



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](https://www.github.com/AndreasBraun5)>  
Matrikel-Nr. 1200197

**Version date:** August 10, 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

<b>Abstract</b>	<b>5</b>
<b>1 Introduction</b>	<b>6</b>
1.1 Motivation . . . . .	6
1.2 Context Free Grammar . . . . .	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
<b>2 Algorithms</b>	<b>13</b>
2.1 Sub modules . . . . .	13
2.2 DiceRollOnlyCYK . . . . .	15
2.3 BottomUpDiceRollVar1 . . . . .	15
2.4 BottomUpDiceRollVar2 . . . . .	18
2.5 SplitThenFill . . . . .	20
2.6 SplitAndFill . . . . .	24
2.7 Comparision of Algorithms . . . . .	25
<b>3 GUI Tool: CYK Instances Generator</b>	<b>26</b>
3.1 Overview GUI . . . . .	26
3.2 Exam Exercises . . . . .	28
3.3 Scoring Model . . . . .	29
3.4 Parsing input with ANTLR . . . . .	30
3.5 Other matters . . . . .	30
3.5.1 Working with the program . . . . .	30
<b>References</b>	<b>31</b>

# 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 *exercises*, 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 *exercises* with a high chance of being suitable as an exam exercise.

## 1.2 Context Free Grammar

### 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.
- $\Sigma$  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 with restrictions

A CFG  $G = (V, \Sigma, S, P)$  is in CNF iff.:

- $P \subseteq V \times (V^2 \cup \Sigma)^*$ .

Throughout this thesis a grammar is always synonymous with Definition 2.  $P$  is not necessarily in CNF because it is possible that there are unreachable variables – from the starting variable. For further convenience the following default values are always true:

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

A rule consists out of a left hand side element (lhse) and a right hand side element (rhse). Example:  $lhse \rightarrow rhse$  applied to  $A \rightarrow c$  and  $B \rightarrow AC$  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  $L(G)$ :

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

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 ( $G \xrightarrow{\text{derivation}} w$ )**

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 ( $w \stackrel{?}{\subseteq} L(G)$ )**

Input:  $w$  and a grammar  $G$  in CNF.

Output:  $w \subseteq L(G) \implies$  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.  $[i, j]$** 

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

**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:



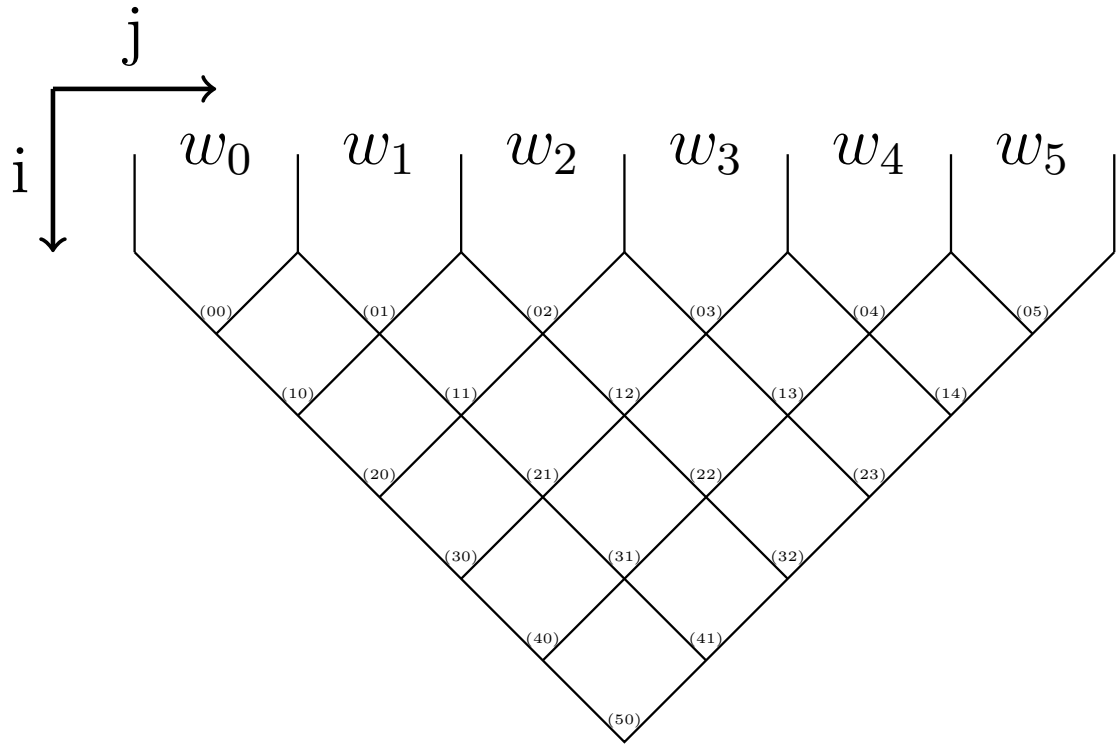


Figure 1: Visual representation of a *Pyramid* with the word  $w$  above it.

**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  $w \in \Sigma^* = \{w_0, w_1, \dots, w_j\}$ **Output:**  $\text{true} \Leftrightarrow w \in L(G)$ 

```

1  $Pyramid = \emptyset;$ 
2 for  $j := 0 \rightarrow i_{max}$  do
3    $Pyramid \cup Cell_{0,j} = \{(X, j) \mid X \rightarrow w_j\}$ 
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        $Pyramid \cup Cell_{i,j} = \{X \mid X \rightarrow YZ, Y \in Cell_{k,j}, Z \in$ 
9          $Cell_{i-k-1,k+j+1}\};$ 
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 2: 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:  $\text{SR-Producibility} = p/N$ , 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:  $\text{SR-Cardinality-Rules} = cr/N$ , 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:  $\text{SR-Pyramid} = p/N$ , whereas  $p$  is the count of *exercises* that fulfil the requirements above.

While checking 1., 2. and 3. a simplification of  $\text{Cell}_{i,j}$  is done:

$\text{Cell}_{i,j} \subseteq \{(V, k) \mid k \in \mathbb{N}\} \longrightarrow \text{Cell}_{i,j} \subseteq V$  See the following example:

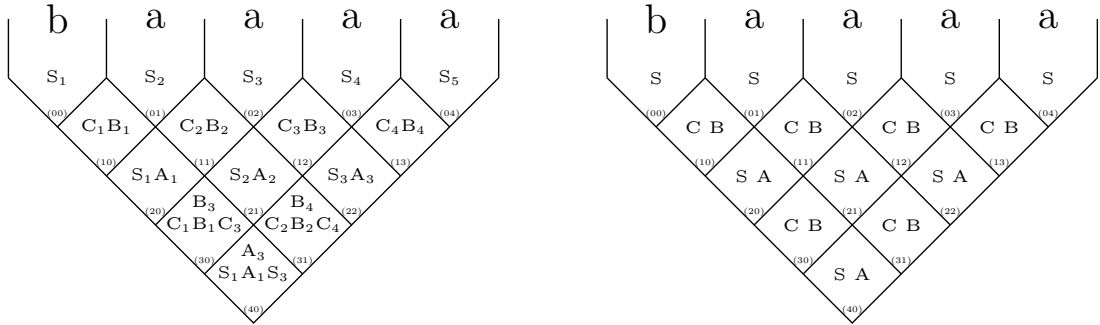


Figure 2: The simplification of cells in a pyramid.

For more detail to how a cell forces a right cell combination (1.) see the following algorithm. But note that a right cell combination can only be forced of cells with index  $i > 1$ . This restriction while calling Algorithm 2 is only thoughtful in the context of a exercise. Students easily find a pattern of how to fill the first two rows of the *pyramid* but do more mistakes beginning at row  $i \geq 2$ . The approach of finding only patterns and not thoroughly understanding the algorithm is countered that way.

**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    $Rhse = \{rhse \mid p \in Prods \wedge p = (v_1, rhse_1) \wedge rhse_1 = rhse\}$ ;
6   if  $Rhse \not\subseteq VarComp$  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 and 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 :

$$C \rightarrow CS \mid a \mid b$$

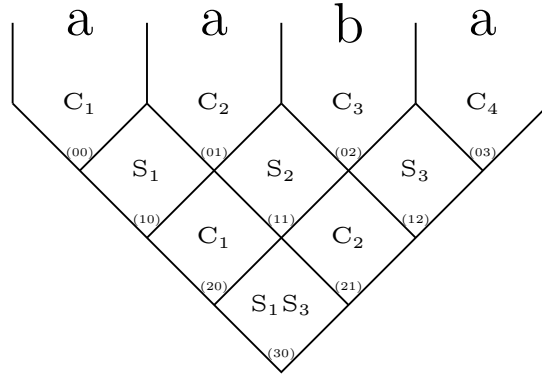
$$S \rightarrow CC$$


Figure 3: Application of Algorithm 2 onto a entire pyramid.

## 2 Algorithms

### 2.1 Sub modules

Sub modules are parts of the algorithms that are denoted circled like (A), (B), (C), (D) and (E). 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 following algorithms.

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

The difference between (A) and (B) is that one time  $\Sigma$  and the other time  $V^2$  are distributed. While distributing the terminals there exists at least one rule for every terminal used in the word  $w$ . The specifics of how they are distributed are the same in both cases as described in the following algorithm:

<b>Algorithm 3:</b> Distribute	
<b>Input:</b> $Rhse \subseteq (V^2 \cup \Sigma), V$	
<b>Output:</b> Set of productions $P \subseteq V \times (V^2 \cup \Sigma)$	
1	<b>foreach</b> $rhse \in Rhse$ <b>do</b>
2	<i>choose <math>n</math> uniform randomly in <math>[i, j]</math>; // <math>i \in \mathbb{N}, j \in \mathbb{N}</math></i>
3	<i><math>V_{add} :=</math> uniform random subset of size <math>n</math> from <math>V</math>;</i>
4	<i><math>P \cup \{(v, rhse) \mid v \in V_{add}, rhse \in Rhse\}</math>;</i>
5	<b>end</b>
6	<b>return</b> $P$ ;

**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] (chapter problem space analysis).

**CalculateSubsetForCell(Pyramid, i, j) (D):**

This procedure works kind of analogous from Line 7 to Line 9 of the CYK algorithm. For one  $Cell_{i,j}$  every possible cell combination is looked at, i.e. if a rule like  $lhse \rightarrow cs$  with  $cs \in CellSet$  is added then automatically  $Cell_{i,j}$  won't be empty any more.

**Algorithm 4:** CalculateSubsetForCell**Input:**  $Pyramid$ ,  $i \in \mathbb{N}$ ,  $j \in \mathbb{N}$ **Output:**  $CellSet \subseteq V^2$ 

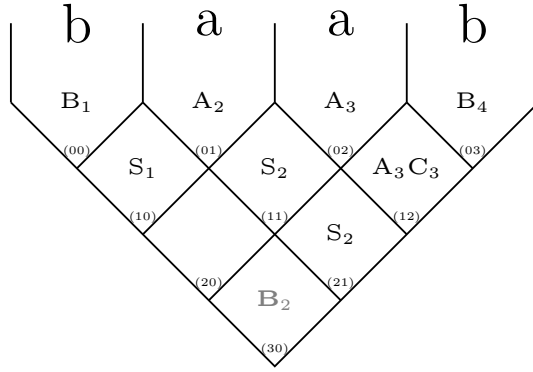
```

1  $CellSet = \emptyset$ ;
2 for  $k := i - 1 \rightarrow 0$  do
3    $CellSet \cup \{YZ \mid X \rightarrow YZ, Y \in Cell_{k,j}, Z \in Cell_{i-k-1,k+j+1}\}$ ;
4 end
5 return  $CellSet$ ;

```

In the following situation a rule should be added so that  $Cell_{3,0}$  won't be empty:

Grammar:  
 $A \rightarrow AB \mid a$   
 $B \rightarrow SC \mid b$   
 $C \rightarrow AB$   
 $S \rightarrow BA \mid AA$



Then the calculation of  $CellSet$  for  $Cell_{3,0}$  results in  $\{SA, SC, BS\}$ , whereas  $SA$  and  $SC$  stem from  $Cell_{1,0}$  together with  $Cell_{1,2}$  and  $BA$  is from  $Cell_{0,0}$  together with  $Cell_{2,1}$ . Now if either one of the rules  $lhse \rightarrow SA$ ,  $lhse \rightarrow SC$  or  $lhse \rightarrow BS$  is added to the grammar, then  $lhse \in Cell_{3,0}$ . Here the rule  $B \rightarrow SC$  has been added and now  $(B, 2)$  is element of  $Cell_{3,0}$ .

**Choose one  $xy$  with probability depending on row  $\textcircled{\text{E}}$  :**

There is the set  $RowSet \subseteq \{(XY, i) \mid X, Y \in V \wedge i \in \mathbb{N}\}$  and one  $xy$  is chosen out of it as following: Firstly the  $RowSet$  is compressed, i.e. every tuple with the same  $xy$  will be merged to its lowest  $i$ . See the example:  $RowSet = \{(AB, 3), (AB, 1), (AB, 5), \dots\}$  will become  $RowSet = \{(AB, 1), \dots\}$ . Afterwards all elements of  $RowSet$  will be placed in the  $RowMultiSet$  that can contain multiple equivalent elements. Now each element of  $RowMultiSet$  will be weighted according to their  $i$ . That means that elements like  $(AB, 1)$  will only occur one time though elements like  $(BC, 3)$  will occur three times and so on. Example:  $RowMultiSet = \{(AB, 1), (BC, 3), \dots\}$  becomes  $RowMultiSet = \{(AB, 1), (BC, 3), (BC, 3), (BC, 3), \dots\}$ . Now one element will be uniform randomly picked out of this weighted  $RowMultiSet$  like  $xy = BC$ .

```

RowSet = {(AB, 3), (AB, 1), (AB, 5), ...}           // compress
RowSet = {(AB, 1), ...}                             // place into RowMultiSet
RowMultiSet = {(AB, 1), (BC, 3), ...}               // weight elements
RowMultiSet = {(AB, 1), (BC, 3), (BC, 3), (BC, 3), ...} // pick element
xy = BC

```

Figure 4: Short example of the procedure E.

## 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 (Line 2) and compound variables (Line 3).

<b>Algorithm 5:</b> DiceRollOnlyCYK	
<b>Input:</b> Word $w \in \Sigma^*$	
<b>Output:</b> Set of productions $P$	
1	$P = \emptyset$ ; // $P \subseteq V \times (V^2 \cup \Sigma)$
2	$P = \text{Distribute}(\Sigma, V)$ ; (A)
3	$P \cup \text{Distribute}(V^2, V)$ ; (B)
4	<b>return</b> $P$ ;

A terminal  $\Sigma$  is distributed to at least one  $lhse$ , but a compound variable  $V^2$  must not be distributed at all. This means that for each terminal of  $\Sigma = \{a, b\}$  there exists at least one rule like  $lhse \rightarrow a$  and  $lhse \rightarrow b$  and for each possible compound variable  $V^2 = \{AA, AB, AC, AS, BB, BC, BS, CC, CS, SS\}$  it is possible that only a smaller subset like  $\{AA, BA, CC, SC\}$  is distributed so that only rules like  $lhse \rightarrow AA, lhse \rightarrow BA, lhse \rightarrow CC$  and  $lhse \rightarrow SC$  exist.

Grammar after Line 2:	Grammar after Line 3:
$C \rightarrow a$	$C \rightarrow BA \mid AA \mid a$
$B \rightarrow b$	$B \rightarrow b$
	$S \rightarrow CC \mid SC$

Figure 5: Short example of Algorithm 5.

## 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.

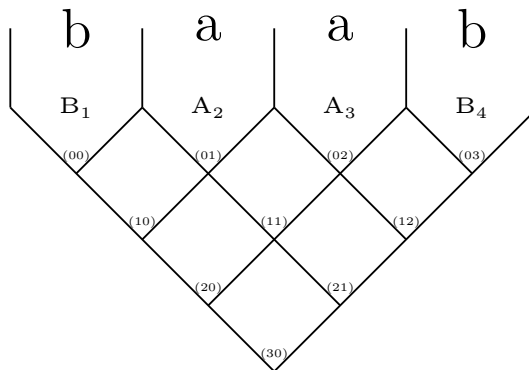
The basic idea behind this algorithm is to guide the choice of rules while distributing the compound variables  $V^2$ . In Algorithm 5 it can happen that the terminals are distributed to the variables  $A$  and  $B$  and Algorithm 5 completely discards this fact during

the distribution of the compound variables. In a situation as seen below it can happen

Grammar:

$A \rightarrow a$

$B \rightarrow b$



that rules like  $lhse \rightarrow CC$  or  $lhse \rightarrow SC$  are added, that obviously not directly help to fill the parsing table and bloat the grammar with useless rules. More reasonably rules to add would be  $lhse \rightarrow BA$ ,  $lhse \rightarrow AA$  and  $lhse \rightarrow AB$ .

Algorithm 6 takes this up: After distributing the terminals (Line 6) the updated parsing table (Line 12) is always taken into consideration while choosing (Line 10) variable compounds and to finally add them (Line 11). I.e. for each chosen cell a *CellSet* is calculated, that only contains reasonably variable compounds. Now only variable compounds are added that directly help to fill the parsing table.



**Algorithm 6:** BottomUpDiceRollVar1**Input:** Word  $w \in \Sigma^*$ **Output:** Set of productions  $P$ 

```

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

```

---

Line 4: Fills the  $i=0$  row of the pyramid.

Line 8: A cell is only visited only once.

## 2.4 BottomUpDiceRollVar2

While examining the Algorithm 6 via its log files [XXX] it can be seen that already a very small number of rules in the grammar is sufficient so that the stopping criteria  $\textcircled{C}$  is met – the cells that indirectly decide what rules to add are mostly from row one ( $i = 1$ ) and sometimes if at all from row two ( $i = 2$ ).

This again leads to another idea to introduce a row dependent  $threshold_i$  (Line 9) that helps that more cells with  $i \geq 2$  are chosen – what possibly can lead to more diverse (= homogeneity criteria, see Chapter 3.3) grammars. The diversity, in context of Algorithm 6, is somewhat too restricted to the *lhse*s that have one of the terminals as its *rhse*. Most of the rules that are part of the grammar will contain one these *lhse*s. This is due to the basic idea of Algorithm 6 but also due to the relatively small number of rules in the grammar.

Further diversification is achieved through the usage of  $\textcircled{E}$  (Line 10). Variable compounds that already have been used in a row with low index  $i$  are at a disadvantage to be picked again as described in Algorithm 4.

As seen in Figure 6 rules with the same *rhse* =  $BA$  and  $AA$  have been added to the variables  $B$  and  $A$  in Grammar1. For Grammar2 instead the rule  $B \rightarrow SS$  was added that contributes to a better diversity.

Grammar0:	Grammar1:	Grammar2:
$C \rightarrow BA \mid AA \mid a$	$C \rightarrow BA \mid AA \mid a$	$C \rightarrow BA \mid AA \mid a$
$B \rightarrow b$	$B \rightarrow BA \mid AA \mid b$	$B \rightarrow SS \mid b$
$S \rightarrow CC \mid SC$	$S \rightarrow BA \mid AA \mid CC \mid SC$	$S \rightarrow CC \mid SC$

Figure 6: Goal of better diversity: Starting point is Grammar0. Grammar2 is of better diversity than Grammar1.

**Algorithm 7:** 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      $RowSet \cup \{(XY, i) \mid XY \in CalculateSubsetForCell(Pyramid, i, j)\}$ ; (D)
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  $i$ ; (E)
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 4: Fills the  $i=0$  row of the pyramid.

## 2.5 SplitThenFill

The idea here is to first distribute the terminals (Line 2 of Algorithm 8) and then to uniform randomly generate a predefined structure of the derivation tree (Line 4 of Algorithm 2 and in general Algorithm 9) Top-Downwards and then again to fill the parsing table Bottom-Upwards accordingly to this derivation tree. The reason behind its name comes from the fact that only after completely generating the structure (splitting of the word in subwords) of the derivation tree then the rules are added to the grammar that allows further filling of the parsing table.

Every time before adding a new rule (Algorithm 9 Line 14) the already available information regarding the other rules is used to determine if a new rule is needed to fill this node of the derivation tree (Line 12 of Algorithm 9).

<b>Algorithm 8:</b> SplitThenFill	
<b>Input:</b> Word $w \in \Sigma^*$	
<b>Output:</b> Set of productions $P$	
$\mathbf{1} \ P = \emptyset; \ // \ P \subseteq V \times (V^2 \cup \Sigma)$ $\mathbf{2} \ P = \text{Distribute}(\Sigma, V); \textcircled{\text{A}}$ $\mathbf{3} \ Sol = (P_{Sol}, Cell_{i,j}); \ // \ P_{Sol} \subseteq P \ \wedge \ Cell_{i,j} \in Pyramid$ $\mathbf{4} \ Sol = \text{SplitThenFill}(P, w, i_{max}, 0);$ $\mathbf{5} \ \text{return } P_{Sol};$	
Line 2: Fills the $i=0$ row of the pyramid.	

For this algorithm it is important to mention that while using  $\textcircled{\text{B}}$  (Line 14 of Algorithm 9) a variable compound is added to at least one  $lhse$ . For every element of  $Vc$  (Line 13 of Algorithm 9) there exists at least one rule  $lhse \rightarrow vc$  with  $vc \in Vc$ .

The construction of the derivation tree for instance is done as following – each number represents the recursion depth of its subtree:

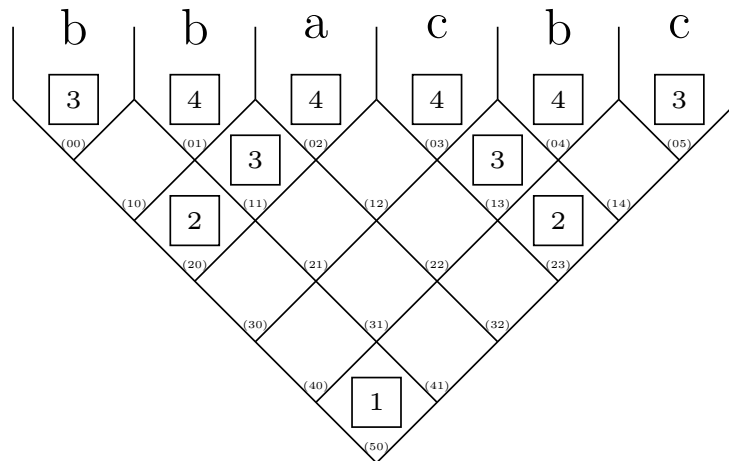


Figure 7: Example derivation tree structure.

<b>Algorithm 9: SplitThenFillRecursion</b>	
<b>Input:</b> $P_{in} \subseteq V \times (V^2 \cup \Sigma)$ , $w \in \Sigma^*$ , $i, j \in \mathbb{N}$	
<b>Output:</b> $(P, Cell_{i,j})$	
1	$P = P_{in};$
2	<b>if</b> $i = 0$ <b>then</b>
3	<b>return</b> $(P, Cell_{i,j});$
4	<b>end</b>
5	<i>choose one <math>m</math> uniform randomly in <math>[j + 1, j + i];</math></i>
6	$(P, Cell_l) = SplitThenFill(P, w, (m - j - 1), j);$
7	$(P, Cell_r) = SplitThenFill(P, w, (j + i - m), m);$
8	$Pyramid = CYK(G, w);$
9	<b>if</b> <i>stopping criteria met</i> $\textcircled{C}$ <b>then</b>
10	<b>return</b> $(P, Cell_{i,j});$
11	<b>end</b>
12	<b>if</b> $Cell_{i,j} = \emptyset$ <b>then</b>
13	$Vc = \text{uniform random subset from } \{vc \mid v \in Cell_l \wedge c \in$
	$Cell_r\}$ <i>with</i> $ Vc  \geq 1;$
14	$P \cup Distribute(Vc, V); \textcircled{B}$
15	<b>end</b>
16	<b>return</b> $(P, Cell_{i,j});$

The same example tree structure is used as seen in Figure 5.

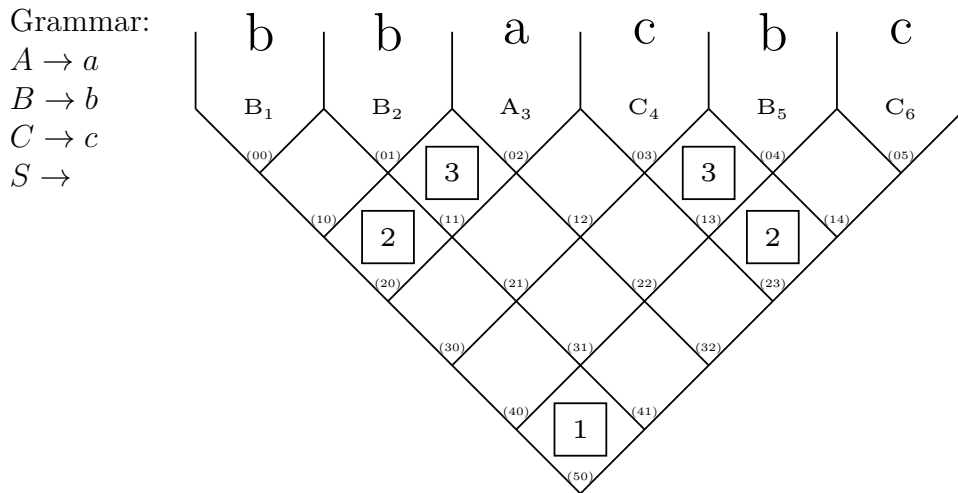


Figure 8: Illustration of Algorithm 8 part 1.

After adding the terminals to the grammar (Line 2 in Algorithm 8) now one must take on the recursion step at  $Cell_{1,1}$ . Now  $Cell_l = \{B_2\}$  and  $Cell_r = \{A_3\}$  and therefore  $Vc = BA$ . Now adding the rule  $S \rightarrow BA$  leads to the following changes:

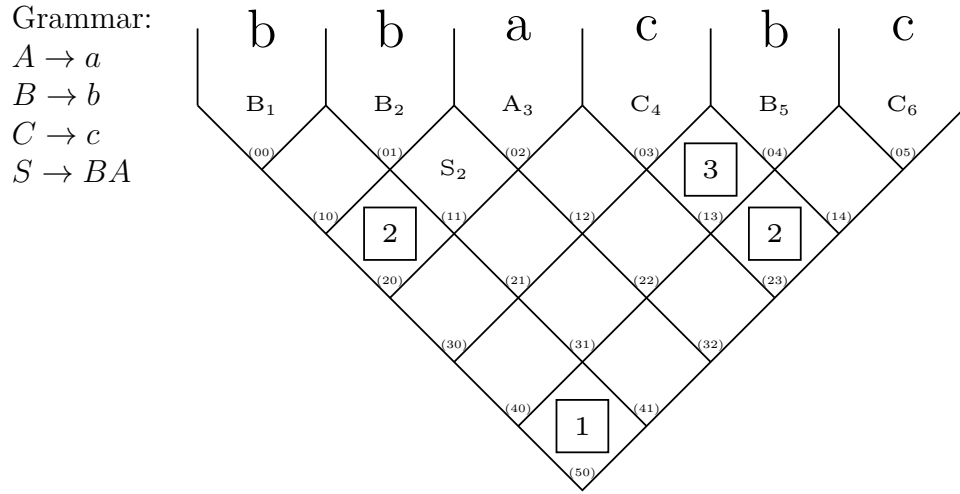


Figure 9: Illustration of Algorithm 8 part 2.

The next recursion step happens in  $Cell_{2,0}$ . Now  $Cell_l = \{B_1\}$  and  $Cell_r = \{S_2\}$ . Analogously the rule  $A \rightarrow BS$  is added to the grammar:

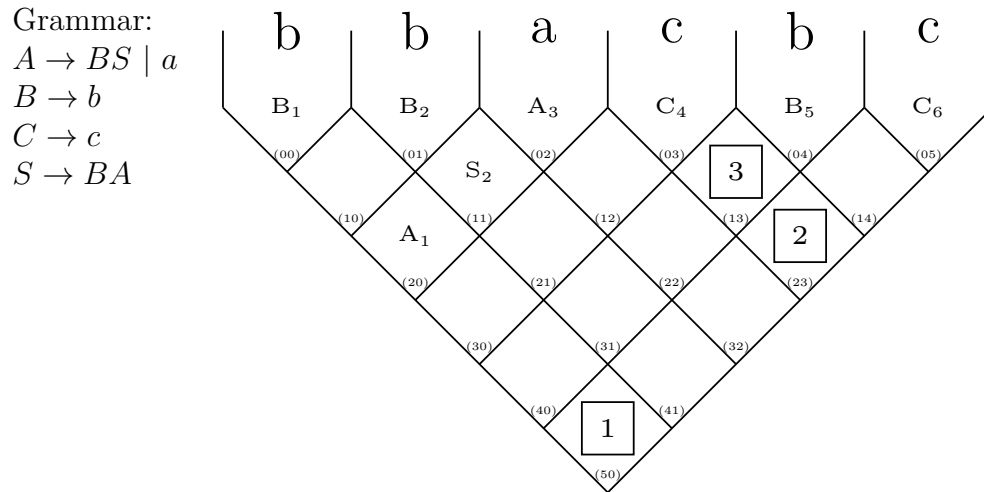


Figure 10: Illustration of Algorithm 8 part 3.

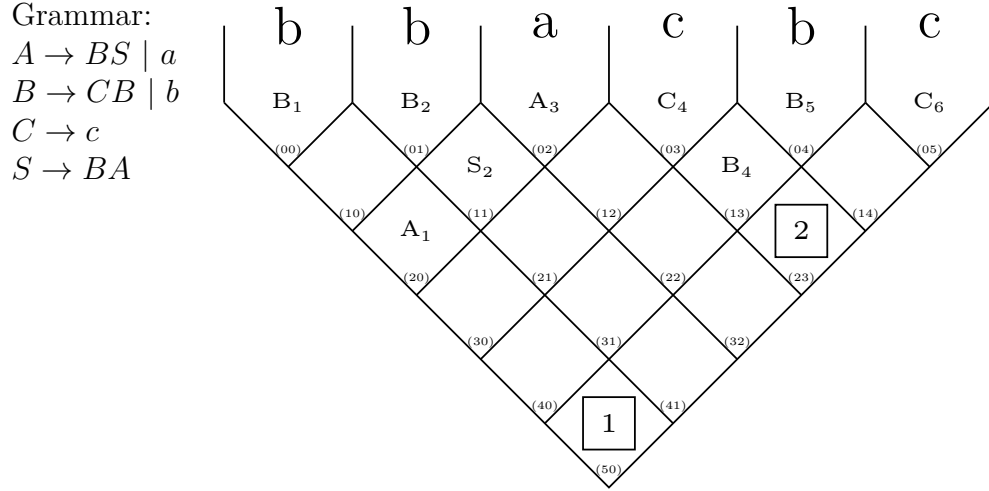


Figure 11: Illustration of Algorithm 8 part 4: Recursion step in  $Cell_{1,3}$  is resolved by adding the rule  $B \rightarrow CB$ .

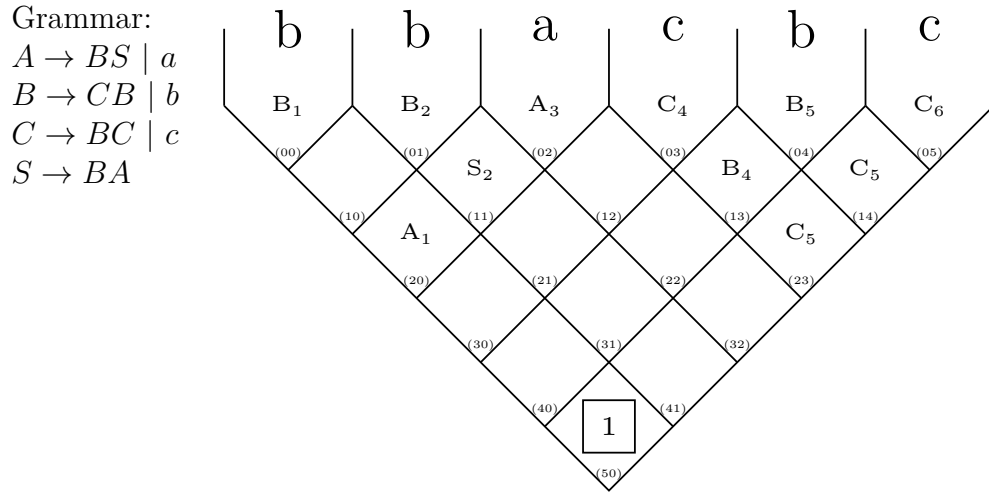


Figure 12: Illustration of Algorithm 8 part 5: : Recursion step in  $Cell_{2,3}$  is resolved by adding the rule  $C \rightarrow BC$ .

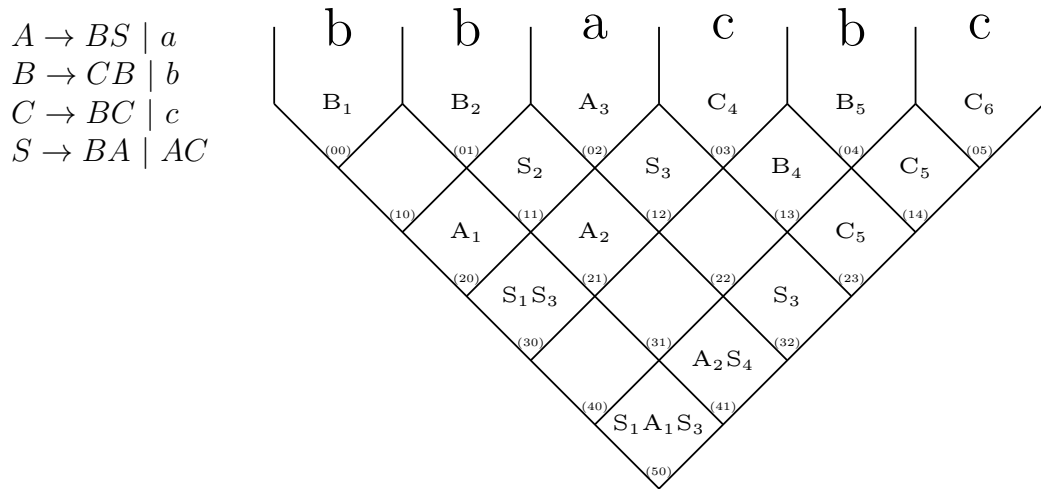


Figure 13: Illustration of Algorithm 8 part 6:: Recursion step in  $Cell_{5,0}$  is resolved by adding the rule  $S \rightarrow AC$ .

## 2.6 SplitAndFill

Algorithm 8 creates the derivation tree structure Top-Downwards but adds rules to the grammar Bottom-Upwards. Another way to do this would be an attempt to do both Top-Downwards.

This algorithm therefore makes use of a part of the same idea as Algorithm 8. It also uses the concept of generating a predefined structure of the derivation tree Top-Downwards. The difference now is, that every time a node of the structure of the derivation tree has been decided on a rule is directly added to the grammar – therefore the name SplitAndFill, which is like "split for a node and then directly add a rule so that the node is then filled with at least one variable".

Note that the count of rules in the grammar is dependent on the count of nodes in the derivation tree and while using  $\textcircled{A}$  a terminal is distributed to exactly one variable.

Algorithm 10: SplitAndFill
<b>Input:</b> Word $w \in \Sigma^*$ <b>Output:</b> Set of productions $P$ 1 $P = \emptyset$ ; // $P \subseteq V \times (V^2 \cup \Sigma)$ 2 $Sol = (P_{Sol}, v)$ ; // $P_{Sol} \subseteq P$ 3 $Sol = SplitAndFill(P, w, i_{max}, 0)$ ; 4 <b>return</b> $P_{Sol}$ ;

Algorithm 11: SplitAndFillRecursion
<b>Input:</b> $P_{in} \subseteq V \times (V^2 \cup \Sigma)$ , $w \in \Sigma^*$ , $i, j \in \mathbb{N}$ <b>Output:</b> $(P, v)$ 1 $P = P_{in}$ ; 2 <b>if</b> $i = 0$ <b>then</b> 3   <b>return</b> $(P \cup (v, w_j), v_{lhse})$ ; 4 <b>end</b> 5 <i>choose one <math>m</math> uniform randomly in <math>[j + 1, j + i]</math></i> ; 6 $(P, v_l) = SplitAndFill(P, w, (m - j - 1), j)$ ; 7 $(P, v_r) = SplitAndFill(P, w, (j + i - m), m)$ ; 8 <b>if</b> $i = i_{max}$ <b>then</b> 9   <b>return</b> $(P \cup (S, v_l v_r), v)$ ; 10 <b>end</b> 11 <b>return</b> $(P \cup (v, v_l v_r), v)$ ;



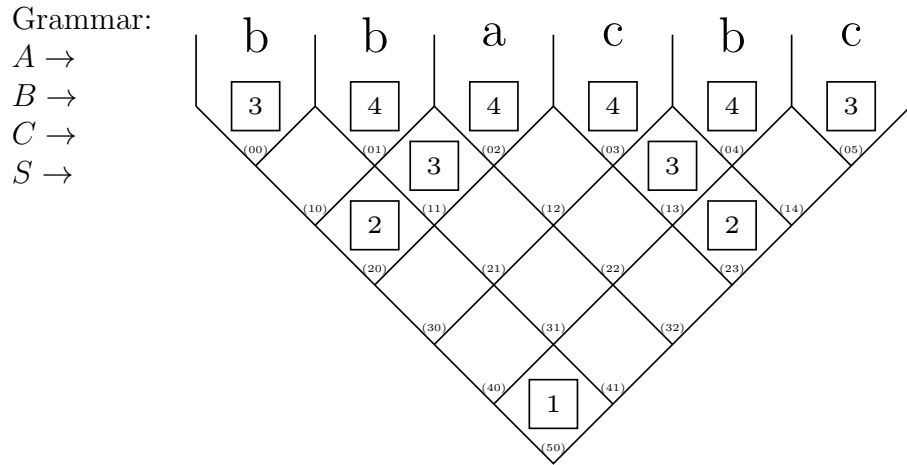


Figure 14: Illustration of Algorithm 10 part 1.

## 2.7 Comparison 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.

Analysis of the problem space possibly visualized in heat maps.

Table 3: My caption

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

## 3 GUI Tool: CYK Instances Generator

### 3.1 Overview GUI

The tool consists out of four major elements as marked in Figure 15.

- In area one elementary input values can be given to the programm.
- The status output of the programm is displayed at area two.
- Area 3 allows to automatically create suitable exercises to choose from.
- In area 4 the chosen exercise can be modified as wanted.

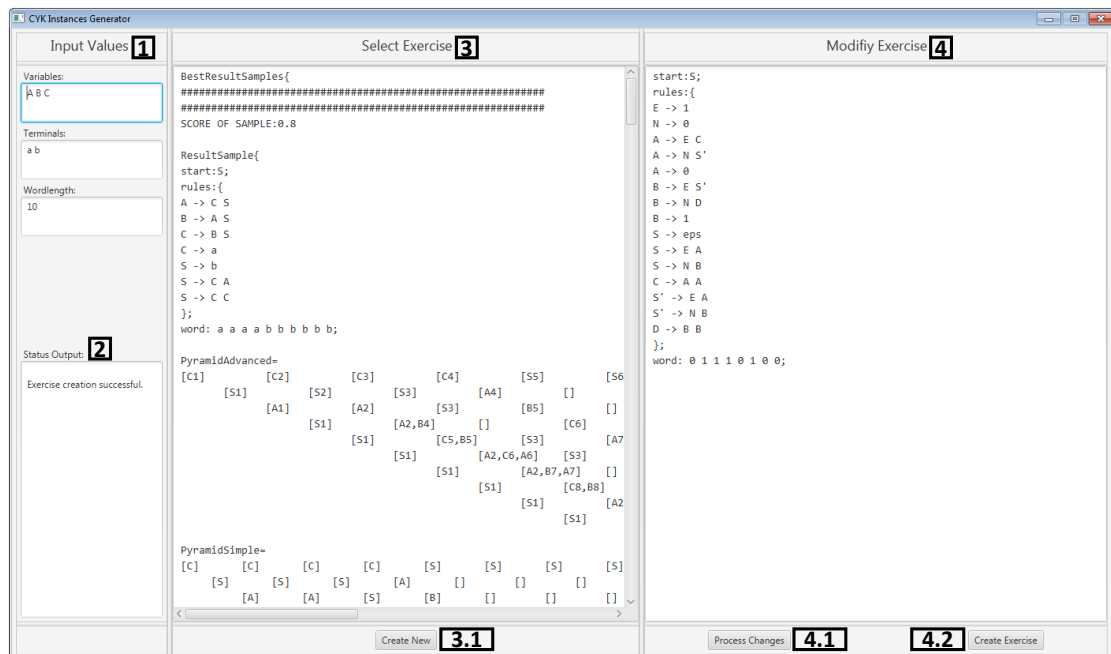


Figure 15: CYK Instances Generator

Clicking the button 3.1 allows the creation of new suitable exercises with the given input values from area 1. Pressing button 4.1 processes the latest input given in area

4 to create a preview analogously to area 3 of how the created exercise would look like and finally button 4.2 creates the desired *exercise*. The output for this *exercise* is done through L<sup>A</sup>T<sub>E</sub>X-code and a pdf-file.

### 3.2 Exam Exercises

The 4-tuple  $exercise = (grammar, word, parse table, derivation tree)$  that is the output of the tool looks as following:

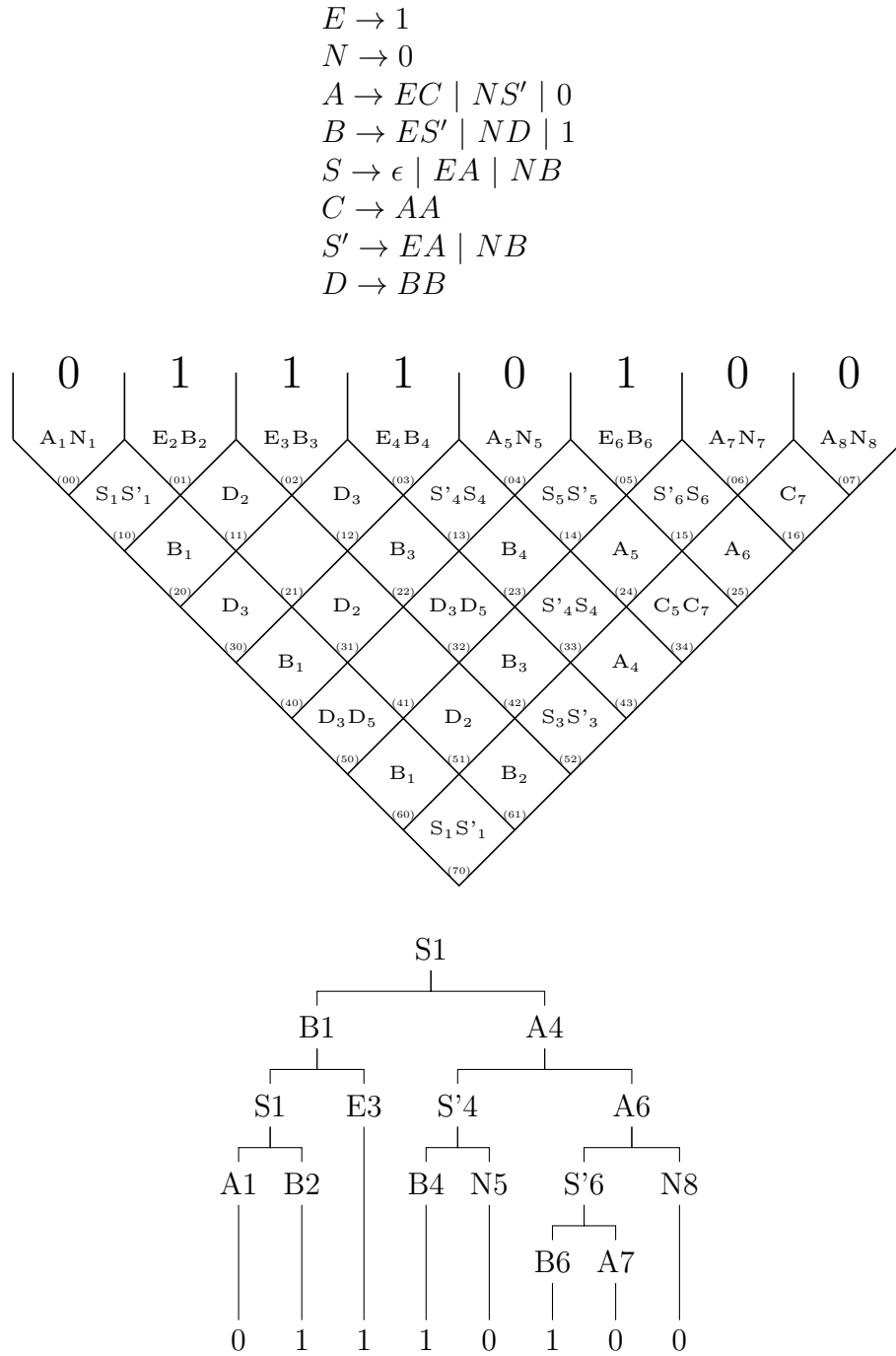


Figure 16: Example output for a *exercise*.

### 3.3 Scoring Model

To be able to find suitable exercises that can be displayed in area 3 of the tool some kind of scoring is needed. ResultSamples are given a score according to Table 4. Parameter that will influence the score are:

- RightCellCombinationsForcedCount
- maxSumOfVarsInPyramidCount
- maxNumberOfVarsPerCellCount
- maxSumOfProductionsCount
- homogeneity
- maxVarKsPerCell

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 *exercise* is scored. The result will be normalized to the maximum possible points -> range 0.0 to 1.0.

## 3.4 Parsing input with ANTLR

## 3.5 Other matters

Technologies that have been used for programming are Github/Sourcetree, Maven, IntelliJ, ANTLR and JavaFX Scene Builder. [XXX]

Important used frameworks are: JUnit and Project Lombok. [XXX]

Altogether around 7100 lines of code have been written, of which 5400 are pure java code, 900 are comment lines and 800 are blank lines.

### 3.5.1 Working with the program

Short introduction on how to run the program and where to find the output exercise.

## References

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

