Blatt 04 – A4.5 AST

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

---

A4.5 – AST-Strukturen und Transformation

1) Welche Informationen muss der AST enthalten?

Der AST soll die inhaltlich relevanten Informationen des Programms repräsentieren, ohne rein syntaktische Klammern und Schlüsselwörter, die für spätere Phasen (Code-Generierung, Interpretation) nicht mehr nötig sind.

Wichtige Informationen:

- Literale:
  - Integer-Werte (z.B. 42)
  - String-Werte (z.B. "hello")
  - Bool-Werte (true/false)

- Variablen-Referenzen:
  - Name der Variable (IDENT)

- Funktions- und Operatoraufrufe:
  - Name der Funktion oder des Operators (+, -, /, print, str, ...)
  - Liste der Argument-Ausdrücke

- Kontrollstrukturen:
  - if:
    - Bedingungs-Ausdruck
    - then-Zweig
    - optionaler else-Zweig
  - do:
    - Liste von Ausdrücken im Block (Reihenfolge wichtig)

- Bindungen:
  - def:
    - Name
    - Wert-Ausdruck
  - defn:
    - Funktionsname

    - Parameterliste (Namen)
    - Rumpf-Ausdruck
  - let:
    - Liste von lokalen Bindungen (Name + Ausdruck)
    - Body-Ausdruck

- Listenoperationen:
  - list:
    - Elemente als Liste von Ausdrücken
  - nth:
    - Listen-Ausdruck
    - Index-Ausdruck
  - head / tail:
    - Listen-Ausdruck

- Programm:
  - Folge von Top-Level-Ausdrücken (z.B. mehrere def/defn/print-Aufrufe)

2) AST-Datenstrukturen in Java

```
import java.util.List;

/**
 * Gemeinsame Basis für alle AST-Knoten.
 * (Optional könnte man hier auch Positionen aus dem Token übernehmen.)
 */
public interface AstNode {
}

/**
 * Oberklasse für alle Ausdrücke.
 */
public abstract class ExprNode implements AstNode {
}

/**
 * Programm: Liste von Top-Level-Ausdrücken.
 */
public class ProgramNode implements AstNode {
    private final List<ExprNode> expressions;
```

```
    public ProgramNode(List<ExprNode> expressions) {
        this.expressions = expressions;
    }

    public List<ExprNode> getExpressions() {
        return expressions;
    }
}

/* ------------------------ Literale ------------------------ */

public class IntLiteralNode extends ExprNode {
    private final int value;

    public IntLiteralNode(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

public class StringLiteralNode extends ExprNode {
    private final String value;

    public StringLiteralNode(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}

public class BoolLiteralNode extends ExprNode {
    private final boolean value;

    public BoolLiteralNode(boolean value) {
        this.value = value;
    }

    public boolean getValue() {
```

```
        return value;
    }
}

/* ------------------ Variablen-Referenzen ------------------ */

public class VarRefNode extends ExprNode {
    private final String name;

    public VarRefNode(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

/* ------------------ Funktions-/Operatoraufrufe ------------------ */

public class CallNode extends ExprNode {
    private final String callee;         // Name des Operators oder der Funktion
    private final List<ExprNode> args;  // Argumente

    public CallNode(String callee, List<ExprNode> args) {
        this.callee = callee;
        this.args = args;
    }

    public String getCallee() {
        return callee;
    }

    public List<ExprNode> getArgs() {
        return args;
    }
}

/* --------------------- Kontrollstrukturen --------------------- */

public class IfNode extends ExprNode {
    private final ExprNode condition;
    private final ExprNode thenBranch;
```

```
    private final ExprNode elseBranch; // kann null sein

    public IfNode(ExprNode condition, ExprNode thenBranch, ExprNode elseBranch)
        this.condition = condition;
        this.thenBranch = thenBranch;
        this.elseBranch = elseBranch;
    }

    public ExprNode getCondition() {
        return condition;
    }

    public ExprNode getThenBranch() {
        return thenBranch;
    }

    public ExprNode getElseBranch() {
        return elseBranch;
    }
}

public class DoNode extends ExprNode {
    private final List<ExprNode> expressions;

    public DoNode(List<ExprNode> expressions) {
        this.expressions = expressions;
    }

    public List<ExprNode> getExpressions() {
        return expressions;
    }
}

/* --------------------- Bindungen (def / defn / let) --------------------- */

public class DefNode extends ExprNode {
    private final String name;
    private final ExprNode value;

    public DefNode(String name, ExprNode value) {
        this.name = name;
        this.value = value;
    }
```

```
    public String getName() {
        return name;
    }

    public ExprNode getValue() {
        return value;
    }
}

public class DefnNode extends ExprNode {
    private final String name;
    private final List<String> params;
    private final ExprNode body;

    public DefnNode(String name, List<String> params, ExprNode body) {
        this.name = name;
        this.params = params;
        this.body = body;
    }

    public String getName() {
        return name;
    }

    public List<String> getParams() {
        return params;
    }

    public ExprNode getBody() {
        return body;
    }
}

public class LetBinding {
    private final String name;
    private final ExprNode value;

    public LetBinding(String name, ExprNode value) {
        this.name = name;
        this.value = value;
    }
```

```
    public String getName() {
        return name;
    }

    public ExprNode getValue() {
        return value;
    }
}

public class LetNode extends ExprNode {
    private final List<LetBinding> bindings;
    private final ExprNode body;

    public LetNode(List<LetBinding> bindings, ExprNode body) {
        this.bindings = bindings;
        this.body = body;
    }

    public List<LetBinding> getBindings() {
        return bindings;
    }

    public ExprNode getBody() {
        return body;
    }
}

/* ---------------------- Listen-Konstrukte ---------------------- */

public class ListLiteralNode extends ExprNode {
    private final List<ExprNode> elements;

    public ListLiteralNode(List<ExprNode> elements) {
        this.elements = elements;
    }

    public List<ExprNode> getElements() {
        return elements;
    }
}

public class NthNode extends ExprNode {
    private final ExprNode listExpr;
```

```
    private final ExprNode indexExpr;

    public NthNode(ExprNode listExpr, ExprNode indexExpr) {
        this.listExpr = listExpr;
        this.indexExpr = indexExpr;
    }

    public ExprNode getListExpr() {
        return listExpr;
    }

    public ExprNode getIndexExpr() {
        return indexExpr;
    }
}

public class HeadNode extends ExprNode {
    private final ExprNode listExpr;

    public HeadNode(ExprNode listExpr) {
        this.listExpr = listExpr;
    }

    public ExprNode getListExpr() {
        return listExpr;
    }
}

public class TailNode extends ExprNode {
    private final ExprNode listExpr;

    public TailNode(ExprNode listExpr) {
        this.listExpr = listExpr;
    }

    public ExprNode getListExpr() {
        return listExpr;
    }
}
```

3) Anpassung des Parsers: Parse-Tree → AST

```
/**
 * Variante des Parsers aus A4.4, bei der die Methoden
 * AST-Knoten zurückliefern, statt nur die Syntax zu prüfen.
 *
 * Zur Übersicht sind nur zentrale Methoden gezeigt (program, expression,
 * form, if, do, def, defn, let, functionCall, list).
 */
public class AstParser {

    private final Lexer lexer;
    private Token lookahead;

    public AstParser(Lexer lexer) {
        this.lexer = lexer;
        consume();
    }

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

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

    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);
    }

    /* ------------------------ Einstieg ------------------------ */

    public ProgramNode parseProgram() {
        List<ExprNode> exprs = expressionList();
        match(TokenType.EOF);
```

```
    return new ProgramNode(exprs);
}

/* ------------------------ expression_list ------------------------ */

private List<ExprNode> expressionList() {
    List<ExprNode> result = new java.util.ArrayList<>();
    while (isExpressionStart(lookahead.getType())) {
        result.add(expression());
    }
    return result;
}

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

/* --------------------------- expression --------------------------- */

private ExprNode expression() {
    switch (lookahead.getType()) {
        case INT:
        case STRING:
        case BOOL:
            return literal();
        case IDENT:
            String name = lookahead.getLexeme();
            match(TokenType.IDENT);
            return new VarRefNode(name);
        case LPAREN:
            match(TokenType.LPAREN);
            ExprNode e = form();
            match(TokenType.RPAREN);
            return e;
```

```
        default:
            throw new RuntimeException("Unerwarteter Ausdrucksbeginn: " + lookahe
    }
}

/* ---------------------------- literal ---------------------------- */

private ExprNode literal() {
    switch (lookahead.getType()) {
        case INT:
            int value = Integer.parseInt(lookahead.getLexeme());
            match(TokenType.INT);
            return new IntLiteralNode(value);
        case STRING:
            String s = lookahead.getLexeme();
            match(TokenType.STRING);
            return new StringLiteralNode(s);
        case BOOL:
            boolean b = Boolean.parseBoolean(lookahead.getLexeme());
            match(TokenType.BOOL);
            return new BoolLiteralNode(b);
        default:
            throw new RuntimeException("Literal erwartet, gefunden: " + lookahead);
    }
}

/* ----------------------------- form ----------------------------- */

private ExprNode form() {
    switch (lookahead.getType()) {
        case IF:
            return parseIfForm();
        case DO:
            return parseDoForm();
        case DEF:
            return parseDefForm();
        case DEFN:
            return parseDefnForm();
        case LET:
            return parseLetForm();
        case LIST:
            return parseListForm();
        case NTH:
```

```
            return parseNthForm();
        case HEAD:
            return parseHeadForm();
        case TAIL:
            return parseTailForm();
        case PLUS:
        case MINUS:
        case TIMES:
        case DIV:
        case EQ:
        case LT:
        case GT:
        case PRINT:
        case STR:
        case IDENT:
            return functionCall();
        default:
            throw new RuntimeException("Ungültige Form in Klammern: " + lookahead
    }
}

/* ------------------------- if / do / def ------------------------- */

private ExprNode parseIfForm() {
    match(TokenType.IF);
    ExprNode cond = expression();
    ExprNode thenBranch = expression();
    ExprNode elseBranch = null;
    if (isExpressionStart(lookahead.getType())) {
        elseBranch = expression();
    }
    return new IfNode(cond, thenBranch, elseBranch);
}

private ExprNode parseDoForm() {
    match(TokenType.DO);
    List<ExprNode> exprs = expressionList();
    return new DoNode(exprs);
}

private ExprNode parseDefForm() {
    match(TokenType.DEF);
    String name = lookahead.getLexeme();
```

```
    match(TokenType.IDENT);
    ExprNode value = expression();
    return new DefNode(name, value);
}

private ExprNode parseDefnForm() {
    match(TokenType.DEFN);
    String name = lookahead.getLexeme();
    match(TokenType.IDENT);
    match(TokenType.LPAREN);
    List<String> params = paramList();
    match(TokenType.RPAREN);
    ExprNode body = expression();
    return new DefnNode(name, params, body);
}

private List<String> paramList() {
    List<String> params = new java.util.ArrayList<>();
    while (lookahead.getType() == TokenType.IDENT) {
        params.add(lookahead.getLexeme());
        match(TokenType.IDENT);
    }
    return params;
}

/* ------------------------------ let ------------------------------ */

private ExprNode parseLetForm() {
    match(TokenType.LET);
    match(TokenType.LPAREN);
    List<LetBinding> bindings = letBindings();
    match(TokenType.RPAREN);
    ExprNode body = expression();
    return new LetNode(bindings, body);
}

private List<LetBinding> letBindings() {
    List<LetBinding> result = new java.util.ArrayList<>();
    // mindestens eine Bindung
    do {
        String name = lookahead.getLexeme();
        match(TokenType.IDENT);
        ExprNode value = expression();
```

```
            result.add(new LetBinding(name, value));
        } while (lookahead.getType() == TokenType.IDENT);
        return result;
    }

    /* ---------------------------- list/nth/head/tail ---------------------------- */

    private ExprNode parseListForm() {
        match(TokenType.LIST);
        List<ExprNode> elems = expressionListNonEmpty();
        return new ListLiteralNode(elems);
    }

    private List<ExprNode> expressionListNonEmpty() {
        List<ExprNode> result = new java.util.ArrayList<>();
        result.add(expression());
        result.addAll(expressionList());
        return result;
    }

    private ExprNode parseNthForm() {
        match(TokenType.NTH);
        ExprNode listExpr = expression();
        ExprNode idxExpr = expression();
        return new NthNode(listExpr, idxExpr);
    }

    private ExprNode parseHeadForm() {
        match(TokenType.HEAD);
        ExprNode listExpr = expression();
        return new HeadNode(listExpr);
    }

    private ExprNode parseTailForm() {
        match(TokenType.TAIL);
        ExprNode listExpr = expression();
        return new TailNode(listExpr);
    }

    /* ------------------------- function_call ------------------------- */

    private ExprNode functionCall() {
        String callee;
```

```
        if (isOperator(lookahead.getType())) {
            callee = tokenTypeToName(lookahead.getType());
            match(lookahead.getType());
        } else {
            callee = lookahead.getLexeme();
            match(TokenType.IDENT);
        }
        List<ExprNode> args = expressionListNonEmpty();
        return new CallNode(callee, args);
    }

    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;
        }
    }

    private String tokenTypeToName(TokenType t) {
        switch (t) {
            case PLUS:  return "+";
            case MINUS: return "-";
            case TIMES: return "*";
            case DIV:   return "/";
            case EQ:    return "=";
            case LT:    return "<";
            case GT:    return ">";
            case PRINT: return "print";
            case STR:   return "str";
            default:
                return t.name();
        }
    }
```

```
}
```

4) Test mit Beispielprogrammen

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

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

        ProgramNode program = parser.parseProgram();

        System.out.println("AST erfolgreich aufgebaut.");
        System.out.println("Anzahl Top-Level-Ausdrücke: "
                           + program.getExpressions().size());
    }
}
```

Der AST entfernt überflüssige Syntax (z. B. äußere Klammern) und speichert nur die Struktur, die später für Interpretation oder Codegenerierung gebraucht wird. Die Transformation vom Parse-Tree in diesen AST passiert hier direkt im Parser: jede Methode gibt statt `void` einen passenden AST-Knoten zurück.