

AIN - Sprachkonzepte

Übungsaufgaben

Bericht

WS2022/23

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

1	Aufgabe 1	1
2	Aufgabe 2	7
3	Aufgabe 3	13
4	Aufgabe 4	18
5	Aufgabe 5	22
6	Aufgabe 6	26
7	Aufgabe 7	30

Abbildungsverzeichnis

1.1	Ausgabe Java Formatter	3
1.2	Ausgabe TimeLexer richtiger Input	6
1.3	Ausgabe TimeLexer falscher Input	6
2.1	Parse Tree Beispiele	8
2.2	UML-Diagramm	9
3.1	Ausgabe für: "new Class(1, 1000000000000, 1000000000000L)"	15
3.2	Ausgabe für: <i>new FunnyClass(22)</i>	17
4.1	Prozedural	21
4.2	Funktional	21
5.1	Aufrufe für die Fakultät	23
5.2	Append	23
5.3	Aufruf von sum([1, 2, 3], R)	24
5.4	Aufruf von verbindung(konstanz, 8.00, mainz, Reiseplan)	25

6.1	Erzeugte HTML-Seite	29
-----	-------------------------------	----

Quelltextverzeichnis

1.1	Funktion für Java Formatter	2
1.2	Input für Formatter	3
1.3	Lexer für Date/Time	4
1.4	alternativer Lexer	5
2.1	Lexer Grammar	7
2.2	Parser Grammar	8
2.3	Grundlegende AST-Klassen	10
2.4	Builder-Klasse für den AST	11
3.1	ParserListener für statische Syntax	13
3.2	<i>SillyClass</i> als Vorgabe für dynamische Semantikprüfung	15
3.3	ParserListener für dynamische Semantik	15
3.4	Erstellt für Input: <i>new SillyClass(22)</i>	17
4.1	Vorgabe	18
4.2	Prozedurale Methoden	19
4.3	Funktionale Methoden	20
4.4	input.txt	20
5.1	Fakultät	22
5.2	Sum	24
5.3	Verbindung	25
6.1	Info Classes	26
6.2	Main	27
6.3	String Template	27
7.1	JavaScript	30

1. Aufgabe 1

a)

Schreiben Sie ein Java-Programm, das in einem String Formatspezifikationen gemäß *java.util.Formatter* findet.

Erstellen Sie dazu mit der Syntax von *java.util.regex.Pattern* einen regulären Ausdruck für eine solche Formatspezifikation.

Sie brauchen darin nicht zu berücksichtigen, dass bestimmte Angaben innerhalb einer Formatspezifikation nur bei bestimmten Konversionen erlaubt sind. Achten Sie aber bei `argument_index`, `width` und `precision` darauf, ob der Zahlbereich bei 0 oder 1 beginnt.

Beispieleingaben:

```
xxx %d yy%n
xxx%1$d yy
%1$-02.3dyyy
Wochentag: %tA Uhrzeit: %tT
```

Beispielausgaben:

```
TEXT("xxx ")FORMAT("%d")TEXT(" yy")FORMAT("%n")
TEXT("xxx")FORMAT("%1$d")TEXT(" yy")
FORMAT("%1$-02.3d")TEXT("yyy")
TEXT("Wochentag:")FORMAT("%tA")TEXT("Uhrzeit:")FORMAT("%tT")
```

a - Lösung

Code

```
1 private static String formatter(String input) {
2     Pattern patternGeneral =
3         Pattern.compile(
4
5             "(%([1-9]+\\$)?[--#0,(\\s]?\\d*(\\.\\d)?[bBhHsScCdoXxEfGgAa%n])"
6
7             );
8     Pattern patternDate =
9         Pattern.compile(
10
11             "(%([1-9]+\\$)?[--#0,(\\s]?\\d*[tT][HIkLMSPqZzsBbhAaCYyjmdeRTrDFc])"
12
13             );
14     Pattern patternLeftover =
15         Pattern.compile(
16
17             "(%([1-9]+\\$)?[--#0,(\\s]?\\d*\\D)"
18
19             );
20     Pattern usePattern = Pattern.compile(
21         patternGeneral.pattern()
22         + "|" + patternDate.pattern()
23         + "|" + patternLeftover.pattern()
24     );
25
26     var builder = new StringBuilder();
27
28     Map<String, String> parts = new
29     TreeMap<>(Comparator.comparing(input::indexOf));
30
31     Arrays.stream(input.split(usePattern.toString()))
32         .forEach(x -> parts.put(x, "TEXT(\"" + x + "\")"));
33
34     usePattern.matcher(input).results()
35         .forEach(x -> parts.put(x.group(), "FORMAT(\"" + x.group() +
36         "\")"));
37
38     parts.forEach((x, y) -> builder.append(y));
39
40     return builder.toString();
41 }
```

Quelltext 1.1: Funktion für Java Formatter

Erklärung

Das Pattern ist realisiert anhand der [Java Formatter Docs](#).

Dazu unterteilen wir in 3 Unterscheidungen:

- generelle Formate

- Datum
- Rest/Unspezifiziert

Was sich alle Formate teilen, ist das beginnende '%'. Formate erlauben einen optionalen Format-Index anzugeben: '[1-9]+\\$', sowie verschiedene Formattierungs-Optionen: '[-+#0,(]?' und eine optionale Angabe für die minimale Anzahl an Charakteren: '\d*'. Zusammenhängend ergibt das: '%([1-9]+\\$)?[-+#0,(\s)?\d*'.

Bei den generellen Formaten gibt es die zusätzliche Option für die Anzahl Nachkommastellen: '(\.\d)?' gefolgt von den möglichen Konversionen: '[bBhHsScCdoXxEfgGaA%n]'. Für ein Datum fehlt noch '[tT]' und Konversionen: '[HIk1LMSpQZzsBbhAaCYyjmdeRTrDFc]'. Alle restlichen Konversionen sind illegal und reserviert für zukünftige Erweiterungen. Diese decken wir mit '\D' ab.

Anschließend wird der String in einzelne Teile zerlegt, die dann in einer Map gespeichert werden. Sortiert wird anhand der Position des Keys im Input-String. Die fertige Map wird dann in einen String umgewandelt.

Programmausgabe

```
1 public static void main(String[] args) {
2     System.out.println(formatter("xxx %d yyy%n"));
3     System.out.println(formatter("xxx%1$d yyy"));
4     System.out.println(formatter("%1$-02.3dyyy"));
5     System.out.println(formatter("Wochentag: %tA Uhrzeit: %tT"));
6 }
```

Quelltext 1.2: Input für Formatter

```
PS C:\Users\Admin\Desktop\Sprachkonzepte\src\u1> java .\Format.java
TEXT("xxx ")FORMAT("%d")TEXT(" yyy")FORMAT("%n")
TEXT("xxx")FORMAT("%1$d")TEXT(" yyy")
FORMAT("%1$-02.3d")TEXT("yyy")
TEXT("Wochentag: ")FORMAT("%tA")TEXT(" Uhrzeit: ")FORMAT("%tT")
```

Abbildung 1.1: Ausgabe Java Formatter

b)

Erkennen Sie mit ANTLR 4 Lexer-Regeln Zeitangaben im digitalen 12-Sunden-Format gemäß https://en.wikipedia.org/wiki/12-hour_clock. Beachten Sie auch die alternativen Schreibweisen 12 midnight und 12 noon. Testen Sie mit *org.antlr.v4.gui.TestRig*.

b - Lösung

Code

```
1  lexer grammar TimeLexer;  
2  
3  Time12H: Default|Noon|Midnight;  
4  
5  fragment Default: ('12:00'|(( [1-9]|'1'[01])'':[0-5][0-9]))WS[ap]'.m.';  
6  fragment Noon: 'Noon'|'12 noon';  
7  fragment Midnight: 'Midnight'|'12 midnight';  
8  
9  WS: [ \t\r\n]+ -> skip;
```

Quelltext 1.3: Lexer für Date/Time

Erklärung

Lexer Grammatiken beschreiben die Token, die vom Lexer erkannt werden sollen.

Fragmente sind Teile der Grammatik, die nicht direkt erkannt werden, sondern nur in anderen Regeln verwendet werden.

Der Ansatz hier war die Zeitangaben in drei Teile zu zerlegen:

- Default: Volle Uhrzeitangaben im klassischen 'HH:MM' Format mit AM/PM Angabe
- Noon: Zusätzlich die Mittagszeit '12 noon' und 'Noon'
- Midnight: Synchron dazu Mitternacht '12 midnight' und 'Midnight'

Noon und Midnight sind hierbei die Ausnahme, aber vorgegeben durch die Aufgabenstellung.

Alternativ könnte man mehr Token beschreiben:

```
1  lexer grammar TimeLexerV2;
2
3  TIME : HOUR SEPERATOR MINUTE (AM | PM)
4  | TWELVE SEPERATOR '00' (AM | PM)
5  | TWELVE 'noon'
6  | TWELVE 'midnight'
7  | 'Noon'
8  | 'Midnight';
9
10 TWELVE : '12';
11
12 HOUR : '1'[0-1]|[0-9];
13 MINUTE : [0-5][0-9];
14 SEPERATOR : ':';
15
16 AM : 'a.m.' ;
17 PM : 'p.m.' ;
18
19 WS: [ \t\r\n]+ -> skip;
```

Quelltext 1.4: alternativer Lexer

Allerdings wird die Lexer Grammatik hier etwas missbraucht, da die 'TIME' Regel eher als Parser Regel genutzt wird. Lexer Regeln sollten eigentlich nur die Token beschreiben, die vom Lexer erkannt werden sollen.

Programmausgabe

Korreakter Input:

12:00 AM

12:00 a.m.

12:00 am

3:00 am

1:12 am

Noon

12 noon

Midnight

11:59 PM


```
C:\PROG_uebungen\Sprachkonzepte\src\ui>java -cp ".;C:\Program Files\Java\lib\antlr-4.11.1-complete.jar;" org.antlr.v4.gui.TestRig TimeLexer tokens -tokens RichtigeBeispiele.txt
[0,0:7='12:00 AM',<Time12H>,1:0]
[01,10:19='12:00 a.m.',<Time12H>,2:0]
[02,22:29='12:00 am',<Time12H>,3:0]
[03,32:38='3:00 am',<Time12H>,4:0]
[04,41:47='1:12 am',<Time12H>,5:0]
[05,52:55='Noon',<Time12H>,7:0]
[06,58:64='12 noon',<Time12H>,8:0]
[07,67:74='Midnight',<Time12H>,9:0]
[08,77:84='11:59 PM',<Time12H>,10:0]
[09,85:84='<EOF>',<EOF>,10:8]
```

Abbildung 1.2: Ausgabe TimeLexer richtiger Input

Falscher Input:

0:00 am
0:00 pm
12:00
noon
12 Noon
13:00 PM

```
C:\PROG_uebungen\Sprachkonzepte\src\ui>java -cp ".;C:\Program Files\Java\lib\antlr-4.11.1-complete.jar;" org.antlr.v4.gui.TestRig TimeLexer tokens -tokens FalscheBeispiele.txt
line 1:0 token recognition error at: '0'
line 1:1 token recognition error at: ':'
line 1:2 token recognition error at: '0'
line 1:3 token recognition error at: '0'
line 1:5 token recognition error at: 'a'
line 1:6 token recognition error at: 'm'
line 2:0 token recognition error at: '0'
line 2:1 token recognition error at: ':'
line 2:2 token recognition error at: '0'
line 2:3 token recognition error at: '0'
line 2:5 token recognition error at: 'p'
line 2:6 token recognition error at: 'm'
line 3:0 token recognition error at: '12:00\r\n'
line 4:1 token recognition error at: 'o'
line 4:2 token recognition error at: 'o'
line 4:3 token recognition error at: 'n'
line 5:0 token recognition error at: '12 N'
line 5:4 token recognition error at: 'o'
line 5:5 token recognition error at: 'o'
line 5:6 token recognition error at: 'n'
line 6:0 token recognition error at: '13'
line 6:2 token recognition error at: ':'
line 6:3 token recognition error at: '0'
line 6:4 token recognition error at: '0'
line 6:6 token recognition error at: 'P'
line 6:7 token recognition error at: 'M'
[0,48:47='<EOF>',<EOF>,6:8]
```

Abbildung 1.3: Ausgabe TimeLexer falscher Input

2. Aufgabe 2

a)

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 Beispieltex te mit Hilfe von *org.antlr.v4.gui.TestRig* den Ableitungsbaum (Parse Tree).

a - Lösung

Dargestellt ist der Lexer einer Sprache, welche die korrekte Kreation einer Java Klasse darstellt. Erlaubt sind Variablen - also einzelne Buchstaben, sowie Zahlen. Parameter werden durch Komma getrennt und dürfen auch eigene, neu erzeugte Klassen sein.

```
1 // CreationLexer.g4
2 lexer grammar CreationLexer;
3
4 KEYWORD : 'new' ;
5
6 NAME : [A-Za-z]+ ;
7 NUM : [0-9]+ ;
8
9 COMMA : ',' ;
10
11 PAR_OPEN : '(' ;
12 PAR_CLOSE : ')' ;
13
14 WS : [ \t\r\n]+ ;
15
16 InvalidChar: . ;
```

Quelltext 2.1: Lexer Grammar

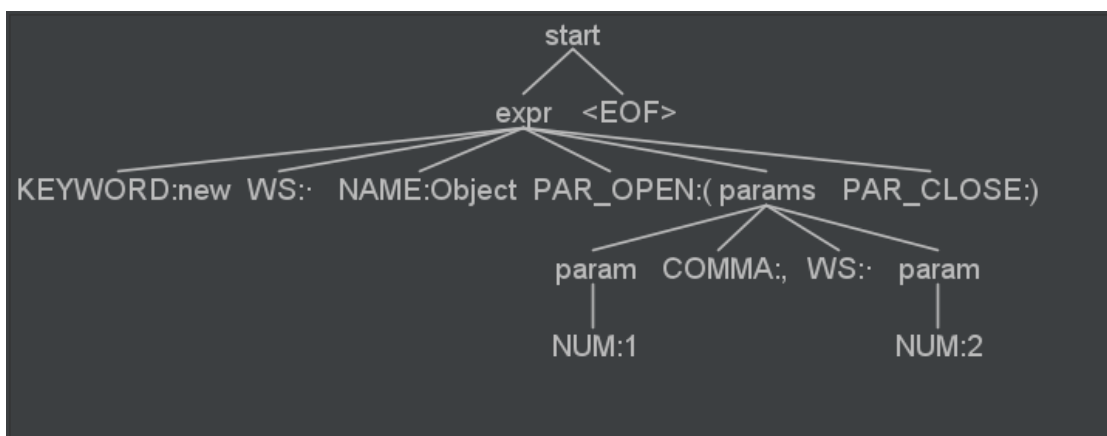
Der dazugehörige Parser:

```

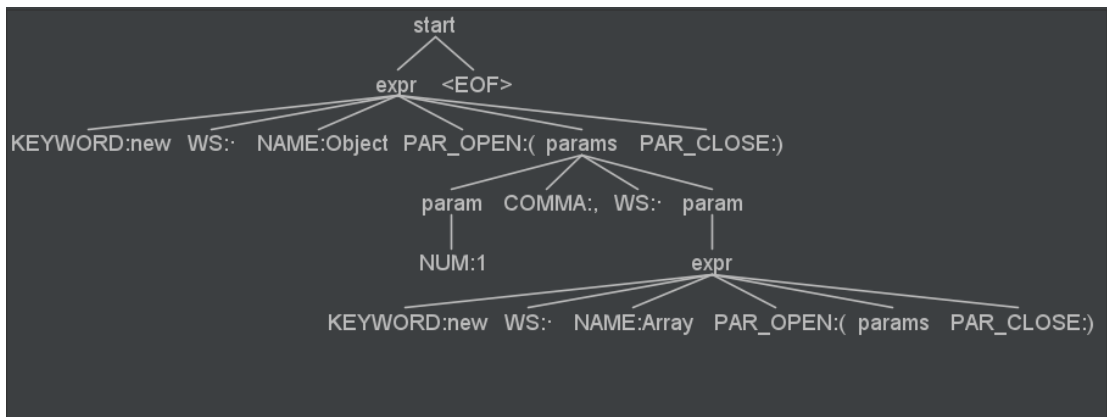
1  // CreationParser.g4
2  parser grammar CreationParser;
3  options { tokenVocab=CreationLexer; }
4
5  start : expr EOF;
6
7  expr : KEYWORD WS NAME PAR_OPEN params PAR_CLOSE ;
8
9  params : (param (COMMA WS? param)*)? ;
10
11 param : (expr | NAME | NUM) ;

```

Quelltext 2.2: Parser Grammar



(a) Input: 'java org.antlr.v4.gui.TestRig Creation start -gui new Object(1, 2)'



(b) Input: 'java org.antlr.v4.gui.TestRig Creation start -gui new Object(1, new Array())'

Abbildung 2.1: Parse Tree Beispiele

Der Parser ist für die grammatikalische Anordnung der durch den Lexer vorgegebenen Token verantwortlich.

b)

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.

b - Lösung

Definition als UML-Diagramm:

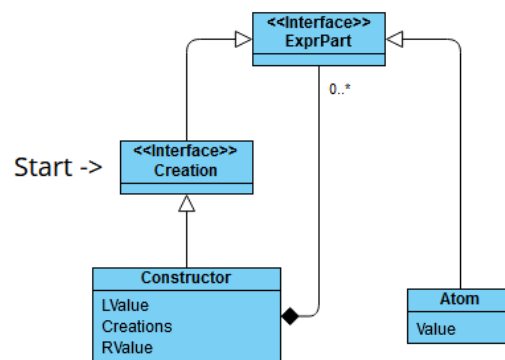


Abbildung 2.2: UML-Diagramm

Eine Baumstufe kann in drei Teile geteilt werden:

Links Der Keyword-, Namen- und Klammer-Teil vor der Mitte.

Mitte Rekursive Parameter-Definition - kann auch leer oder eine neue *Creation* sein.

Rechts In diesem Fall die Klammer zum Schließen der Parameter.

Dadurch ergibt sich die Vorgabe von mindestens einer *Constructor* Instanz. *L-* und *RValue* sind definierbare Strings und *Creations* ist die Liste der Parameter innerhalb der Klammern/Strings, dargestellt in der Relation. So können neue *Constructor* Objekte, sowie Werte definiert in der Klasse *Atom* als Parameter übergeben werden.

Um die Struktur beizubehalten, werden beispielhaft also folgende Klassen erstellt:

```
1 public interface Expr {
2 }
3
4 public interface Creation extends Expr {
5 }
6
7 public class Atom implements Expr {
8     private final String val;
9
10    public Atom(String val) {
11        this.val = val;
12    }
13    public String getVal() {
14        return val;
15    }
16    @Override
17    public String toString() {
18        return this.val;
19    }
20 }
21
22 public class Constructor implements Creation {
23     private final String leftVal;
24     private final List<Expr> params;
25     private final String rightVal;
26
27    public Constructor(String leftVal, List<Expr> params, String
rightVal) {
28        this.leftVal = leftVal;
29        this.params = params;
30        this.rightVal = rightVal;
31    }
32    public String getLeftVal() {
33        return leftVal;
34    }
35    public List<Expr> getParams() {
36        return params;
37    }
38    public String getRightVal() {
39        return rightVal;
40    }
41    @Override
42    public String toString() {
43        return this.leftVal + this.params + this.rightVal;
44    }
45 }
```

Quelltext 2.3: Grundlegende AST-Klassen

Um diese Struktur in einem Builder umzusetzen, muss ein eigener Stack für jedes neue Constructor Objekt angelegt werden.


```
51     }  
52 }  
53  
54 }
```

Quelltext 2.4: Builder-Klasse für den AST

Für jedes angefangene *Constructor* Objekt wird ein neuer Stack angelegt in der *enterExpr* Methode. Zusätzlich wird die Tiefe erhöht, um jeweils beim aktuell noch nicht abgeschlossenen *Constructor* Objekt zu bleiben. Dadurch bleibt die Reihenfolge erhalten und innerhalb eines noch nicht abgeschlossenen *Constructor* Objekts können *Atom* Objekte und weitere *Constructor* Objekte erstellt werden, ohne dass Konstruktoren die auf dem Stack liegenden Objekte in sich aufnehmen.

Die *exitExpr* Methode arbeitet den aktuellen Stack ab und setzt ihn als Parameter-Liste für das aktuelle *Constructor* Objekt. Danach wird der aktuelle Stack geleert und die Tiefe verringert. Abschließend wird der aktuelle Konstruktor auf den Stack gelegt für den nächsten Konstruktor oder den finalen Rückgabewert.

Die *enterParam* Methode ist der unterste Baustein des Baums, und dient zur Erstellung von *Atom* Objekten, die dann auf den Stack gelegt werden für Konstruktoren. Wenn die *Param* Regel wieder eine *Expr* Regel enthält, wird sie in der Methode ignoriert, da diese bereits in der *exitExpr* Methode abgearbeitet wird.

3. 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

```
1 public class CreationStatic extends CreationParserBaseListener {
2
3     public static void main(String[] args) throws IOException {
4         CreationLexer lexer = new CreationLexer(args.length >= 1 ?
5             CharStreams.fromString(args[0]) :
6             CharStreams.fromStream(System.in));
7         CreationParser parser = new CreationParser(new
8             CommonTokenStream(lexer));
9         ParseTree tree = parser.parse();
10
11         if (parser.getNumberOfSyntaxErrors() > 0) {
12             System.err.printf("%d error(s) detected%n",
13                 parser.getNumberOfSyntaxErrors());
14             System.exit(1);
15         }
16
17         new ParseTreeWalker().walk(new CreationStatic(), tree);
18     }
19
20     @Override
21     public void enterParam(CreationParser.ParamContext ctx) {
22
23         if (ctx.start.getType() == CreationParser.NUM) {
24             System.out.println("Found a Number: " + ctx.getText());
25
26             var literal = ctx.NUM().getText();
27
28             if (literal.endsWith("L") || literal.endsWith("l")) {
29                 System.out.print("Expected Long: ");
30             }
31         }
32     }
33 }
```



```

27         long value;
28         try {
29             value = Long.parseLong(literal.substring(0,
literal.length() - 1));
30         } catch (NumberFormatException e) {
31             System.out.println("Bigger than Long!");
32             throw new RuntimeException(e);
33         }
34         if (value > Integer.MAX_VALUE) {
35             System.out.println("Found Long!");
36         } else {
37             System.out.println("Wrong range!");
38         }
39     } else {
40         System.out.print("Expected Integer: ");
41
42         try {
43             Integer.parseInt(literal);
44             System.out.println("Found Integer!");
45         } catch (NumberFormatException e) {
46             System.out.println("Wrong range!");
47         }
48     }
49     System.out.println();
50 }
51 }
52 }
53 }

```

Quelltext 3.1: ParserListener für statische Syntax

Umsetzung der statischen Semantikprüfung mit der ANTLR4 Java Grammatik.

Beim Parsen wird jeder Paramter mit der *enterParam* Methode des Listeners überprüft, ob dieser ein im Falle einer Zahl, ein Integer oder Long ist. Dies geschieht indem aus dem Kontext der Regeltyp hergeleitet wird. Sollte der Typ *NUM* sein, wird überprüft ob der Literalwert mit einem *L* oder *l* für *Long* endet. Falls dies der Fall ist, wird der Wert in einen Long geparst und überprüft ob dieser größer als der Maximalwert von Integer ist. Somit ist sichergestellt, dass der Wert im Bereich von Long liegt. Falls dies nicht der Fall ist, wird der Wert in einen Integer geparst und somit indirekt überprüft ob dieser im Bereich von Integer liegt. Das Parsen ist jeweils in einem Try-Catch Block, da bei einem zu großen Wert eine *NumberFormatException* geworfen wird. Dadurch ist gegeben, dass die Wertebereiche stimmen.

```

Found a Number: 1
Expected Integer: Found Integer

Found a Number: 1000000000000
Expected Integer: Wrong range!

Found a Number: 1000000000000L
Expected Long: Found Long

```

Abbildung 3.1: Ausgabe für: "new Class(1, 1000000000000, 1000000000000L)"

b)

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

b - Lösung

```

1 public class SillyClass {
2     int x = 0;
3
4     public SillyClass(int x) {
5         this.x = x;
6     }
7 }

```

Quelltext 3.2: *SillyClass* als Vorgabe für dynamische Semantikprüfung

In der Aufgabe wird für die vorgegebene *SillyClass* eine dynamische Semantikprüfung implementiert.

Code

```

1 public class CreationDynamicAnalyzer extends CreationParserBaseListener {
2
3     static String pre = ""
4         package u3;
5
6     public class CreationDynamic {
7         public static void main(String[] args) {
8             System.out.println(
9                 "";

```

```

10     static String post = ""
11
12         );
13     }
14     }""";
15
16     StringBuilder sb = new StringBuilder(pre).append("\t\t\t");
17
18     public static void main(String[] args) throws IOException {
19         CreationLexer lexer = new CreationLexer(args.length >= 1 ?
20             CharStreams.fromString(args[0]) :
21             CharStreams.fromStream(System.in));
22         CreationParser parser = new CreationParser(new
23             CommonTokenStream(lexer));
24         ParseTree tree = parser.start();
25
26         if (parser.getNumberOfSyntaxErrors() > 0) {
27             System.err.printf("%d error(s) detected%n",
28                 parser.getNumberOfSyntaxErrors());
29             System.exit(1);
30         }
31
32         new ParseTreeWalker().walk(new CreationDynamicAnalyzer(), tree);
33
34     }
35
36     @Override
37     public void enterExpr(CreationParser.ExprContext ctx) {
38         if (ctx.getChildCount() == 6) {
39             if
40             (!ctx.getChild(2).getText().equals(SillyClass.class.getSimpleName()))
41             {
42                 System.out.println(ctx.getChild(2).getText() + " == " +
43                     SillyClass.class.getSimpleName());
44                 throw new RuntimeException("Wrong class name!");
45             }
46             for (int i = 0; i < ctx.getChildCount(); i++) {
47                 sb.append(ctx.getChild(i).getText());
48             }
49             sb.append(post);
50
51             if (ctx.params().param().size() != 1) {
52                 throw new RuntimeException("Wrong number of
53                 parameters!");
54             }
55
56             try {
57                 FileWriter myWriter = new
58                 FileWriter("./src/u3/CreationDynamic.java");
59                 myWriter.write(sb.toString());

```

```

54         myWriter.close();
55         System.out.println("Successfully wrote to the file.");
56     } catch (IOException e) {
57         System.out.println("An error occurred.");
58         e.printStackTrace();
59     }
60 }
61 }
62 }

```

Quelltext 3.3: ParserListener für dynamische Semantik

In der *enterExpr* Methode wird dynamisch überprüft ob die Klasse, die erstellt werden soll, die richtige ist. Dabei stellen wir zuerst sicher, dass die Anzahl der Parameter stimmt, welche bei der Klasse *SillyClass* nur ein Parameter ist und somit die Anzahl an Kindern des Kontextes 6 ist.

Siehe Parser: *expr* : *KEYWORD WS NAME PAR_OPEN params PAR_CLOSE* ;

Danach wird abgefangen, wenn der Klassenname nicht der Klasse *SillyClass* entspricht. Im selben Stil wird abgefragt ob die Anzahl der Parameter stimmt. Erst wenn diese Bedingungen erfüllt sind, wird der Code in eine Datei geschrieben. Dabei sind *pre* und *post* Strings, welche den Code vor und nach dem erstellen eines *SillyClass* Objektes enthalten.

Entstandener Code

```

1 public class CreationDynamic {
2     public static void main(String[] args) {
3         System.out.println(
4             new SillyClass(22)
5         );
6     }
7 }

```

Quelltext 3.4: Erstellt für Input: *new SillyClass(22)*

Ausgabe bei falscher Klasse

```

new FunnyClass(22)
^D
line 1:18 extraneous input '\n' expecting <EOF>
1 error(s) detected
Exception in thread "main" java.lang.RuntimeException Create breakpoint : Wrong class name!
    at u3.CreationDynamicAnalyzer.enterExpr(CreationDynamicAnalyzer.java:53)
    at u2.CreationParser$ExprContext.enterRule(CreationParser.java:358)
    at org.antlr.v4.runtime.tree.ParseTreeWalker.enterRule(ParseTreeWalker.java:50)
    at org.antlr.v4.runtime.tree.ParseTreeWalker.walk(ParseTreeWalker.java:33)
    at org.antlr.v4.runtime.tree.ParseTreeWalker.walk(ParseTreeWalker.java:36)
    at u3.CreationDynamicAnalyzer.main(CreationDynamicAnalyzer.java:42)
FunnyClass == SillyClass

```

Abbildung 3.2: Ausgabe für: *new FunnyClass(22)*

4. Aufgabe 4

a)

Vervollständigen Sie das folgende Java-Programm, indem Sie die aufgerufenen Klassenmethoden ergänzen. Was an dem Java-Programm ist eindeutig prozeduraler Stil?

```
1  import java.nio.file.Files;
2  import java.nio.file.Paths;
3  import java.io.BufferedReader;
4  import java.io.IOException;
5
6  import java.util.LinkedList;
7  import java.util.List;
8
9  public final class Procedural {
10     private Procedural() { }
11
12     private static final int MIN_LENGTH = 20;
13
14     public static void main(String[] args) throws IOException {
15         var input = Files.newBufferedReader(Paths.get(args[0]));
16         var lines = new LinkedList<String>();
17
18         long start = System.nanoTime();
19
20         readLines(input, lines);
21         removeEmptyLines(lines);
22         removeShortLines(lines);
23         int n = totalLineLengths(lines);
24
25         long stop = System.nanoTime();
26
27         System.out.printf("result = %d (%d microsec)%n", n, (stop -
28         start) / 1000);
29     }
30
31     private static void readLines(BufferedReader input, List<String>
32     lines) throws IOException {
33         String line;
34         while ((line = input.readLine()) != null) {
35             lines.add(line);
36         }
37     }
38 }
```

```

37 // TODO: Klassenmethoden readLines, removeEmptyLines,
    removeShortLines, totalLineLengths
38 }

```

Quelltext 4.1: Vorgabe

a - Lösung

```

1  private static void removeEmptyLines(List<String> lines) {
2      for (var it = lines.iterator(); it.hasNext(); ) {
3          if (it.next().isEmpty()) {
4              it.remove();
5          }
6      }
7  }
8
9  private static void removeShortLines(List<String> lines) {
10     for (var it = lines.iterator(); it.hasNext(); ) {
11         if (it.next().length() < MIN_LENGTH) {
12             it.remove();
13         }
14     }
15 }
16
17 private static int totalLineLengths(List<String> lines) {
18     int n = 0;
19     for (var it = lines.iterator(); it.hasNext(); ) {
20         n += it.next().length();
21     }
22     return n;
23 }

```

Quelltext 4.2: Prozedurale Methoden

Die Aufrufe der Klassenmethoden sind im Prozeduralen Stil. Typisch dafür ist die beschreibende Programmierung, bei der die Methoden in der Reihenfolge aufgerufen werden, wie man es auch beschreiben würde. Methoden geben dann typischerweise keinen Wert zurück, sondern verändern den Zustand der Objekte, die sie als Parameter erhalten.

b)

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

b - Lösung

```

1 public final class ProceduralFunctional {
2     private ProceduralFunctional() { }
3
4     private static final int MIN_LENGTH = 20;
5
6     public static void main(String[] args) throws IOException {
7         var input = Files.newBufferedReader(Paths.get(args[0]));
8
9         long start = System.nanoTime();
10
11         int n = input.lines()
12             .filter(s -> !s.isEmpty())
13             .filter(s -> s.length() >= MIN_LENGTH)
14             .mapToInt(String::length)
15             .sum();
16
17         long stop = System.nanoTime();
18
19         System.out.printf("result = %d (%d microsec)%n", n, (stop -
20             start) / 1000);
21     }
22 }

```

Quelltext 4.3: Funktionale Methoden

Die vorherigen Klassenmethoden wurden durch funktionalen Stil ersetzt. Zu erst wird ein Stream aus den Zeilen der Datei erstellt, auf welchen dann die Filteroperationen angewendet werden. Die Filteroperationen sind in der Reihenfolge aufgelistet, wie sie auch in der Beschreibung des Programms stehen - Könnten aber auch zusammengefasst werden. Die Filteroperationen geben einen neuen Stream zurück, auf welchem dann die nächste Operation angewendet wird. Gefilterte Zeilen werden auf die Länge gemappt und anschließend wird die Gesamtsumme gebildet, so wie es auch der prozedurale Stil getan hat.

c)

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

c - Lösung

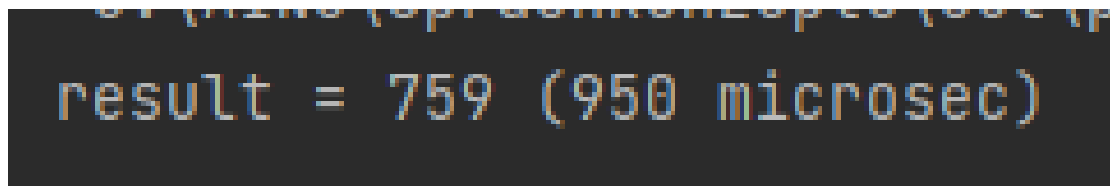
```

1 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
2 Sed et metus aliquam, hendrerit metus vel, rhoncus dolor.
3 Nulla facilisi. Nullam a sapien libero.
4 Morbi in erat sit amet enim sagittis semper.
5 Fusce at leo vitae ligula malesuada luctus.
6 Curabitur sed lectus in nisi vulputate sodales. Nulla facilisi.
7 Morbi auctor, dolor nec ultricies tincidunt, justo felis pharetra
   mauris, a lacinia quam arcu et nisl.

```

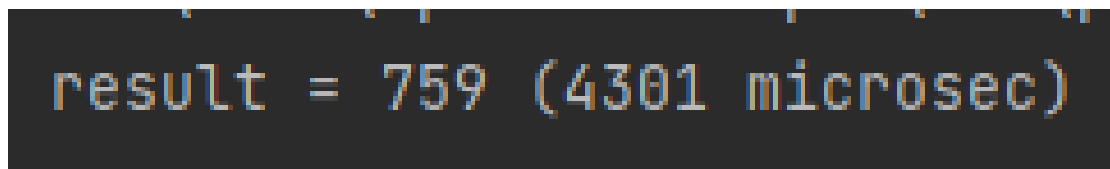
```
8 Suspendisse potenti. Morbi euismod augue lacus, vitae faucibus libero
  bibendum non.
9 Nullam nec augue ut mi cursus hendrerit. In hac habitasse platea
  dictumst.
10
11 Nullam ut neque ligula. Aliquam erat volutpat. Nulla facilisi.
12 Duis nec urna sed lacus vestibulum viverra.
13 Donec nec ante sit amet erat ultrices iaculis.
14 Sed id tortor quis turpis pellentesque egestas.
15 Pellentesque.
```

Quelltext 4.4: input.txt



```
result = 759 (950 microsec)
```

Abbildung 4.1: Prozedural



```
result = 759 (4301 microsec)
```

Abbildung 4.2: Funktional

Wie erwartet ist der funktionale Stil deutlich langsamer als der prozedurale Stil. Dies ist zurückzuführen auf die Tatsache, dass der funktionale Stil mit *Streams*, intern mit Rekursion arbeitet. Im prozeduralen Stil werden die Operationen in einfachen Schleifen ausgeführt, wodurch die Laufzeit kurz bleibt.

5. Aufgabe 5

a)

Lösen Sie die Aufgaben von Folie 25 (rechte Spalte der Tabelle), 26 (Berechnung Fakultät) und 28 (Anfragen letzter Spiegelpunkt) aus Eck-Prolog.pdf.

a - 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,horser]	[horse,X]	false
[white Q]	[P,horse]	P = white, Q = [horse]

Tabelle 5.1: Tabelle

```

1 fak(0, 1).
2 fak(N, F):-
3     N > 0,
4     N1 is N - 1,
5     fak(N1, F1),
6     F is N * F1.
```

Quelltext 5.1: Fakultät

Die Exit-Strategie der Fakultät ist in der Regel *fak(0, 1)*. definiert. Das Prädikat *fak(N, F)* ist die Rekursion, die die Fakultät der nächst kleineren Zahl berechnet und mit der aktuellen Zahl multipliziert.

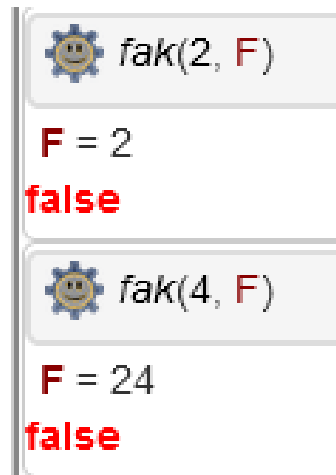


Abbildung 5.1: Aufrufe für die Fakultät

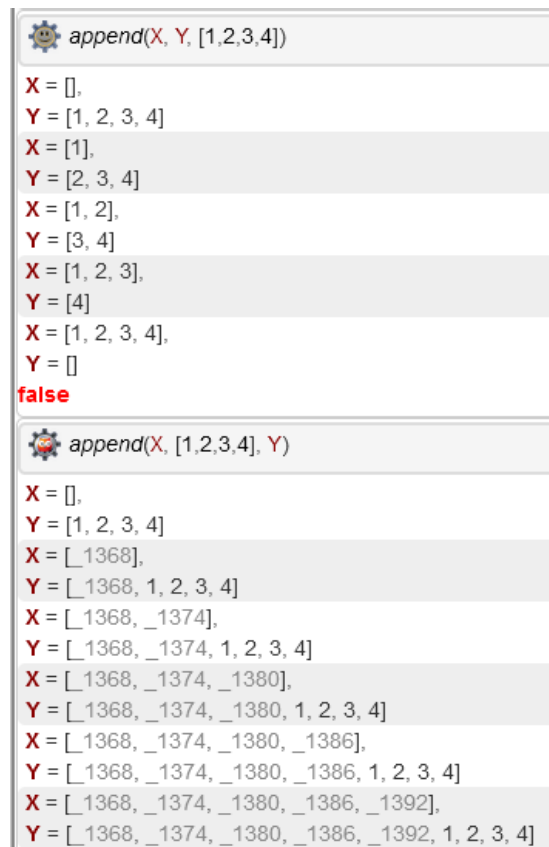


Abbildung 5.2: Append

Die erste *append* Anfrage liefert alle möglichen Kombinationen von *X* und *Y* zurück, welche zusammen die Liste *[1, 2, 3, 4]* ergeben. Wichtig ist dabei, dass die Reihenfolge erhalten bleibt. Somit ergeben sich 5 mögliche Kombinationen.

Die zweite *append* Anfrage liefert alle möglichen Kombinationen von *X* und der Liste *[1, 2, 3, 4]* zurück, welche zusammen die Liste *Y* ergeben. Somit also unendlich Kombinationen, an denen mindestens die Liste *[1, 2, 3, 4]* anhängt.

b)

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

b - Lösung

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

Quelltext 5.2: Sum



sum([1, 2, 3], R)	
R	
6	1

Abbildung 5.3: Aufruf von **sum([1, 2, 3], R)**

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, 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).
```

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.*

Ein Abfrage **verbindung(konstanz, 8.00, mainz, Reiseplan)** soll nacheinander die möglichen Reiseverbindungen nach 8 Uhr in der Listenvariablen **Reiseplan** liefern.

c - Lösung

```
1 zug(konstanz, 08.39, offenburg, 10.59).
2 zug(konstanz, 08.39, karlsruhe, 11.49).
3 zug(konstanz, 08.53, singen, 09.26).
4 zug(singen, 09.37, stuttgart, 11.32).
5 zug(offenburg, 11.29, mannheim, 12.24).
6 zug(karlsruhe, 12.06, mainz, 13.47).
7 zug(stuttgart, 11.51, mannheim, 12.28).
8 zug(mannheim, 12.39, mainz, 13.18).
9
10 verbindung(Start,Uhr,Ziel,Res) :-
11     zug(Start,SUhr,Ziel,EndZeit),
12     SUhr >= Uhr,
13     append([Start,SUhr,Ziel,EndZeit],[],Res)
14     ;
15     zug(Start,SUhr,Neu,EndZeit),
16     SUhr >= Uhr,
17     verbindung(Neu,EndZeit,Ziel,L),
18     append([Start,SUhr,Neu,EndZeit],L,Res).
```

Quelltext 5.3: Verbindung

Das *Semikolon (;)* teilt das Prädikat in zwei Fälle auf. Der erste stellt dabei unsere Exit-Strategie der Rekursion dar. Hier wird, wenn eine Zugverbindung von *Start* nach *Ziel* existiert, die Liste *Res* mit den entsprechenden Daten befüllt.

Wenn keine direkte Verbindung existiert, wird der zweite Fall aufgerufen. Dabei wird erst die Grundbedingung geprüft, ob ein Zug *Neu* existiert, für den eine Verbindung existiert, bei der die Abfahrtszeit größer oder gleich der gegebenen Uhrzeit ist. Anschließend wird rekursiv geprüft, ob eine Folgeverbindung existiert, welche dann ihre Teilstrecke in eine Liste *L* schreibt. So sammeln sich die Teillisten rekursiv in der Gesamtliste *Res* zusammen.



Reiseplan	
[konstanz, 8.39, offenburg, 10.59, offenburg, 11.29, mannheim, 12.24, mannheim, 12.39, mainz, 13.18]	1
[konstanz, 8.39, karlsruhe, 11.49, karlsruhe, 12.06, mainz, 13.47]	2
[konstanz, 8.53, singen, 9.26, singen, 9.37, stuttgart, 11.32, stuttgart, 11.51, mannheim, 12.28, mannheim, 12.39, mainz, 13.18]	3

Abbildung 5.4: Aufruf von **verbindung(konstanz, 8.00, mainz, Reiseplan)**

6. Aufgabe 6

Aufgabe

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

Da aus StringTemplate heraus keine statischen Funktionen aufgerufen werden können, wie etwa `c.getInterfaces()`, werden deren Werte im Voraus in einer Wrapper-Klasse als Instanzvariablen abgelegt:

```
1 final class ClassInfo {
2     public final String name;
3     public final boolean hasNoInterface;
4     public final boolean hasMethods;
5     public final List<String> classMethods;
6     public List<InterfaceInfo> interfaces;
7
8     public ClassInfo(Class<?> c) {
9         this.name = c.getName();
10        this.interfaces =
11        Arrays.stream(c.getInterfaces()).map(InterfaceInfo::new).toList();
12        this.hasNoInterface = interfaces.isEmpty();
13
14        this.classMethods = Arrays.stream(c.getMethods())
15            .map(x -> x.getReturnType() + " " + x.getName() + "(" +
16            Arrays.toString(x.getParameterTypes()) + ")")
17            .filter(o -> this.interfaces.stream().noneMatch(i ->
18            i.methods.contains(o)))
19            .toList();
20        this.hasMethods = !classMethods.isEmpty();
21    }
22 }
```

```

21 final class InterfaceInfo {
22     public final String name;
23     public final List<String> methods;
24
25     public InterfaceInfo(Class<?> i) {
26         this.name = i.getName();
27         this.methods = Arrays.stream(i.getMethods()).map(
28             x -> x.getReturnType() + " " + x.getName()
29                 + "(" + Arrays.toString(x.getParameterTypes())
30                 .replace('[', ' ').replace(']', ' ') + ")"
31         ).toList();
32     }
33 }

```

Quelltext 6.1: Info Classes

In der main Methode, werden Klassen-Objekte mit, über den *args* Parameter übergebene, Klassennamen und *Class.forName* erzeugt und anschließend in unsere Wrapper-Klassen gepackt. Eine Liste dieser Wrapper wird schließlich dem Stringtemplate übergeben und daraus die HTML-Seite generiert.

```

1 public final class HtmlJavaDoc {
2
3     public static void main(String[] args) {
4         ST templ = new
5         STGroupFile("u6/htmljavadoc.stg").getInstanceOf("docpage");
6
7         List<? extends Class<?>> classes = Arrays.stream(args)
8             .map(arg -> {
9                 try {
10                     return Class.forName(arg);
11                 } catch (ClassNotFoundException e) {
12                     throw new RuntimeException(e);
13                 }
14             })
15             .toList();
16         templ.add("p", classes.stream().map(ClassInfo::new).toList());
17
18         String result = templ.render();
19         System.out.println(result);
20     }
21 }

```

Quelltext 6.2: Main

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

```

1 // htmljavadoc.stg
2 delimiters "$", "$"
3
4 docpage(p) ::= <<
5 <!DOCTYPE html>
6 <html lang="en">

```

```

7 <head>
8   <meta charset="UTF-8">
9   <title>Sprachkonzepte Aufgabe 6</title>
10
11   <style>
12     html { background: whitesmoke; }
13     th, td { border-bottom: thin solid; padding: 4px; text-align: left; }
14     td { font-family: monospace }
15   </style>
16 </head>
17 <body>
18 <h1>Sprachkonzepte Aufgabe 6</h1>
19 $p:classdoc(); separator="\n"$
20 </body>
21 </html>
22 >>
23
24 classdoc(l) ::= <<
25 <h2>$l.name$</h2>
26 <table>
27   <tbody>
28     <tr>
29       $if(!l.hasNoInterface)$<th>Interface</th>$endif$
30       <th>Methods</th>
31     </tr>
32     $l.interfaces:rowdoc(); separator="\n"$
33     $if(l.hasNoInterface && l.hasMethods)$<tr><td>$l.classMethods;
34       separator="<br>"$</td></tr>$endif$
35   </tbody>
36 </table>
37 >>
38 rowdoc(c) ::= <<
39 <tr>
40   <td valign="top">
41     $c.name$
42   </td>
43   <td>
44     $c.methods; separator="<br>"$
45   </td>
46 </tr>
47 >>

```

Quelltext 6.3: String Template

Die Einstiegsregel *docpage* nimmt die Liste der Wrapper-Klassen entgegen und ruft für jede Klasse die Regel *classdoc* auf. Dabei wird unterschieden zwischen Klassen, die Interfaces implementieren und Klassen, die keine Interfaces implementieren. Wenn Interfaces implementiert werden, wird für jedes Interface, mit der Regel *rowdoc* eine eigene Zeile in der Tabelle erzeugt. Wenn keine Interfaces implementiert werden, wird eine Zeile erzeugt, die die Methoden der Klasse auflistet.

Sprachkonzepte Aufgabe 6

java.lang.String

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

java.util.Iterator

Methods

```
void remove([])  
void forEachRemaining([interface java.util.function.Consumer])  
boolean hasNext([])  
class java.lang.Object next([])
```

java.time.Month

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

Abbildung 6.1: Erzeugte HTML-Seite

7. Aufgabe 7

Aufgabe

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

```
1 const url = 'https://feiertage-api.de/api/';
2 const params = '?jahr=2022&nur_land=BW';
3 let request = url + params
4
5 function isValidRequest(url) {
6     return
7     !!url.match(/^https?:\/\/\/.+.(de|com|net)\/\w*\/?(\?\w+=.+(&\D+=.+)*)?$/);
8 }
9
10 if (isValidRequest(request)) {
11     const response = fetch(request)
12     response
13     .then(r => r.json())
14     .then(data => {
15         console.log(data)
16         console.log(data['Karfreitag'])
17         request = data;
18         console.log(request);
19     });
20 }
```

Quelltext 7.1: JavaScript

Das Skript fragt die Feiertage für das Jahr 2022 in Baden-Württemberg ab. Dabei sind einige Eigenschaften einer Skriptsprache zu erkennen. Zum einen werden Konstanten und Variablen, Datentypen automatisch zugewiesen. So sind die Variablen *url* und *params* vom Typ *String* - *request* zunächst auch. Mit der Funktion *isValidRequest* wird überprüft, ob die URL gültig ist. Dabei wird die URL mit einem regulären Ausdruck überprüft. Die Methode *match* gibt ein Array zurück, wenn der reguläre Ausdruck mit der URL übereinstimmt - ansonsten *null*. Mit dem Doppel-Not-Operator (*!!*) wird das Ergebnis auf *true* oder *false* umgewandelt.

Hier kommt die Eigenschaft der dynamischen Typisierung zum Tragen. Sollte die URL gültig sein, wird eine Anfrage an die URL, mit der Methode *fetch*, gesendet. Die Antwort wird in der Variable *response* gespeichert. Auch hier wird die dynamische Typisierung genutzt. Die Antwort ist vom Typ *Promise*. Mit der Methode *then* wird auf den *Promise* gewartet und die Antwort in der Lambda-Variablen *r* gespeichert. Das empfangene JSON-Objekt wird in die Variable *data* übergeben und auf die Konsole ausgegeben. Um auf die einzelnen Elemente des JSON-Objekts zuzugreifen, wird der Schlüssel verwendet. In diesem Fall ist der Schlüssel der Name des Feiertags. Hier wird also der String *Karfreitag* als Array-Index verwendet. Abschließend wird die Variable *request* mit dem JSON-Objekt überschrieben und auf der Konsole ausgegeben. Somit hat sich der Datentyp von *String* implizit auf *Object* geändert.