FCW 1x	**Übung zu Formale Sprachen, Compiler- und Werkzeugbau 1**	**WS 2024/25, Übung 3**

Abgabetermin: in der KW 48

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

Name Yvonne Marneth **Aufwand in h** 20h

Punkte ________ **Übungsleiter** ______

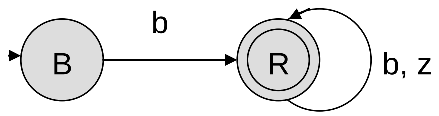
1. Objektorientierte Implementierung endlicher Automaten (6 Punkte)

Machen Sie sich zuerst mit der oo Implementierung endlicher Automaten in C++ vertraut. Sie finden diese im *moodle*-Kurs in der Datei *FiniteAutomataForStudents*, im Wesentlichen in den beiden Klassen *DFA* und *NFA*: Studieren Sie die Quelltexte anhand des Testprogramms in *Main.cpp*.

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

2. DFA, Erkennung und Mealy- oder Moore-Automat (1 + 3 Punkte)

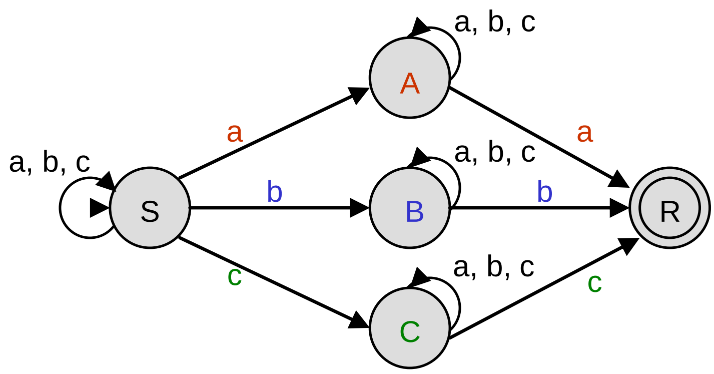
a) Schreiben Sie ein Programm, das den unten dargestellten Automaten für einfache Bezeichner erzeugt (in Form eines Objekts der Klasse *DFA*) und testen Sie die *accepts*-Methode sowohl für gültige als auch für ungültige Bandinhalte.



b) Entwickeln Sie ausgehend von der Klasse *DFA* eine neue Klasse (*Mealy* oder *Moore*), die einen endlichen Transformationsautomaten (nach George H. Mealy oder Edward F. Moore) simuliert. Testen Sie Ihre Klasse, indem Sie Bezeichner gemäß des obigen Automaten (*b* steht für *Buchstabe*, *z* steht für *Ziffer*, z. B. *bzzb*) in's Englische übersetzen (*c* für *character* und *d* für *digit*), also damit z.B. den dt. Bezeichner *bzzb* in den engl. Bezeichner *cddc* übersetzen.

3. NFA, Transformation NFA -> DFA und Zustandsminimierung (1 + 2 + 1 + 1 Punkte)

a) Schreiben Sie ein Programm, das den unten dargestellten Automaten für spezielle *abc*-Folgen in Form eines Objekts der Klasse *NFA* erzeugt, und versuchen Sie mit den drei Methoden *accepts1* (mit Multithreading), *accepts2* (mit Backtracking) und *accepts3* (durch Verfolgung von Zustandsmengen) sowohl gültige als auch ungültige Bandinhalte zu erkennen.



b) Instrumentieren Sie die drei *acceptsX*-Methoden so, dass Sie Maßzahlen für den Zeitaufwand und die Laufzeit (mittels *Timer.h* + *.cpp*) der Erkennung ermitteln können.

c) Berechnen Sie mit der Methode *NFA::dfaOf* den deterministischen Automaten für obigen nichtdeterministischen Automaten und stellen Sie diesen graphisch 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)

Hier die Grammatik für die Definition von Konstanten in der Sprache MiniC++ (einer kleinen Teilmenge von C++):

```
ConstDef =  'const' Type ident Init { ',' ident Init } ';' .
Type =      'bool' | 'int' .
Init =      '=' ( false | true | [ '+' | '-' ] number ) .
```

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 e*rweiterten 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

```
const int max = 100;
```

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 (1 + 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 der Länge 1 ($First_1$) aller Alternativen und terminalen Nachfolger der Länge 1 ($Follow_1$) 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 dann, dass Ihre Grammatik tatsächlich LL(1) ist.

Aufgabe 1

Umwandeln einer regulären Grammatik in einen NFA oder DFA

Die Funktion `*faOf(const Grammar *g)` konvertiert eine reguläre Grammatik in einen nichtdeterministischen endlichen Automaten (NFA). Dieser Ansatz wurde gewählt, weil ein NFA auch in einen DFA transformiert werden kann.

1. Die Funktion setzt das Startsymbol der Grammatik als Startzustand des NFAs.
2. Für jede Produktionsregel der Grammatik, die ein Nichtterminalsymbol enthält, wird für das Startsymbol und das erste Symbol der Regel ein Übergang im NFA erstellt. Wenn das Ziel der Regel ein Nichtterminal ist, wird ein Übergang zwischen den entsprechenden Zuständen hinzugefügt.
3. Wenn eine Regel mit einem terminalen Symbol endet, wird der aktuelle Zustand als Endzustand im NFA hinzugefügt.

Am Ende wird der NFA durch den FABuilder erstellt und zurückgegeben.

Einschränkungen:

- Die Grammatik darf nur regulären Sequenzen beinhalten

Sinnvoll wäre noch eine Behandlung von Epsilon gewesen, leider bin ich zu spät darauf gekommen und habe keine Zeit mehr das auch zu implementieren.

```
bool isRegularSequence(const Sequence &seq) {
    const auto length = seq.length();
    return length > 0 && length < 3 && seq.symbolAt(0)->isT() &&
seq.symbolAt(length - 1)->isT();
}

NFA *faOf(const Grammar *g) {
    FABuilder builder;

    // Step 1: Set the start state of the NFA based on the root of the
grammar
    const auto startStateSymbol = g->root;
    builder.setStartState(startStateSymbol->name);

    for (const auto &[key, value]: g->rules) {
        auto const &nt = key;
        auto const &sequenceSet = value;

        for (const Sequence *seq: sequenceSet) {
            if (!isRegularSequence(*seq)) {
                throw runtime_error("Grammar is not regular");
            }

            const State srcState(nt->name);
            const TapeSymbol tapeSymbol = seq->symbolAt(0)->name[0];

            if (const auto targetSymbol = seq->back(); targetSymbol->isNT())
{
                // Step 2: Add transitions (delta functions) based on the
production rules of the grammar
                const State targetState(targetSymbol->name);
                builder.addTransition(srcState, tapeSymbol, targetState);
            } else {
                // Step 3: Add final states for rules that produce terminal
strings
                builder.addFinalState(nt->name);
            }
        }
    }

    return builder.buildNFA();
}
```

Getestet wurde die Funktion mit der folgenden Grammatik, die auch in der bereits vorhandenen Testfunktion `testNFAFromGrammar` verwendet wird:

```
void testNFAFromGrammar() {
    cout << "3. NFA from Grammar" << endl;
    cout << "--------------------" << endl;
    cout << endl;

    const GrammarBuilder gb4(
        "G(1):                          \n\
                 1 -> a 2 | b 1            \n\
                 2 -> a 2 | b 1 | b 3 | a \n\
                 3 -> a 2 | b 4            \n\
                 4 -> a 4 | b 4 | b | a     ");

    const auto *grammar = gb4.buildGrammar();

    cout << "Creating NFA from Grammar..." << endl;
    const unique_ptr<NFA> nfa(faOf(grammar));

    cout << "nfaFromGrammar:" << endl << *nfa;
    vizualizeFA("nfaFromGrammar", nfa.get());
}
```

und produziert die folgende Ausgabe

```
nfaFromGrammar:
->     1 -> a 2 | b 1
   () 2 -> a 2 | b 1 | b 3
      3 -> a 2 | b 4
   () 4 -> a 4 | b 4
```

und Darstellung

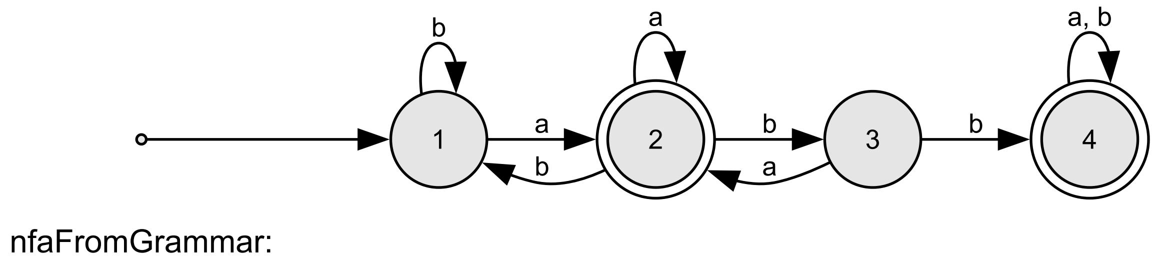


Figure 1: NFA from Grammar

Umgekehrte Transformation von NFA in eine reguläre Grammatik

Die Funktion `Grammar *grammarOf(const NFA *nfa)` konvertiert einen nichtdeterministischen endlichen Automaten (NFA) zurück in eine Grammatik. Hier ist die Zusammenfassung der Schritte:

1. Der Startzustand des NFAs wird als Wurzelsymbol der Grammatik gesetzt und ein GrammarBuilder initialisiert.
2. Für jeden Zustand im NFA und seine Übergänge werden addProductionRules Produktionsregeln für die Grammatik generiert. Diese Regeln basieren auf den Übergängen im NFA.
3. Wenn ein Zustand sowohl ein Endzustand des NFAs ist als auch der Quellzustand eines Übergangs einem Zielzustand entspricht, wird eine zusätzliche Produktionsregel hinzugefügt, die nur das Terminalsymbol enthält, um das Ende der Produktion zu kennzeichnen.

Zum Schluss wird die Grammatik mit dem GrammarBuilder erzeugt und zurückgegeben.

```
void addProductionRules(GrammarBuilder *builder,
                        SymbolPool &sp,
                        const std::string &srcState,
                        const std::string &tapeSymbol,
                        const std::set<std::string> &destStates,
                        const std::set<std::string> &finalStates) {
    auto *srcSymbol = sp.ntSymbol(srcState);
    auto *terminalSymbol = sp.tSymbol(tapeSymbol);

    for (const auto &destState: destStates) {
        auto *destSymbol = sp.ntSymbol(destState);

        auto *sequence = new Sequence();
        sequence->append(terminalSymbol);
        sequence->append(destSymbol);

        builder->addRule(srcSymbol, sequence);

        if (finalStates.find(srcState) != finalStates.end() &&
srcState.at(0) == destState.at(0)) {
            builder->addRule(srcSymbol, new Sequence(terminalSymbol));
        }
    }
}

Grammar *grammarOf(const NFA *nfa) {
    SymbolPool sp;

    auto *rootSymbol = sp.ntSymbol(nfa->s1);
    const auto builder = std::make_unique<GrammarBuilder>(rootSymbol);

    for (const auto &[srcState, transitions]: nfa->delta) {
        for (const auto &[tapeSymbol, destStates]: transitions) {
            addProductionRules(builder.get(), sp, srcState, std::string(1,
tapeSymbol), destStates, nfa->F);
        }
    }

    return builder->buildGrammar();
}
```

Überprüfen lässt sich das Ergebnis, indem aus einer Grammatik G(S) ein NFA mit der Funktion `*faOf(const Grammar *g)` in einen NFA umgewandelt und anschließend mit `Grammar *grammarOf(const NFA *nfa)` wieder in eine Grammatik transformiert wird. Die resultierende Grammatik sollte der ursprünglichen Grammatik entsprechen.

Getestet wurde die Funktion daher mit der folgenden Grammatik, die auch in der bereits vorhandenen Testfunktion `testNFAFromGrammar` verwendet wird:

```
void testGrammarOfNFA() {
    cout << "4. Grammar from NFA" << endl;
    cout << "--------------------" << endl;
    cout << endl;

    const auto fab = make_unique<FABuilder>(
        "-> 1 -> a 2 | b 1         \n\
         () 2 -> a 2 | b 1 | b 3  \n\
            3 -> a 2 | b 4        \n\
         () 4 -> a 4 | b 4");

    const unique_ptr<NFA> nfa(fab->buildNFA());

    cout << "Creating Grammar from NFA..." << endl;
    const unique_ptr<Grammar> grammar(grammarOf(nfa.get()));

    cout << "grammarOfNFA:" << endl << *grammar;
}
```

und produziert die folgende Ausgabe, welche der ursprünglichen Grammatik entspricht:

```
grammarOfNFA:

G(1):
1 -> a 2 | b 1
2 -> a | a 2 | b 1 | b 3
3 -> a 2 | b 4
4 -> a | a 4 | b | b 4
---
VNt = { 1, 2, 3, 4 }, deletable: {  }
VT  = { a, b }
```

Aufgabe 2

a)

Implementation

```
    FABuilder builder;
    builder.setStartState("B")
            .addFinalState("R")
            .addTransition("B", 'b', "R")
            .addTransition("R", 'b', "R")
            .addTransition("R", 'z', "R");

    const unique_ptr<DFA> dfa(builder.buildDFA());

    cout << "dfa:" << endl << *dfa;
    vizualizeFA("dfa", dfa.get());

    auto testInput = [&](const string &input) {
        cout << "dfa->accepts(\"" << input << "\") => ";
        if (dfa->accepts(input)) {
            cout << " (accepted)" << endl;
        } else {
            cout << " (rejected)" << endl;
        }
    };

    testInput("b");
    testInput("bzb");
    testInput("bbb");
    testInput("bbzbb");
    testInput("bzzb");
    testInput("z");
    testInput("bbba");
```

Ergebnis

```
dfa->accepts("b") =>  (accepted)
dfa->accepts("bzb") =>  (accepted)
dfa->accepts("bbb") =>  (accepted)
dfa->accepts("bbzbb") =>  (accepted)
dfa->accepts("bzzb") =>  (accepted)
dfa->accepts("z") =>  (rejected)
dfa->accepts("bbba") =>  (rejected)
```

b)

Ein Mealy-Automat ist für diese Aufgabe wahrscheinlich besser geeignet als ein Moore-Automat, da er die Ausgaben basierend auf der aktuellen Kombination aus Zustand und Eingabesymbol generiert. Auf diese Weise kann auch das Bandsymbol in die Übersetzung mit einbezogen werden. Ein Moore-Automat müsste für jede Ausgabe einen eigenen Zustand definieren, was zu einem größeren Automaten mit höherer Komplexität führen würde.

Implementierung eines Mealy-Automaten

Der Konstruktor eines Mealy-Automaten initialisiert den Automaten mit einer Lambda-Funktion, die die Ausgaben direkt abhängig von der Kombination aus aktuellem Zustand und Eingabesymbol definiert.

Die `accepts`-Methode überprüft, ob eine Eingabe akzeptiert wird, indem sie Ausgaben nicht nur vom Zustand, sondern auch vom gerade gelesenen `TapeSymbol` ableiten können. Die Methode liest das aktuelle `TapeSymbol` und bestimmt die entsprechende Ausgabe anhand der Lambda-Funktion. Dann geht sie zum nächsten Zustand über, der durch die Übergangsfunktion definiert ist.

```
MealyDFA::MealyDFA(const StateSet &S, const TapeSymbolSet &V,
         const State &s1, const StateSet &F,
         const DDelta &delta,
         const map<pair<State, TapeSymbol>, char> &mealyLambda)
    : DFA(S, V, s1, F, delta), lambda(mealyLambda) {
}

bool MealyDFA::accepts(const Tape &tape) const {
    int i = 0;
    TapeSymbol tSy = tape[i];
    State s = s1;

    std::cout << lambda.at(make_pair(s, tSy));

    while (tSy != eot) {
        s = delta[s][tSy];

        if (!defined(s)) {
            return false;
        }

        i++;
        tSy = tape[i];

        if (tSy != eot) {
            std::cout << lambda.at(make_pair(s, tSy));
        }
    }

    return F.contains(s);
}
```

Erweiterung des FABuilders

Die Methode `setMealyLambda` erlaubt das Festlegen von Paaren aus Zuständen und Symbolen und den dazugehörigen Ausgabesymbolen. Sie stellt außerdem sicher, dass die Zustände und Symbole im Automaten gültig sind. Die Methode `buildMealyDFA` erzeugt eine Instanz eines `MealyDFA` und prüft dabei, ob die Lambda-Funktion definiert ist und der Automat deterministisch ist.

```
FABuilder &FABuilder::setMealyLambda(const
std::initializer_list<std::pair<std::pair<State, TapeSymbol>, char> > il) {
    for (const auto &[key, value]: il) {
        const auto &stateSymbolPair = key;
        const State &state = stateSymbolPair.first;
        const TapeSymbol &symbol = stateSymbolPair.second;
        const char &output = value;

        if (S.find(state) == S.end()) {
            throw runtime_error("State " + state + " not found in the
automaton");
        }

        if (V.find(symbol) == V.end()) {
            throw runtime_error("Symbol " + string(1, symbol) + " not found
in the alphabet");
        }

        mealyLambda[stateSymbolPair] = output;
    }

    return *this;
}

MealyDFA *FABuilder::buildMealyDFA() const {
    if (mealyLambda.empty())
        throw domain_error("cannot build MealyDFA, no lambda function set");
    if (!representsDFA())
        throw domain_error("cannot build DFA, builder's delta represents a
NFA");
    checkStates();
    return new MealyDFA(S, V, s1, F, dDeltaOf(delta), mealyLambda);
}
```

Test

```
    FABuilder builder;

    builder.setStartState("B")
            .addFinalState("R")
            .addTransition("B", 'b', "R")
            .addTransition("R", 'b', "R")
            .addTransition("R", 'z', "R")
            .setMealyLambda({
                {{"B", 'b'}, 'c'},
                {{"B", 'z'}, 'd'},
                {{"R", 'b'}, 'c'},
                {{"R", 'z'}, 'd'}
            });

    const std::unique_ptr<MealyDFA> mealyDfa(builder.buildMealyDFA());

    std::cout << "Eingabeband: bzzb" << std::endl;
    std::cout << "Ausgabeband: ";

    if (mealyDfa->accepts("bzzb")) {
        cout << endl;
        std::cout << "Eingabe akzeptiert!" << std::endl;
    } else {
        cout << endl;
        std::cout << "Eingabe nicht akzeptiert!" << std::endl;
    }

    cout << endl;
```

Ergebnis

```
Eingabeband: bzzb
Ausgabeband: cddc
Eingabe akzeptiert!
```

Aufgabe 3

a)

`accepts1`

Diese Methode verwendet Multithreading, um Nichtdeterminismus zu simulieren. Sie erstellt für jede mögliche Transition einen neuen Thread. Wenn ein Thread am Ende des Bands einen Endzustand erreicht, wird eine gemeinsame Variable `accepted` auf `true` gesetzt. Diese Variable wird geschützt, um Threadsicherheit zu gewährleisten. Bevor die Funktion beendet wird, werden alle Threads zusammengeführt (joined).

`accepts2`

Diese Methode verwendet rekursives Backtracking, um Nichtdeterminismus zu simulieren. Ausgehend vom Anfangszustand werden alle möglichen Transitionen entlang des Bandes rekursiv erkundet. Wenn ein Pfad am Ende des Bandes einen Endzustand erreicht, gibt die Funktion true zurück. Andernfalls wird zurückgegangen und andere Pfade werden untersucht. Dieser Ansatz ist einfacher und speichereffizienter als Multithreading, könnte aber bei sehr tiefer Rekursion zu einem Stackoverflow führen.

`accepts3`

Diese Methode simuliert Nichtdeterminismus, indem sie Mengen von Zuständen verfolgt, anstatt jeden Pfad einzeln zu erkunden. Ausgehend vom Anfangszustand wird iterativ die Menge aller möglichen Zielzustände für jedes Bandsymbol berechnet. Der Prozess wird bis zum Ende des Bands fortgesetzt. Wenn die endgültige Zustandsmenge eine Schnittmenge mit der Menge der Endzustände bildet, wird die Eingabe akzeptiert.

NFA Automaton

Aus dem gegebenen Automaten lässt sich die folgende Implementierung ableiten:

```
FABuilder builder;
builder.setStartState("S")
        .addFinalState("R")
        .addTransition("S", 'a', "S")
        .addTransition("S", 'b', "S")
        .addTransition("S", 'c', "S")
        .addTransition("S", 'a', "A")
        .addTransition("S", 'b', "B")
        .addTransition("S", 'c', "C")
        .addTransition("A", 'a', "A")
        .addTransition("A", 'b', "A")
        .addTransition("A", 'c', "A")
        .addTransition("A", 'a', "R")
        .addTransition("B", 'a', "B")
        .addTransition("B", 'b', "B")
        .addTransition("B", 'c', "B")
        .addTransition("B", 'b', "R")
        .addTransition("C", 'a', "C")
        .addTransition("C", 'b', "C")
        .addTransition("C", 'c', "C")
        .addTransition("C", 'c', "R");

const unique_ptr<NFA> nfa(builder.buildNFA());
```

wodurch folgender Automat entsteht:

```
nfa:
->    S -> a A | a S | b B | b S | c C | c S
      A -> a A | a R | b A | c A
      B -> a B | b B | b R | c B
      C -> a C | b C | c C | c R
   () R
```

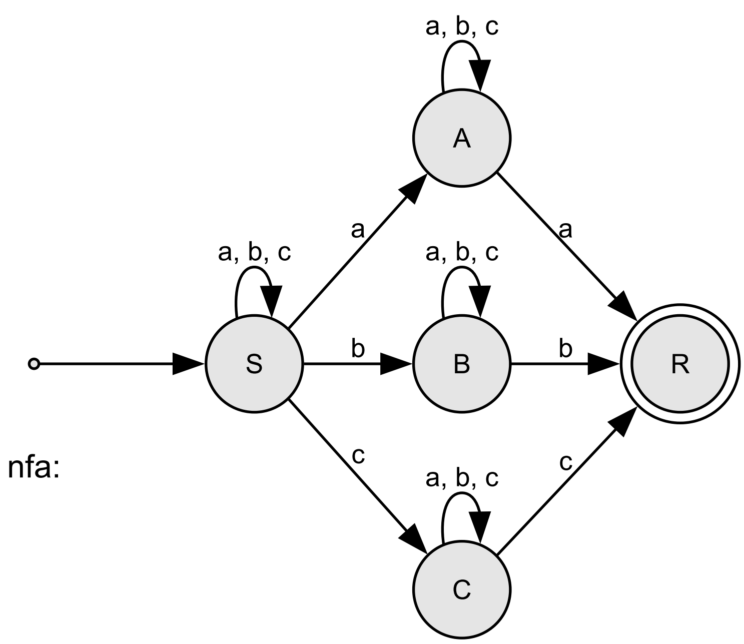


Figure 2: Visualized Automaton

Tests

Um mehrere Testwerte effizient und konsistent zu testen, können wir eine Liste von Teststrings und eine wiederverwendbare Lambda-Funktion `testMethod` definieren, um jede der akzeptierten Methoden auszuwerten.

Die Teststrings werden iteriert und die angegebene Methode wird auf jeden String angewendet, wobei auf der Konsole ausgegeben wird, ob der NFA ihn akzeptiert oder ablehnt.

```
    vector<string> testStrings = {
        "a", "b", "c",
        "ab", "abc", "abca", "abcb", "abcc",
        "aaaabbbbcccc", "abcabcabcabc", "aabbccbbcca",
        "aaaaaaaa", "bbbbbbbb", "cccccccc",
        "xyz"
    };

    auto testMethod = [&](const string &methodName, auto acceptsMethod) {
        cout << "Testing " << methodName << ":" << endl;
        for (const auto &input: testStrings) {
            cout << "nfa->" << methodName << "(\"" << input << "\") =>";
            if (acceptsMethod(input)) {
                cout << " (accepted)" << endl;
            } else {
                cout << " (rejected)" << endl;
            }
        }
        cout << endl;
    };

    testMethod("accepts1", [&](const string &input) { return nfa-
>accepts1(input); });
    testMethod("accepts2", [&](const string &input) { return nfa-
>accepts2(input); });
    testMethod("accepts3", [&](const string &input) { return nfa-
>accepts3(input); });
```

Testergebnisse

```
Testing accepts1:
nfa->accepts1("a") => (rejected)
nfa->accepts1("ab") => (rejected)
nfa->accepts1("abc") => (rejected)
nfa->accepts1("abca") => (accepted)
nfa->accepts1("abcb") => (accepted)
nfa->accepts1("abcc") => (accepted)
nfa->accepts1("aaaabbbbcccc") => (accepted)
nfa->accepts1("abcabcabcabc") => (accepted)
nfa->accepts1("aabbccbbcca") => (accepted)
nfa->accepts1("aaaaaaaa") => (accepted)
nfa->accepts1("bbbbbbbb") => (accepted)
nfa->accepts1("cccccccc") => (accepted)
nfa->accepts1("xyz") => (rejected)
```

```
Testing accepts2:
nfa->accepts2("a") => (rejected)
nfa->accepts2("ab") => (rejected)
nfa->accepts2("abc") => (rejected)
nfa->accepts2("abca") => (accepted)
nfa->accepts2("abcb") => (accepted)
nfa->accepts2("abcc") => (accepted)
nfa->accepts2("aaaabbbbcccc") => (accepted)
nfa->accepts2("abcabcabcabc") => (accepted)
nfa->accepts2("aabbccbbcca") => (accepted)
nfa->accepts2("aaaaaaaa") => (accepted)
nfa->accepts2("bbbbbbbb") => (accepted)
nfa->accepts2("cccccccc") => (accepted)
nfa->accepts2("xyz") => (rejected)
```

```
Testing accepts3:
nfa->accepts3("a") => (rejected)
nfa->accepts3("ab") => (rejected)
nfa->accepts3("abc") => (rejected)
nfa->accepts3("abca") => (accepted)
nfa->accepts3("abcb") => (accepted)
nfa->accepts3("abcc") => (accepted)
nfa->accepts3("aaaabbbbcccc") => (accepted)
nfa->accepts3("abcabcabcabc") => (accepted)
nfa->accepts3("aabbccbbcca") => (accepted)
nfa->accepts3("aaaaaaaa") => (accepted)
nfa->accepts3("bbbbbbbb") => (accepted)
nfa->accepts3("cccccccc") => (accepted)
nfa->accepts3("xyz") => (rejected)
```

Wichtig ist, dass jede `accept`-Implementierung das gleiche Ergebnis für gleiche Teststrings liefert und die Ergebnisse zwischen den getesteten Funktionen so konsistent untereinander ist.

b)

Um die Performance der `accept`-Implementierungen zu vergleichen, habe ich meine `testFunction` so modifiziert, sodass sie die Zeit jeder Auswertung erfasst:

```
    auto testMethod = [&](const string &methodName, auto acceptsMethod) {
        cout << "Testing " << methodName << ":" << endl;
        for (const auto &input: testStrings) {
            cout << "nfa->" << methodName << "(\"" << input << "\") =>";

            startTimer();
            const auto result = acceptsMethod(input);
            stopTimer();

            if (acceptsMethod(input)) {
                cout << " (accepted) - " << elapsedTime() << endl;
            } else {
                cout << " (rejected) - " << elapsedTime() << endl;
            }
        }
        cout << endl;
    };
```

Zunächst resultiert das in einem eher weniger aussagekräftigen Ergebnis:

```
Testing accepts1:
nfa->accepts1("a") => (rejected) - 0ms
nfa->accepts1("ab") => (rejected) - 0ms
nfa->accepts1("abc") => (rejected) - 0ms
...
```

```
Testing accepts2:
nfa->accepts2("a") => (rejected) - 0ms
nfa->accepts2("ab") => (rejected) - 0ms
nfa->accepts2("abc") => (rejected) - 0ms
...
```

```
Testing accepts3:
nfa->accepts3("a") => (rejected) - 0ms
nfa->accepts3("ab") => (rejected) - 0ms
nfa->accepts3("abc") => (rejected) - 0ms
...
```

Die Ausgabe schien auf Millisekunden-Niveau zu ungenau zu sein, um die Implementierungen miteinander vergleichen zu können. Um dieses Problem zu lösen, habe ich zwei Varianten ausprobiert:

Messgenauigkeit erhöhen auf Nanosekunden

Um die Messgenauigkeit zu erhöhen, habe ich die Berechnung der vergangenen Zeit angepasst:

```
double elapsedTime() {
  return chrono::duration_cast<chrono::nanoseconds>(stop_tp -
start_tp).count() / 1e6;
}
```

Das neue Ergebnis:

```
Testing accepts1:
nfa->accepts1("a") => (rejected) - 0.2695ms
nfa->accepts1("ab") => (rejected) - 0.3732ms
nfa->accepts1("ac") => (rejected) - 0.3339ms
nfa->accepts1("abc") => (rejected) - 0.5803ms
nfa->accepts1("abca") => (accepted) - 0.8879ms
nfa->accepts1("abcb") => (accepted) - 0.8665ms
nfa->accepts1("abcc") => (accepted) - 0.896ms
nfa->accepts1("aaaabbbbcccc") => (accepted) - 6.3958ms
nfa->accepts1("abcabcabcabc") => (accepted) - 6.1135ms
nfa->accepts1("aabbccbbcca") => (accepted) - 5.2142ms
nfa->accepts1("aaaaaaaa") => (accepted) - 3.5449ms
nfa->accepts1("bbbbbbbb") => (accepted) - 3.7402ms
nfa->accepts1("cccccccc") => (accepted) - 3.8015ms
nfa->accepts1("xyz") => (rejected) - 0.0069ms
```

```
Testing accepts2:
nfa->accepts2("a") => (rejected) - 0.0109ms
nfa->accepts2("ab") => (rejected) - 0.0173ms
nfa->accepts2("abc") => (rejected) - 0.0136ms
nfa->accepts2("abca") => (accepted) - 0.0041ms
nfa->accepts2("abcb") => (accepted) - 0.0057ms
nfa->accepts2("abcc") => (accepted) - 0.0164ms
nfa->accepts2("aaaabbbbcccc") => (accepted) - 0.0396ms
nfa->accepts2("abcabcabcabc") => (accepted) - 0.0251ms
nfa->accepts2("aabbccbbcca") => (accepted) - 0.0064ms
nfa->accepts2("aaaaaaaa") => (accepted) - 0.011ms
nfa->accepts2("bbbbbbbb") => (accepted) - 0.0056ms
nfa->accepts2("cccccccc") => (accepted) - 0.0109ms
nfa->accepts2("xyz") => (rejected) - 0.0127ms
```

```
Testing accepts3:
nfa->accepts3("a") => (rejected) - 0.0546ms
nfa->accepts3("ab") => (rejected) - 0.0397ms
nfa->accepts3("abc") => (rejected) - 0.0472ms
nfa->accepts3("abca") => (accepted) - 0.0443ms
nfa->accepts3("abcb") => (accepted) - 0.0476ms
nfa->accepts3("abcc") => (accepted) - 0.0507ms
nfa->accepts3("aaaabbbbcccc") => (accepted) - 0.0763ms
nfa->accepts3("abcabcabcabc") => (accepted) - 0.0861ms
nfa->accepts3("aabbccbbcca") => (accepted) - 0.131ms
nfa->accepts3("aaaaaaaa") => (accepted) - 0.0634ms
nfa->accepts3("bbbbbbbb") => (accepted) - 0.0476ms
nfa->accepts3("cccccccc") => (accepted) - 0.0464ms
nfa->accepts3("xyz") => (rejected) - 0.0248ms
```

Auf diese Weise sind die Werte vergleichbarer.

Auswertung und weitere Möglichkeiten

Alternativ könnte man auch die Durchschnittszeit für jeden Teststring für viele Iterationen berechnet werden. Ein Vergleich lässt sich aber mit den bisherigen Ergebnissen schon machen.

Bei `accepts1` liegen die Messwerte im Bereich 0.003ms bis 6.4ms. Für längere und komplexere Strings (z.B. "aaaabbbbcccc" mit 6.3958ms) scheint das Multithreading einen größeren Overhead zu erzeugen, denn bei kurzen Eingaben (z.B. "a", "b", "c") ist die Verarbeitungszeit entsprechend kurz.

Bei `accepts2` liegen die Zeiten im Bereich von 0.002ms bis 0.0396ms. Die Methode ist deutlich schneller als `accepts1`, insbesondere bei kurzen Eingaben. Es gibt eine leichte Steigerung bei komplexeren Eingaben, aber die Zeiten bleiben insgesamt sehr niedrig, was darauf hindeutet, dass das Backtracking effizienter ist und mit geringem Overhead arbeitet.

Bei `accepts3` liegen die Zeiten im Bereich von 0.023ms bis 0.131ms. Diese Implementierung hat im Vergleich zu den anderen Beiden eine mittlere Performance. Sie ist schneller als `accepts1`, aber langsamer als `accepts2`. Die Zeit für längere und komplexere Eingaben ist etwas höher als bei `accepts2`, was darauf hinweist, dass das Tracing von State-Sets hier zusätzliche Berechnungen und Overhead verursacht.

c)

```
dfaOfNfa:
->     S -> a A+S | b B+S | c C+S
       A+S -> a A+R+S | b A+B+S | c A+C+S
       B+S -> a A+B+S | b B+R+S | c B+C+S
       C+S -> a A+C+S | b B+C+S | c C+R+S
    () A+R+S -> a A+R+S | b A+B+S | c A+C+S
       A+B+S -> a A+B+R+S | b A+B+R+S | c A+B+C+S
       A+C+S -> a A+C+R+S | b A+B+C+S | c A+C+R+S
    () B+R+S -> a A+B+S | b B+R+S | c B+C+S
       B+C+S -> a A+B+C+S | b B+C+R+S | c B+C+R+S
    () C+R+S -> a A+C+S | b B+C+S | c C+R+S
    () A+B+R+S -> a A+B+R+S | b A+B+R+S | c A+B+C+S
       A+B+C+S -> a A+B+C+R+S | b A+B+C+R+S | c A+B+C+R+S
    () A+C+R+S -> a A+C+R+S | b A+B+C+S | c A+C+R+S
    () B+C+R+S -> a A+B+C+S | b B+C+R+S | c B+C+R+S
    () A+B+C+R+S -> a A+B+C+R+S | b A+B+C+R+S | c A+B+C+R+S
```

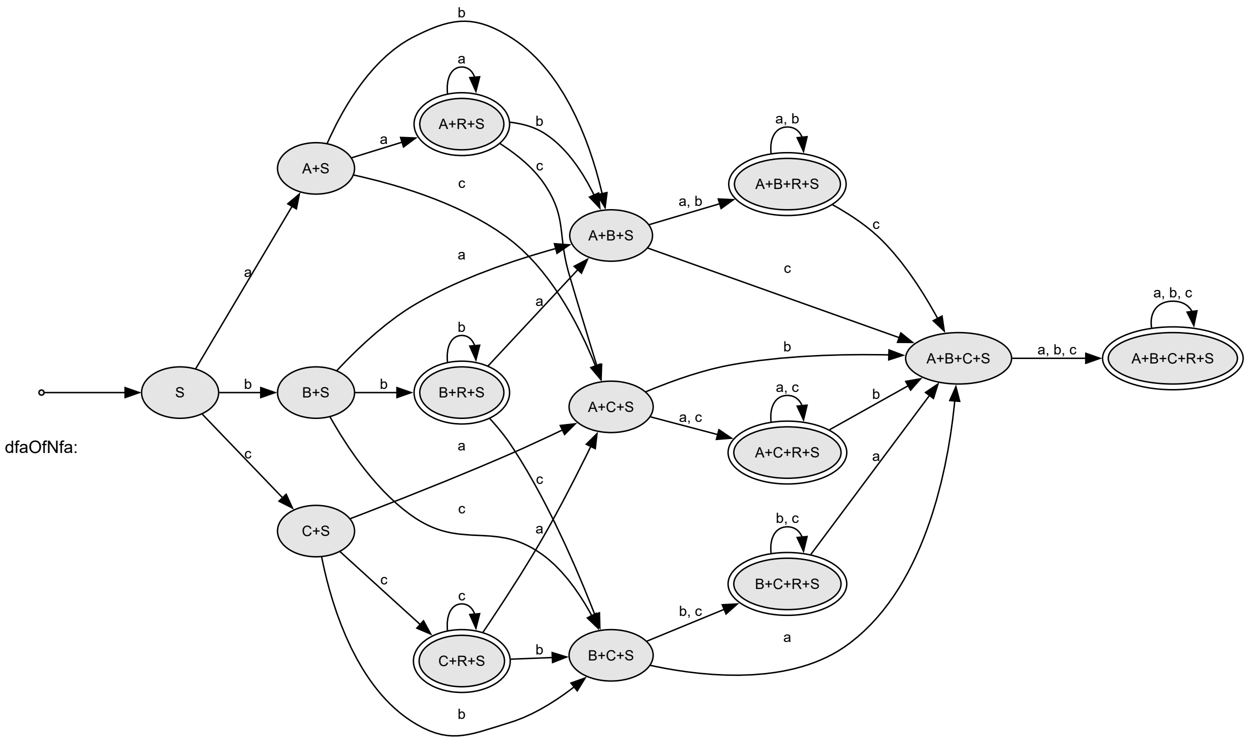


Figure 3: Visualized Automaton

d)

Die Methode `minimalOf` erstellt einen minimalen Deterministischen Endlichen Automaten (DFA) aus einem gegebenen DFA, indem sie die Äquivalenz von Zuständen überprüft und nicht äquivalente Zustände zusammenfasst.

```
minDfaOfNfa:
->     S -> a A+S | b B+S | c C+S
       A+S -> a A+R+S | b A+B+S | c A+C+S
       B+S -> a A+B+S | b B+R+S | c B+C+S
       C+S -> a A+C+S | b B+C+S | c C+R+S
    () A+R+S -> a A+R+S | b A+B+S | c A+C+S
       A+B+S -> a A+B+R+S | b A+B+R+S | c A+B+C+S
       A+C+S -> a A+C+R+S | b A+B+C+S | c A+C+R+S
    () B+R+S -> a A+B+S | b B+R+S | c B+C+S
       B+C+S -> a A+B+C+S | b B+C+R+S | c B+C+R+S
    () C+R+S -> a A+C+S | b B+C+S | c C+R+S
    () A+B+R+S -> a A+B+R+S | b A+B+R+S | c A+B+C+S
       A+B+C+S -> a A+B+C+R+S | b A+B+C+R+S | c A+B+C+R+S
    () A+C+R+S -> a A+C+R+S | b A+B+C+S | c A+C+R+S
    () B+C+R+S -> a A+B+C+S | b B+C+R+S | c B+C+R+S
    () A+B+C+R+S -> a A+B+C+R+S | b A+B+C+R+S | c A+B+C+R+S
```

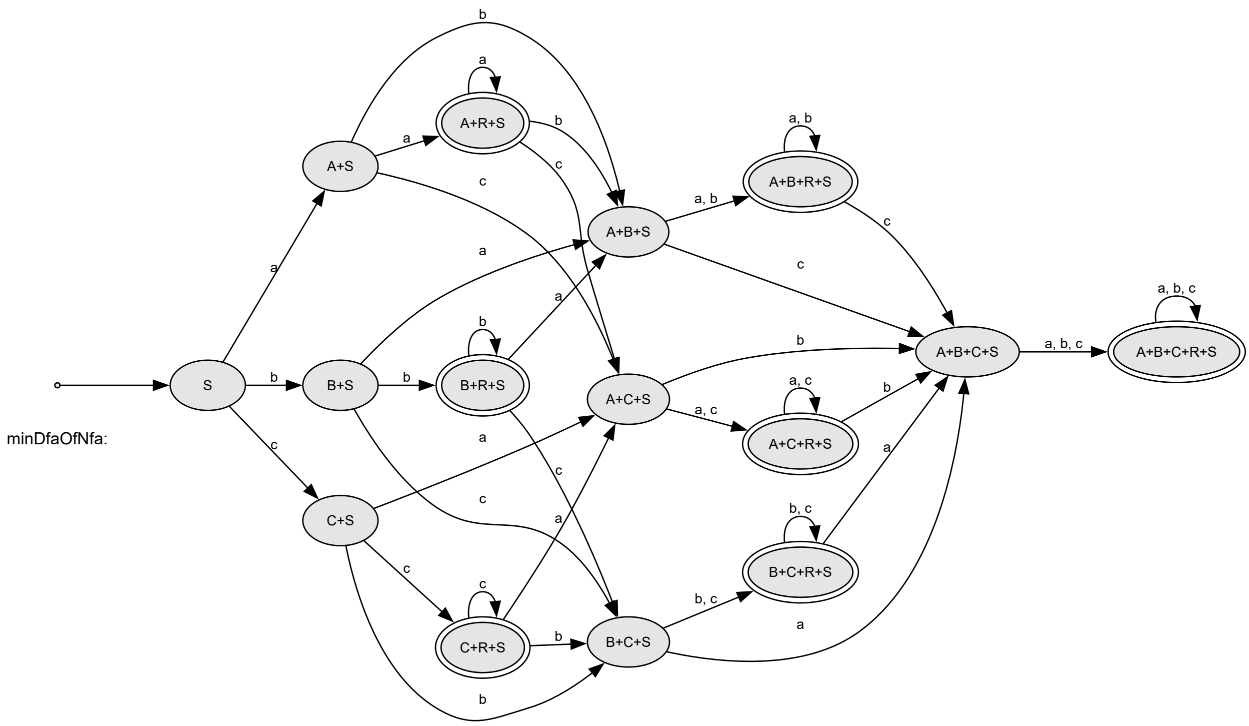


Figure 4: Visualized Automaton

Der DFA von Aufgabe 3c) verändert sich nicht, wenn man `minimalOf` darauf ausführt, da der DFA bereits minimal ist.

Das bedeutet, dass der DFA keine äquivalenten Zustände hat, die zusammengefasst werden könnten, ohne das Verhalten des Automaten zu verändern.

Aufgabe 4

a)

```
ConstDef = 'const' Type ident Init { ',' ident Init } ';' .
Type = 'bool' | 'int' .
Init = '=' ( false | true | [ '+' | '-' ] number ) .
```

```
ConstDef -> 'const' Type ident Init IdList ';'
IdList -> ',' ident Init IdList | eps
Type -> 'bool' | 'int'
Init -> '=' BoolOrNumber
BoolOrNumber -> 'false' | 'true' | OptSign number
OptSign -> '+' | '-' | eps
```

b)

Gegebener Kellerautomat-Algorithmus (top-down):

- S.1: Für jede Regel A→α wird ein Übergang der Form erzeugt:

  δ(Z, ϵ, A)=(Z, α^R), wobei α^R die umgekehrte Reihenfolge der Symbole in α ist.

- S.2: Für jedes Terminalsymbol α wird ein Übergang der Form erzeugt:

  δ(Z, α, α) = (Z, ϵ).

S.1 A -> α

`ConstDef`

δ(Z, ϵ, ConstDef) = (Z, ';' IdList Init ident Type 'const')

`IdList`

δ(Z, ϵ, IdList)=(Z, IdList Init ident ',')

δ(Z, ϵ, IdList)=(Z, ϵ)

`Type`

δ(Z, ϵ, Type) = (Z, 'bool')

δ(Z, ϵ, Type) = (Z, 'int')

`Init`

δ(Z, ϵ, Init) = (Z, BoolOrNumber '=')

`BoolOrNumber`

δ(Z, ϵ, BoolOrNumber) = (Z, 'false')

δ(Z, ϵ, BoolOrNumber) = (Z, 'true')

δ(Z, ϵ, BoolOrNumber) = (Z, numberOptSign)

`OptSign`

δ(Z, ϵ, Init) = (Z, number OptSign)

δ(Z, ϵ, Optsign) = (Z, +)

δ(Z, ϵ, Optsign) = (Z, -)

δ(Z, ϵ, Optsign) = (Z, ϵ)

S.2 Übergänge für Terminalsymbole α

Schlüsselwörter

δ(Z,'const', 'const') = (Z, ϵ)

δ(Z,'bool', 'bool') = (Z, ϵ)

δ(Z,'int', 'int') = (Z, ϵ)

δ(Z,'false', 'false') = (Z, ϵ)

δ(Z,'true', 'true') = (Z, ϵ)

Symbole:

δ(Z,'=', '=') = (Z, ϵ)

δ(Z,'+', '+') = (Z, ϵ)

δ(Z,'−', '−') = (Z, ϵ)

δ(Z,',', ',') = (Z, ϵ)

δ(Z,';', ';') = (Z, ϵ)

Literale:

δ(Z, number, number) = (Z, ϵ)

c)

Gegebener erweiterter Kellerautomat-Algorithmus (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 → α einen Übergang
  δ(Z, ϵ, α) = (Z, A).
- S.2: Erzeuge für jedes Terminalsymbol a einen Übergang
  δ(Z, a, x) = (Z, xa) für alle x ϵ V ∪ {$}.
- S.3: Erzeuge den Übergang δ(Z, ϵ, $S) = (R, ϵ).

S.1 A -> α

δ(Z, ϵ, 'const' Type ident Init IdList ';') = (Z, ConstDef)

δ(Z, ϵ, ',' ident Init IdList) = (Z, IdList)

δ(Z, ϵ, ϵ) = (Z, IdList)

δ(Z, ϵ, 'bool') = (Z, Type)

δ(Z, ϵ, 'int') = (Z, Type)

δ(Z, ϵ, '=' BoolOrNumber) = (Z, Init)

δ(Z, ϵ, 'false') = (Z, BoolOrNumber)

δ(Z, ϵ, 'true') = (Z, BoolOrNumber)

δ(Z, ϵ, OptSign number) = (Z, BoolOrNumber)

δ(Z, ϵ, '+') = (Z, OptSign)

δ(Z, ϵ, '-') = (Z, OptSign)

δ(Z, ϵ, ϵ) = (Z, OptSign)

S.2 Übergänge für Terminalsymbole α

Schlüsselwörter

δ(Z,'bool', 'const') = (Z, 'const' 'bool')

δ(Z,'int', 'const') = (Z, 'const' 'int')

δ(Z,'false', '=') = (Z, '=' 'false')

δ(Z,'true', '=') = (Z, '=' 'true')

Symbole:

$delta$(Z, '=', ident) = (Z, ident '=')

$delta$(Z, '+', '=') = (Z, '=' '+')

$delta$(Z, '-', '=') = (Z, '=' '-')

δ(Z, ',', Init) = (Z, Init ',')

δ(Z, ';', IdList) = (Z, IdList ';')

Literale:

δ(Z, number, OptSign) = (Z, OptSign number)

S.3 Übergang zum Endzustand

δ(Z, ϵ, \$ConstDef) = (R, ϵ)

d)

Gegebener Satz:

`const int max = 100;`

Kellerautomat (top-down):

=> beginnt mit S und arbeitet sich bis zu den Terminals herunter

```
(Z, S.const int max = 100;)
|- (Z, ';' Idlist Init ident Type 'const' . const int max = 100;)
|- (Z, ';' Idlist Init ident Type . int max = 100’)
|- (Z, ';' Idlist Init ident int. int max = 100;)
|- (Z, ';' Idlist Init ident . max = 100;)
|- (Z, ';' Idlist Init . = 100;)
|- (Z, ';' Idlist BoolOrNumber = . = 100;)
|- (Z, ';' Idlist BoolOrNumber . 100;)
|- (Z, ';' Idlist number . 100;)
|- (Z, ';' Idlist . ;)
|- (Z, ';' . ;)
|- (Z, ϵ . ;)
```

=> accepted

Erweiterter Kellerautomat (bottom-up)

=> beginnt mit den Terminals und reduziert sie schrittweise zu S

```
(Z, $S . const int max = 100;)
|- (Z, $ 'const' . int max = 100;)
|- (Z, $ 'const' int . max = 100;)
|- (Z, $ 'const' Type . max = 100;)
|- (Z, $ 'const' Type ident . = 100;)
|- (Z, $ 'const' Type ident '=' . 100;)
|- (Z, $ 'const' Type ident '=' 100 . ;)
|- (Z, $ 'const' Type ident Init . ;)
|- (Z, $ 'const' Type ident Init ';' . ϵ)
|- (Z, $ ConstDef . ϵ)
|- (R, ϵ . ϵ)
```

Aufgabe 5

a)

Regel	First1	Follow1
`progmod`	`{MODULE}`	`{}`
`priority`	`{const, ϵ}`	`{';'}`
`imppart`	`{FROM, IMPORT}`	`{DECL, BEGIN}`
`implist`	`{id}`	`{DECL, BEGIN}`
`block`	`{DECL, BEGIN}`	`{id}`
`declpart`	`{DECL}`	`{BEGIN}`
`statpart`	`{BEGIN}`	`{id}`
`statseq`	`{STAT}`	`{';', STAT}`

b)

`k=0`

`k=0` würde bedeuten, dass der Parser keine Lookahead-Symbole verwendet und allein durch den Zustand entscheiden muss, welche Regel anzuwenden ist.
Da es aber Alternativen in mehreren Regeln gibt (z. B. `imppart`), trifft das nicht zu.

`k=1`

Für `k=1` gibt es einen Konflikt der $First_2$-Mengen von `imppart`:

$$\text{imppart} \rightarrow \text{FROM id IMPORT implist} \mid \text{IMPORT implist}$$

$$\text{Follow}_1(\text{imppart}) = \{\text{DECL}, \text{BEGIN}\}$$

Um diesen Konflikt zu vermeiden, benötigen wir mindestens k=2, da der Parser so genug Informationen aus dem Kontext erhalten kann, um zwischen den Alternativen zu entscheiden.

`k=2`

Bei der Regel `statseq`

$$\text{statseq} \rightarrow \text{STAT} \mid \text{STAT ; statseq}$$

$$\text{First}_1(\text{statseq}) = \{\text{STAT}\}, \quad \text{Follow}_1(\text{statseq}) = \{\text{';'}, \text{END}\}$$

kann der Parser auch mit `k=2` nicht entscheiden, ob nach `STAT` die Produktion endet oder ob ein weiteres `;` statseq folgt, da `STAT` in beiden Alternativen vorkommt.

Erst mit 3 Symbolen kann der Parser entscheiden, ob ein weiteres `STAT` folgt, da entweder ein ob ein `STAT` oder ein `END` folgt.

`k=3`

Die Grammatik mit LL(3) geparst werden können.

c)

Regeln mit Konflikten:

- imppart -> FROM id IMPORT implist | IMPORT implist
- implist -> id , implist | id
- dclpart -> DECL dclpart' | DECL
- statseq -> STAT ; statseq | STAT

Die für den LL(1)-Umbau relevanten Regeln können aufgeteilt werden, um die Konflikte aufzulösen:

```
progmod -> MODULE id : priority ; imppart block id .
priority -> const | ϵ
imppart -> FROM id optimport
optimport -> IMPORT implist | ϵ
implist -> id moreimplist
moreimplist -> , implist | ϵ
block -> dclpart statpart | statpart
declpart -> DECL dclpartlist
dclpartlist -> ; dclpart | ϵ
statpart -> BEGIN statseq ; END
statseq -> STAT statseqlist
statseqlist -> ; statseq | ϵ
```

First-Mengen:

Regel	First (Ursprünglich)	First (Transformiert)
`progmod`	`{MODULE}`	`{MODULE}`
`priority`	`{const, ϵ}`	`{const, ϵ}`
`imppart`	`{FROM, IMPORT}`	`{FROM}`
`optimport`		`{IMPORT}`
`implist`	`{id}`	`{id}`
`moreimplist`		`{',' ϵ}`
`block`	`{DECL, BEGIN}`	`{DECL, BEGIN}`
`declpart`	`{DECL}`	`{DECL}`
`dclpartlist`		`{';', ϵ}`
`statpart`	`{BEGIN}`	`{BEGIN}`
`statseq`	`{STAT}`	`{STAT}`
`statseqlist`		`{';', ϵ}`

Nach der Umstrukturierung in eine LL(1)-Grammatik erfüllen die Regeln nun die LL(1)-Bedingung