Blatt 04 – A4.4 Parser mit recursive descent

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

---

A4.4 – Parser (LL(1), rekursiver Abstieg)

1) Grundstruktur des Parsers

```
/**
 * Rekursiver LL(1)-Parser für die Lisp-artige Sprache.
 *
 * Der Parser arbeitet auf dem Tokenstrom des Lexers aus A4.3 und
 * überprüft, ob der Input zur Grammatik aus A4.2 passt.
 *
 * In A4.4 erzeugen die Methoden noch keinen AST, sondern prüfen
 * nur die Syntax. In A4.5 wird hierauf aufgebaut.
 */
public class Parser {

    private final Lexer lexer;
    private Token lookahead; // aktuelles Lookahead-Token

    public Parser(Lexer lexer) {
        this.lexer = lexer;
        consume(); // erstes Token laden
    }

    /**
     * Haupteinstieg: parst ein komplettes Programm.
     * Wenn am Ende kein EOF steht oder unterwegs ein Fehler auftritt,
     * wird eine RuntimeException mit Fehlermeldung geworfen.
     */
    public void parseProgram() {
        program();
        match(TokenType.EOF); // Programm muss sauber mit EOF enden
    }

    // ------------------------------------------------------------
    // Hilfsfunktionen match() und consume()
    // ------------------------------------------------------------
```

```
private void match(TokenType expected) {
    if (lookahead.getType() == expected) {
        consume();
    } else {
        throw parseError(expected, lookahead);
    }
}

private void consume() {
    lookahead = lexer.nextToken();
}

private RuntimeException parseError(TokenType expected, Token found) {
    String msg = "Parser-Fehler: erwartetes Token " + expected
            + ", aber " + found.getType()
            + " ('" + found.getLexeme() + "') an Position "
            + found.getLine() + ":" + found.getColumn();
    return new RuntimeException(msg);
}

private RuntimeException parseError(String message) {
    String msg = "Parser-Fehler: " + message
            + " (gefunden " + lookahead.getType()
            + " '" + lookahead.getLexeme() + "' an "
            + lookahead.getLine() + ":" + lookahead.getColumn() + ")";
    return new RuntimeException(msg);
}

// ============================================================
// 2) Zu jeder Grammatikregel eine passende Methode
// ============================================================

// ------------------------------------------------------------
// program : expression_list ;
// ------------------------------------------------------------
private void program() {
    expressionList();
}

// ------------------------------------------------------------
// expression_list
//   : expression expression_list
```

```
//   | ε
// ----------------------------------------------------------
private void expressionList() {
    // FIRST(expression) = { INT, STRING, BOOL, IDENT, LPAREN }
    while (isExpressionStart(lookahead.getType())) {
        expression();
    }
    // epsilon erlaubt, daher kein Fehler, wenn nichts mehr passt
}

private boolean isExpressionStart(TokenType t) {
    switch (t) {
        case INT:
        case STRING:
        case BOOL:
        case IDENT:
        case LPAREN:
            return true;
        default:
            return false;
    }
}

// ----------------------------------------------------------
// expression
//   : literal
//   | IDENT
//   | LPAREN form RPAREN
// ----------------------------------------------------------
private void expression() {
    switch (lookahead.getType()) {
        case INT:
        case STRING:
        case BOOL:
            literal();
            break;
        case IDENT:
            // Variable / Funktionsname alleine
            match(TokenType.IDENT);
            break;
        case LPAREN:
            match(TokenType.LPAREN);
            form();
```

```
            match(TokenType.RPAREN);
            break;
        default:
            throw parseError("Unerwarteter Ausdrucksbeginn");
    }
}

// ------------------------------------------------------------
// literal : INT | STRING | BOOL ;
// ------------------------------------------------------------
private void literal() {
    switch (lookahead.getType()) {
        case INT:
            match(TokenType.INT);
            break;
        case STRING:
            match(TokenType.STRING);
            break;
        case BOOL:
            match(TokenType.BOOL);
            break;
        default:
            throw parseError("Literal erwartet");
    }
}

// ------------------------------------------------------------
// form
//   : IF expression expression opt_expression
//   | DO expression_list
//   | DEF IDENT expression
//   | DEFN IDENT LPAREN param_list RPAREN expression
//   | LET LPAREN let_bindings RPAREN expression
//   | LIST expression_list_nonempty
//   | NTH expression expression
//   | HEAD expression
//   | TAIL expression
//   | function_call
//
// Der erste Token nach '(' entscheidet die Alternative.
// ------------------------------------------------------------
private void form() {
    switch (lookahead.getType()) {
```

```
    case IF:
        parseIfForm();
        break;
    case DO:
        parseDoForm();
        break;
    case DEF:
        parseDefForm();
        break;
    case DEFN:
        parseDefnForm();
        break;
    case LET:
        parseLetForm();
        break;
    case LIST:
        parseListForm();
        break;
    case NTH:
        parseNthForm();
        break;
    case HEAD:
        parseHeadForm();
        break;
    case TAIL:
        parseTailForm();
        break;
    // Operatoren oder normaler Funktionsname
    case PLUS:
    case MINUS:
    case TIMES:
    case DIV:
    case EQ:
    case LT:
    case GT:
    case PRINT:
    case STR:
    case IDENT:
        functionCall();
        break;
    default:
        throw parseError("Ungültige Form innerhalb von Klammern");
}
```

```
}

// ---------------------------------------------------------
// if-Form: (if cond then [else])
// ---------------------------------------------------------
private void parseIfForm() {
    match(TokenType.IF);
    expression();        // Bedingung
    expression();        // then-Zweig
    optExpression();     // optionaler else-Zweig
}

// opt_expression : expression | ε ;
private void optExpression() {
    if (isExpressionStart(lookahead.getType())) {
        expression();
    } else {
        // epsilon
    }
}

// ---------------------------------------------------------
// do-Form: (do e1 e2 ...)
// ---------------------------------------------------------
private void parseDoForm() {
    match(TokenType.DO);
    expressionList();
}

// ---------------------------------------------------------
// def-Form: (def name value)
// ---------------------------------------------------------
private void parseDefForm() {
    match(TokenType.DEF);
    match(TokenType.IDENT);
    expression();
}

// ---------------------------------------------------------
// defn-Form: (defn name (params) body)
// ---------------------------------------------------------
private void parseDefnForm() {
    match(TokenType.DEFN);
```

```
    match(TokenType.IDENT);
    match(TokenType.LPAREN);
    paramList();
    match(TokenType.RPAREN);
    expression(); // Funktionskörper
}

// param_list : IDENT param_list_tail | ε ;
private void paramList() {
    if (lookahead.getType() == TokenType.IDENT) {
        match(TokenType.IDENT);
        paramListTail();
    } else {
        // epsilon
    }
}

// param_list_tail : IDENT param_list_tail | ε ;
private void paramListTail() {
    while (lookahead.getType() == TokenType.IDENT) {
        match(TokenType.IDENT);
    }
}

// ------------------------------------------------------------
// let-Form: (let (name value ...) body)
// ------------------------------------------------------------
private void parseLetForm() {
    match(TokenType.LET);
    match(TokenType.LPAREN);
    letBindings();
    match(TokenType.RPAREN);
    expression(); // body
}

// let_bindings : IDENT expression let_bindings_tail ;
private void letBindings() {
    match(TokenType.IDENT);
    expression();
    letBindingsTail();
}

// let_bindings_tail : IDENT expression let_bindings_tail | ε
```

```
private void letBindingsTail() {
    while (lookahead.getType() == TokenType.IDENT) {
        match(TokenType.IDENT);
        expression();
    }
}

// ---------------------------------------------------------
// list-Form: (list e1 e2 ...)
// ---------------------------------------------------------
private void parseListForm() {
    match(TokenType.LIST);
    expressionListNonEmpty();
}

// expression_list_nonempty : expression expression_list ;
private void expressionListNonEmpty() {
    expression();
    expressionList();
}

// ---------------------------------------------------------
// nth-Form: (nth list-expr index-expr)
// ---------------------------------------------------------
private void parseNthForm() {
    match(TokenType.NTH);
    expression(); // Listenausdruck
    expression(); // Indexausdruck
}

// ---------------------------------------------------------
// head-Form: (head list-expr)
// ---------------------------------------------------------
private void parseHeadForm() {
    match(TokenType.HEAD);
    expression();
}

// ---------------------------------------------------------
// tail-Form: (tail list-expr)
// ---------------------------------------------------------
private void parseTailForm() {
    match(TokenType.TAIL);
```

```
        expression();
    }

    // ---------------------------------------------------------
    // function_call
    //   : operator expression_list_nonempty
    //   | IDENT expression_list_nonempty
    // ---------------------------------------------------------
    private void functionCall() {
        if (isOperator(lookahead.getType())) {
            // operator-Call, z.B. (+ 1 2 3)
            match(lookahead.getType()); // PLUS/MINUS/...
            expressionListNonEmpty();
        } else if (lookahead.getType() == TokenType.IDENT) {
            // Funktionsaufruf, z.B. (foo 1 2)
            match(TokenType.IDENT);
            expressionListNonEmpty();
        } else {
            throw parseError("Funktions- oder Operatoraufruf erwartet");
        }
    }

    private boolean isOperator(TokenType t) {
        switch (t) {
            case PLUS:
            case MINUS:
            case TIMES:
            case DIV:
            case EQ:
            case LT:
            case GT:
            case PRINT:
            case STR:
                return true;
            default:
                return false;
        }
    }
}
```

3) Nutzung des Parsers

```
public class Main {
    public static void main(String[] args) {
        String source = "(def x 42)\n" +
                        "(print (+ x 1))\n";

        Lexer lexer = new Lexer(source);
        Parser parser = new Parser(lexer);

        // Wirft eine Exception, falls ein Syntaxfehler auftritt:
        parser.parseProgram();

        System.out.println("Programm ist syntaktisch korrekt.");
    }
}
```

Der Parser folgt exakt dem in der Vorlesung gezeigten Muster: Zu jeder Grammatikregel gibt es eine Methode. Über `lookahead` und `match()` wird das nächste Token geprüft und konsumiert. Die Fehlerbehandlung gibt die Abweichung zwischen erwartetem und tatsächlich gefundenem Token mit Zeile/Spalte aus. In A4.5 können die Methoden dann so erweitert werden, dass sie statt `void` passende AST-Knoten zurückliefern.