1. Grammatiken – Grundbegriffe

a)

| VT(G) = { `"DATA"`, `","`, `"/"`, `"*"`, `id`, `num`, `str`, `"+"`, `"-"`, `"("`, `")"`, `"="`, `expr` } | = 13

| VN(G) = { `DataDecl`, `DataDeclRest`, `DataNameList`, `DataValueList`, `DataName`, `DataNameList`, `DataDoList`, `DataValue`, `DataDoListRest` } | = 9

b)

shortest:

- `DATA id / num /`
- `DATA id / str /`
- `DATA id / id /`

c)

Direkt rekursiv:

- `DataDeclRest`: links
- `DataNameList`: rechts
- `DataValueList`: links
- `DataDoList`: zentral
- `DataDoListRest`: links

Indirekt rekursiv:

- `DataDoList` => `DataDoListRest`: zentral
- `DataDoListRest` => `DataDoList`: zentral

d)

DataStat -> "Data" DataDecl DataDeclRest .

DataDeclRest -> ε | DataDeclRest DataDecl | DataDeclRest `","` DataDecl .

DataDecl -> DataNameList `"/"` DataValueList `"/"` .

DataNameList -> DataName | DataName `","` DataNameList .

DataName -> id | DataDoList .

DataValueList -> DataValue | DataValueList `","` DataValue .

DataValue -> OptSign num | str | id

- | num `"*"` id
- | num `"*"` OptSign num

- | num "*" str
- | id "*" id
- | id "*" OptSign num
- | id "*" str
- .

OptSign -> ε | "+" | "+" .

DataDoList -> "(" DataDoList DataDoListRest ")"

- | "(" id "(" IdList ")" DataDoListRest ")"
- . IdList -> id | IdList "," id .

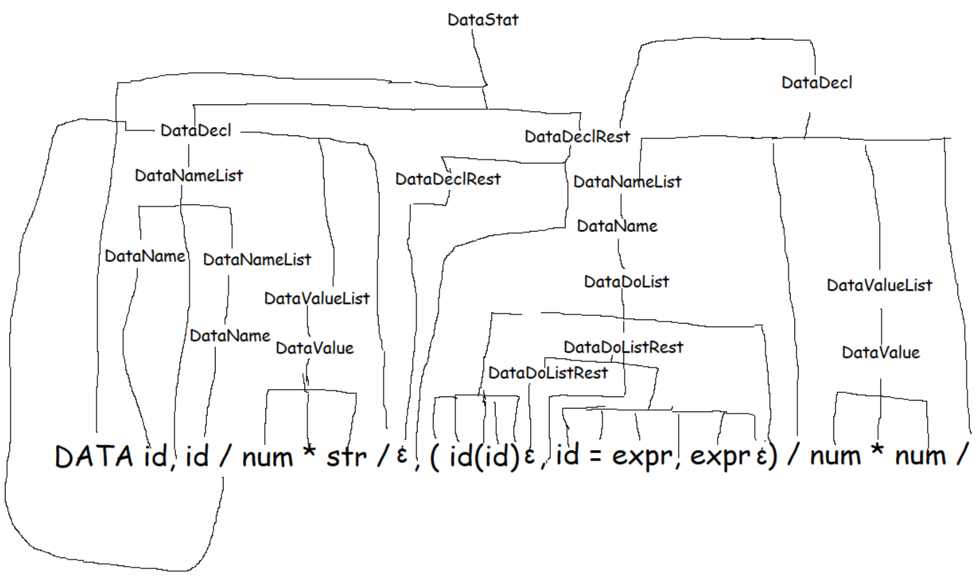
DataDoListRest -> ε

- | DataDoListRest "," DataDoList
- | DataDoListRest "," id "(" ExprList ")"
- | DataDoListRest "," id "=" expr "," expr
- | DataDoListRest "," id "=" expr "," expr, expr
- .

ExprList -> expr | ExprList "," expr .

EBNF ist lesbarer, da man mit weniger Alternativen durch Verwendung von "[" und "]" benötigt kann und keine Rekursion mit NTs für das mehrfache Vorkommen von [Terminal-]Symbolen verwenden muss.

e)



Für diesen Satz gib es nur einen Syntaxbaum, da es bei jeder Regel nur eine Alternative gibt, die zu diesem gegeben Satz führt.

2. Konstruktion einer Grammatik

Regelsystem

S -> OptSign LeadingDigit MiddleDigits UnevenNaturalDigit | OptSign UnevenNaturalDigit . // man könnte auch OptSign weglassen und dafür 4 weitere Optionen im NT "S" hinzufügen

OptSign -> ε | + | - .

MiddleDigits -> ε | 0 MiddleDigits | LeadingDigit MiddleDigits .

UnevenNaturalDigit -> 1 | 3 | 5 | 7 | 9 .

LeadingDigit -> UnevenNaturalDigit | 2 | 4 | 6 | 8 .

EBNF

S = [+ | -] [(1|2|3|4|5|6|7|8|9) { (0|1|2|3|4|5|6|7|8|9) }] (1|3|5|7|9) .

3. Oo-Implementierung von Grammatiken

No changes made to existing code. I used C++20.

a)

main.cpp

```
Grammar* newEpsilonFreeGrammarOf(Grammar* g) {
    // step 1
    VNt deletable = g->deletableNTs();

    // step 2

    // use symbolpool to get instances by name
    // (symbols from initial creation are still stored in SymbolPoolData)
    SymbolPool sp{};
    GrammarBuilder gb{g->root}; // reuse old root for now

    // for each rule
    // c++20 structureed binding
    for (const auto& [nt, sequenceSet] : g->rules)
    {
        // iterate over old sequence set
        for (const Sequence* seq : sequenceSet)
        {
            // begone epsilon
            if (seq->isEpsilon()) continue;

            // add copy
            gb.addRule(nt, new Sequence(*seq));

            // evaluate which indices of current sequence are deletable NTs
            std::vector<int> deletableNTindices{};
            for (int i = 0; i < seq->size(); i++) {
                Symbol* currSy = seq->at(i);
                if (currSy->isNT() &&
                    deletable.contains(dynamic_cast<NTSymbol*>(currSy))) {
                    deletableNTindices.push_back(i);
                }
            }

            // add the current sequence with every possible combination
            // of not including NTs in deletableNTindices
             // 2^n(-1) iterations
            for (int i = 0; i < 1 << deletableNTindices.size(); ++i) {
                Sequence* copy = new Sequence(*seq);
                for (int j = deletableNTindices.size() - 1; j >= 0; --j) {
                    // generate all possible combinations
                    // of indices in deletableNTindices
                    int symbolsRemoved = 0;
                    if (((1 << j) & i) > 0) {
                        copy->removeSymbolAt(deletableNTindices[j - symbolsRemoved]);
                        symbolsRemoved += 1;
                    }
```

```
                }
                // don't add empty alternatives
                // also duplicates are ignored
                if (!copy->isEpsilon()) gb.addRule(nt, copy);
            }
        }
    }

    // step 3
    if (deletable.contains(g->root)) {
        // add S' (or rather name of original root node + ')
        NTSymbol* newRoot = sp.ntSymbol(g->root->name + "'");
        gb.addRule(newRoot, { new Sequence({g->root}), new Sequence() /* eps */});
        gb.setNewRoot(newRoot);
    }

    return gb.buildGrammar();
}
```

Testcode:

```cpp
#elif TESTCASE == 4

        gb2 = new GrammarBuilder(string("G1.txt"));
        g2 = gb2->buildGrammar();
        Grammar* epsilonFree = newEpsilonFreeGrammarOf(g2);
        // or for short: g2 = GrammarBuilder(string("G.txt")).buildGrammar();

        cout << "grammar from text file:" << endl << *g2 << endl;
        cout << "newEpsilonFreeGrammarOf(g2):" << endl << *epsilonFree << endl;


#elif TESTCASE == 5
```

Result:

```
START Main

symbol pool: 0 terminals and 0 nonterminals
  terminals     = {  }
  nonterminals = {  }

TESTCASE 4

grammar from text file:

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


newEpsilonFreeGrammarOf(g2):

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



symbol pool: 2 terminals and 5 nonterminals
  terminals     = { a, b }
  nonterminals = { C, S, A, B, S' }

elapsed time: 0.009

END Main
```

b) and also c)

main.cpp

```cpp
void languageOfRecursive(
    Language * language,
    const RulesMap & rules,
    const SequenceSet& sequences,
    // copy ctor of Sequence copies the collection, making use of call stack
    Sequence currSentence,
    int maxLen
) {
    if (currSentence.length() >= maxLen) return;

    for (const Sequence* rule : sequences) {
        // look at each symbol of current rule (alternative)
        for (Symbol* sy : *rule) {
            if (sy->isNT()) {
                // go to coresponding NTSymbol in the RulesMap
                NTSymbol* ntSy = dynamic_cast<NTSymbol*>(sy); // cannot be null
                languageOfRecursive(language, rules, rules[ntSy], currSentence, maxLen);
            }
            else {
                // add TSymbol to current sentence
                currSentence.append(sy);
            }
        }
        if (currSentence.length() <= maxLen) {
            // copy is necessary here because otherwise
            // we would get the TSymbols of the next alternative
            // in the previously added sentence (which we don't want)
            language->addSentence(new Sequence(currSentence));
        }
    }
}

Language* languageOf(const Grammar* g, int maxLen) {
    Language* language = new Language(maxLen);
    Sequence s{};
    languageOfRecursive(language, g->rules, g->rules[g->root], s, maxLen);
    return language;
}
```

Language.h

```
// Language.h:                                          SWE, 2022
// ----------------
// Lengwidsch
//================================

#ifndef Language_h
#define Language_h

#include <vector>
#include <set>
#include <iostream>

class Sequence;

class Language {

    friend std::ostream& operator <<(std::ostream& os, const Language& language);

    private:
        std::set<Sequence*> sentences{};
        int maxLength;

    public:
        Language(int maxLength);

        Sequence& at(int i) const;
        void addSentence(Sequence* s);
        bool hasSentence(Sequence* s) const;

};

#endif

// end of GrammarBuilder.h
//====================================================================
```

Language.cpp

```
// Language.h:                                                  SWE, 2022
// ----------------
// Lengwidsch
//================================

#include <exception>

#include "Language.h"
#include "SymbolStuff.h"
#include "SequenceStuff.h"

std::ostream& operator <<(std::ostream& os, const Language& language) {
    os << "L(G(S)): maxLength=" << language.maxLength << " {\n";
    for (const Sequence* sentence : language.sentences) {
        os << *sentence << "\n";
    }
    os << "}";
    return os;
}


Sequence& Language::at(int idx) const {
    if (idx >= sentences.size() || idx < 0)
        throw std::invalid_argument("invalid index");
    auto it = sentences.cbegin();
    std::advance(it, idx);
    return **it;
}

Language::Language(int maxLength)
    : maxLength{maxLength} {

}

void Language::addSentence(Sequence* s) {
    sentences.insert(s);
}

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

    for (const Symbol* sy : *s) {
        if (sy->isNT())
            throw std::runtime_error("NT found in sentence");
    }

    for (const Sequence* curr : sentences) {
        // Sequence already has equality comparison (op ==) implemented
        if (*curr == *s) {
            return true;
        }
    }
    return false;
}
```

```cpp
// end of GrammarBuilder.h
//======================================================================
```

Testcode:

```cpp
#elif TESTCASE == 5

        gb2 = new GrammarBuilder(string("G23.txt"));
        g2 = gb2->buildGrammar();
        Grammar* epsilonFree = newEpsilonFreeGrammarOf(g2);

        Language* languageG2 = languageOf(epsilonFree, 6);
        Sequence& s1 = languageG2->at(1);
        Sequence madeUpSequence{
            sp->symbolFor("a"),
            sp->symbolFor("a"),
            sp->symbolFor("a"),
            sp->symbolFor("b")
        };
        Sequence madeUpSequenceNotContained{
            sp->symbolFor("a"),
            sp->symbolFor("b"),
            sp->symbolFor("b"),
            sp->symbolFor("b")
        };

        cout << "grammar from text file:" << endl << *g2 << endl;
        cout << "newEpsilonFreeGrammarOf(g2):" << endl << *epsilonFree << endl;
        cout << "language(g2):" << endl << *languageG2 << endl;
        cout << "s1: " << s1 << endl;
        cout << "languageG2.hasSentence(s1): " << boolalpha
            << languageG2->hasSentence(&s1) << endl;
        cout << "madeUpSequence: " << madeUpSequence << endl;
        cout << "languageG2.hasSentence(madeUpSequence): " << boolalpha
            << languageG2->hasSentence(&madeUpSequence) << endl;
        cout << "madeUpSequence: " << madeUpSequenceNotContained << endl;
        cout << "languageG2.hasSentence(madeUpSequenceNotContained): " << boolalpha
            << languageG2->hasSentence(&madeUpSequenceNotContained) << endl;

#else // none of the TESTCASEs above
```

Result:

Microsoft Visual Studio Debug Console

```
START Main

symbol pool: 0 terminals and 0 nonterminals
  terminals    = {  }
  nonterminals = {  }

TESTCASE 5

grammar from text file:

G(S):
S -> a A | b A
```

```
A -> a | a S | b A A
B -> a B B | b | b S
---
VNt = { A, B, S }, deletable: {  }
VT  = { a, b }


newEpsilonFreeGrammarOf(g2):

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


language(g2):
L(G(S)): maxLength=6 {
a b a a a
a b a a a b
a b a a a a
a b a a
a a a b a a
a a a b a a
a a a b a
a a a
a
a b a
a a a b
a a a a b
a a a b a
a a a a b a
a a a a
a a
a a a a a
a a a a a a
a b a a b a
a b a a b a
a b a a b
a b
}
s1: a b a a a b
languageG2.hasSentence(s1): true
madeUpSequence: a a a b
languageG2.hasSentence(madeUpSequence): true
madeUpSequence: a b b b
languageG2.hasSentence(madeUpSequenceNotContained): false

symbol pool: 2 terminals and 3 nonterminals
  terminals    = { a, b }
  nonterminals = { S, A, B }

elapsed time: 0.009

END Main
```

Man kann erkennen, dass die länge der generierten Sätze immer gerade ist und jeder Satz gleich viele a wie b hat.

Ja kann man. Der einfachste Satz wäre `S -> b A -> b a`, indem man die erste Alternative von `A` verwendet. Bei der dritten Alternative vom NTSymbol `A` gibt es gleich viele TSymbole `b` wie NTSymbole `A`. Jedes NTSymbol `A` terminiert in TSymbol `a` oder es kommt zur Ableitung in die dritten Alternative, wodurch effektiv wider ein TSymbol `a` und ein NTSymbol `A` hinzukommen. Beim Ableiten der zweiten alternative kommt wie bei der ersten Alternative auch ein TSymbol `a` hinzu und ein neuer Satz `S`. Wenn alle Ableitungen vor dem Ableiten von `S` durchgeführt werden, befinden sich bereits gleich viele `a` wie `b` im Satz. Jedes vorkommende `S` muss daher auch gleich viele `a` wie `b` erzeugen. Das gleiche gilt auch in die andere Richtung `S -> a B`, da die Regeln im NTSymbol `B` nur `b` und `A` mit `a` und `B` getauscht haben.