if (match(MiniLangLexer.TokenType.ID) || match(MiniLangLexer.TokenType.NUMBER)) {
        // Handled by match() calls
    } else if (match(MiniLangLexer.TokenType.LPAREN)) {
        parseExpr();
        expect(MiniLangLexer.TokenType.RPAREN);
    } else {
        reportInvalidFactor(); // Error handling: invalid expression factor
    }
}

// ===== UTILITY METHODS =====

private boolean isRelOp(MiniLangLexer.TokenType type) {
    return type == MiniLangLexer.TokenType.GT ||
        type == MiniLangLexer.TokenType.LT ||
        type == MiniLangLexer.TokenType.ASSIGN;
}

private boolean isAddOp(MiniLangLexer.TokenType type) {
    return type == MiniLangLexer.TokenType.PLUS ||
        type == MiniLangLexer.TokenType.MINUS;
}

private boolean isMulOp(MiniLangLexer.TokenType type) {

```

```

        return type == MiniLangLexer.TokenType.TIMES ||
            type == MiniLangLexer.TokenType.DIVIDE;
    }

    // Error handling: unexpected token
    private void handleUnexpectedToken(MiniLangLexer.Token t) {
        System.out.println("Syntax error at line " + t.line +
            ": Unexpected token " + t.type);
        System.exit(1);
    }

    // Error handling: missing relational operator
    private void reportMissingRelOp() {
        System.out.println("Syntax error at line " +
            (peek() != null ? peek().line : "EOF") +
            ": Expected relational operator");
        System.exit(1);
    }

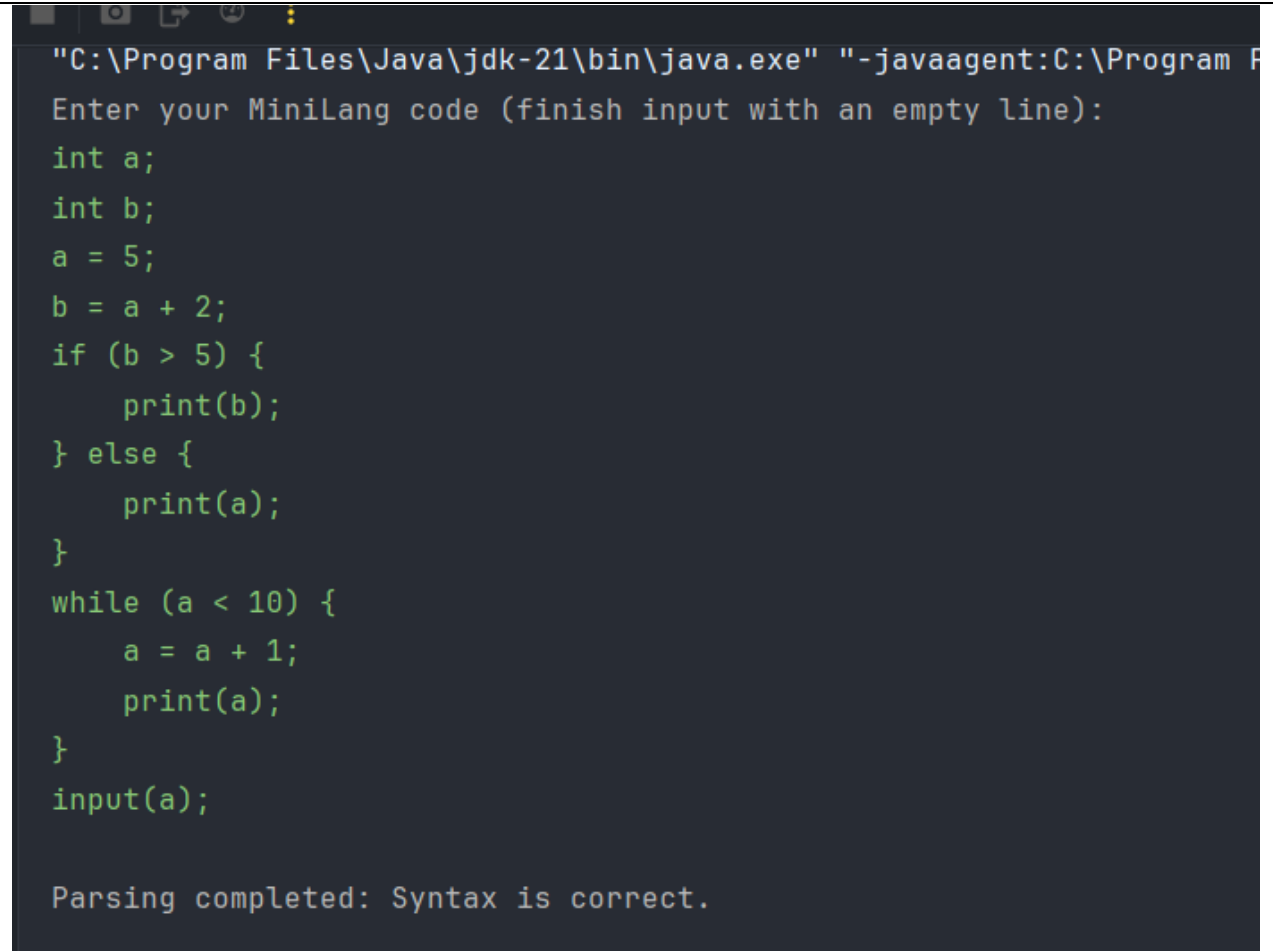
    // Error handling: invalid factor in expression
    private void reportInvalidFactor() {
        System.out.println("Syntax error at line " +
            (peek() != null ? peek().line : "EOF") +
            ": Expected ID, NUMBER, or '('");
        System.exit(1);
    }

    // ===== MAIN METHOD =====
    public static void main(String[] args) {
        System.out.println("Enter your MiniLang code (finish input with an empty line):");
        Scanner scanner = new Scanner(System.in);
        StringBuilder inputBuilder = new StringBuilder();
        String line;
        // Read input until empty line
        while (true) {
            line = scanner.nextLine();
            if (line.trim().isEmpty()) break;
            inputBuilder.append(line).append("\n");
        }
        String input = inputBuilder.toString();
        if (input.trim().isEmpty()) {

```

```
        System.out.println("No input provided. Exiting.");
        return;
    }
    // Lexical analysis
    List<MiniLangLexer.Token> tokens = MiniLangLexer.lex(input);
    // Syntax analysis
    MiniLangParser parser = new MiniLangParser(tokens);
    parser.parseProgram();
    System.out.println("Parsing completed: Syntax is correct.");
}
}
```

### Input:



```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program F
Enter your MiniLang code (finish input with an empty line):
int a;
int b;
a = 5;
b = a + 2;
if (b > 5) {
    print(b);
} else {
    print(a);
}
while (a < 10) {
    a = a + 1;
    print(a);
}
input(a);

Parsing completed: Syntax is correct.
```



```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program File
Enter your MiniLang code (finish input with an empty line):
int a
a = 5;

Syntax error at line 2: Expected SEMICOLON, found ID

Process finished with exit code 1
```

## Semantic Analysis Implementation

### Overview

The Semantic Analysis phase ensures that the MiniLang program is meaningful and correct beyond syntax. It checks for correct variable declarations, prevents redeclaration, ensures variables are used only after declaration, and reports semantic errors with clear messages and line numbers. This phase uses a symbol table to track declared variables.

### MiniLangParser.java

```
import java.util.List;
import java.util.Scanner;

public class MiniLangParser {
    private final List<MiniLangLexer.Token> tokens;
    private int current = 0;
    private final SymbolTable symbolTable = new SymbolTable();

    public MiniLangParser(List<MiniLangLexer.Token> tokens) {
        this.tokens = tokens;
    }
}
```

```

// ===== Helper Methods =====
private MiniLangLexer.Token peek() {
    return current < tokens.size() ? tokens.get(current) : null;
}
private void advance() {
    if (current < tokens.size()) current++;
}
private boolean match(MiniLangLexer.TokenType expected) {
    if (peek() != null && peek().type == expected) {
        advance();
        return true;
    }
    return false;
}
private void expect(MiniLangLexer.TokenType expected) {
    if (!match(expected)) {
        MiniLangLexer.Token t = peek();
        System.out.println("Syntax error at line " + (t != null ? t.line : "EOF") +
            ": Expected " + expected + ", found " + (t != null ? t.type : "EOF"));
        System.exit(1);
    }
}

// ===== Entry Point =====
public void parseProgram() {
    while (peek() != null) {
        parseStmt();
    }
    if (symbolTable.hasErrors()) {
        System.err.println("Semantic analysis completed with errors.");
        System.exit(1);
    }
}

// ===== Statement Parsers with Semantic Checks =====
private void parseStmt() {
    MiniLangLexer.Token t = peek();
    if (t == null) return;
}

```

```

switch (t.type) {
    case INT:  parseDeclStmt(); break;
    case ID:   parseAssignStmt(); break;
    case IF:   parseIfStmt(); break;
    case WHILE: parseWhileStmt(); break;
    case PRINT: parsePrintStmt(); break;
    case INPUT: parseInputStmt(); break;
    default:   handleUnexpectedToken(t);
}
}

// Declaration: int <id>;
private void parseDeclStmt() {
    expect(MiniLangLexer.TokenType.INT);
    MiniLangLexer.Token idToken = peek();
    expect(MiniLangLexer.TokenType.ID);
    symbolTable.declare(idToken.value, idToken.line); // Semantic check
    expect(MiniLangLexer.TokenType.SEMICOLON);
}

// Assignment: <id> = <expr>;
private void parseAssignStmt() {
    MiniLangLexer.Token idToken = peek();
    expect(MiniLangLexer.TokenType.ID);
    symbolTable.checkDeclared(idToken.value, idToken.line); // Semantic check
    expect(MiniLangLexer.TokenType.ASSIGN);
    parseExpr();
    expect(MiniLangLexer.TokenType.SEMICOLON);
}

// If statement
private void parseIfStmt() {
    expect(MiniLangLexer.TokenType.IF);
    expect(MiniLangLexer.TokenType.LPAREN);
    parseCond();
    expect(MiniLangLexer.TokenType.RPAREN);
    parseBlock();
    if (match(MiniLangLexer.TokenType.ELSE)) {

```

```

        parseBlock();
    }
}

// While statement
private void parseWhileStmt() {
    expect(MiniLangLexer.TokenType.WHILE);
    expect(MiniLangLexer.TokenType.LPAREN);
    parseCond();
    expect(MiniLangLexer.TokenType.RPAREN);
    parseBlock();
}

// Print statement
private void parsePrintStmt() {
    expect(MiniLangLexer.TokenType.PRINT);
    expect(MiniLangLexer.TokenType.LPAREN);
    MiniLangLexer.Token idToken = peek();
    expect(MiniLangLexer.TokenType.ID);
    symbolTable.checkDeclared(idToken.value, idToken.line); // Semantic check
    expect(MiniLangLexer.TokenType.RPAREN);
    expect(MiniLangLexer.TokenType.SEMICOLON);
}

// Input statement
private void parseInputStmt() {
    expect(MiniLangLexer.TokenType.INPUT);
    expect(MiniLangLexer.TokenType.LPAREN);
    MiniLangLexer.Token idToken = peek();
    expect(MiniLangLexer.TokenType.ID);
    symbolTable.checkDeclared(idToken.value, idToken.line); // Semantic check
    expect(MiniLangLexer.TokenType.RPAREN);
    expect(MiniLangLexer.TokenType.SEMICOLON);
}

// Block: { ... } or single statement
private void parseBlock() {
    if (match(MiniLangLexer.TokenType.LBRACE)) {

```

```

        while (!match(MiniLangLexer.TokenType.RBRACE)) {
            parseStmt();
        }
    } else {
        parseStmt();
    }
}

// Condition: <expr> relop <expr>
private void parseCond() {
    parseExpr();
    if (peek() != null && isRelOp(peek().type)) {
        advance();
        parseExpr();
    } else {
        reportMissingRelOp();
    }
}

// ===== Expression Parsing with Semantic Checks =====
private void parseExpr() {
    parseTerm();
    while (peek() != null && isAddOp(peek().type)) {
        advance();
        parseTerm();
    }
}

private void parseTerm() {
    parseFactor();
    while (peek() != null && isMulOp(peek().type)) {
        advance();
        parseFactor();
    }
}

private void parseFactor() {
    if (match(MiniLangLexer.TokenType.ID)) {

```

```

    MiniLangLexer.Token idToken = tokens.get(current - 1);
    symbolTable.checkDeclared(idToken.value, idToken.line); // Semantic check
} else if (match(MiniLangLexer.TokenType.NUMBER)) {
    // No semantic check needed for number
} else if (match(MiniLangLexer.TokenType.LPAREN)) {
    parseExpr();
    expect(MiniLangLexer.TokenType.RPAREN);
} else {
    reportInvalidFactor();
}
}

// ===== Utility Methods =====
private boolean isRelOp(MiniLangLexer.TokenType type) {
    return type == MiniLangLexer.TokenType.GT ||
           type == MiniLangLexer.TokenType.LT ||
           type == MiniLangLexer.TokenType.ASSIGN;
}
private boolean isAddOp(MiniLangLexer.TokenType type) {
    return type == MiniLangLexer.TokenType.PLUS ||
           type == MiniLangLexer.TokenType.MINUS;
}
private boolean isMulOp(MiniLangLexer.TokenType type) {
    return type == MiniLangLexer.TokenType.TIMES ||
           type == MiniLangLexer.TokenType.DIVIDE;
}
private void handleUnexpectedToken(MiniLangLexer.Token t) {
    System.out.println("Syntax error at line " + t.line +
        ": Unexpected token " + t.type);
    System.exit(1);
}
private void reportMissingRelOp() {
    System.out.println("Syntax error at line " +
        (peek() != null ? peek().line : "EOF") +
        ": Expected relational operator");
    System.exit(1);
}
private void reportInvalidFactor() {

```

```

        System.out.println("Syntax error at line " +
            (peek() != null ? peek().line : "EOF") +
            ": Expected ID, NUMBER, or '('");
        System.exit(1);
    }

    // ===== Main Method for Testing =====
    public static void main(String[] args) {
        System.out.println("Enter your MiniLang code (finish input with an empty line):");
        Scanner scanner = new Scanner(System.in);
        StringBuilder inputBuilder = new StringBuilder();
        String line;
        while (true) {
            line = scanner.nextLine();
            if (line.trim().isEmpty()) break;
            inputBuilder.append(line).append("\n");
        }
        String input = inputBuilder.toString();
        if (input.trim().isEmpty()) {
            System.out.println("No input provided. Exiting.");
            return;
        }
        // Lexical analysis
        List<MiniLangLexer.Token> tokens = MiniLangLexer.lex(input);
        // Syntax and semantic analysis
        MiniLangParser parser = new MiniLangParser(tokens);
        parser.parseProgram();
        System.out.println("Parsing and semantic analysis completed: No errors found.");
    }
}

```

SymbolTable.java

```

import java.util.HashSet;
import java.util.Set;

```

```

// SymbolTable manages variable declarations and semantic error tracking
public class SymbolTable {
    private final Set<String> declaredVars = new HashSet<>();
    private boolean errorOccurred = false;

    // Declare a variable; report error if already declared
    public void declare(String varName, int line) {
        if (declaredVars.contains(varName)) {
            System.err.println(" Semantic Error: Variable '" + varName +
                "' already declared at line " + line);
            errorOccurred = true;
        } else {
            declaredVars.add(varName);
        }
    }

    // Check if a variable is declared; report error if not
    public void checkDeclared(String varName, int line) {
        if (!declaredVars.contains(varName)) {
            System.err.println("Semantic Error: Variable '" + varName +
                "' not declared at line " + line);
            errorOccurred = true;
        }
    }

    // Returns true if any semantic errors occurred
    public boolean hasErrors() {
        return errorOccurred;
    }
}

```

### Highlights:

Prevents variable redeclaration.

Detects use of undeclared variables.

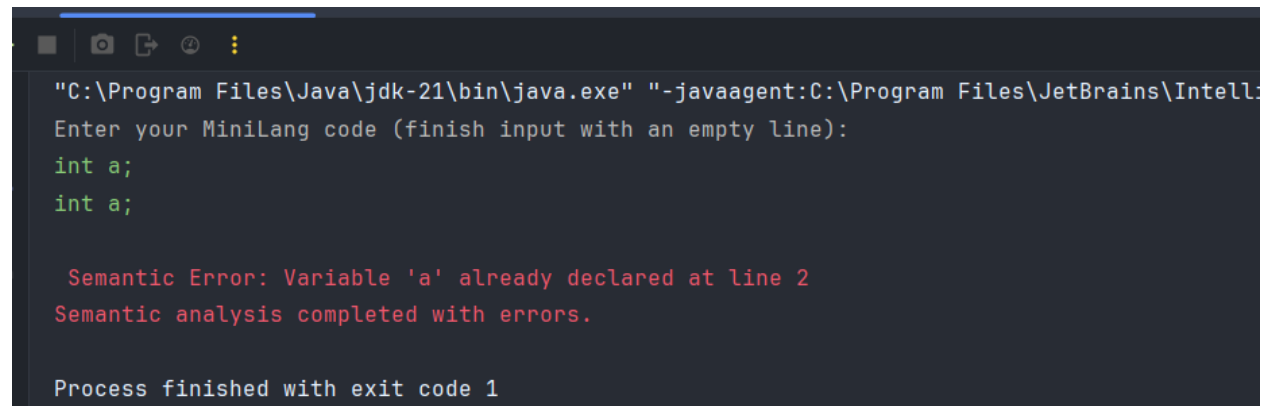
Tracks if any semantic errors have occurred.



**Error Handling** : If any semantic errors are detected, the compiler prints a summary and exits.

```
public void parseProgram() {  
    while (peek() != null) {  
        parseStmt();  
    }  
    if (symbolTable.hasErrors()) {  
        System.err.println("Semantic analysis completed with errors.");  
        System.exit(1);  
    }  
}
```

Input:



```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA\lib\idea_rt.jar" 1.0.0  
Enter your MiniLang code (finish input with an empty line):  
int a;  
int a;  
  
Semantic Error: Variable 'a' already declared at line 2  
Semantic analysis completed with errors.  
  
Process finished with exit code 1
```

```

"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA\bin\idea_rt.jar"
Enter your MiniLang code (finish input with an empty line):
int a;
int b;
a = 5;
b = a + 3;
print(a);
if (b > 5) {
    b = b - 1;
    print(b);
} else {
    a = a + 1;
    print(a);
}
while (a < 10) {
    a = a + 1;
    print(a);
}
input(b);

Parsing and semantic analysis completed: No errors found.

```

## Intermediate Code Generation

### Overview

The Intermediate Code Generation phase translates MiniLang source code (after passing lexical, syntax, and semantic analysis) into a lower-level, three-address code (TAC) format. This code is easier to optimize and serves as a bridge to target machine code or a virtual machine. The code generator also enforces error handling: if any syntax or semantic error is found, it reports the error (with line number) and halts code generation, as required by your assignment

### Error Handling

```

private void expect(MiniLangLexer.TokenType expected) {
    if (!match(expected)) {
        MiniLangLexer.Token t = peek();
        System.out.println("Syntax error at line " + (t != null ? t.line : "EOF") +
            ": Expected " + expected + ", found " + (t != null ? t.type : "EOF"));
    }
}

```

```

        System.exit(1);
    }
}

private void checkVariableDeclared(String varName, int line) {
    if (!symbolTable.containsKey(varName)) {
        System.out.println("Semantic error at line " + line + ": Variable '" + varName + "' used
before declaration.");
        System.exit(1);
    }
}
}

```

Syntax errors: If the expected token is not found, a clear error message with the line number is printed and code generation stops.

Semantic errors: If a variable is redeclared or used before declaration, a semantic error is reported and code generation halts.

No intermediate code is emitted if errors are found, as required by best practices and your assignment

### MiniLangCompiler.java

```

import java.util.*;

public class MiniLangCompiler {
    private List<MiniLangLexer.Token> tokens;
    private int current = 0;
    private HashMap<String, Integer> symbolTable = new HashMap<>(); // variable -> line
    declared

    // For code generation
    private int tempCount = 1;
    private int labelCount = 1;

    public MiniLangCompiler(List<MiniLangLexer.Token> tokens) {

```

```

    this.tokens = tokens;
}

// Helper methods for code generation
private String newTemp() {
    return "t" + (tempCount++);
}
private String newLabel() {
    return "L" + (labelCount++);
}
private void emit(String code) {
    System.out.println(code);
}

// Token helpers
private MiniLangLexer.Token peek() {
    return current < tokens.size() ? tokens.get(current) : null;
}
private void advance() {
    if (current < tokens.size()) current++;
}
private boolean match(MiniLangLexer.TokenType expected) {
    if (peek() != null && peek().type == expected) {
        advance();
        return true;
    }
    return false;
}
private void expect(MiniLangLexer.TokenType expected) {
    if (!match(expected)) {
        MiniLangLexer.Token t = peek();
        System.out.println("Syntax error at line " + (t != null ? t.line : "EOF") +
            ": Expected " + expected + ", found " + (t != null ? t.type : "EOF"));
        System.exit(1);
    }
}
private void checkVariableDeclared(String varName, int line) {
    if (!symbolTable.containsKey(varName)) {

```

```
        System.out.println("Semantic error at line " + line + ": Variable '" + varName + "' used  
before declaration.");  
        System.exit(1);  
    }  
}
```

```
// Main parse function  
public void parseProgram() {  
    while (peek() != null) {  
        parseStmt();  
    }  
}
```

```
private void parseStmt() {  
    MiniLangLexer.Token t = peek();  
    if (t == null) return;  
    switch (t.type) {  
        case INT:  parseDeclStmt(); break;  
        case ID:   parseAssignStmt(); break;  
        case IF:   parseIfStmt(); break;  
        case WHILE: parseWhileStmt(); break;  
        case PRINT: parsePrintStmt(); break;  
        case INPUT: parseInputStmt(); break;  
        default:  
            System.out.println("Syntax error at line " + t.line + ": Unexpected token " + t.type);  
            System.exit(1);  
    }  
}
```

```
// int a;  
private void parseDeclStmt() {  
    expect(MiniLangLexer.TokenType.INT);  
    MiniLangLexer.Token idToken = peek();  
    expect(MiniLangLexer.TokenType.ID);  
    expect(MiniLangLexer.TokenType.SEMICOLON);
```

```
    String varName = idToken.value;  
    if (symbolTable.containsKey(varName)) {
```

```
        System.out.println("Semantic error at line " + idToken.line + ": Variable '" + varName +  
        "' redeclared.");  
        System.exit(1);  
    } else {  
        symbolTable.put(varName, idToken.line);  
        emit(" " + varName);  
    }  
}
```

```
// a = expr;
```

```
private void parseAssignStmt() {  
    MiniLangLexer.Token idToken = peek();  
    expect(MiniLangLexer.TokenType.ID);  
    checkVariableDeclared(idToken.value, idToken.line);  
    expect(MiniLangLexer.TokenType.ASSIGN);  
    String rhs = parseExpr();  
    expect(MiniLangLexer.TokenType.SEMICOLON);  
    emit("= " + idToken.value + ", " + rhs);  
}
```

```
// if (cond) block [else block]
```

```
private void parseIfStmt() {  
    expect(MiniLangLexer.TokenType.IF);  
    expect(MiniLangLexer.TokenType.LPAREN);  
    String condTemp = parseCond();  
    expect(MiniLangLexer.TokenType.RPAREN);  
    String labelElse = newLabel();  
    String labelEnd = newLabel();  
    emit("?:= " + labelElse + ", " + condTemp);  
    parseBlock();  
    if (peek() != null && peek().type == MiniLangLexer.TokenType.ELSE) {  
        emit(":= " + labelEnd);  
        emit(": " + labelElse);  
        advance();  
        parseBlock();  
        emit(": " + labelEnd);  
    } else {  
        emit(": " + labelElse);  
    }  
}
```

```
}  
}
```

```
// while (cond) block
```

```
private void parseWhileStmt() {  
    expect(MiniLangLexer.TokenType.WHILE);  
    String labelStart = newLabel();  
    String labelEnd = newLabel();  
    emit(": " + labelStart);  
    expect(MiniLangLexer.TokenType.LPAREN);  
    String condTemp = parseCond();  
    expect(MiniLangLexer.TokenType.RPAREN);  
    emit("?:= " + labelEnd + ", " + condTemp);  
    parseBlock();  
    emit(":= " + labelStart);  
    emit(": " + labelEnd);  
}
```

```
// print(a);
```

```
private void parsePrintStmt() {  
    expect(MiniLangLexer.TokenType.PRINT);  
    expect(MiniLangLexer.TokenType.LPAREN);  
    MiniLangLexer.Token idToken = peek();  
    expect(MiniLangLexer.TokenType.ID);  
    checkVariableDeclared(idToken.value, idToken.line);  
    expect(MiniLangLexer.TokenType.RPAREN);  
    expect(MiniLangLexer.TokenType.SEMICOLON);  
    emit("> " + idToken.value);  
}
```

```
// input(a);
```

```
private void parseInputStmt() {  
    expect(MiniLangLexer.TokenType.INPUT);  
    expect(MiniLangLexer.TokenType.LPAREN);  
    MiniLangLexer.Token idToken = peek();  
    expect(MiniLangLexer.TokenType.ID);  
    checkVariableDeclared(idToken.value, idToken.line);  
    expect(MiniLangLexer.TokenType.RPAREN);
```

```

    expect(MiniLangLexer.TokenType.SEMICOLON);
    emit("< " + idToken.value);
}

// { ... } or single statement
private void parseBlock() {
    if (peek() != null && peek().type == MiniLangLexer.TokenType.LBRACE) {
        expect(MiniLangLexer.TokenType.LBRACE);
        while (peek() != null && peek().type != MiniLangLexer.TokenType.RBRACE) {
            parseStmt();
        }
        expect(MiniLangLexer.TokenType.RBRACE);
    } else {
        parseStmt();
    }
}

// cond: expr relop expr (like a > 0)
private String parseCond() {
    String left = parseExpr();
    MiniLangLexer.Token opToken = peek();
    if (peek() != null &&
        (peek().type == MiniLangLexer.TokenType.GT ||
         peek().type == MiniLangLexer.TokenType.LT ||
         peek().type == MiniLangLexer.TokenType.ASSIGN)) { // '=' as '=='
        advance();
        String right = parseExpr();
        String temp = newTemp();
        String op = opToken.type == MiniLangLexer.TokenType.GT ? ">" :
            opToken.type == MiniLangLexer.TokenType.LT ? "<" : "=";
        emit(op + " " + temp + " , " + left + " , " + right);
        return temp;
    } else {
        System.out.println("Syntax error at line " + (peek() != null ? peek().line : "EOF") + ":
Expected relational operator");
        System.exit(1);
        return null;
    }
}

```



```
}
```

```
// expr: term { (+|-) term }
```

```
private String parseExpr() {
```

```
    String left = parseTerm();
```

```
    while (peek() != null &&
```

```
        (peek().type == MiniLangLexer.TokenType.PLUS ||
```

```
        peek().type == MiniLangLexer.TokenType.MINUS)) {
```

```
        MiniLangLexer.Token opToken = peek();
```

```
        advance();
```

```
        String right = parseTerm();
```

```
        String temp = newTemp();
```

```
        String op = opToken.type == MiniLangLexer.TokenType.PLUS ? "+" : "-";
```

```
        emit(op + " " + temp + " " + left + " " + right);
```

```
        left = temp;
```

```
    }
```

```
    return left;
```

```
}
```

```
// term: factor { (*|/) factor }
```

```
private String parseTerm() {
```

```
    String left = parseFactor();
```

```
    while (peek() != null &&
```

```
        (peek().type == MiniLangLexer.TokenType.TIMES ||
```

```
        peek().type == MiniLangLexer.TokenType.DIVIDE)) {
```

```
        MiniLangLexer.Token opToken = peek();
```

```
        advance();
```

```
        String right = parseFactor();
```

```
        String temp = newTemp();
```

```
        String op = opToken.type == MiniLangLexer.TokenType.TIMES ? "*" : "/";
```

```
        emit(op + " " + temp + " " + left + " " + right);
```

```
        left = temp;
```

```
    }
```

```
    return left;
```

```
}
```

```
// In parseFactor: Check for invalid factors
```

```

private String parseFactor() {
    if (peek() != null && peek().type == MiniLangLexer.TokenType.ID) {
        MiniLangLexer.Token idToken = peek();
        advance();
        checkVariableDeclared(idToken.value, idToken.line);
        return idToken.value;
    } else if (peek() != null && peek().type == MiniLangLexer.TokenType.NUMBER) {
        String num = peek().value;
        advance();
        return num;
    } else if (peek() != null && peek().type == MiniLangLexer.TokenType.LPAREN) {
        expect(MiniLangLexer.TokenType.LPAREN);
        String expr = parseExpr();
        expect(MiniLangLexer.TokenType.RPAREN);
        return expr;
    } else {
        System.out.println("Syntax error at line " + (peek() != null ? peek().line : "EOF") +
            ": Expected ID, NUMBER, or '('");
        System.exit(1);    // Stop code generation on error
        return null;
    }
}

// =====MAIN METHOD=====
public static void main(String[] args) {
    System.out.println("Enter your MiniLang code (finish input with an empty line):");
    Scanner scanner = new Scanner(System.in);
    StringBuilder inputBuilder = new StringBuilder();
    String line;
    while (true) {
        line = scanner.nextLine();
        if (line.trim().isEmpty()) break;
        inputBuilder.append(line).append("\n");
    }
    String input = inputBuilder.toString();
    if (input.trim().isEmpty()) {
        System.out.println("No input provided. Exiting.");
        return;
    }
}

```

```

    }
    List<MiniLangLexer.Token> tokens = MiniLangLexer.lex(input);
    MiniLangCompiler compiler = new MiniLangCompiler(tokens);
    compiler.parseProgram();
    System.out.println("Intermediate code generation completed.");
}
}

```

input:

int a; int b; a = 5; b = a + 2; print(b);

```

PS D:\OneDrive - University of Kelaniya\4th Year (Semester 7)\COSC 44283 Theory of Compilers\Final Assignment> java MiniLangCompiler
Enter your MiniLang code (finish input with an empty line):
int a;
int b;
a = 5;
b = a + 2;
print(b);

. a
. b
= a, 5
+ t1, a, 2
= b, t1
.> b
Intermediate code generation completed.
PS D:\OneDrive - University of Kelaniya\4th Year (Semester 7)\COSC 44283 Theory of Compilers\Final Assignment> 

```