FCW 1x Übung zu Formale Sprachen, Compiler- und Werkzeugbau 1 — WS 2016/17, Übung 1

Abgabetermin: in der KW 43

☐ **Gr. 1,** Dr. H. Dobler
☐ **Gr. 2,** Dr. G. Kronberger

Name _____ **Aufwand in h** _____

Punkte _____ **Übungsleiter** _____

1. Grammatiken – Grundbegriffe (8 Punkte)

Die folgende Grammatik *G*(*DataStat*) beschreibt in vereinfachter Form den Aufbau der *DATA*[1] Anweisung der Programmiersprache Fortran 77:

```
DataStat       = "DATA" DataDecl DataDeclRest.
DataDeclRest   = ε | DataDeclRest ["," ] DataDecl.
DataDecl       = DataNameList "/" DataValueList "/".
DataNameList   = DataName | DataName "," DataNameList.
DataName       = id | DataDoList.
DataValueList  = DataValue | DataValueList "," DataValue.
DataValue      = ( (num | id) ["*" (id | ("+"|"-"|ε) num | str)] ) |
                 ( ["+"|"-"] num | str ).
DataDoList     = "(" ( id "(" id {"," id} ")" | DataDoList )
                 DataDoListRest ")".
DataDoListRest = ε |
                 DataDoListRest "," (id ( "(" expr { "," expr} ")"
                                        | "=" expr "," expr (ε | expr)
                                        )
                                    | DataDoList
                                    ).
```

a) Bestimmen Sie die Mengen V_T und V_N.

b) Geben Sie den/die kürzesten Satz/Sätze an, den/die man mit dieser Grammatik bilden kann.

c) Ermitteln Sie alle rekursiven Nonterminalsymbole. Geben Sie für jedes dieser Nonterminalsymbole an, ob es direkt- oder indirekt- und links-, zentral- oder rechtsrekursiv ist.

d) Transformieren Sie die gegebene Grammatik in das Regelsystem der formalen Sprachen. Welche Grammatikschreibweise halten Sie für lesbarer? Begründen Sie Ihre Antwort.

e) Zeichnen Sie den Syntaxbaum für folgenden Satz (verwenden Sie dazu die gegebene Grammatik *G*(*DataStat*)):

```
DATA id, id / num * str /, ( id(id), id = expr, expr ) / num * num /
```

Gibt es mehrere Syntaxbäume für diesen Satz? (Mit Begründung!)

2. Konstruktion einer Grammatik (4 Punkte)

Konstruieren Sie eine Grammatik (in der Schreibweise des Regelsystems der formalen Sprachen) für die Menge aller ungeraden ganzen Dezimalzahlen mit optionalem Vorzeichen. Die Zahlen dürfen keine führenden Nullen enthalten. Geben Sie Ihre Grammatik nun auch in Wirth'scher EBNF mit möglichst wenig Regeln an.

[1] Eine genauere Erklärung dieser Anweisung finden Sie am Ende des Übungszettels.

3. Oo-Implementierung von Grammatiken (4 + 6 + 2 Punkte)

Studieren Sie die oo Implementierung von Grammatiken in *FormalLanguagesForStudents* aus dem *moodle*-Kurs. Das UML-Klassendiagramm dafür finden Sie in der VL-Präsentation für den FS-Teil.

a) In der Vorlesung wurden Algorithmen auf Grammatiken besprochen, insbesondere jener zur Beseitigung leerer Alternativen. Implementieren Sie diesen Algorithmus in Form einer Funktion

```
Grammar *epsilonFreeGrammarOf(Grammar *g);
```

Testen Sie Ihre Implementierung mit der Beispielgrammatik im Foliensatz der VL (auf S. 26).

b) Die Sprache *L* einer Grammatik *G* mit dem Statzsymbol *S*, also *L*(*G*(*S*)), ist nichts anderes als die Menge aller terminalen Ketten σ, die sich aus *S* mit den Regeln aus der Grammatik ableiten lassen. Eine Kette wird durch ein Objekt der Klasse *Sequence* repräsentiert. Entwickeln Sie eine Klasse *Language*, die eine Menge solcher Ketten speichert und eine Funktion

```
Language *generateLanguage(Grammar *g, int maxLen);
```

die alle Sätze bis zur Länge *maxLen* einem *Language*-Objekt erzeugt.

c) Wenn Sie in die Klasse *Language* noch eine Methode

```
bool Language::hasSentence(Sequence *s) const;
```

einbauen, haben Sie einen einfachen Mechanismus für die Syntaxanalyse.

Testen Sie Ihre Implementierungen von b) und c) mit folgender Grammatik *G*(*S*):

```
S  → aB  |  bA
A  → a   |  aS  |  bAA
B  → b   |  bS  |  aBB
```

indem Sie alle Sätze σ dieser Grammatik erzeugen, für die $|\sigma| \leq 6$ gilt und beantworten Sie folgende Fragen: Weisen diese Sätze eine besondere Eigenschaft auf? Haben alle Sätze dieser Grammatik diese Eigenschaft? Kann man diese Eigenschaft schon aus der Grammatik ableiten?

Hintergrundinformation zur Aufgabe 1

Die *DATA*-Anweisung der Programmiersprache Fortran 77 wird zur Initialisierung von Variablen, Feldern, Feldelementen und Zeichenketten benutzt. Die Anweisung ist nicht ausführbar und kann deshalb nur im Deklarationsteil eines Fortran-Programms verwendet werden. Struktur vereinfacht dargestellt:

```
DATA var_list / val_list / { , var_list / val_list / }
```

Wobei *var_list* für durch Komma getrennte Listen von Variablen und *val_list* für durch Komma getrennte Listen von Werten steht. Die Werte sind entweder Konstanten oder Gebilde der Form *r*c* sind, wobei *r* die Anzahl der Wiederholungen (*repetitions*) der Konstante *c* angibt.

Im folgenden Beispiel werden die Variablen *A* – *L* deklariert und dann mit Initialwerten belegt:

```
BLOCKDATA SETUP
INTEGER A, B, C
REAL    I, J, K, L
COMMON  /AREA1/ A, B, C
COMMON  /AREA2/ I, J, K, L
DATA    A, B, C, I, J, K, L / 0, 1, 2, 10.0, -20.0, 30.0, -40.0 /
END
```

1 Grammatiken - Grundbegriffe

Dieser Abschnitt behandelt die Aufgabe 1 der ersten Übung.

1.1 Die Mengen V_T und V_N

V_N = { DataStat, DataDeclRest, DataDecl, DataNameList, DataName, DataValueList, DataValue, DataDoList, DataDoListRest }

Alle Nichtterminalsymbole befinden sich links in der Grammatik, wobei das Nichtterminalsymbol *DataStat* das Satzsymbol ist.

V_T = { “DATA”,“,”, “/”, “(”, “)”,“=”, “*”, “+”, “-”, “=”, expr, id, num, str }

Alle Terminalsymbole kommen nicht auf der linken Seite der Grammatik vor und können nicht weiter abgeleitet werden. Das Symbol “ϵ” ist ein Metasymbol, das die leere Kette repräsentiert und ist weder ein Nichtterminalsymbol noch ein Terminalsymbol.

$V = V_T \cup V_N$

Die Menge V ist das Alphabet der Grammatik und ist die Vereinigung der Menge der Nichtterminalsymbole und der Menge der Terminalsymbole. Da das Symbol ϵ ein Metasymbol ist, ist es nicht Teil der Grammatik und auch kein Teil des Alphabets der Grammatik.

1.2 Kürzeste Sätze der Grammatik

DataStat $\overset{L}{\Rightarrow}$ “DATA” DataDecl DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataNameList “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataName “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValue “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” num “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” num “/”

DataStat $\overset{L}{\Rightarrow}$ “DATA” DataDecl DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataNameList “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataName “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValue “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” id “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” id “/”

DataStat $\overset{L}{\Rightarrow}$ “DATA” DataDecl DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataNameList “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataName “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValue “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” str “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” str “/”

Das Nichtterminalsymbol *DataValue* kann in drei Varianten abgeleitet werden, *id*, *num* und *str*. Die Satzlänge bleibt bei jeder der drei Ableitungen des Nichtterminalsymbols *DataValue* gleich.

1.3 Rekursionen der Nichtterminalsymbole

Nichtterminalsymbol	direkt/indirekt rek.	links/rechts rek.
DataDeclRest	direkt rek.	links rek.
DataDecl	-	-
DataNameList	direkt rek.	rechts rek.
DataName	-	-
DataValueList	direkt rek.	links rek.
DataValue	-	-
DataDoList	direkt rek.	zentral rek.
DataDoList	indirekt rek.	zentral rek.
DataDoListRest	indirekt rek.	zentral rek.

Die beiden indirekten Rekursionen der Nichtterminalsymbole *DataDoList* und *DataDoListRest* ergeben sich aus der Tatsache, dass wenn es eine indirekte Rekursion gibt, es auch eine zweite Rekursion geben muss. Ich gehe davon aus, dass ich alle Rekursionen gefunden habe, wobei angemerkt sei, dass es sehr schwer ist Rekursionen aus einer Grammatik auszulesen.

1.4 Transformation in das Regelsystem der formalen Sprachen

DataStat	→	DATA DataDecl DataDeclRest
DataDeclRest	→	ϵ | DataDeclRest , DataDecl | DataDeclRest DataDecl
DataDecl	→	DataNameList / DataValueList /
DataNameList	→	DataName | DataName , DataNameList
DataName	→	id | DataDoList
DataValueList	→	DataValue | DataValueList , DataValue
DataValue	→	num DerefValue | id DerefValue | NumOrStr
NumValue	→	num | + num | - num
NumOrStr	→	NumValue | str
DerefValue	→	ϵ | * id | * NumOrStr
DataDoList	→	(IdList DataDoListRest) | (DataDoList DataDoListRest)
CommaId	→	ϵ | , id CommaId
IdList	→	id (id CommaId)
DataDoListRest	→	ϵ | DataDoListRest , DoListRestOpt
ComaExpr	→	ϵ | , expr CommaExpr
ExprList	→	(expr CommaExpr)
OptExpr	→	ϵ | expr
EqualExpr	→	= expr , expr OptExpr
DoListRestOpt	→	id ExprList | id EuqlExpr | DataDoList

Um diesen Punkt der Aufgabe 1 zu lösen wurden die Optionen und Schleifen von innen nach außen aufgelöst und in eigene Nichtterminalsybole ausgelagert. Das ist notwendig, da das Regelwerk der formalen Sprachen diese Konstrukte nicht kennt und die Konstrukte Optionen und Schleifen in *Oder* Konstrukte umgewandelt werden müssen. Bsp.: [num, str] $\equiv \epsilon$ | num | str

Ich halte die Schreibweise der Wirth'schen EBNF für lesbarer, da in dieser Schreibweise mehr Konstrukte wie Optionen und Schleifen zur Verfügung stehen und dadurch die Grammatik durch weniger Regeln definiert werden kann, was die Grammatik übersichtlicher macht.

1.5 Syntaxbaum

Die Abbildung 1 zeigt den Syntaxbaum des Satzes
*DATA id, id / num * str /, (id(id), id = expr, expr) / num * num /.*

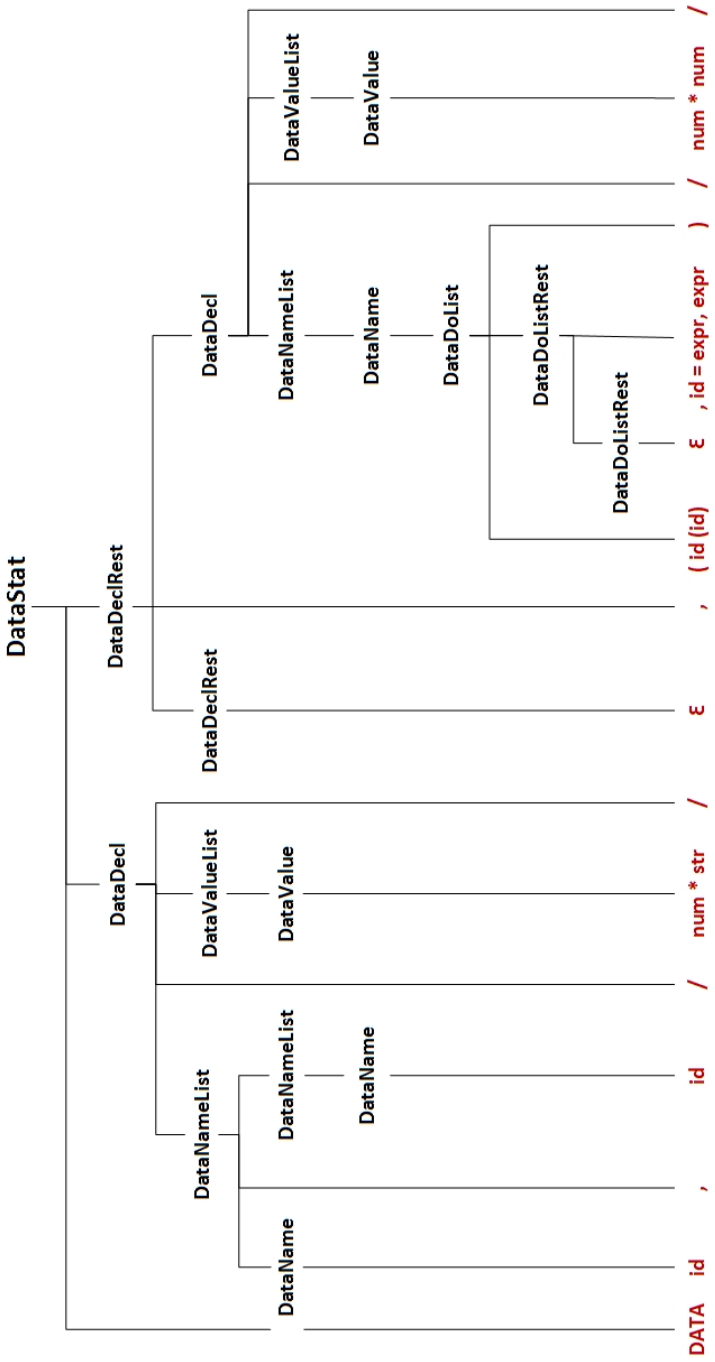

Abbildung 1: Syntaxbaum

Es gibt für den gegebenen Satz nur einen Syntaxbaum, da es keine Möglichkeit gibt, den Satz mit der gegebenen Grammatik über andere Wege in einen anderen Syntaxbaum zu überführen. Das bedeutet aber nicht dass die Grammatik eindeutig ist, da es nötig wäre alle möglichen Sätze in alle ihre möglichen Syntaxbäume zu überführen, was mit der gegebenen Grammatik nicht möglich ist, da mit der gegebenen Grammatik unendliche viele Sätze erstellt werden können.

2 Konstruktion einer Grammatik

Dieser Abschnitt behandelt die Aufgabe 2 der ersten Übung.

2.1 Mit dem Regelsystem der formalen Sprachen

In diesem Abschnitt wird die Grammatik mit dem Regelsystem der formalen Sprachen angeführt.

SignedOddDecimal	→	Sign IntegerNoZero AllIntegerOpt, AllIntegerOpt OddInteger
Sign	→	ϵ | + | -
OddInteger	→	1 | 3 | 5 | 7 | 9
IntegerNoZero	→	2 | 4 | 6 | 8 | OddInteger
AllInteger	→	0 | IntegerNoZero
AllIntegerOpt	→	ϵ | AllInteger | AllIntegerOpt

2.2 Mit der Wirth'schen EBNF

Dieser Abschnitt wird die Grammatik mit der With'schen EBNF angeführt.

SignedOddDecimal	=	[+ | -] IntegerNoZero {0 | IntegerNoZero}, {0 | IntegerNoZero} OddInteger
OddInteger	=	1 | 3 | 5 | 7 | 9
IntegerNoZero	=	2 | 4 | 6 | 8 | OddInteger

3 Oo-Implementierung von Grammatiken

Dieser Abschnitt behandelt die Aufgabe 3 der ersten Übung.

3.1 *Grammar *epsilonFreeGrammarOf(Grammar *grammar)*

Die Implementierung erfolgte in den Quelltextdateien *GrammarUtil.hpp* und *GrammarUtil.cpp* und wurde im Namensraum *GrammarUtil* organisiert.

3.1.1 Quelltexte

Listing 1: GrammarUtil.hpp

```
1  //
2  // Created by Thomas on 10/15/16.
3  //
4  #ifndef _INCLUDES_LANGUAGE_UTILS_HPP
5  #define _INCLUDES_LANGUAGE_UTILS_HPP
6
7  #include "Grammar.h"
8  #include "string"
9
10 #endif
11
12 using namespace std;
13
14 // see: http://www.informatikseite.de/theorie/node43.php
15 namespace GrammarUtil {
16
17     /**
18      * Generates a epsilon free grammar of the this grammar.
19      * @param _grammar the grammar to transform to epsilon free grammar
```

```
20      * @return the transformed epsilon free grammar, or the given grammar if already epsilon free
21      * @throw invalid_argument if an parameter is null
22      */
23     Grammar *epsilonFreeGrammarOf(const Grammar *_grammar);
24
25     /**
26      * Finds all NTSymbols the given sequence can be reduced to.
27      * @param grammar the related grammar
28      * @param _sequence the sequene to be tried to be reduced
29      * @return the set of found NTSymbols
30      */
31     set<NTSymbol *> findNTSymbolSForSequence(const Grammar *grammar, Sequence *_sequence);
32
33     /**
34      * Reduces the given sentences up to the root NTSymbol.
35      * @param grammar the related grammar
36      * @param _sentence the sentence to be reduced
37      * @return the last symbol which should be the root symbol of the grammar if the sentence
38      *          is a sentence of the grammar related language
39      */
40     Symbol *reduce(const Grammar *grammar, Sequence _sentence);
41 }
```

Listing 2: GrammarUtil.cpp

```
1  //
2  // Created by Thomas on 10/15/16.
3  //
4
5  #ifndef _INCLUDES_GRAMMAR_UTIL_CPP
6  #define _INCLUDES_GRAMMAR_UTIL_CPP
7
8  #include <cstdarg>
9  #include <iostream>
10 #include <queue>
11
12 #include "GrammarUtil.hpp"
13
14 #endif
15
16 namespace GrammarUtil {
17
18     //region Private Functions Definition
19     /**
20      * Transforms the the sequences of the given rule of an deletable NTSymbol.
21      *
22      * @param thr rule of an deletable NTSymbol to be transformed
23      * @return the set of trqansformed sequences. Not of type SequenceSet because SeuqenceSet
 ↪   deletes its items
24      * @throw invalid_argument if the given SequenceSet is a nullptr
25      */
26     set<Sequence *> *transformSequence(const SequenceSet *_oldSequences);
27     //endregion
28
29     //region Public Functions  Implementation
30     Grammar *epsilonFreeGrammarOf(const Grammar *_grammar) {
31         if (_grammar == nullptr) {
32             throw invalid_argument("invalid nullptr for _grammar");
33         } // if
34
35         // identify deletable rules on given _grammar
36         _grammar->identifyDeletableNTs();
```

```
37
38          // return _grammar if already epsilon free
39          if (_grammar->isEpsilonFree()) {
40              return nullptr;
41          } // if
42
43          // Create new root which contains the alternative with the epsilon an the old root.
44          NTSymbol *root = SymbolPool::getInstance()->ntSymbol(_grammar->root->name + "'");
45          Grammar *epsilonFreeGrammar = new Grammar(_grammar->root);
46          epsilonFreeGrammar->addRule(root, 2, new Sequence(), new Sequence(_grammar->root));
47
48          // iterate over all rules
49          for (auto &rule: _grammar->rules) {
50              if (_grammar->isDeletable(rule.first)) { // transform sequence if NTSymbol is
    ↪  deletable
51                  set<Sequence *> *newSequences = transformSequence(&rule.second);
52                  for (auto newSeq : *newSequences) {
53                      epsilonFreeGrammar->addRule(rule.first, 1, *newSeq);
54                  } // for
55
56                  // delete set (not items !!)
57                  delete (newSequences);
58              } else { // copy sequence if NTSymbol is not deletable
59                  for (auto oldSeq : rule.second) {
60                      epsilonFreeGrammar->addRule(rule.first, 1, new Sequence(*oldSeq));
61                  } // for
62              } // if
63          } // for
64
65          return epsilonFreeGrammar;
66      } // Grammar::epsilonFreeGrammarOf
67
68      set<NTSymbol *> findNTSymbolSForSequence(const Grammar *_grammar, Sequence *_sequence) {
69          if (_grammar == nullptr) {
70              throw invalid_argument("Language::reduceToNTSymbol: invalid nullptr for grammar");
71          } // if
72          if (_sequence == nullptr) {
73              throw invalid_argument("Language::reduceToNTSymbol: invalid nullptr for sentence");
74          } // if
75
76          set<NTSymbol *> result;
77          for (auto ruleIt = _grammar->rules.begin(); ruleIt != _grammar->rules.end(); ruleIt++) {
78              if (ruleIt->second.find(_sequence) != ruleIt->second.end()) {
79                  result.insert(ruleIt->first);
80              } // if
81          } // for
82
83          return result;
84      } // Grammar::Rule findRuleForSequence
85
86
87      Symbol *reduce(const Grammar *grammar, Sequence _sentence) {
88          Symbol *result = nullptr;
89          if ((_sentence.length() == 1) && ((*_sentence.begin())->isNT())) {
90              result = *_sentence.begin();
91          }
92          for (int i = 0; i < _sentence.length(); ++i) {
93              for (int j = i; j < _sentence.length(); ++j) {
94                  Sequence sequence;
95                  sequence.insert(sequence.begin(), _sentence.begin() + i, _sentence.begin() + j +
    ↪  1);
96
97                  set<NTSymbol *> ntSymbols = findNTSymbolSForSequence(grammar, &sequence);
```

```
98                  if (!ntSymbols.empty()) {
99                      for (auto foundSymbol: ntSymbols) {
100                         Sequence tmp(_sentence);
101                         tmp.erase(tmp.begin() + i, tmp.begin() + j + 1);
102                         tmp.insert(tmp.begin() + i, foundSymbol);
103                         result = reduce(grammar, tmp);
104                     }
105                 }
106             }
107         }
108
109         return result;
110     }
111     //endregion
112
113     //region Private Functions Implementation
114     /**
115      * Transforms the the sequences of the given rule of an deletable NTSymbol.
116      *
117      * @param _oldSequences the sequences of the old rule
118      * @return the tranformed rule
119      * @throw invalid_argument if the given set is a nullptr
120      */
121     set<Sequence *> *transformSequence(const SequenceSet *_oldSequences) {
122         if (_oldSequences == nullptr) {
123             throw invalid_argument("Given _oldSequences set is a nullptr.");
124         }
125
126         // use ordinary set and not SequenceSet because SequenceSet deletes its items
127         set<Sequence *> *transformedSet = new set<Sequence *>();
128
129         // iterate over all _oldSequences
130         for (Sequence *seq: *_oldSequences) {
131             int cursorIdx = -1;
132             int length = seq->length();
133             int innerCursorIdx = 0;
134
135             // epsilon not part of transformed
136             if (seq->isEpsilon()) {
137                 continue;
138             } // if
139
140             // original sequence is always part of transformed
141             Sequence *originalSequence = new Sequence(*seq);
142             if (!transformedSet->insert(originalSequence).second) {
143                 delete originalSequence;
144             } // if
145
146             // if only terminals, then no transformation needed
147             if (seq->hasTerminalsOnly()) {
148                 continue;
149             } // if
150
151             // add sequence with first NT if NTSymbols only, otherwise get lost by following
    ↪  algorithm
152             // TODO: For loop should not be necessary
153             if (seq->hasNonTerminalsOnly()) {
154                 for (auto it = seq->begin(); it != seq->end(); it++) {
155                     Symbol *symbol = seq->symbolAt(it);
156                     if (symbol->isNT()) {
157                         Sequence *firstSymbolSequence = new Sequence(symbol);
158                         if (!transformedSet->insert(firstSymbolSequence).second) {
159                             delete firstSymbolSequence;
```

```
160                         } // if
161                         break;
162                     } // if
163
164                     // move cursor by one which will stay one symbol before added one
165                     cursorIdx++;
166                 } // for
167             }
168
169             // iterate as long the cursorIdx cursor can be moved
170             while (cursorIdx != (length - 1)) {
171                 // iterate as long as inner cursor can be moved
172                 while (innerCursorIdx < length) {
173                     Sequence *tmpSequence = new Sequence();
174                     int itIdx = -1;
175                     // iterate over each symbol in the sequence
176                     for (auto it = seq->begin(); it != seq->end(); it++) {
177                         itIdx++;
178                         Symbol *symbol = seq->symbolAt(it);
179
180                         // do not escape leading TSymbols
181                         if (symbol->isT()) {
182                             tmpSequence->appendSymbol(symbol);
183                             // move inner cursor if it stands on TSymbol
184                             if (itIdx == innerCursorIdx) {
185                                 innerCursorIdx++;
186                             } // if
187                             continue;
188                         } //if
189
190                         // ignore all before cursor and inner cursor position
191                         if ((itIdx > cursorIdx) && (itIdx != innerCursorIdx)) {
192                             tmpSequence->appendSymbol(symbol);
193                         } // if
194                     } // for
195
196                     // move inner cursor
197                     innerCursorIdx++;
198
199                     // insert and delete sequence if insert failed
200                     if ((tmpSequence->empty()) || (!transformedSet->insert(tmpSequence).second)) {
201                         delete tmpSequence;
202                     } // if
203                 } // while
204
205                 // move cursors
206                 cursorIdx++;
207                 innerCursorIdx = cursorIdx + 1;
208
209             } // while
210         } // for
211
212         return transformedSet;
213     }
214     //endregion
215 }
```

3.1.2 Tests

Dieser Abschnitt behandelt die Tests der Implementierung aus dem Abschnitt 3.1.

```
TESTCASE 4

-----------------------------------------
Grammar transformation tests
-----------------------------------------
-----------------------------------------
Grammar from the presentation
-----------------------------------------

G(S):
A -> EPS | B B
B -> C C | a
C -> A A | b
S -> A B C
---
VNT = { A, B, C, S }, deletable: { A, B, C, S }
VT  = { a, b }

-----------------------------------------
Epsilon free grammar from the presentation
-----------------------------------------

G(S):
A -> B | B B
B -> C | C C | a
C -> A | A A | b
S -> A | A B | A B C | A C | B | B C | C
S' -> EPS | S
---
VNT = { A, B, C, S, S' }, deletable: { S' }
VT  = { a, b }

-----------------------------------------
```

Abbildung 2: Epsilonfreie Grammatik der vorgegebenen Grammatik

Die Tests aus Abbildung 2 zeigen die vorgegebenen Grammatik aus den Vorlesungsunterlagen und die epsilonfreie Variante dieser Grammatik.

```
------------------------------------------
Grammar from the internet
------------------------------------------

G(S):
A -> D D D D | a A A | a a b
B -> A A C | b
C -> EPS | S B
D -> A a c b S | C E
E -> C | b c a
S -> A C B | a C b
---
VNT = { A, B, C, D, E, S }, deletable: { A, B, C, D, E, S }
VT  = { a, b, c }

------------------------------------------
Epsilon free grammar from the internet
------------------------------------------

G(S):
A -> D | D D | D D D | D D D D | a | a A | a A A | a a b
B -> A | A A | A A C | A C | C | b
C -> B | S | S B
D -> A a c b | A a c b S | C | C E | E | a c b | a c b S
E -> C | b c a
S -> A | A B | A C | A C B | B | C | C B | a C b | a b
S' -> EPS | S
---
VNT = { A, B, C, D, E, S, S' }, deletable: { S' }
VT  = { a, b, c }

------------------------------------------
```

Abbildung 3: Epsilonfreie Grammatik des Internetbeispiels

Die Tests aus Abbildung 3 zeigen eine Grammatik, die im Internet unter folgender *URL* `http://www.informatikseite.de/theorie/node43.php` gefunden werden kann und die epsilonfreie Variante dieser Grammatik.

3.2 *Language *generateLanguage(Grammar *g, int maxLen)*

Alle möglichen Sätze der Sprache bis zur maximal erlaubten Länge, die durch die vorgegebene Grammatik möglich sind, werden in der Funktion `derive(const Grammar *_grammar, int maxLength)` durch das Ermitteln aller möglichen Ableitungen bis zur erlaubten Satzlänge der Grammatik ermittelt.

Die Sätze der Sprache bestehen aus Wiederholungen von ab, ba und Folgen von nur a oder nur b. Die Wiederholungen von ab und ba lassen sich schon aus der Grammatik aus den beiden Sequenzen bS und aS auslesen.

3.2.1 Quelltexte

In diesem Abschnitt werden die Quelltextdateien der Implementierung angeführt. Die Implementierung erfolgte in den Quelltextdateien *LanguageUtil.hpp*, *LanguageUtil.cpp*, *Language.hpp* und *Language.hpp*.

Listing 3: Language.hpp

```
1  //
2  // Created by Thomas on 10/16/16.
3  //
4  #ifndef _INCLUDES_LANGUAGE_HPP
5  #define _INCLUDES_LANGUAGE_HPP
6
7  #include "SequenceStuff.h"
8  #include "ObjectCounter.h"
9  #include "Grammar.h"
10
11 #endif
12
13 using namespace std;
14
15 /**
16  * Orders the sequences ascending by the symbol length
17  */
18 struct LessSequenceComparator {
19     bool operator()(const Sequence *seq1, const Sequence *seq2) const;
20 }; // LessSequenceComparator
21
22 /**
23  * This class represents a set of sentences which are part of the language of a grammar.
24  */
25 class Language : public std::set<Sequence *, LessSequenceComparator>,
26                  private ObjectCounter<Language> {
27
28 private:
29     /**
30      * The custom type definition of the extended Base
31      */
32     typedef std::set<Sequence *, LessSequenceComparator> Base;
33
34 public:
35
36     const Grammar *grammar;
37
38     /**
39      * Empty constructo for creating an empty Language
40      * @return the empty language instance
41      */
42     Language(const Grammar *grammar);
43
```

```
44      /**
45       * Copy constructor for copying the Language
46       * @param language the language to copy
47       * @return the copied language
48       * @throws invalid_argument if the language object is null
49       */
50      Language(const Language &language);
51
52      /**
53       * Destructor for deleting the Language and the held Sequences
54       */
55      ~Language();
56
57      /**
58       * Appends a sentence to the language
59       * @param _sentence the sentence set representing the sentence
60       * @return true if inserted, false otherwise
61       * @throws invalid_argument if the seuqnece object is null
62       */
63      bool appendSentence(Sequence *_sentence);
64
65      /**
66       * Answers the question if the given sequence is part of this language.
67       * @param _sentence the sequence to check if a sentence of this laguage
68       * @return  true if part of this language, false otherwise
69       * @throws invalid_argument if the seuqnece object is null
70       */
71      bool hasSentence(Sequence *_sentence);
72
73      /**
74       * Answers the question if a sentence represented by the given sequence is already part of
    ↪   this language instance
75       * @param _sentence the pointer to the sequence object
76       * @return true if sentece already registered in language, false otherwise
77       */
78      bool containsSentence(Sequence *_sentence);
79  };
80
81  /**
82   * The outstream operator implementation.
83   *
84   * @param os the current ouput stream
85   * @param language the language object to be put on stream
86   * @return the manipulated output stream
87   */
88  std::ostream &operator<<(std::ostream &os, const Language &language);
```

Listing 4: Language.cpp

```
1   // Created by Thomas on 10/16/16.
2   //
3
4   #ifndef _INCLUDES_LANGUAGE_CPP
5   #define _INCLUDES_LANGUAGE_CPP
6
7   #include <cstdarg>
8   #include "GrammarUtil.hpp"
9   #include "Language.hpp"
10
11  #include <iostream>
12  #include <stdexcept>
13
```

```
14  #endif
15
16  using namespace std;
17
18  //region Constructors and Destructors
19  Language::Language(const Grammar *_grammar) {
20      if (_grammar == nullptr) {
21          throw invalid_argument("Language::Language: invalid nullptr for grammar");
22      } // if
23      this->grammar = _grammar;
24  } // Language::Language
25
26  Language::Language(const Language &ss) : Base(ss), ObjectCounter<Language>() {
27      // nothing left to do
28  } // Language::Language
29
30  Language::~Language() {
31      for (Sequence *sequence: *this) {
32          delete (sequence);
33      } // for
34  } // Language::~Language
35  //endregion
36
37  //region Public Methods
38  bool Language::appendSentence(Sequence *_sentence) {
39      if (_sentence == nullptr) {
40          throw invalid_argument("Language::appendSentence: invalid nullptr for _sentence");
41      } // if
42
43      return (!containsSentence(_sentence)) ? insert(_sentence).second : false;
44  } // Language::appendSentence
45
46  bool Language::hasSentence(Sequence *_sentence) {
47      if (_sentence == nullptr) {
48          throw invalid_argument("Language::hasSentence: invalid nullptr for sequence");
49      } // if
50
51      Symbol *ntSymbol = GrammarUtil::reduce(this->grammar, *_sentence);
52
53      return ((ntSymbol != nullptr) && (this->grammar->root->name == ntSymbol->name));
54  } // Language::hasSentence
55
56  bool Language::containsSentence(Sequence *_sentence) {
57      for (Sequence *tmp : *this) {
58          if (*tmp == *_sentence) {
59              return true;
60          } // if
61      } // for
62
63      return false;
64  } // Language::containsSentence
65  //endregion
66
67  //region Other stuff
68  bool LessSequenceComparator::operator()(const Sequence *seq1,
69                                          const Sequence *seq2) const {
70      return seq1->length() <= seq2->length();
71  } // LessSequenceComparator::operator()
72
73  std::ostream &operator<<(std::ostream &os, const Language &language) {
74      cout << "Language contained sentences:" << endl;
75      for (const Sequence *sentence: language) {
76          cout << *sentence << endl;
```

```
77      } // for
78      return os;
79  } // operator<<
80  //endregion
```

Listing 5: LanguageUtil.hpp

```
1   //
2   // Created by Thomas on 10/26/16.
3   //
4   #ifndef _INCLUDES_LANGUAGE_UTILS_HPP
5   #define _INCLUDES_LANGUAGE_UTILS_HPP
6
7   #include "Grammar.h"
8   #include "Language.hpp"
9   #include "string"
10
11  #endif
12
13  using namespace std;
14
15  // see: http://www.informatikseite.de/theorie/node43.php
16  namespace LanguageUtil {
17
18      /**
19       * Generates a language of the given grammar with sentences length up to given maxLength
20       * @param grammar the grammar to generate language for
21       * @param maxLength the max length of the to generate sentences
22       * @return the generatd language
23       */
24      Language *generateLanguage(Grammar *grammar, int maxLength);
25
26      /**
27       * Derives the given grammar to all sentences with TSymbols only up to maxLength.
28       * @param _grammar the grammar to derive sentences for
29       * @param maxLength the maximum length of the sentences to derive
30       * @return the set of all possible sentences
31       */
32      set<Sequence *> derive(const Grammar *_grammar, int maxLength);
33  }
```

Listing 6: LanguageUtil.cpp

```
1   //
2   // Created by Thomas on 10/26/16.
3   //
4   #ifndef _INCLUDES_LANGUAGE_UTILS_CPP
5   #define _INCLUDES_LANGUAGE_UTILS_CPP
6
7   #include <cstdarg>
8   #include <iostream>
9   #include <queue>
10  #include "LanguageUtil.hpp"
11
12  #endif
13
14  namespace LanguageUtil {
15
16      //region Private Functions Definition
17      /**
```

```
18      * Derives the given grammar recursively to all possible sentences with TSymbols only up to
    ↪   maxLength.
19      *
20      * @param _grammar the grammar to get sentences up to maxLenth for the language of the grammar
21      * @param _currentSentence the current sentence to derive
22      * @param sentences the sentences set holding all TSymbols sentences only
23      * @param maxLength the maximum length of the sentences
24      */
25     void deriveRecursive(const Grammar *_grammar, Sequence _currentSentence, set<Sequence *>
    ↪   *sentences,
26                          const int maxLength);
27     //endregion
28
29     //region Public Functions Implementation
30     Language *generateLanguage(Grammar *_grammar, int _maxLength) {
31         if (_grammar == nullptr) {
32             throw invalid_argument("GrammarUtil::Language *generateLanguage: invalid nullptr for
    ↪   grammar");
33         } // if
34         if (_maxLength <= 0) {
35             throw invalid_argument("GrammarUtil::Language *generateLanguage: maxLength must be
    ↪   greater than zero");
36         } // if
37
38         set<Sequence *> result = derive(_grammar, _maxLength);
39         Language *language = new Language(_grammar);
40         for (Sequence *sequence : result) {
41             if (!language->appendSentence(sequence)) {
42                 delete (sequence);
43             } // if
44         } // for
45         return language;
46     } // Language *generateLanguage
47
48     set<Sequence *> derive(const Grammar *_grammar, int maxLength) {
49         set<Sequence *> sentences;
50         Sequence sentence(_grammar->root);
51         deriveRecursive(_grammar, sentence, &sentences, maxLength);
52         return sentences;
53     } // set<Sequence *> derive
54     //endregion
55
56     //region Private Functions Implementation
57     void deriveRecursive(const Grammar *_grammar, Sequence _currentSentence, set<Sequence *>
    ↪   *sentences,
58                          const int maxLength) {
59         // anchor
60         if (_currentSentence.hasTerminalsOnly()) {
61             if (_currentSentence.length() <= maxLength) {
62                 sentences->insert(new Sequence(_currentSentence));
63             }
64             return;
65         } else if (_currentSentence.length() > maxLength) {
66             return;
67         } // if
68
69         for (int i = 0; i < _currentSentence.length(); i++) {
70             Symbol *symbol = _currentSentence[i];
71             if (symbol->isNT()) {
72                 auto const rule = _grammar->rules.find((NTSymbol *) symbol);
73                 if (rule != _grammar->rules.end()) {
74                     for (const auto sequence : rule->second) {
75                         Sequence newSentence;
```

```
76                          newSentence.insert(newSentence.begin(), _currentSentence.begin(),
    ↪  _currentSentence.begin() + i);
77                          newSentence.insert(newSentence.end(), sequence->begin(), sequence->end());
78                          newSentence.insert(newSentence.end(), _currentSentence.begin() + i + 1,
    ↪  _currentSentence.end());
79                          deriveRecursive(_grammar, newSentence, sentences, maxLength);
80                      } // for
81                  } // if
82              } // if
83          } // for
84      } // void deriveRecursive
85      //endregion
86  }
```

3.2.2 Tests

Dieser Abschnitt behandelt die Tests der Implementierung aus dem Abschnitt 3.2.

```
-------------------------------------------
Language generation tests
-------------------------------------------

G(S):
A -> a | a S | b A A
B -> a B B | b | b S
S -> a B | b A
---
VNT = { A, B, S }, deletable: {  }
VT  = { a, b }


Language contained sentences:
b a
a b
b a a b
b b a a
b a b a
a b b a
a b a b
a a b b
b b b a a a
b a b b a a
b a b a b a
b a b a a b
b a a b b a
b a a b a b
b a a a b b
a b a b b a
a b a b a b
a b a a b b
a b b a a b
a b b a b a
b b a a a b
b b a a b a
a a b b b a
a b b b a a
a a b b a b
a a b a b b
a a a b b b
b b a b a a

Invalid sentence of language: 0
```

Abbildung 4: Tests der Generierung der Sprache

Die Tests aus Abbildung 4 zeigen die generierte Sprache für die vorgegebene Grammatik.

3.3 *bool Language::hasSentence(Sequence *s) const*

Dieser Abschnitt behandelt die Implementierung der Klassenmethode *hasSentence* für die Klasse *Language*, die überprüft ob ein gegebener Satz ein Satz dieser Sprache ist. Die Implementierung ermittelt rekursiv ob ein Satz über Reduktion zum Satzsymbol reduziert werden kann. Kann ein Satz bis zum Satzsymbol reduziert werden, dann ist dieser Satz auch ein Satz dieser Sprache. Da der implementierte Algorithmus alle möglichen Reduktionen durchprobiert verschlechtert sich das Laufzeitverhalten dieser Methode erheblich je länger die Sätze sind.

3.3.1 Quelltexte

Die Quelltexte der Implementierung sind bereits in den Quelltexten 4[Language.cpp] und 2[GrammarUtil.cpp] angeführt.