

☐ Gr. 1, Dr. H. DoblerName Stefan WeißensteinerAufwand in h 9☒ Gr. 2, Dr. G. Kronberger

Punkte \_\_\_\_\_

Übungsleiter \_\_\_\_\_

**1. Grammatiken – Grundbegriffe****(8 Punkte)**

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

```

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

```

- Bestimmen Sie die Mengen  $V_T$  und  $V_N$ .
- Geben Sie den/die kürzesten Satz/Sätze an, den/die man mit dieser Grammatik bilden kann.
- Ermitteln Sie alle rekursiven Nonterminalsymbole. Geben Sie für jedes dieser Nonterminalsymbole an, ob es direkt- oder indirekt- und links-, zentral- oder rechtsrekursiv ist.
- Transformieren Sie die gegebene Grammatik in das Regelsystem der formalen Sprachen. Welche Grammatikschreibweise halten Sie für lesbarer? Begründen Sie Ihre Antwort.
- Zeichnen Sie den Syntaxbaum für folgenden Satz, verwenden Sie dazu die gegebene Grammatik  $G(DataStat)$  von oben:

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

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

**2. Konstruktion einer Grammatik****(4 Punkte)**

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

<sup>1</sup> Eine genauere Erklärung dieser Anweisung finden Sie am Ende des Übungszettels.

### 3. Oo-Implementierung von Grammatiken

(4 + 7 + 1 Punkte)

Studieren Sie die oo Implementierung von Grammatiken in *FormalLanguagesForStudents* aus dem moodle-Kurs. Die UML-Klassendiagramme dafür finden Sie in der VL-Präsentation für den FS-Teil auf den Seiten 23 und 24.

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

```
Grammar *newEpsilonFreeGrammarOf (Grammar *g);
```

Testen Sie Ihre Implementierung mit der Beispielgrammatik im Foliensatz der VL (auf S. 27).

- b) Die **Sprache**  $L$  einer Grammatik  $G$  mit dem Startsymbol  $S$ , also  $L(G(S))$ , ist nichts anderes als die **Menge aller** terminalen Ketten (**Sätze**)  $\sigma$ , die sich aus  $S$  mit den Regeln der Grammatik ableiten lassen. Eine Symbolkette wird durch ein Objekt der Klasse *Sequence* repräsentiert. Entwickeln Sie eine Klasse *Language*, die eine Menge solcher Symbolketten (die nur aus Terminalsymbolen bestehen) repräsentiert und eine Funktion

```
Language *languageOf (const Grammar *g, int maxLen);
```

die alle Sätze der Grammatik  $g$  bis zur maximalen Länge  $maxLen$  in einem *Language*-Objekt erzeugt und dieses als Ergebnis liefert. (Sie können davon ausgehen, dass die Grammatik  $g$  epsilonfrei ist.)

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

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

einbauen, haben Sie einen einfachen Mechanismus für die Syntaxanalyse.

Testen Sie Ihre Implementierungen für b) und c) mit folgender Grammatik  $G(S)$ :

```
S  → a B | b A
A  → a   | a S | b A A
B  → b   | b S | a B B
```

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

### Hintergrundinformation zur Aufgabe 1

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

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

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

Im folgenden Beispiel werden die Variablen  $A - L$  deklariert und dann mit Initialwerten belegt:

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

# 1. Grammatiken – Grundbegriffe

---

a)

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

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

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 ->  $\epsilon$  | 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



## 2. Konstruktion einer Grammatik

---

### Regelsystem

$S \rightarrow \text{OptSign LeadingDigit MiddleDigits UnevenNaturalDigit} \mid \text{OptSign UnevenNaturalDigit}$  . // man könnte auch OptSign weglassen und dafür 4 weitere Alternativen in "S" hinzufügen

$\text{OptSign} \rightarrow \epsilon \mid + \mid -$  .

$\text{MiddleDigits} \rightarrow \epsilon \mid 0 \text{ MiddleDigits} \mid \text{LeadingDigit MiddleDigits}$  .

$\text{UnevenNaturalDigit} \rightarrow 1 \mid 3 \mid 5 \mid 7 \mid 9$  .

$\text{LeadingDigit} \rightarrow \text{UnevenNaturalDigit} \mid 2 \mid 4 \mid 6 \mid 8$  .

### EBNF

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

## 3. Oo-Implementierung von Grammatiken

---

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

a)

main.cpp

(next page)

```

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 structured binding
    for (const auto& [ NTSymbol *const & nt, const SequenceSet & 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(_Pos: i);
                if (currSy->isNT() &&
                    deletable.contains(dynamic_cast<NTSymbol*>(currSy))) {
                    deletableNTindices.push_back(_Val: 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[idx: [j - symbolsRemoved]]);
                        symbolsRemoved += 1;
                    }
                }
                // don't add empty alternatives
                // also duplicates are ignored
                if (!copy->isEpsilon()) gb.addRule(nt, seq: copy);
                else delete copy;
            }
        }
    }

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

    return gb.buildGrammar();
}

```

Testcode:

```
#elif TESTCASE == 4

    gb2 = new GrammarBuilder(string("G1.txt"));
    g2 = gb2->buildGrammar();
    Grammar* epsilonFree = newEpsilonFreeGrammarOf(g2);

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

    delete epsilonFree;

#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: 0.021

END Main
```

b) and also c)

## main.cpp

```

void languageOfRecursive(
    Language* language,
    NTSymbol* const originalNTSymbol,
    const RulesMap& rules,
    Sequence* currSentence,
    int maxLen
) {
    int i = 0;
    while (i < currSentence->size() && (*currSentence)[i]->isT()) {
        i++;
    }

    // only tSymbols left?
    if (i == currSentence->size()) {
        if (currSentence->size() <= maxLen)
            language->addSentence(currSentence);
        else
            delete currSentence;
        return;
    }

    NTSymbol* ntSy = dynamic_cast<NTSymbol*>((*currSentence)[i]);

    // do same stuff recursive for all alternatives substituted
    for (Sequence * alternative : rules[ntSy])
    {
        // this alternative makes the sentence too long - skip
        if (currSentence->length() + alternative->length() - 1 > maxLen) continue;

        // ignore this alternative if it does not contribute to the language directly
        if (alternative->length() == 1 && (*alternative)[0]->isNT()
            && *originalNTSymbol == (*alternative)[0]) continue;

        Sequence* derivedSentence = new Sequence(*currSentence);
        derivedSentence->removeSymbolAt(idx: i);
        derivedSentence->append(seq: alternative);

        languageOfRecursive(language, originalNTSymbol: ntSy, rules, currSentence: derivedSentence, maxLen);
    }
    delete currSentence;
}

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

```

## Language.h

(next page)



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

#ifndef Language_h
#define Language_h

#include <vector>
#include <set>
#include <iostream>
#include "ObjectCounter.h"
#include "SequenceStuff.h"

class Language :
    private ObjectCounter<Language> {

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

private:
    SequenceSet sentences{};
    int maxLength;

public:
    Language(int maxLength);

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

#endif

// end of Language.h
//=====
```

Language.cpp

(next page)

```

// 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 SequenceSet::const_iterator it = sentences.cbegin();
    std::advance(&_Where: it, _Off: idx);
    return **it;
}

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

}

void Language::addSentence(Sequence* s) {
    if (hasSentence(s)) {
        delete s;
        return;
    }
    sentences.insert(_Val: 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;
}

// end of Language.h
//=====

```

Testcode:

```
#elif TESTCASE == 5

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

    Language* languageG2 = languageOf(g2, epsilonFree, maxlen: 6);
    Sequence& s1 = languageG2->at(1);
    Sequence madeUpSequence{
        sp->symbolFor(name: "a"),
        sp->symbolFor(name: "a"),
        sp->symbolFor(name: "b"),
        sp->symbolFor(name: "b")
    };
    Sequence madeUpSequenceNotContained{
        sp->symbolFor(name: "a"),
        sp->symbolFor(name: "b"),
        sp->symbolFor(name: "b"),
        sp->symbolFor(name: "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;

    delete epsilonFree;
    delete languageG2;

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

```

newEpsilonFreeGrammarOf(g2):
G(S):
S -> a B | b A
B -> a B B | b | b S
A -> a | a S | b A A
---
VNT = { A, B, S }, deletable: { }
VT = { a, b }

language(g2):
L(G(S)): maxLength=6 {
a a b b a b
a b
a a b b b a
a a b a b b
a a b b
a a b b a b
a a b b b a
a a a b b b
a b a a b b
a b b a
a b b a a b
a b a b b a
a b b a b a
a b a b a b
a b a b
b a a b b a
b a
a b b b a a
b a a a b b
b a a b
b a a b a b
b a b a
b a b a a b
b b a a
b b a a b a
b b a b a a
b a b b a a
b a b a b a
b b a a a b
b b a a b a
b b a a a b
b b b a a a
\Sys}
\Sys s1: a b
\Sys languageG2.hasSentence(s1): true
\Sys madeUpSequence: a a b b
\Sys languageG2.hasSentence(madeUpSequence): true
\Sys madeUpSequence: a b b b
\Sys languageG2.hasSentence(madeUpSequenceNotContained): false
symbol pool: 2 terminals and 3 nonterminals
terminals = { a, b }
nonterminals = { S, A, B }

elapsed time: 0.024

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. Jedes NT  $B$  terminiert in genau ein  $b$  und jedes NT  $A$  terminiert in genau ein  $a$ . Wenn die Ableitung mit  $S \rightarrow a B$  anfängt, dann befindet sich schon ein  $a$  im Satz und das  $B$  wird schlussendlich zu einem  $b$ . Bei der dritten Alternative von  $B$  kommen ein  $a$  sowie zwei  $B$  hinzu. Die Ableitung  $S \rightarrow a B \rightarrow a a B B$  hat 2  $a$  und 2  $B$  und wir wissen bereits, dass jedes  $B$  in genau ein  $b$  terminiert oder es geschieht wieder die gleiche Ableitung von  $B \rightarrow a B B$ , wodurch effektiv nur 1 weiteres  $a$  und 1 weiteres  $B$  hinzukommen. Wenn schlussendlich alle  $B$  in  $b$  abgeleitet werden, gibt es gleich viele  $a$  wie  $b$ . Das gleiche gilt auch für die Ableitung  $B \rightarrow b S$ , da wie bei der ersten Alternative ein weiteres  $B$  mit  $b$  ersetzt wird und ein weiterer Satz  $S$  dazukommt, der später auch wieder in gleich viele  $a$  und  $b$  abgeleitet werden kann und jedes  $B$  wieder in ein  $b$  abgeleitet wird oder in ein  $b$  und ein  $S$ . Das gleiche gilt auch in die andere Richtung  $S \rightarrow a B$ , da die Regeln im NTSymbol  $B$  nur  $b$  und  $A$  mit  $a$  und  $B$  getauscht haben.