AIN - Sprachkonzepte

Oskar Borkenhagen

WS2022/23

Bericht zu den Übungsaufgaben der Vorlesung Sprachkonzepte - AIN

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 argment_index, width und precision darauf, ob der Zahlbereich bei 0 oder 1 beginnt.

Beispieleingaben:
xxx %d yyy%n
xxx%1$d yyy
%1$-02.3dyyy
Wochentag: %tA Uhrzeit: %tT

Beispielausgaben:
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")

a - Lösung

```java
private static String formatter(String input) {
    Pattern patternGeneral =
            Pattern.compile(
                    "(%([1-9]\\$)?[-+#0,(\s]?\\d*(\\.\\d)?[bBhHsScCdoxXeEfgGaA%n])"
            );
    Pattern patternDate =
            Pattern.compile(
                    "(%([1-9]\\$)?[-+#0,(\s]?\\d*[tT][HIklLMSpQZzsBbhAaCYyjmdeRTrDFc])"
            );
    Pattern patternLeftover =
            Pattern.compile(
                    "(%[-+#0,(\s]?\\d*\\D)"
            );
    Pattern usePattern = Pattern.compile(
            patternGeneral.pattern()
                    + "|" + patternDate.pattern()
                    + "|" + patternLeftover.pattern()
    );

    var builder = new StringBuilder();

    Map<String, String> parts = new
        TreeMap<>(Comparator.comparing(input::indexOf));

    Arrays.stream(input.split(usePattern.toString()))
            .forEach(x -> parts.put(x, "TEXT(\"" + x + "\")"));

    usePattern.matcher(input).results()
            .forEach(x -> parts.put(x.group(), "FORMAT(\"" +
                x.group() + "\")"));

    parts.forEach((x, y) -> builder.append(y));

    return builder.toString();
}
```

Pattern realisiert anhand Java Formatter Docs.
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.

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

```
1   // TimeLexer.g4
2   lexer grammar TimeLexer;
3
4   Time12H: Default|Noon|Midnight;
5
6   fragment Default:
        ('12:00'|(([1-9]|'1'[01])':'[0-5][0-9]))WS[ap]'.m.';
7   fragment Noon: 'Noon'|'12 noon';
8   fragment Midnight: 'Midnight'|'12 midnight';
9
10  WS: [ \t\r\n]+ -> skip;
```

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:

```
// TimeLexerV2.g4
lexer grammar TimeLexerV2;

TIME : HOUR SEPERATOR MINUTE (AM | PM)
| TWELVE SEPERATOR '00' (AM | PM)
| TWELVE 'noon'
| TWELVE 'midnight'
| 'Noon'
| 'Midnight';

TWELVE : '12';

HOUR : '1'[0-1]|[0-9];
MINUTE : [0-5][0-9];
SEPERATOR : ':';

AM : 'a.m.' ;
PM : 'p.m.' ;

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

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.

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

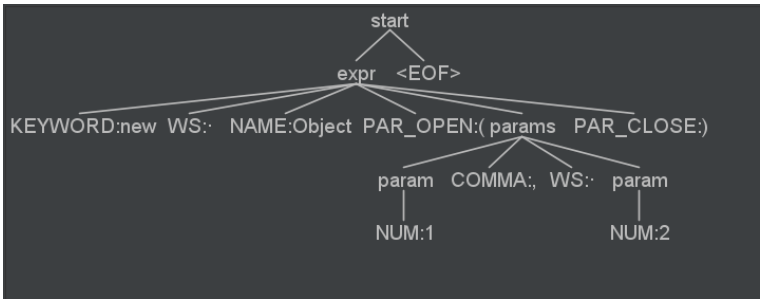
a - Lösung

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 eigene neu erzeugte Klassen sein.

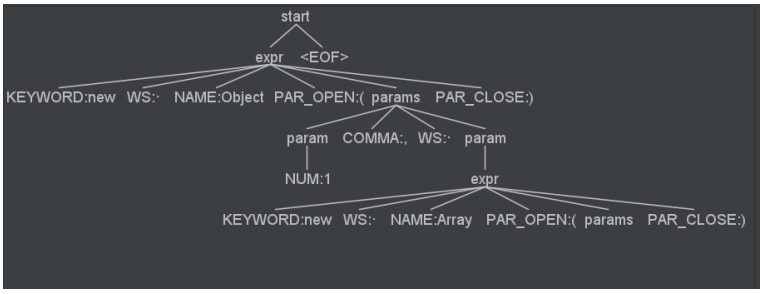
```
1   // CreationLexer.g4
2   lexer grammar CreationLexer;
3
4   KEYWORD : 'new' ;
5
6   NAME : [A-Za-z]+ ;
7   NUM : [0-9]+ ;
8
9   COMMA : ',' ;
10
11  PAR_OPEN : '(' ;
12  PAR_CLOSE : ')' ;
13
14  WS : [ \t\r\n]+ ;
15
16  InvalidChar: . ;
```

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) ;
```



(a) Input: new Object(1, 2)



(b) Input: new Object(1, new Array())

Figure 1: Parse Tree Beispiele

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