Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Oskar Borkenhagen
Marcel Biselli

os741bor@htwg-konstanz.de
ma741bis@htwg-konstanz.de

AIN - Sprachkonzepte

Übungsaufgaben Bericht

WS2022/23

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

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

Beispieleingaben:

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

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();
}
```

Quelltext 1.1: Funktion für Date/Time Formatter

Das Pattern ist realisiert anhand der 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  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 1.2: Lexer für Date/Time

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 1.3: 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.

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 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 auch eigene, neu erzeugte Klassen sein.

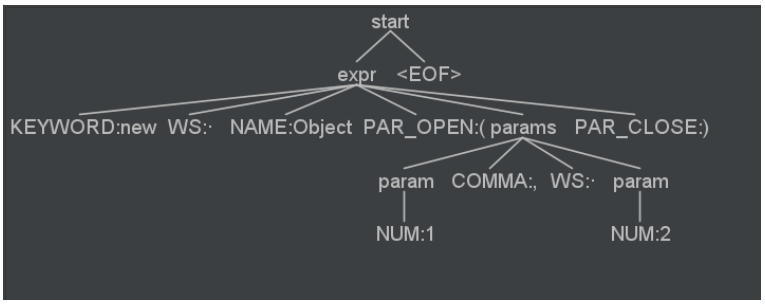
```
1   // CreationLexer.g4
2   lexer grammar CreationLexer;
3
4   KEYWORD : 'new' ;
5
6   NAME : [A-Za-z]+ ;
7   NUM : [0-9]+ ;
8
9   COMMA : ',' ;
10
11  PAR_OPEN : '(' ;
12  PAR_CLOSE : ')' ;
13
14  WS : [ \t\r\n]+ ;
15
16  InvalidChar: . ;
```

Quelltext 2.1: Lexer Grammar A2a

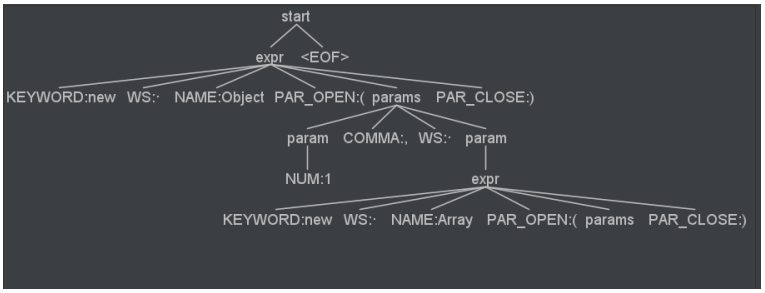
Der dazugehörige Parser:

```
// CreationParser.g4
parser grammar CreationParser;
options { tokenVocab=CreationLexer; }

start : expr EOF;

expr : KEYWORD WS NAME PAR_OPEN params PAR_CLOSE ;

params : (param (COMMA WS? param)*)? ;

param : (expr | NAME | NUM) ;
```

Quelltext 2.2: Parser Grammar A2a



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



(b) Input: 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:

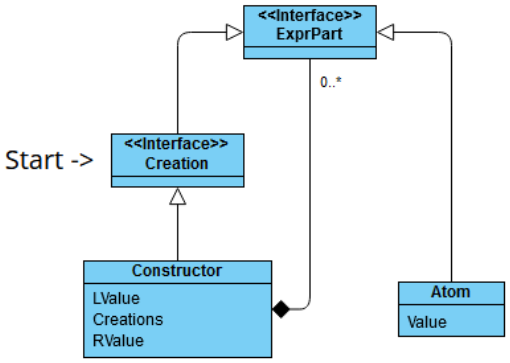


Abbildung 2.2: UML-Diagramm

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:

```java
public interface Expr {
}

public interface Creation extends Expr {
}

public class Atom implements Expr {
    private final String val;

    public Atom(String val) {
        this.val = val;
    }
    public String getVal() {
        return val;
    }
    @Override
    public String toString() {
        return this.val;
    }
}

public class Constructor implements Creation {
    private final String leftVal;
    private final List<Expr> params;
    private final String rightVal;

    public Constructor(String leftVal, List<Expr> params, String
    rightVal) {
        this.leftVal = leftVal;
        this.params = params;
        this.rightVal = rightVal;
    }
    public String getLeftVal() {
        return leftVal;
    }
    public List<Expr> getParams() {
        return params;
    }
    public String getRightVal() {
        return rightVal;
    }
    @Override
    public String toString() {
        return this.leftVal + this.params + this.rightVal;
    }
}
```

Quelltext 2.3: Grundlegende AST-Klassen

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

```java
public class CreationBuilder extends CreationParserBaseListener {

    private final List<Stack<Expr>> stackList = new LinkedList<>();
    private int depth = -1;

    public Creation build(ParseTree tree) {
        new ParseTreeWalker().walk(this, tree);

        return (Creation) this.stackList.get(this.depth).pop();
    }

    @Override
    public void enterExpr(CreationParser.ExprContext ctx) {
        this.stackList.add(new Stack<>());
        this.depth++;
    }

    @Override
    public void exitExpr(CreationParser.ExprContext ctx) {
        if (ctx.getChildCount() == 6) {
            var l = new StringBuilder();
            for (int i = 0; i < 4; i++) {
                l.append(ctx.getChild(i).getText());
            }


            var c = new Constructor(
                    l.toString(),
                    new LinkedList<>(this.stackList.get(this.depth)),
                    ctx.getChild(5).getText()
            );
            this.stackList.get(this.depth).clear();

            if (this.depth > 0)
                this.depth--;
            this.stackList.get(this.depth).push(c);

        }
    }

    @Override
    public void enterParam(CreationParser.ParamContext ctx) {
        if (ctx.start.getType() == CreationParser.NUM) {

            this.stackList.get(this.depth).push(new
    Atom(ctx.NUM().getText()));

        } else if (ctx.start.getType() == CreationParser.NAME) {

            this.stackList.get(this.depth).push(new
    Atom(ctx.NAME().getText()));

```

```
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

4. Aufgabe 4