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 31, 2017
1. Supervisor:	Prof. Dr. Wim Martens
2. Supervisor:	Dr. Matthias Niewerth

Abstract

Every year, lecturers of theoretical computer science or a related field are confronted with the task of examining whether their students have understood the workings of the Cocke-Younger-Kasami algorithm. There are several implementations and smaller online tools of the CYK algorithm already in place, but none of them support the actual process of creating a suitable exercise for it.
Different algorithms were first designed to generate suitable exercises and then compared with each other using their success rate. The different approaches of these algorithms involve the uniformly randomly distribution of elements and the general Bottom-Up and Top-Down parsing approaches.
A GUI tool has been implemented to automatically generate exam exercises. The functionality of the tool includes that input parameters such as the number of variables, the number of terminals and the word length can be made. Suitable exam exercises are automatically generated from which one can be selected for further modification and creation of the exam exercise.

Zusammenfassung

Jedes Jahr stehen Dozenten der theoretischen Informatik oder Doezenten eines verwandten Bereiches vor der Aufgabe Klausuraufgaben zu erstellen, um zu prüfen ob ihre Studenten die Arbeitsweise des Cocke-Younger-Kasami-Algorithmus verstanden haben. Verschiedene Implementierungen und kleinere Online-Tools des CYK-Algorithmus gibt es bereits, aber Keines unterstützt beim Prozess des Erstellens einer Aufgabe.
Verschiedene Algorithmen wurden zuerst entworfen, um genau passende Aufgaben zu generieren und wurden anschließend miteinander über ihre Erfolgsrate verglichen. Die unterschiedlichen Ansätze für die Algorithmen beinhalten das gleichmäßig zufällige Verteilen von Elementen und die allgemeinen Ansätze des Bottom-Up und Top-Down Parsings.
Es wurde ein GUI-Tool implementiert um automatisch Klausuraufgaben zu generieren. Die Funktionalität des Tools beinhaltet, dass Eingabewerte wie die Anzahl der Variablen, die Anzahl der Terminale und die Wortlänge gemacht werden können. Geeignete Klausuraufgaben werden automatisch generiert von denen Eine für weitere Modifikation und für die Klausuraufgabenerstellung ausgewählt werden kann.

Contents

Abstract 4

1 Introduction 5
- 1.1 Motivation 5
- 1.2 Context Free Grammar 5
- 1.3 General approaches of parsing 7
- 1.4 Data Structure Pyramid 8
- 1.5 Cocke-Younger-Kasami Algorithm 9

2 Algorithms 11
- 2.1 Sub Modules 11
- 2.2 Dice rolling the distributions only 13
- 2.3 Dice rolling and Bottom-Up variant one 14
- 2.4 Dice rolling and Bottom-Up variant two 17
- 2.5 Split Top-Down and fill Bottom-Up variant one 20
- 2.6 Split Top-Down and fill Bottom-Up variant two 26
- 2.7 Evaluation of Algorithms 31
  - 2.7.1 Success Rates 31
  - 2.7.2 Problem space exploration 33
    - 2.7.2.1 Comparison of the stopping criteria 34
    - 2.7.2.2 Picking the best algorithm 35

3 GUI Tool: CYK Instances Generator 36
- 3.1 Overview GUI 36
  - 3.1.1 Working with the program 36
- 3.2 Exam Exercise 37
- 3.3 Scoring Model 38
- 3.4 Parsing input with ANTLR 39
- 3.5 Other Matters 40

Erklärung 42

1 Introduction

1.1 Motivation

Every year, lecturers of theoretical computer science or a related field face the task to create the 4-tuple exam $exercise = (grammar,\ word,\ parse\ table,\ derivation\ \ tree)$ that tests if their students have understood the workings of the Cocke-Younger-Kasami (CYK) algorithm. For that exercises need to be created which is a bit of a time consuming task.
There are several implementations and smaller online tools of the CYK algorithm already in place [1] [2] [3], but none of them support the actual process of creating a suitable exercise for it. Therefore algorithms are needed to generate specifically suitable exercises with a high chance of success. Also a GUI tool, that allows automatic generation of the suitable exam exercises and further modification, is required and so a own solution is implemented.

1.2 Context Free Grammar

Firstly, we define a Context Free Grammar (CFG) as follows:

Definition 1. Context Free Grammar (CFG)
A CFG is a 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 holds: $\Sigma \cap V = \emptyset$.

Secondly, we define a CFG with restrictions (CFGR) as:

Definition 2. CFG with restrictions (CFGR)
A CFG $G = (V,\ \Sigma,\ S,\ P)$ is a CFGR if:

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

Throughout this thesis a grammar is always synonymous with Definition 2. Note that a CFGR is not necessarily in Chomsyk Normal Form (CNF) because it is still possible that there are unreachable variables – from the starting variable – or useless rules. For further convenience the following default values are always assumed in this thesis:

[1]CYK online tool: http://lxmls.it.pt/2015/cky.html
[2]CYK parser implementation: http://jflap.org/tutorial/grammar/cyk/index.html
[3]CYK algorithm implementation in Java: https://github.com/ajh17/CYK-Java

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

A rule consists of a left hand side element ($lhse \in Lhse$) and a right hand side element ($rhse \in Rhse$).

Example: lhse and rhse
$lhse \longrightarrow rhse$ applied to $A \longrightarrow c$ and $B \longrightarrow AC$ means that A and B are a $lhse$ and c and AC are a $rhse$. Elements of V^2 are often referred to as variable compounds.

For a word and a sub word Definition 3 holds:

Definition 3. Word w and sub word

- Word w: $w = w_0 \cdot w_1 \cdot ... \cdot w_j$ and $w \in \Sigma^*$.
- Sub word sw of a word w: $sw = w_k \cdot ... \cdot w_{l+k}$ where $0 \leq i$ and $l + k \leq j$.

For a language and a language over a grammar Definition 4 holds:

Definition 4. Language L and language $\mathbf{L(G)}$

- Language L: L is a language over an alphabet Σ, that is a subset of Σ^*, meaning it is a set of words over alphabet Σ.
- Language $L(G)$: $L(G)$ is the language over a grammar G, that is the set of words the grammar G describes.

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.

1.3 General approaches of parsing

Next, the basic approach that may help finding a good algorithm is explained informally analogous to [1]. At first, parsing is described in general and afterwards its two characteristics are explained.

Definition 5. Derivation
A derivation of a word $w \in \Sigma^*$ over a grammar $G = (V,\ \Sigma,\ S,\ P)$ is a sequence of words $v_0, v_1, ..., v_n,\ v_i \in (V \cup \Sigma)^*$, $v_0 = S$, $v_n = w$, where v_{i+1} is obtained by replacing an occurrence of some *lhse* of a rule $p \in P$ by the corresponding *rhse* in word v_i.

Definition 6. Backward Problem = Parsing ($\mathbf{w} \overset{?}{\in} \mathbf{L(G)}$
Input: w and a grammar G.
Output: Derivation of w over G or error if $w \notin L(G)$.

It is called parsing if a word w given and it is of interest to know if it is element of $L(G)$. Parsing is also the basis of the Cocke-Younger-Kasami algorithm.
After having defined what parsing in general is, it is important to know the two different ways of parsing, that will act as an idea provider for the algorithms.

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

Actually, Bottom-Up parsing is the method used in the Cocke-Younger-Kasami algorithm, which fills the parse table from the "bottom up" [1].
Bottom-up parsing starts by recognizing the words smallest sub words before its midsize sub words and leaves the largest overall word as the last.

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

"Top-Down parsing starts with the root node and successively applies rules from P, with the goal of finding a derivation of the test sentence w." [1] (The so called test sentence is synonymous to a word w.)

1.4 Data Structure Pyramid

To be able to describe how the different algorithms work in a simpler way, the help data structure *Pyramid* is defined – note that *Pyramid* is a set and starts with an upper case letter.

Definition 7. Pyramid
$Pyramid := \{Cell_{i,j} \mid i \in [0,\ i_{max}],\ j \in [0,\ j_{max,i}],\ i_{max} = |w|-1,\ j_{max,i} = i_{max}-i\}$
where $Cell_{i,j} \subseteq \{(V,k) \mid k \in \mathbb{N}\}$ denotes the contents of the j'th cell in row i
and $[i,\ j] := \{i,\ i+1, ...,\ j-1,\ j\} \subseteq \mathbb{N}$.

The cell $Cell_{i_{max},0}$ is called the root of such a *Pyramid* and Figure 1.4 shows the visual representation of one.

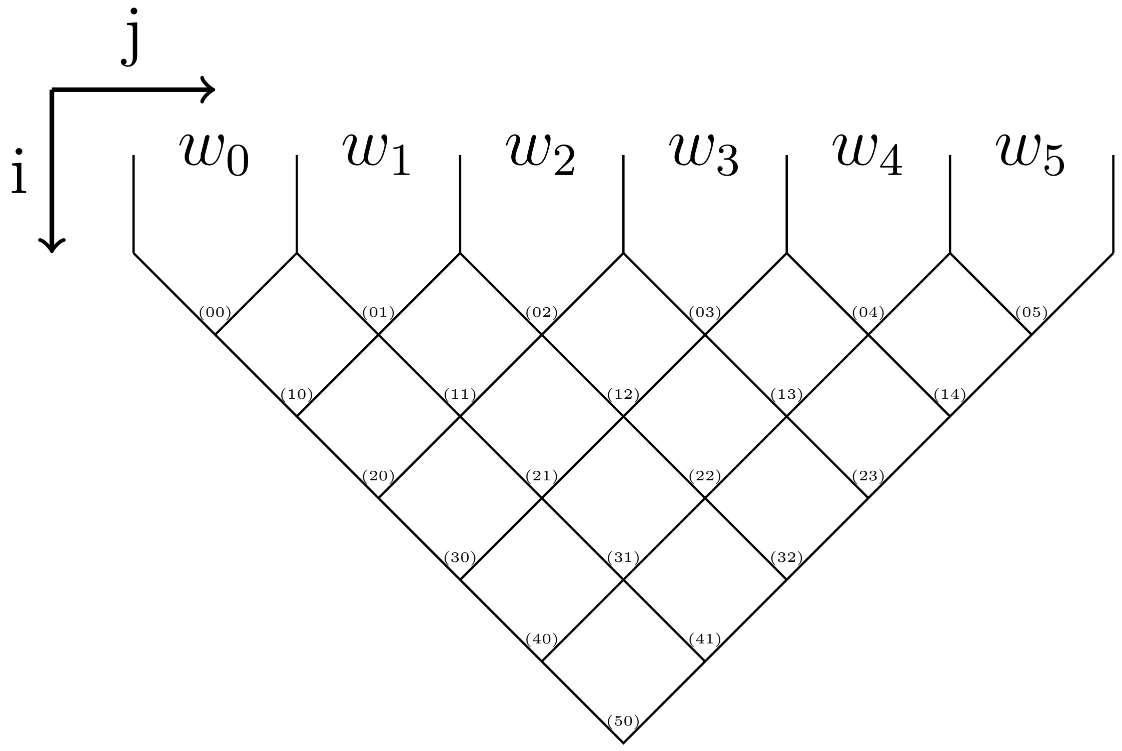


Figure 1: Visual representation of a *Pyramid* with the word w written above it.

1.5 Cocke-Younger-Kasami Algorithm

The Cocke-Younger-Kasami Algorithm (CYK) was independently developed in the 1960s by Itiroo Sakai [2], John Cocke and Jacob Schwartz [3], Tadao Kasami [4] and Daniel Younger [5].
The idea is to find all possible derivations of each sub word starting with size one and to consecutively use this information to find all possible derivations with a larger size of the subword up to the size of w. Finally it is checked whether $w \in L(G)$ through the presence of the start variable in the root of the pyramid.
The description of the algorithm follows the source [6] adjusted to the data structure *Pyramid*. Later on it can be seen that the CYK algorithm can be used as a basis to find good algorithms.

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

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 13: True if $Cell_{i_{max},0}$ contains the start variable.

Example: Algorithm CYK

During the execution of the CYK algorithm the parsing table is filled as shown in Figure 2. At first the row with index $i = 0$ is filled after Line 2 to Line 4 of the CYK algorithm, i.e. a $Cell_{0,j}$ will contain the variable if it has the terminal w_j as its *rhse*. Then for each row i every cell with index j is looked at. Every possible combination of sub words for a cell are taken into account, i.e. for $Cell_{4,1}$ there are the combinations of $(Cell_{0,1}, Cell_{3,2})$, $(Cell_{1,1}, Cell_{2,3})$, $(Cell_{2,1}, Cell_{1,4})$ and $(Cell_{3,1}, Cell_{0,5})$. Applying Line 8 for example to the cell combination $(Cell_{2,1}, Cell_{1,4})$ it leads to $X \rightarrow AC$ here and because the compound variable AC is *rhse* of the variable S, the $Cell_{4,1}$ contains the element $(S, 4)$.

$A \rightarrow BS \mid a$
$B \rightarrow CB \mid b$
$C \rightarrow BC \mid c$
$S \rightarrow BA \mid AC$

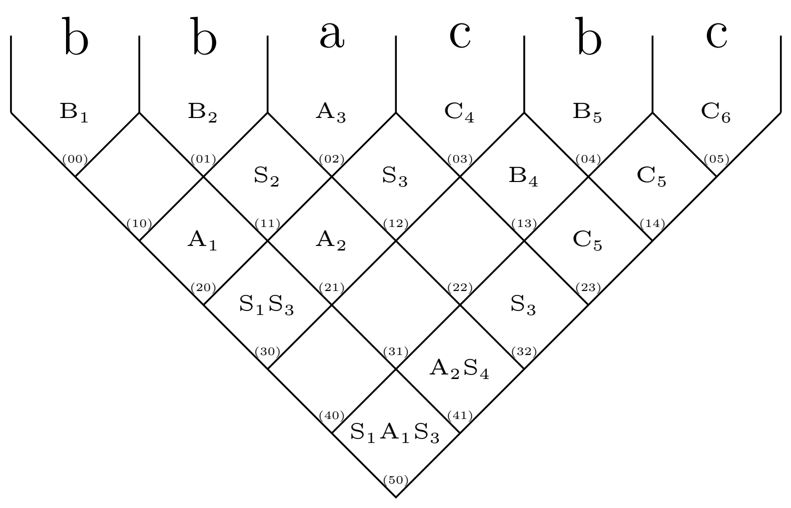


Figure 2: The CYK algorithm fills the cells of the pyramid during execution of Line 3 and Line 8.

2 Algorithms

2.1 Sub Modules

Sub modules are parts of the algorithms that are denoted circled with $\textcircled{A}$, $\textcircled{B}$, $\textcircled{C}$, $\textcircled{D}$ and $\textcircled{E}$. They are procedures that will be explained in more detail for a better understanding of the algorithms in the following chapters. $\textcircled{E}$ is explained in Chapter 2.4 because it is needed only there.

Distribute$(\Sigma,\ V)$ $\textcircled{\textbf{A}}$ **and Distribute**$(V^2,\ V)$ $\textcircled{\textbf{B}}$**:**

The difference between $\textcircled{A}$ and $\textcircled{B}$ is that one time Σ and the other time V^2 are distributed. In both cases a uniform random subset of Σ or V^2 uniformly randomly distributed over the set of available variables V. 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 described in the following algorithm:

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

Stopping Criteria $\textcircled{\textbf{C}}$**:**

Two kinds of stopping criteria are used to determine whether an algorithm should terminate early on because an already suitable exercise has been found:

- stop if more than half of the pyramid cells are not empty any more.
- stop if the root of the pyramid is not empty any more.

Both stopping criteria are compared in Chapter 2.7 to see which one leads to more suitable exam exercises.

CalculateSubsetForCell(Pyramid, i, j) $\textcircled{\textbf{D}}$:

This procedure is needed to determine all possible compound variables out of all possible cell combinations for one specific cell. It works kind of analogous from Line 7 to Line 9 of the CYK algorithm (Algorithm 1).

Algorithm 3: CalculateSubsetForCell

```
    Input: $Pyramid,\ i \in \mathbb{N},\ j \in \mathbb{N}$
    Output: $CellSet \subseteq V^2$
1   $CellSet = \emptyset$;
2   for $k := i - 1 \to 0$ do
3       $CellSet = CellSet \cup \{YZ \mid X \longrightarrow YZ,\ (Y, ...) \in Cell_{k,j},\ (Z, ...) \in Cell_{i-k-1,k+j+1}\}$;
4   end
5   return $CellSet$;
```

Example: Algorithm CalculateSubsetForCell

In the following situation a rule is added to P while using while using Algorithm 3 on $Cell_{3,0}$.

Grammar:

$A \to AB \mid a$

$\mathbf{B} \to \mathbf{SC} \mid b$

$C \to AB$

$S \to BA \mid AA$

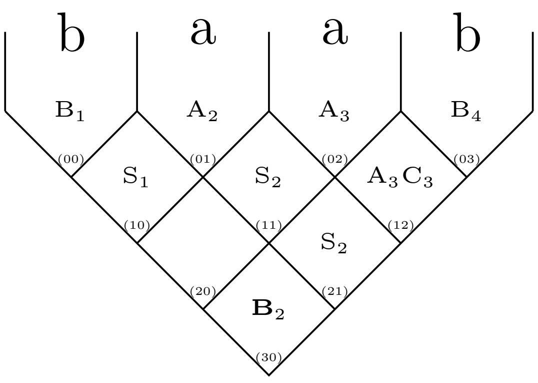


Figure 3: Example of Algorithm 3 while applying it on $Cell_{3,0}$ via adding the rule $B \to SC$.

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 comes from $Cell_{0,0}$ together with $Cell_{2,1}$. Now if either one of the rules $lhse \to SA$, $lhse \to SC$ or $lhse \to BS$ is added to the grammar, then $lhse \in Cell_{3,0}$. Here the rule $\mathbf{B} \to \mathbf{SC}$ has been added and finally $(B, 2)$ is element of $Cell_{3,0}$.

In general if for one $Cell_{i,j}$ a rule like $lhse \to cs$ with $cs \in CellSet$ (Line 3) is added, then automatically $Cell_{i,j}$ is not empty any more.

2.2 Dice rolling the distributions only

We start off by a primitive way of generating grammars, which can be used the lower boundary while comparing the algorithms. Note that later on in Chapter 2.7.1 it is described what "performing better" means in the context of this thesis.

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

The algoritm DiceRollOnly (Algorithm 4) distributes terminals Σ to at least one *lhse*. Note that for each terminal of $\Sigma = \{a, b\}$ at least one rule like $lhse \rightarrow a$ and $lhse \rightarrow b$ is generated. On the other hand not every compound variable $vc \in V^2$ has to be distributed.

Example: Algorithm DiceRollOnlyCYK
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 (Figure 4) 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 4: Shortened overview of the example of Algorithm 4.

2.3 Dice rolling and Bottom-Up variant one

Another approach to design an algorithm uses the Bottom-Up approach (Chapter 1.3) in which the parsing table is filled starting from the leaves in direction of the root node. The basic idea is to guide the choice of rules while distributing the compound variables V^2. In Algorithm 4, the naive approach, it is possible that the terminals are distributed to the variables A and B and Algorithm 4 completely discards this fact during the distribution of the compound variables (see Figure 6 in the middle).

Example: Disregarding already added rules
Figure 5 shows the starting situation for this example:

Grammar:
$A \rightarrow a$
$B \rightarrow b$

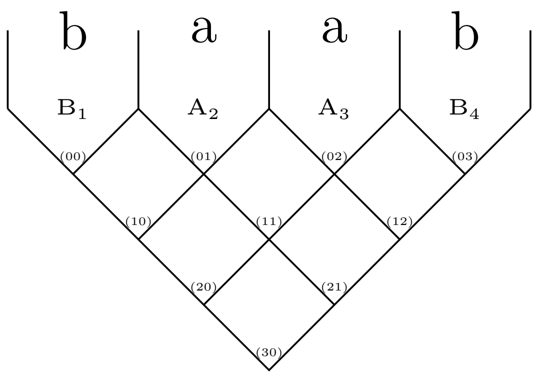


Figure 5: Disregarding the already added rules: Starting situation.

If rules like $lhse \rightarrow CC$ or $lhse \rightarrow SC$ are added they do not directly help to fill the parsing table and bloat the grammar with rules (see Figure 6). This is a unfortunate adding of rules that does not help to fill the parsing table and this may happen in Algorithm 4.

Grammar:
$A \rightarrow \mathbf{CC} \mid a$
$B \rightarrow b$

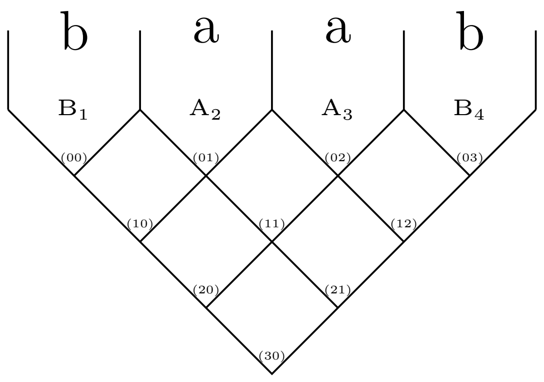


Figure 6: Disregarding the already added rules: Unfortunate adding.

More reasonable rules to add would be $lhse \rightarrow BA$, $lhse \rightarrow AA$ or $lhse \rightarrow AB$ (see Figure 7). This is an advantageous adding of rules as intended in Algorithm 5 that helps to fill the *pyramid*.

Grammar:
$A \rightarrow a$
$B \rightarrow \mathbf{AB} \mid b$

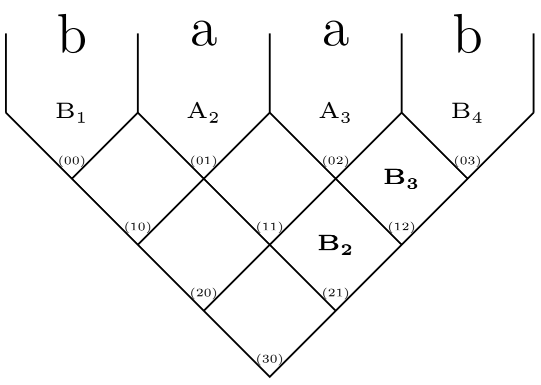


Figure 7: Disregarding the already added rules: Advantageous adding of rules.

Algorithm 5 is seen on the next page.

Algorithm 5 continues on this idea: After distributing the terminals (Line 2) the updated parsing table (Line 12) is always taken into consideration while calculating variable compounds (Line 10) and to finally add a part of them as rules (Line 11) to the grammar. As explanation, for each chosen cell a *CellSet* (Line 10) is calculated, that only contains reasonable variable compounds. This way only variable compounds are added that directly help to fill the parsing table.

Algorithm 5: BottomUpDiceRollVar1

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

Line 3: Fills the i=0 row of the pyramid. Line 9: A cell is visited only once.

2.4 Dice rolling and Bottom-Up variant two

While examining Algorithm 5 via its log file (Figure 8) it can be observed 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 ($i = 2$).

```
Final cell worked with Index: 1,2
Final cell worked with Index: 1,0
Final cell worked with Index: 1,6
Final cell worked with Index: 1,0
Final cell worked with Index: 1,2
Final cell worked with Index: 1,3
Final cell worked with Index: 2,4
```

Figure 8: Digest of the log file of Algorithm 5 with $|V| = 4$ and $|\Sigma| = 2$.

This again leads to the next improvement idea. This is to introduce a row dependent $threshold_i$ (Line 9 of Algorithm 6 BottomUpDiceRollVars) which helps that more cells with $i \geq 2$ are chosen – what possibly leads to more diverse grammars being generated. The diversity, in context of the procedure BottomUpDiceRollVar1 (Algorithm 5), 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 of these *lhse*s as explained in Chapter 2.3. This is caused by the basic idea of Algorithm 5 but also due to the relatively small number of rules that are added to the grammar altogether.
Further diversification is achieved through the usage of $\textcircled{E}$ (Line 10 of Algorithm 6 BottomUpDiceRollVars), i.e. the variable compounds that already have been used in a row with low index i are at a disadvantage to be picked again (A more detailed explanation is found at the and of this chapter).

Example: Better diversity in a grammar
As seen in Figure 9 the rules with BA and AA are added to the variables B and A in Grammar1. For Grammar2 instead the rule $B \to SS$ is added that contributes to a better diversity compared to Grammar1. Grammar2 contains one more unique *rhse* (SS) compared to Grammar1.

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

Figure 9: Better diversity.: Starting point is Grammar0 and Grammar2 is of better diversity than Grammar1.

Algorithm 6: BottomUpDiceRollVar2

```
   Input: Word $w \in \Sigma^*$
   Output: Set of rules $P$
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  $P = Distribute(\Sigma,\ V)$; Ⓐ
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 $(xy, i)$ from RowSet uniformly randomly with
             probability depending on $i$; Ⓔ
11         $P = P \cup Distribute(xy,\ V)$; Ⓑ
12         $Pyramid = CYK(G,\ w)$;
13         if stopping criteria met Ⓒ then
14             return $P$;
15         end
16     end
17 end
18 return $P$;
```

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

Choose one xy from (xy,i) $\in$ RowSet uniformly randomly with probability depending on row i Ⓔ :

At some point a decision needs to me made about what rule $lhse \rightarrow xy$ with $xy \in V^2$ will be added to the grammar. Depending on the chosen xy the influence on the entire pyramid varies. Some xy only change the parsing table in one of its later rows ($i >> 1$) but other xy even change it in one of the first rows. If there is a change in one of the first rows it is more likely that the entire pyramid will be filled with more elements. Now the task of choosing rules to add, that only change the pyramid in one of the later rows with a higher probability than the others is tackled with Ⓔ.

The approach here only makes sense together with Ⓓ where all possible compound variables are calculated that would help to fill one specific cell. $RowSet \subseteq \{(xy, i) \mid x, y \in V \wedge i \in \mathbb{N}\}$ where the xy are calculated with Ⓓ and i is the row number of the specific cell.

Example: Procedure Ⓔ

With *RowSet* the choice can be influenced depending on the row number i: Firstly the *RowSet* is compressed, i.e. every tuple with the same xy will be merged to its lowest i, as following: $RowSet = \{(AB,3),\ (AB,1),\ (AB,5),\ ...\}$ will become $RowSet = \{(AB,1),\ ...\}$. 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 while elements like $(BC,3)$ will occur three times and so on: $RowMultiSet = \{(AB,1),\ (BC,3),\ ...\}$ becomes $RowMultiSet = \{(AB,1),\ (BC,3),\ (BC,3),\ (BC,3),\ ...\}$. Now one element will be chosen uniformly randomly out of this weighted *RowMultiSet*. In Figure 10 this results in $xy = BC$.

$$\begin{array}{ll}
RowSet = \{(AB,3),(AB,1),(AB,5),...\} & \text{// compress} \\
RowSet = \{(AB,1),...\} & \text{// put into RowMultiSet} \\
RowMultiSet = \{(AB,1),(BC,3),...\} & \text{// weight elements} \\
RowMultiSet = \{(AB,1),(BC,3),(BC,3),(BC,3),...\} & \text{// pick element} \\
xy = BC &
\end{array}$$

Figure 10: Overview of the procedure E.

2.5 Split Top-Down and fill Bottom-Up variant one

Until now only algorithms have been discussed that purely use the Bottom-Up approach. Another way is to utilize the Top-Down approach in combination with the Bottom-Up approach.
The idea here is first to distribute the terminals (Line 2 of Algorithm 7 SplitThenFill-CYK) and then to uniformly randomly generate a predefined structure of the derivation tree (Line 5 of Algorithm 7 and in general Algorithm 8 SplitThenFillCYKRec) Top-Downwards and then again to fill the parsing table Bottom-Upwards accordingly to fill this derivation tree. The structure of the derivation tree for instance can look as follows:

Example: Derivation structure in the pyramid and as a derivation tree
The numbers correspond to the depth of cell in the tree.

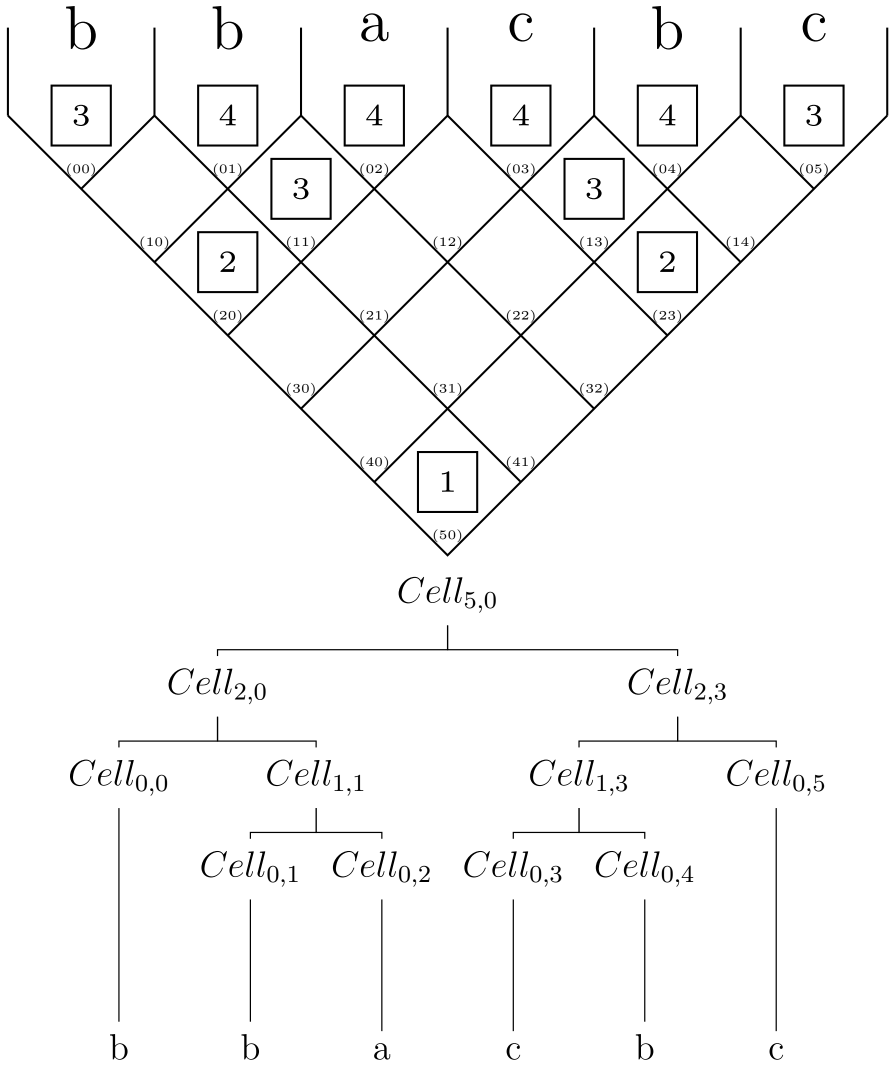


Figure 11: Derivation structure shown as a pyramid in the top and as a derivation tree below.

As the name of the algorithms implies only after completely generating the structure of the derivation tree (splitting of the word in sub words) the rules are added to the grammar that help filling the cells occurring in this derivation tree. The phrase CYK is included in the name because in every recursion the CYK algorithm is executed (Line 8 of Algorithm 8) once.

Algorithm 7: SplitThenFillCYK

```
Input: Word $w \in \Sigma^*$
Output: Set of rules $P$
1 $P = \emptyset$; // $P \subseteq V \times (V^2 \cup \Sigma)$
2 $P = Distribute(\Sigma,\ V)$; $\textcircled{A}$
3 $Pyramid = CYK(G,\ w)$;
4 $Sol = (P_{Sol},\ Cell_{i_{max},0})$; // $P_{Sol} \subseteq P \ \wedge \ Cell_{i_{max},0} \in Pyramid$
5 $Sol = SplitThenFillCYKRec(P,\ w,\ i_{max},\ 0)$;
6 return $P_{Sol}$;
```

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

Now every time before adding a new rule (Algorithm 8 SplitThenFillCYKRec Line 15) 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 8 SplitThenFillCYKRec).

Algorithm 8: SplitThenFillCYKRec

```
   Input: $P_{in} \subseteq V \times (V^2 \cup \Sigma),\ w \in \Sigma^*,\ i, j \in \mathbb{N}$
   Output: $(P,\ Cell_{i,j})$
1  $P = P_{in}$;
2  if $i = 0$ then
3      return $(P,\ Cell_{i,j})$;
4  end
5  choose one $m$ uniformly randomly in $[j+1,\ j+i]$;
6  $(P,\ Cell_l) = SplitThenFillCYKRec(P,\ w,\ (m-j-1),\ j)$;
7  $(P,\ Cell_r) = SplitThenFillCYKRec(P,\ w,\ (j+i-m),\ m)$;
8  $Pyramid = CYK(G,\ w)$;
9  if stopping criteria met $\textcircled{C}$ then
10     return $(P,\ Cell_{i,j})$;
11 end
12 if $Cell_{i,j} = \emptyset$ then
13     $VarComp$ = uniform random subset of $\{vc \mid v \in Cell_l \wedge$
14         $c \in Cell_r\}$ with $|VarComp| \geq 1$;
15     foreach $vc \in VarComp$ choose $v \in V$ uniformly randomly and add the
           rule $v \longrightarrow vc$ to $P$;
16 end
17 return $(P,\ Cell_{i,j})$;
```

Line 8: Needed to update the *pyramid* to be able to check if the stopping criteria is met in Line 9.

Example: Algorithm SplitThenFillCYK

The same example tree structure as in Figure 11 is used here – remember that each number represents the recursion depth of its subtree.

The situation after adding the terminals (Line 2 in Algorithm 7 SplitThenFillCYK) to the grammar is shown in Figure 12:

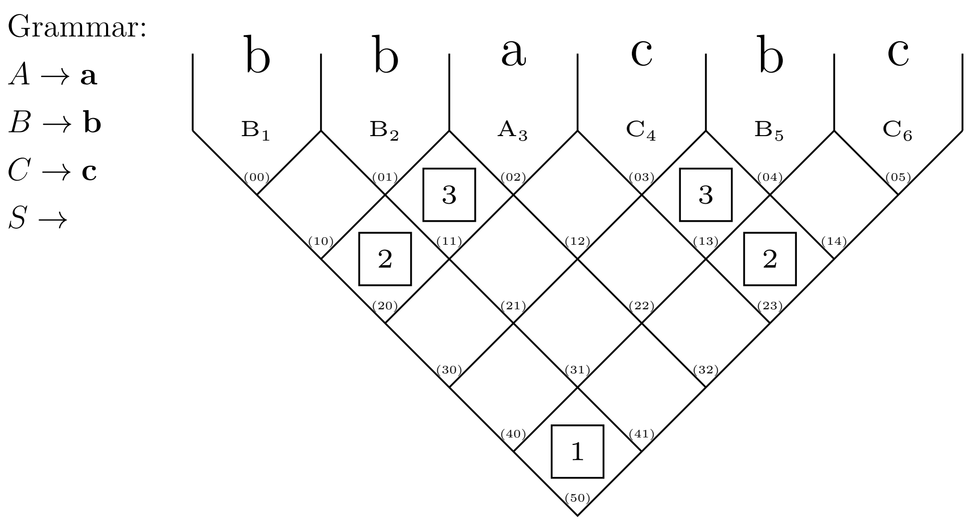


Figure 12: Illustration of Algorithm 7 SplitThenFillCYK part 1 after adding the rules $A \rightarrow a$, $B \rightarrow b$ and $C \rightarrow c$.

After adding the rules for the terminals to the grammar the recursion step at $Cell_{1,1}$ is taken on. Now $Cell_l = \{B_2\}$ and $Cell_r = \{A_3\}$ and therefore $VarComp = \{BA\}$. Adding the rule $S \rightarrow BA$ leads to the following *Pyramid* shown in Figure 13:

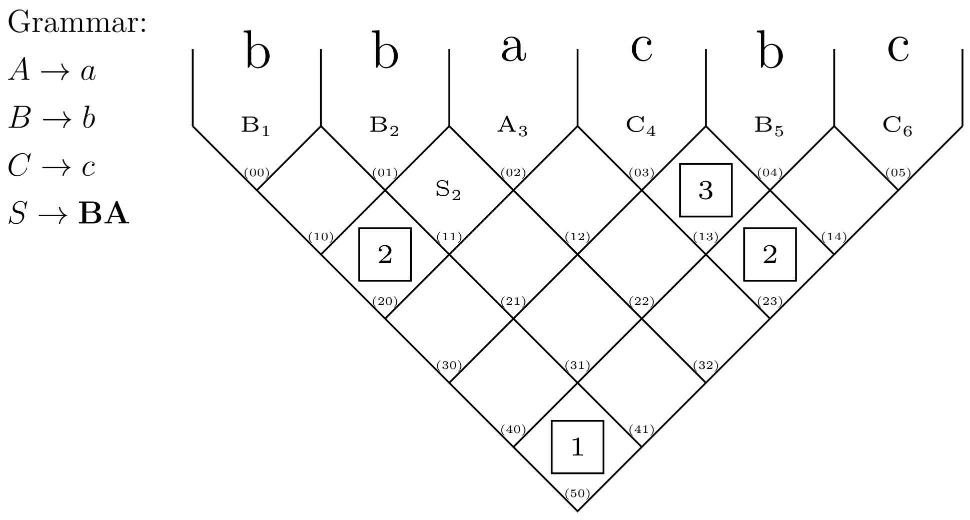


Figure 13: Illustration of Algorithm 7 SplitThenFillCYK part 2 after adding the rule $S \rightarrow BA$.

The next recursion step happens in $Cell_{2,0}$. Now $Cell_l = \{B_1\}$ and $Cell_r = \{S_2\}$. Analogously the rule $A \to BS$ is added to the grammar as seen in Figure 14:

Grammar:
$A \to \mathbf{BS} \mid a$
$B \to b$
$C \to c$
$S \to BA$

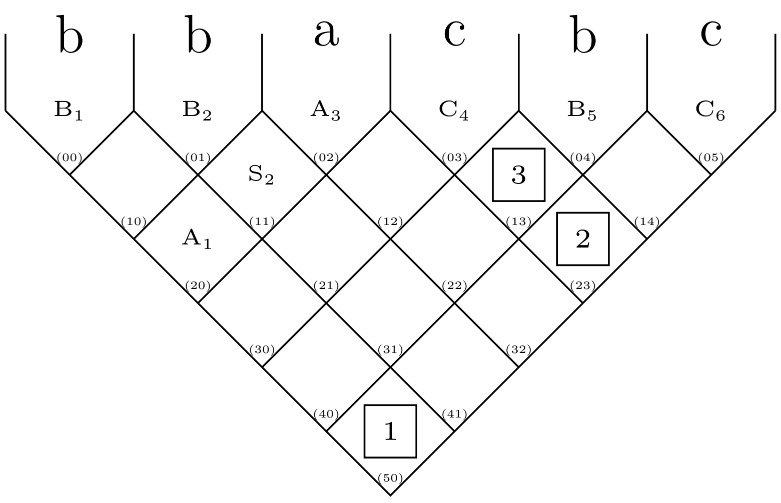


Figure 14: Illustration of Algorithm 7 SplitThenFillCYK part 3 after adding the rule $A \to BS$.

The next two steps are described analogously with Figure 15 and with Figure 16.

Grammar:
$A \to BS \mid a$
$B \to \mathbf{CB} \mid b$
$C \to c$
$S \to BA$

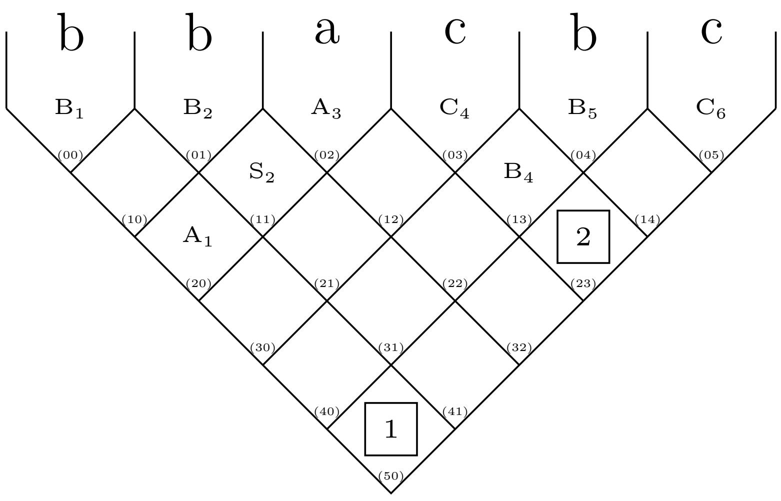


Figure 15: Illustration of Algorithm 7 SplitThenFillCYK part 4. The recursion step in $Cell_{1,3}$ is resolved by adding the rule $B \to CB$.

Grammar:
$A \rightarrow BS \mid a$
$B \rightarrow CB \mid b$
$C \rightarrow \mathbf{BC} \mid c$
$S \rightarrow BA$

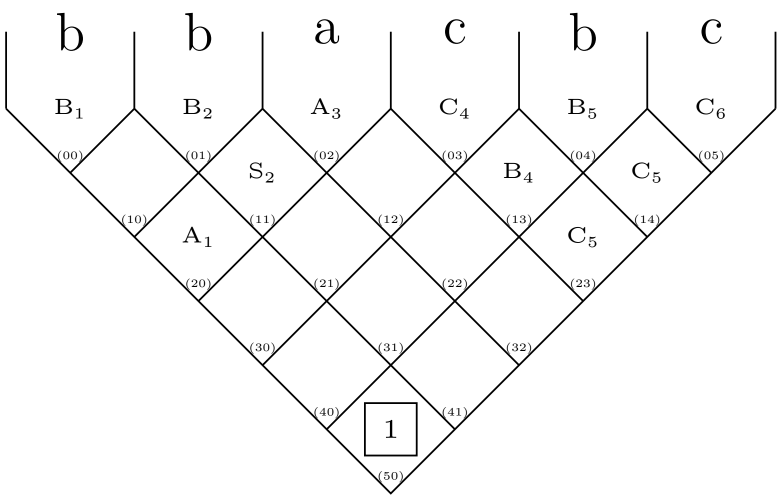


Figure 16: Illustration of Algorithm 7 SplitThenFillCYK part 5. The recursion step in $Cell_{2,3}$ is resolved by adding the rule $C \rightarrow BC$.

Finally the last recursion step that decides on the content of the root cell is shown in Figure 17. In this case the start variable is luckily contained in the root of the pyramid.

$Grammar$:
$A \rightarrow BS \mid a$
$B \rightarrow CB \mid b$
$C \rightarrow BC \mid c$
$S \rightarrow BA \mid \mathbf{AC}$

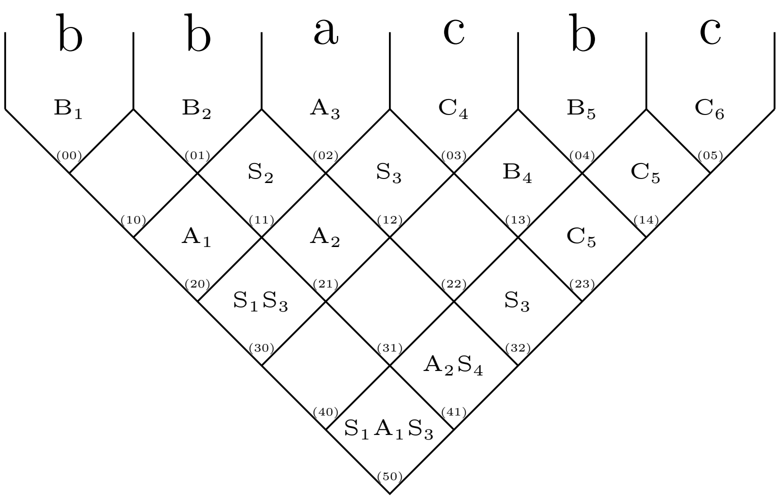


Figure 17: Illustration of Algorithm 7 SplitThenFillCYK part 6. The recursion step in $Cell_{5,0}$ is resolved by adding the rule $S \rightarrow AC$.

2.6 Split Top-Down and fill Bottom-Up variant two

Yet another algorithm that uses the split Top-Down and fill Bottom-Up behaviour is Algorithm 9 SplitAndFill.
The algorithm recursively generates the derivation tree structure in a Top-Down fashion, while the rules are added to the grammar from the bottom up. In Line 2 to Line 8 of Algorithm 10 it is ensured that every terminal of the word is *rhse* of exactly one rule. Line 13, of the same algorithm, guarantees that the start variable is always element of the root cell and Line 16 is responsible to add rules that would fill the other cell combinations according to the predefined tree structure.
One difference to Algorithm 7 is that no CYK algorithm is run once in every recursion step to evaluate what elements are already in the pyramid and as a consequence the algorithm is quite fast.

Algorithm 9: SplitThenFill

```
  Input: Word $w \in \Sigma^*$
  Output: Set of rules $P$
1 $P = \emptyset$; // $P \subseteq V \times (V^2 \cup \Sigma)$
2 $Sol = (P_{Sol},\ v)$; // $P_{Sol} \subseteq P$
3 $Sol = SplitThenFillRec(P,\ w,\ i_{max},\ 0)$;
4 return $P_{Sol}$;
```

Line 2: v can be any random element $v \in V$.

Algorithm 10: SplitThenFillRec

```
   Input: $P_{in} \subseteq V \times (V^2 \cup \Sigma),\ w \in \Sigma^*,\ i, j \in \mathbb{N}$
   Output: $(P,\ v)$
1  $P = P_{in}$;
2  if $i = 0$ then
3      if terminal $w_j$ not distributed yet then
4          choose $v \in V$ uniformly randomly;
5          return $(P \cup \{(v,\ w_j)\},\ v)$;
6      end
7      return $(P,\ v_{lhse})$; // see note below
8  end
9  choose one $m$ uniform randomly in $[j + 1,\ j + i]$;
10 $(P,\ v_l) = SplitThenFillRec(P,\ w,\ (m - j - 1),\ j)$;
11 $(P,\ v_r) = SplitThenFillRec(P,\ w,\ (j + i - m),\ m)$;
12 if $i = i_{max}$ then
13     return $(P \cup \{(S,\ v_l v_r)\},\ S)$;
14 end
15 choose $v \in V$ uniformly randomly;
16 return $(P \cup \{(v,\ v_l v_r)\},\ v)$;
```

Line 7: $v_{lhse} \longrightarrow w_j \in P$ at this point, as terminal w_j was already distributed.

Example: Algorithm SplitThenFill

According to this algorithm, only productions corresponding to the tree structure are added to the grammar. The same example tree structure as in Figure 11 is used here – remember that each number represents the recursion depth of its subtree.
For illustration purposes, the pyramid is shown to reflect the immediate changes of the added rules to the pyramid in the Figures 18 to 23. Note that the pyramid is actually not filled during the execution of the algorithm.

At first the rule $B \rightarrow b$ is added as shown in Figure 18.

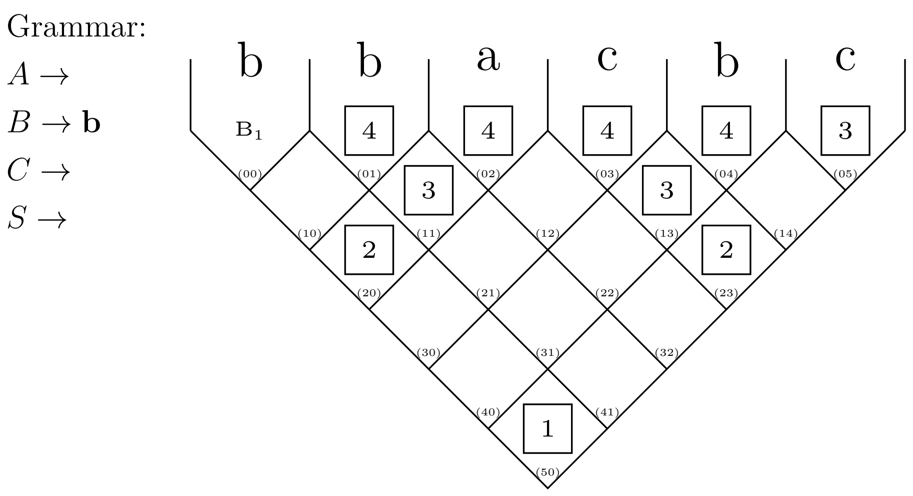


Figure 18: Illustration of Algorithm 9 SplitThenFill part 1. To resolve the recursion step that fills $Cell_{0,0}$ the rule $B \rightarrow b$ is added.

Next are the rules $A \rightarrow a$ and $S \rightarrow BA$ as seen in Figure 19. To resolve the recursion step that fills $Cell_{0,1}$ no rule is added because a rule $lhse \rightarrow b$ already exists. To fill $Cell_{0,2}$ the rule $A \rightarrow a$ and regarding $Cell_{1,1}$ the rule $S \rightarrow BA$ is added.

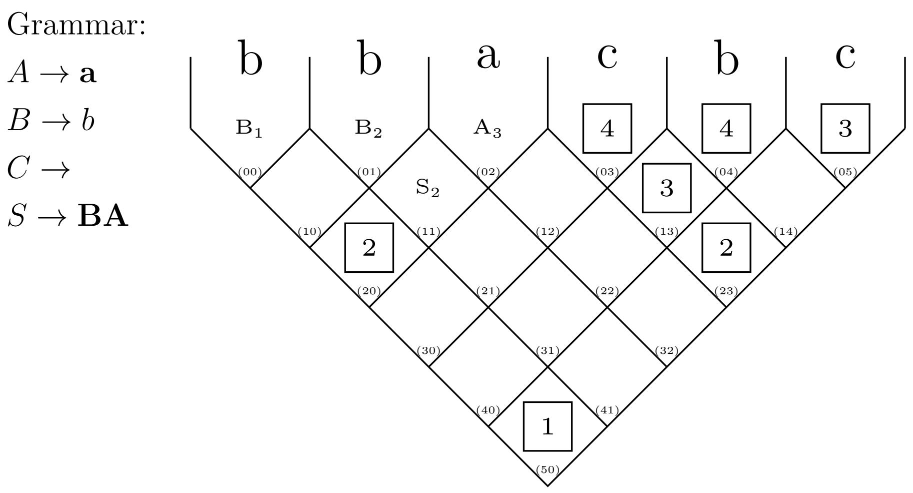


Figure 19: Illustration of Algorithm 9 SplitThenFill part 2. Adding of the rules $A \rightarrow a$ and $S \rightarrow BA$.

To fill the $Cell_{2,0}$ the rule $C \rightarrow BS$ is added, see Figure 20:

Grammar:
$A \rightarrow a$
$B \rightarrow b$
$C \rightarrow \mathbf{BS}$
$S \rightarrow BA$

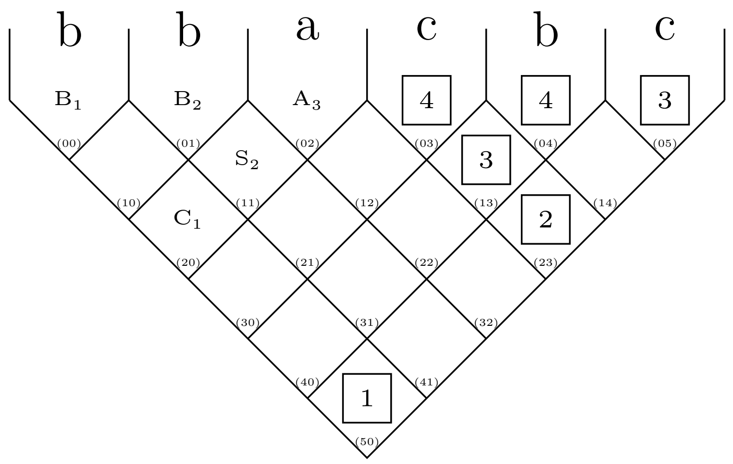


Figure 20: Illustration of Algorithm 9 SplitThenFill part 3. Adding of the rule $C \rightarrow BS$ to fill the $Cell_{2,0}$.

Analogously the other cells are filled. $Cell_{0,3}$ is responsible for the rule $C \rightarrow c$, $Cell_{0,4}$ does not cause a rule because again already the rule $B \rightarrow b$, $Cell_{1,3}$ contributes for the rule $B \rightarrow CB$, $Cell_{0,5}$ does not add a rule because of $C \rightarrow c$ and to fill $Cell_{2,3}$ the rule $A \rightarrow BC$ is added.

Grammar:
$A \rightarrow \mathbf{BC} \mid a$
$B \rightarrow \mathbf{CB} \mid b$
$C \rightarrow BS \mid \mathbf{c}$
$S \rightarrow BA$

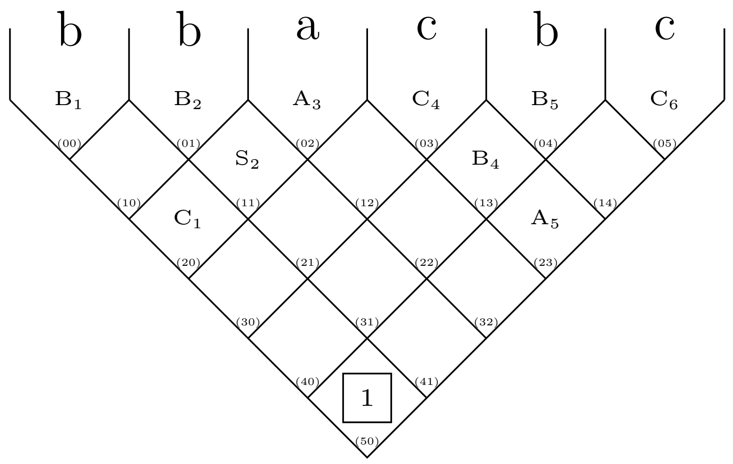


Figure 21: Illustration of Algorithm 9 SplitThenFill part 4. Adding of the rules $C \rightarrow c$, $B \rightarrow CB$ and $A \rightarrow BC$.

To fill the cell in the root a rule must be added that has the start variable as its *lhse* which guarantees $w \in L(G)$. Here the rule $S \rightarrow CA$ is added.

Grammar:
$A \rightarrow BC \mid a$
$B \rightarrow CB \mid b$
$C \rightarrow BS \mid c$
$S \rightarrow BA \mid$ **CA**

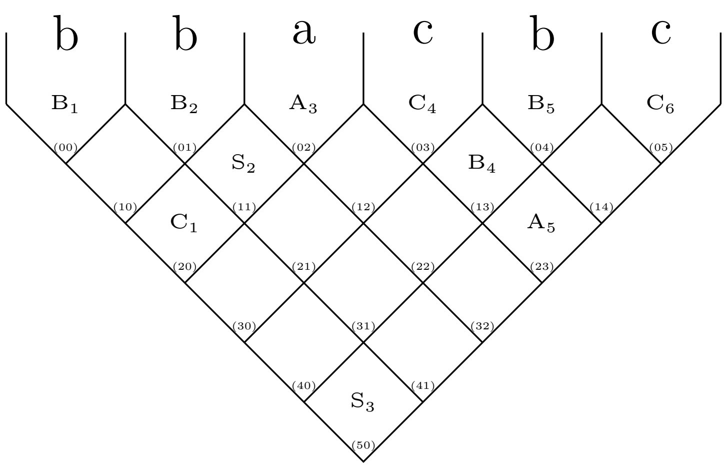


Figure 22: Illustration of Algorithm 9 SplitThenFill part 5. Adding of the rule $S \rightarrow CA$.

Finally a comparison of Figure 22 and Figure 23 shows the difference between the parsing table after the last step of the algorithm and the completely updated parsing table. Additional variables are found in the cells $Cell_{1,4}$, $Cell_{2,3}$, $Cell_{4,0}$ and $Cell_{5,0}$.

Grammar:
$A \rightarrow BC \mid a$
$B \rightarrow CB \mid b$
$C \rightarrow BS \mid c$
$S \rightarrow BA \mid CA$

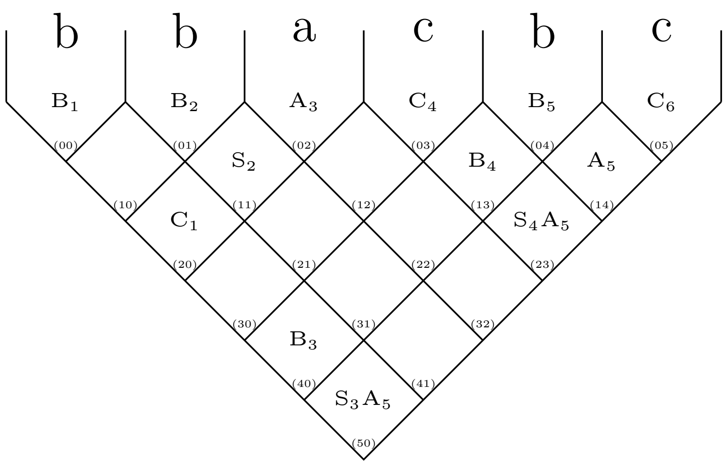


Figure 23: Illustration of Algorithm 9 SplitThenFill part 6. Comparison of the last step of the algorithm and the complete parsing table.

2.7 Evaluation of Algorithms

2.7.1 Success Rates

Until now different algorithms have been described that could be used in a application to create suitable exam exercises. But it is of interest to find out which algorithm performs the best and which algorithm should actually be used to generate the exercises. Therefore a composite *Success Rate* is defined that measures the algorithms performance for the different requirements towards an exam exercise.

Here $N \in \mathbb{N}$ is the sample size of all generated grammars while examining the algorithms. Before defining the overall Success Rate (SR) three other Success Rates set the basis for it.

Success Rate Producibility: A generated exercise contributes to the SR-Producibility if the CYK algorithm's output (Algorithm 1) is true or in other words $w \in L(G)$.
SR-Producibility $= p/N$, where p is the count of exercises that fulfil the requirement.

Success Rate Cardinality-Rules: A generated exercise contributes to the SR-Cardinality-Rules if the grammar has less than a certain count x of rules, i.e. $|P| \leq x$ of the grammar $G = (V,\ \Sigma,\ S,\ P)$.
SR-Cardinality-Rules $= cr/N$, where cr is the count of exercises that fulfil the requirement.

Success Rate Pyramid: A generated exercise contributes to the SR-Pyramid if the following conditions are met:

1. At least one cell enforces to a correct cell combination – see the description of Algorithm 11 CheckforceCombinationPerCell for more information.
2. There are less than 100 variables in the entire pyramid.
3. There are less than 3 variables in each cell of the pyramid.

SR-Pyramid $= p/N$, where p is the count of exercises that fulfil the three requirements above.

For the sake of these checks (1., 2. and 3.) the definition of a *cell* in the *pyramid* is simplified as following:
$Cell_{i,j} \subseteq \{(V,k) \mid k \in \mathbb{N}\} \longrightarrow Cell_{i,j} \subseteq V$
For further illustration see Figure 24 on the next page.

Example: Simplification of cells in a pyramid
The pyramid with the simpler cells is shown right.

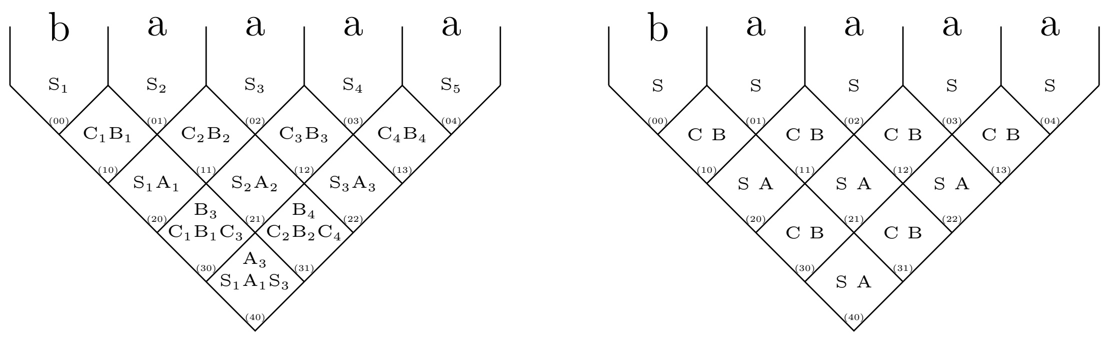


Figure 24: The simplification of cells in a pyramid.

Description of the Algorithm CheckForceCombinationPerCell:
The experience of professor Martens shows that usually most students easily find a pattern of how to fill the first two rows of the *pyramid* during the execution of the CYK algorithm, but do more mistakes starting at row $i \geq 2$. Students often do not know exactly what cell combinations need to be considered while filling one specific cell of the pyramid. They simply take only the one top left cell and the one top right cell and try to find rules in the grammar that match the resulting compound variables. The approach of only finding patterns and not thoroughly understanding the algorithm is countered by Algorithm 11 CheckForceCombinationPerCell. Here a cell forces if it is possible to see if the student has clearly understood the algorithm and not only takes the next top left cell and the next top right cell.

Algorithm 11: CheckForceCombinationPerCell

```
  Input: $CellBottom,\ CellTopLeft,\ CellTopRight \subseteq V;\ P \subseteq V \times (V^2 \cup \Sigma)$
  Output: $true \iff |VarsForcing| > 0$
1 $VarsForcing = \emptyset$; // $VarsForcing \subseteq V$
2 $VarComp = \{xy \mid x \in CellTopLeft \ \wedge \ y \in CellTopRight\}$;
3 foreach $v \in CellBottom$ do
4     $Rhses = \{rhse \mid p \in P \ \wedge \ p = (v, rhse)\}$;
5     if $Rhses \not\subseteq VarComp$ then
6         $VarsForcing = VarsForcing \cup v$;
7     end
8 end
9 return $|VarsForcing| > 0$;
```

Note: $CellBottom = Cell_{i,j}$, $CellTopLeft = Cell_{i-1,j}$ and $CellTopRight = Cell_{i-1,j-1}$.
Line 4: Get all rules of P that have v as the *lhse* and add their *rhse* to *Rhses*.
Line 5: If no $rhse \in Rhse$ can be found in VarComp, then this variables forces, concluding that this cell as a hole forces.

Example: Algorithm CheckForceCombinationPerCell
In Figure 25, the variables in $Cell_{2,0}$ and in $Cell_{2,1}$ each force a right cell combination and in both cases $VarComp = \{SS\}$. The variable $v = C$ does not have SS as one of its rhses and therefore the variable C forces. $Cell_{3,0}$ does not force because $VarComp = \{CC\}$ and the variable $v = S$ has CC as its rhse. Note again, that cells with index $i \leq 1$ can not force at all.

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

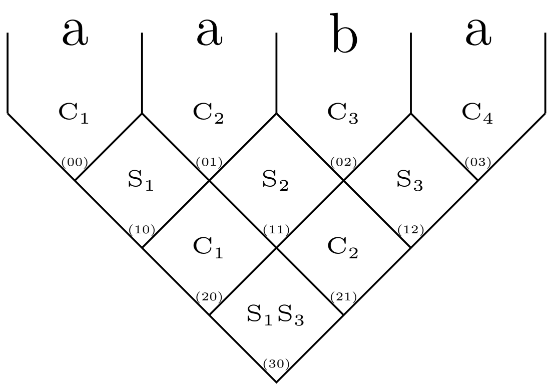


Figure 25: Application of Algorithm 11 CheckForceCombinationPerCell onto an entire pyramid.

The three success rates have been explained and finally the overall Success Rate can be specified.
Success Rate: A generated exercise contributes to the Success Rate (SR) if it contributes to the SR-Producibility, to the SR-Cardinality-Rules and to the SR-Pyramid at the same time.
It holds: $SR = n/N$, where n is the count of exercises that fulfils the three requirements.

2.7.2 Problem space exploration

Suitable ranges of parameters, to create exam exercises with, are:

- count of variables = [2; 8]
- count of terminals = [2; 8]
- wordlength = [4; 11]

This input parameter ranges are used during further comparison. Each calculated SR is based on a batch size N = 1024.

2.7.2.1 Comparison of the stopping criteria

As described in Chapter 2.1 Sub Modules, two different stopping criteria Ⓒ are used and it is of interest to know which one helps to generate more suitable exam exercises so that the better one can be used in a real world application. Therefore both variants of the stopping criteria are compared in Table 1 and Table 2.
Table 1 shows the average SRs over all configurations of parameters for each algorithm.

	MoreThanHalf	RootNotEmpty
DiceRollOnly	0.9%	**0.9%**
DiceRollVar1	1.6%	**4.3%**
DiceRollVar2	1.9%	**6.0%**
SplitThenFillCYK	4.9%	**4.9%**
SplitThenFill	29.4%	**29.5%**

Table 1: Average SRs over all the configurations of parameters for each algorithm.

In Table 2 the best possible SRs of each algorithm are compared.

	MoreThanHalf	RootNotEmpty
DiceRollOnly	17%	**17%**
DiceRollVar1	18%	**36%**
DiceRollVar2	20%	**38%**
SplitThenFillCYK	36%	**36%**
SplitThenFill	74%	**74%**

Table 2: Comparison of the best possible SRs of each algorithm. (N = 1024)

As seen in the two tables above (Table 1 and Table 2), the stopping criteria RootNotEmpty performs better or at least equally good in every case. RootNotEmpty wins through and all further discussion in the following Chapter 2.7.2.2 is done with it.

2.7.2.2 Picking the best algorithm

Now that the choice of the stopping criteria has been decided in favour of RootNotEmpty it is time to determine the one best algorithm.

As seen in Table 3 the Algorithm 9 SplitThenFill performs the best in all three sub success rates Producibility, Cardinality-Rules and Pyramid and therefore the algorithm has the best average overall SR.

Algorithm	SR	Producibility	Cardinality-Rules	Pyramid			
					Force-Right	Vars-PerCell	VarsIn-Pyramid
DiceRollOnly	**0.9%**	18.2%	10.2%	42.0%	55.3%	86.7%	98.7%
BottomUpVar1	**4.3%**	26.9%	47.1%	41.4%	68.8%	72.7%	91.1%
BottomUpVar2	**6.0%**	38.1%	30.3%	47.2%	88.6%	57.9%	90.1%
SplitThenFillCYK	**4.9%**	14.9%	50.3%	50.9%	54.8%	96.0%	99.2%
SplitThenFill	**29.5%**	100%	60.4%	51.5%	64.0%	88.2%	95.1%

Table 3: More detailed comparision of the average SRs over all the configurations of parameters for each algorithm.

While looking at Table 4, that displays the best possible SRs, Algorithm 9 SplitThenFill again performs the best.

Algorithm	SR	Producibility	Cardinality-Rules	Pyramid			
					Force-Right	Vars-PerCell	VarsIn-Pyramid
DiceRollOnly	**17%**	21%	98%	47%	47%	100%	100%
BottomUpVar1	**36%**	60%	91%	65%	69%	100%	93%
BottomUpVar2	**38%**	56%	95%	69%	69%	100%	100%
SplitThenFillCYK	**36%**	51%	97%	71%	71%	100%	99%
SplitThenFill	**74%**	100%	100%	74%	74%	100%	100%

Table 4: More detailed comparison of the best possible SRs of each algorithm. (N = 1024)

The Algorithm 9 SplitThenFill wins through against the other four as it leads to the best SR on average and also has the best possible SR of all algorithms.

Therefore the Algorithm SplitThenFill is the best choice to use in an application.

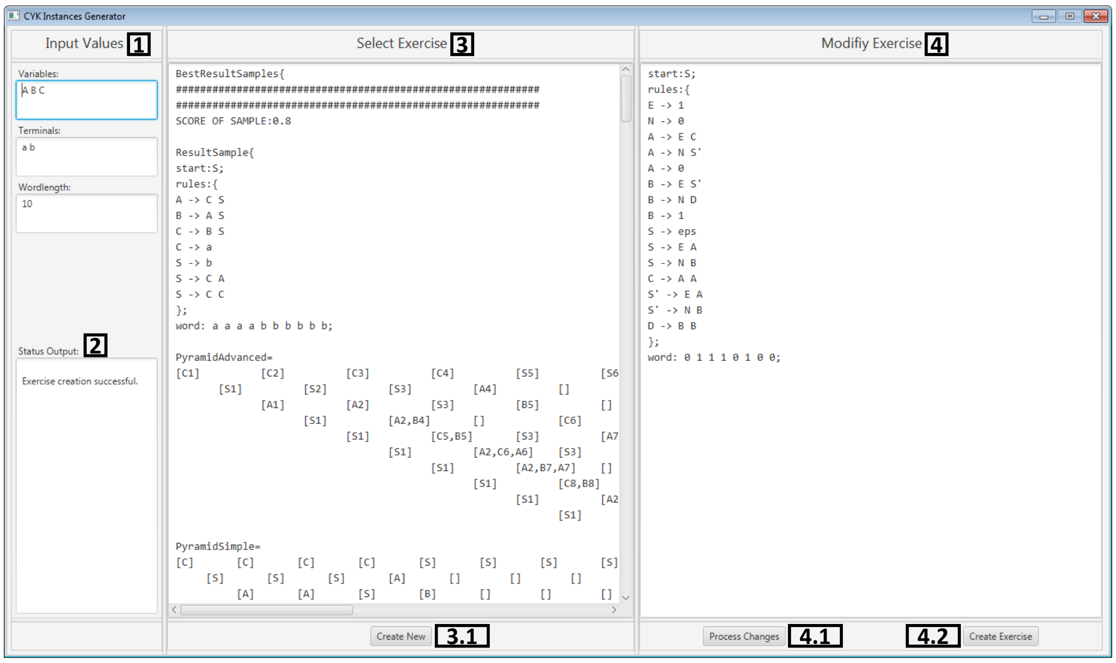
3 GUI Tool: CYK Instances Generator

One of the goals of the thesis is to get a tool that assists in creating exam exercises. A Graphical User Interface (GUI) tool is preferred over a Command Line (CL) tool because of the ease of use and the better overview during the exercise creation.

3.1 Overview GUI

The developed tool consists of four major elements as marked in Figure 26.

1. Elementary input values can be given to the programm.
2. The status output of the programm is displayed.
3. Suitable exercises are calculated automatically and one can be selected.
4. The chosen exercise can be modified as wanted.



Variables: A B C

Terminals: a b

Wordlength: 10

Status Output: Exercise creation successful.

```
BestResultSamples{
############################################################
############################################################
SCORE OF SAMPLE:0.8

ResultSample{
start:S;
rules:{
A -> C S
B -> A S
C -> B S
C -> a
S -> b
S -> C A
S -> C C
};
word: a a a a b b b b b b;

PyramidAdvanced=
[C1]          [C2]          [C3]          [C4]          [S5]          [S6
       [S1]          [S2]          [S3]          [A4]          []
              [A1]          [A2]          [S3]          [B5]          []
                     [S1]          [A2,B4]       []            [C6]
                            [S1]          [C5,B5]       [S3]          [A7
                                   [S1]          [A2,C6,A6]    [S3]
                                          [S1]          [A2,B7,A7]    []
                                                 [S1]          [C8,B8]
                                                        [S1]          [A2
                                                               [S1]


PyramidSimple=
[C]       [C]       [C]       [C]       [S]       [S]       [S]       [S]
     [S]       [S]       [S]       [A]       []        []        []
          [A]       [A]       [S]       [B]       []        []        []
```

```
start:S;
rules:{
E -> 1
N -> 0
A -> E C
A -> N S'
A -> 0
B -> E S'
B -> N D
B -> 1
S -> eps
S -> E A
S -> N B
C -> A A
S' -> E A
S' -> N B
D -> B B
};
word: 0 1 1 1 0 1 0 0;
```

Figure 26: 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 exercise is created as a LaTeX-code-file and a pdf-file. The LaTeX-file is standalone compilable and allows further modification, whereas the pdf-file shows directly what the exercise looks like.

3.1.1 Working with the program

The application structure contains only the executable "bachelor_thesis_cyk.jar"-file and the folder named "exercise". The mechanics of the application is mostly self explanatory. Just note that after clicking button 4.2 "Create Exercise" a new "exerciseLatex.tex"-file and the corresponding "exerciseLatex.pdf"-file will be generated within it and any files with the same name are overridden.

3.2 Exam Exercise

A exam *exercise* is a 4-tuple $exercise = (grammar, word, parse\,table, derivation\,tree)$.

Example: 4-tuple exercise

The pdf-file output of the tool looks similar to this:

$$
\begin{aligned}
&E \rightarrow 1\\
&N \rightarrow 0\\
&A \rightarrow EC \mid NS' \mid 0\\
&B \rightarrow ES' \mid ND \mid 1\\
&S \rightarrow EA \mid NB \mid \epsilon\\
&C \rightarrow AA\\
&S' \rightarrow EA \mid NB\\
&D \rightarrow BB
\end{aligned}
$$

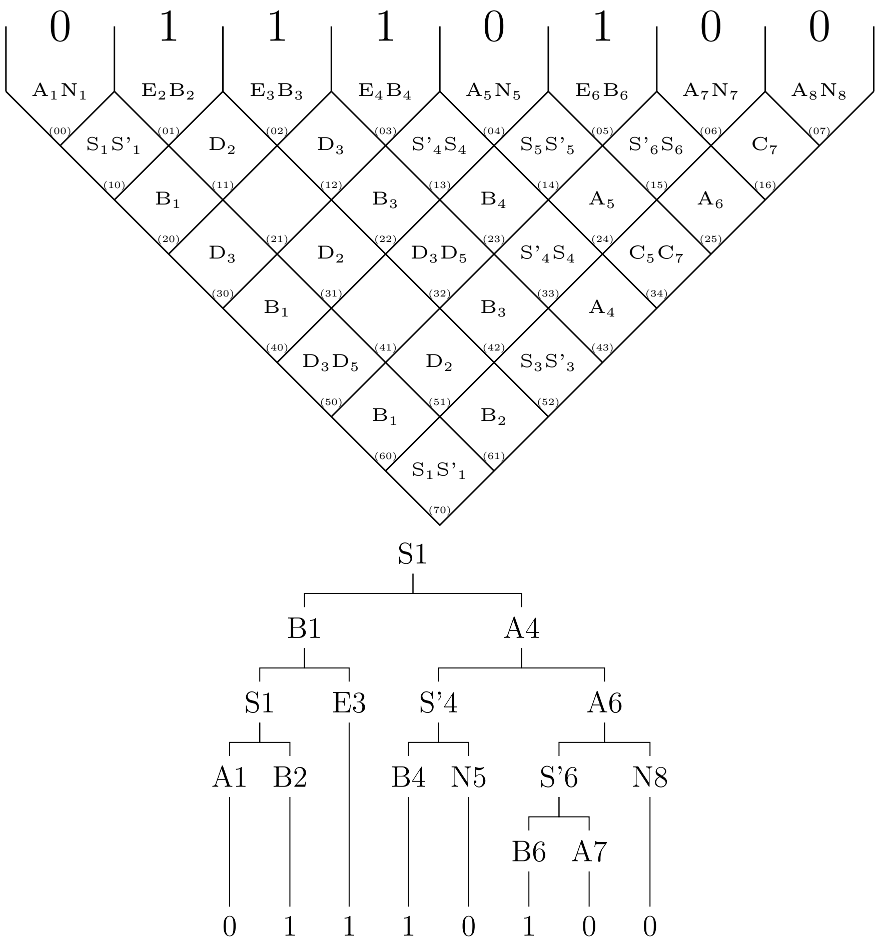


Figure 27: Example output for an *exercise*. Top: Context free grammar. Middle: A *pyramid* filled after the CYK algorithm. Bottom: Exemplary random derivation tree.

3.3 Scoring Model

Not every exercise is actually suitable for an exam. Therefore a scoring model is needed so that suitable exercise can be displayed in area 3 of the tool. Each exercise is given a score according to Table 5 and the parameters that influence the score are:

- countRightCellCombinationsForced, i.e. number of times a student is forced to make the right choice to fill the parsing table.
- sumOfVarsInPyramid, i.e. all variables in the pyramid.
- countVarsPerCell, i.e. maximum count of variables per cell.
- sumOfRules, i.e. all rules in the grammar.
- countUniqueCells, excluding row $i = 0$.

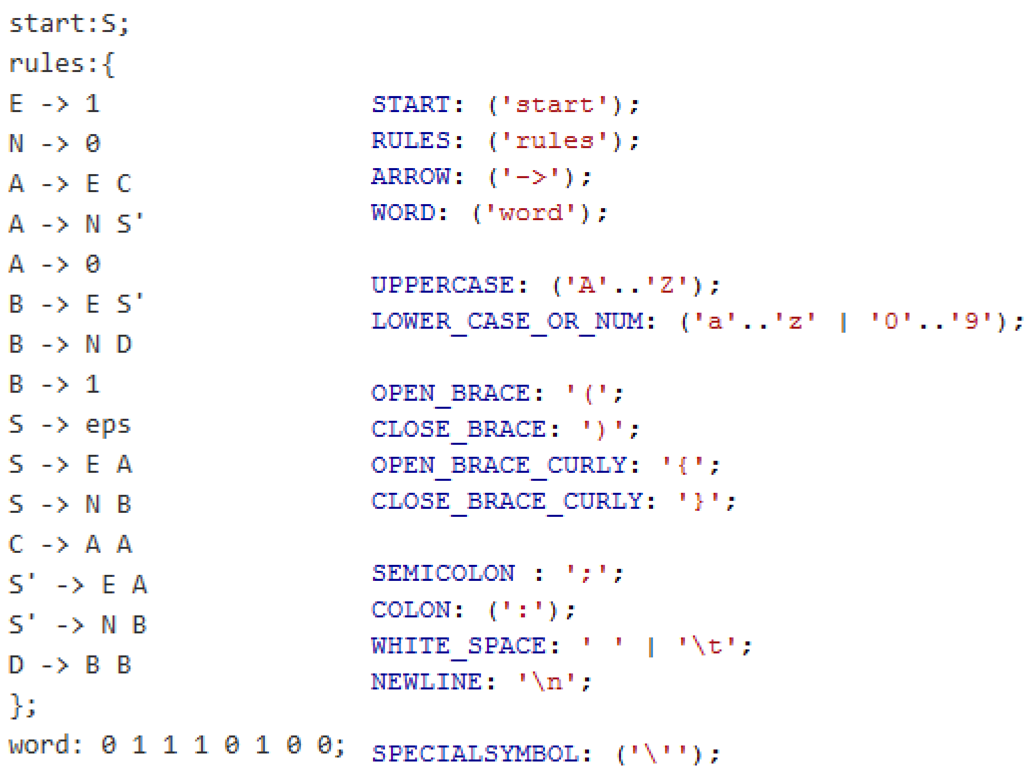
Parameter	Points					
	2	4	6	8	10	-100
#cellCombinationsForced	[0,10]	[11,20]	[21,30]	[41,50]	[31,40]	>50
sumVarsInPyramid	[0,10]	[11,20]	[21,30]	[41,50]	[31,40]	>50
#VarsPerCell	[5,5]	[4,4]	[1,1]	[3,3]	[2,2]	>5
sumOfRules	[1,2]	[3,4]	[5,6]	[9,10]	[7,8]	>10
countUniqueCells	[3,3]	[4,4]	[5,5]	[6,6]	[7,7]	$\leq$2

Table 5: Scoring of the different parameter values.

The score of each *exercise* is normalized to the maximum possible points so the maximum score is 1.0. The score is calculated as following:
$score = (\#Parameter \cdot 10)^{-1} \cdot \sum\limits_{parameter} points.$
One negative score is already sufficient to avoid examples in area 3 with undesired properties. One negative score is already sufficient that the overall score of the exercise is negative.

3.4 Parsing input with ANTLR

In area 4 of the application a context free grammar is given as input as seen in Figure 28 on the left. This input is parsed with ANTLR [4] because it allows a clear separation between the language definition and the Java code.
The first step here is the tokenization of the input in token as seen in Figure 28 on the right. After that with the help of the Grammar as seen in Figure 29 an abstract syntax tree is generated out of which Java souce code is generated.

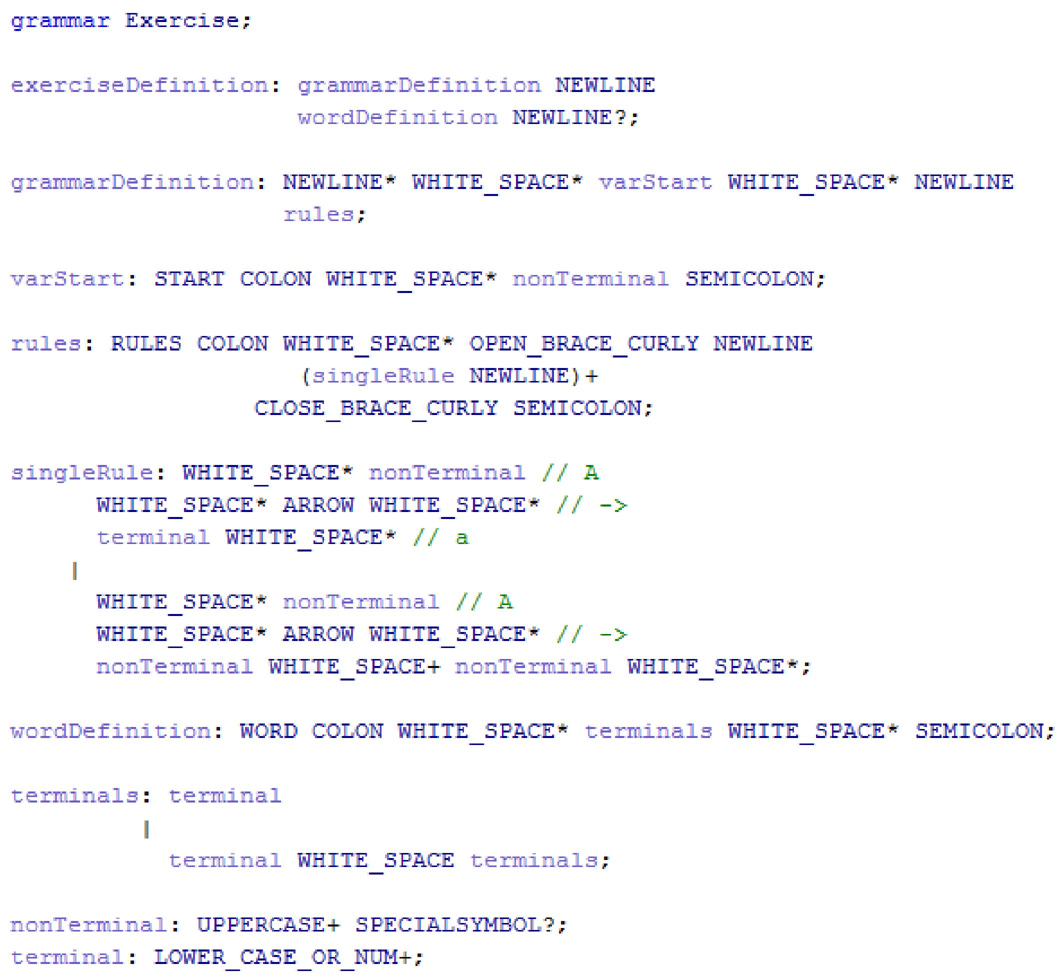


```
start:S;
rules:{
E -> 1
N -> 0
A -> E C
A -> N S'
A -> 0
B -> E S'
B -> N D
B -> 1
S -> eps
S -> E A
S -> N B
C -> A A
S' -> E A
S' -> N B
D -> B B
};
word: 0 1 1 1 0 1 0 0;
```

```
START: ('start');
RULES: ('rules');
ARROW: ('->');
WORD: ('word');

UPPERCASE: ('A'..'Z');
LOWER_CASE_OR_NUM: ('a'..'z' | '0'..'9');

OPEN_BRACE: '(';
CLOSE_BRACE: ')';
OPEN_BRACE_CURLY: '{';
CLOSE_BRACE_CURLY: '}';

SEMICOLON : ';';
COLON: (':');
WHITE_SPACE: ' ' | '\t';
NEWLINE: '\n';

SPECIALSYMBOL: ('\'');
```

Figure 28: Left: Input grammar example of the application. Right: Formal definition of the used ANTLR grammar tokens to parse the input grammar.

[4]ANTLR works with LL(k) grammars, which means that each derivation step can be distinctly identified through the next k tokens.



```
grammar Exercise;

exerciseDefinition: grammarDefinition NEWLINE
                    wordDefinition NEWLINE?;

grammarDefinition: NEWLINE* WHITE_SPACE* varStart WHITE_SPACE* NEWLINE
                   rules;

varStart: START COLON WHITE_SPACE* nonTerminal SEMICOLON;

rules: RULES COLON WHITE_SPACE* OPEN_BRACE_CURLY NEWLINE
                    (singleRule NEWLINE)+
                 CLOSE_BRACE_CURLY SEMICOLON;

singleRule: WHITE_SPACE* nonTerminal // A
      WHITE_SPACE* ARROW WHITE_SPACE* // ->
      terminal WHITE_SPACE* // a
    |
      WHITE_SPACE* nonTerminal // A
      WHITE_SPACE* ARROW WHITE_SPACE* // ->
      nonTerminal WHITE_SPACE+ nonTerminal WHITE_SPACE*;

wordDefinition: WORD COLON WHITE_SPACE* terminals WHITE_SPACE* SEMICOLON;

terminals: terminal
         |
           terminal WHITE_SPACE terminals;

nonTerminal: UPPERCASE+ SPECIALSYMBOL?;
terminal: LOWER_CASE_OR_NUM+;
```

Figure 29: Formal definition of the used ANTLR grammar rules.

3.5 Other Matters

Lastly, just some general information about the implementation are given here.
Technologies that have been used for programming are Github [5] with Sourcetree [6] for version control, Maven [7] for build management, IntelliJ [8] as the IDE, ANTLR [9] with ANTLRWorks for parsing input and JavaFX Scene Builder [10] to create the GUI.
Important used frameworks are: JUnit [11] for testing and Project Lombok [12] to greatly reduce boilerplate code.
Altogether around 7100 lines of code have been written, of which 5400 are pure java code lines, 900 are comment lines and 800 are blank lines.

[5]Github: https://github.com/
[6]Sourcetree: https://www.sourcetreeapp.com/
[7]Maven: https://maven.apache.org/
[8]IntelliJ: https://www.jetbrains.com/idea/
[9]ANTLR: http://www.antlr.org/
[10]JavaFX Scene Builder: http://gluonhq.com/products/scene-builder/
[11]JUnit: http://junit.org/junit4/
[12]Project Lombok: https://projectlombok.org/features/all

References

[1] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley-Interscience, 2. aufl. edition, 2012.

[2] Itiroo Sakai. *Syntax in universal translation*. Her Majesty's Stationery Office, London, 1962.

[3] John Cocke, Jacob T. Schwartz. *Programming languages and their compilers. Preliminary notes*. Courant Institute of Mathematical Sciences of New York University, New York, 1970.

[4] Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. 1966.

[5] Daniel Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

[6] Dirk Hoffmann. *Theoretische Informatik*. Hanser, Carl, München, 1., neu bearbeitete auflage edition, 2015.

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Unterschrift :   *Ort*, *Datum* :