

Sprachkonzepte Übung WS24/25

Stefan Ptacek und Patrick Zedler

Bericht über die Bearbeitung der Aufgaben.

Aufgabe 1

Aufgabenstellung

Sie sollen das Vokabular eines Textes mit ANTLR4 Lexer-Regeln beschreiben und eine Anwendung erstellen, die einen entsprechenden Text einliest und als Tokenfolge wieder ausgibt (siehe ExprTokenizer.java aus der Vorlesung).

Zwei Texte stehen zur Auswahl (natürlich dürfen Sie auch beide bearbeiten): Informationen aus der Abfahrtstafel Konstanz der Deutschen Bahn in abfahrten-kn.txt oder Angaben zu den Öffnungszeiten der Mainau-Gastronomie in mainau-gastronomie.txt. Die beiden Texte finden Sie auf der Moodle-Seite.

Den Text abfahrten-kn.txt habe ich aus der Webseite <https://www.bahn.de/buchung/abfahrten-ankuenfte> für den Bahnhof Konstanz extrahiert. Sie können gerne weitere Abfahrten ergänzen.

Den Text mainau-gastronomie.txt habe ich aus der Webseite <https://www.mainau.de/de/oeffnungszeiten> extrahiert. Sie können sich gerne weitere gastronomische Angebote ausdenken.

Welche Vokabular-Kategorien von Folie 2-4 kommen im Text vor? Bedenken Sie dazu, dass ja nicht nur der eine vorgegebene Text erfolgreich zerlegt werden soll, sondern auch andere Beispieltex te der gleichen Art.

Lösung Code

TokenPrinter.java:

```
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.CommonTokenStream;

import java.io.IOException;

public class TokenPrinter {
    public static void main(String[] args) {
        try {
            // Lese den Text von der Konsole oder aus einer Datei
            CharStream input = CharStreams.fromFileName("mainau-gastronomie.txt");

            // Erstelle eine Instanz des Lexers mit dem Eingabetext
            OpeningTimesLexer lexer = new OpeningTimesLexer(input);

            // Erstelle einen TokenStream, um die Token zu verarbeiten
            TokenStream tokens = new CommonTokenStream(lexer);
```

```

        // Schleife durch die Token und gib sie aus
        Token token;
        while ((token = lexer.nextToken()).getType() != Token.EOF) {
            String tokenType =
lexer.getVocabulary().getSymbolicName(token.getType());
            String tokenText = token.getText();
            System.out.println("Token Type: " + tokenType + ", Token Text: " +
tokenText);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

OpeningTimesLexer.g4:

```

lexer grammar OpeningTimesLexer;

WHITESPACE : [ \t\r\n]+ -> skip ;

KW_BIS : 'bis' ;
KW_TAEGlich : 'täglich' ;
KW_RUHETAG : 'Ruhetag' ;
KW_AN : 'an' ;
KW_FREITAG : 'Freitag' ;
KW_MONTAG : 'Montag' ;
KW_DIENSTAG : 'Dienstag' ;
KW_MITTWOCH : 'Mittwoch' ;
KW_DONNERSTAG : 'Donnerstag' ;
KW_SAMSTAG : 'Samstag' ;
KW_SONNTAG : 'Sonntag' ;
KW_VORUEBERGEHEND : 'vorübergehend' ;
KW_GESCHLOSSEN : 'geschlossen' ;
KW_BEI : 'bei' ;
KW_GUTEM_WETTER : 'gutem Wetter' ;
KW_UHR : 'Uhr' ;

NUMBER : [0-9]+ ;
DATE : NUMBER '.' [A-Z\u00E4\u00F6\u00FC\u00C4\u00D6\u00DC\u00DF][a-
z\u00E4\u00F6\u00FC\u00C4\u00D6\u00DC\u00DF]+ ;
TIME : NUMBER '.' NUMBER ' Uhr'? ;
ID : [a-zA-Z\u00E4\u00F6\u00FC\u00C4\u00D6\u00DC\u00DF\u00E9]+ ;
DOT : '.' ;
COMMA : ',' ;
MINUS : '-' ;
COLON : ':' ;

```

Erklärung

In dieser Aufgabe haben wir einen Lexer für die mainau-gastronomie mithilfe von ANTLR4 erstellt, um die Öffnungszeiten von Restaurants aus einem Text in einzelne Token zu zerlegen. Zuerst haben wir eine Grammatikdatei (.g4) geschrieben, die festlegt, wie verschiedene Textbestandteile wie Schlüsselwörter, Datumsangaben, Uhrzeiten und Trennzeichen erkannt werden. Die Grammatik beschreibt die Regeln, um Bezeichner, Literale und Operatoren zu unterscheiden. Anschließend haben wir den Lexer in Java generiert und eine Anwendung erstellt, die den Lexer verwendet, um den Text zu verarbeiten. Der Lexer liest den Text ein und zerlegt ihn in Token, die dann in einer bestimmten Reihenfolge ausgegeben werden. Jeder Teil des Textes, wie Wochentage, Zeiträume und Sonderangaben, wird dabei als spezifischer Token klassifiziert. Zum Beispiel werden Datumsangaben als Zahlenliterals erkannt, während Wörter wie „täglich“ oder „Ruhetag“ als Schlüsselwörter eingeordnet werden. Die Ausgabe der Anwendung zeigt die Tokenfolge, die den Text strukturiert darstellt. Damit haben wir erreicht, dass auch ähnliche Texte automatisch analysiert werden können. Diese Methode hilft, die Struktur und Bedeutung von Zeitplänen aus Texten zu extrahieren.

Probleme hatten wir dabei, Sonderzeichen richtig einzulesen, da diese trotz der richtigen Formatierung aller Dateien nicht korrekt eingelesen wurden und somit als Error dargestellt wurden. Die Sonderzeichen wurden bei dem Kompilieren der Grammatik-Datei nämlich nur als Warnung angegeben weshalb dieses Problem nicht sofort aufgefallen ist. Ebenso sind Leerzeichen in der Erstellung der Grammatikdatei wichtig, damit die Token korrekt ausgelesen werden.

In der Besprechung wurde uns folgendes als Empfehlung gesagt:

- Wochentage nicht als separate Tokens, stattdessen einen Token Wochentag, den man auch in der Grammatik einfacher nutzen kann
- Sonderbehandlung von Sonderzeichen sollte bei UTF-8 eigentlich nicht nötig sein
- Uhrzeit und "Uhr" separate Tokens
- NUMBER nicht rechts verwenden, stattdessen ein Fragment number nutzen und Datum in Number mit Punkt und Number ohne Punkt aufteilen
- Punkt als separater Token ist wahrscheinlich unnötig
- "bei gutem Wetter" kann auch als ein Token zusammengefasst werden

Ausführung Befehle

- `java -jar ../antlr-4.13.2-complete.jar OpeningTimesLexer.g4`
- `javac -cp ".;..\antlr-4.13.2-complete.jar;" .\TokenPrinter.java .\OpeningTimesLexer.java`
- `java -cp ".;..\antlr-4.13.2-complete.jar;" .\TokenPrinter.java .\OpeningTimesLexer.java`

Ausgabe des Programms

```
Token Type: ID, Token Text: Restaurant
Token Type: ID, Token Text: Schwedenschenke
Token Type: DATE, Token Text: 15. März
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 20. Oktober
Token Type: KW_MITTWOCH, Token Text: Mittwoch
Token Type: KW_BIS, Token Text: bis
Token Type: KW_SONNTAG, Token Text: Sonntag
Token Type: TIME, Token Text: 11.00
```

Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 17.00 Uhr
Token Type: DATE, Token Text: 1. Mai
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 8. September
Token Type: KW_AN, Token Text: an
Token Type: ID, Token Text: Sonn
Token Type: MINUS, Token Text: -
Token Type: ID, Token Text: und
Token Type: ID, Token Text: Feiertagen
Token Type: TIME, Token Text: 11.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 20.00 Uhr
Token Type: KW_MONTAG, Token Text: Montag
Token Type: COMMA, Token Text: ,
Token Type: KW_DIENSTAG, Token Text: Dienstag
Token Type: KW_RUHETAG, Token Text: Ruhetag
Token Type: ID, Token Text: Rathaus
Token Type: ID, Token Text: Seeterrassen
Token Type: DATE, Token Text: 1. Juni
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 8. September
Token Type: ID, Token Text: täglich
Token Type: TIME, Token Text: 9.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 20.00 Uhr
Token Type: DATE, Token Text: 9. September
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 3. November
Token Type: ID, Token Text: täglich
Token Type: TIME, Token Text: 9.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 18.00 Uhr
Token Type: ID, Token Text: Würstle
Token Type: ID, Token Text: Grill
Token Type: DATE, Token Text: 1. Mai
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 8. September
Token Type: ID, Token Text: täglich
Token Type: TIME, Token Text: 10.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 17.00 Uhr
Token Type: DATE, Token Text: 9. September
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 20. Oktober
Token Type: ID, Token Text: täglich
Token Type: TIME, Token Text: 12.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 16.00 Uhr
Token Type: ID, Token Text: Mainau
Token Type: MINUS, Token Text: -
Token Type: ID, Token Text: Träff
Token Type: ID, Token Text: mit
Token Type: ID, Token Text: Hofladen

Token Type: ID, Token Text: vorübergehend
Token Type: KW_GESCHLOSSEN, Token Text: geschlossen
Token Type: ID, Token Text: Schlosscafé
Token Type: ID, Token Text: täglich
Token Type: TIME, Token Text: 11.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 17.00 Uhr
Token Type: ID, Token Text: ab
Token Type: DATE, Token Text: 13. Mai
Token Type: KW_FREITAG, Token Text: Freitag
Token Type: KW_RUHETAG, Token Text: Ruhetag
Token Type: ID, Token Text: Biergarten
Token Type: ID, Token Text: am
Token Type: ID, Token Text: Hafen
Token Type: DATE, Token Text: 23. März
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 6. Oktober
Token Type: KW_MITTWOCH, Token Text: Mittwoch
Token Type: KW_BIS, Token Text: bis
Token Type: KW_SONNTAG, Token Text: Sonntag
Token Type: TIME, Token Text: 11.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 18.00 Uhr
Token Type: KW_MONTAG, Token Text: Montag
Token Type: COMMA, Token Text: ,
Token Type: KW_DIENSTAG, Token Text: Dienstag
Token Type: KW_RUHETAG, Token Text: Ruhetag
Token Type: ID, Token Text: Bäckerei
Token Type: ID, Token Text: Täglich
Token Type: ID, Token Text: Brot
Token Type: DATE, Token Text: 1. Mai
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 6. Oktober
Token Type: KW_FREITAG, Token Text: Freitag
Token Type: KW_BIS, Token Text: bis
Token Type: KW_DIENSTAG, Token Text: Dienstag
Token Type: TIME, Token Text: 10.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 17.00 Uhr
Token Type: DATE, Token Text: 7. Oktober
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 20. Oktober
Token Type: KW_FREITAG, Token Text: Freitag
Token Type: KW_BIS, Token Text: bis
Token Type: KW_DIENSTAG, Token Text: Dienstag
Token Type: TIME, Token Text: 11.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 16.00 Uhr
Token Type: KW_MITTWOCH, Token Text: Mittwoch
Token Type: COMMA, Token Text: ,
Token Type: KW_DONNERSTAG, Token Text: Donnerstag
Token Type: KW_RUHETAG, Token Text: Ruhetag
Token Type: ID, Token Text: Eisdiele
Token Type: ID, Token Text: am

```
Token Type: ID, Token Text: Hafen
Token Type: DATE, Token Text: 23. März
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 6. Oktober
Token Type: TIME, Token Text: 11.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 17.00 Uhr
Token Type: KW_BEI, Token Text: bei
Token Type: KW_GUTEM_WETTER, Token Text: gutem Wetter
Token Type: ID, Token Text: Imbiss
Token Type: ID, Token Text: am
Token Type: ID, Token Text: Schmetterlingshaus
Token Type: DATE, Token Text: 23. März
Token Type: KW_BIS, Token Text: bis
Token Type: DATE, Token Text: 6. Oktober
Token Type: TIME, Token Text: 11.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 16.00 Uhr
Token Type: KW_BEI, Token Text: bei
Token Type: KW_GUTEM_WETTER, Token Text: gutem Wetter
Token Type: ID, Token Text: Café
Token Type: ID, Token Text: Vergissmeinnicht
Token Type: TIME, Token Text: 10.00
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 15.30 Uhr
Token Type: KW_SONNTAG, Token Text: Sonntag
Token Type: TIME, Token Text: 10.30
Token Type: KW_BIS, Token Text: bis
Token Type: TIME, Token Text: 16.00 Uhr
Token Type: KW_SAMSTAG, Token Text: Samstag
Token Type: KW_RUHETAG, Token Text: Ruhetag
```

Aufgabe 2

2a)

Aufgabenstellung

Denken Sie sich eine kleine Sprache aus. Definieren Sie deren Vokabular mit einer ANTLR4 lexer grammar und deren Grammatik mit einer ANTLR4 parser grammar. Erzeugen Sie für einige Beispieltexte mit Hilfe von `org.antlr.v4.gui.TestRig` den Ableitungsbaum (Parse Tree).

Falls Ihnen nichts Eigenes einfällt, bauen Sie eines der beiden Beispiele aus Aufgabe 1 aus.

Idee eigene Sprache

Wir haben uns die Sprache Öffnungszeiten ausgedacht. Sie dient der Definition von Öffnungszeiten für verschiedene Einrichtungen. Die Sprache soll die folgenden Elemente unterstützen:

- Bezeichnung der Einrichtung
- Datumsangaben
- Zeitangaben

- Wochentage
- spezielle Schlüsselwörter zur Definition von Öffnungs- und Schließzeiten
- Regeln für Ausnahmen (z. B. Ruhetage)

Befehle

- `java -jar ../antlr-4.13.2-complete.jar OpeningHoursLexer.g4 OpeningHoursParser.g4`
- `javac -cp ";\..\antlr-4.13.2-complete.jar" OpeningHoursLexer.java OpeningHoursParser.java OpeningHoursParser*.java`
- `java -cp ";\..\antlr-4.13.2-complete.jar" org.antlr.v4.gui.TestRig OpeningHoursParser openingHours -gui example.txt`

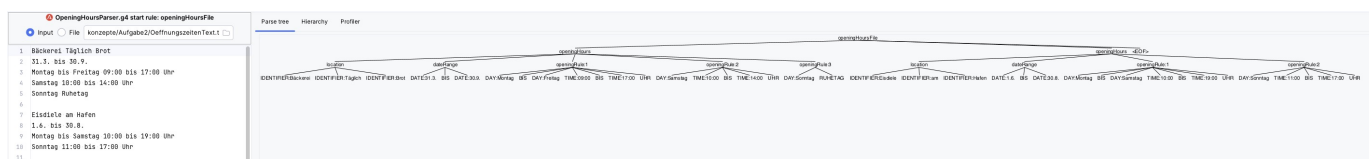
Erklärung

Intelij plugin antlr zur ausführung/erstellung der parse Tree, da wir aktuell keine VM haben und es so deutlich schneller geht.

Relevant files:

- OeffnungszeitenText.txt
- OpeningHoursLexer.g4
- OpeningHoursParser.g4

Parser Bild



Erklärung

- `openingHoursFile`: Zusatzregel, die mehrere aufeinander folgende `openingHours` beschreibt.
- `openingHours`: Hauptregel, die eine Struktur aus `location`, `dateRange` und `openingRule`-Einträgen beschreibt.
- `location`: Name der Einrichtung (z. B. "Restaurant").
- `dateRange`: Gibt den Zeitraum an, in dem die Öffnungszeiten gelten, z. B. 1.3. bis 30.9.
- `openingRule`: Definiert die Öffnungszeitenregeln für einen bestimmten Zeitraum oder Ruhetage.

2b)

Aufgabenstellung

Definieren Sie mit Java-Klassen die abstrakte Syntax Ihrer Sprache aus a) und schreiben Sie ein Java-Programm, das den ANTLR4 Parse Tree in einen AST überführt. Welche Terminale und Nichtterminale aus dem Ableitungsbaum werden in Ihrem AST weggelassen?

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

Im AST bleiben im Wesentlichen die Knoten übrig, die die Bedeutung und Struktur der Öffnungszeiten repräsentieren (z. B. Öffnungszeitenraum, Wochentage, Uhrzeiten), während Zwischenräume, Trennzeichen und andere strukturelle Terminals/Nichtterminals weggelassen werden.

Erklärung

Terminals und Nichtterminals, die im AST weggelassen werden

- Terminals:
 - KW_BIS, KW_UHR und KW_RUHETAG: Diese Schlüsselwörter sind strukturell wichtig, um die Syntax zu definieren, aber im AST überflüssig, da die Klassen ihre Bedeutung direkt kodieren.
 - SEPARATOR, DOT, COLON: Diese Trennzeichen werden im AST nicht benötigt, da die syntaktischen Zusammenhänge bereits in den Strukturen DateRange, OpenHoursRule und RestDayRule abgebildet sind.
- Nichtterminals:
 - openingHours: Die oberste Regel des Parsers wird direkt in den AST überführt, indem OpeningHoursProgram als Wurzelknoten dient.
 - location, dateRange und openingRule: Diese Nichtterminals werden ebenfalls in den AST-Strukturen direkt abgebildet, aber viele Detailknoten und rekursive Teile werden im AST vereinfacht.

Relevant Files

- OpeningHoursProgramm.java
- OpeningHoursProgramm.java

Aufgabe 3

a)

Sie haben in Aufgabe 2 eine kleine Sprache mit konkreter und abstrakter Syntax definiert. Lässt sich eine statische Semantik für Ihre abstrakte Syntax angeben? Erlaubt Ihre konkrete Syntax Formulierungen, die die statische Semantik verletzen? Ergänzen Sie gegebenenfalls eine statische Semantikprüfung für Ihre Sprache. Falls Ihre eigene Sprache hinsichtlich statischer Semantik nichts hergibt, laden Sie die ANTLR4 Java Grammatik herunter und schreiben Sie mit Hilfe der generierten Listener-Klasse eine statische Semantikprüfung, die sicherstellt, dass ganzzahlige Literale ohne L im Zahlbereich von int und mit L im Zahlbereich von long liegen.

a - Lösung

Test Text

```
Bäckerei Täglich Brot
11.3. bis 30.2.
Montag bis Freitag 09:00 bis 05:00 Uhr
Samstag 10:00 bis 14:00 Uhr
Sonntag Ruhetag
```


Eisdiele am Hafen
1.7. bis 30.8.
Montag bis Samstag 20:00 bis 19:00 Uhr
Sonntag 11:00 bis 17:00 Uhr

Statischer Semantic Checker

```
import java.util.*;
import org.antlr.v4.runtime.tree.ParseTreeListener;
import java.text.ParseException;
import java.text.SimpleDateFormat;

public class StaticSemanticsChecker extends OpeningHoursParserBaseListener {
    private final List<String> errors = new ArrayList<>();
    private static final String DATE_FORMAT = "dd.MM";

    @Override
    public void exitDateRange(OpeningHoursParser.DateRangeContext ctx) {
        String startDate = ctx.DATE(0).getText();
        String endDate = ctx.DATE(1).getText();

        if (!isValidDate(startDate) || !isValidDate(endDate)) {
            errors.add("Ungueltiges Datum im Bereich: " + startDate + " bis " +
endDate);
        }
    }

    @Override
    public void exitOpeningRule(OpeningHoursParser.OpeningRuleContext ctx) {
        if (ctx.TIME().size() == 2) {
            String startTime = ctx.TIME(0).getText();
            String endTime = ctx.TIME(1).getText();

            if (!isValidTimeRange(startTime, endTime)) {
                errors.add("Ungueltiger Zeitraum: " + startTime + " bis " +
endTime + " Uhr");
            }
        }
    }

    private boolean isValidDate(String date) {
        SimpleDateFormat sdf = new SimpleDateFormat(DATE_FORMAT);
        sdf.setLenient(false);
        try {
            sdf.parse(date);
            return true;
        } catch (ParseException e) {
            return false;
        }
    }

    private boolean isValidTimeRange(String startTime, String endTime) {
```

```
        return startTime.compareTo(endTime) <= 0;
    }

    public List<String> getErrors() {
        return errors;
    }
}
```

Erklärung

Wir haben eine statische Semantikprüfung für unsere Öffnungszeiten-Sprache implementiert, die sicherstellt, dass Zeiträume logisch sinnvoll sind. Die Prüfung kontrolliert, ob der Starttag vor dem Endtag liegt und die Startzeit vor der Endzeit, um inkonsistente Angaben zu vermeiden. Fehlerhafte Formulierungen werden so frühzeitig erkannt und gemeldet.

Die statische Semantik prüft, ob ein Programm unabhängig von der Ausführung gültig ist. In der Sprache aus Aufgabe 2 können wir folgende statische Semantikregeln definieren:

- Datum-Validierung: Die im `dateRange` verwendeten Datumsangaben müssen gültige Kalendertage sein. Beispielsweise darf es keinen 31. Februar geben.
- Zeit-Validierung: Die Zeiten im `openingRule` müssen im gültigen 24-Stunden-Format angegeben sein (was durch die Lexer-Regeln für `TIME` sichergestellt wird).

Es wäre erst rein Statische Smeantik Überprüfung, wenn es im Beispiel Text z.B. mehrere Geschäfte gäbe und man überprüft, das kein Geschäft mehrfach vorkommt. Dadurch wäre die Eindeutigkeit erreicht.

Verstöße durch die konkrete Syntax

Die konkrete Syntax erlaubt durch die Definition in Aufgabe 2 folgende potenzielle Verstöße gegen die statische Semantik:

- Ein Datum wie 31.2. könnte spezifiziert werden, obwohl es ungültig ist.
- Die Startzeit kann nach der Endzeit liegen, z. B. Montag 18:00 BIS 09:00 Uhr.
- Regeln könnten sich gegenseitig überschneiden oder widersprechen.

b)

Programmieren Sie für Ihre eigene Sprache aus Aufgabe 2 mindestens eine dynamische Semantik.

b - Lösung

```
import java.util.Map;
import java.util.List;
import java.util.HashMap;
import java.util.ArrayList;

public class OpeningHoursInterpreter {
    private final Map<String, List<OpeningRule>> schedule = new HashMap<>();

    public void addRule(String location, String day, String startTime, String
```

```

endTime) {
    schedule.computeIfAbsent(location, k -> new ArrayList<>())
        .add(new OpeningRule(day, startTime, endTime));
}

public boolean isOpen(String location, String day, String time) {
    if (!schedule.containsKey(location)) {
        return false;
    }

    for (OpeningRule rule : schedule.get(location)) {
        if (rule.day.equals(day) &&
            rule.startTime.compareTo(time) <= 0 &&
            rule.endTime.compareTo(time) >= 0) {
            return true;
        }
    }

    return false;
}

private static class OpeningRule {
    String day;
    String startTime;
    String endTime;

    OpeningRule(String day, String startTime, String endTime) {
        this.day = day;
        this.startTime = startTime;
        this.endTime = endTime;
    }
}
}

```

Main

```

import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

import java.nio.file.*;
import java.time.format.DateTimeFormatter;
import java.io.IOException;

public class Main {
    public static void main(String[] args) throws IOException {
        // Read input file
        String inputFilePath = "../Aufgabe2/OeffnungszeitenText.txt";
        String input = Files.readString(Path.of(inputFilePath));

        // Lexical and syntactic analysis
        CharStream charStream = CharStreams.fromString(input);
        OpeningHoursLexer lexer = new OpeningHoursLexer(charStream);
    }
}

```

```

CommonTokenStream tokens = new CommonTokenStream(lexer);
OpeningHoursParser parser = new OpeningHoursParser(tokens);

ParseTree tree = parser.openingHoursFile();

// Static Semantics Check
StaticSemanticsChecker semanticsChecker = new StaticSemanticsChecker();
ParseTreeWalker walker = new ParseTreeWalker();
walker.walk(semanticsChecker, tree);

if (semanticsChecker.getErrors().isEmpty()) {
    System.out.println("Static semantics valid.");
} else {
    System.out.println("Static semantics errors:");
    semanticsChecker.getErrors().forEach(System.out::println);
}

System.out.println("-----");
// Dynamic Semantics Example
OpeningHoursInterpreter interpreter = new OpeningHoursInterpreter();
interpreter.addRule("Restaurant", "Montag", "09:00", "17:00");

boolean isOpen = interpreter.isOpen("Restaurant", "Montag", getNow());
System.out.println("Is Restaurant open at " + getNow() + " on Montag?
" + isOpen);
}

private static String getNow() {
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("HH:mm");
    return dtf.format(java.time.LocalDateTime.now());
}
}

```

Erklärung

Dynamische Semantik für die Sprache

Eine dynamische Semantik beschreibt das Verhalten der Sprache während der Laufzeit. Für die Öffnungszeiten-Sprache könnten wir folgende dynamische Semantik realisieren:

- Abfrage von Öffnungszeiten: Implementation ob ein Gasthaus geöffnet oder geschlossen ist und überprüft wird auf die aktuelle lokale Zeit

Ausführung

- `javac -d ../Aufgabe2/ -cp ";\..\antlr-4.13.2-complete.jar;..\Aufgabe2/" *.java`
- `java -cp ";\..\antlr-4.13.2-complete.jar;..\Aufgabe2/" Main`

```
Static semantics errors:
Ungueltiges Datum im Bereich: 11.3. bis 30.2.
Ungueltiger Zeitraum: 09:00 bis 05:00 Uhr
Ungueltiger Zeitraum: 20:00 bis 19:00 Uhr
-----
Is Restaurant open at 10:38 on Montag? true
```

Aufgabe 4

4a)

Aufgabenstellung

Vervollständigen Sie das folgende Java-Programm, indem Sie die aufgerufenen Klassenmethoden ergänzen. Implementieren Sie die Klassenmethoden mit Schleifen und Verzweigungen. Was an dem vervollständigten Java-Programm ist alles eindeutig imperativer bzw. prozeduraler Stil? Hinweise: Leere Zeilen sind Zeilen, die nichts oder nur Whitespace enthalten. Kurze Zeilen sind Zeilen, die weniger als MIN_LENGTH Zeichen enthalten.

Lösung

```
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();
        // Sequenzielle Verarbeitung der Zeilen nacheinander
        // Seiteneffekt auf lines
        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 IOException {
        String line;
        while ((line = reader.readLine()) != null) {
            // Eindeutig prozedurale Verarbeitung der Zeilen
            lines.add(line);
        }
    }
}
```

```

private static void removeEmptyLines(LinkedList<String> lines) {
    for (Iterator<String> it = lines.iterator(); it.hasNext(); ) {
        String line = it.next();
        if (line.trim().isEmpty()) {
            // Eindeutig prozedurales Löschen der Zeilen
            it.remove();
        }
    }
}

private static void removeShortLines(LinkedList<String> lines) {
    for (Iterator<String> it = lines.iterator(); it.hasNext(); ) {
        String line = it.next();
        if (line.length() < MIN_LENGTH) {
            // Eindeutig prozedurales Löschen der Zeilen
            it.remove();
        }
    }
}

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

```

4b)

Aufgabenstellung

Stellen Sie das Programm aus 4a mithilfe von `java.util.streams` und Lambdas auf einen funktionalen Stil um. Ihr Programm darf nach der Umstellung keine Schleifen, Verzweigungen und Seiteneffekte mehr enthalten.

Lösung

```

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]);

        long start = System.nanoTime();

        // Keine Seiteneffekte auf eine Liste
        int n = Files.lines(input)
            .filter(line -> !line.trim().isEmpty())

```

```
        .mapToInt(String::length)
        .filter(length -> length >= MIN_LENGTH)
        .sum();

    long stop = System.nanoTime();

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

4c)

Aufgabenstellung

Vergleichen Sie die Laufzeiten der Programme aus 4a und 4b.

Ausführung Befehle

- javac Procedural.java
- javac Functional.java
- java Procedural input.txt
- java Functional input.txt

Lösung

Bei der kompletten englischen Bibel mit einer totalen Zeilenlänge von ca. 4 Mio. Zeichen benötigt das prozedurale Programm aus 4 a) etwa 60 Millisekunden, während das funktionale Programm aus 4 b) etwa 67 Millisekunden benötigt. Der kaum vorhandene Unterschied ist wahrscheinlich auf das Betriebssystem (MacOS) zurückzuführen, da die Laufzeiten sehr ähnlich sind. Grundsätzlich sollte aber der prozedurale Stil langsamer sein, da er mehr Overhead mit Iteratoren hat.

Aufgabe 5

a)

Lösen Sie die Aufgaben von

Folie 25 (rechte Spalte der Tabelle)

Lösung

List 1	List 2	Result
[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]	false
[white Q]	[P,horse]	P = white, Q = [horse]

26 (Berechnung Fakultät)


28 (Anfragen letzter Spiegelpunkt) aus Eck-Prolog.pdf.

b)

Programmieren Sie ein Prädikat `sum`, das die Summe einer Liste von Zahlen berechnet. Hinweis: Sie müssen Rekursion verwenden.

Lösung

```
sum ([ ] , 0).
sum ([H|T], R) :-
  sum (T, Rest ),
  R is H + Rest .
```



The screenshot shows a Prolog interpreter window. The query `sum([1, 2, 3], R)` is entered in the top bar. Below the bar, the variable `R` is shown in red, indicating it is a result. At the bottom left, the number `6` is displayed, which is the sum of the list [1, 2, 3].

c)

Sie wollen zu einer Werksbesichtigung von BioNTech in Mainz reisen. Dazu brauchen Sie eine Bahnverbindung. Gegeben sind die folgenden 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)`.

Definieren Sie ein Prädikat `verbindung`, das beschreibt, ob zwischen zwei Städten nach einer gegebenen Abfahrtszeit eine Verbindung inklusive Umsteigen existiert. Hinweise: Sie brauchen auch hier Rekursion. Beim Umsteigen muss Abfahrtszeit > Ankunftszeit gelten.

Eine Abfrage `verbindung(konstanz, 8.00, mainz, Reiseplan)` soll nacheinander die möglichen Reiseverbindungen nach 8 Uhr in der Variablen `Reiseplan` liefern. Die Variable `Reiseplan` soll eine Liste von Teilstrecken in Form von `zug`-Strukturen sein.

Lösung

```
zug( konstanz , 08.39 , offenburg , 10.59) .
zug( konstanz , 08.39 , karlsruhe , 11.49) .
zug( konstanz , 08.53 , singen , 09.26) .
zug(singen , 09.37 , stuttgart , 11.32) .
zug( offenburg , 11.29 , mannheim , 12.24) .
zug( karlsruhe , 12.06 , mainz , 13.47) .
zug( stuttgart , 11.51 , mannheim , 12.28) .
zug( mannheim , 12.39 , mainz , 13.18) .

verbindung(Start, Startzeit, Ziel, [zug(Start, Abfahrtszeit, Ziel, Ankunftszeit)])
:-
```



```

zug(Start, Abfahrtszeit, Ziel, Ankunftszeit),
Abfahrtszeit >= Startzeit.

verbindung(Start, Abfahrtszeit, Ziel, [zug(Start, Abfahrtszeit, Zwischenhalt,
Ankunftszeit) | Rest]) :-
zug(Start, Abfahrtszeit, Zwischenhalt, Ankunftszeit),
Abfahrtszeit >= Abfahrtszeit,
verbindung(Zwischenhalt, Ankunftszeit, Ziel, Rest).

```

Aufgabe 6

Implementieren Sie eine Java-Anwendung, die für beliebige Java-Klassen und -Interfaces eine HTML-Seite im Format der Beispieldatei aufgabe6.html (siehe Moodle-Kursseite) generiert. Leiten Sie dazu aus aufgabe6.html eine Stringtemplategroup-Datei aufgabe6.stg ab. Die Java-Anwendung soll die gewünschten voll qualifizierten Klassen- und Interfacenamen als Aufrufparameter bekommen und mit Hilfe der Templates die HTML-Darstellung erzeugen.

Hinweise:

- Übergeben Sie dem Wurzel-Template eine Collection oder ein Array von Class<?>-Objekten. Die Objekte erzeugen Sie mit Class.forName(String). Die Stringtemplate-Bibliothek ist in der Antlr-Bibliothek enthalten, die Sie bei vorhergehenden Aufgaben bereits verwendet haben.

Lösung

Eine Java-Anwendung wurde entwickelt, die aus voll qualifizierten Klassen- und Interfacenamen automatisch eine HTML-Datei erstellt. Diese Datei zeigt die implementierten Interfaces und Methoden der angegebenen Klassen oder Interfaces an.

```

import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class HTMLGenerator {

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

        StringBuilder html = new StringBuilder();
        html.append("<!DOCTYPE html>\n<html lang=\"de\">\n<head>\n<style
type=\"text/css\">\n")
            .append("th, td { border-bottom: thin solid; padding: 4px; text-align:
left; }\n")
            .append("td { font-family: monospace }\n</style>\n</head>\n<body>\n")
            .append("<h1>Sprachkonzepte, Aufgabe 6</h1>\n");
    }
}

```

```

        for (String className : args) {
            try {
                Class<?> clazz = Class.forName(className);
                generateHTMLForClass(clazz, html);
            } catch (ClassNotFoundException e) {
                html.append("<p><b>Klasse oder Interface nicht gefunden:</b>")
            }.append(className).append("</p>\n");
        }

        html.append("</body>\n</html>");
        System.out.println(html);
    }

    private static void generateHTMLForClass(Class<?> clazz, StringBuilder html) {
        html.append("<h2>").append(clazz.isInterface() ? "interface " : "class ")
            .append(clazz.getName()).append(":</h2>\n");
        .append("<table>\n");

        if (clazz.isInterface()) {
            // Methoden direkt anzeigen, wenn es ein Interface ist
            appendMethods(clazz.getMethods(), "Methods", html);
        } else {
            // Interfaces anzeigen
            Class<?>[] interfaces = clazz.getInterfaces();
            if (interfaces.length > 0) {
                html.append("<tr><th>Interface</th><th>Methods</th></tr>\n");
                for (Class<?> iface : interfaces) {
                    html.append("<tr>\n<td ")
                }.append(iface.getName()).append("</td>\n");
                html.append("
<td>").append(formatMethods(iface.getMethods())).append("</td>\n</tr>\n");
            }
        }

        html.append("</table>\n<br>\n");
    }

    private static void appendMethods(Method[] methods, String header,
        StringBuilder html) {
        if (methods.length > 0) {
            html.append("<tr><th>").append(header).append("</th></tr>\n<tr><td>")
                .append(formatMethods(methods))
                .append("</td></tr>\n");
        }
    }

    private static String formatMethods(Method[] methods) {
        List<String> methodSignatures = new ArrayList<>();
        for (Method method : methods) {
            String signature = method.getReturnType().getTypeName() + " " +
                method.getName() + "(" +

```

```

        Arrays.stream(method.getParameterTypes())
            .map(Class::getTypeName)
            .reduce((a, b) -> a + ", " + b)
            .orElse("") +
        ")";
        methodSignatures.add(signature);
    }
    return String.join("<br>\n", methodSignatures);
}
}

```

Das Stringtemplate selbst ist in der Datei aufgabe6.stg abgelegt und sieht wie folgt aus:

```

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>
<% for c in classes %>
<h2><% if (c.name.startsWith("interface")) { "interface " } else { "class " } %><%
c.name %>:</h2>
<table>
<% if (c.interfaces.size() > 0) { %>
<tr><th>Interface</th><th>Methods</th></tr>
<% for i in c.interfaces %>
<tr>
<td valign=top><% i.name %></td>
<td><% i.methods; separator="<br>" %></td>
</tr>
<% end %>
<% } %>
<% if (c.methods != null && c.methods.size() > 0) { %>
<tr><th>Methods</th></tr>
<tr><td><% c.methods; separator="<br>" %></td></tr>
<% } %>
</table>
<br>
<% end %>
</body>
</html>
>>

```

Die Anwendung liest die Klassennamen aus den Eingabeparametern, verwendet Reflection, um Informationen wie Methoden und Interfaces auszulesen, und formatiert diese Daten in HTML. Die Ausgabe wird direkt in die

Konsole geschrieben oder kann in eine HTML-Datei umgeleitet werden.

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()

Aufgabe 7

Implementieren Sie eine kleine Anwendung mit einer Scriptsprache und analysieren Sie, welche typischen Eigenschaften einer Scriptsprache Sie dabei ausnutzen.

Vorschläge:

- JavaScript in einer Webseite
- Abfrage eines Webservice mit Python, z.B. Feiertage bei feiertage-api.de oder Wechselkurse bei zoll.de -> Service -> Online-Fachanwendungen ...

Lösung

Wir haben die Script Sprache Python verwendet und das vorgeschlagene Beispiel einen Webservice aufzurufen.

```
import requests

def get_holidays(year, country_code):
    """
    Ruft Feiertage für ein bestimmtes Jahr und Land ab.
    """
```

```
url = f"https://feiertage-api.de/api/?jahr={year}&nur_land={country_code}"
try:
    response = requests.get(url)
    response.raise_for_status()
    holidays = response.json()
    return holidays
except requests.RequestException as e:
    print(f"Fehler beim Abrufen der Feiertage: {e}")
    return None

def main():
    year = input("Geben Sie das Jahr ein (z.B. 2024): ")
    country_code = input("Geben Sie den Ländercode ein (z.B. BW für Baden  
Württemberg): ").upper()
    holidays = get_holidays(year, country_code)

    if holidays:
        print(f"\nFeiertage in {country_code} für {year}:")
        for name, details in holidays.items():
            print(f"- {name}: {details['datum']}")
    else:
        print("Keine Feiertage gefunden.")

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

Erklärung

Durch die dynamische Typisierung können Variablen ohne explizite Typdeklaration verwendet werden, was die Entwicklung vereinfacht und beschleunigt. Da Python eine interpretierte Sprache ist, wird der Code direkt zur Laufzeit ausgeführt, wodurch Änderungen sofort getestet werden können, ohne dass ein Kompilierungsschritt erforderlich ist. Die hohe Abstraktion der Sprache ermöglicht es, komplexe Aufgaben wie HTTP-Anfragen mit wenigen Zeilen Code und Bibliotheken wie requests umzusetzen. Fehlerbehandlung ist einfach und robust durch die Verwendung von Ausnahmen, wodurch beispielsweise Netzwerkprobleme kontrolliert abgefangen werden. Zusätzlich erweitert die Unterstützung durch umfangreiche Bibliotheken den Funktionsumfang erheblich, ohne dass komplexer Eigenaufwand nötig ist. Schließlich erlaubt die interaktive Entwicklungsumgebung von Python ein schnelles Prototyping und erleichtert die Anpassung des Codes während der Entwicklung.