Bericht Sprachkonzepte

Marc Bohner, Stefan Willmann

Abgabe 1

Aufgabe

Es sollte ein Text mit ANTLR4 Lexer-Regeln beschrieben und eine Anwendung erstellt werden, die den Text einliest und als Tokenfolge ausgibt.

Vorgehensweise

Zur Bearbeitung haben wir uns für den Text "abfahrt-kn.txt" entschieden. In diesem Text geht es um die Abfahrtszeiten von Bussen, Zügen sowie Schiffen mit Zielort und genauen Datumsangaben mit Wochentagen, Monaten, Jahren sowie Ausnahmen von Tagen an denen das Verkehrsmittel nicht fährt.

1. Lexer Regeln (AbfahrtenLexer.g4) Die Lexer Regeln wurden in der Datei "AbfahrtenLexer.g4" definiert. Hierbei wurde der Text in verschiedene Token unterteilt. Die Token sind:

```
lexer grammar AbfahrtenLexer;

// Define tokens for specific words
EXCEPTION: 'nicht';
DAILY: 'täglich';
ALSO: 'auch';
UNTIL: 'bis';
TO: 'nach';

// Define tokens for special characters
MINUS: '-';
COMMA: ',';
SEMICOLON: ';';
SLASH: '/';
LPAREN: '(';
RPAREN: ')';

// Define tokens for days, months and years
WEEKDAY: 'Mo' | 'Di' | 'Mi' | 'Do' | 'Fr' | 'Sa' | 'So';
DAY: ([1-9] | [12][0-9] | '30' | '31')'.';
MONTH: 'Jan' | 'Feb' | 'Mär' | 'Apr' | 'Mai' | 'Jun' | 'Jul' | 'Aug' | 'Sep' | 'Okt' | 'Nov'
YEAR: [0-9]+;

// Define tokens for time, transport types, and destinations
TIME: ([0-1][0-9] | [2][0-3]) ':' [0-5][0-9];
```

```
BUS: 'Bus ' [0-9]+;
TRAIN: 'RE ' [0-9]+;
SHIP: 'KAT';
DESTINATION: [a-zA-ZäöüÄÖÜß/()]+;


// Whitespace handling
WS: [ \t\r\n]+ -> channel(HIDDEN);
```

Durch die Ausführung des Befehls `antlr4 AbfahrtenLexer.g4` wird der Lexer generiert.

2. Expression Tokenizer (ExpressionTokenizer.java) Die Klasse `ExpressionTokenizer` wurde erstellt, um die Tokenfolge aus dem Text zu generieren. Dazu wird die Datei "abfahrt-kn.txt" eingelesen, anschließend die generierte Klassen `AbfahrtenLexer` aufgerufen, um die Token zu generieren und die Tokenfolge ausgegeben.

```java
public class ExpressionTokenizer {
    public static void main(String[] args) throws IOException {
        String inputFile = "Abgabe1/abfahrt-kn.txt";
        String input = new String(Files.readAllBytes(Paths.get(inputFile)));

        AbfahrtenLexer lexer = new AbfahrtenLexer(CharStreams.fromString(input));
        var tokens = lexer.getAllTokens();

        for (Token t : tokens) {
            if (t.getChannel() == Token.HIDDEN_CHANNEL) {
                continue;
            }
            System.out.printf(
                    "%s(\"%s\") ",
                    lexer.getVocabulary().getSymbolicName(t.getType()),
                    t.getText());
        }
        System.out.println();
    }

    private static String replaceWhitespace(String s) {
        return s.replace("\n", "\\n").replace("\r", "\\r").replace("\t", "\\t");
    }
}
```

3. Ergebnis In der Konsole beim Ausführen des Programms wird die Tokenfolge ausgegeben. Diese sieht wie folgt aus:

```
TIME("09:45") BUS("Bus 700") TO("nach") DESTINATION("Bahnhof") COMMA(",") DESTINATION("Raver
```

```
WEEKDAY("Mo") MINUS("-") WEEKDAY("Mi")
EXCEPTION("nicht") DAY("20.") MONTH("Mai")

TIME("09:46") BUS("Bus 1") TO("nach") DESTINATION("Staad/Autofähre") COMMA(",") DESTINATION(
WEEKDAY("Mo") MINUS("-") WEEKDAY("Fr")
EXCEPTION("nicht") DAY("9.") COMMA(",") DAY("20.") COMMA(",") DAY("30.") MONTH("Mai") COMMA(

TIME("09:48") BUS("Bus 9") TO("nach") DESTINATION("Universität") COMMA(",") DESTINATION("Kon
EXCEPTION("nicht") DAILY("täglich")
DAY("27.") MONTH("Mai") UNTIL("bis") DAY("18.") MONTH("Okt") YEAR("2024") WEEKDAY("Mo") MINU

TIME("10:00") SHIP("KAT") TO("nach") DESTINATION("Friedrichshafen") DESTINATION("Hafen") DES
DAILY("täglich")

TIME("10:39") TRAIN("RE 4720") TO("nach") DESTINATION("Karlsruhe") DESTINATION("Hbf")
DAILY("täglich")
EXCEPTION("nicht") DAY("11.") COMMA(",") DAY("12.") MONTH("Mai") COMMA(",") DAY("1.") MONTH(
```

Probleme

1. Die Library von ANTLR4 wurde zuerst nicht richtig erkannt. Dieses Problem konnte durch Importieren der Library in IntelliJ gelöst werden.
2. Wörter wie “bis”, “nicht”, “nach” usw. wurden als “DESTINATION” Token erkannt. Dieses Problem konnte durch explizite Angabe der Keywörter und Beachtung der Reihenfolge der Token in der Lexer-Datei gelöst werden.
3. Allgemeines Verständnis zum Unterschied von Lexer und Parser. Unser erster Versuch war direkt Regeln mit der richtigen Reihenfolge, also z.B. TIME, BUS, DESTINATION zu definieren, was allerdings bereits einem Parser gleicht und für Lexer Regeln zu komplex gedacht ist.

Vokabular-Kategorien aus VL Folie 2-4 für den Text

In diesem Text wurden folgende Kategorien für die Token verwendet:

1. Zwischenraum (whitespace) für Leerzeichen, Tabs und Zeilenumbrüche
2. Schlüsselwörter für Wörter wie bis, nach, täglich und nicht
3. Trennzeichen für Komma, Minus (welches bis bedeutet), Semikolon usw.
4. Literale für Zeichenkette wie die Destination, Monate, Tage usw.

Abgabe 2

Aufgabe

Für den Teil a) der zweiten Abgabe sollte sich eine kleine Sprache ausgedacht werden, für welche ein Lexer und Parser erstellt werden soll. Anschließend sollte für einige Beispieltexte der Parse Tree mit org.antlr.v4.gui.TestRig visualisiert werden. Im Teil b) sollte die abstrakte Syntax aus Teil a) definiert und ein Java-Programm zur Überführung des Parse Trees in einen abstrakten Syntaxbaum erstellt werden.

Vorgehensweise

a) Lexer und Parser Für den Teil a) haben wir uns für eine simple Programmiersprache entschieden, welche simple Anweisungen wie if-else, print und Rechenoperationen unterstützt. Die Art der Syntax ist sehr JavaScript ähnlich.

Folgende Beispieltexte werden von der Sprache unterstützt:

```
// Test 1.txt
let x = 5;
print(x);
```

```
// Test 2.txt
let y = 10;
if (y > 5) {
  print(y);
} else {
  print(0);
}
```

```
// Test 3.txt
let a = 3;
let b = a + 2;
print(b);
```

Um die Sprache in Token zu unterteilen, wurden folgende Lexer Regeln definiert:

```
// SimpleLangLexer.g4
lexer grammar SimpleLangLexer;

// Schlüsselwörter
LET: 'let';
IF: 'if';
ELSE: 'else';
PRINT: 'print';

// Bezeichner
ID: [a-zA-Z_] [a-zA-Z_0-9]*;
```

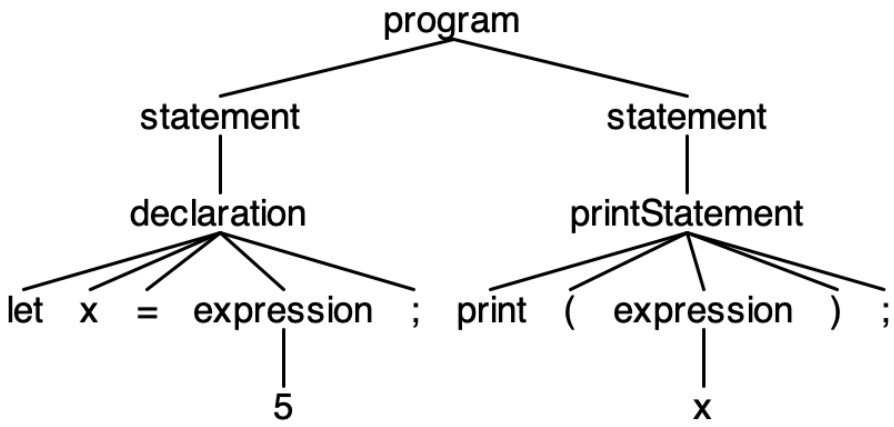
```
// Zahlen
NUMBER: [0-9]+;

// Operatoren
PLUS: '+';
MINUS: '-';
STAR: '*';
SLASH: '/';
EQ: '==';
NEQ: '!=';
LT: '<';
GT: '>';
LE: '<=';
GE: '>=';

// Trennzeichen
SEMI: ';';
ASSIGN: '=';
LPAREN: '(';
RPAREN: ')';
LBRACE: '{';
RBRACE: '}';

// Whitespace
WS: [ \t\r\n]+ -> skip;
```

Außerdem haben wir folgende Parser Regeln definiert:

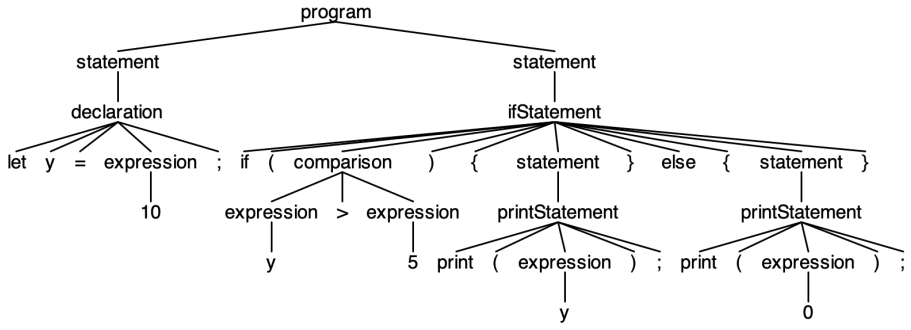
```
// SimpleLangParser.g4
parser grammar SimpleLangParser;

options { tokenVocab = SimpleLangLexer; }

program: statement+;

statement:
    declaration
    | printStatement
    | ifStatement
    ;

declaration: LET ID ASSIGN expression SEMI;

printStatement: PRINT LPAREN expression RPAREN SEMI;

ifStatement: IF LPAREN comparison RPAREN LBRACE statement+ RBRACE (ELSE LBRACE statement+ RB
```

```
expression:
    ID
    | NUMBER
    | expression PLUS expression
    | expression MINUS expression
    | expression STAR expression
    | expression SLASH expression
    ;

comparison:
    expression (EQ | NEQ | LT | GT | LE | GE) expression
    ;
```

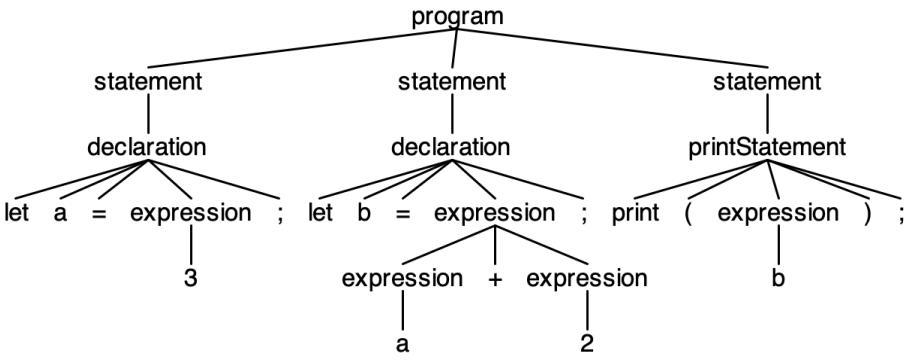
Nach Ausführen von `antlr4 SimpleLangLexer.g4` und `antlr4 SimpleLangParser.g4` konnten die erstellten Dateien mit javac *.java kompiliert werden. Anschließend konnte mit dem Command `grun SimpleLang program -gui <Testfile>` der Parse Tree visualisiert werden. Dies haben wir für alle 3 Testfiles durchgeführt.



Hier das Ergebnis für Test 1.txt:



Ergebnis für Test 2.txt:



Ergebnis für Test 3.txt:

Probleme a) Bei der Erstellung des Parse Trees gab es Schwierigkeiten mit der Ausführung von TestRig. Hierbei wurde der Befehl `grun` anfangs nicht erkannt. Dies lag an einem Problem beim Hinzufügen des Alias in der zsh config. Dabei wurde der Alias nicht richtig angelegt und musste durch Öffnen der .zshrc Datei und Hinzufügen des Alias manuell gelöst werden. Außerdem hatten wir Probleme beim Ausführen des grun Befehls, da wir es zuerst mit dem SimpleLangLexer versucht haben. Allerdings wurde nur `SimpleLang` im command benötigt.

b) AST in Java Für den Teil b) haben wir die abstrakte Syntax definiert und ein Java-Programm erstellt, welches den Parse Tree in einen abstrakten Syntaxbaum überführt. Um die abstrakte Syntax in Java zu definieren haben wir die einzelnen Klassen für die verschiedenen Anweisungen, wie z.B. expressions, if-else, usw. erstellt.

Außerdem haben wir eine Klasse `SimpleLangBuilder.java` erstellt, welche Methoden zum traversieren des Parse Trees enthält. Dabei wird für jede Anweisung im Parse Tree eine Methode aufgerufen, welche die entsprechende abstrakte Syntax Klasse erstellt und zurückgibt.

```java
// SimpleLangBuilder.java
public class SimpleLangBuilder extends SimpleLangParserBaseListener {
    private final Stack<ASTNode> stack = new Stack<>();

    public ASTNode build(ParseTree tree) {
        new ParseTreeWalker().walk(this, tree);
        return this.stack.pop();
    }

    @Override
    public void exitDeclaration(SimpleLangParser.DeclarationContext ctx) {
        String id = ctx.ID().getText();
        ExpressionNode expr = (ExpressionNode) this.stack.pop();
        this.stack.push(new DeclarationNode(id, expr));
    }
```

```java
@Override
public void exitPrintStatement(SimpleLangParser.PrintStatementContext ctx) {
    ExpressionNode expr = (ExpressionNode) this.stack.pop();
    this.stack.push(new PrintNode(expr));
}

@Override
public void exitIfStatement(SimpleLangParser.IfStatementContext ctx) {
    List<StatementNode> elseBranch = new ArrayList<>();
    if (ctx.ELSE() != null) {
        for (SimpleLangParser.StatementContext stmtCtx : ctx.statement().subList(1, ctx.
            elseBranch.addFirst((StatementNode) this.stack.pop());
        }
    }

    List<StatementNode> thenBranch = new ArrayList<>();
    for (SimpleLangParser.StatementContext stmtCtx : ctx.statement().subList(0, 1)) {
        thenBranch.addFirst((StatementNode) this.stack.pop());
    }

    ComparisonNode condition = (ComparisonNode) this.stack.pop();

    this.stack.push(new IfNode(condition, thenBranch, elseBranch));
}

@Override
public void exitExpression(SimpleLangParser.ExpressionContext ctx) {
    if (ctx.children.size() == 1) {
        if (ctx.ID() != null) {
            this.stack.push(new IdentifierNode(ctx.ID().getText()));
        } else if (ctx.NUMBER() != null) {
            this.stack.push(new NumberNode(Integer.parseInt(ctx.NUMBER().getText())));
        }
    } else if (ctx.children.size() == 3) {
        ExpressionNode right = (ExpressionNode) this.stack.pop();
        ExpressionNode left = (ExpressionNode) this.stack.pop();
        String operator = ctx.getChild(1).getText();
        this.stack.push(new BinaryOperationNode(left, operator, right));
    }
}

@Override
public void exitComparison(SimpleLangParser.ComparisonContext ctx) {
    ExpressionNode right = (ExpressionNode) this.stack.pop();
    ExpressionNode left = (ExpressionNode) this.stack.pop();
```

```
        String operator = ctx.getChild(1).getText();
        this.stack.push(new ComparisonNode(left, operator, right));
    }

    @Override
    public void exitProgram(SimpleLangParser.ProgramContext ctx) {
        List<StatementNode> statements = new ArrayList<>();
        int statementCount = ctx.statement().size();
        for (int i = 0; i < statementCount; i++) {
            statements.addFirst((StatementNode) this.stack.pop());
        }
        this.stack.push(new ProgramNode(statements));
    }
}
```

Die einzelnen Node Klassen des AST haben wir in der SimpleLangBuilder.java Datei erstellt. Diese Klassen bauen sich die Expressions und Statements rekursiv auf und speichern die Werte in den entsprechenden Klassen. Mit der toString Methode kann der AST ausgegeben werden.

```
abstract class ASTNode {}

class ProgramNode extends ASTNode {
    List<StatementNode> statements;

    ProgramNode(List<StatementNode> statements) {
        this.statements = statements;
    }

    public String toString() {
        return statements.toString();
    }
}

abstract class StatementNode extends ASTNode {}

class DeclarationNode extends StatementNode {
    String id;
    ExpressionNode expression;

    DeclarationNode(String id, ExpressionNode expression) {
        this.id = id;
        this.expression = expression;
    }

    public String toString() {
        return id + " = " + expression;
```

```
    }
}

class PrintNode extends StatementNode {
    ExpressionNode expression;

    PrintNode(ExpressionNode expression) {
        this.expression = expression;
    }

    public String toString() {
        return String.format("print(%s)", this.expression);
    }
}

class IfNode extends StatementNode {
    ExpressionNode condition;
    List<StatementNode> thenBranch;
    List<StatementNode> elseBranch;

    IfNode(ExpressionNode condition, List<StatementNode> thenBranch, List<StatementNode> els
        this.condition = condition;
        this.thenBranch = thenBranch;
        this.elseBranch = elseBranch;
    }

    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("if (").append(this.condition).append(") {\n");
        for (StatementNode stmt : this.thenBranch) {
            sb.append("  ").append(stmt).append("\n");
        }
        sb.append("}");
        if (!this.elseBranch.isEmpty()) {
            sb.append(" else {\n");
            for (StatementNode stmt : this.elseBranch) {
                sb.append("  ").append(stmt).append("\n");
            }
            sb.append("}");
        }
        return sb.toString();
    }
}

abstract class ExpressionNode extends ASTNode {}
```

```
class IdentifierNode extends ExpressionNode {
    String name;

    IdentifierNode(String name) {
        this.name = name;
    }

    public String toString() {
        return this.name;
    }
}

class ComparisonNode extends ExpressionNode {
    private final ExpressionNode left;
    private final String operator;
    private final ExpressionNode right;

    public ComparisonNode(ExpressionNode left, String operator, ExpressionNode right) {
        this.left = left;
        this.operator = operator;
        this.right = right;
    }

    public String toString() {
        return String.format("(%s %s %s)", this.left, this.operator, this.right);
    }
}

class NumberNode extends ExpressionNode {
    int value;

    NumberNode(int value) {
        this.value = value;
    }

    public String toString() {
        return Integer.toString(this.value);
    }
}

class BinaryOperationNode extends ExpressionNode {
    ExpressionNode left;
    String operator;
    ExpressionNode right;

    BinaryOperationNode(ExpressionNode left, String operator, ExpressionNode right) {
```

```
        this.left = left;
        this.operator = operator;
        this.right = right;
    }

    public String toString() {
        return String.format("(%s %s %s)", this.left, this.operator, this.right);
    }
}
```

Um den AST zu erstellen wurde eine Java-Klasse `SimpleLangToAST.java` erstellt, welche ähnlich wie in Abgabe 1 eine Textdatei einliest, die Token generiert und den Parse Tree erstellt und diesen mithilfe von `SimpleLangBuilder` in einen AST umwandelt.

```
// SimpleLangToAST.java
import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.CommonTokenStream;
import org.antlr.v4.runtime.tree.ParseTree;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class SimpleLangToAst {
    private SimpleLangToAst() {
    }

    public static void main(String[] args) throws IOException {
        String inputFile = "Abgabe2/Test3.txt";
        String input = new String(Files.readAllBytes(Paths.get(inputFile)));

        SimpleLangLexer lexer = new SimpleLangLexer(CharStreams.fromString(input));
        SimpleLangParser parser = new SimpleLangParser(new CommonTokenStream(lexer));
        ParseTree tree = parser.program();

        if (parser.getNumberOfSyntaxErrors() > 0) {
            System.out.printf("Found %d syntax errors.%n", parser.getNumberOfSyntaxErrors()
            System.exit(1);
        }

        ASTNode ast = new SimpleLangBuilder().build(tree);
        System.out.printf("AST: %n---%n%s%n---%n", ast.toString().substring(0, ast.toString(
    }
}
```

Folgende Ausgaben werden durch SimpleLangToAST für die Testfiles generiert:

```
// Test1.txt
AST:
---
Program([
Declaration('x', Number(5), Print(Identifier(x))
])
---

// Test2.txt
AST:
---
Program([
Declaration('y', Number(10),
If(
    Comparison('Identifier(y)', '>', 'Number(5)'),
    Then(
        Print(Identifier(y))
    ),
    Else(
        Print(Number(0))
    )
)
])
---

// Test3.txt
AST:
---
Program([
Declaration('a', Number(3),
Declaration('b', BinaryOperation('Identifier(a)', '+', 'Number(2)'), Print(Identifier(b))
])
---
```

Welche Terminale und Nichtterminale aus dem Ableitungsbaum werden in Ihrem AST weggelassen? Folgende Terminale werden in unserem AST weggelassen:

1. Klammern: (,), {, }
2. Trennzeichen: ;
3. Whitespace: Leerzeichen, Tabs und Zeilenumbrüche

Folgenden Nichtterminale werden in unserem AST weggelassen:

1. Primary-Knoten: Diese Knoten, die nur ein Kind haben, werden weggelassen, um die Struktur zu vereinfachen.
2. Expr- und MultExpr-Knoten: Wenn diese Knoten nur ein Kind haben, werden sie weggelassen.

3. Einige spezifische Nichtterminale: Wie statement, expression, comparison, die nur als Container für andere Knoten dienen und keine eigene Bedeutung im AST haben.

Abgabe 3

Aufgabe

Im ersten Teil der dritten Abgabe sollten zunächst folgende Fragen zu der zuvor definierten Sprache beantwortet werden:

- Lässt sich eine statische Semantik für Ihre abstrakte Syntax angeben?
- Erlaubt Ihre konkrete Syntax Formulierungen, die die statische Semantik verletzen?

Daraufhin sollte eine statische Semantikprüfung für die Sprache ergänzt werden.

Im zweiten Teil der Abgabe sollte für die eigene Sprache mindestens eine dynamische Semantik programmiert werden.

Vorgehensweise

a) Statische Semantik Lässt sich eine statische Semantik für Ihre abstrakte Syntax angeben? Ja, eine statische Semantik kann für die abstrakte Syntax angegeben werden:

- Variablendeklarationen: Eine Variable muss deklariert werden, bevor sie in einer Expression verwendet wird.
- Mehrfachdeklaration: Eine Variable darf nicht mehrmals im selben Gültigkeitsbereich deklariert werden.

Erlaubt Ihre konkrete Syntax Formulierungen, die die statische Semantik verletzen? Ja, die konkrete Syntax erlaubt Formulierungen, die die statische Semantik verletzen könnten:

- Verwendung nicht deklarierter Variablen: Die Grammatik erlaubt es, dass eine Variable in einem Ausdruck verwendet wird, ohne dass sie vorher deklariert wurde.
- Mehrfachdeklaration: Die Grammatik erlaubt es, dass eine Variable mehrmals im selben Gültigkeitsbereich deklariert wird.

Statische Semantikprüfung Um die statische Semantikprüfung zu implementieren, haben wir unsere Klasse `SimpleLangBuilder` erweitert. Zuerst haben wir eine Map `symbolTable` erstellt, um die Variablendeklarationen zu speichern. Außerdem wurde eine Methode zur Ausgabe eines Semantikfehlers hinzugefügt. Diese Methode wird aufgerufen, wenn ein Fehler in der statischen Semantik gefunden wird und gibt die Zeile und Position des Fehlers sowie die Ursache aus.

Anschließend haben wir die Methoden `exitDeclaration` sowie `exitExpression` erweitert, um die Variablendeklarationen zu überprüfen. Bei 'exitDeclaration' wird überprüft, ob die Variable bereits deklariert wurde, um sicherzustellen, dass keine Variable doppelt deklariert wird. Bei 'exitExpression' wird überprüft, ob die Variable vorher deklariert wurde, damit sie ohne Deklaration aufgerufen werden kann.

```
private Map<String, String> symbolTable = new HashMap<>();

private void semanticErr(Token token, String message) {
    System.err.printf("Line %d:%d - %s%n", token.getLine(), token.getCharPositionInLine(), m
}

@Override
public void exitDeclaration(SimpleLangParser.DeclarationContext ctx) {
    String id = ctx.ID().getText();

    // Check if the identifier is already declared
    if (this.symbolTable.containsKey(id)) {
        this.semanticErr(ctx.ID().getSymbol(), String.format("Identifier '%s' is already dec
    } else {
        this.symbolTable.put(id, "int");
    }

    ExpressionNode expr = (ExpressionNode) this.stack.pop();
    this.stack.push(new DeclarationNode(id, expr));
}

@Override
public void exitExpression(SimpleLangParser.ExpressionContext ctx) {
    if (ctx.children.size() == 1) {
        if (ctx.ID() != null) {
            // Check if the identifier is declared
            String id = ctx.ID().getText();
            if (!this.symbolTable.containsKey(id)) {
                this.semanticErr(ctx.ID().getSymbol(), String.format("Identifier '%s' is not
            }

            this.stack.push(new IdentifierNode(ctx.ID().getText()));
        } else if (ctx.NUMBER() != null) {
            this.stack.push(new NumberNode(Integer.parseInt(ctx.NUMBER().getText())));
        }
    } else if (ctx.children.size() == 3) {
        ExpressionNode right = (ExpressionNode) this.stack.pop();
        ExpressionNode left = (ExpressionNode) this.stack.pop();
        String operator = ctx.getChild(1).getText();
        this.stack.push(new BinaryOperationNode(left, operator, right));
    }
}
```

Um zu Testen, ob die statische Semantikprüfung funktioniert, haben wir folgenden Testfall erstellt:

```
// Test 4.txt
```

```
let a = 3;
let b = 6;
print(b);

let a = 4;
print(c);
```

Die Ausgabe für Test 4.txt sieht wie folgt aus:

```
Line 5:4 - Identifier 'a' is already declared.
Line 6:6 - Identifier 'c' is not declared.
AST:
---
Program([
Declaration('a', Number(3),
Declaration('b', Number(6), Print(Identifier(b)),
Declaration('a', Number(4), Print(Identifier(c))
])
---
```

b) Dynamische Semantik Um eine dynamische Semantikprüfung einzufügen haben wir unsere Sprache etwas erweitern müssen. Dazu haben wir in unserer 'SimpleLangLexer.g4' Datei die Möglichkeit hinzugefügt, einer Variablen nicht nur Numbers, sondern auch Strings zuzuordnen. In unserem 'SimpleLangParser.g4' haben wir die Möglichkeit hinzugefügt, Variablen während der Laufzeit zu verändern.

Die neuen Lexer und Parser Dateien sehen nun wie folgt aus:

```
// SimpleLangLexer.g4
lexer grammar SimpleLangLexer;

// Schlüsselwörter
LET: 'let';
IF: 'if';
ELSE: 'else';
PRINT: 'print';

// Bezeichner
ID: [a-zA-Z_] [a-zA-Z_0-9]*;

// Zahlen
NUMBER: [0-9]+;

STRING: '"' (~["\r\n])* '"';

// Operatoren
```

```
PLUS: '+';
MINUS: '-';
STAR: '*';
SLASH: '/';
EQ: '==';
NEQ: '!=';
LT: '<';
GT: '>';
LE: '<=';
GE: '>=';

// Trennzeichen
SEMI: ';';
ASSIGN: '=';
LPAREN: '(';
RPAREN: ')';
LBRACE: '{';
RBRACE: '}';

// Whitespace
WS: [ \t\r\n]+ -> skip;
```

```
// SimpleLangParser.g4
parser grammar SimpleLangParser;

options { tokenVocab = SimpleLangLexer; }

program: statement+;

statement:
    declaration
    | assignment
    | printStatement
    | ifStatement
    ;

declaration: LET ID ASSIGN expression SEMI;

assignment: ID ASSIGN expression SEMI;

printStatement: PRINT LPAREN expression RPAREN SEMI;

ifStatement: IF LPAREN comparison RPAREN LBRACE statement+ RBRACE (ELSE LBRACE statement+ RB

expression:
    ID
```

```
    | NUMBER
    | STRING
    | expression PLUS expression
    | expression MINUS expression
    | expression STAR expression
    | expression SLASH expression
    ;

comparison:
    expression (EQ | NEQ | LT | GT | LE | GE) expression
    ;
```

Außerdem haben wir die statische Semantik zur Eintragung von Strings in die Symboltabelle erweitert und erneute Assignments zugelassen, um Variablen während der Laufzeit zu verändern:

```
@Override
    public void exitDeclaration(SimpleLangParser.DeclarationContext ctx) {
        String id = ctx.ID().getText();

        // Check if the identifier is already declared
        if (this.symbolTable.containsKey(id)) {
            this.semanticErr(ctx.ID().getSymbol(), String.format("Identifier '%s' is already
        } else if (ctx.expression().NUMBER() != null) {
            this.symbolTable.put(id, "number");
        } else if (ctx.expression().STRING() != null) {
            this.symbolTable.put(id, "string");
        }

        ExpressionNode expr = (ExpressionNode) this.stack.pop();
        this.stack.push(new DeclarationNode(id, expr));
    }

    @Override
    public void exitAssignment(SimpleLangParser.AssignmentContext ctx) {
        String id = ctx.ID().getText();

        // Check if the identifier is declared
        if (!this.symbolTable.containsKey(id)) {
            this.semanticErr(ctx.ID().getSymbol(), String.format("Identifier '%s' is not dec
        }

        if (this.symbolTable.get(id).equals("number") && ctx.expression().STRING() != null)
            symbolTable.put(id, "string");
        } else if (this.symbolTable.get(id).equals("string") && ctx.expression().NUMBER() !=
            symbolTable.put(id, "number");
        }
```

```
        ExpressionNode expr = (ExpressionNode) this.stack.pop();
        this.stack.push(new AssignmentNode(id, expr));
    }
```

Um die dynamische Semantikprüfung zu testen, haben wir folgenden Testfall erstellt:

```
// Test 5.txt
let a = 4;
let b = 5;

a = "test";

if (a == b) {
    print(a);
} else {
    print(b);
}
```

Was den folgenden Output generiert:

```
Line 6:4 - Type mismatch: cannot compare string with number
AST:
---
Program([
Declaration('a', Number(4),
Declaration('b', Number(5), Assignment('a', String("test")),
If(
    Comparison('Identifier(a)', '==', 'Identifier(b)'),
    Then(
        Print(Identifier(a))
    ),
    Else(
        Print(Identifier(b))
    )
)
])
---
```

Abgabe 4

Aufgabe

In Aufgabenteil a) sollte ein gegebenes Java Programm um die Klassenmethoden readLines, removeEmptyLines, removeShortLines und totalLineLengths mit Verwendung von Schleifen und Verzweigungen erweitert werden. Um Teil b) zu lösen, sollte das in a) geschriebene Programm mithilfe von java.util.streams und Lambdas auf einen funktionalen Stil umgestellt werden, wobei das Programm danach keine Schleifen, Verzweigungen und Seiteneffekte mehr aufweisen darf. Anschließend sollten die beiden Programme in Teil c) verglichen werden.

Teil a)

Die einzelnen Methoden wurden in der Klasse `Procedural` implementiert. Die Methoden `readLines`, `removeEmptyLines`, `removeShortLines` und `totalLineLengths` wurden mit Schleifen und Verzweigungen implementiert.

```java
// Procedural.java
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;

import java.io.BufferedReader;
import java.util.LinkedList;

public final class Procedural {
    private Procedural() { }

    private static final int MIN_LENGTH = 20;

    public static void main(String[] args) throws IOException {
        var input = Paths.get(args[0]);
        var lines = new LinkedList<String>();

        long start = System.nanoTime();

        readLines(Files.newBufferedReader(input), lines);
        removeEmptyLines(lines);
        removeShortLines(lines);
        int n = totalLineLengths(lines);

        long stop = System.nanoTime();

        System.out.printf("result = %d (%d microsec)%n", n, (stop - start) / 1000);
    }
```

```
    private static void readLines(BufferedReader reader, LinkedList<String> lines) throws IO
        String line;
        while ((line = reader.readLine()) != null) {
            lines.add(line);
        }
    }

    private static void removeEmptyLines(LinkedList<String> lines) {
        for (int i = 0; i < lines.size(); i++) {
            if (lines.get(i).trim().isEmpty()) {
                lines.remove(i);
                i--;
            }
        }
    }

    private static void removeShortLines(LinkedList<String> lines) {
        for (int i = 0; i < lines.size(); i++) {
            if (lines.get(i).length() < MIN_LENGTH) {
                lines.remove(i);
                i--;
            }
        }
    }

    private static int totalLineLengths(LinkedList<String> lines) {
        int totalLength = 0;
        for (String line : lines) {
            totalLength += line.length();
        }
        return totalLength;
    }
}
```

Teil b)

Das Programm wurde in der Klasse `Functional` umgeschrieben, um die Methoden mithilfe von Streams und Lambdas in einen funktionalen Stil umzustellen. Dabei wurden die Methoden `readLines`, `removeEmptyLines`, `removeShortLines` und `totalLineLengths` umgeschrieben.

```
// Functional.java
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;
```

```java
import java.io.BufferedReader;
import java.util.LinkedList;
import java.util.stream.Collectors;

public final class Functional {
    private Functional() { }

    private static final int MIN_LENGTH = 20;

    public static void main(String[] args) throws IOException {
        var input = Paths.get(args[0]);
        var lines = new LinkedList<String>();

        long start = System.nanoTime();

        readLines(Files.newBufferedReader(input), lines);
        removeEmptyLines(lines);
        removeShortLines(lines);
        int n = totalLineLengths(lines);

        long stop = System.nanoTime();

        System.out.printf("result = %d (%d microsec)%n", n, (stop - start) / 1000);
    }

    private static void readLines(BufferedReader reader, LinkedList<String> lines) {
        lines.addAll(reader.lines().toList());
    }

    private static void removeEmptyLines(LinkedList<String> lines) {
        var filtered = lines.stream()
                .filter(line -> !line.trim().isEmpty())
                .collect(Collectors.toCollection(LinkedList::new));

        lines.clear();
        lines.addAll(filtered);
    }

    private static void removeShortLines(LinkedList<String> lines) {
        var filtered = lines.stream()
                .filter(line -> line.length() >= MIN_LENGTH)
                .collect(Collectors.toCollection(LinkedList::new));

        lines.clear();
        lines.addAll(filtered);
    }
```

```
    private static int totalLineLengths(LinkedList<String> lines) {
        return lines.stream()
                .mapToInt(String::length)
                .sum();
    }
}
```

Teil c) Vergleich

Für den Vergleich haben wir 2 Testdateien erstellt, eine kurz und eine lang, um die Unterschiede in der Laufzeit zu sehen. Die längere Testdatei hat denselben Inhalt, allerdings mehrfach kopiert, um insgesamt 28013 Zeilen zu haben. Die kurze Testdatei hat 6 Zeilen.

Inhalt der Testdateien:

```
Dies ist eine Testzeile mit mehr als zwanzig Zeichen.
Kurz.

Zeile mit genau zwanzig Zeichen.
12345678901234567890
Noch eine lange Zeile mit mehr als zwanzig Zeichen.
```

Für beide Testdateien haben wir die Laufzeit für die prozedurale und funktionale Implementierung gemessen. Das Programm wurde jeweils 5 Mal ausgeführt und die Laufzeit gemittelt.

Ergebnisse für die kurze Testdatei:

Durchlauf	Prozedural (microsec)	Funktional (microsec)
1	10472	15080
2	7917	11350
3	7960	13729
4	11545	12535
5	8966	17404

Ergebnisse für die lange Testdatei:

Durchlauf	Prozedural (microsec)	Funktional (microsec)
1	988918	36396
2	1012826	36025
3	961305	35151
4	948288	40268
5	928869	37637

Die Ergebnisse im Durchschnitt (gerundet) übersichtlich dargestellt:

Testdatei	Total line lengths	Prozedural (microsec)	Funktional (microsec)
Kurz	156	10072	14020
Lang	624312	949041	37095

Bei den Ergebnissen fällt auf, dass die Laufzeit des funktionalen Programms bei der langen Testdatei deutlich kürzer ist als die des prozeduralen Programms. Bei der kurzen Testdatei ist die Laufzeit des funktionalen Programms allerdings etwas länger. Dies liegt daran, dass die Verarbeitung von Streams und Lambdas bei kleinen Datenmengen langsamer sein kann als die Verarbeitung mit Schleifen und Verzweigungen. Bei großen Datenmengen hingegen ist die Verarbeitung mit Streams und Lambdas schneller, da sie parallelisiert werden können.

Abgabe 5

5.a.1)

Liste 1	Liste 2	Instanziierung
[X,Y,Z]	[john,likes,fish]	X = john, Y = likes, Z = fish
[cat]	[X|Y]	X = cat, Y = []
[X,Y|Z]	[mary,likes,wine]	X = mary, Y = likes, Z = [wine]
[[the,Y]|Z]	[[X,hare],[is,here]]	X = the, Y = hare, Z = [[is,here]]
[golden|T]	[golden,norfolk]	T = [norfolk]
[white,horse]	[horse,X]	Listen stimmen nicht überein
[white|Q]	[P,horse]	P = white, Q = [horse]

5.a.2)

```
fakultaet(0, 1).

fakultaet(N, F) :-
    N > 0,
    N1 is N - 1,
    fakultaet(N1, F1),
    F is N * F1.
```

Hierbei gibt es zwei Fälle:

1. Der Basisfall, wenn N = 0 ist, dann ist die Fakultät 1.
2. Der rekursive Fall, wenn N > 0 ist, dann wird die Fakultät von N berechnet, indem die Fakultät von N-1 berechnet wird und mit N multipliziert wird.

5.a.3)

`append(X, Y, [1,2,3,4]).` sucht nach zwei Listen X und Y, die zusammengefügt die Liste `[1,2,3,4]` ergeben.

`append(X, [1,2,3,4], Y).` sucht nach einer Liste X, die mit der Liste `[1,2,3,4]` zusammengefügt die Liste Y ergibt.

5 b)

```
sum([], 0).

sum([H|T], Sum) :-
    sum(T, RestSum),
    Sum is H + RestSum.
```

Hierbei gibt es zwei Fälle:

1. Der Basisfall, wenn die Liste leer ist, dann ist die Summe 0.

2. Der rekursive Fall, wenn die Liste nicht leer ist, dann wird die Summe der Liste berechnet, indem das erste Element der Liste mit der Summe des Rests der Liste addiert wird.

5 c)

```
% Fakten
zug(konstanz, 08.39, offenburg, 10.59).
zug(konstanz, 08.39, karlsruhe, 11.49).
zug(konstanz, 09.06, singen, 09.31).
zug(singen, 09.36, stuttgart, 11.32).
zug(offenburg, 11.28, mannheim, 12.24).
zug(karlsruhe, 12.06, mainz, 13.47).
zug(stuttgart, 11.51, mannheim, 12.28).
zug(mannheim, 12.39, mainz, 13.18).

% Basisfall: Direkte Verbindung
verbindung(Start, AbfahrtszeitMin, Ziel, [zug(Start, Abfahrtszeit, Ziel, Ankunftszeit)]) :-
    zug(Start, Abfahrtszeit, Ziel, Ankunftszeit),
    Abfahrtszeit >= AbfahrtszeitMin.

% Rekursiver Fall: Verbindung mit Umsteigen
verbindung(Start, AbfahrtszeitMin, Ziel, [zug(Start, Abfahrtszeit, Zwischenhalt, Ankunftszei
    zug(Start, Abfahrtszeit, Zwischenhalt, Ankunftszeit),
    Abfahrtszeit >= AbfahrtszeitMin,
    verbindung(Zwischenhalt, Ankunftszeit, Ziel, Rest).
```

In diesem Prolog-Programm gibt es zwei Fälle:

1. Der Basisfall, wenn es eine direkte Verbindung zwischen Start- und Zielort gibt, dann wird ein Zug hinzugefügt.
2. Der rekursive Fall, wenn es keine direkte Verbindung gibt, dann wird ein Zug zum Zwischenhalt hinzugefügt und die Verbindung zum Zielort wird rekursiv gesucht.

Ergebnis für `verbindung(konstanz, 8.00, mainz, Reiseplan).`:

```
?- verbindung(konstanz, 8.00, mainz, Reiseplan).
Reiseplan = [zug(konstanz, 8.39, offenburg, 10.59), zug(offenburg, 11.28, mannheim, 12.24),
Reiseplan = [zug(konstanz, 8.39, karlsruhe, 11.49), zug(karlsruhe, 12.06, mainz, 13.47)] ;
Reiseplan = [zug(konstanz, 9.06, singen, 9.31), zug(singen, 9.36, stuttgart, 11.32), zug(stu
false.
```

Abgabe 6

Aufgabe

Es sollte eine Java-Klasse implementiert werden, die für beliebige Java-Klassen und -Interfaces eine HTML-Seite im Format der Beispieldatei aufgabe6.html generiert.

Vorgehensweise

Zuerst haben wir eine Klasse `Aufgabe6.java` erstellt:

```java
import org.stringtemplate.v4.ST;
import org.stringtemplate.v4.STGroup;
import org.stringtemplate.v4.STGroupFile;

import java.util.*;

public class Aufgabe6 {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Bitte geben Sie mindestens eine Klasse oder ein Interface an
            return;
        }

        List<Class<?>> classList = new ArrayList<>();

        try {
            for (String className : args) {
                classList.add(Class.forName(className));
            }

            STGroup stGroup = new STGroupFile("Aufgabe6/aufgabe6.stg");
            ST template = stGroup.getInstanceOf("root");

            template.add("classes", classList);

            System.out.println(template.render());
        } catch (ClassNotFoundException e) {
            System.err.println("Klasse nicht gefunden: " + e.getMessage());
        }
    }
}
```

Die Klasse `Aufgabe6` liest die Klassennamen aus den Argumenten und erstellt eine Liste von `Class`-Objekten. Anschließend wird ein `STGroup`-Objekt erstellt, um die Templates aus der Datei `aufgabe6.stg` zu laden. Das `root`-Template wird geladen und die Liste der Klassen hinzugefügt. Das gerenderte Template

wird dann ausgegeben.

Die Datei `aufgabe6.stg` enthält die Templates für die HTML-Generierung:

```
group aufgabe6;

delimiters "$", "$"

root(classes) ::= <<
<!DOCTYPE html>
<html lang="de">
<head>
<style type="text/css">
th, td { border-bottom: thin solid; padding: 4px; text-align: left; }
td { font-family: monospace }
</style>
</head>
<body>
<h1>Sprachkonzepte, Aufgabe 6</h1>
$classes:{class |
<h2>$if(class.interface)$interface$else$class$endif$ $class.name$:</h2>
<table>
$if(class.interface)$
<tr><th>Methods</th></tr>
<tr><td>$class.methods:{method | $method.returnType.name$ $method.name$($method.parameters:
$else$
<tr><th>Interface</th><th>Methods</th></tr>
$class.interfaces:{interface |
<tr>
<td valign="top">$interface.name$</td>
<td>$interface.methods:{method | $method.returnType.name$ $method.name$($method.parameters:
}$
$endif$
</table>
<br>
}$
</body>
</html>
>>
```

Die Templates enthalten eine Schleife über die Liste der Klassen, um für jede Klasse ein HTML-Element zu generieren. Für jede Klasse wird der Name und die Methoden aufgelistet. Falls es sich um eine Klasse mit Interfaces handelt, wird zusätzlich eine Liste der Interfaces angezeigt.

Aufgerufen wird das Programm mit den Klassennamen als Argumente:

```
java -cp ../antlr-4.9.2-complete.jar Aufgabe6.java java.lang.String java.util.Iterator java.
```

Das Programm generiert dann die HTML-Seite und gibt sie auf der Konsole aus. Das Ergebnis ist nahezu identisch mit der Beispieldatei `aufgabe6.html` (nur die Reihenfolge der Methoden kann leicht abweichen).:

Sprachkonzepte, Aufgabe 6

class java.lang.String:

Interface	Methods
java.io.Serializable	
java.lang.Comparable	int compareTo(java.lang.Object)
java.lang.CharSequence	int length() java.lang.String toString() int compare(java.lang.CharSequence, java.lang.CharSequence) char charAt(int) boolean isEmpty() java.util.stream.IntStream codePoints() java.lang.CharSequence subSequence(int, int) java.util.stream.IntStream chars()
java.lang.constant.Constable	java.util.Optional describeConstable()
java.lang.constant.ConstantDesc	java.lang.Object resolveConstantDesc(java.lang.invoke.MethodHandles$Lookup)

interface java.util.Iterator:

Methods
void remove() void forEachRemaining(java.util.function.Consumer) boolean hasNext() java.lang.Object next()

class java.time.Month:

Interface	Methods
java.time.temporal.TemporalAccessor	int get(java.time.temporal.TemporalField) long getLong(java.time.temporal.TemporalField) boolean isSupported(java.time.temporal.TemporalField) java.lang.Object query(java.time.temporal.TemporalQuery) java.time.temporal.ValueRange range(java.time.temporal.TemporalField)
java.time.temporal.TemporalAdjuster	java.time.temporal.Temporal adjustInto(java.time.temporal.Temporal)

Figure 1: Ergebnis

Probleme

- Beim Auslesen der `aufgabe6.stg` Datei trat das Problem auf, dass durch das `<!DOCTYPE html>` ein Kommentar ausgelöst aber nie beendet wurde. Die konnte durch Setzen anderer delimiter gelöst werden: `delimiters "$", "$"`.

Abgabe 7

Aufgabe

Die Aufgabe bestand darin, ein Programm in einer Skriptsprache zu schreiben und zu analysieren, welche typischen Eigenschaften einer Skriptsprache darin ausgenutzt werden.

Vorgehensweise

Für die Aufgabe haben wir uns für Python entschieden und ein simples Programm geschrieben, welches die vorgeschlagene feiertage-api aufruft:

```python
import requests

def get_holidays(year, state):
    """
    Ruft Feiertage für ein bestimmtes Jahr und Bundesland von der API ab.
    """
    url = f"https://feiertage-api.de/api/?jahr={year}&nur_land={state}"
    try:
        response = requests.get(url)
        response.raise_for_status()  # Fehler bei HTTP-Statuscodes erkennen
        holidays = response.json()
        return holidays
    except requests.exceptions.RequestException as e:
        print(f"Fehler bei der Anfrage: {e}")
        return None

def main():
    """
    Hauptfunktion zur Eingabe und Ausgabe der Feiertage.
    """
    year = input("Geben Sie das Jahr ein (z. B. 2025): ")
    state = input("Geben Sie das Bundesland ein (z. B. BW für Baden-Württemberg): ")

    holidays = get_holidays(year, state)
    if holidays:
        print(f"\nFeiertage in {state} im Jahr {year}:\n")
        for name, details in holidays.items():
            print(f"{name}: {details['datum']}")
    else:
        print("Keine Feiertage gefunden oder ein Fehler ist aufgetreten.")

if __name__ == "__main__":
    main()
```

Dazu muss man das Jahr sowie das Bundesland eingeben, für welches man die

Feiertage ausgegeben haben will.

Hierbei werden folgende typische Eigenschaften für eine Skriptsprache ausgenutzt:

- **Dynamische Typisierung**: Variablen wie year, state und holidays müssen nicht explizit deklariert werden. Der Datentyp wird zur Laufzeit bestimmt.
- **Standardbibliotheken**: Mit Bibliotheken wie requests können HTTP-Anfragen einfach durchgeführt werden.
- **Interpretation**: Python-Code wird zur Laufzeit interpretiert und nicht kompiliert.
- **String-Interpolation**: Mit f-Strings können Variablen direkt in Strings eingebettet werden.
- **Einfache Syntax**: Python hat eine einfache und leicht verständliche Syntax, die das Schreiben von Code erleichtert und wodurch das Entwickeln von Prototypen vereinfacht wird.
- **Plattformunabhängigkeit**: Python-Code kann auf verschiedenen Plattformen ausgeführt werden, ohne dass Änderungen am Code erforderlich sind.