Sprachkonzepte

WS 23/24 – Tobias Mack & Julian Bihl

GitHub

Link zu GitHub-Repository: GitHub-Sprachkonzepte

Aufgabe 1 a)

Regex:
according to java.util.Formatter:

%[argument_index$][flags][width][.precision]conversion

- The optional argument_index is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "1$", the second by "2$", etc.
- The optional flags is a set of characters that modify the output format. The set of valid flags depends on the conversion.
- The optional width is a positive decimal integer indicating the minimum number of characters to be written to the output.
- The optional precision is a non-negative decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.
- The required conversion is a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type.

Da für Date/Time Conversions auf ein 't' oder 'T' ein weiterer Buchstabe folgt wird dies gesondert behandelt.

Um die Übersicht zu behalten, wird für jede Kategorie ein Regulärer Ausdruck erstellt:

```
    private static String getFormatterRegex() {
        String argumentIndexRegex = "%(\\d+\\$)?";
        String flagsRegex = "([-#+ 0,(]*)";
        String widthRegex = "(\\d+)?";
        String precisionRegex = "(\\.\\d+)?";
        String conversionRegex = "[a-zA-Z%]";

        // Time-related specifiers
        String timeSpecifierRegex = "[tT]([a-zA-Z])";

        return argumentIndexRegex + flagsRegex + widthRegex + precisionRegex + "(" + timeSpecifierRegex + "|" + conversionRegex + ")";
    }
}
```

Input:

```
String[] inputs = {
    "xxx %d yyy%n",
    "xxx% 012d yyy%%",
    "xxx%1$d yyy",
    "%1$0+(32.10fyyy",
    "Wochentag: %tA Uhrzeit: %tT"
};
```

Output:

TEXT("xxx ")FORMAT("%d")TEXT(" yyy")FORMAT("%n")
TEXT("xxx")FORMAT("% 012d")TEXT(" yyy")FORMAT("%%")
TEXT("xxx")FORMAT("%1$d")TEXT(" yyy")
FORMAT("%1$0+(32.10f")TEXT("yyy")
TEXT("Wochentag: ")FORMAT("%tA")TEXT(" Uhrzeit: ")FORMAT("%tT")

Aufgabe 1 b)

Für den Lexer haben wir eine Rule Clock die den Token für eine Zeitangabe erkennt und an den Parser weitergeben kann. Weitere fragment Rules bauen die Logik des Uhrzeitformats auf.

```
1   // ExprLexer.g4
2   lexer grammar TwelveHourClockLexer;
3
4
5   Clock: Normal | Noon | Midnight ;
6
7   fragment Normal: Hour ':' Minute ' ' Meridiem ;
8   fragment Noon: 'Noon' | '12 noon' ;
9   fragment Midnight: 'Midnight' | '12 midnight' ;
10
11  fragment Hour: '1'[0-2]|'0'[1-9] ;
12  fragment Minute: [0-5][0-9] ;
13  fragment Meridiem: 'p.m.' | 'a.m.' ;
14
15  WS: [ \t\r\n]+ -> channel(HIDDEN);
```

Zeile 5: Es gibt 3 Möglichkeiten wie eine Uhrzeit aussieht
Zeile 6: Normal ist eine Uhrzeit, die in Stunde:Minute und Tageszeitangabe a.m. p.m. aufgeteilt wird.
Zeile7&8: Bei Noon und Midnight gibt es jeweils die zwei aufgeführten Optionen.
Zeile11: Stunde beginnt entweder mit 1 und es folgt 0-2 oder sie beginnt mit einer 0 und es folgt eine 1-9.
Zeile12: Bei der Minute kann die erste Ziffer nur eine 0-5 sein.

Zeile 13: Meridiem gibt es nur die 2 aufgeführten Operationen
Zeile 15: Whitespace soll nicht als Token angezeigt werden.

Testdaten in test.txt

```
12 noon
09:12 p.m.
Midnight
```

Test:
java -cp ".antlr-4.13.1-complete.jar" org.antlr.v4.gui.TestRig Time tokens -tokens "test.txt"

Ergebnis:

[@0,0:6='12 noon',<Clock>,1:0]
[@1,7:8='\r\n',<WS>,channel=1,1:7]
[@2,9:18='09:12 p.m.',<Clock>,2:0]
[@3,19:20='\r\n',<WS>,channel=1,2:10]
[@4,21:28='Midnight',<Clock>,3:0]
[@5,29:28='<EOF>',<EOF>,3:8]

Aufgabe 2 a)

Die folgende Sprache soll eine Reisedatenbeschreibung verstehen, die z.B. wie folgt aussieht.

Beispiel 1 (excample2.txt)

```
DepartureDate: 1/3/23
TripTitle: [Beach Holiday.]
City:    [Miami]
Country: [United States]
ReturnDate: 3/04/23
```

Für den Lexer wurden dafür die jeweiligen Bezeichnungen als Token beschrieben. Die Einträge werden durch eckige Klammern erkannt, und können beliebig viele Zeichen enthalten. Das Datum besteht aus einem Tag einem Monat und einem Jahr, welche über '/' getrennt werden.

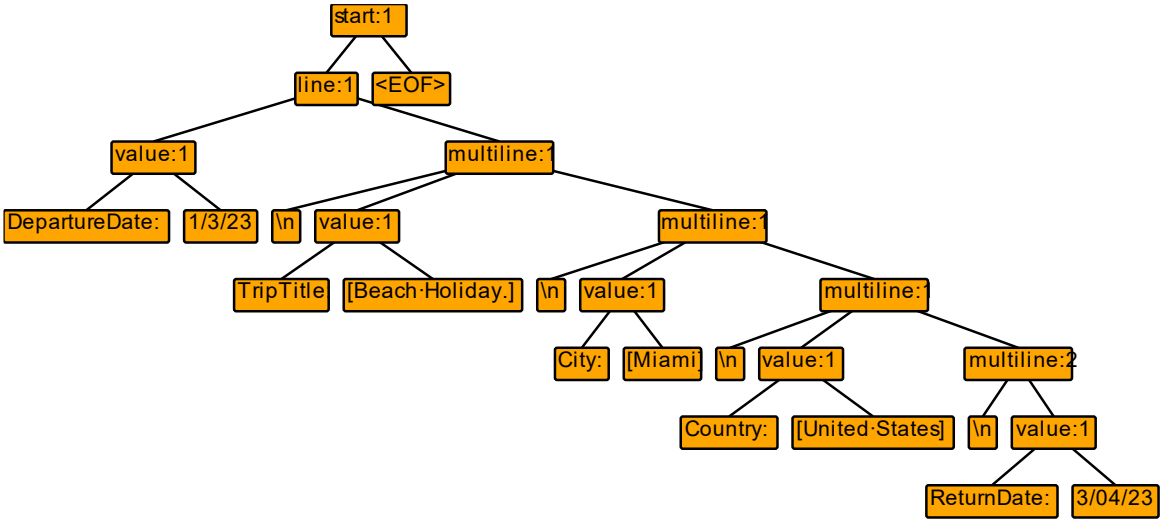
Im Parser setzt man nun die Bestandteile zusammen und fächert die Subkategorien weiter auf. Die Regel multiline dient dazu das Dokument weiterzuführen bzw. über ein NEWLINE die values zu trennen.

```
start: line EOF;
line: value multiline;
multiline: NEWLINE value multiline | NEWLINE value;
value: (TITLE ( DATE | ENTRY ));
```

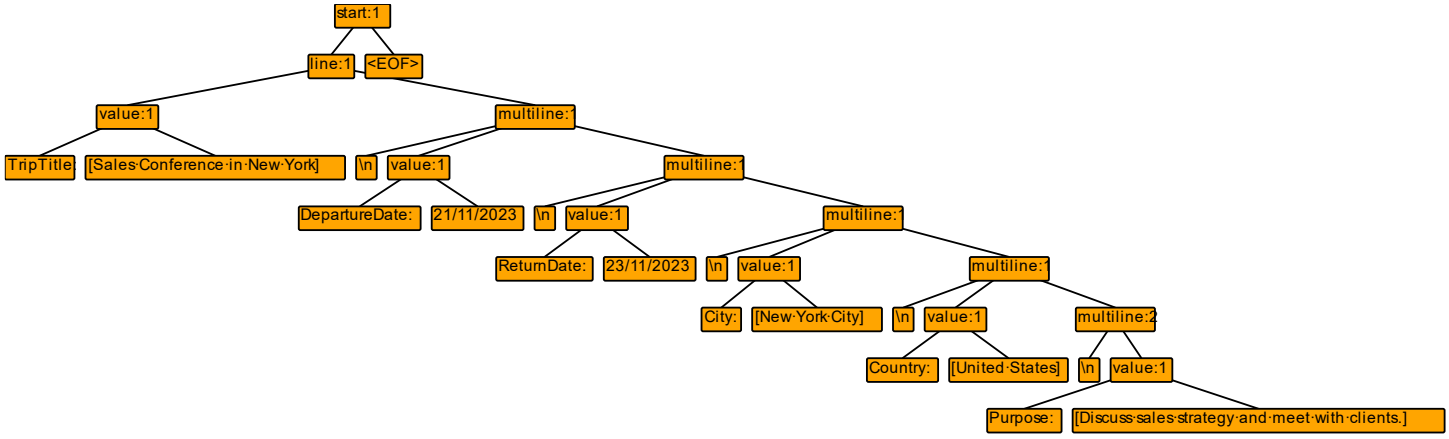
Anschließend wurde mit den Antlr-tools der Ableitungsbaum generiert mit dem Befehl:

antlr4-parse .\TripLexer.g4 .\TripParser.g4 trip -gui ".\example.txt"

Beispiel 1 (excample2.txt)



Beispiel 2 (example.txt)



Aufgabe 2 b)

Zuerst wurde der Java Lexer, Parser und Listener mit dem folgenden Befehl aus der Grammatik generiert:

antlr4 .\TripLexer.g4 .\TripParser.g4

Anschließend wurden die einzelnen Klassen erstellt die von der abstrakten Klasse "Trip" erben.

- Line
- Multiline
- Value

TripToAst:

Der Lexer liest einen CharStream welcher dann dem Parser übergeben wird. Durch den Parser wird der Parsetree erstellt und dem TripBuilder übergeben.

TripBuilder:

Insgesamt dient der TripBuilder dazu, den Parsebaum zu durchlaufen und die darin enthaltenen Informationen in Form von Trip-Objekten zu strukturieren. Dabei wird ein Stack verwendet, um temporäre Trip-Objekte zu verwalten, während der Parsebaum durchlaufen wird. Er verarbeitet verschiedene Parser-Regeln wie "Value", "Line" und "Multiline" welche in den Exitfunktionen aufgegriffen und verarbeitet werden. Dies ermöglicht die Erstellung eines hierarchisch strukturierten Objekts, das den analysierten Inhalt repräsentiert.

Aufgabe 3 a)

Die statische Semantik ist durch die Methode 'staticSemanticTest' in TripToAst umgesetzt.

Es wird überprüft ob die Entries eine Länge von min 5 - max 50 Zeichen haben. So kann bereits vor Laufzeit geklärt werden ob ein Fehler auftritt.

Aufgabe 3b)

Die dynamische Semantik ist durch die Methode 'dynamicSemanticTest' in TripToAst umgesetzt.

Hier findet keine Überprüfung in dem Sinne statt, sondern eine Astverarbeitung, also wird dies erst zur Laufzeit durchgeführt.

Statischer Teil der Semantikprüfung:

- Wenn mind. zwei Daten im Text gefunden werden, wird überprüft ob das erste Datum zeitlich vor dem zweiten Datum liegt.

Dynamischer Teil der Semantikprüfung:

- eine weitere Verarbeitung des Asts wird durchgeführt: Die Daten werden in das importierte Datumsformat SimpleDateFormat weiterverarbeitet und die Funktion gibt ein Objekt des Typs Date zurück. Funktioniert das nicht kommt es zu einer ParseException.

Aufgabe 4 a)

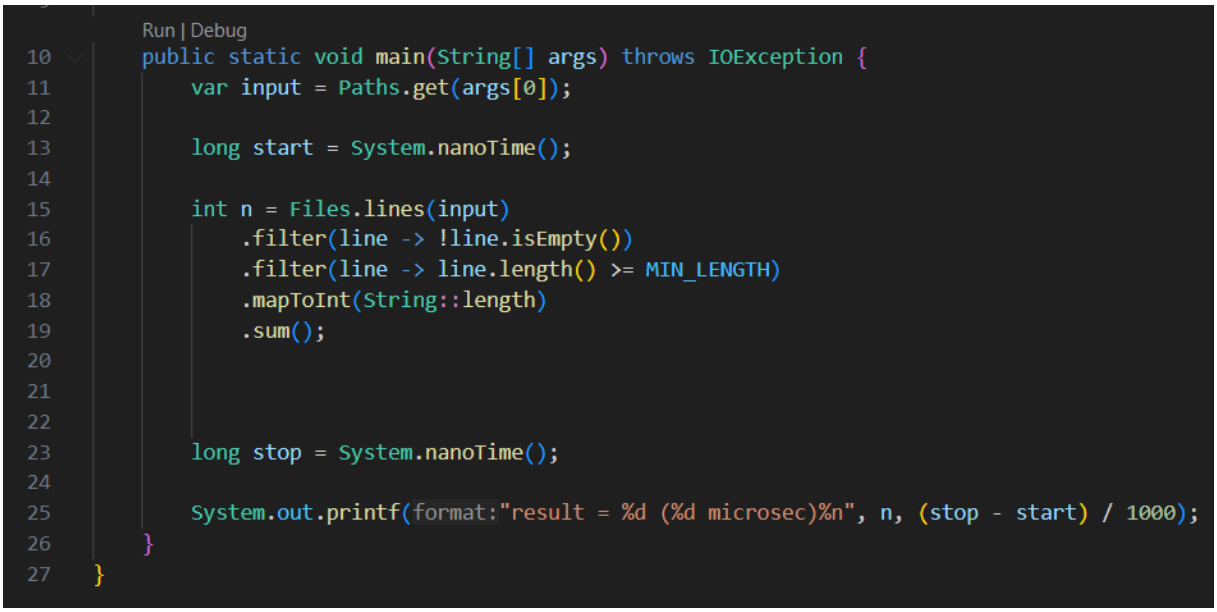
Das Aufrufen der einzelnen Methoden in Zeile 19-22 sind eindeutig im Proceduralen Stil. Sie akzeptieren Argumente als Parameter und mutieren diese als Seiteneffekt. Die einzelnen Methoden sind wiederrum ebenfalls stark Procedural durch die Verwendung von Schleifen und bedingten Anweisungen um auf Daten zuzugreifen und sie zu ändern.

```
18
19          readLines(Files.newBufferedReader(input), lines);
20          removeEmptyLines(lines);
21          removeShortLines(lines);
22          int n = totalLineLengths(lines);
```

```
30      private static void readLines(BufferedReader input, LinkedList<String> lines) throws IOException {
31          String line;
32          while((line = input.readLine()) != null) {
33              lines.add(line);
34          }
35      }
36
37      private static void removeEmptyLines(LinkedList<String> lines) {
38          for (int i = 0; i < lines.size() - 1; i++){
39              lines.remove(o:"");
40          }
41      }
42
43      private static void removeShortLines(LinkedList<String> lines) {
44          var list = new LinkedList<String>();
45          for (String line : lines) {
46              if (line.length()<MIN_LENGTH) {
47                  list.add(line);
48              }
49          }
50          for (String shortLine : list) {
51              lines.remove(shortLine);
52          }
53      }
54
55      private static int totalLineLengths(LinkedList<String> lines) {
56          int num = 0;
57          for(String line: lines) {
```

Aufgabe 4 b)

Das auf den funktionalen Stil umgeänderte Programm:



```
Run | Debug
public static void main(String[] args) throws IOException {
    var input = Paths.get(args[0]);

    long start = System.nanoTime();

    int n = Files.lines(input)
        .filter(line -> !line.isEmpty())
        .filter(line -> line.length() >= MIN_LENGTH)
        .mapToInt(String::length)
        .sum();



    long stop = System.nanoTime();

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

Aufgabe 4 c)

Um das Laufzeitverhalten zu überprüfen haben wir mehrmals mit geringer, mittlere und hoher Zeilenanzahl getestet.

Bei kleinen Testdaten ist die Prozedurale Implementierung deutlich schneller. Das könnte am Overhead des Streams liegen, bei welchem viele temporäre Objekte erstellt und wieder freigeben werden müssen. Je größer die Testdaten werden, umso schneller wird das Funktionale Programm im Vergleich zum Prozeduralen. Eine Erklärung ist die Anzahl der Durchläufe der Daten. Das Prozedurale durchläuft die Daten mehrmals. Einmal zum Lesen der Zeilen, zum Entfernen der leeren Zeilen und zum Entfernen der zu kurzen Zeilen. Das Funktionale Programm wiederum nur einmal.

Zum anderen könnte die Verwendung der LinkedList die Laufzeit erhöhen. Weil die Entfernung von vielen Elementen zeitaufwändig sein kann, da die LinkedList jedes Mal neu organisiert werden muss. Das Prozedurale braucht vermutlich bei großer Datenmenge mehr Speicher, da alle Daten gleichzeitig im Speicher gehalten werden. Beim Funktionalen, wird durch Streams die lazy Verarbeitung ermöglicht, wobei die Daten erst verarbeitet werden, wenn man sie braucht.

Testgröße	Prozedural	Funktional
355 Zeilen	8848 microsec	17380 microsec
2000 Zeilen	27725 microsec	18017 microsec
10000 Zeilen	300007 microsec	22326 microsec

Aufgabe 5 a)

1. Matching der 2 Listen:

Regel:
p(Liste1-Value). zb. p([X,Y,Z]).

Abfrage:
?-p(Liste2-Value). zb. ?-p([john,likes,fish]). => X = john, Y = likes, Z = fish

2. Die Fakultät muss die Zahl N rekursiv mit N-1 multiplizieren:

```
fakultaet(N, Ergebnis) :-
        N > 0,
        N_minus_1 is N - 1,
        fakultaet(N_minus_1, Ergebnis_minus_1),
        Ergebnis is N * Ergebnis_minus_1.
```

?-fakultaet(3, X).

Ergebnis = 6

3. Funktionen zum Konkatenieren von zwei Listen

```
append([],L,L).
append([H|T1],L,[H|T2]) :- append(T1,L,T2).
```

append(X, Y, [1,2,3,4]). berechnet alle Möglichkeiten die die Ergebnisliste ausgeben:

X = [],
Y = [1, 2, 3, 4]
X = [1],
Y = [2, 3, 4]
X = [1, 2],
Y = [3, 4]
X = [1, 2, 3],
Y = [4]
X = [1, 2, 3, 4],
Y = []

append(X, [1,2,3,4], Z). berechnet alle Möglichkeiten, wenn Liste X mit Liste [1,2,3,4] appended wird.

X = [],
Z = [1, 2, 3, 4]
X = [_1368],
Z = [_1368, 1, 2, 3, 4]
X = [_1368, _1374],
Z = [_1368, _1374, 1, 2, 3, 4]
X = [_1368, _1374, _1380],
Z = [_1368, _1374, _1380, 1, 2, 3, 4]
...

unendlich viele sinnfreie Möglichkeiten mit Verwendung von Cut vermeidbar:

```
append([],L,L) :- !.
append([H|T1],L,[H|T2]) :- append(T1,L,T2).
```

Aufgabe 5 b)

Summenberechnung einer Liste ebenfalls rekursiv wie bei Fakultät:

```
sum([], 0).
sum([Head|Tail], Sum) :-
sum(Tail, TailSum),
Sum is Head + TailSum.
```

Beispielausgabe:

```
?-sum([2, 3, 40, 5], Result).
Result = 50
```

Aufgabe 5 c)

Gegebene Fakten:

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, 11.27, mannheim, 12.24).
zug(karlsruhe, 12.06, mainz, 13.47).
zug(stuttgart, 11.51, mannheim, 12.28).
zug(mannheim, 12.39, mainz, 13.18).

Die Abfrage soll wie folgt aussehen:

?-verbindung(konstanz, 8.00, mainz, Reiseplan)

Um nun eine direkte Verbindung zu finden wie zb Konstanz - Offenburg wird folgendes Prädikat verwendet:

```
verbindung(Start, Abfahrtszeit, Ziel, Reiseplan) :-
        zug(Start, Abfahrt, Ziel, Ankunft),
        Abfahrt > Abfahrtszeit,
        Reiseplan = [Start, Abfahrt, Ziel, Ankunft].
```

Um Zwischenziele verwenden zu können muss ein weiteres Prädikat hinzugefügt werden, welches verbindung rekursiv aufruft und die bestehende Reise weitergibt, damit der "Fahrplan" nicht verloren geht.

```
verbindung(Start, Abfahrtszeit, Ziel, Reiseplan) :-
        zug(Start, Abfahrt, Zwischenziel, Ankunft),
        Abfahrt > Abfahrtszeit,
        Reiseplan = [(Start, Abfahrt, Zwischenziel, Ankunft) | Reise],
        verbindung(Zwischenziel, Ankunft, Ziel, Reise).
```

Die Abfrage gibt anschließend 3 Möglichkeiten aus. Bei dem vierten Versuch kommt ein false da keine Möglichkeiten mehr gefunden werden.

?-verbindung(konstanz, 8.00, mainz, Reiseplan)
Reiseplan = [(konstanz,8.39,offenburg,10.59), (offenburg,11.27,mannheim,12.24), mannheim, 12.39, mainz, 13.18]
Reiseplan = [(konstanz,8.39,karlsruhe,11.49), karlsruhe, 12.06, mainz, 13.47]
Reiseplan = [(konstanz,8.53,singen,9.26), (singen,9.37,stuttgart,11.32), (stuttgart,11.51,mannheim,12.28), mannheim, 12.39, mainz, 13.18]
false

Aufgabe 6

Um eine Html-Datei im richtigen Format für beliebige Java-Klassen -Interfaces zu generieren haben wir eine Stringtemplategroup-Datei mit den Templates aufgabe6, classInfo und interfaceInto erstellt.

```
// aufgabe06.stg
delimiters "$", "$"

aufgabe06(list) ::= <<
<!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>
        $list:classInfo(); separator="\n"$
    </body>
</html>
>>
```

```
classInfo(c) ::= <<
<h2>$if(c.isInterface)$interface $else$class $endif$$c.name$:</h2>
    <table>
        <tr>
            $if(c.isInterface)$
                <th>Methods</th>
            $else$
                <th>Interface</th>
                <th>Methods</th>
            $endif$
        </tr>
            $c.interfaces:interfaceInfo(); separator="\n"$
    </table>
<br>
>>

interfaceInfo(i)::= <<
<tr>
    $if(c.isInterface)$
        <td>$i.methods :{ m | $m$ <br>}$</td>
    $else$
        <td valign=top>$i.name$</td>
        <td>$i.methods :{ m | $m$ <br>}$</td>
    $endif$
</tr>
>>
```

Die Klassen ClassInfo und InterfaceInfo wurden erstellt um die notwendigen Informationen der Java-Klassen und -Interfaces bereitzustellen.

```
32      public ClassInfo(Class<?> c) {
33          this.name = c.getName();
34          this.interfaces = new LinkedList<>();
35          this.isInterface = false;
36
37          if (c.isInterface()) {
38              this.isInterface = true;
39              var currentInterface = new InterfaceInfo(c.getName());
40              for (var m : c.getMethods()) {
41                  var parameterTypes = Arrays.stream(m.getParameterTypes())
42                          .map(Class::getName)
43                          .collect(Collectors.joining(delimiter:", "));
44                  currentInterface.methods
45                          .add(m.getReturnType().getName() + " " + m.getName() + "(" + parameterTypes + ")");
46              }
47              this.interfaces.add(currentInterface);
48          } else {
49              for (var i : c.getInterfaces()) {
50                  var currentInterface = new InterfaceInfo(i.getName());
51
52                  for (var j : i.getMethods()) {
53                      var parameterTypes = Arrays.stream(j.getParameterTypes())
54                              .map(Class::getName)
55                              .collect(Collectors.joining(delimiter:", "));
56                      currentInterface.methods
57                              .add(j.getReturnType().getName() + " " + j.getName() + "(" + parameterTypes + ")");
58                  }
59
60                  this.interfaces.add(currentInterface);
61              }
62          }
63      }
64
65  }
```

```
67  final class InterfaceInfo {
68
69      public final String name;
70      public final LinkedList<String> methods;
71
72      public InterfaceInfo(String name) {
73          this.name = name;
74          this.methods = new LinkedList<>();
75      }
76  }
```

Aufgabe 7

Das Python Programm ruft die API feiertage-api.de mit den Parametern Jahr und Bundesland=Baden-Württemberg auf. Dabei werden die Feiertage die im JSON format ankommen mit dem aktuellen Datum abgeglichen und gefiltert sodass die nächsten Feiertage mit Wochentag ausgegeben werden. Die Ausgabe sieht wie folgt aus:

Es sind 12 Feiertage in diesem Jahr übrig.
Die nächsten 3 Feiertage sind:

Gründonnerstag am 2024-03-28 (Thursday)
Karfreitag am 2024-03-29 (Friday)
Ostermontag am 2024-04-01 (Monday)

Typische Eigenschaften von Scriptsprachen die hier verwendet wurden sind:

- Syntax ist generell sehr einfach gehalten -> hohe Abstraktion auch durch Nutzung externer Bibliotheken
- eine deklarationsfreie Syntax, sprich eine implizite Deklaration von Namen und eine dynamische Typisierung
- automatische Speicherverwaltung, es wurde keine manuelle über gc o.Ä. benutzt
- zeilenweise Interpretation, vor allem in Verbindung mit JupyterNotebook hilfreich für Entwicklungsprozess
- Umgang mit Strings vereinfacht: Variablen können in String eingefügt werden:

api_url = f"https://feiertage-api.de/api/?jahr={year}&nur_land={bundesland}"