UNIVERSITÄT
BAYREUTH

University of Bayreuth

Institute for Computer Science

Bachelor Thesis

in Applied Computer Science

Topic:	A Constrained CYK Instances Generator: Implementation and Evaluation
Author:	Andreas Braun <www.github.com/AndreasBraun5> Matrikel-Nr. 1200197
Version date:	June 23, 2017
1. Supervisor:	Prof. Dr. Wim Martens
2. Supervisor:	M.Sc. 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

1.1 Motivation

The starting point of this thesis was to get a command line interface (CLI) tool to automatically generate the 4-tuples $exercise = (grammar,\ word,\ parse\,table,\ derivation\,tree)$, which are used to test if the students have understood the way of working of the CYK algorithm.
Various implementations of the Cocke-Younger-Kasami (CYK) algorithm can be found. Nevertheless none of them seemed to meet the easy to use requirements to automatically generate suitable *exercise*s, that afterwards also could be modified as wanted.
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 *exercise*s with a high chance of being suitable as an exam exercise was added.

1.2 Grammar in Chromsky Normal Form

Let there be a grammar $G = (V,\ \Sigma,\ S,\ P)$ for which the following holds:

- V is a finite set of variables.
- Σ is an alphabet – called terminals.
- S is the start symbol and $S \in V$.
- P is a finite set of rules: $P \subseteq V \times (V \cup \Sigma)^*$ – called productions.

Further it is assumed that the productions are more restricted and it holds: $P \subseteq V \times (V^2 \cup \Sigma)$. Additionally let there be a word $w \in \Sigma^*$ and a language $L(G)$ of the Grammar G.
Regarding further convenience for explaining the following default values are true:

- $V = \{A, B, ...\}$
- $(V^2 \cup\ \Sigma)^* = \{a, b, ...\} \cup \{AB, BS, AC, ...\}$

Moreover in the context while talking about sets, a set is always described beginning with an upper case letter, while one specific element of a set is described beginning with a lower case letter. Example: A "$Pyramid$" is a set consisting of multiple "$Cell$"s, which again is a subset of the set of variables "V". A "$cellElement$" is one specific element of a "$Cell$". (For further reasoning behind this example see chapter XXX "help data structure")

1.3 General approaches

Two basic approaches, that may help finding a good algorithm are explained informally.

1.3.1 Forward Problem & Backward Problem

The Forward Problem and the Backward Problem are two ways as how to determine if $w \in L(G)$.

Forward Problem ($G \xrightarrow{derivation} w$):
Input: Grammar G in CNF.
Output: Derivation d that shows implicitly $w \subseteq L$.

Informal description: It is called Forward Problem, if you are given a grammar G and form a derivation from its root node to a final word w. The final word w is always element of $L(G)$.
[Informal definition: "Forming a derivation from a root node to a final sentence.", Duda 8.6.3 page 426]

Backward Problem = Parsing ($w \overset{?}{\subseteq} L(G)$):
Input: w and a grammar G in CNF.
Output: $w \subseteq L(G) \Longrightarrow$ derivation d.

Informal description: It is called Backward Problem, if you are given a word w and want to determine if it is element of $L(G)$. "This process, called parsing, is virtually always much more difficult than forming a derivation."
[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]]

1.3.2 Parsing Bottom-Up & Top-Down

There are again two ways in which you can classify the approach for parsing.

Bottom-Up: Bottom-Up parsing means to start parsing from the leaves up to the node.

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

Top-Down: Top-Down parsing means to start parsing from the node down to the leaves.

Informal description: "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]

2 Simple Scoring Model

Short preface to the rationale about the scoring model. Add diversity criteria = homogenity of the cells to the scoring matrix.

2.1 Elimination Criteria and Selection Criteria

Success rates: Producibility: $w \subseteq L(G)$
Grammar restricitons: $n = |w|$, maxNumberOfVarsPerCell; **Delete SuccessRates-GrammarRestrictions class. Move maxNumberOfVarsPerCell to exam restrictions class and use n only as parameter.**
Exam restrictions: rightCellCombinationsForcedCount, maxSumOfProductions, maxSumOfVarsInPyramid

Picture of used scoring model without weights here. Maybe one picture together with the next subsection.

2.2 Weighting of the criteria

Picture of the final used scoring model with weights here.

2.3 Direct Ranking vs. Preference Analysis vs. ...

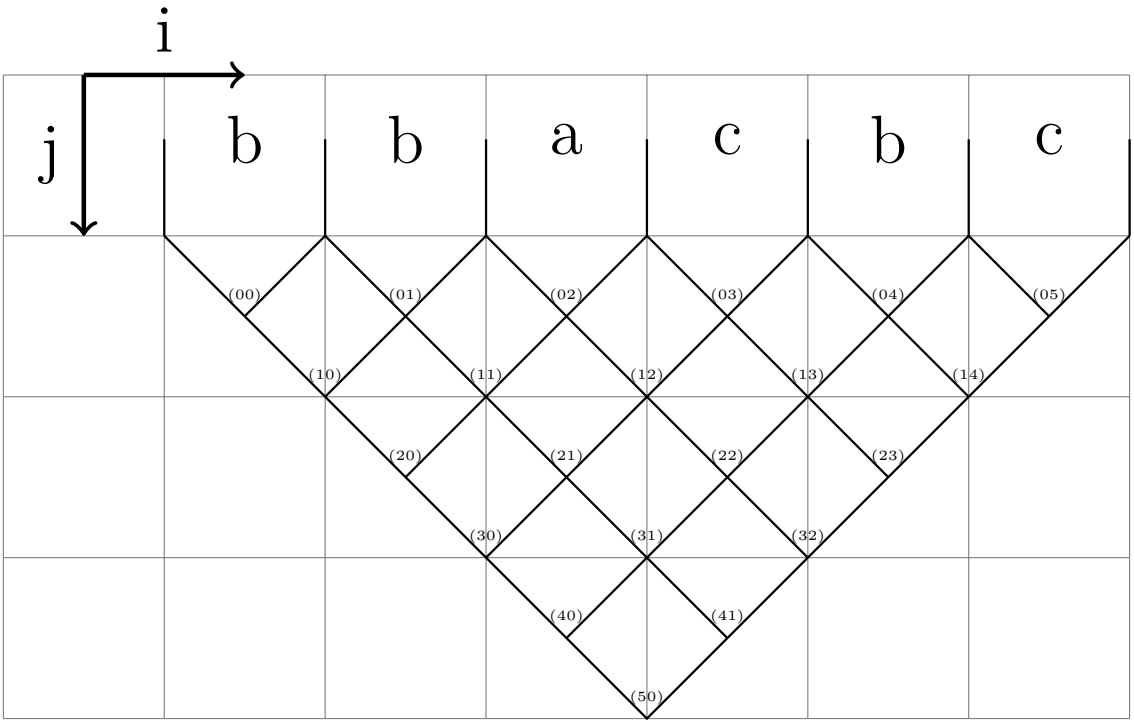
What method is used to compare the results out of the scoring model.
Direct Ranking is the simplest way.

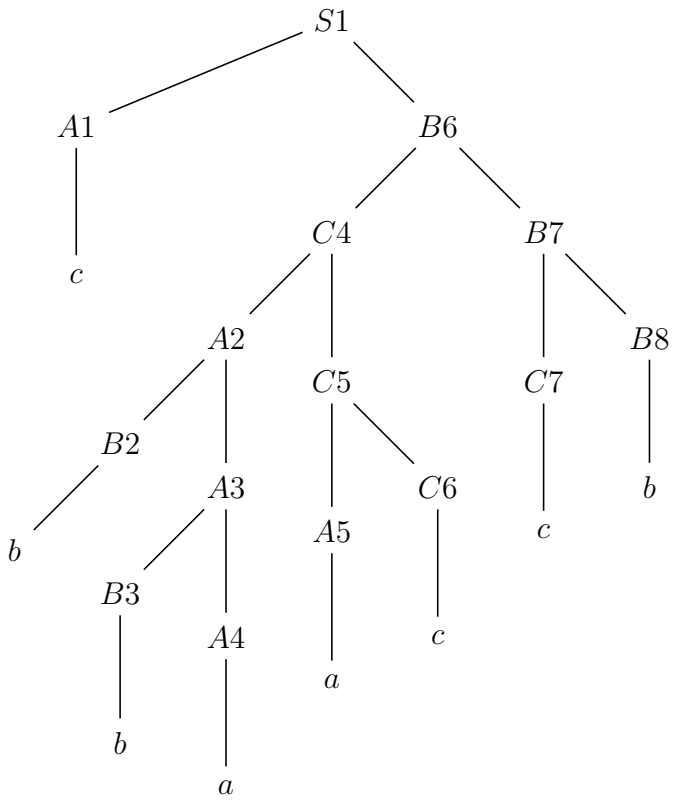
3 Algorithms

3.1 Help Data Structure Pyramid and Others

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

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




S1
A1
B6
c
C4
B7
A2
C5
C7
B8
B2
A3
C6
b
A5
c
b
B3
A4
c
a
b
a

3.2 Exam Exercise Generating Algorithms

3.2.1 Algorithm: AlgorithmName

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 the $distributeRhse$ module.
Bias is only allowed top vs down regarding the pyramid. No left or right bias intended yet.
(A), (B), ... represent exchangeable algorithm modules. Make hyperrefs für (A)

3.2.1.1 Basic Idea

3.2.1.2 Tweak Idea 1 for Algorithm

3.2.1.3 Tweak Idea 2 for Algorithm

3.2.1.4 Finished Algorithm break

3.2.2 Algorithm: GeneratorGrammarDiceRollOnly

Very naive way of generating grammars. This is intended to be the starting point for our algorithms we find. Each found algorithm must have a higher score than this algorithm or otherwise it would be worse than simple dice rolling and then removing the useless productions.

Algorithm 1: GeneratorGrammarDiceRollOnly

Input: Word $w \in \Sigma^*$

Output: Set of productions P

1 $P = \emptyset$; // $P \subseteq V \times (V^2 \cup \Sigma)$

2 $P = Distribute(\Sigma,\ V)$; (A)

3 $P = P \cup Distribute(V^2,\ V)$; (B)

4 $P = P \setminus \{p \mid p = (v,\ x) \in P \wedge \nexists\ Cell \in Pyramid :\ v \in Cell\}$;

5 **return** P;

Line 4: Removes all useless productions. But unreachable productions still possible.

Remove line 4 because this has nothing to do with the algorithm and write it into another procedure. In the code this is done only at the ResultCalculator.createChunkResult.
Does the output $P \subseteq V \times (V^2 \cup \Sigma)$ imply that G is in CNF? CNF does only have useful variables [TI script Def. 8.3 page 210] vs. $P \subseteq V \times (V^2 \cup \Sigma)$.
More of a problem is that the set P is not necessarily in CNF. It is possible that there are unreachable variables – from the starting variable.

3.2.3 Algorithm: BottomUp GeneratorGrammarDiceRollMartens

Algorithm 2: GeneratorGrammarDiceRollVar1

Input: Word $w \in \Sigma^*$

Output: Set of productions P

```
1  P = ∅;  // P ⊆ V × (V² ∪ Σ)
2  P = Distribute(Σ, V); (A)
3  Pyramid = CYK(G, w);
4  for i := 1 to i_max do
5      J = ∅;  // J ⊆ ℕ
6      CellSet = ∅;  // CellSet ⊆ V²
7      while |J| < j_max − i do
8          choose one j ∉ J uniform randomly in [0, j_max − i] ;
9          J = J ∪ j;
10         CellSet = CalculateSubsetForCell(Pyramid, i, j);
11         P = P ∪ Distribute(CellSet, V); (B)
12         Pyramid = CYK(G, w);
13         if stopping criteria met (C) then
14             return P;
15         end
16     end
17 end
18 return P;
```

Line 2: Fills the i=0 row of the pyramid.
Line 8: A cell is only visited only once.
Note: Maybe modify algorithm to also work with the threshold.

Stopping criteria regarding the entire pyramid.

Algorithm 3: GeneratorGrammarDiceRollVar2

Input: Word $w \in \Sigma^*$

Output: Set of productions P

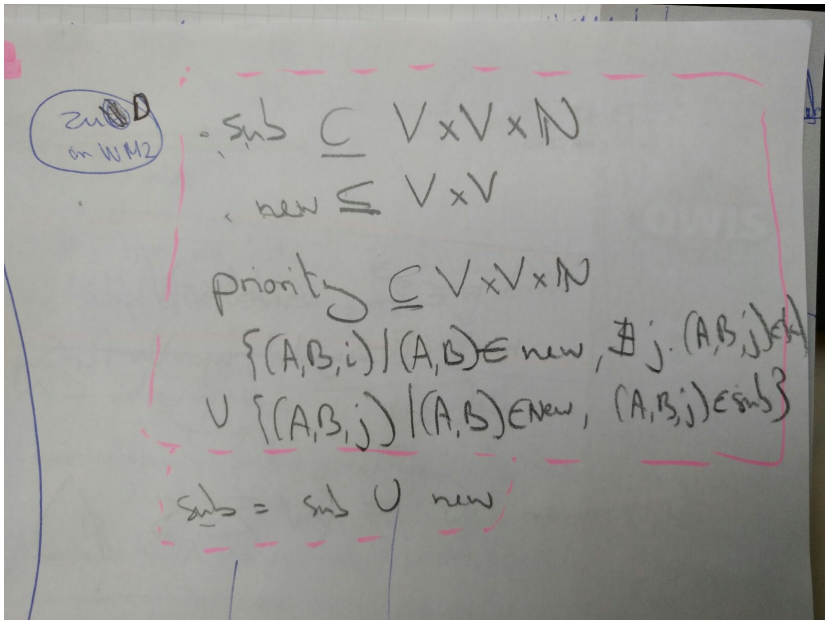
1 $P = \emptyset$; // $P \subseteq V \times (V^2 \cup \Sigma)$
2 $RowSet = \emptyset$; // $RowSet \subseteq \{(XY, i) \mid X, Y \in V \wedge i \in \mathbb{N}\}$
3 $P = Distribute(\Sigma,\ V)$; (A)
4 $Pyramid = CYK(G,\ w)$;
5 **for** $i := 1$ ***to*** i_{max} **do**
6 **for** $j := 0$ ***to*** $j_{max} - i$ **do**
7 $RowSet = RowSet \cup \{(XY, i) \mid XY \in CalculateSubsetForCell(Pyramid,\ i,\ j)\}$;
8 **end**
9 **while** $threshold_i$ *not reached* **do**
10 *choose one xy out of* $(XY,\ i) \in RowSet\ uniform\ randomly\ with\ probability\ depending\ on\ i$; (D)
11 $P = P \cup Distribute(xy,\ V)$; (B)
12 $Pyramid = CYK(G,\ w)$;
13 **if** *stopping criteria met* (C) **then**
14 **return** P;
15 **end**
16 **end**
17 **end**
18 **return** P;

Line 2: 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.
Note Line 10: 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).

thresholdi is regarding a line. Threshold, Linear or log function $f(i)$?

$P = P \cup \{distribute\ vc \in rowSet\ over\ V\}$; (B) vs.
$P = P \cup \{distribute\ rowSet \subseteq V^2 over\ V\}$; (B)

(AB,3) and (AB,1) -> (AB,1)
break


zu tun
in WM2
sub ⊆ V×V×ℕ
new ⊆ V×V
priority ⊆ V×V×ℕ
{(A,B,i) | (A,B) ∈ new, ∄j. (A,B,j) ∈ sub}
∪ {(A,B,j) | (A,B) ∈ new, (A,B,j) ∈ sub}
sub = sub ∪ new

3.2.4 Algorithm: SplitThenFill (Idea 1)

Algorithm 4: SplitThenFillPrep

Input: Word $w \in \Sigma^*$

Output: Set of productions P

1 $P = \emptyset$; // $P \subseteq V \times (V^2 \cup \Sigma)$

2 $P = Distribute(\Sigma,\ V)$; (A)

3 $Sol = \emptyset$; // $Sol \subseteq \{(P_{Sol},\ Cell_{i,j}) \mid P_{Sol} \subseteq P \wedge Cell_{i,j} \in Pyramid\}$

4 $Sol = SplitThenFill(P,\ w,\ i_{max},\ 0)$;

5 **return** P_{Sol};

Line 2: Fills the i=0 row of the pyramid.
Line **??**: $Cell_{i,j} \subseteq V \wedge Cell_{i,j} \in Pyramid$. The pyramid represents the upper part of the upper triangular matrix of the CYK. Reflection at the diagonal of the matrix and rotation of -45 degrees.
Line **??**: Starting recursively from the tip of the pyramid.

Algorithm 5: SplitThenFill

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

Output: $(P,\ Cell_{i,j})$

1 **if** $i = 0$ **then**

2 | **return** $(P,\ Cell_{i,j})$;

3 **end**

4 **if** *stopping criteria met* (C) **then**

5 | **return** $(P,\ Cell_{i,j})$;

6 **end**

7 *choose one m uniform randomly in* $[j+1,\ j+i]$;

8 $(P,\ Cell_l) = SplitThenFill(P,\ w,\ (m-j-1),\ j)$;

9 $(P,\ Cell_r) = SplitThenFill(P,\ w,\ (j+i-m),\ m)$;

10 $Pyramid = CYK(G,\ w)$;

11 **if** $Cell_{i,j} = \emptyset$ **then**

12 | $SetVc = uniform\ random\ subset\ from\ \{vc \mid v \in Cell_l\ \wedge\ c \in Cell_r\}$;

13 | $P = P \cup Distribute(SetVc,\ V)$; (B)

14 **end**

15 **return** $(P,\ Cell_{i,j})$;

Line 2: Recursion anchor that returns the up to this point modified productions P and the variables in the cell with index i and j.
Line 8 + 15: Analogous to the CYK-algorithm a cell combination $Cell_l$ and $Cell_r$ is chosen that can generate the sub string.
Line 5: A Recalculation is done at each recursive call because only the updated production set P is returned recursively.

The stopping criteria would be, that each marked $cell_{i,j} \neq \emptyset$ and it must be possible

to get from $cell_{m,j}$ and $cell_{i-m,m+j+1}$ to $cell_{i,j}$ through applying one of the production rules.
Algorithm **??** Idea1 uniform randomly generates a predefined structure of the derivation tree. You always update the pyramid after adding one production to the grammar. Now there are two options to fill the parse table:

1. Bottom Up: The parse table is filled relatively evenly. All information regarding the upper cells are available and can be used. Similar to the CYK Algorithm approach.

2. Top Down: The parse table is filled quiet unevenly. You don't have all information available. Think about adding a production for the node cell: You can add a production so that its producing cells fill the node cell, but you don't know what actually would be the best to fill in these producing cells because they themselves aren't looked at yet. This problem is kept until the last depth of the recursion, where the cells in row $i = 0$ are taken into account. Only starting there you know what variables actually produce the terminals.
   Maybe solution: For the Top Down approach, don't assume that the terminals are already distributed over the V. Distribute the terminals over the variables in an ideal way that fits your already generated productions best.

3.2.5 Algorithm: Idea 2, How often cells are used for subset calculations

3.2.6 Algorithm: SplitAndFill

Algorithm 6: SplitAndFillPrep

Input: Word $w \in \Sigma^*$

Output: Set of productions P

1 $P = \emptyset$; // $P \subseteq V \times (V^2 \cup \Sigma)$

2 $Pyr = EmptyPyramid$;

3 $Cell_{i_{max},0} = Cell_{i_{max},0} \cup \{S\}$; // $Cell_{i,j} \subseteq V \wedge Cell_{i,j} \in Pyramid$

4 ;

5 $Sol = \emptyset$; // $Sol \subseteq \{(P_{Sol},\ Pyramid) \mid P_{Sol} \subseteq P\}$

6 $Sol = SplitThenFill(Pyr,\ P,\ w,\ i_{max},\ 0)$;

7 $P = P \cup P_{Sol}$;

8 **return** P;

Line 2: $EmptyPyramid \Leftrightarrow \forall i\ \forall j\ Cell_{i,j} = \emptyset$

Line 5: $Cell_{i,j} \subseteq V \wedge Cell_{i,j} \in Pyramid$. The pyramid represents the upper part of the upper triangular matrix of the CYK. Reflection at the diagonal of the matrix and rotation of -45 degrees.

Algorithm 7: SplitAndFill

Input: $Pyramid\ Pyr,\ P \subseteq V \times (V^2 \cup \Sigma),\ w \in \Sigma^*,\ i, j \in \mathbb{N}$

Output: $(P,\ Pyr)$

if *stopping criteria met* (C) **then**
  return $(P,\ Pyr)$;
end
$SetAll = V \times V$;
$Pyramid = CYK(G,\ w)$;
choose one m *uniform randomly in* $[j+1,\ j+i]$;
if $(m - j - i) \neq 0$ **then**
  $(P,\ Pyr) = SplitAndFill(Pyr,\ w,\ (m - j - 1),\ j)$;
end
else then
  $v =$ *choose one variable uniform randomly from* $Cell_{i,j}$;
  $P = P \cup (v - - > w_j)$;
end
if $(j + i - m) \neq 0$ **then**
  $(P,\ Pyr) = SplitAndFill(Pyr,\ w,\ (j + i - m),\ m)$;
end
else then
  $v =$ *choose one variable uniform randomly from* $Cell_{i,j}$;
  $P = P \cup (v - - > w_j)$;
end **return** $(P,\ Cell_{i,j})$;

3.2.7 Tweaking Sub Procedures in more detail

Maybe don't keep this so that the Algorithms can be read without flipping pages.
Ⓐ, Ⓑ, ...

Algorithm 8: Distribute

Input: $Rhse \subseteq (V^2 \cup \Sigma),\ V$

Output: Set of productions P

$i \in \mathbb{N},\ j \in \mathbb{N}$;
foreach $rhse \in Rhse$ **do**
  choose n uniform randomly in $[i, j]$;
  $V_{add} := uniform\ random\ subset\ of\ size\ n\ from\ V$;
  $P = P \cup \{(v, rhse) \mid v \in V_{add},\ rhse \in Rhse\}$;
end
return P;

Algorithm 8 isn't needed anymore for the descriptions of the basic idea of the algorithm. It will be a module later on while tweaking the algorithms.

Algorithm 9: CalculateSubsetForCell

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

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

$V_{i,j} = \emptyset$;
for $k := i - 1 \to 0$ **do**
  $V_{i,j} = V_{i,j} \cup \{X \mid X \longrightarrow YZ,\ Y \in V_{k,j},\ Z \in V_{i-k-1,k+j+1}\}$;
end
return $V_{i,j}$;

Algorithm **??** describes the magic of the CKY-algorithm. It shows what cells are taken into account while filling one cell of the parse table.

3.3 Criteria Checking Procedures

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

Algorithm 10: checkForceCombinationPerCell

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 $cell_{i-1,j+1} = cellUpperRight$

Algorithm 10 is a check that needs to be explained.

Algorithm 11: checksumOfProductions

Input: $max \in \mathbb{N}_{\geq 0}$
Output: $true \iff sum \leq max$

```
1  if |P| > max then
2  |   return fales;
3  end
4  return true;
```

Algorithm 11 can be explained via the Output of the algorithm alone.

Algorithm 12: checkMaxNumberOfVarsPerCell

Input: $max \in \mathbb{N}_{\geq 0}$

Output: $true \iff \forall cell_{i,j} \in pyramid,\ |cell_{i,j}| \leq max$

```
1 for i := 1 to i_max do
2     for j := 0 to j_max − i do
3         if |cell_{i,j}| > max then
4             return false;
5         end
6     end
7 end
8 return true;
```

Algorithm 12 can be explained via the Output of the algorithm alone.

Algorithm 13: checkMaxSumOfVarsInPyramid

Input: $max \in \mathbb{N}_{\geq 0}$

Output: $true \iff sum \leq max$

```
1  sum = 0;
2  for i := 1 to i_max do
3      for j := 0 to j_max − i do
4          sum = sum + |cell_{i,j}|;
5          if sum > max then
6              return false;
7          end
8      end
9  end
10 return true;
```

Algorithm 13 could possible be explained via a simple mathematical statement like the algorithms 11 and 12.

4 CLI Tool

Write much of this stuff in the appendix.

4.1 Short Requirements Specification

Use Cases i ⟶ "Lastenheft".
Input and Output parameter identification.
Here is described what the finished tool must and can do.

Generating the latex code and storing it in .tex-file. Then converting the .tex-file to .pdf-file via:
Runtime rt = Runtime.getRuntime();
Process pr = rt.exec("pdflatex mydoc.tex");
Process pr = rt.exec("pdflatex mydoc.tex");
Process pr = rt.exec("pdflatex mydoc.tex");
The triple invocation of LaTeX is to ensure that all references have been properly resolved and any page layout changes due to inserting the references have been accounted for. [http://www.arakhne.org/autolatex/]

4.1.1 Exam Exercises

An exam exercise consists out of a grammar, a word, a parsing table and a derivation tree. Creating a exam exercises must be possible. Therefore it is needed:

- Selection of a possible exam exercise out of high scoring samples ⟶ calculateSamples.jar [Input parameter: countOfNewSamples (better scoring samples in exchange for longer computation time)], which upon execution fills samples.txt with new high scoring samples, together with its actual scoring model parameters. Out of this samples one can be selected manually that is used for an exam exercise.

- Modifying of a exam exercise candidate: Changing the grammar and changing the word. [?changing the pyramid (I think no, because of the strong interconnection between the grammar and the parsing table it is already covered through being able to change the grammar)?] ⟶ calculateExamExercise.jar [Input parameter: examExercise.txt], that updates pre defined information for one sample upon execution.

- Predefined Information: It is a printable version of the finished exam exercise like grammar.png, parsingTable.png and derivationTable.png together with its latex code, that was used for its creation - modification later one possible. Also it is

examExerciseInfo.txt, that has the information about its actual scoring model parameters.

4.1.2 Fun With CNF's and CYK

Trying out stuff freestyle:

- Se
-

4.2 Overview - UML

UML-Diagramm showing the general idea of the implementation.
List noteworthy used libraries here, too.
Maybe some information out of the statistics tool of IntelliJ.

4.2.1 UML: More Detail 1

4.2.2 UML: More Detail 2

4.3 User Interaction

Here the specific must can do's are explained with short examples.

4.3.1 Use Case 1

4.3.2 Use Case i

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

Github
IntelliJ IDEA

Algorithms

This section contains all algorithms referenced in this thesis.

References

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