Aufgabe A5.1 – Grammatik und AST

November 21, 2025

Aufgabe A5.1 – Grammatik und AST

(a) Angepasste Grammatik

Die ursprüngliche MiniC-Grammatik wurde erweitert um

- boolesche Literale `T` und `F`,
- den Typ `bool`,
- gelabelte Alternativen für Ausdrücke zur besseren AST-Erzeugung.

Die vollständige ANTLR Grammatik sieht wie folgt aus:

```
grammar MiniC;

// Parser
program : stmt* EOF ;

stmt
    : vardecl
    | assign
    | fndecl
    | expr ';'
    | block
    | whileStmt
    | ifStmt
    | returnStmt
    ;

vardecl : type ID ('=' expr)? ';' ;
assign  : ID '=' expr ';' ;

fndecl  : type ID '(' params? ')' block ;
params  : param (',' param)* ;
param   : type ID ;
returnStmt : 'return' expr ';' ;

fncall  : ID '(' args? ')' ;
```

```
args     : expr (',' expr)* ;

block    : '{' stmt* '}' ;
whileStmt : 'while' '(' expr ')' block ;
ifStmt : 'if' '(' expr ')' block ('else' block)? ;

expr
    : fncall                                          # FnCallExpr
    | left=expr op=('*'|'/') right=expr               # MulDivExpr
    | left=expr op=('+'|'-') right=expr               # AddSubExpr
    | left=expr op=('=='|'!='|'>'|'<') right=expr     # CmpExpr
    | ID                                              # VarExpr
    | NUMBER                                          # IntLitExpr
    | STRING                                          # StringLitExpr
    | BOOL                                            # BoolLitExpr
    | '(' expr ')'                                    # ParenExpr
    ;

type : 'int' | 'string' | 'bool' ;

// Lexer
ID       : [a-z][a-zA-Z0-9]* ;
NUMBER   : [0-9]+ ;
STRING   : '"' (~[\n\r"])* '"' ;
BOOL     : 'T' | 'F' ;

COMMENT : '#' ~[\n\r]* -> skip ;
WS       : [ \t\n\r]+ -> skip ;
```

(b) AST-Struktur

Für die semantische Analyse werden folgende AST-Knoten benötigt:

- **Programmknoten**: Liste von Statements
- **Statements**: Variablendeklarationen, Zuweisungen, Funktionsdeklarationen, Blöcke, Kontrollstrukturen, Return
- **Ausdrücke**: Binäre Operatoren, Variablen, Literale, Funktionsaufruf
- **Typknoten**: `int`, `string`, `bool`

Die Java-Implementierung des AST sieht wie folgt aus:

```
// Basis
public interface AstNode {
    int getLine();
    int getColumn();
}
```

```
public abstract class Stmt implements AstNode { }
public abstract class Expr implements AstNode { }

// Programm
public class Program implements AstNode {
    public final List<Stmt> statements;
    public Program(List<Stmt> statements) {
        this.statements = statements;
    }
    public int getLine() { return 0; }
    public int getColumn() { return 0; }
}

// Typen
public enum TypeName { INT, STRING, BOOL; }

public class TypeNode implements AstNode {
    public final TypeName name;
    public TypeNode(TypeName name) { this.name = name; }
    public int getLine() { return 0; }
    public int getColumn() { return 0; }
}
```

Statements

```
// int x = 5;
public class VarDecl extends Stmt {
    public final TypeNode type;
    public final String name;
    public final Expr init;
    private final int line, column;

    public VarDecl(TypeNode type, String name, Expr init, int line, int colu
        this.type = type;
        this.name = name;
        this.init = init;
        this.line = line;
        this.column = column;
    }
    public int getLine() { return line; }
    public int getColumn() { return column; }
}

// x = expr;
public class Assign extends Stmt {
    public final String name;
    public final Expr value;
    private final int line, column;
```

```
    public Assign(String name, Expr value, int line, int column) {
        this.name = name;
        this.value = value;
        this.line = line;
        this.column = column;
    }
}
```

Kontrollstrukturen

```
public class WhileStmt extends Stmt {
    public final Expr cond;
    public final Block body;
    ...
}

public class IfStmt extends Stmt {
    public final Expr cond;
    public final Block thenBlock;
    public final Block elseBlock;
    ...
}
```

Ausdrücke

```
public enum BinaryOp { MUL, DIV, ADD, SUB, EQ, NEQ, GT, LT; }

public class BinaryExpr extends Expr {
    public final Expr left, right;
    public final BinaryOp op;
    ...
}

public class VarExpr extends Expr {
    public final String name;
    ...
}

public class BoolLiteral extends Expr {
    public final boolean value;
    ...
}

public class CallExpr extends Expr {
    public final String functionName;
    public final List<Expr> args;
```

```
    ...
}
```

(c) Konstruktion des AST aus dem Parse Tree

Der AST wird in einem Visitor aufgebaut:

```
public class AstBuilder extends MiniCBaseVisitor<AstNode> {

    @Override
    public AstNode visitProgram(ProgramContext ctx) {
        List<Stmt> stmts = new ArrayList<>();
        for (var s : ctx.stmt()) {
            stmts.add((Stmt) visit(s));
        }
        return new Program(stmts);
    }

    @Override
    public AstNode visitMulDivExpr(MulDivExprContext ctx) {
        Expr left = (Expr) visit(ctx.left);
        Expr right = (Expr) visit(ctx.right);
        BinaryOp op = ctx.op.getText().equals("*")
                      ? BinaryOp.MUL : BinaryOp.DIV;
        return new BinaryExpr(left, op, right,
                              ctx.start.getLine(),
                              ctx.start.getCharPositionInLine());
    }
}
```

Dieses Visitor übersetzt jede Parse-Tree-Node in ein geeignetes AST-Objekt.