Blatt 04 – A4.3 Lexer in Java

Modul: Compilerbau — Gruppe: Adrian Kramkowski, Abdelhadi Fares, Yousef Al Sahli, Abdelraoof Sahli

---

A4.3 – Lexer (Token-Strukturen und Implementierung)

1) Token-Strukturen in Java

```
/**
 * Token-Typen für die Lisp-artige Zielsprache.
 * Diese Typen entsprechen den Terminals der Grammatik aus A4.2.
 */
public enum TokenType {
    // Klammern
    LPAREN,     // (
    RPAREN,     // )

    // Literale und Identifikatoren
    INT,        // z.B. 42
    STRING,     // z.B. "hello"
    BOOL,       // true, false
    IDENT,      // Bezeichner (Variablen- und Funktionsnamen)

    // Operatoren
    PLUS,       // +
    MINUS,      // -
    TIMES,      // *
    DIV,        // /
    EQ,         // =
    LT,         // <
    GT,         // >

    // Schlüsselwörter / spezielle Formen
    IF,         // if
    DO,         // do
    DEF,        // def
    DEFN,       // defn
    LET,        // let
    LIST,       // list
    NTH,        // nth
```

```
    HEAD,       // head
    TAIL,       // tail
    PRINT,      // print
    STR,        // str

    // Dateiende
    EOF
}

/**
 * Repräsentiert ein einzelnes Token im Tokenstrom.
 * Neben Typ und Lexem speichern wir Zeilen- und Spaltennummer
 * zur besseren Fehlerdiagnose.
 */
public class Token {
    private final TokenType type;
    private final String lexeme;
    private final int line;
    private final int column;

    public Token(TokenType type, String lexeme, int line, int column) {
        this.type = type;
        this.lexeme = lexeme;
        this.line = line;
        this.column = column;
    }

    public TokenType getType() {
        return type;
    }

    public String getLexeme() {
        return lexeme;
    }

    public int getLine() {
        return line;
    }

    public int getColumn() {
        return column;
    }
```

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

2) Manuell implementierter Lexer (recursive descent über Zeichen)

```
/**
 * Handgeschriebener Lexer für die Lisp-artige Sprache.
 * Eingabe ist ein kompletter String mit Quelltext.
 *
 * - Whitespace (Leerzeichen, Tabs, Newlines) wird übersprungen.
 * - Kommentare beginnen mit ";;" und gehen bis zum Zeilenende.
 * - Strings werden in "..." geschrieben.
 * - Zahlen sind Integer-Literale (Folge von Ziffern).
 * - Bezeichner werden auf Schlüsselwörter/Bool-Werte gemappt.
 */
public class Lexer {

    private final String input;
    private int pos = 0;
    private int line = 1;
    private int column = 1;

    public Lexer(String input) {
        this.input = input;
    }

    /**
     * Liefert das nächste Token aus dem Eingabestrom.
     * Bei unerwarteten Zeichen wird eine RuntimeException mit
     * einer Fehlermeldung geworfen, die das problematische Zeichen
     * und seine Position enthält.
     */
    public Token nextToken() {
        skipWhitespaceAndComments();

        if (isAtEnd()) {
            return new Token(TokenType.EOF, "<EOF>", line, column);
        }
```

```
char c = advance(); // aktuelles Zeichen "verbrauchen"
int tokenLine = line;
int tokenColumn = column - 1; // advance() hat column schon erhöht

switch (c) {
    case '(':
        return new Token(TokenType.LPAREN, "(", tokenLine, tokenColumn);
    case ')':
        return new Token(TokenType.RPAREN, ")", tokenLine, tokenColumn);
    case '+':
        return new Token(TokenType.PLUS, "+", tokenLine, tokenColumn);
    case '-':
        return new Token(TokenType.MINUS, "-", tokenLine, tokenColumn);
    case '*':
        return new Token(TokenType.TIMES, "*", tokenLine, tokenColumn);
    case '/':
        return new Token(TokenType.DIV, "/", tokenLine, tokenColumn);
    case '=':
        return new Token(TokenType.EQ, "=", tokenLine, tokenColumn);
    case '<':
        return new Token(TokenType.LT, "<", tokenLine, tokenColumn);
    case '>':
        return new Token(TokenType.GT, ">", tokenLine, tokenColumn);
    case '"':
        // String-Literal lesen (ohne umschließende Anführungszeichen)
        String str = readString(tokenLine, tokenColumn);
        return new Token(TokenType.STRING, str, tokenLine, tokenColumn);
    default:
        if (Character.isDigit(c)) {
            String number = readNumber(c);
            return new Token(TokenType.INT, number, tokenLine, tokenColumn);
        } else if (isIdentifierStart(c)) {
            String ident = readIdentifier(c);
            TokenType type = classifyIdentifier(ident);
            if (type == TokenType.BOOL) {
                return new Token(TokenType.BOOL, ident, tokenLine, tokenColumn);
            } else {
                return new Token(type, ident, tokenLine, tokenColumn);
            }
        } else {
            // Unerwartetes Zeichen → rudimentäre Fehlerbehandlung
            throw unexpectedCharError(c, tokenLine, tokenColumn);
```

```
            }
        }
    }

    // ---------------------------------------------------------
    // Hilfsfunktionen
    // ---------------------------------------------------------

    private boolean isAtEnd() {
        return pos >= input.length();
    }

    private char peek() {
        if (isAtEnd()) {
            return '\0';
        }
        return input.charAt(pos);
    }

    private char advance() {
        char c = isAtEnd() ? '\0' : input.charAt(pos);
        pos++;
        if (c == '\n') {
            line++;
            column = 1;
        } else {
            column++;
        }
        return c;
    }

    /**
     * Überspringt Whitespace und Kommentare.
     * Kommentare beginnen mit ";;" und gehen bis zum Zeilenende.
     */
    private void skipWhitespaceAndComments() {
        boolean again;
        do {
            again = false;

            // Whitespace
            while (!isAtEnd()) {
                char c = peek();
```

```
            if (c == ' ' || c == '\t' || c == '\r' || c == '\n') {
                advance();
            } else {
                break;
            }
        }

        // Kommentar
        if (!isAtEnd() && peek() == ';') {
            // prüfen, ob ";;" folgt
            int savePos = pos;
            int saveLine = line;
            int saveColumn = column;
            char first = advance();
            char second = isAtEnd() ? '\0' : peek();
            if (second == ';') {
                // zweites ';' auch konsumieren
                advance();
                // Rest der Zeile ignorieren
                while (!isAtEnd() && peek() != '\n') {
                    advance();
                }
                again = true; // danach erneut Whitespace prüfen
            } else {
                // nur ein ';' → zurückrollen und als normales Zeichen behandeln
                pos = savePos;
                line = saveLine;
                column = saveColumn;
            }
        }

    } while (again);
}

private String readNumber(char firstDigit) {
    StringBuilder sb = new StringBuilder();
    sb.append(firstDigit);
    while (!isAtEnd() && Character.isDigit(peek())) {
        sb.append(advance());
    }
    return sb.toString();
}
```

```
private boolean isIdentifierStart(char c) {
    return Character.isLetter(c) || c == '_';
}

private boolean isIdentifierPart(char c) {
    return Character.isLetterOrDigit(c) || c == '_';
}

private String readIdentifier(char firstChar) {
    StringBuilder sb = new StringBuilder();
    sb.append(firstChar);
    while (!isAtEnd() && isIdentifierPart(peek())) {
        sb.append(advance());
    }
    return sb.toString();
}

/**
 * Ordnet einen gelesenen Bezeichner entweder einem Schlüsselwort,
 * einem Bool-Literal oder einem "normalen" IDENT zu.
 */
private TokenType classifyIdentifier(String ident) {
    switch (ident) {
        case "if":   return TokenType.IF;
        case "do":   return TokenType.DO;
        case "def":  return TokenType.DEF;
        case "defn": return TokenType.DEFN;
        case "let":  return TokenType.LET;
        case "list": return TokenType.LIST;
        case "nth":  return TokenType.NTH;
        case "head": return TokenType.HEAD;
        case "tail": return TokenType.TAIL;
        case "print":return TokenType.PRINT;
        case "str":  return TokenType.STR;
        case "true":
        case "false":
            return TokenType.BOOL;
        default:
            return TokenType.IDENT;
    }
}

/**
```

```
 * Lies ein String-Literal bis zum abschließenden ".
 * Wenn kein " mehr kommt, wird ein Fehler geworfen.
 */
private String readString(int tokenLine, int tokenColumn) {
    StringBuilder sb = new StringBuilder();
    while (!isAtEnd()) {
        char c = advance();
        if (c == '"') {
            // abschließendes Anführungszeichen
            return sb.toString();
        }
        if (c == '\0') {
            // unerwartetes Ende
            throw expectedCharError('"', c, tokenLine, tokenColumn);
        }
        sb.append(c);
    }
    // EOF erreicht, aber kein schließendes "
    throw expectedCharError('"', '\0', tokenLine, tokenColumn);
}

/**
 * Erzeugt eine RuntimeException für ein komplett unerwartetes Zeichen.
 */
private RuntimeException unexpectedCharError(char found, int line, int column) {
    String msg = "Unerwartetes Zeichen '" + found + "' an Position "
            + line + ":" + column;
    return new RuntimeException(msg);
}

/**
 * Rudimentäre Fehlerbehandlung, wenn ein bestimmtes Zeichen erwartet wurde,
 * tatsächlich aber ein anderes (oder EOF) eingelesen wurde.
 * Dies entspricht der Aufgabenstellung:
 * "… Abweichung vom erwarteten Zeichen zum tatsächlich eingelesenen Zeichen
 */
private RuntimeException expectedCharError(char expected, char found,
                                           int line, int column) {
    String real = (found == '\0') ? "<EOF>" : String.valueOf(found);
    String msg = "Lexer-Fehler: erwartetes Zeichen '" + expected
            + "', aber '" + real + "' gefunden an Position "
            + line + ":" + column;
    return new RuntimeException(msg);
```

```
    }
}
```

Der Lexer arbeitet wie in der Vorlesung gezeigt mit einem zeichenweisen Zugriff auf den Eingabestrom. Über `nextToken()` wird jeweils das nächste Token berechnet. Whitespace und Kommentare werden still entfernt, Fehler bei unerwarteten Zeichen bzw. nicht korrekt abgeschlossenen Strings werden mit einer Meldung *erwartet vs. gefunden* abgebrochen.