UNIVERSITÄT
BAYREUTH

University of Bayreuth

Institute for Computer Science

Bachelor Thesis

in Applied Computer Science

Topic:	Randomly Generated CFGs, The CYK-Algorithm And A CLI
Author:	Andreas Braun <andreasbraun5@aol.com> Matrikel-Nr. 1200197
Version date:	February 26, 2017
1. Supervisor:	Prof. Dr. Wim Martens
2. Supervisor:	Tina Trautner

To my parents.

Abstract

The abstract of this thesis will be found here.

Zusammenfassung

Hier steht die Zusammenfassung dieser Bachelorarbeit.

Contents

1 Introduction

Let there be a gramar $G = (V,\ \Sigma,\ S,\ P)$ in chomsky normal form (CNF).
V is a finite set of variables.
Σ is an alphabet.
S is the starting symbol and $S \in V$.
P is a finite set of rules: $P \subseteq V \times (V \cup \Sigma)^*$. G is in CNF and therefore, more specifically, it holds: $P \ \subseteq \ V \times (V^2 \cup \Sigma)$.

Vs are Variables like "A, B, ...".
$(V^2 \cup \ \Sigma)^*$ are terminals like "a, b, ..." and compound variables like "AB, BS, AC, ...".

Let there be a word $w \in \Sigma^*$, a language $L(G)$ and a grammar G in CNF.

1.1 Forward Problem vs. Backward Problem

Forward Problem ($G \xrightarrow{derivation} w$):
Informal definition: "Forming a derivation from a root node to a final sentence." [Duda 8.6.3 page 426]
Input: Grammar G in CNF.
Output: Derivation d that shows implicitly $w \subseteq L$.

Backward Problem = Parsing ($w \overset{?}{\subseteq} L(G)$):
Informal definition: "Given a particular w, find a derivation in G that leads to w. This process, called parsing, is virtually always much more difficult than forming a derivation." [Duda 8.6.3 page 426]
Input: w and a grammar G in CNF.
Output: $w \subseteq L(G) \Longrightarrow$ derivation d.

1.2 Parsing: Bottom-Up vs Top-Down

Bottom-Up: Bottom-Up parsing is "the general method used in the Cocke-Younger-Kasami(CYK) algorithm, which fills a parse table from the "bottom up"." (Bottom up means starting from the leaves.) [Duda 8.6.3 page 426]

Top-Down: "Top-Down parsing starts with the root node and successively applies productions from P, with the goal of finding a derivation of the test sentence w. Because it is rare indeed that the sentence is derived in the first production attempted, it is necessary to specify some criteria to guide the choice of which rewrite rule to

apply. Such criteria could include beginning the parse at the first (left) character in the sentence (i.e., finding a small set of rewrite rules that yield the first character), then iteratively expanding the production to derive subsequent characters, or instead starting at the last (right) character in the sentence." [Duda 8.6.3 page 428]

1.3 Scope of this thesis

The starting point of this thesis was to get a command line interface (CLI) tool to automatically generate $exercises = (grammar,\ word,\ parse\ table,\ derivation\ tree)$, which are used to test if the students have understood the way of working of the CYK algorithm. A scoring model is used to evaluate the generated exercises regarding their usability in an exam.

This alone doesn't meet the requirements for being an adequate topic for a bachelor thesis. Therefore the task of finding a clever algorithm to get exercises with a high chance of being usable as an exam exercise was added.

2 Scoring Model

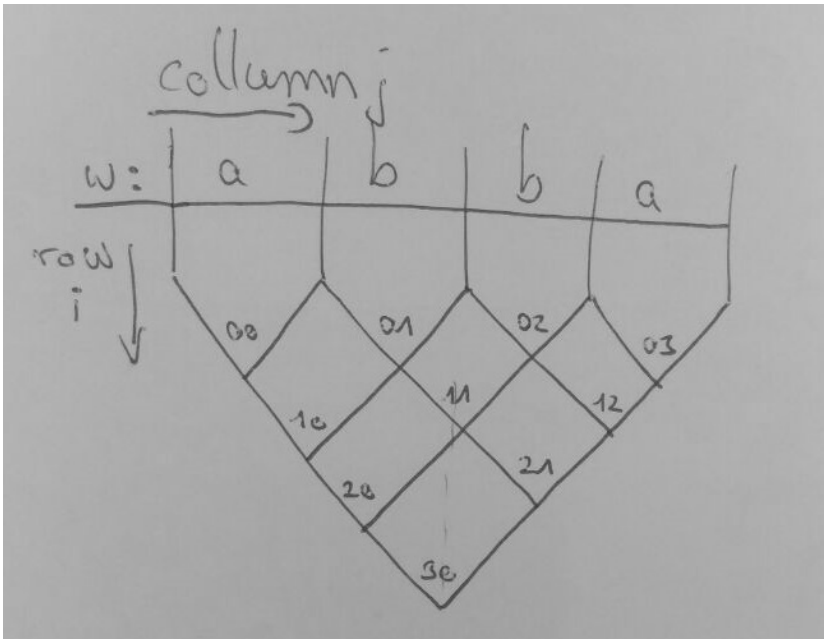
2.1 öalkjfds

2.2 öaslfajdsl

3 Algorithms

Define $[i,\ j]$:
$[i,\ j] := \{i,\ i+1, ..., j-1,\ j\} \subseteq \mathbb{N}_{\geq 0}$.

Define $Pyramid$:
$Pyramid := \{cell_{i,j} \mid i \in \mathbb{N}_{\geq 0},\ j \in [0,\ j_{max} - i],\ i_{max} = j_{max} = |word| - 1\}$.
$cell_{i,j} = \{c \mid c \subseteq V\}$.
$EmptyPyramid \Leftrightarrow \forall i\ \forall j\ cell_{i,j} = \emptyset$.
Regarding one $cell_{i,j}$: $cell_{i,j} = cellDown$, $cell_{i-1,j} = cellUpperLeft$ and $cell_{i-1,j+1} = cellUpperRight$



Algorithm 1: distributeRhse2

Input: $Rhse \subseteq (V^2 \cup \Sigma),\ i \in \mathbb{N},\ j \in \mathbb{N}$

Output: *Grammar G in CNF with uniform randomly distributed Rhse's.*

```
1 choose n uniform randomly in [i, j];
2 choose V_add := uniform random subset of size n from V;
3 P = P ∪ {"v ⟶ rhse" | v ∈ V_add, rhse ∈ Rhse};
4 return G;
```

Algorithm 2: calculateSubsetForCell

Input: $cell_{i,j} \in pyramid$

Output: $V_{i,j} \subseteq V^2$

```
1 V_{i,j} = ∅;
2 for m := i − 1 → 0 do
3 |   V_{i,j} = V_{i,j} ∪ {X | X ⟶ YZ, Y ∈ V_{m,j}, Z ∈ V_{i−m−1,m+j+1}};
4 end
5 return V_{i,j};
```

Algorithm 3: checkForceCombinationForCell

Input: $cell_{i,j} \subseteq V,\ cell_{i-1,j} \subseteq V,\ cell_{i-1,j+1} \subseteq V,\ P \subseteq V \times (V^2 \cup \Sigma)$

Output: $varsForcing \subseteq V$

```
1  varsForcing ⊆ V = ∅;
2  varComp = {XY | X ∈ cell_{i−1,j} ∧ Y ∈ cell_{i−1,j+1}};
3  foreach v ∈ cell_{i,j} do
4  |   prods = {p | p ⊆ P, v is left in p};
5  |   rhses = {rhse | rhse is right in p ∈ prods};
6  |   if varComp ⊈ rhses then
7  |   |   varsForcing = varsForcing ∪ v;
8  |   end
9  end
10 return varsForcing;
```

Input: $cell_{i,j} = cellDown$, $cell_{i-1,j} = cellUpperLeft$ and $cell_{i-1,j+1} = cellUpperRight$

Things like the $G = (V, \Sigma, S, P)$ can be assumed as known.
$P = P \cup \{distribute\ \{\sigma \mid \sigma \in w\}\ uniform\ randomly\ over\ \{v \mid v \in V\}\}$ which equals *distributeRhse* method.
Bias is only allowed top vs down regarding the pyramid. Not left or right bias.

Algorithm 4: GeneratorGrammarDiceRollMartens

Input: $Word\ w \in \Sigma^*,\ P \subseteq V \times (V^2 \cup \Sigma) = \emptyset,$
Output: $Grammar\ G\ in\ CNF$

```
1  P = P ∪ {distribute {σ | σ ∈ w} over {v | v ∈ V}};
2  pyramid = CYK(G, w);
3  for i := 1 to i_max do
4      J ⊆ ℕ = ∅;
5      cellSet ⊆ V² = ∅;
6      while |J| < j_max − i do
7          choose one j ∉ J uniform randomly in [0, j_max − i];
8          J = J ∪ j;
9          cellSet = calculateSubsetForCell(pyramid, i, j);
10         P = P ∪ {distribute {vc | vc ∈ cellSet} over {v | v ∈ V}};
11         pyramid = CYK(G, w);
12         evaluate stopping criteria regarding the pyramid;
13         if stopping criteria = true then
14             return G;
15         end
16     end
17 end
18 return G;
```

Line 2: Fills the i=0 row of the pyramid.
Line 4: Instead of going from left to right, choose j uniform randomly with the restrictions that one cell is only visited one time.
Note: Maybe modify algorithm to also work with the threshold.

Algorithm 5: GeneratorGrammarDiceRollMartens2

Input: $Word\ w \in \Sigma^*,\ P \subseteq V \times (V^2 \cup \Sigma) = \emptyset$,

Output: *Grammar G in CNF*

```
1  P = P ∪ {distribute {σ | σ ∈ w} over {v | v ∈ V}};
2  pyramid = CYK(G, word);
3  for i := 1 → i_max do
4      for j := 0 → j_max − i do
5          rowSet = rowSet ∪ {(XY, i) | X, Y ∈ V,
           XY ∈ calculateSubsetForCell(Pyramid, i, j)};
6      end
7      while threshold_i = false do
8          choose one vc ∈ rowSet with priority, depending on i,
           uniform randomly;
9          P = P ∪ {distribute {vc | vc ∈ cellSet} over {v | v ∈ V}};
10         pyramid = CYK(G, w);
11         evaluate and update threshold_i, regarding line i;
12         evaluate stopping criteria, regarding the pyramid;
13         if stopping criteria = true then
14             return G;
15         end
16     end
17 end
18 return G;
```

Line 2: Fills the i=0 row of the pyramid.

Line 6: $(AB,1),(AB,2),(BC,3)... \in sub \rightarrow$ multiple occurrences of AB are allowed. This considers "more important" compound variables.

Line 9: One vc can be chosen several times.

Note: Threshold: Linear or log function $f(i)$?

Note: Priority mechanism: In line $i+1$ the $k = \{(A,l) \mid (A,l) \in sub,\ l = i\}$ are preferred over the $m = \{(A,n) \mid (A,n) \in sub,\ n < i\}$. In what way are they preferred? Using some kind of factor to weight the i of (A,i).

Algorithm 6: GeneratorGrammarDiceRollMartensFeb24

Input: $word \in \Sigma^*,\ V,\ \Sigma,\ S,\ P = \emptyset,\ minCount\Sigma,\ maxCount\Sigma,$
$minCountVarComp,\ maxCountVarComp$

Output: G

```
1  G = (V, Σ, S, P);
2  G = distributeRhse(G, Σ, minCountΣ, maxCountΣ);
3  Pyramid = CYK.calculatePyramid(G, word);
4  for i := 1 → i_max do
5      for j := 0 → j_max − i do
6          sub = calculateSubsetForCell(Pyramid, i, j);
7          foreach vc ∈ sub do
8              distributeRhse(G, vc, minCountVarComp, maxCountVarComp);
9              Pyramid = CYK.calculatePyramid(G, word);
10             Evaluate stopping criteria;
11             if stopping criteria = true then
12                 return G;
13             end
14         end
15     end
16 end
17 return G;
```

Line 3: Fills the i=0 row of the pyramid.
line 5: Instead of going from left to right, choose j uniform randomly with the restrictions that one cell is only visited one time.
Note: The algorithm tends to finish already within $i = 1$ loop.

Algorithm 7: GeneratorGrammarDiceRollMartens2Feb24

Input: $word \in \Sigma^*,\ V,\ \Sigma,\ S,\ P = \emptyset,\ minCount\Sigma,\ maxCount\Sigma,$
$minCountVarComp,\ maxCountVarComp$

Output: G

1 $G = (V, \Sigma, S, P)$;
2 $G = distributeRhse(G,\ \Sigma,\ minCount\Sigma,\ maxCount\Sigma)$;
3 $Pyramid = CYK.calculatePyramid(G,\ word)$;
4 $sub = \emptyset$;
5 **for** $i := 1\ \rightarrow\ i_{max}$ **do**
6 **for** $j := 0\ \rightarrow\ j_{max} - i$ **do**
7 $sub = sub \cup \{(A, i) \mid A \in calculateSubsetForCell(Pyramid,\ i,\ j)\}$;
8 **end**
9 $sub_b = \{B \mid B \in sub,\ sub_b\ models\ i\text{-}dependent\ priority\ mechanism\}$;
10 **while** $cell_{i_{max},0} = \emptyset\ \wedge\ threshold_i = false$ **do**
11 *choose one vc uniform randomly* $\in sub_b$;
12 $distributeRhse(G,\ vc,\ minCountVarComp,\ maxCountVarComp)$;
13 $Pyramid = CYK.calculatePyramid(G,\ word)$;
14 *evaluate and update* $threshold_i$;
15 **end**
16 **end**
17 **return** G;

Line 3: Fills the i=0 row of the pyramid.
Line 7: $(AB, 1), (AB, 2), (BC, 3)... \in sub \rightarrow$ multiple occurrences of AB are allowed. This considers "more important" compound variables.
Line 11: One vc can be chosen several times.
Note: Threshold: Linear or log function $f(i)$?
Note: Priority mechanism: In line $i + 1$ the $k = \{(A, l) \mid (A, l) \in sub,\ l = i\}$ are preferred over the $m = \{(A, n) \mid (A, n) \in sub,\ n < i\}$. In what way are they preferred? Using some kind of factor to weight the i of (A, i).

```
1 Algorithm: CYK.calculateSetVAdvanced
2 Input: grammar, word;
3 Output: Set<VariableK>[][] cYKMatrix;
4
5 Set<VariableK>[][] cYKMatrix = new Set<VariableK>[wordSize][wordSize];
6 cYKMatrix = calculateCYKMatrix;
7 return cYKMatrix;
```

Listing 1: CYK.calculateSetVAdvanced

```
1  Algorithm: GeneratorGrammarDiceRollOnly
2  Input: settings;
3  Output: grammar;
4  Note: A lot of productions are generated, that later on are not needed
5  for parsing the specific word.
6
7  Grammar grammar = new Grammar();
8  // Part1: Distribute the terminals.
9  grammar = distributeDiceRollRightHandSideElements(
10         grammar, settingsTerminals, minCountTerminals,
11         maxCountTerminals, settingsListVariables);
12 // Part2: Distribute the compound variables.
13 Set<Variables> vars = settings.getVariables();
14 Set<VariablesCompound> setVarComp;
15 setVarComp = calculate all the possible tupels of ({vars}, {vars});
16 grammar = distributeDiceRollRightHandSideElements(
17         grammar, settingsTerminals, minCountVariableCompound,
18         maxCountVariableCompound, setVarComp);
19 return grammar
```

Listing 2: GeneratorGrammarDiceRollOnly

```
1  Algorithm: GeneratorGrammarDiceRollOnlyBias
2  Input: settings;
3  Output: grammar;
4  Note: A lot of productions are generated, that later on are not needed
5  for parsing the specific word.
6
7  Grammar grammar = new Grammar();
8  // Distribute the terminals.
9  grammar = distributeDiceRollRightHandSideElementsBias(
10         grammar, settingsTerminals, settingsMinCountTerminals,
11         settingsMCountTerminals, settingsListVars, settingsFavouritism);
12
13 // Distribute the compound variables.
14 Set<VariablesCompound> setVarComp;
15 setVarComp = calculate all the possible tupels of ({vars}, {vars});
16 grammar = distributeDiceRollRightHandSideElementsBias(
17         grammar, varComp, settingsMinCountVars,
18         settingsMaxCountVars, settingsListVars, settingsFavouritism);
19 return grammar;
```

Listing 3: GeneratorGrammarDiceRollOnlyBias

```
1  Algorithm: distributeDiceRollRightHandSideElementsBias
2  Input: grammar, setRhse, minCount, maxCount, listVars, favouritismList;
3  Output: grammar;
4  Note: Because of dice rolling anyways and lots of grammars being
5  generated, no rhse is added if the production already exists.
6
7  // Calculate the bloated varSet.
8  List<Variable> varsBloated;
9  for(Variables varTemp :  settings.getVariables()){
10         tempFavour = randomly pick favouritism[i];
11         varsBloated.add({tempFavour times varTemp});
12         favouritism.remove(tempFavour);
13 }
14 // Because of dice rolling anyways and lot of grammars being generated,
15 // just no rhse is added if the production already exists.
16 grammar = distributeDiceRollRightHandSideElements( grammar,
17         varsBloated, minCount, maxCount, listVars );
18 return grammar;
```

Listing 4: distributeDiceRollRightHandSideElementsBias

Algorithm 8: distributeDiceRollRightHandSideElementsBias

Input: G, $rhse \subseteq RHSE$, $0 \leq minCount \leq maxCount \leq |G.V|$, $favouritism = \{x \mid x \in \mathrm{N} \wedge |favouritism| = |G.V|\}$

Output: G

1 $favour = \{(v, f) \mid v \in V \wedge f \in favouritism \wedge$ *tupel are created via dice roll*$\}$;

2 $varsBloated_b = \emptyset$;

3 **foreach** *fav in favour* **do**

4 $\quad varsBloated_b = varsBloated_b \uplus \{v^f\}$;

5 **end**

6 **return** $distributeDiceRollRhse(G, MISTAKEHEREvarsBloated_b, minCount, maxCount)$;

7 Still working on. One more Pprameter needed for distributeDiceRollRighthandSideElement. Parameter V that defines the variables the rhse are added to. OR make this algorithm independent.

Line 6: Note that $varsBloated_b$ is a multiset, but should actually be a set. Exceptions causing a duplicate production to the grammar are not relevant because G.P is a set.

Description of the checks here.
All test of the GrammarValidityChecker class are based on the simple setV matrix.

isValid = isWordProducible && isExamConstraints && isGrammarRestrictions

isWordProducible = CYK.algorithmAdvanced()

isExamConstraints = isRightCellCombinationsForced && isMaxSumOfProductionsCount && isMaxSumOfVarsInPyramidCount && countRightCellCombinationsForced

isGrammarRestrictions = isSizeOfWordCount && isMaxNumberOfVarsPerCellCount

```
1 Algorithm: checksumOfProductions
2 Input: grammar, maxSumOfProduction;
3 Output: isSumOfProductions;
4
5 return grammar.getProductionsAsList().size() <= maxSumOfProductions;
```

Listing 5: checksumOfProductions

```
1  Algorithm: checkMaxNumberOfVarsPerCell
2  Input: setVSimple, maxNumberOfVarsPerCell;
3  Output: isMaxNumberOfVarsPerCell;
4  Note: Checking for maxNumberOfVarsPerCell <= zero isn't allowed;
5
6  int tempMaxNumberOfVarsPerCell = 0;
7  int wordLength = tempSetV[0].length;
8  for ( int i = 0; i < wordLength; i++ ) {
9          for ( int j = 0; j < wordLength; j++ ) {
10                 if ( tempSetV[i][j].size() > numberOfVarsPerCell ) {
11                         numberOfVarsPerCell = tempSetV[i][j].size();
12                 }
13         }
14 }
15 return tempMaxNumberOfVarsPerCell <= maxNumberOfVarsPerCell;
```

Listing 6: checkMaxNumberOfVarsPerCell

```
1  Algorithm: checkMaxSumOfVarsInPyramid
2  Input: setVSimple, maxSumOfVarsInPyramid;
3  Output: isMaxSumOfVarsInPyramid;
4
5  // put all vars of the matrix into one list and use its length.
6  List<Varaible> allVarsList = new ArrayList<>();
7  for ( int i = 0; i < setVSimple.length; i++ ) {
8          for ( int j = 0; j < setVSimple.length; j++ ) {
9                  tempVars.addAll( setVSimple[i][j] );
10         }
11 }
12 return allVarsList.size() <= maxSumOfVarsInPyramid;
```

Listing 7: checkMaxSumOfVarsInPyramid

```
Algorithm: rightCellCombinationsForced
Input: setVSimple, minCountForced, grammar;
Output: isForced, countForced, setVSimpleVarsThatForce;
Note: Keep in mind that the setV matrix is a upper right matrix. But the
description of how the algorithm works is done, as if the setV pyramid
points downwards (reflection on the diagonal + rotation to the left).
Regarding one cell, its upper left cell and its upper right cell
are looked at. setV[i][j] = down cell. setV[i + 1][j] = upper right cell
setV[i][j - 1] = upper left cell.

int countForced = 0;
Set<Variable>[][] setVMarkedVarsThatForce;
for(Cell cell : setVSimple){
        // Trivial cases that would fulfil the restrictions each time.
        Ignore the upper two rows of the pyramid;
        isRightCellCombinationForced = true;
        if(!upperLeftCell.isEmpty() && !upperRightCell.isEmpty()) {
                break;
        }
        setVariableCompound = calculate all the possible tupels of
                ({varLeft}, {varRight});
        for(Variable var : cellToBeVisited) {
                varDownProdList = grammar.getProdList(varDown);
                for(VariableCompound varComp : setVariableCompound) {
                        for(Production prod : varDownProdList){
                                if(prod.getRhse() == varComp) {
                                        isForced = false;
                                }
                        }
                }
                if(isRightCellCombinationForced) {
                        rightCellCombinationsForced++;
                        // Cell has index i and j.
                        setVMarkedVarsThatForce[i][j].add(var)
                }
        }
}
boolean isForced = countForced >= minCountRightCellCombinationsForced;
return isForced, countForced, setVMarkedVarsThatForce;
```

Listing 8: rightCellCombinationsForced

```
1  Algorithm: Util.removeUselessProductions
2  Input: grammar, setVSimple, word
3  Output: grammar
4  Note: Very similar to the calculateSetVAdvanced algorithm. Additionally
5  to storing the k, it is also saved, which production have been used.
6  All productions that haven't been need are removed, from the grammar.
7
8  Set<Production> allProductions = grammar.getProductions();
9  Set<Production> onlyUsefulProductions;
10 onlyUsefulProductions = calculate useful productions with the input of
11         grammar, setVSimple and word ;
12 grammar.remove(allProductions);
13 return grammar.add(onlyUsefulProductions);
```

Listing 9: Util.removeUselessProductions

4 Implementation

5 Notes

The informal goal is to find a suitable combination of a grammar and a word that meets the demands of an exam exercise. Also the CYK pyramid and one derivation tree of the word must be generated automatically as a "solution picture".
Firstly the exam exercise must have an upper limit of variables per cell while computing the CYK-pyramid.
Secondly the exam exercise must have one or more "special properties" so that it can be checked if the students have clearly understood the algorithm, e.g. "Excluding the possibility of luck."

The more formal goal is identify and determine parameters that in general can be used to define the properties of a grammar, so that the demanded restrictions are met. Also parameters could be identified for words, but "which is less likely to contribute, than the parameters of the grammar." [Second appointment with Martens]

Some introductory stuff:

Possible basic approaches for getting these parameters are the Rejection Sampling method and the "Tina+Wim" method.
Also backtracking plays some role, but right now I don't know where to put it. Backtracking is underapproach to Rejection Sampling.
Note: Starting with one half of a word and one half of a grammar.

Identify restrictions (=parameters) regarding the grammar.
Maybe find restrictions regarding the words, too.

Procedures for automated generation. Each generation procedure considers different restrictions and restriction combinations. The restrictions within one generation procedure can be optimized on its own. Up till now:
Generating grammars: DiceRolling, ...
Generating words: DiceRolling, ...

Parameter optimisation via theoretical and/or benchmarking approach.
Benchmarking = generate N grammars and test them, (N=100000).
Define a success rate and try to increase it.

The overall strategy is as following:

1.) Identify theoretically a restriction/parameter for the grammar. Think about the influence it will have. Think also about correlations between the restrictions.
2.) Validate the theoretical conclusion with the benchmark. Test out the influence of this parameter upon the success rate. Try different parameter settings.

The ordering of step 1 and step 2 can be changed.

Used software

Listings

Following are some interesting classes referenced in the thesis that were too long to fit into the text.

Tables

This section contains all tables referenced in this thesis.

References

[1] JSR 220: Enterprise Java Beans 3.0 https://jcp.org/en/jsr/detail?id=220, 09/09/2015