FCW 1x — Übung zu Formale Sprachen, Compiler- und Werkzeugbau 1 — WS 2016/17, Übung 3

Abgabetermin: in der KW 47

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

Name _____ **Aufwand in h** _____

Punkte _____ **Übungsleiter** _____

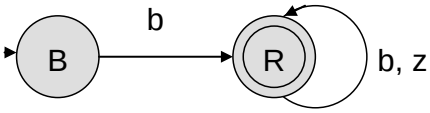
1. Objektorientierte Implementierung endlicher Automaten (1 + 4 Punkte)

a) Machen Sie sich mit der objektorientierten Implementierung endlicher Automaten in C++ vertraut. Sie finden diese im *moodle*-Kurs in der Datei *AutomataForStudents*, im Wesentlichen in den beiden Klassen *DFA* und *NFA*: Studieren Sie die Quelltexte anhand der beiden Testprogramme *DFATest* und *NFATest*.

b) Um das Verständnis (auch der oo Implementierung von Grammatiken) weiter zu festigen, erstellen Sie eine Funktion zur *XFA *xfaOf(const Grammar *g)* Transformation einer regulären Grammatik (gegeben in Form eines *Grammar*-Objekts) in einen endlichen Automaten (also in ein Objekt der Klasse *DFA* oder *NFA*, je nachdem welche Klasse Ihnen dafür besser geeignet erscheint) sowie eine Funktion *Grammar *grammarOf(const XFA *xfa)* für die umgekehrte Transformation (also *NFA* oder *DFA* nach *Grammar*).

2. DFA, Erkennung und Mealy- oder Moore-Automat (1 + 2 Punkte)

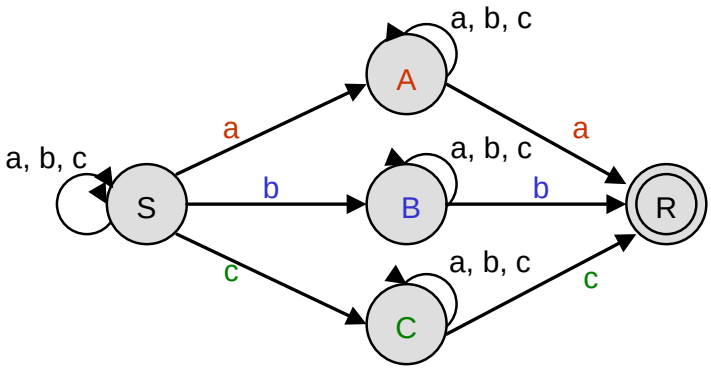
a) Schreiben Sie ein Programm, das den unten dargestellten Automaten für einfache Bezeichner erzeugt (in Form eines Objekts der Klasse *DFA*) und versuchen Sie, mit der *accepts*-Methode sowohl gültige als auch ungültige Bandinhalte zu erkennen.



b) Entwickeln Sie ausgehend von der Klasse *DFA* eine neue Klassen (*Mealy* oder *Moore*), die einen endlichen Transformationsautomaten (nach Mealy oder Moore) simuliert. Testen Sie Ihre Klasse, indem Sie einfache Bezeichner (*b* steht für *Buchstabe*, *z* steht für *Ziffer*, z. B. *bzzb*) in's Englische übersetzen (*c* für *character* und *d* für *digit*, also z.B. *bzzb -> cddc*).

3. NFA, Transformation NFA -> DFA und Zustandsminimierung (2 + 2 + 2 Punkte)

a) Schreiben Sie ein Programm, das den unten dargestellten Automaten für spezielle *abc*-Folgen erzeugt (in Form eines Objekts der Klasse *NFA*) und versuchen Sie mit den beiden Methoden *accepts* (verwendet Backtracking) und *accepts2* (verwendet Zustandsmengen) sowohl gültige als auch ungültige Bandinhalte zu erkennen.



b) Instrumentieren Sie die beiden *accepts*-Methoden so, dass Sie zur Laufzeit Maßzahlen für den Zeitaufwand der Erkennung ermitteln können.

c) Berechnen Sie mit der Methode *NFA::dfaOf* den deterministischen Automaten für obigen nichtdeterminitischen Automaten und stellen Sie diesen graphisch (am besten mittels *GraphViz* von *www.graphviz.org*) dar.

d) Stellen Sie fest, ob der in c) berechnete deterministische Automat minimal ist, indem Sie dafür, mit der Methode *DFA::minimalDfaOf* den Minimalautomaten berechnen und schauen, ob ...

4. Kellerautomat und erweiterter Kellerautomat (1 + 1 + 1 + 2 Punkte)

Die Grammatik für Variablendeklarationen in der Sprache MiniModula-2 lautet:

```
Declaration = VAR { VarDecl ";" } .
VarDecl     = IdentList ":" Type .
IdentList   = ident { "," ident } .
Type        = [ ARRAY "(" number ")" OF ] TypeIdent .
TypeIdent   = INTEGER | BOOLEAN | CHAR .
```

a) Transformieren Sie diese Grammatik in die Schreibweise der formalen Sprachen.

b) Konstruieren Sie einen *Kellerautomaten* für Sätze dieser Grammatik. (Algorithmus siehe unten.)

c) Konstruieren Sie einen *erweiterten Kellerautomaten* für die Sätze dieser Grammatik. (Algorithmus siehe unten.)

d) Geben Sie die Zugfolgen der beiden Kellerautomaten aus b) und c) an, die sie bei der Erkennung des Satzes

```
VAR a, b: INTEGER;
```

durchlaufen.

Algorithmus Kellerautomat aus Grammatik (nichtdeterministisch, *top-down*):
Der Kellerautomat besitzt nur einen einzigen Zustand *Z* (Start- und Endzustand), zu Beginn enthält der Keller nur das Satzsymbol *S* und erkennt Sätze durch leeren Keller.

S.1: Erzeuge für jede Regel $A \rightarrow \alpha$ einen Übergang $\delta(Z, \varepsilon, A) = (Z, \alpha^R)$.
Hierbei ist α^R die Umkehrung von α.

S.2: Erzeuge für jedes Terminalsymbol *a* einen Übergang $\delta(Z, a, a) = (Z, \varepsilon)$.

Algorithmus erweiterter Kellerautomat aus Grammatik (nichtdeterministisch, *bottom-up*):
Der erweiterte Kellerautomat besitzt zwei Zustände, *Z* und *R*. Dabei ist *R* ist Endzustand. Sein Keller enthält im Startzustand das nicht zur Grammatik gehörende Symbol *$*.

S.1: Erzeuge für jede Regel $A \rightarrow \alpha$ einen Übergang $\delta(Z, \varepsilon, \alpha) = (Z, A)$.

S.2: Erzeuge für jedes Terminalsymbol *a* einen Übergang $\delta(Z, a, x) = (Z, xa)$ für alle $x \in V \cup \{\$\}$.

S.3: Erzeuge den Übergang $\delta(Z, \varepsilon, \$S) = (R, \varepsilon)$.

5. Term. Anfänge/Nachfolger, LL(k)-Bedingung u. Transformation (2 + 2 + 1 Punkte)

Wir betrachten eine abgeänderte und vereinfachte Form von Modula-2-Programmoduln und beschreiben sie durch folgende Grammatik:

```
progmod  → MODULE id : priority ; imppart block id .
priority → const | ε
imppart  → FROM id IMPORT implist | IMPORT implist
implist  → id | id , implist
block    → dclpart statpart | statpart
dclpart  → DECL | DECL ; dclpart
statpart → BEGIN statseq ; END
statseq  → STAT | STAT ; statseq
```

a) Bestimmen Sie die terminalen Anfänge und Nachfolger der Länge 1 aller Nonterminalsymbole dieser Grammatik.

b) Ist diese Grammatik LL(*k*)? Wenn ja, wie groß ist *k*; wenn nein, warum nicht?

c) Transformieren Sie diese Grammatik in eine äquivalente LL(1)-Grammatik und zeigen Sie, dass Ihre Grammatik tatsächlich LL(1) ist.

1 OO-Implementierung endlicher Automaten

Dieser Abschnitt behandelt die Aufgabe 1 der dritten Übung.

Listing 1: AutomateUtil.hpp

```
1  //
2  // Created by Thomas on 11/21/16.
3  //
4
5  #ifndef AUTOMATAFORSTUDENTSV4_AUTOMATEUTIL_HPP
6  #define AUTOMATAFORSTUDENTSV4_AUTOMATEUTIL_HPP
7
8  #include <ctime>
9  #include <string>
10 #include "Grammar.h"
11 #include "NFA.h"
12 #include "DFA.h"
13
14 #define UNDEF_ENTRY  'u'
15 #define EQUIVALENT_ENTRY 'e'
16 #define NOT_EQUIVALENT_ENTRY  'n'
17
18 namespace AutomateUtil {
19
20     /**
21      * The type for the table entries used during minimal check
22      */
23     typedef char TableEntry;
24
25     /**
26      * Simple Clock for time measureing
27      */
28     struct Clock {
29         double duration = 0;
30         std::clock_t start;
31         std::clock_t end;
32
33         void startMeasure() {
34             start = end = std::clock();
35         }
36
37         void endMeasure() {
38             end = std::clock();
39             std::cout << "Duration: " << (duration = (end - start) / (double) CLOCKS_PER_SEC) <<
   ↪  std::endl;
40         }
41     };
42
43     /**
44      * Converts a grammar to a non-determenistic automat called NFA.
45      * @param grammar the grammar to build the NFA for
46      * @return the build nfa
47      * @throws invalid_argument if grammar is either nll or not a regular one.
48      */
49     NFA *nfaOfGrammar(const Grammar *grammar);
50
51     /**
52      * Creates the Grammar corresponding zu the given NFA.
53      * @param nfa the nfa instance to get grammar for
54      * @return the created grammar
55      * @throw invalid_argument if the nfa instance is null
56      */
```

```
57      Grammar *grammarOfNfa(const NFA *nfa);
58  }
59  #endif //AUTOMATAFORSTUDENTSV4_AUTOMATEUTIL_HPP
```

Listing 2: AutomateUtil.cpp

```
1   //
2   // Created by Thomas on 11/21/16.
3   //
4   #include "AutomateUtil.hpp"
5   #include <stdexcept>
6
7   using namespace std;
8
9   NFA *AutomateUtil::nfaOfGrammar(const Grammar *grammar) {
10      if ((grammar == nullptr) || (!grammar->isRegular())) {
11          throw invalid_argument("Grammar must be regular and not null");
12      } // if
13
14      NFA *nfa = new NFA();
15
16      // build state map = left side of grammar
17      StateSet stateSet;
18      for (const auto &item : grammar->vNt) {
19          stateSet.insert(item->name[0]);
20      } // for
21      nfa->S = stateSet;
22
23      // build symbol set = TSymbols
24      SymbolSet symbolSet;
25      for (const auto &item : grammar->vT) {
26          symbolSet.insert(item->name[0]);
27      } // for
28      nfa->V = symbolSet;
29
30      // build transitions
31      for (Grammar::Rule rule : grammar->rules) {
32          for (const auto &sequence : rule.second) {
33              // add transition to itself which is terminal state too if only TSymbol
34              if (sequence->size() == 1) {
35                  nfa->delta[rule.first->name[0]][((*(sequence->begin()))->name[0])].insert(
36                          rule.first->name[0]);
37              }
38                  // add transition to NTSymbol via TSymbol
39              else {
40                  nfa->delta[rule.first->name[0]][((*(sequence->begin()))->name[0])].insert(
41                          ((*(--sequence->end()))->name[0]));
42              } // if
43          } // for
44      } // for
45
46      return nfa;
47  } // NFA *AutomateUtil::nfaOfGrammar
48
49  Grammar *AutomateUtil::grammarOfNfa(const NFA *nfa) {
50      if (nfa == nullptr) {
51          throw invalid_argument("nfa  must not be null");
52      } // if
53
54      SymbolPool *sp = SymbolPool::getInstance();
55      Grammar *grammar = nullptr;
56
```

```
57      // iterator over all possible states
58      for (const auto &state : nfa->S) {
59          NTSymbol *ntSymbol = sp->ntSymbol(string(1, state));
60          // init grammar on root symbol
61          if (grammar == nullptr) {
62              grammar = new Grammar(ntSymbol);
63          }
64
65          // iterate over all symbols
66          for (const auto &symbol: nfa->V) {
67              // for all destination states for the source state and current symbol
68              for (const auto &destinationState: nfa->delta[state][symbol]) {
69                  Sequence *sequence = new Sequence();
70                  sequence->append(sp->tSymbol(string(1, symbol)));
71                  sequence->append(sp->ntSymbol(string(1, destinationState)));
72                  // add rule to grammar
73                  grammar->addRule(ntSymbol, sequence);
74              } // for
75          } // for
76      } // for
77
78      return grammar;
79  } // Grammra *AutomateUtil::grammarOfNfa
```

2 DFA und Mealy Automat

Dieser Abschnitt behandelt die Aufgabe 2 der dritten Übung. Die Konvertierung erfolgt über eine Funktion, die über einen Lambda-Ausdruck der Klasse übergeben werden kann. Die Methode *observeCurrentTapeSymbol* ist eine virtuelle Methode, die der Klasse *DFA* hinzugefügt wurde. In der Methode *accepts* wird beim Setzen des aktuellen Symbols die Beobachtermethode aufgerufen.

Listing 3: Mealy.hpp

```
1   //
2   // Created by Thomas on 11/10/16.
3   //
4
5   #ifndef AUTOMATAFORSTUDENTSV4_MEALY_HPP
6   #define AUTOMATAFORSTUDENTSV4_MEALY_HPP
7
8   #include "DFA.h"
9   #include "string.h"
10
11  using namespace std;
12
13  typedef std::function<char(char)> Mapper;
14
15  class Mealy : public DFA {
16
17  private:
18      std::function<char(char)> mapper;
19
20  protected:
21      inline void observeCurrentTapeSymbol(const TapeSymbol *symbol) const {
22          if (mapper != nullptr) {
23              cout << "translated '" << *symbol << "' -> '" << mapper(*symbol) << "'" << endl;
24          }
25      };
26
27  public:
```

```
28      Mealy(Mapper mapper) : mapper(mapper) {}
29  };
30
31
32  #endif //AUTOMATAFORSTUDENTSV4_MEALY_HPP
```

Listing 4: TestMealy.cpp

```
1   // DFATest.cpp:                                              HDO, 2006-2014
2   // -----------
3   // Test program for deterministic finite automata.
4   //=============================================================================
5
6   #include <cstdio>  // for getchar only
7   #include <cstdlib> // for system only
8
9   #include <iostream>
10  #include <map>
11  #include <set>
12  #include <string>
13  #include "Mealy.hpp"
14
15  using namespace std;
16
17  #include "SymbolStuff.h"
18  #include "StateStuff.h"
19  #include "FA.h"
20  #include "DFA.h"
21  #include "NFA.h"
22  #include "AutomateUtil.hpp"
23  #include "Grammar.h"
24
25  using namespace std;
26
27  int main(int argc, char **args) {
28      Mealy mealy([](char symbol) -> char {
29          switch (symbol) {
30              case 'b':
31                  return 'c';
32              case 'z':
33                  return 'd';
34              default:
35                  return symbol;
36          } // switch
37      });
38
39      mealy.S = StateSet("{B,R}");
40      mealy.V = SymbolSet("{b,z}");
41      mealy.delta['B']['b'] = 'R';
42      mealy.delta['R']['b'] = 'R';
43      mealy.delta['R']['z'] = 'R';
44      mealy.s1 = 'B';
45      mealy.F = StateSet("{R}");
46
47      cout << "Test Mealy. S={B,R}, V={b,z}" << endl;
48
49      string tape = "bzbz";
50      // current state B (=start), no visited states
51      cout << "Valid: mealy.s1=B, mealy.F={R}, tape=[b]" << endl
52           << "tape='" << tape << "'" << endl
53           << "accepted: " << ((mealy.accepts(tape.data())) ? "true" : "false") << endl
54           << "-------------------------------------------------" << endl;
```

```
55 } // int main
56
57 // end of DFATest.cpp
58 //========================================================================
```

```
Test Mealy. S={B,R}, V={b,z}
state = B, symbol = b
translated 'z' -> 'd'
state = R, symbol = z
translated 'b' -> 'c'
state = R, symbol = b
translated 'z' -> 'd'
state = R, symbol = z
translated ' ' -> ' '
Valid: mealy.s1=B, mealy.F={R}, tape=[b]
tape='bzbz'
accepted: true
-------------------------------------------------

ObjectCounter<8NTSymbol>: +0 -0 = 0 alive

ObjectCounter<7TSymbol>: +0 -0 = 0 alive

ObjectCounter<10SymbolPool>: +0 -0 = 0 alive

ObjectCounter<9SymbolSet>: +2 -2 = 0 alive

ObjectCounter<14SetOfStateSets>: +0 -0 = 0 alive

ObjectCounter<8StateSet>: +4 -4 = 0 alive

ObjectCounter<21StateSetToStateMapper>: +0 -0 = 0 alive

ObjectCounter<3DFA>: +1 -1 = 0 alive

ObjectCounter<3NFA>: +0 -0 = 0 alive

ObjectCounter<N10VocabularyI6SymbolE7XSymbolE>: +0 -0 = 0 alive

ObjectCounter<10VocabularyI6SymbolE>: +0 -0 = 0 alive

ObjectCounter<10VocabularyI7TSymbolE>: +0 -0 = 0 alive

ObjectCounter<10VocabularyI8NTSymbolE>: +0 -0 = 0 alive

ObjectCounter<7Grammar>: +0 -0 = 0 alive

ObjectCounter<11SequenceSet>: +0 -0 = 0 alive

ObjectCounter<8Sequence>: +0 -0 = 0 alive

Process finished with exit code 0
```

Abbildung 1: Tests für Mealy Automat

3 NFA → DFA Transformation und Zustandsminimierung

Dieser Abschnitt behandelt die Aufgabe 3 der dritten Übung.

Listing 5: TestNfaToDfa.cpp

```
// DFATest.cpp:                                                HDO, 2006-2014
// -----------
// Test program for deterministic finite automata.
//=============================================================================

#include <cstdio>  // for getchar only
#include <cstdlib> // for system only

#include <iostream>
#include <map>
#include <set>
#include <string>
#include "Mealy.hpp"

using namespace std;

#include "SymbolStuff.h"
#include "StateStuff.h"
#include "FA.h"
#include "DFA.h"
#include "NFA.h"
#include "AutomateUtil.hpp"
#include "Grammar.h"

using namespace std;


/**
 * Answers the question if the given DFA is a minimal one.
 * @param dfa the dfa to check
 * @return true if minimal false otherwise
 */
bool isMinimalAutomat(const DFA *dfa);

int main(int argc, char **args) {
    NFA nfa;
    AutomateUtil::Clock clock;

    nfa.S = StateSet("{S,A,B,C,R}");
    nfa.V = SymbolSet("{a,b,c}");

    nfa.delta['S']['a'] = StateSet("{S,A}");
    nfa.delta['S']['b'] = StateSet("{S,B}");
    nfa.delta['S']['c'] = StateSet("{S,C}");

    nfa.delta['A']['a'] = StateSet("{A,R}");
    nfa.delta['A']['b'] = StateSet("{A}");
    nfa.delta['A']['c'] = StateSet("{A}");

    nfa.delta['B']['a'] = StateSet("{B,R}");
    nfa.delta['B']['b'] = StateSet("{B}");
    nfa.delta['B']['c'] = StateSet("{B}");

    nfa.delta['C']['a'] = StateSet("{C,R}");
    nfa.delta['C']['b'] = StateSet("{C}");
    nfa.delta['C']['c'] = StateSet("{C}");

```

```
58      nfa.s1 = 'S';
59      nfa.F = StateSet("{R}");
60
61      cout << "Test NFA RGB. S={S,A,B,C,R}, V={a,b,c}" << endl;
62
63      string tape =
    ↪  "aaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbccccccccccccccccccccccccca";
64      tape = tape +
    ↪  "ccccccccccccccccccccccccccccbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
65      tape = tape +
    ↪  "ccccccccccccccccccccccccccccbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
66      tape = tape +
    ↪  "ccccccccccccccccccccccccccccbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
67      tape = tape +
    ↪  "ccccccccccccccccccccccccccccbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
68      tape = tape +
    ↪  "ccccccccccccccccccccccccccccbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
69      tape = tape +
    ↪  "aaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbccccccccccccccccccccccccca";
70      // current state B (=start), no visited states
71      cout << "Valid: nfa.s1=S, nfa.F={R}, tape=[aaabbbcccaba]" << endl
72           << "tape='" << tape << "'" << endl;
73      clock.startMeasure();
74      cout << "accepts() : " << ((nfa.accepts(tape.data())) ? "true" : "false") << endl;
75      clock.endMeasure();
76      clock.startMeasure();
77      cout << "accepts2(): " << ((nfa.accepts2(tape.data())) ? "true" : "false") << endl;
78      clock.endMeasure();
79      cout << "-----------------------------------------------------" << endl;
80
81      tape = "abc";
82      // current state B (=start), no visited states
83      cout << "Valid: nfa.s1=S, nfa.F={R}, tape=[aaabbbcccaba]" << endl
84           << "tape='" << tape << "'" << endl;
85      clock.startMeasure();
86      cout << "accepts() : " << ((nfa.accepts(tape.data())) ? "true" : "false") << endl;
87      clock.endMeasure();
88      clock.startMeasure();
89      cout << "accepts2(): " << ((nfa.accepts2(tape.data())) ? "true" : "false") << endl;
90      clock.endMeasure();
91      cout << "-----------------------------------------------------" << endl;
92
93      // Calculate deterministic automat
94      DFA dfa = nfa.dfaOf();
95      dfa.generateGraphVizFile("2c-automat.gv");
96      DFA minimizedDfa = dfa.minimalDfaOf();
97      minimizedDfa.generateGraphVizFile("2c-minimized-automat.gv");
98      cout << endl << "Is minimal DFA ? " << (isMinimalAutomat(&dfa) ? "true" : "false") << endl;
99  }
100
101 /**
102  * Valdiates the given automat if its a minimal one
103  * @param dfa the dfa to check
104  * @return true if minimal false otherwise.
105  */
106 bool isMinimalAutomat(const DFA *dfa) {
107     if (dfa == nullptr) {
108         throw invalid_argument("DFA must not be null");
109     } // if
110
111     // transform transition table
112     // http://www.fh-wedel.de/~si/vorlesungen/cb/LexikalischeAnalyse/DFAmin.html
113     // https://www.youtube.com/watch?v=mW6b9FgG768#t=388.966624
```

```
    // https://www.youtube.com/watch?v=rw4vSM2mxrQ#t=32.584981

    // init transformed table
    AutomateUtil::TableEntry transformed[STATES][STATES];
    for (const auto &fromState: dfa->S) {
        for (const auto &toState: dfa->S) {
            if (fromState == toState) {
                transformed[fromState][toState] = EQUIVALENT_ENTRY;
                break;
            } else if ((dfa->F.contains(fromState) && !dfa->F.contains(toState))
                       || (!dfa->F.contains(fromState) && dfa->F.contains(toState))) {
                transformed[fromState][toState] = NOT_EQUIVALENT_ENTRY;
                transformed[toState][fromState] = NOT_EQUIVALENT_ENTRY;
            } else {
                transformed[fromState][toState] = UNDEF_ENTRY;
                transformed[toState][fromState] = UNDEF_ENTRY;
            };
        } // for
    } // for

    int undefCount = -1;
    int oldUndefCount = undefCount;
    bool run = true;
    while (run) {
        for (const auto &fromState : dfa->S) {
            for (const auto &toState : dfa->S) {
                // ignore mirror side and mark self reference
                if (fromState == toState) {
                    break;
                }

                // skip if already marked or ignored
                if ((transformed[fromState][toState] != UNDEF_ENTRY)) {
                    continue;
                }
                    // determine if equivalent
                else {
                    for (const auto &symbol: dfa->V) {
                        State fromDestState = dfa->delta[fromState][symbol];
                        State toDestState = dfa->delta[toState][symbol];
                        // non existing transition on from or to
                        if (transformed[fromDestState][toDestState] == NOT_EQUIVALENT_ENTRY) {
                            transformed[fromState][toState] = NOT_EQUIVALENT_ENTRY;
                            transformed[toState][fromState] = NOT_EQUIVALENT_ENTRY;
                            break;
                        }
                    }
                }

                if (transformed[fromState][toState] == UNDEF_ENTRY) {
                    undefCount = undefCount + 1;
                }
            }
        }

        run = false;
        cout << endl;
        for (const auto &fromState: dfa->S) {
            cout << endl;
            for (const auto &toState: dfa->S) {
                AutomateUtil::TableEntry entry = transformed[fromState][toState];
                cout << entry << " ";
                if (fromState == toState) {
```

```
177                     break;
178                 }
179                 // rerun only if changes have been detected
180                 if ((entry == UNDEF_ENTRY) && ((oldUndefCount == -1) || (undefCount <
 ↪  oldUndefCount))) {
181                     run = true;
182                 }
183             } // for
184         } // for
185 
186         oldUndefCount = undefCount;
187         undefCount = 0;
188     } // while
189 
190     return oldUndefCount == 0;
191 } // bool isMinimalAutomat
192 
193 // end of DFATest.cpp
194 //============================================================================
```

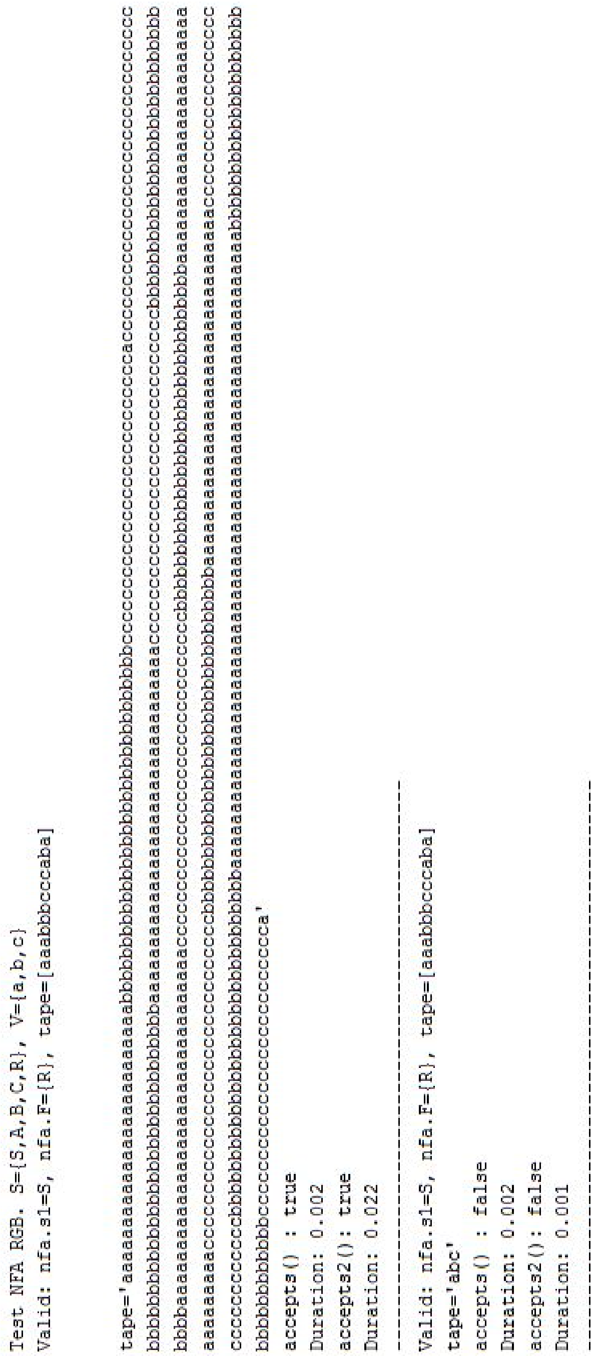


Abbildung 2: Tests für die beiden Accepts-Methoden

Es fällt auf, dass die Methode *accepts2* bei langen Bändern deutlich langsamer ist als die Methode *accepts*.

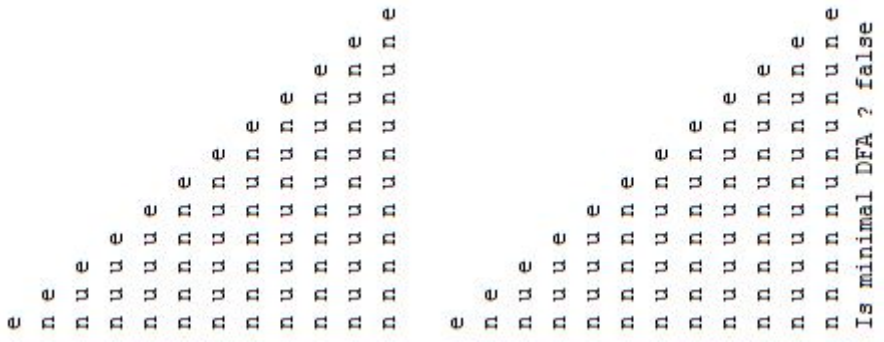


Abbildung 3: Tests für minimalen Automaten

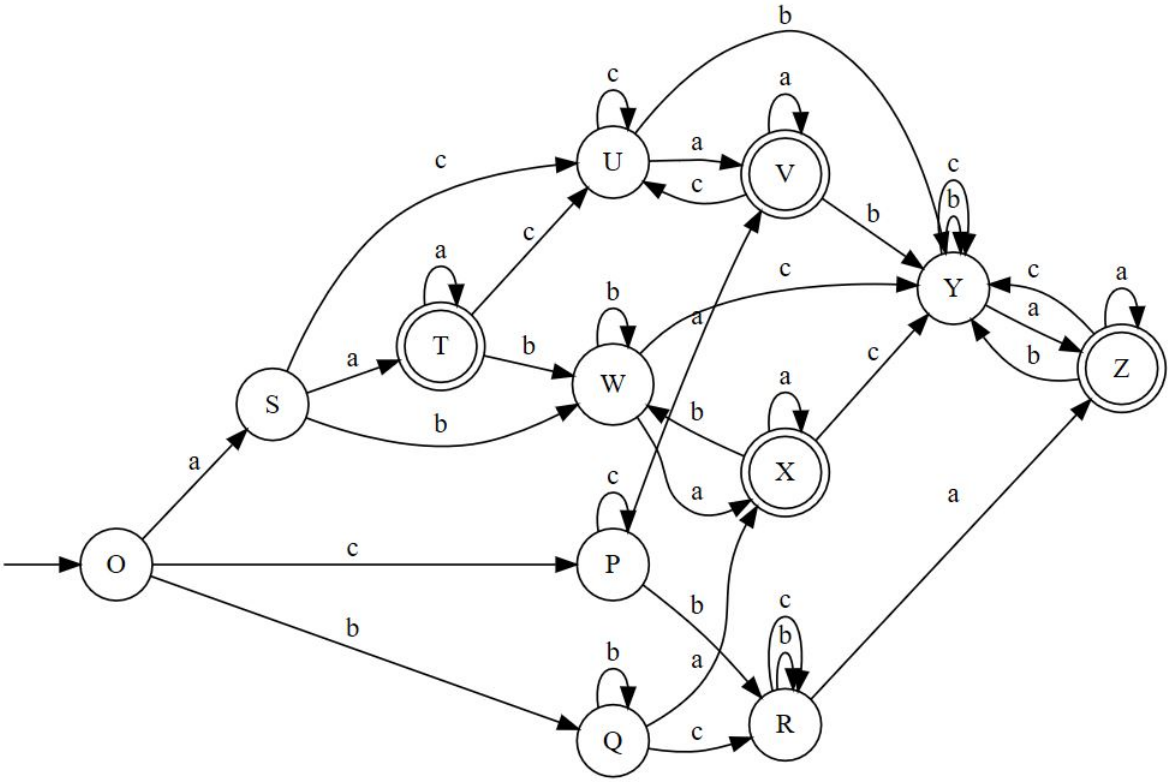


Abbildung 4: Nicht minimaler DFA generiert aus NFA

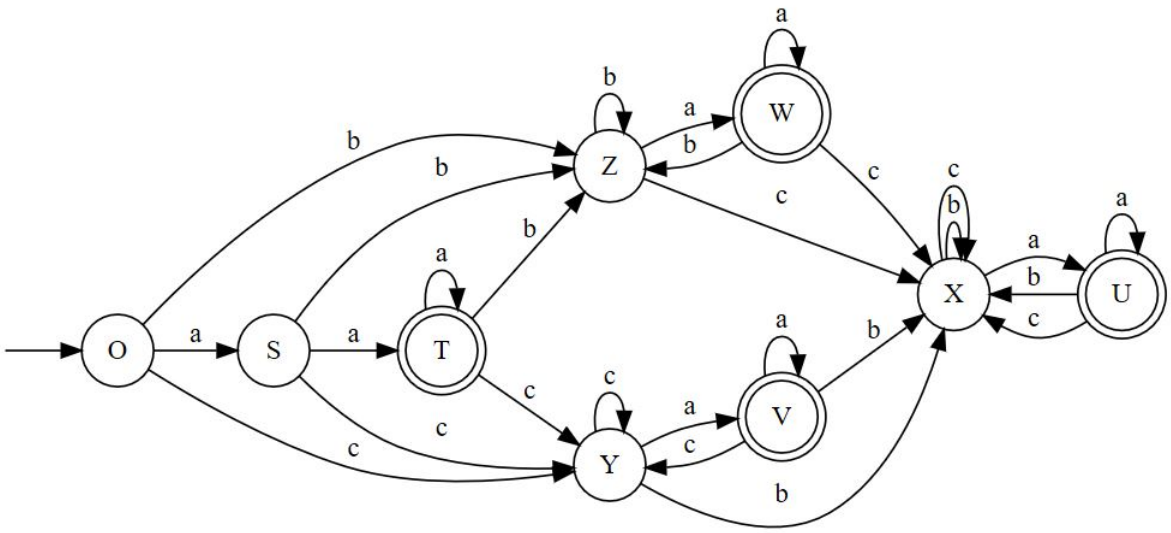


Abbildung 5: Minimaler DFA generiert aus NFA

4 Kellerautomat und erweiterter Kellerautomat

Dieser Abschnitt behandelt die Aufgabe 3 der dritten Übung.

Folgende Grammatik ist die gegebene Grammatik in der Schreibweise der formalen Sprachen.

Declaration	→	VAR | VarDeclList
VarDeclList	→	ϵ | VarDecl ';' VarDeclList
VarDecl	→	IdentList ':' Type
IdentList	→	ident | RepeatIdentList
RepeatIdentList	→	ϵ | ',' ident MoreIdentList
Type	→	OptArray TypeIdent
OptArray	→	ϵ | ARRAY '(' number ')' OF
TypeIdent	→	INTEGER | BOOLEAN | CHAR

Folgende Tabelle zeigt die Zustandsüberführungsfunktionen des Kellerautomaten.

$Zustand_P$	$Eingabe_P$	$Keller_P$	$Zustand_F$	$Keller_F$	$Schritt$
Declaration	VAR	-	E1	-	1
Declaration	ϵ	-	VarDeclList	-	2
VarDeclList	ϵ	-	E2	-	3
VarDeclList	ϵ	$2	VarDecl	$2	4
$2	;	-	VarDeclList	-	5
VarDeclList	ϵ	-	E2	-	6
VarDecl	ϵ	$3	IdentList	$3	7
$3	;	-	Type	-	8
Type	ϵ	$4	OptArray	$4	9
$4	ϵ	-	TypeIdent	-	10
TypeIdent	INTEGER	-	E3	-	11
TypeIdent	BOOLEAN	-	E3	-	12
TypeIdent	CHAR	-	E3	-	13
OptArray	ϵ	-	E4	-	14
OptArray	ARRAY	$5	-	$5	15
$5	(	$6	-	$6	16
$6	number	$7	-	$7	17
$7	)	$8	-	$8	18
$8	OF	-	E5	-	19

5 LL(k)-Bedingung und Transformation

Dieser Abschnitt behandelt die Aufgabe 4 der dritten Übung.

5.1 Terminale Anfänge und Nachfolger Länge 1

Folgende Tabelle zeigt die Ermittlung der terminalen Anfänge:

progmod	→ **MODULE** id : priority ; imppart block id	First(progmod)	$= \{MODULE\}$
priority	→ **const** | ϵ	First(priority)	$= \{const, \epsilon\}$
imppart	→ **FROM** id IMPORT implist | **IMPORT** implist	First(imppart)	$= \{FROM, IMPORT\}$
implist	→ **id** | **id** , implist	First(implist)	$= \{id\}$
block	→ **dclpart** statpart | statpart	First(block)	$= \{dclpart\}$
dclpart	→ **DECL** | **DECL** ; dclpart		$= \{DECL\}$
block	→ dclpart statpart | **statpart**		$= \{statpart\}$
statpart	→ **BEGIN** statseq ; END		$= \{BEGIN\}$
	→		$= \{BEGIN, DECL\}$
dclpart	→ **DECL** | **DECL** ; dclpart	First(dclpart)	$= \{DECL\}$
statpart	→ **BEGIN** statseq ; END	First(statseq)	$= \{BEGIN\}$
statseq	→ **STAT** | **STAT** ; statseq		$= \{STAT\}$

Folgende Tabelle zeigt die Ermittlung der terminalen Nachfolger:

$$
\begin{array}{lcl}
\text{Follow(progmod)} & = & \{\} \\
\text{Follow(priority)} & = & \{;\} \\
\text{Follow(imppart)} & = & \{\} \cup First(block) \\
 & = & \{DECL, BEGIN\} \\
\text{Follow(implist)} & = & \{\} \cup Follow(imppart) \\
 & = & \{DECL, BEGIN\} \\
\text{Follow(block)} & = & \{id\} \\
\text{Follow(dclpart)} & = & \{\} \cup First(statpart) \\
 & = & \{BEGIN\} \\
\text{Follow(statpart)} & = & \{\} \cup Follow(block) \\
 & = & \{id\} \\
\text{Follow(statseq)} & = & \{;\}
\end{array}
$$

5.2 Ist eine LL(k) Grammatik

Es handelt sich hierbei um eine LL(k)-Grammatik wobei $k > 0$, da es Regeln gibt die Optionen haben. Es handelt sich um eine LL(3)-Grammatik da es maximal 3 *Lookahead* benötigt, damit auf ein Terminalsymbol abgeleitet werden kann.

5.3 Transformation zu einer LL(1)-Grammatik

progmod	→	MODULE id : priority ; imppart block id .
priority	→	const | ϵ
imppart	→	FROM id IMPORT implist | IMPORT implist
implist	→	id implistRest
implistRest	→	, id implrest | ϵ
block	→	dclpart statpart | statprt
dclpart	→	DECL | DECL ; dclpart
statpart	→	BEGIN statseq ; END
statseq	→	STAT statseqList
statseqList	→	; statseqList | ϵ