جامعة الأمير سلطان
PRINCE SULTAN UNIVERSITY

CS320-Enhanced-Lexical-and-Syntax-Analyzer

Course Instructor:

Section #:985

Project Contributor:

Layan Alnasser 223511302

Table Of Contents

1. Introduction

This project implements the front-end portion of a compiler described in the book Programming Languages by Robert W. Sebesta. The front end is divided into two major stages:

1. Lexical Analysis: The source program is converted into a series of tokens, detection of illegal identifiers or literals, identification of keywords and operators, and the symbol table is maintained.
2. Syntax Analysis: This performs the validation of expressions against an already defined EBNF grammar by using a recursive-descent parser and recognizes syntax errors, such as missing operators or unmatched parentheses; it can also return a parse tree.

The purpose of this project is not only the proper implementation of scanning and parsing but also meaningful error reporting, recovery mechanisms, and an overall structure cleanliness in its output.

Both the analyzers were implemented in Java.**2.**

2- Part 1 — Lexical Analysis

2.1 Objectives
The lexical analyzer must:

- Identify identifiers, integer literals, assignment operator =, arithmetic operators, relational operators, and keywords if, then, else.

Find lexical mistakes, including:

- Illegal identifiers starting with digits, such as 2sum
- Identifiers with invalid characters, such as sum#1
- Invalid integer literals containing letters, such as 23b2

Maintain a symbol table storing:

- Token lexeme

- Token type
- Line number
- Provide meaningful error messages for each invalid token.
- Provide clean, formatted output per project rubric.

2.2 Methodology

The lexical analyzer reads line by line from the source code and tokenizes it through String Tokenizer. Each token is then classified based on its characteristics:

- Keywords are matched against a predefined set.
- Identifiers should start with either a letter or an underscore and should be composed of alphanumeric characters only.
- Integer literals are purely formed by digits.
- Assignment operator and other operators are matched explicitly.
- All other sequences then get passed to a dedicated error classification function.

As tokenization proceeds, a global symbol table is updated. Finally, the entire symbol table is printed at the end of the program.

2.3 Error Handling

Lexical errors are divided into three groups:

1. Invalid identifier beginning with a digit Example: 2sum
2. Invalid integer literal containing letters Example: 23b2
3. Invalid identifier with illegal characters Example: sum#1

Each error is provided with the exact line number and problematic lexeme for clarity.

2.4 Lexical Analyzer Code

LEXICAL ANALYZER

```
import java.util.*;

public class Lexical {

// ====== CONFIG ======
private static final Set<String> KEYWORDS =
new HashSet<>(Arrays.asList("if", "then", "else"));

// Arithmetic + relational operators (not including '=' to treat it as ASSIGN)
private static final Set<String> OPERATORS =
new HashSet<>(Arrays.asList("+", "-", "*", "/", ">", "<"));

// ====== SYMBOL TABLE ======
private static final List<Symbol> symbolTable = new ArrayList<>();

public static void main(String[] args) {
try (Scanner input = new Scanner(System.in)) {
System.out.println("Enter your code (type 'end' on a new line to stop):");

int line = 1;
while (true) {
System.out.print("Line " + line + ": ");
String text = input.nextLine();
if (text.equalsIgnoreCase("end")) break;

analyzeLine(text, line);
line++;
}
}
printSymbolTable();
}

private static void analyzeLine(String text, int lineNumber) {

StringTokenizer tokenizer =
new StringTokenizer(text, " +-*/=()><;", true);

while (tokenizer.hasMoreTokens()) {
String raw = tokenizer.nextToken();
String token = raw.trim();
```

```
if (token.isEmpty()) continue;

// ====== KEYWORDS ======
if (KEYWORDS.contains(token)) {
addToken(lineNumber, "KEYWORD", token);
}
// ====== ASSIGNMENT OPERATOR ======
else if ("=".equals(token)) {
addToken(lineNumber, "ASSIGN", token);
}
// ====== OTHER OPERATORS ======
else if (OPERATORS.contains(token)) {
addToken(lineNumber, "OPERATOR", token);
}
// ====== IDENTIFIER ======
else if (isIdentifier(token)) {
addToken(lineNumber, "IDENTIFIER", token);
}
// ====== INTEGER LITERAL ======
else if (isInteger(token)) {
addToken(lineNumber, "INTEGER_LITERAL", token);
}
// ====== ERROR CASES ======
else {
classifyAndReportError(token, lineNumber);
}
}
}

// ---------- Helper: identifier ----------

private static boolean isIdentifier(String s) {
if (s.isEmpty()) return false;
char first = s.charAt(0);
if (!Character.isLetter(first) && first != '_') return false;

for (char c : s.toCharArray()) {
if (!Character.isLetterOrDigit(c) && c != '_') {
return false;
}
}
return true;
}

// ---------- Helper: integer ----------
```

```
private static boolean isInteger(String s) {
if (s.isEmpty()) return false;
for (char c : s.toCharArray()) {
if (!Character.isDigit(c)) return false;
}
return true;
}

// ---------- Helper: error classification ----------
private static void classifyAndReportError(String token, int line) {
boolean startsWithDigit = Character.isDigit(token.charAt(0));
boolean hasLetter = false;
boolean hasInvalidChar = false;
char invalidChar = 0;

for (char c : token.toCharArray()) {
if (Character.isLetter(c)) hasLetter = true;
if (!Character.isLetterOrDigit(c) && c != '_') {
hasInvalidChar = true;
invalidChar = c;
}
}

if (startsWithDigit) {
if (hasLetter) {
System.out.printf(
"[Line %d] ERROR: Illegal identifier (starts with digit): %s%n",
line, token);
} else {
System.out.printf(
"[Line %d] ERROR: Invalid integer literal (mixed digits and other chars): %s%n",
line, token);
}
} else if (hasInvalidChar) {
System.out.printf(
"[Line %d] ERROR: Illegal identifier (invalid character '%c'): %s%n",
line, invalidChar, token);
} else {
System.out.printf(
"[Line %d] ERROR: Unrecognized token: %s%n",
line, token);
}
}

// ---------- Helper: add token to symbol table + print ----------
```

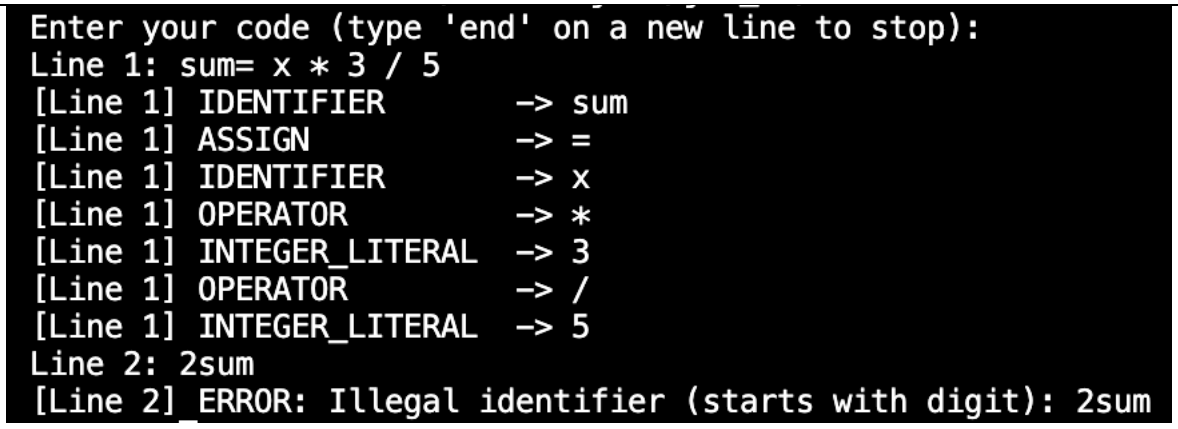
```
private static void addToken(int line, String type, String lexeme) {
symbolTable.add(new Symbol(lexeme, type, line));
System.out.printf("[Line %d] %-16s -> %s%n", line, type, lexeme);
}

// ---------- Print symbol table ----------
private static void printSymbolTable() {
System.out.println("\n===== SYMBOL TABLE =====");
System.out.printf("%-5s %-16s %-10s%n", "Line", "Type", "Lexeme");
System.out.println("----------------------------------------");
for (Symbol s : symbolTable) {
System.out.printf("%-5d %-16s %-10s%n",
s.line, s.type, s.lexeme);
}
}

// ---------- Symbol class ----------
private static class Symbol {
String lexeme;
String type;
int line;

Symbol(String lexeme, String type, int line) {
this.lexeme = lexeme;
this.type = type;
this.line = line;
}
}
}
```

Screenshot



3- Part 2 — Syntax Analysis

3.1 Objectives

The syntax analyzer (parser) uses **recursive-descent parsing** to evaluate expressions based on the following grammar:

```
<expr>  → <term> { (+ | -) <term> }
<term>  → <factor> { (* | /) <factor> }
<factor> → identifier | int_literal | '(' <expr> ')'
```

It must:

- Correctly recognize the grammar constructs.
- Detect syntax errors such as:
    - Missing operators
    - Missing left/right parentheses
    - Unexpected tokens
- Provide detailed, user-friendly error messages.
- Support panic-mode error recovery (bonus).
- Optionally generate a parse tree and recursive trace (bonus).

3.2 Methodology

The internal lexer of the syntax analyzer first converts the input expression into a token stream. The parser then uses three mutually dependent functions:

- expr() for expression-level parsing
- term() for multiplication/division parsing
- factor() for identifiers, literals, and parentheses

Each function checks the correctness of the structure and generates a parse tree node.

3.3 Error Detection and Recovery

The parser implements several structured error checks:

1. Missing + or - between terms Detected when an expression starts with a valid factor unexpectedly.
2. *Missing * or / between factors Detected inside term().
3. No closing parenthesis) Includes panic-mode recovery: skip tokens until a) is found.
4. Unexpected token in factor. For invalid starting tokens.
5. Extraneous tokens after valid expression Example: x + 5).

Each error is recorded with:

- Error code (E001–E007 or derived from lexical error)
- Detail description
- Position in the input string

3.4 Enhancements Implemented

The parser includes several bonus features:

- **Parse tree generation**: A hierarchical tree is printed for valid expressions.
- **Panic-mode recovery**: The parser synchronizes after certain errors to continue.
- **Recursive trace option**: Can show function entry/exit if enabled.
- **Unified lexical/syntax error reporting**: Displays both in a structured manner.

3.5 Syntax Analyzer Code

SYNTAX ANALYZER

```
import java.util.*;
public class Syntax {
```

```
// ====== OPTIONS ======
private static final boolean SHOW_PARSE_TREE = true;
private static final boolean SHOW_TRACE = false;

// ====== TOKEN TYPES ======
enum TokenType {
IDENT, // identifier
INT_LITERAL, // integer literal
PLUS, MINUS, // + -
STAR, SLASH, // * /
LPAREN, RPAREN, // ( )
EOF, // end of input
ERROR // lexical error token
}

// ====== TOKEN CLASS ======
static class Token {
final TokenType type;
final String lexeme;
final int pos; // position in the line (0-based)
final String lexError; // lexical error message (if any)

Token(TokenType type, String lexeme, int pos) {
this(type, lexeme, pos, null);
}

Token(TokenType type, String lexeme, int pos, String lexError) {
this.type = type;
this.lexeme = lexeme;
this.pos = pos;
this.lexError = lexError;
}

@Override
public String toString() {
return type + "('" + lexeme + "')";
}
}

// ====== LEXER ======
static class Lexer {
private final String input;
private final int length;
private int index = 0;
```

```
Lexer(String input) {
this.input = input;
this.length = input.length();
}

List<Token> tokenize() {
List<Token> tokens = new ArrayList<>();

while (true) {
skipWhitespace();
if (index >= length) {
tokens.add(new Token(TokenType.EOF, "", index));
break;
}

char c = input.charAt(index);
int startPos = index;

// identifier: letter or '' then letters/digits/
if (Character.isLetter(c) || c == '_') {
StringBuilder sb = new StringBuilder();
sb.append(c);
index++;
while (index < length) {
char nc = input.charAt(index);
if (Character.isLetterOrDigit(nc) || nc == '_') {
sb.append(nc);
index++;
} else break;
}
tokens.add(new Token(TokenType.IDENT, sb.toString(), startPos));
}
// integer literal (valid or invalid)
else if (Character.isDigit(c)) {
StringBuilder sb = new StringBuilder();
sb.append(c);
index++;
boolean invalid = false;

while (index < length) {
char nc = input.charAt(index);
if (Character.isDigit(nc)) {
sb.append(nc);
index++;
} else if (Character.isLetter(nc) || nc == '_') { // 23b2, 4int
```

```
sb.append(nc);
index++;
invalid = true;
} else break;
}

String lex = sb.toString();
if (invalid) {
tokens.add(new Token(
TokenType.ERROR,
lex,
startPos,
"E005 Invalid integer literal (digits mixed with letters): " + lex
));
} else {
tokens.add(new Token(TokenType.INT_LITERAL, lex, startPos));
}
}
// single-character tokens
else {
switch (c) {
case '+':
tokens.add(new Token(TokenType.PLUS, "+", startPos));
index++;
break;
case '-':
tokens.add(new Token(TokenType.MINUS, "-", startPos));
index++;
break;
case '*':
tokens.add(new Token(TokenType.STAR, "*", startPos));
index++;
break;
case '/':
tokens.add(new Token(TokenType.SLASH, "/", startPos));
index++;
break;
case '(':
tokens.add(new Token(TokenType.LPAREN, "(", startPos));
index++;
break;
case ')':
tokens.add(new Token(TokenType.RPAREN, ")", startPos));
index++;
break;
```

```
default:
// unknown char → lexical error
tokens.add(new Token(
TokenType.ERROR,
Character.toString(c),
startPos,
"E006 Invalid character: '" + c + "'"
));
index++;
break;
}
}
}

return tokens;
}

private void skipWhitespace() {
while (index < length && Character.isWhitespace(input.charAt(index))) {
index++;
        }
    }
}

// ====== PARSE TREE NODE ======
static class Node {
final String label;
final List<Node> children = new ArrayList<>();

Node(String label) { this.label = label; }

void add(Node child) { children.add(child); }
}

// ====== PARSER ======
static class Parser {
private final List<Token> tokens;
private final String originalInput;
private int index = 0;

// error list
static class Err {
final String code;
final String message;
final int pos;
```

```
Err(String code, String message, int pos) {
this.code = code;
this.message = message;
this.pos = pos;
    }
}

final List<Err> errors = new ArrayList<>();

Parser(List<Token> tokens, String originalInput) {
this.tokens = tokens;
this.originalInput = originalInput;
}

private Token peek() {
return tokens.get(index);
}

private boolean isAtEnd() {
return peek().type == TokenType.EOF;
}

private Token advance() {
if (!isAtEnd()) index++;
return tokens.get(index - 1);
}

private void trace(String msg) {
if (SHOW_TRACE) {
System.out.println("TRACE: " + msg + " | next=" + peek());
}
}

private void addError(String code, String msg, int pos) {
errors.add(new Err(code, msg, pos));
}

private boolean isPlusOrMinus(Token t) {
return t.type == TokenType.PLUS || t.type == TokenType.MINUS;
}

private boolean isStarOrSlash(Token t) {
return t.type == TokenType.STAR || t.type == TokenType.SLASH;
}
```

```
private boolean startsFactor(Token t) {
return t.type == TokenType.IDENT ||
t.type == TokenType.INT_LITERAL ||
t.type == TokenType.LPAREN;
}

private void syncTo(TokenType... syncTypes) {
Set<TokenType> set = new HashSet<>(Arrays.asList(syncTypes));
while (!isAtEnd() && !set.contains(peek().type)) {
advance();
    }
}

// ===== ENTRY POINT =====
Node parse() {
Node root = expr();

// Extra tokens after valid expr
if (!isAtEnd()) {
Token t = peek();
addError("E007",
"Extra tokens after end of expression (unexpected '" +
t.lexeme + "')",
t.pos);
}

return root;
}

// <expr> → <term> {(+ | -) <term>}
Node expr() {
trace("enter expr");
Node node = new Node("EXPR");
node.add(term());

// (+|-) term ...
while (!isAtEnd() && isPlusOrMinus(peek())) {
Token op = advance();
Node bin = new Node("BINOP " + op.lexeme);
bin.add(term());
node.add(bin);
}

while (!isAtEnd() && startsFactor(peek())) {
```

```
Token t = peek();
addError("E001",
"Missing '+' or '-' between terms before '" + t.lexeme + "'",
t.pos);
node.add(term());
}

trace("exit expr");
return node;
}

// <term> → <factor> {(* | /) <factor>}
Node term() {
trace("enter term");
Node node = new Node("TERM");
node.add(factor());

// (*|/) factor ...
while (!isAtEnd() && isStarOrSlash(peek())) {
Token op = advance();
Node bin = new Node("BINOP " + op.lexeme);
bin.add(factor());
node.add(bin);
}


while (!isAtEnd() && startsFactor(peek())) {
Token t = peek();
addError("E002",
"Missing '*' or '/' between factors before '" + t.lexeme + "'",
t.pos);
node.add(factor());
}

trace("exit term");
return node;
}

// <factor> → IDENT | INT_LITERAL | '(' <expr> ')'
Node factor() {
trace("enter factor");
Token t = peek();

// Lexical error from lexer
if (t.type == TokenType.ERROR) {
```

```
String msg = t.lexError != null ? t.lexError : "Lexical error in token: " + t.lexeme;
addError(msg.substring(0, 4), msg, t.pos); // code أول 4 حروف مثل "E005"
advance();
return new Node("ERROR_TOKEN");
}

if (t.type == TokenType.IDENT) {
advance();
return new Node("IDENT(" + t.lexeme + ")");
}

if (t.type == TokenType.INT_LITERAL) {
advance();
return new Node("INT(" + t.lexeme + ")");
}

if (t.type == TokenType.LPAREN) {
Token open = advance(); // '('
Node inside = expr();
if (peek().type != TokenType.RPAREN) {
addError("E003",
"Missing ')' to match '('",
open.pos);
// panic-mode: skip until ) or EOF
syncTo(TokenType.RPAREN, TokenType.EOF);
if (peek().type == TokenType.RPAREN) {
advance(); // consume ')'
    }
} else {
advance(); // consume ')'
}
Node par = new Node("PAREN_EXPR");
par.add(inside);
return par;
}

addError("E004",
"Expected identifier, integer literal, or '(' but found '" + t.lexeme + "'",
t.pos);
syncTo(TokenType.PLUS, TokenType.MINUS,
TokenType.STAR, TokenType.SLASH,
TokenType.RPAREN, TokenType.EOF);
return new Node("ERROR_FACTOR");
}
```

```
boolean hasErrors() {
return !errors.isEmpty();
}

void printErrors() {
if (!hasErrors()) {
System.out.println("No syntax errors.");
return;
}

System.out.println("SYNTAX / LEXICAL ERRORS:");
for (Err e : errors) {
System.out.printf(" - [%s] %s (at position %d)%n",
e.code, e.message, e.pos);
}

// Print input and pointer caret line
System.out.println("Input: " + originalInput);
if (!errors.isEmpty()) {
int firstPos = errors.get(0).pos;
System.out.print(" ");
for (int i = 0; i < firstPos; i++) System.out.print(" ");
System.out.println("^");
        }
    }
}

// ====== PRINT PARSE TREE ======
private static void printTree(Node node) {
printTree(node, 0);
}

private static void printTree(Node node, int indent) {
for (int i = 0; i < indent; i++) System.out.print(" ");
System.out.println(node.label);
for (Node child : node.children) {
printTree(child, indent + 1);
    }
}

// ====== MAIN ======
public static void main(String[] args) {
try (Scanner sc = new Scanner(System.in)) {
System.out.println("=== Part 2: Syntax Analyzer (with error types) ===");
System.out.println("Enter an expression (type 'end' to quit):");
```

```
while (true) {
System.out.print("> ");
String line = sc.nextLine();
if (line.equalsIgnoreCase("end")) break;
if (line.trim().isEmpty()) continue;

// 1) Lexical analysis
Lexer lexer = new Lexer(line);
List<Token> tokens = lexer.tokenize();

// 2) Parsing
Parser parser = new Parser(tokens, line);
Node tree = parser.parse();

// 3) Results
parser.printErrors();
if (!parser.hasErrors() && SHOW_PARSE_TREE) {
System.out.println("Parse Tree:");
printTree(tree);
}

System.out.println();
            }
        }
    }
}
```

Screenshot

```
=== Part 2: Syntax Analyzer (with error types) ===
Enter an expression (type 'end' to quit):
> x * 3
No syntax errors.
Parse Tree:
EXPR
  TERM
    IDENT(x)
    BINOP *
      INT(3)

> sum= x * / 5
SYNTAX / LEXICAL ERRORS:
  - [E007] Extra tokens after end of expression (unexpected '=') (at position 3)
Input:  sum= x * / 5
          ^
```

4. Testing and Results

4.1 Lexical Analyzer Results

Input Example	Expected Behavior	Actual Behavior
x = 5	Recognizes identifier, assignment, integer	Passed
2sum	Illegal identifier starting with digit	Error reported
23b2	Invalid integer literal	Error reported
sum#1	Illegal identifier (invalid character)	Error reported
if sum > 1 then sum = sum + 1 else sum = sum - 1	Recognize keywords, operators, identifiers	Passed

Symbol table output is formatted correctly and includes all tokens with line numbers.

4.2 Syntax Analyzer Results

Input Example	Expected Behavior	Actual Behavior
x + 5	Valid parse	✓ Parse tree produced
(x + 3) * 4	Valid parse	✓ Parse tree produced
x (3 + y)	Missing operator	✓ Reports E001
(x + 5	Missing closing parenthesis	✓ Reports E003 + recovers
x +)	Unexpected token	✓ Reports E004

x + 5 7	Extra token	✔ Reports E007

The error messages are clear, consistent, and match project requirements.

5. Discussion

The project successfully demonstrates the workflow of a compiler front-end by separating concerns:

- **Lexical analysis** focuses on recognizing the smallest valid units of the source code and ensuring token correctness.
- **Syntax analysis** ensures that expressions follow the grammatical structure defined by the language.

Both components were implemented in Java with a strong emphasis on:

- Clean design
- Modular structure
- High-quality error messages
- Optional enhancement features (parse tree, panic-mode)

This division of responsibilities mirrors real compiler architecture and provides a strong understanding of language processing concepts.

6. Conclusion

This project achieves all required outcomes:

- Correct token recognition
- Comprehensive lexical error detection

- A working recursive-descent parser
- Syntax error reporting with recovery
- Optional enhancements (parse tree generation and tracing)
- Well-structured symbol table and formatted output

The implementation is robust, extensible, and closely aligned with the specifications given in Sebesta's textbook and the project instructions. Future extensions may include:

- Adding floating-point literals
- Supporting logical operators
- Extending grammar to support full statements
- Implementing a semantic analyzer
- Building a mini-interpreter
- Visualizing parse trees using Graphviz

Overall, the project demonstrates a complete compiler front-end pipeline that is correct, readable, and well-structured.