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). Die Vorgehensweise lässt sich wie folgt zusammenfassen:

1. Startzustand festlegen: Die Funktion extrahiert das Startsymbol der Grammatik (in der Regel das Wurzelsymbol) und setzt es als Startzustand des NFAs.
2. Übergänge basierend auf Produktionsregeln hinzufügen: Für jede Produktionsregel der Grammatik, die eine Nichtterminalsymbole (nt) 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. Endzustände für terminale Produktionen: Wenn eine Regel mit einem terminalen Symbol endet (also keine Nichtterminalsymbole im Ziel hat), wird der aktuelle Zustand als Endzustand im NFA hinzugefügt.

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

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

    // Step 1: Set the start state of the NFA based on the root of t
    const auto startStateSymbol = dynamic_cast<NTSymbol *>(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) {
            const State srcState(nt->name);
            const TapeSymbol tapeSymbol = seq->symbolAt(0)->name[0]

            if (const auto targetSymbol = seq->back(); targetSymbol
                // Step 2: Add transitions (delta functions) based o
                const State targetState(targetSymbol->name);
                builder.addTransition(srcState, tapeSymbol, targetSt
            } else {
                // Step 3: Add final states for rules that produce t
                builder.addFinalState(nt->name);
            }
        }
    }

    return builder.buildNFA();
}
```

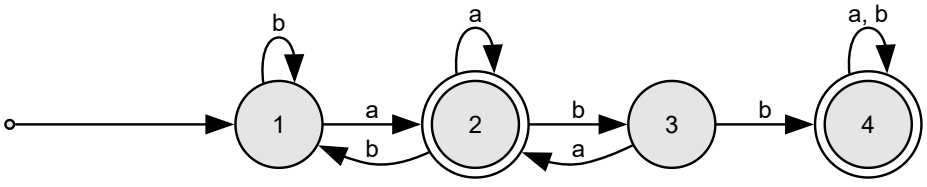
Getestet wurde die Funktion mit der folgenden Grammatik, die auch in der bereits vorhandenen Testfunktion testNFAFromGrammar verwendet wird:

```
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



nfaFromGrammar:

Figure 1: NFA from Grammar

Umgekehrte Transformation von NFA oder DFA 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. Erstellen des Wurzelsymbols und Initialisieren des Builders: Der Startzustand des NFAs wird als Wurzelsymbol der Grammatik gesetzt. Ein GrammarBuilder wird mit diesem Symbol initialisiert.
2. Übergangsregeln durchgehen und Produktionsregeln hinzufügen: Für jeden Zustand im NFA und seine Übergänge wird die Funktion addProductionRules aufgerufen, um Produktionsregeln für die Grammatik zu generieren. Diese Regeln basieren auf den Übergängen im NFA, wobei das Tape-Symbol und die Zielzustände verwendet werden.
3. Endzustände behandeln: 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.
4. Grammatik zurückgeben: Nachdem alle Regeln hinzugefügt wurden, wird die Grammatik mit dem GrammarBuilder erzeugt und zurückgegeben.

Die Funktion `addProductionRules` erstellt Produktionsregeln für jede mögliche Übergangskombination und sorgt dafür, dass bei Endzuständen die entsprechenden Regeln für die Grammatik ergänzt werden.

```
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() && srcS
            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>(rootSymbo

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

    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 anfänglichen 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

```
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)

```
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());
    cout << "dfa->accepts(\"b\") = " << dfa->accepts("b") << endl;
    cout << "dfa->accepts(\"bzb\") = " << dfa->accepts("bzb") << en
    cout << "dfa->accepts(\"bbb\") = " << dfa->accepts("bbb") << en
    cout << "dfa->accepts(\"bbzbb\") = " << dfa->accepts("bbzbb") <
    cout << "dfa->accepts(\"z\")   = " << dfa->accepts("z") << endl
    cout << "dfa->accepts(\"zzbb\") = " << dfa->accepts("zzbb") << 
    cout << "dfa->accepts(\"zzzz\") = " << dfa->accepts("zzzz") << 
    cout << "dfa->accepts(\"bbba\") = " << dfa->accepts("bbba") << 
    cout << endl;
```

b)

// TODO

Aufgabe 3

a)

`accept1`

This method uses multithreading to simulate non-determinism. It spawns a new thread for each possible transition (epsilon transitions or symbol transitions). If any thread reaches a final state at the end of the tape, a shared accepted variable is set to true, protected by a mutex to ensure thread safety. All threads are joined before the function completes. This method is computationally intensive but showcases parallel exploration of possible paths.

```
// NFA::accepts1: uses multithreading to simulate non-determinism
//--------------
static bool accepted;
static mutex mtx; // used to synchronize access to variable accepte

static void accept1(const NDelta &delta, const StateSet &F,
                    const State &s, const Tape &tape, int i) {
    vector<thread> tv; // thread vector
    for (const State &epsDest: delta[s][eps]) // eps. transitions
        tv.emplace_back(accept1, cref(delta), cref(F),
                        epsDest, tape, i);
    const TapeSymbol tSy = tape[i];
    if (tSy == eot && F.contains(s)) {
        // end of tape and s is final
        mtx.lock();
        accepted = true;
        mtx.unlock();
    } else
        for (const State &tSyDest: delta[s][tSy]) // symbol transit
            tv.emplace_back(accept1, cref(delta), cref(F),
                            tSyDest, tape, i + 1);
    for (auto &t: tv)
        t.join(); // join thread t with current thread
}

bool NFA::accepts1(const Tape &tape) const {
    accepted = false;
    accept1(this->delta, this->F, this->s1, tape, 0); // normal cal
    // ... within current thread
    return accepted;
}
```

accept2

This method uses recursive backtracking to simulate non-determinism. Starting from the initial state, it recursively explores all possible transitions (epsilon and symbol) along the tape. If any path reaches a final state at the end of the tape, the function returns true. Otherwise, it backtracks and continues exploring other paths. This approach is simpler and more memory-efficient than multithreading but can encounter stack overflow for deep recursion.

```
// NFA::accepts2: uses backtracking to simulate non-determinism
//--------------
bool NFA::accepts2(const Tape &tape) const {
    return accepts2(s1, tape, 0); // see below
}

bool NFA::accepts2(const State &s,
                   const Tape &tape, int i) const {
    for (const State &epsDest: delta[s][eps]) // eps transitions
        if (accepts2(epsDest, tape, i)) // recursive call
            return true;
    TapeSymbol tSy = tape[i];
    if (tSy == eot)
        return F.contains(s); // accepted <==> s is final
    for (const State &tSyDest: delta[s][tSy]) // symbol transitions
        if (accepts2(tSyDest, tape, i + 1)) // recursive call
            return true;
    return false; // not accepted as no call succeeded
}
```

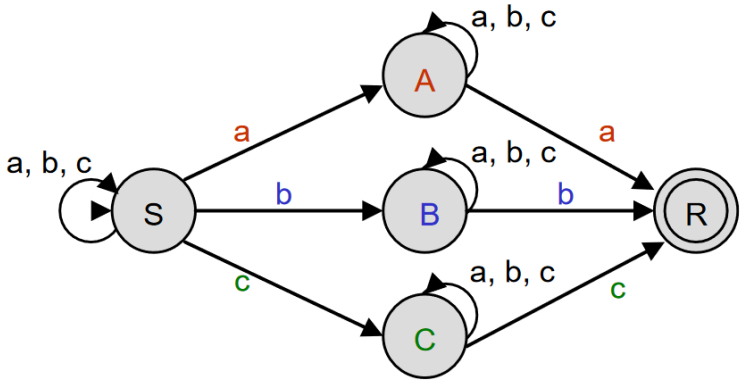
accept3

This method simulates non-determinism by tracing sets of states instead of exploring each path individually. Starting from the epsilon closure of the initial state, it iteratively computes the set of all possible destination states for each tape symbol. The process continues until the end of the tape. If the final set of states intersects with the set of final states, the input is accepted. This method is efficient and avoids recursion or threading by working with state sets.

```
// NFA::accepts3: tracing of state sets to simulate non-determinism
//--------------
bool NFA::accepts3(const Tape &tape) const {
    int i = 0; // index of first symbol
    TapeSymbol tSy = tape[i]; // fetch first symbol
    StateSet ss = epsClosureOf(StateSet(s1));

    while (tSy != eot) {
        // eot = end of tape
        StateSet dest = allDestsFor(ss, tSy);
        if (!defined(dest))
            return false; // undefined, so no acceptance
        ss = epsClosureOf(dest);
        i++;
        tSy = tape[i];
    }
    return !empty(ss ^ F); // accepted <==> (ss ^ F) != {}
}
```

NFA Automaton

Aus dem gegebenen Automaten



Figure

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
```

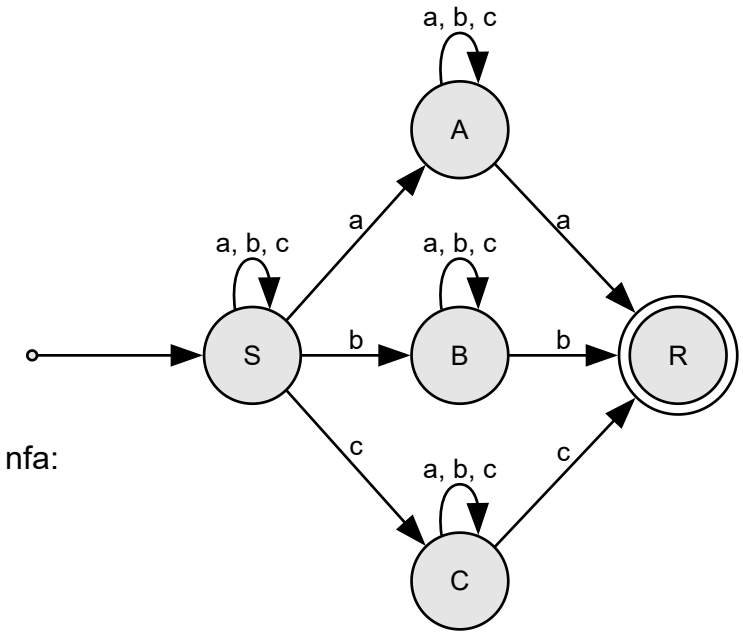


Figure TODO: Visualized Automaton

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",
        "aa", "ab", "ac",
        "abc", "abca", "abcb", "abcc",
        "aaaabbbbcccc", "abcabcabcabc", "aabbccbbcca",
        "aaaaaaaa", "bbbbbbbb", "cccccccc",
        "d","xyz"
    };

    auto testMethod = [&](const string &methodName, auto acceptsMetl
        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->a
    testMethod("accepts2", [&](const string &input) { return nfa->a
    testMethod("accepts3", [&](const string &input) { return nfa->a
```

Das produziert den Output:

```
Testing accepts1:
nfa->accepts1("a") => (rejected)
nfa->accepts1("b") => (rejected)
nfa->accepts1("c") => (rejected)
nfa->accepts1("aa") => (accepted)
nfa->accepts1("ab") => (rejected)
nfa->accepts1("ac") => (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("d") => (rejected)
nfa->accepts1("xyz") => (rejected)
```

```
Testing accepts2:
nfa->accepts2("a") => (rejected)
nfa->accepts2("b") => (rejected)
nfa->accepts2("c") => (rejected)
nfa->accepts2("aa") => (accepted)
nfa->accepts2("ab") => (rejected)
nfa->accepts2("ac") => (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("d") => (rejected)
nfa->accepts2("xyz") => (rejected)
```

```
Testing accepts3:
nfa->accepts3("a") => (rejected)
nfa->accepts3("b") => (rejected)
nfa->accepts3("c") => (rejected)
nfa->accepts3("aa") => (accepted)
nfa->accepts3("ab") => (rejected)
nfa->accepts3("ac") => (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("d") => (rejected)
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 acceptsMetl
    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("b") => (rejected) - 0ms
nfa->accepts1("c") => (rejected) - 0ms
nfa->accepts1("aa") => (accepted) - 0ms
nfa->accepts1("ab") => (rejected) - 0ms
nfa->accepts1("ac") => (rejected) - 0ms
nfa->accepts1("abc") => (rejected) - 0ms
nfa->accepts1("abca") => (accepted) - 0.001ms
nfa->accepts1("abcb") => (accepted) - 0ms
nfa->accepts1("abcc") => (accepted) - 0ms
nfa->accepts1("aaaabbbbcccc") => (accepted) - 0.006ms
nfa->accepts1("abcabcabcabc") => (accepted) - 0.005ms
nfa->accepts1("aabbccbbcca") => (accepted) - 0.004ms
nfa->accepts1("aaaaaaaa") => (accepted) - 0.003ms
nfa->accepts1("bbbbbbbb") => (accepted) - 0.003ms
nfa->accepts1("cccccccc") => (accepted) - 0.004ms
nfa->accepts1("d") => (rejected) - 0ms
nfa->accepts1("xyz") => (rejected) - 0ms
```

```
Testing accepts2:
nfa->accepts2("a") => (rejected) - 0ms
nfa->accepts2("b") => (rejected) - 0ms
nfa->accepts2("c") => (rejected) - 0ms
nfa->accepts2("aa") => (accepted) - 0ms
nfa->accepts2("ab") => (rejected) - 0ms
nfa->accepts2("ac") => (rejected) - 0ms
nfa->accepts2("abc") => (rejected) - 0ms
nfa->accepts2("abca") => (accepted) - 0ms
nfa->accepts2("abcb") => (accepted) - 0ms
nfa->accepts2("abcc") => (accepted) - 0ms
nfa->accepts2("aaaabbbbcccc") => (accepted) - 0ms
nfa->accepts2("abcabcabcabc") => (accepted) - 0ms
nfa->accepts2("aabbccbbcca") => (accepted) - 0ms
nfa->accepts2("aaaaaaaa") => (accepted) - 0ms
nfa->accepts2("bbbbbbbb") => (accepted) - 0ms
nfa->accepts2("cccccccc") => (accepted) - 0ms
nfa->accepts2("d") => (rejected) - 0ms
nfa->accepts2("xyz") => (rejected) - 0ms
```

```
Testing accepts3:
nfa->accepts3("a") => (rejected) - 0ms
nfa->accepts3("b") => (rejected) - 0ms
nfa->accepts3("c") => (rejected) - 0ms
nfa->accepts3("aa") => (accepted) - 0ms
nfa->accepts3("ab") => (rejected) - 0ms
nfa->accepts3("ac") => (rejected) - 0ms
nfa->accepts3("abc") => (rejected) - 0ms
nfa->accepts3("abca") => (accepted) - 0ms
nfa->accepts3("abcb") => (accepted) - 0ms
nfa->accepts3("abcc") => (accepted) - 0ms
nfa->accepts3("aaaabbbbcccc") => (accepted) - 0ms
nfa->accepts3("abcabcabcabc") => (accepted) - 0ms
nfa->accepts3("aabbccbbcca") => (accepted) - 0ms
nfa->accepts3("aaaaaaaa") => (accepted) - 0ms
nfa->accepts3("bbbbbbbb") => (accepted) - 0ms
nfa->accepts3("cccccccc") => (accepted) - 0ms
nfa->accepts3("d") => (rejected) - 0ms
nfa->accepts3("xyz") => (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:

Variante 1: Messgenauigkeit erhöhen auf Nanosekunden

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

```
double elapsedTime() {
  return chrono::duration_cast<chrono::nanoseconds>(stop_tp - start_
}
```

Das neue Ergebnis:

```
Testing accepts1:
nfa->accepts1("a") => (rejected) - 0.2695ms
nfa->accepts1("b") => (rejected) - 0.1893ms
nfa->accepts1("c") => (rejected) - 0.5607ms
nfa->accepts1("aa") => (accepted) - 0.4595ms
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("d") => (rejected) - 0.0033ms
nfa->accepts1("xyz") => (rejected) - 0.0069ms
```

```
Testing accepts2:
nfa->accepts2("a") => (rejected) - 0.0109ms
nfa->accepts2("b") => (rejected) - 0.0086ms
nfa->accepts2("c") => (rejected) - 0.0097ms
nfa->accepts2("aa") => (accepted) - 0.0192ms
nfa->accepts2("ab") => (rejected) - 0.0173ms
nfa->accepts2("ac") => (rejected) - 0.0185ms
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("d") => (rejected) - 0.0023ms
nfa->accepts2("xyz") => (rejected) - 0.0127ms


Testing accepts3:
nfa->accepts3("a") => (rejected) - 0.0546ms
nfa->accepts3("b") => (rejected) - 0.0516ms
nfa->accepts3("c") => (rejected) - 0.0383ms
nfa->accepts3("aa") => (accepted) - 0.023ms
nfa->accepts3("ab") => (rejected) - 0.0397ms
nfa->accepts3("ac") => (rejected) - 0.0465ms
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("d") => (rejected) - 0.0388ms
nfa->accepts3("xyz") => (rejected) - 0.0248ms
```

Auf diese Weise sind die Werte vergelichbarer.

Variante 2: Mehr iterationen pro Testwert

Dazu führe ich `accept` für jeden Testfall mehrfach aus, die Performance über viele Aufrufe hinweg zu vergleichen:

```
Testing accepts1:
nfa->accepts1("a") => - Time for 1000 iterations: 0.154ms
nfa->accepts1("b") => - Time for 1000 iterations: 0.136ms
nfa->accepts1("c") => - Time for 1000 iterations: 0.13ms
nfa->accepts1("aa") => - Time for 1000 iterations: 0.318ms
nfa->accepts1("ab") => - Time for 1000 iterations: 0.276ms
nfa->accepts1("ac") => - Time for 1000 iterations: 0.273ms
nfa->accepts1("abc") => - Time for 1000 iterations: 0.464ms
nfa->accepts1("abca") => - Time for 1000 iterations: 0.736ms
nfa->accepts1("abcb") => - Time for 1000 iterations: 0.737ms
nfa->accepts1("abcc") => - Time for 1000 iterations: 0.737ms
nfa->accepts1("aaaabbbbcccc") => - Time for 1000 iterations: 5.114ms
nfa->accepts1("abcabcabcabc") => - Time for 1000 iterations: 5.188ms
nfa->accepts1("aabbccbbcca") => - Time for 1000 iterations: 4.349ms
nfa->accepts1("aaaaaaaa") => - Time for 1000 iterations: 3.401ms
nfa->accepts1("bbbbbbbb") => - Time for 1000 iterations: 3.446ms
nfa->accepts1("cccccccc") => - Time for 1000 iterations: 3.353ms
nfa->accepts1("d") => - Time for 1000 iterations: 0ms
nfa->accepts1("xyz") => - Time for 1000 iterations: 0ms


Testing accepts2:
nfa->accepts2("a") => - Time for 1000 iterations: 0ms
nfa->accepts2("b") => - Time for 1000 iterations: 0ms
nfa->accepts2("c") => - Time for 1000 iterations: 0ms
nfa->accepts2("aa") => - Time for 1000 iterations: 0.001ms
nfa->accepts2("ab") => - Time for 1000 iterations: 0.001ms
nfa->accepts2("ac") => - Time for 1000 iterations: 0.001ms
nfa->accepts2("abc") => - Time for 1000 iterations: 0.003ms
nfa->accepts2("abca") => - Time for 1000 iterations: 0.001ms
nfa->accepts2("abcb") => - Time for 1000 iterations: 0.003ms
nfa->accepts2("abcc") => - Time for 1000 iterations: 0.004ms
nfa->accepts2("aaaabbbbcccc") => - Time for 1000 iterations: 0.029ms
nfa->accepts2("abcabcabcabc") => - Time for 1000 iterations: 0.013ms
nfa->accepts2("aabbccbbcca") => - Time for 1000 iterations: 0.003ms
nfa->accepts2("aaaaaaaa") => - Time for 1000 iterations: 0.002ms
nfa->accepts2("bbbbbbbb") => - Time for 1000 iterations: 0.002ms
nfa->accepts2("cccccccc") => - Time for 1000 iterations: 0.003ms
nfa->accepts2("d") => - Time for 1000 iterations: 0ms
nfa->accepts2("xyz") => - Time for 1000 iterations: 0ms
```

```
Testing accepts3:
nfa->accepts3("a") => - Time for 1000 iterations: 0.006ms
nfa->accepts3("b") => - Time for 1000 iterations: 0.006ms
nfa->accepts3("c") => - Time for 1000 iterations: 0.006ms
nfa->accepts3("aa") => - Time for 1000 iterations: 0.011ms
nfa->accepts3("ab") => - Time for 1000 iterations: 0.01ms
nfa->accepts3("ac") => - Time for 1000 iterations: 0.01ms
nfa->accepts3("abc") => - Time for 1000 iterations: 0.016ms
nfa->accepts3("abca") => - Time for 1000 iterations: 0.023ms
nfa->accepts3("abcb") => - Time for 1000 iterations: 0.023ms
nfa->accepts3("abcc") => - Time for 1000 iterations: 0.023ms
nfa->accepts3("aaaabbbbcccc") => - Time for 1000 iterations: 0.069ms
nfa->accepts3("abcabcabcabc") => - Time for 1000 iterations: 0.078ms
nfa->accepts3("aabbccbbcca") => - Time for 1000 iterations: 0.067ms
nfa->accepts3("aaaaaaaa") => - Time for 1000 iterations: 0.039ms
nfa->accepts3("bbbbbbbb") => - Time for 1000 iterations: 0.039ms
nfa->accepts3("cccccccc") => - Time for 1000 iterations: 0.039ms
nfa->accepts3("d") => - Time for 1000 iterations: 0.002ms
nfa->accepts3("xyz") => - Time for 1000 iterations: 0.002ms
```

Auswertung und weitere Möglichkeiten

Weiterführend könnte man auch die Durschnittszeit für jeden Teststring aus Variante 2 berechnen. 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", "d") 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 `accepts2` 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)

Die Funktion dfaOf konvertiert einen Nichtdeterministischen Endlichen Automaten (NFA) in einen Deterministischen Endlichen Automaten (DFA).

Zuerst wird die Zustandsübergangsfunktion für den DFA aufgebaut, indem die epsilon-Abschlüsse der Zustände und deren Übergänge für jedes mögliche Symbol berechnet werden. Neue Zustandsmengen werden dabei iterativ erstellt, bis alle möglichen Zustandsmengen untersucht wurden.

Dann wird der Startzustand des DFA gesetzt, der der epsilon-Abschluss des Startzustands des NFA ist.

Schließlich werden die Finalzustände des DFA definiert, indem überprüft wird, ob einer der Zustände der NFA-Finalzustände enthält.

Am Ende gibt die Funktion den neu konstruierten DFA zurück.

```
DFA *NFA::dfaOf() const {
    FABuilder fab;

    // 1. construct new delta function for DFA (S and V implicitly)
    const StateSet startStateSet = epsClosureOf(s1);
    SetOfStateSets allStateSets = startStateSet;
    SetOfStateSets sstc = startStateSet; // StateSets to check

    while (!sstc.empty()) {
        StateSet srcStateSet = sstc.anyElement();
        sstc.erase(srcStateSet);
        for (const TapeSymbol tSy: V) {
            StateSet destStateSet =
                    epsClosureOf(allDestsFor(srcStateSet, tSy));
            if (!destStateSet.empty()) {
                // transition is defined
                if (!allStateSets.contains(destStateSet)) {
                    allStateSets.insert(destStateSet);
                    sstc.insert(destStateSet);
                }
                fab.addTransition(srcStateSet.stateOf(), tSy,
                                  destStateSet.stateOf());
            }
        }
    }

    // 2. define new start state s1 for DFA
    fab.setStartState(startStateSet.stateOf());

    // 3. look for final states f and define new F for DFA
    for (const StateSet &stateSet: allStateSets)
        for (const State &f: F)
            if (stateSet.contains(f))
                fab.addFinalState(stateSet.stateOf());

    return fab.buildDFA();
}
```

Ergebnis:

```
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
```

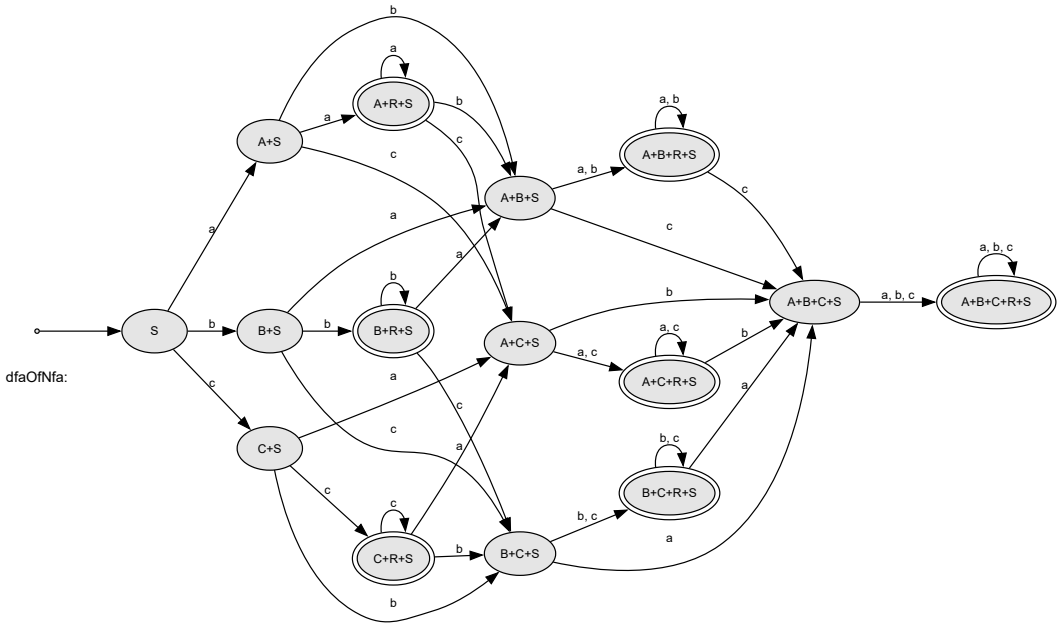


Figure TODO: Visualized Automaton

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.

```
DFA *DFA::minimalOf() const {

  NeTable ne; // table to define non-equivalent states

  // 1.  "table filling algorithm"
  // 1.a initialize ne table with false
  for (const State &si: S)
    for (const State &sj: S)
      ne[si][sj] = false;
  // 1.b final and non-final states are not equivalent
  for (const State &f : F)
    for (const State &s : (S - F))
      ne[f][s] =
      ne[s][f] = true;
  // 1.c now compute (non-)equivalent states
  bool anyChange = true;
  while (anyChange) {
    anyChange = false;
    for (const State &si: S)
      for (const State &sj: S)
          if ( (si != sj) && !ne[si][sj]) // si, sj seem to be equi
          for (TapeSymbol tSy: V) {
            State destSi = delta[si][tSy];
            State destSj = delta[sj][tSy];
            if ( (destSi != destSj) &&
                 ( !defined(destSi) ||
                   !defined(destSj) ||
                   ne[destSi][destSj] ) )
              ne[si][sj] = // true  // si and sj ...
              ne[sj][si] = // true  // ... are not equivalent
              anyChange  = true;
          }
  }

  // 2. from ne table create the partition of S (= a set of subsets,
  SetOfStateSets partition;
  for (const State &si: S) {
    StateSet subset = StateSet(si);
    for (const State &sj: S)
      if ( (    si != sj  ) &&
           (!ne[si]  [sj]) )  // si and sj are equiv.
        subset.insert(sj);
    partition.insert(subset);
  }

  FABuilder fab; // builder for the minimal DFA

  // 3. compute transitions for minimal DFA in builder
  for (const StateSet &srcStateSet: partition) {
    const State &srcState = srcStateSet.anyElement();
    for (TapeSymbol tSy: V) {
      const State &destState = delta[srcState][tSy];
```

```
      for (const StateSet &subset: partition)
        if (subset.contains(destState)) {
          fab.addTransition(srcStateSet.stateOf(), tSy,
                    /*dest*/subset.     stateOf());
          break;
        }
    }
  }

  // 4. look for start state s1 and define new s1 for min. DFA
  for (const StateSet &subset: partition)
    if (subset.contains(s1)) {
      fab.setStartState(subset.stateOf());
      break; // as s1 is in one subset only
    }

  // 5. look for final states f and define new F for min. DFA
  for (const StateSet &subset: partition)
    for (const State &f: F)
      if (subset.contains(f))
        fab.addFinalState(subset.stateOf());

  return fab.buildDFA();
}
```

Ergebnis:

```
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
```

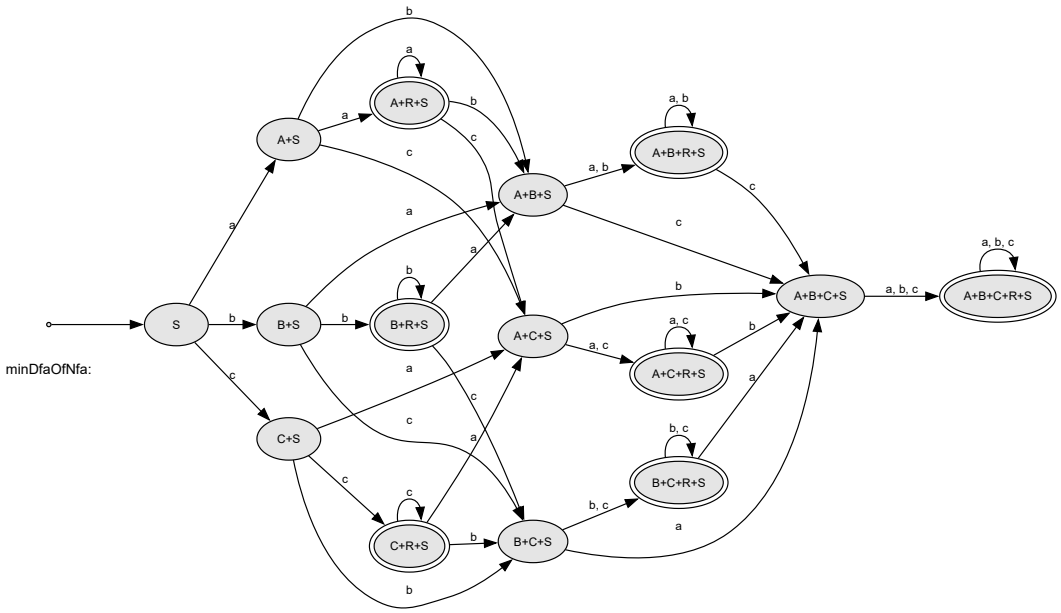


Figure TODO: Visualized Automaton

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)

Init -> OptSign number OptSign -> + | - | eps

S1

$\delta(Z, \epsilon, Iniz)$ = $Z, numberOptSign$ $\delta(Z, \epsilon, Optsign)$ = Z, +$

S2

$\delta(Z, number,$

b)

c)

d)

(Z, S.const int max = 100;) |- (Z,';' Idlist Type 'const'. cosnt int max = 100;) |- (Z, ';' IdList Type.int max = 100;)

Aufgabe 5

a)

b)

c)