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: August 7, 2017

1. Supervisor: Prof. Dr. Wim Martens
2. Supervisor: M.Sc. Tina Trautner

ABC

Abstract

The abstract of this thesis will be found here.

Zusammenfassung

Hier steht die Zusammenfassung dieser Bachelorarbeit.

Contents

Abstract **5**

1 Introduction **6**
1.1 Motivation . . . 6
1.2 Context Free Grammar in Chromsky Normal Form . . . 6
1.3 General approaches . . . 7
1.3.1 Forward Problem & Backward Problem . . . 7
1.3.2 Parsing Bottom-Up & Top-Down . . . 7
1.4 Data Structure Pyramid . . . 8
1.5 Cocke-Younger-Kasami Algorithm . . . 9
1.6 Success Rates . . . 10

2 Algorithms **13**
2.1 Sub modules . . . 13
2.2 DiceRollOnlyCYK . . . 14
2.3 BottomUpDiceRollVar1 . . . 15
2.4 BottomUpDiceRollVar2 . . . 16
2.5 SplitThenFill . . . 17
2.6 SplitAndFill . . . 19
2.7 Comparision of Algorithms . . . 21

3 CLI Tool **22**
3.1 Scoring Model . . . 22
3.2 Exam Exercises . . . 22
3.3 Overview . . . 24

References **25**

1 Introduction

1.1 Motivation

The starting point of this thesis is to get a tool to automatically generate a suitable 4-tuple $exercise = (grammar,\ word,\ parse\ table,\ derivation\ tree)$, that is used to test if the students have understood the way of working of the CYK algorithm.
Various implementations and small online tools of the Cocke-Younger-Kasami (CYK) algorithm can be found [XXX]. Nevertheless it is required to automatically generate suitable *exercise*s, that afterwards can be modified as wanted. This is the reason an own implementation has been made. It is also a task to find a more clever algorithm to automatically generate *exercise*s with a high chance of being suitable as an exam exercise.

1.2 Context Free Grammar in Chromsky Normal Form

Definition 1. Context Free Grammar (CFG)
We define a CFG as the 4-tuple $G = (V,\ \Sigma,\ S,\ P)$:

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

It is valid that $\Sigma \cap V = \emptyset$.

Definition 2. CFG in Chromsky Normal Form (CNF)
A CFG $G = (V,\ \Sigma,\ S,\ P)$ is in CNF iff.:

- $P \subseteq V \times (V \cup \Sigma)^*$.

Throughout this thesis a grammar is always synonymous with Definition 2. For further convenience the following default values are always true:

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

A rule consists out of a left hand side element (lhse) and a right hand side element (rhse). Example: $lhse \longrightarrow rhse$ applied to $A \longrightarrow c$ and $B \longrightarrow AC$ means that A and B are a $lhse$ and c and AC are a $rhse$.

Definition 3. Word w and language $L(G)$
Word w and language $L(G)$:

- $w \in \Sigma^* = \{w_0,\ w_1,\ ...,\ w_j\}$.
- A language $L(G)$ over an alphabet Σ is a set of words over Σ .

Moreover in the context of 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, whereas a *Cell* is again 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 $w \in L(G)$.

Definition 4. Forward Problem ($G \xrightarrow{derivation} w$)
Input: Grammar G in CNF.
Output: Derivation d that shows implicitly $w \subseteq L$.

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 $L(G)$.

Definition 5. Backward Problem = Parsing ($w \overset{?}{\subseteq} L(G)$)
Input: w and a grammar G in CNF.
Output: $w \subseteq L(G) \Longrightarrow$ derivation d.

If you are given a word w and want to determine if it is element of $L(G)$, it is called Backward Problem or parsing.

1.3.2 Parsing Bottom-Up & Top-Down

There are again two ways to classify the approach of parsing.

Definition 6. Bottom-Up parsing
Bottom-Up parsing means to start parsing from the leaves up to the root node.

"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].

Definition 7. Top-Down parsing
Top-Down parsing means to start parsing from the node down to the leaves.

"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." [XXX] (The so called test sentence is synonymous to an word w.) Reasonably criteria to guide the choice of which rewrite rule to apply could include to begin the parsing at the first (left) or last (right) character of the word w [XXX][Duda 8.6.3 page 428]

1.4 Data Structure Pyramid

To be able to describe the way of working of the different algorithms easier the help data structure *Pyramid* will be defined – note that *Pyramid* starts with upper case and therefore is a set). But before that:

Definition 8. $[i, j]$
$[i,\ j] := \{i,\ i+1, ..., j-1,\ j\} \subseteq \mathbb{N}_{\geq 0}$.

Definition 9. $Cell_{i,j}$
$Cell_{i,j} \subseteq \{(V, k) \mid k \in \mathbb{N}\}$

Now *Pyramid* can be defined as following:

Definition 10. $Pyramid$
$Pyramid := \{Cell_{i,j} \mid i \in [0,\ i_{max}],\ j \in [0,\ j_{max,i}],\ i_{max} = |w|-1,\ j_{max,i} = i_{max}-i\}$.

The following is the visual representation of a *Pyramid* that additionally has written the word w above it:

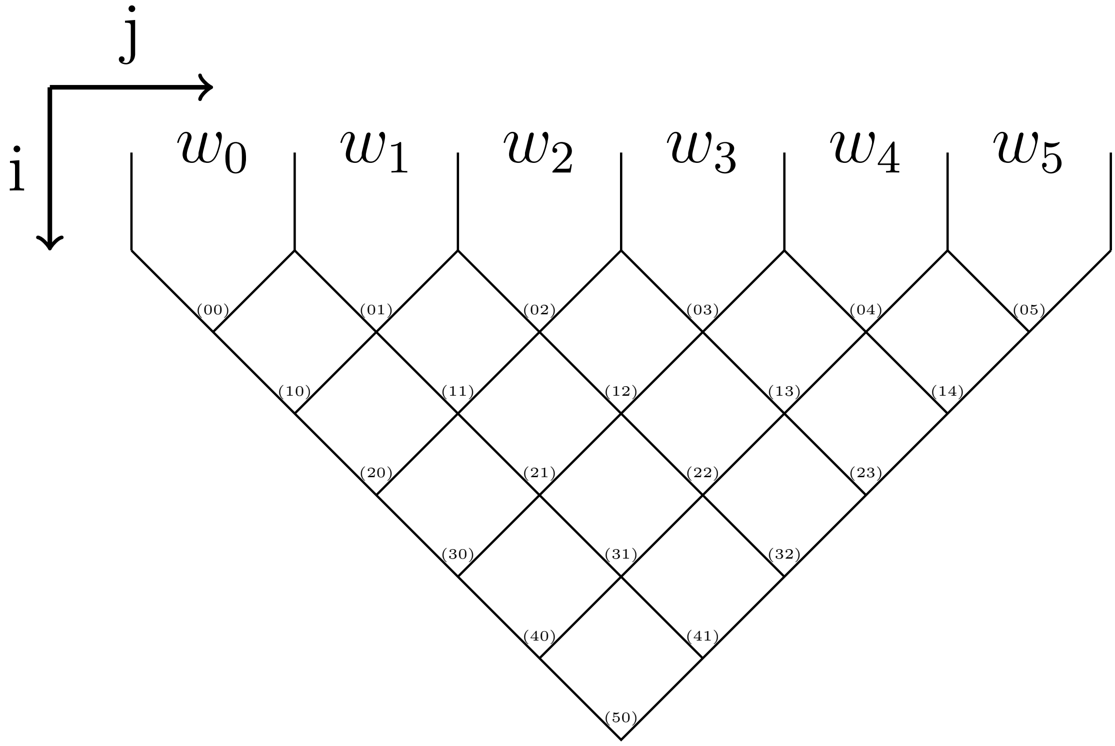


Figure 1: Visual representation of a *Pyramid* with the word w above it.

Definition 11. *CellDown, CellUpperLeft and CellUpperRight*
Let there be a $Cell_{i,j}$ then the following is true:

- $CellDown = Cell_{i,j}$.
- $CellUpperLeft = Cell_{i-1,j}$.
- $CellUpperRight = Cell_{i-1,j+1}$.

1.5 Cocke-Younger-Kasami Algorithm

The Cocke-Younger-Kasami Algorithm (CYK) has been developed independently in the 1960s by Itiroo Sakai, John Cocke, Tadao Kasami, Jacob Schwartz and Daniel Younger that uses the principle of dynamic programming. [wiki and the four sources] The description of the algorithm follows [TI Hofmann] adjusted to the help data structure *Pyramid*.

Algorithm 1: CYK

Input: Grammar $G = (V,\ \Sigma,\ S,\ P)$ and word $w \in \Sigma^* = \{w_0,\ w_1,\ ...,\ w_j\}$
Output: true $\Leftrightarrow w \in L(G)$

1 $Pyramid = \emptyset$;
2 **for** $j := 0 \rightarrow i_{max}$ **do**
3     $Pyramid \cup Cell_{0,j} = \{(X, j) \mid X \longrightarrow w_j\}$
4 **end**
5 **for** $i := 1 \rightarrow i_{max}$ **do**
6   **for** $j := 0 \rightarrow j_{max,i}$ **do**
7   **for** $k := i - 1 \rightarrow 0$ **do**
8             $Pyramid \cup Cell_{i,j} = \{X \mid X \longrightarrow YZ,\ Y \in Cell_{k,j},\ Z \in Cell_{i-k-1,k+j+1}\}$;
9   **end**
10   **end**
11 **end**
12 $wInL = false$;
13 **if** $(S, i) \in Cell_{i_{max},0}$ **then**
14     $wInL = true$;
15 **end**
16 **return** $wInL$;

Line 2: First row.
Line 5: All rows except the first.
Line 6: All cells in each row.
Line 7: All possible cell combinations for each cell.
Line 14: True iff $Cell_{i_{max},0}$ contains the start variable.

1.6 Success Rates

Success Rates (SR) are used to compare the algorithms accounting to their performance of the different requirements. $N \in \mathbb{N}$ is the count of all generated grammars of the examined algorithm.

Success Rate: An generated *exercise* contributes to the Success Rate (SR) iff it contributes to the SR-Producibility, to the SR-Cardinality-Rules and to the SR-Pyramid at the same time.
It holds: $SR = n/N$, whereas n is the count of *exercises* that fulfil the requirements in this case.

Success Rate Producibility: An generated *exercise* contributes to the SR-Producibility iff the CYK algorithm's output (Algorithm 1) is true.

It holds: SR-Producibility = p/N, whereas p is the count of *exercises* that fulfil the requirement.

Success Rate Cardinality-Rules An generated *exercise* contributes to the SR-Cardinality-Rules iff the grammar has got less than a certain amount of productions. It is true: SR-Cardinality-Rules = cr/N, whereas cr is the count of *exercises* that fulfil this requirement.

Success Rate Pyramid An generated *exercise* contributes to the SR-Pyramid iff the following conditions are met:

1. At least one cell forces a right cell combination.
2. There are less than a certain amount of variables in the entire pyramid, per default 100.
3. There are less than a certain amount of variables in each cell of the pyramid, per default 3.

It holds: SR-Pyramid = p/N, whereas p is the count of *exercises* that fulfil the requirements above.
While checking 1., 2. and 3. a simplification of $Cell_{i,j}$ is done:
$Cell_{i,j} \subseteq \{(V,k) \mid k \in \mathbb{N}\} \longrightarrow Cell_{i,j} \subseteq V$ See the following example:

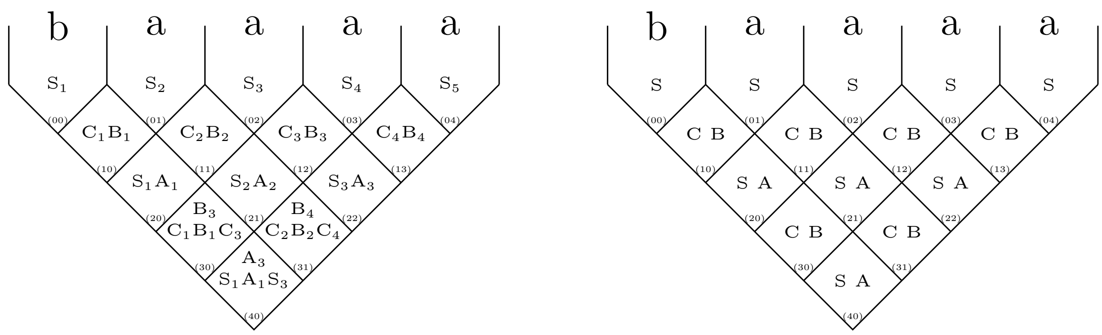


Figure 2: The simplification of cells in a pyramid in more detail.

For more detail to how a cell forces a right cell combination (1.) see the following algorithm. Note that a right cell combination can only be forced of cells with index $i > 1$.

Algorithm 2: checkForceCombinationPerCell

Input: $CellDown,\ CellUpperLeft,\ CellUpperRight \subseteq V,\ P \subseteq V \times (V^2 \cup \Sigma)$

Output: $true \iff at\ least\ one\ variable \in CellDown\ forces$

1 $VarsForcing = \emptyset$; // $VarsForcing \subseteq V$

2 $VarComp = \{xy \mid x \in CellUpperLeft\ \wedge\ y \in CellUpperRight\}$;

3 **foreach** $v \in CellDown$ **do**

4     $Prods = \{p \mid p \in P\ \wedge\ p = (v_1, rhse_1)\ \wedge\ v_1 = v\}$;

5     $Rhses = \{rhse \mid p \in Prods\ \wedge\ p = (v_1, rhse_1)\ \wedge\ rhse_1 = rhse\}$;

6   **if** $Rhses \nsubseteq VarComp$ **then**

7         $VarsForcing = VarsForcing \cup v$;

8   **end**

9 **end**

10 **return** $|VarsForcing| > 0$;

Line 4: Get all rules of P that have v on their left side.
Line 5: Get the rhse of each element of $Prods$.
Line 6: If no $rhse \in Rhse$ can be found in VarComp, then this variables forces, concluding that this cell as a hole forces.

As seen in Figure 3 the variables in $Cell_{2,0}$ and in $Cell_{2,1}$ force each a right cell combination. In both cases $VarComp = \{SS\}$. The variable $v = C$ doesn't have SS as one of its rhses. Therefore the variable C forces. $Cell_{3,0}$ doesn't force because $VarComp = \{CC\}$ and the variable $v = S$ has CC as its rhse. Remember that cells with index $i \leq 1$ can't force at all.

$Grammar:$
$C \rightarrow CS \mid a \mid b$
$S \rightarrow CC$

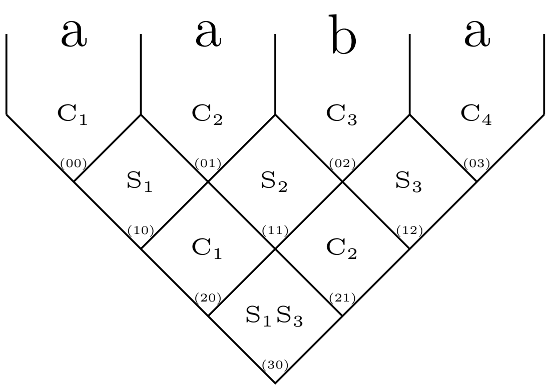


Figure 3: Example grammar and pyramid for the application of Algorithm 2.

2 Algorithms

2.1 Sub modules

Sub modules are parts of the algorithms that are denoted circled like A, B, C and D. They are procedures that should to be explained in more detail a little bit for a better understanding of the way of working of the algorithms.

Distribute$(\Sigma,\ V)$ **A and Distribute**$(V^2,\ V)$ **B:**

The difference between A and B is that one time Σ and the other time V^2 are distributed. The specifics of how they are distributed are the same in both cases as described in the following algorithm:

Algorithm 3: Distribute

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

Output: Set of productions P

foreach $rhse \in Rhse$ **do**

  $choose\ n\ uniform\ randomly\ in\ [i, j];$ // $i \in \mathbb{N},\ \ j \in \mathbb{N}$

  $V_{add} := uniform\ random\ subset\ of\ size\ n\ from\ V;$

  $P \cup \{(v, rhse) \mid v \in V_{add},\ rhse \in Rhse\};$

end

return P;

Stopping Criteria C:

Two kinds of C have been used. One is that it is true iff more than half of the pyramid cells are not empty and the other one is that there is at least one variable in the tip of the pyramid. It is to be taken in consideration that the latter is somewhat dependent on the count possible variables as seen in [XXX].

ChooseXYDependingOnIFromRowSet D:

$RowSet \subseteq \{(XY, i) \mid X, Y \in V \wedge i \in \mathbb{N}\}$

Compression of the RowSet like: (AB,3) and (AB,1) -> (AB,1) –> RowSetCompressed rowListWeighted = add i times XY to rowListWeighted. XXX

2.2 DiceRollOnlyCYK

This is a naive way of generating grammars, which will be the lower boundary while comparing the algorithms. Each future algorithm should have a higher score than this algorithm or otherwise it would be worse, than simple dice rolling the distribution of terminals (A) and compound variables (B).

Algorithm 4: DiceRollOnlyCYK

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

Output: Set of productions P

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

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

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

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

return P;

2.3 BottomUpDiceRollVar1

This algorithm uses the Bottom-Up approach (Chapter 1.3) whereby the parsing table is filled starting from the leaves in direction to the root node.

Algorithm 5: BottomUpDiceRollVar1

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 $Pyramid = CYK(G,\ w)$;
4 **for** $i := 1$ ***to*** i_{max} **do**
5 $\quad J = \{0,\ ...\ ,\ j_{max} - 1\}$; // $J \subseteq \mathbb{N}$
6 $\quad CellSet = \emptyset$; // $CellSet \subseteq V^2$
7 $\quad$ **while** $|J| > 0$ **do**
8 $\quad\quad$ *choose one* $j \in J$ *uniform randomly*;
9 $\quad\quad J = J \setminus \{j\}$;
10 $\quad\quad CellSet = CalculateSubsetForCell(Pyramid,\ i,\ j)$;
11 $\quad\quad P \cup Distribute(CellSet,\ V)$; B
12 $\quad\quad Pyramid = CYK(G,\ w)$;
13 $\quad\quad$ **if** *stopping criteria met* C **then**
14 $\quad\quad\quad$ **return** P;
15 $\quad\quad$ **end**
16 $\quad$ **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.

While taking a short look at the SRs in Table 2.7 it is seen that the SR-Pyramid is a little bit lower compared to Algorithm 4. Further investigation of the logs shows that a relatively small number of rules is already sufficient to invoke the stopping criteria C.

A good chosen stopping criteria allows a higher success rate.

2.4 BottomUpDiceRollVar2

As seen in algorithm 5 a small number of productions is sufficient to make the parsing table quite full. If an cell is nearer to the leaves its chance to be in the set of one of the calculated sub sets for a cell is higher. Therefore you could introduce a bias that favours cells with an higher index i to allow different cell combinations.

Berechnung von RowSet für alles Restlichen Zellen in der Zeile!!!!!?????????!!!!!!! Bisher nur für alle bisher Verwendeten.

Algorithm 6: BottomUpDiceRollVar2

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         $RowSet \cup \{(XY, i) \mid XY \in CalculateSubsetForCell(Pyramid,\ i,\ j)\}$;
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         $P \cup Distribute(xy,\ V)$; 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 here yet.
Note Line 9: threshold is reached iff more than half of the cells of one row aren't empty.

2.5 SplitThenFill

The basic idea for this algorithm is to uniform randomly generate a predefined structure of the derivation tree that helps adding the "right" productions. You always update the pyramid after adding one production to the grammar. This is also some kind of BottumUp approach - 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.
It is important to distribute the varComp exactly to one var.

Algorithm 7: 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 $Sol = (P_{Sol},\ Cell_{i,j})$; // $P_{Sol} \subseteq P \ \wedge \ Cell_{i,j} \in Pyramid$

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

5 **return** P_{Sol};

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

Algorithm 8: SplitThenFill

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

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

$P = P_{in}$;
if $i = 0$ **then**
    return $(P,\ Cell_{i,j})$;
end
choose one m uniform randomly in $[j+1,\ j+i]$;
$(P,\ Cell_l) = \mathit{SplitThenFill}(P,\ w,\ (m-j-1),\ j)$;
$(P,\ Cell_r) = \mathit{SplitThenFill}(P,\ w,\ (j+i-m),\ m)$;
$Pyramid = CYK(G,\ w)$;
if *stopping criteria met* C **then**
    return $(P,\ Cell_{i,j})$;
end
if $Cell_{i,j} = \emptyset$ **then**
    $Vc = uniform\ random\ subset\ from\ \{vc \mid v \in Cell_l\ \wedge\ c \in Cell_r\}\ with\ |Vc| \geq 1$;
    $P \cup \mathit{Distribute}(Vc,\ V)$; B
end
return $(P,\ Cell_{i,j})$;

The stopping criteria is met if the tip of the pyramid is not empty. It is a valid approach because if this cell is not empty it means that there is a chance of being able to generate the word. To add further productions only results in a grammar that has to many productions with its pyramid having to many variables.

2.6 SplitAndFill

It is dependent on the length of the word.
It is important that the terminals and the varcomps are distributed to exactly one var. The stopping criteria will be that each cell with index i = 0 must be not empty. Now there is a second option to fill the parse table:

1. 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.

The problem is that we have as much productions as splits in the derivation tree exist. The productions count can be reduced via merging duplicate productions and via reducing the split count in the tree.
Merging productions means: If there are A –> BS and C –> BS then only one Production of these two can remain.

Algorithm 9: SplitAndFillPrep

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

Output: Set of productions P

$P = \emptyset$; // $P \subseteq V \times (V^2 \cup \Sigma)$
$Sol = (P_{Sol},\ v)$; // $P_{Sol} \subseteq P$
$Sol = SplitAndFill(P,\ w,\ i_{max},\ 0)$;
$Merge\ productions\ with\ the\ same\ variableCompound\ in\ P_{Sol}$;
return P_{Sol};

Algorithm 10: SplitAndFill

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

Output: $(P,\ v)$

$P = P_{in}$;
if $i = 0$ **then**
  return $(P \cup (v,\ w_j),\ v_{lhse})$;
end
$choose\ one\ m\ uniform\ randomly\ in\ [j+1,\ j+i]$;
$(P,\ v_l) = SplitAndFill(P,\ w,\ (m-j-1),\ j)$;
$(P,\ v_r) = SplitAndFill(P,\ w,\ (j+i-m),\ m)$;
if $i = i_{max}$ **then**
  return $(P \cup (S,\ v_l v_r),\ v)$;
end
return $(P \cup (v,\ v_l v_r),\ v)$;

Table 3: My caption

				Pyramid			
Algorithm	SR	Producibility	Cardinality-Rules		Force-Right	Vars-PerCell	VarsIn-Pyramid
DiceRollOnly	09%			67%			
BottomUpVar1							
BottomUpVar2							
SplitThenFill							
SplitAndFill							

2.7 Comparision of Algorithms

Write about the standard configuration used.

Algorithm	SR	SR-Producibility	SR-Cardinality-Rules	SR-Pyramid
DiceRollOnly	09 %	24 %	88 %	39 %
BotomUpVar1	16 %	52 %	90 %	41 %
BotomUpVar2	19 %	47 %	93 %	53 %
SplitThenFill	24 %	40 %	97 %	67 %
SplitAndFill	11 %	100 %	69 %	15 %

Table 1: Comparison of the SRs of the algorithms. Stopping criteria root not empty.

Finding of ideal parameter for each algorithm.

Algorithm	SR	SR-Producibility	SR-Cardinality-Rules	SR-Pyramid
DiceRollOnly	09 %	23 %	88 %	38 %
BotomUpVar1	11 %	30 %	99 %	58 %
BotomUpVar2	13 %	26 %	99 %	66 %
SplitThenFill	24 %	40 %	97 %	67 %
SplitAndFill	11 %	100 %	70 %	14 %

Table 2: Comparison of the SRs of the algorithms. Stopping criteria more than half.

3 CLI Tool

Write much of this stuff in the appendix.

3.1 Scoring Model

Only valid ResultSamples are given a score. Parameters to be scored:

- RightCellCombinationsForcedCount
- maxSumOfVarsInPyramidCount
- maxNumberOfVarsPerCellCount
- maxSumOfProductionsCount

Maybe add a diversity criterion = homogeneity of the cells to the scoring matrix.

Parameter	Points					
	2	4	6	8	10	-100
cellCombinationsForced	[0,10]	[11,20]	[21,30]	[41,50]	[31,40]	>50
sumVarsInPyramid	[0,10]	[11,20]	[21,30]	[41,50]	[31,40]	>50
maxVarsPerCell	[5,5]	[4,4]	[1,1]	[3,3]	[2,2]	>5
sumProductions	[1,2]	[3,4]	[5,6]	[9,10]	[7,8]	>10
homogeneity	[ ]	[ ]	[ ]	[ ]	[ ]	>10
maxVarKsPerCell	[ ]	[ ]	[ ]	[ ]	[ ]	>10

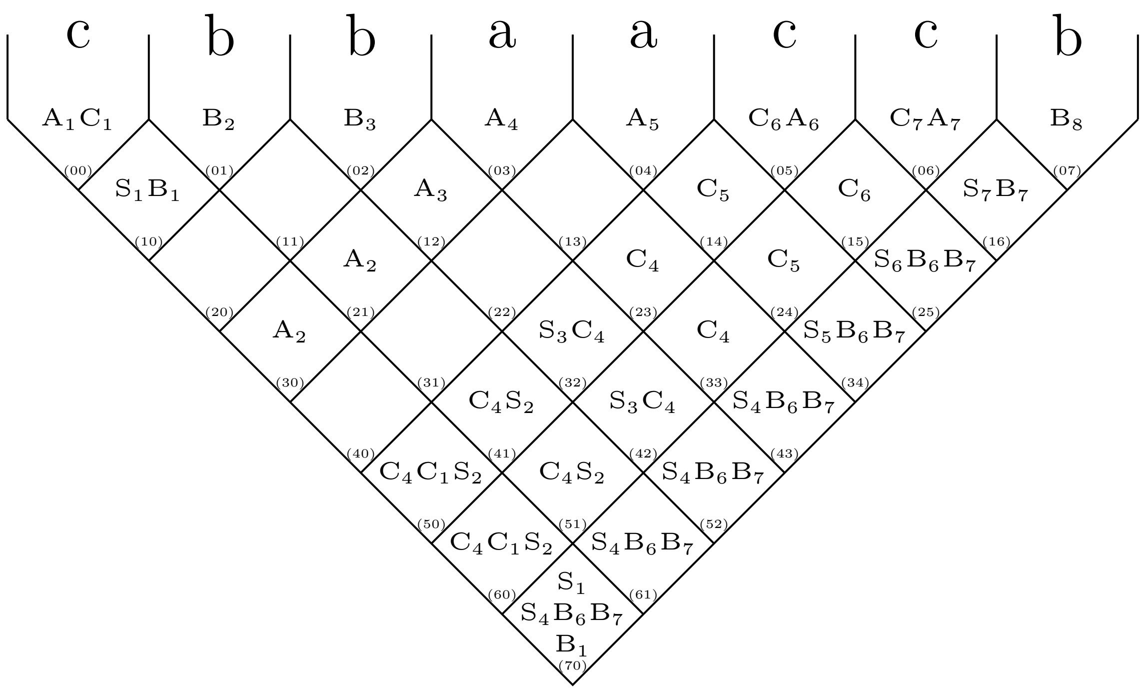
Table 4: Scoring of the different parameter values

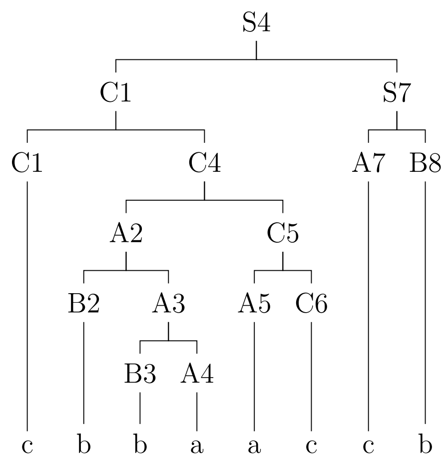
Based on table 4 each result sample is scored. Out of the #??? best result samples one can choose.
The result will be normalized to the maximum possible points -> range 0.0 to 1.0.

3.2 Exam Exercises

4-tuples $exercise = (grammar,\ word,\ parse\ table,\ derivation\ tree)$ that needs to be printed.

$$
\begin{aligned}
A &\to a \mid c \mid BA \\
B &\to b \mid CB \\
C &\to c \mid AC \\
S &\to AB \mid BC
\end{aligned}
$$





Does the output $P \subseteq V \times (V^2 \cup \Sigma)$ imply that G is in oCNF? 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.

3.3 Overview

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.
Show the stuff about Antler.

References

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