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 23, 2017
1. Supervisor:	Prof. Dr. Wim Martens
2. Supervisor:	M.Sc. Tina Trautner

ABC

Abstract

Every year, lecturer in the field of theoretical computer science or an related one face the task to create an exam exercise that tests if their students have understood the way of working of the Cocke-Younger-Kasami algorithm. Various implementations and small online tools of the CYK algorithm can be found, but none actually assists during the process of creating a exercise.
Therefore various algorithms to generate specifically suitable exercises have been designed and compared through their success rates. The different approaches for these algorithms involve the uniform randomly distribution of elements and the general Bottom-Up and Top-Down parsing approaches.
A GUI tool to automatically generate these exam exercises has been implemented. Its functionality contains that input parameters such as the count of variables, the count of terminals and the size of the word can be given. Suitable exam exercises are generated and one can be chosen for further modification and creation of the final exam exercise.

Zusammenfassung

Jedes Jahr stehen Dozenten der theoretischen Informatik oder eines verwandten Bereiches vor der Aufgabe Klausuraufgaben zu erstellen, die 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 Erstellen einer Aufgabe.
Verschiedene Algorithmen wurden zuerst entworfen, um genau passende Aufgaben zu generieren und wurden anschließend auch 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 letztendlich für die Klausuraufgabenerstellung ausgewählt wird.

Contents

1 Introduction

1.1 Motivation

Every year, lecturer in the field of theoretical computer science or an related one face the task to create the 4-tuple exam *exercise* = (*grammar*, *word*, *parse table*, *derivation tree*) that tests if their students have understood the way of working of the Cocke-Younger-Kasami (CYK) algorithm. For it *exercises* need to be created which is a bit of a time consuming task that can only be done in $O(n^3)$ [1].
Various implementations and small online tools of the CYK algorithm can be found [1] [2] [3] but none actually assists during the process of creating an exercise.
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 separate 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 iff:

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

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

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

A rule consists of a left hand side element (lhse) and a right hand side element (rhse). Example: $lhse \longrightarrow rhse$ applied to $A \longrightarrow c$ and $B \longrightarrow AC$ means that A and B are a $lhse$ and c and AC are a $rhse$. Elements of V^2 are often referred to as variable compounds.
While talking about a word w or a language $L(G)$ Definition 3 holds:

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

- Word: $w = w_0 \cdot w_1 \cdot ... \cdot w_j$ *and* $w \in \Sigma^*$.
- Language: $L(G)$ over an alphabet Σ is a set of words over Σ .

Moreover in the context of talking about sets, a set is always described beginning with an upper case letter, while one specific element of a set is described beginning with a lower case letter. Example: A "*Pyramid*" is a set consisting of multiple "*Cell*"s, whereas a *Cell* is again a subset of the set of variables "V". A "*cellElement*" is one specific element of a "*Cell*". (For further explanation behind this example see chapter 1.4)

1.3 General approaches of parsing

Next, the basic approach that may help finding a good algorithm is explained informally analogous to [2]. At first, parsing is described in general and afterwards its two characteristics are explained.

Definition 4. Backward Problem = Parsing ($\mathbf{w \overset{?}{\subseteq} L(G)}$)
Input: w and a grammar G.
Output: $w \subseteq L(G) \Longrightarrow$ derivation d.

If you are given a word w and want to determine if it is element of $L(G)$, it is called parsing, which 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" [2].
Bottom-up parsing starts by recognizing the words smallest sub words before its midsize sub words, and leaving 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 productions from P, with the goal of finding a derivation of the test sentence w." [2] (The so called test sentence is synonymous to an word w.) [2].

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* starts with upper case and therefore is a set.

Definition 5. *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's 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.

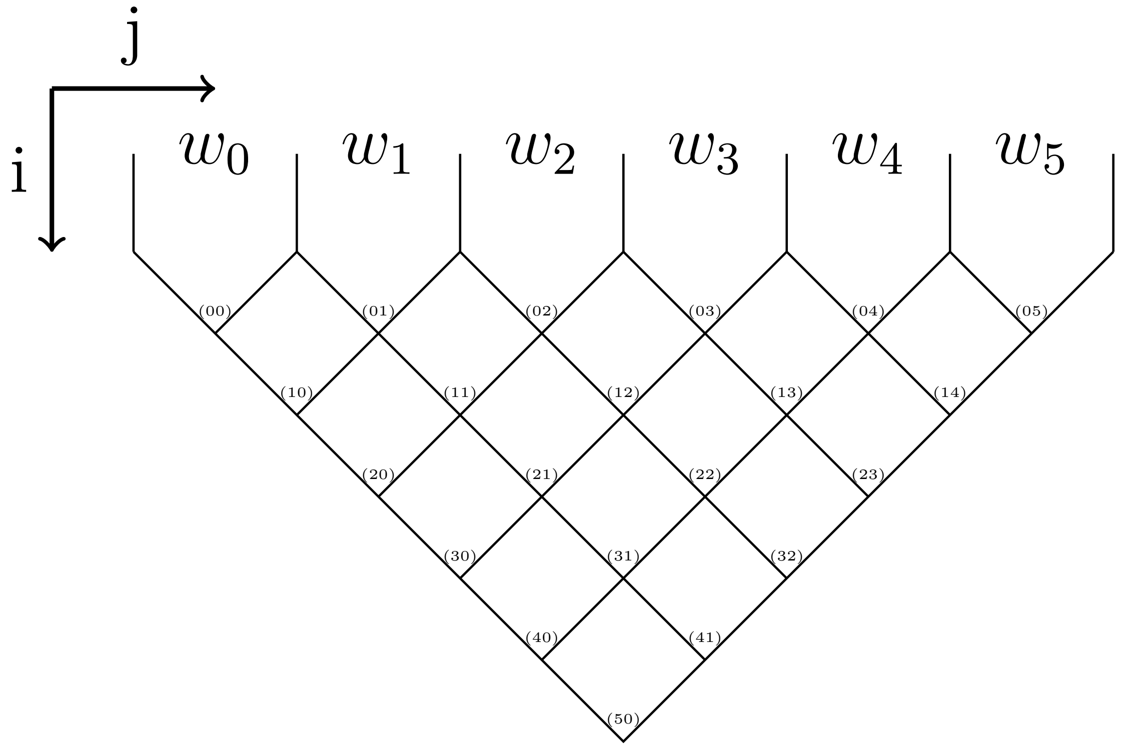


Figure 1: Visual representation of a *Pyramid* with the word w written above it.

1.5 Cocke-Younger-Kasami Algorithm

The Cocke-Younger-Kasami Algorithm (CYK) was independently developed in the 1960s by Itiroo Sakai [3], John Cocke and Jacob Schwartz [4], Tadao Kasami [5] and Daniel Younger [1].
The idea is to find all possible derivations of each subword 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      $Pyramid \cup = \{(X, j+1) \mid X \longrightarrow w_j\}$; // Fills cells $Cell_{0,j}$
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              $Pyramid\ \cup \{(X, k) \mid X \longrightarrow YZ,\ Y \in Cell_{k,j},\ Z \in Cell_{i-k-1,k+j+1}\}$;
                   // Fills cells $Cell_{i,j}$ ??k?? XXX and $Y \in Cell_{k,j}$ is wrong like
                   in circled D
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 iff $Cell_{i_{max},0}$ contains the start variable.

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 ascending 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 $X \rightarrow AC$ and because the compound variable AC is *rhse* of the variable S the $Cell_{4,1}$ contains the element $(S, 4)$.

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

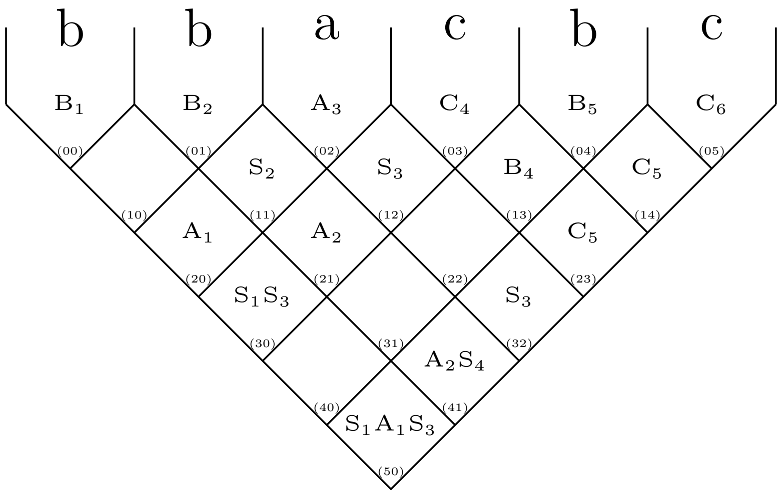


Figure 2: The CYK algorithm fills the cells of the pyramid during execution (Line 3 and Line 8).

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 way of working of algorithms in the following chapters. $\textcircled{E}$ is explained not until Chapter 2.4 because it is needed only there.

Distribute(Σ, V) $\textcircled{A}$ and Distribute(V^2, V) $\textcircled{B}$:
The difference between $\textcircled{A}$ and $\textcircled{B}$ is that one time Σ and the other time V^2 are distributed. But in both cases a uniform random subset of the *Rhse* is taken and again uniform 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 productions $P \subseteq V \times V^2$ or $P \subseteq V \times \Sigma$
foreach $rhse \in Rhse$ do
    $choose\ n\ uniformly\ randomly\ in\ [i,j]$;  // $i \in \mathbb{N},\ j \in \mathbb{N}$
    $V_{add} := uniform\ random\ subset\ of\ size\ n\ from\ V$;
    $P \cup \{(v, rhse) \mid v \in V_{add},\ rhse \in Rhse\}$;
end
return $P$;
```

Stopping Criteria $\textcircled{C}$:
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) Ⓓ:

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      $CellSet \cup \{YZ \mid X \longrightarrow YZ,\ Y \in Cell_{k,j},\ Z \in Cell_{i-k-1,k+j+1}\}$;
4  end
5  return $CellSet$;
```

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

Grammar:
$A \to AB \mid a$
$\mathbf{B} \to \mathbf{SC} \mid b$
$C \to AB$
$S \to BA \mid AA$

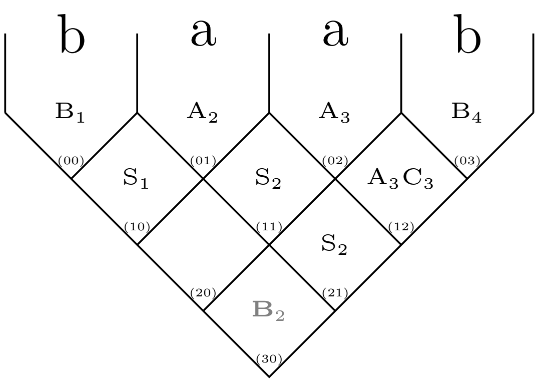


Figure 3: Example of Algorithm 3 while applying it on $Cell_{3,0}$ via adding the rule $B \to SC$.

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 $Cell_{i,j}$ won't be empty any more.

2.2 Dice rolling the distributions only

We start off by a primitive way of generating grammars, which will be 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 productions $P$
1 $P = \emptyset$; // $P \subseteq V \times (V^2 \cup \Sigma)$
2 $P = Distribute(\Sigma,\ V)$; $\textcircled{A}$
3 $P \cup Distribute(V^2,\ V)$; $\textcircled{B}$
4 return $P$;
```

The algoritm DiceRollOnly (Algorithm 4) distributes terminals Σ to at least one $lhse$, but a compound variable V^2 does not has to be distributed at all. Note that for each terminal of $\Sigma = \{a, b\}$ at least one rule like $lhse \to a$ and $lhse \to b$ is generated. But 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 $lhse \to AA$, $lhse \to BA$, $lhse \to CC$ and $lhse \to SC$ exist.

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

Figure 4: Shortend overview of an example of Algorithm 4 as described before.

2.3 Dice rolling and Bottom-Up variant one

Another approach to design an algorithm is after 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 5, the middle part of the example).

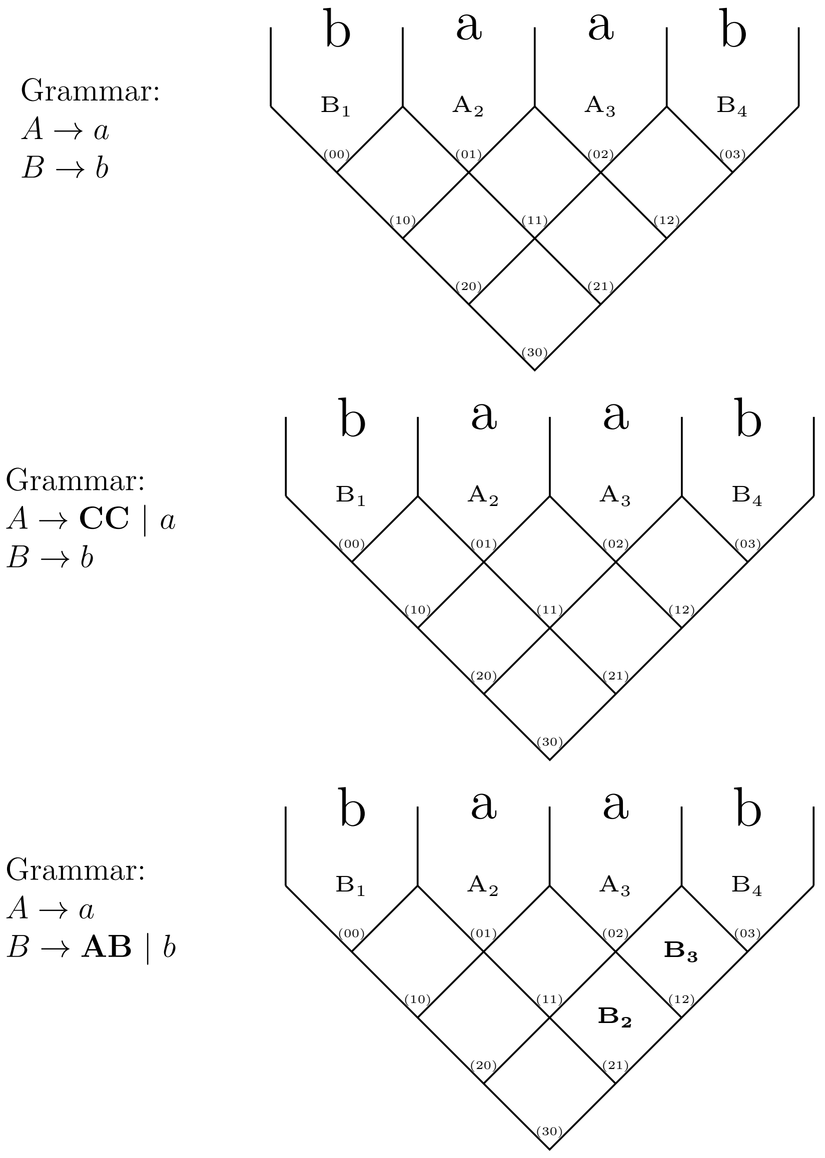


Figure 5: Example of disregarding the already added rules. Top: starting situation. Middle: Unfortunate adding of rules that doesn't help to fill the parsing table and can happen in Algorithm 4. Bottom: Good adding of rules as intended in Algorithm 5 that helps filling.

If rules like $lhse \to CC$ or $lhse \to SC$ are added they don't directly help to fill the

parsing table and bloat the grammar with useless rules (see Figure 5, the middle part of the example). More reasonable rules to add would be $lhse \to BA$, $lhse \to AA$ or $lhse \to AB$ (see Figure 5, the bottom part of the example).
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 (Line 10) variable compounds and to finally add a part of them (Line 11) in form of rules to the grammar. In 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 productions $P$
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          $choose\ one\ j \in J\ uniform\ randomly$;
9          $J = J \setminus \{j\}$;
10         $CellSet = CalculateSubsetForCell(Pyramid,\ i,\ j)$; $\textcircled{D}$
11         $P \cup Distribute(CellSet,\ V)$; $\textcircled{B}$
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 6) 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 ($i = 2$).

```
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 6: Digest of log files of Algorithm 5 with $|V| = 4$ and $|\Sigma| = 2$.

This again leads to a further improvement idea 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 Figure 5. 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).
As seen in Figure 7 the rules with BA and AA are added to the variables B and A in Grammar1. For Grammar2 instead the rule $B \rightarrow SS$ is added that contributes to a better diversity compared to Grammar1.

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 7: Example for better diversity. Starting point is Grammar0. Grammar2 is of better diversity than Grammar1.

Algorithm 6: BottomUpDiceRollVar2

```
    Input: Word $w \in \Sigma^*$
    Output: Set of productions $P$
 1  $P = \emptyset$; // $P \subseteq V \times (V^2 \cup \Sigma)$
 2  $RowSet = \emptyset$; // $RowSet \subseteq \{(xy, i) \mid x, y \in V \wedge i \in \mathbb{N}\}$
 3  $P = Distribute(\Sigma,\ V)$; $\textcircled{A}$
 4  $Pyramid = CYK(G,\ w)$ ;
 5  for $i := 1$ to $i_{max}$ do
 6      for $j := 0$ to $j_{max} - i$ do
 7          $RowSet \cup \{(xy, i) \mid xy \in CalculateSubsetForCell(Pyramid,\ i,\ j)\textcircled{D}\}$;
 8      end
 9      while $threshold_i$ not reached do
10          choose one $xy$ from $(xy,\ i) \in RowSet$ uniform randomly with
             probability depending on $i$; $\textcircled{E}$
11          $P \cup Distribute(xy,\ V)$; $\textcircled{B}$
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 4: Fills the i=0 row of the pyramid.

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

At some point a decision needs to me made about what rule $lhse \to xy$ with $xy \in V^2$ will be added to the grammar. Depending on which xy is chosen 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 $\textcircled{E}$.
The approach here only makes sense together with $\textcircled{D}$ in which all possible compound variables are calculated that help to fill one specific cell. $RowSet \subseteq \{(xy, i) \mid x, y \in V \wedge i \in \mathbb{N}\}$ whereas the xy are calculated with $\textcircled{D}$ and i is the row number of the specific cell.

With this $RowSet$ the choice can be influenced regarding 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 though 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 the example in Figure 18 this results in $xy = BC$.

```
$RowSet = \{(AB,3),(AB,1),(AB,5),...\}$                          // 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 8: Shortened example of the procedure E as before in the text.

2.5 Split Top-Down and fill Bottom-Up

Until now we have only discussed algorithms that purely use the Bottom-Up approach, so 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) and then to uniformly randomly generate a predefined structure of the derivation tree (Line 4 of Algorithm 7 and in general Algorithm 8 SplitThenFillRec) 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:

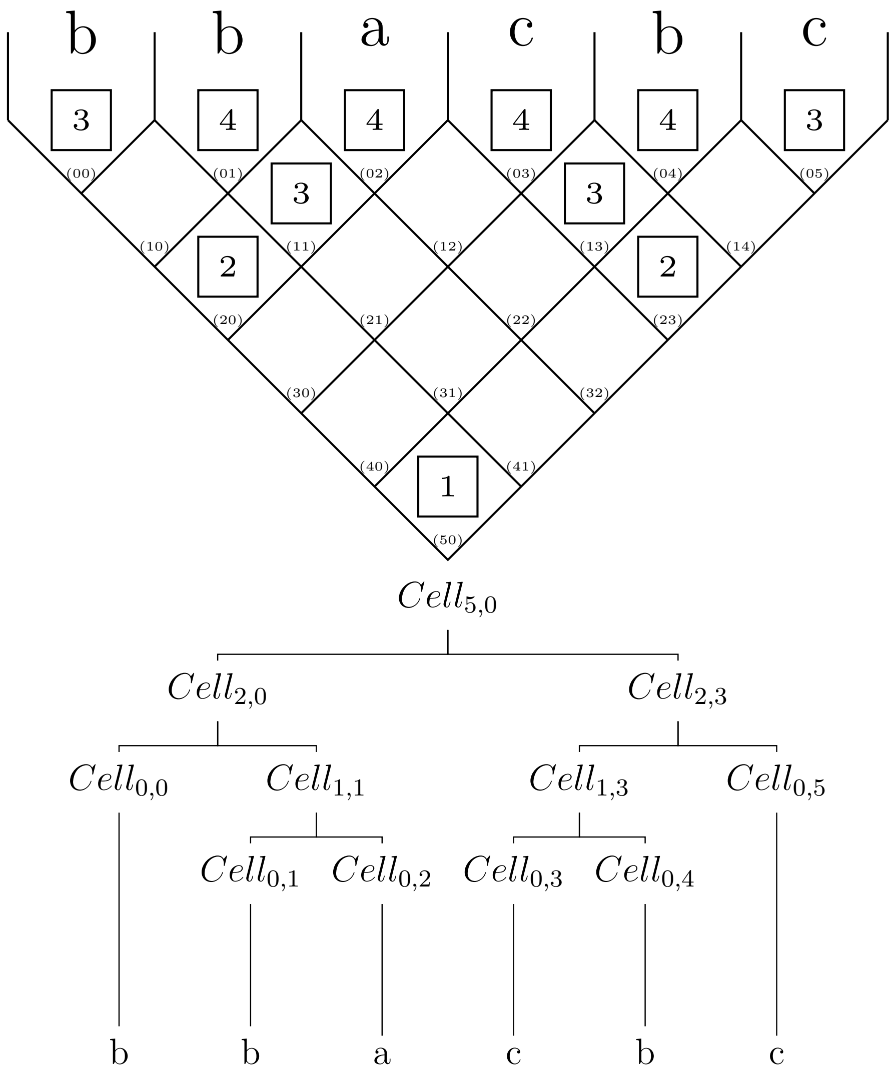


Figure 9: Example derivation tree structure. Top: Shown in the pyramid, the numbers correspond to the depth in the tree. Down: Shown as a derivation tree.

As the name of the algorithms implies only after completely generating the structure of the derivation tree (splitting of the word in subwords) the rules are added to the grammar that help filling the cells occurring in the derivation tree.
Now every time before adding a new rule (Algorithm 8 SplitThenFillRec 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 SplitThen-FillRec).

Algorithm 7: SplitThenFill

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

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

For this algorithm it is important to mention that while using Ⓑ (Line 15 of Algorithm 8) a variable compound is added to at least one *lhse*. For every element of $vc \in VarComp$ (Line 14 of Algorithm 8) there exists at least one rule $lhse \rightarrow vc$.

Algorithm 8: SplitThenFillRec

```
   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 $m$ uniform randomly in $[j+1,\ j+i]$;
 6 $(P,\ Cell_l) = SplitThenFillRec(P,\ w,\ (m-j-1),\ j)$;
 7 $(P,\ Cell_r) = SplitThenFillRec(P,\ w,\ (j+i-m),\ m)$;
 8 $Pyramid = CYK(G,\ w)$;
 9 if stopping criteria met Ⓒ then
10     return $(P,\ Cell_{i,j})$;
11 end
12 if $Cell_{i,j} = \emptyset$ then
13     $VarComp =$ uniform random subset from $\{vc \mid v \in Cell_l \wedge$
14         $c \in Cell_r\}$ with $|VarComp| \geq 1$;
15     $P \cup Distribute(VarComp,\ V)$; Ⓑ
16 end
17 return $(P,\ Cell_{i,j})$;
```

The same example tree structure as in Figure 2.5 is used in the following example – each number represents the recursion depth of its subtree:

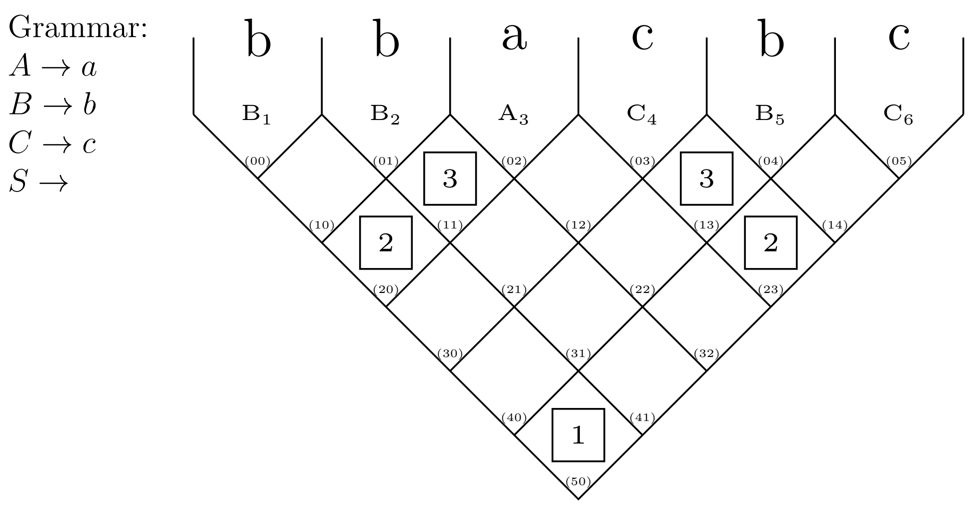


Figure 10: Illustration of Algorithm 7 SplitThenFill part 1 after adding $A \to a$, $B \to b$ and $C \to c$.

After adding the terminals to the grammar (Line 2 in Algorithm 7 SplitThenFill) now 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 $S \to BA$ leads to the following *Pyramid*:

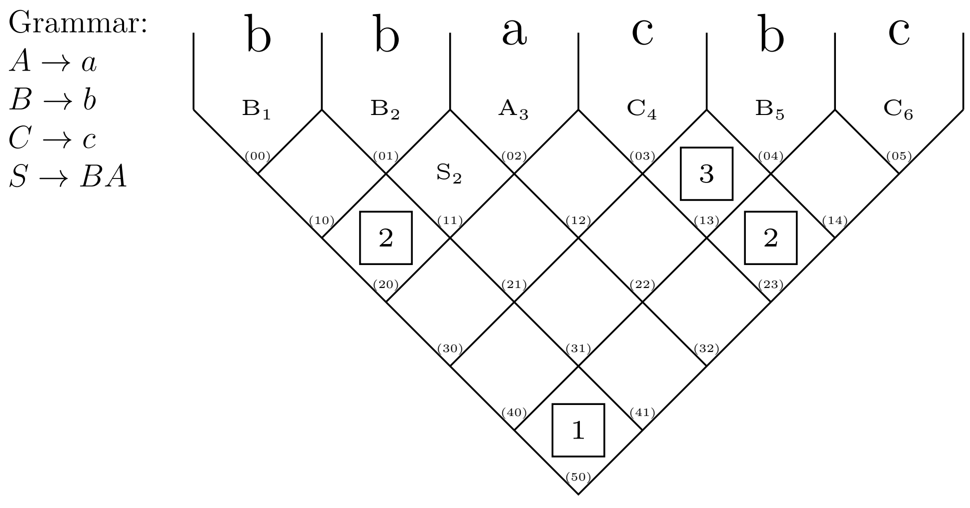


Figure 11: Illustration of Algorithm 7 SplitThenFill part 2 after adding $S \to BA$.

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:

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

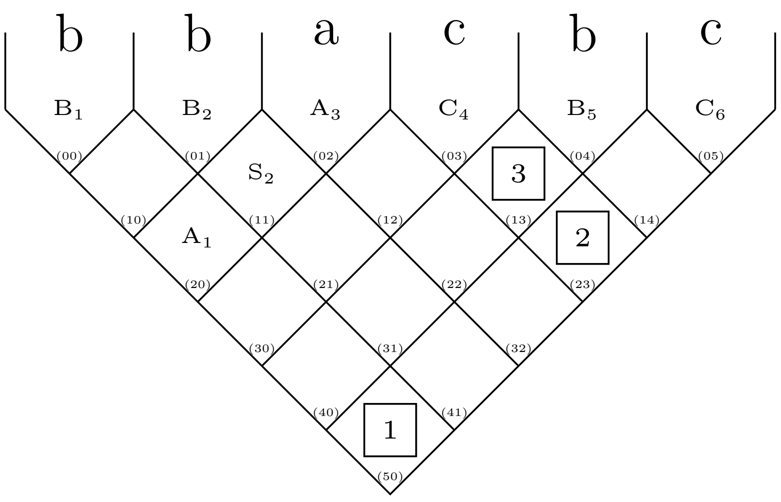


Figure 12: Illustration of Algorithm 7 SplitThenFill part 3 after adding the rule $A \to BS$.

The next two analogous steps are described in Figure 13 and in Figure 14.

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

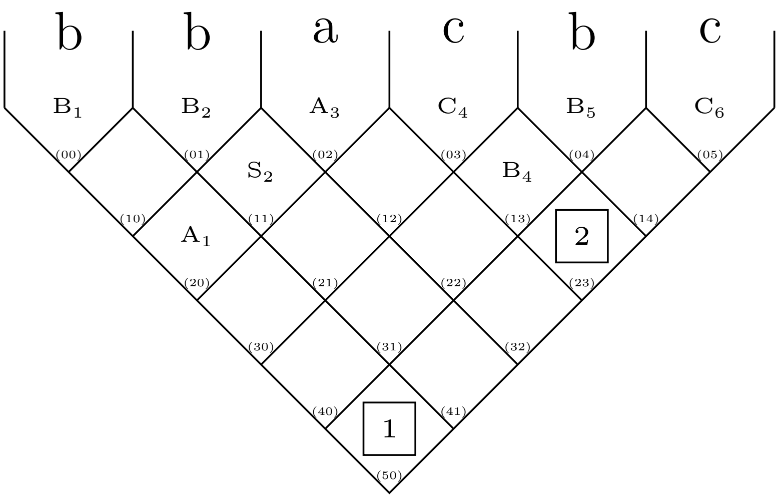


Figure 13: Illustration of Algorithm 7 SplitThenFill part 4. The recursion step in $Cell_{1,3}$ is resolved by adding the rule $B \to CB$.

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

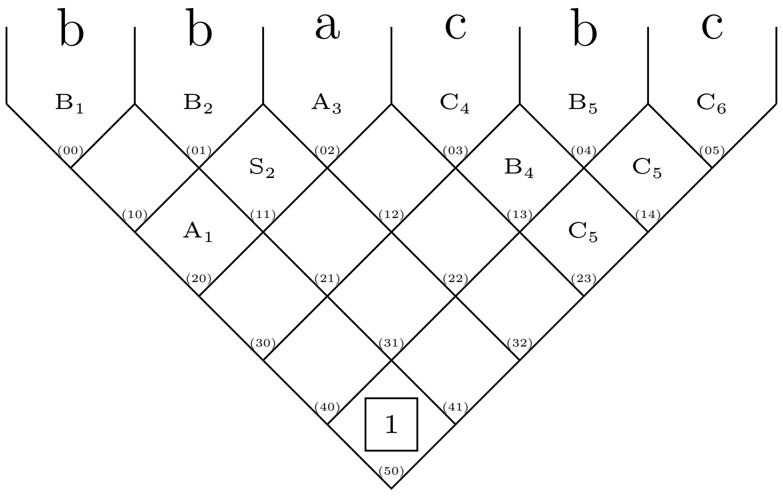


Figure 14: Illustration of Algorithm 7 SplitThenFill part 5. The recursion step in $Cell_{2,3}$ is resolved by adding the rule $C \to BC$.

Finally the last recursion step that decides on the content of the root cell is shown in Figure 15:

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

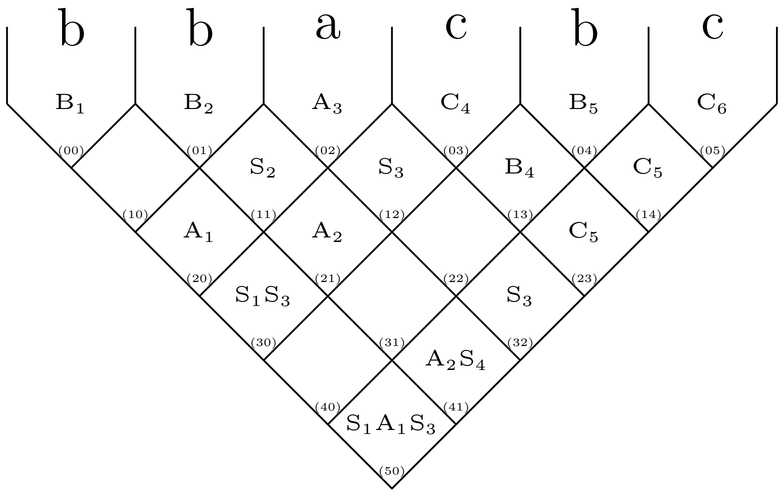


Figure 15: Illustration of Algorithm 7 SplitThenFill part 6. The recursion step in $Cell_{5,0}$ is resolved by adding the rule $S \to AC$.

2.6 Split Top-Down and fill Top-Down

After defining an algorithm that uses the combination of the Bottom-Up approach and the Top-Down approach (Algorithm 7) a step further is to find an algorithm that only makes use of the Top-Down approach to see if an even better algorithm can be found. This algorithm again generates a predefined structure of the derivation tree Top-Downwards. Every time a node of the structure of the derivation tree has been decided on, a rule is immediately 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 a terminal is distributed to only one variable. While resolving the last recursion step (Line 12) of Algorithm 10 SplitAndFillRec the start variable will be in the root of the pyramid that always leads to $w \in L(G)$.

Algorithm 9: SplitAndFill

```
  Input: Word $w \in \Sigma^*$
  Output: 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 $Sol = SplitAndFillRec(P,\ w,\ i_{max},\ 0)$;
4 return $P_{Sol}$;
```

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

Algorithm 10: SplitAndFillRec

```
   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         return $(P \cup \{(v,\ w_j)\},\ v_{lhse})$;
 5     end
 6     return $(P,\ v_{lhse})$;
 7 end
 8 choose one $m$ uniform randomly in $[j+1,\ j+i]$;
 9 $(P,\ v_l) = SplitAndFillRec(P,\ w,\ (m-j-1),\ j)$;
10 $(P,\ v_r) = SplitAndFillRec(P,\ w,\ (j+i-m),\ m)$;
11 if $i = i_{max}$ then
12     return $(P \cup \{(S,\ v_l v_r)\},\ S)$;
13 end
14 return $(P \cup \{(v,\ v_l v_r)\},\ v)$;
```

Line 4 and Line 6: There is the rule $v_{lhse} \to w_j$, then v_{lhse} is the variable on the left side of the one rule that has the terminal w_j as its rhse. Line 4 and Line 14: v is a random element $v \in V$.

According to this algorithm, only productions corresponding to the tree structure are added to the grammar. For illustration purposes, the pyramid is also shown to reflect the immediate changes of the added rules to the pyramid in the Figures 16 to 21. Again the predefined derivation tree structure of Figure 2.5 is used.

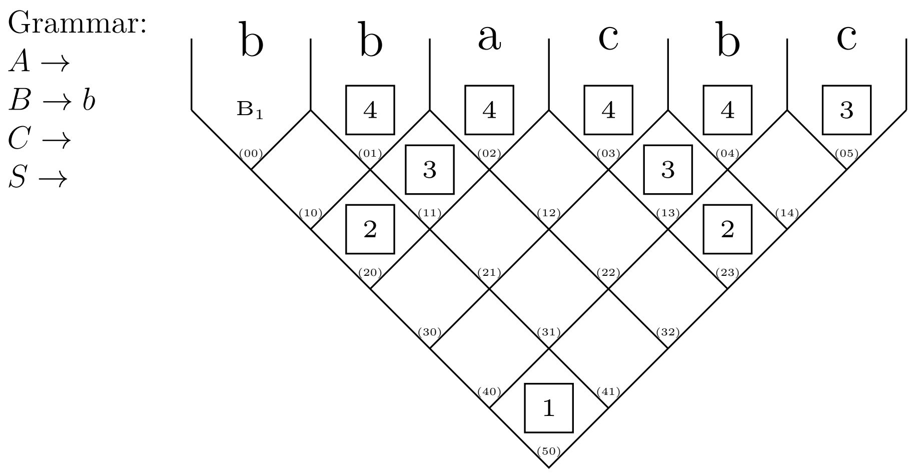


Figure 16: Illustration of Algorithm 9 part 1. To resolve the recursion step that fills $Cell_{0,0}$ the rule $B \to b$ is added.

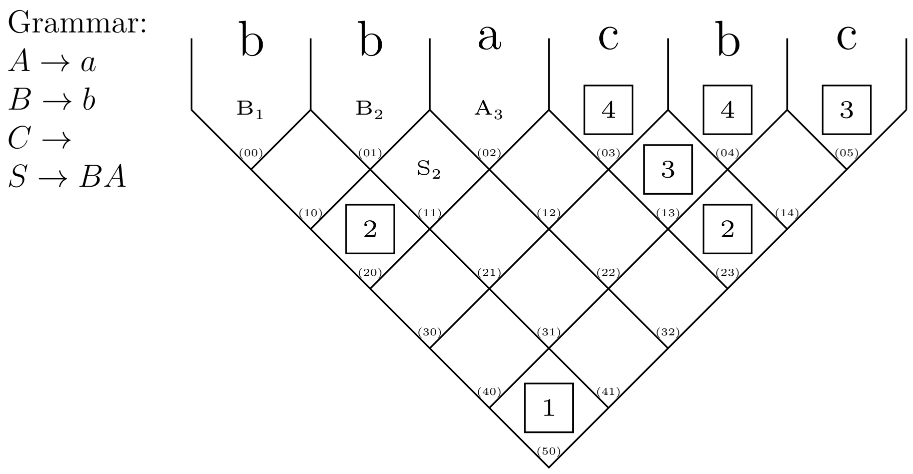


Figure 17: Illustration of Algorithm 9 part 2. Resolving the recursion step that fills $Cell_{0,1}$ no rule is added because a rule $lhse \to b$ already exists. To fill $Cell_{0,2}$ the rule $A \to a$ is added. Regarding $Cell_{1,1}$ the rule $S \to BA$ is added.

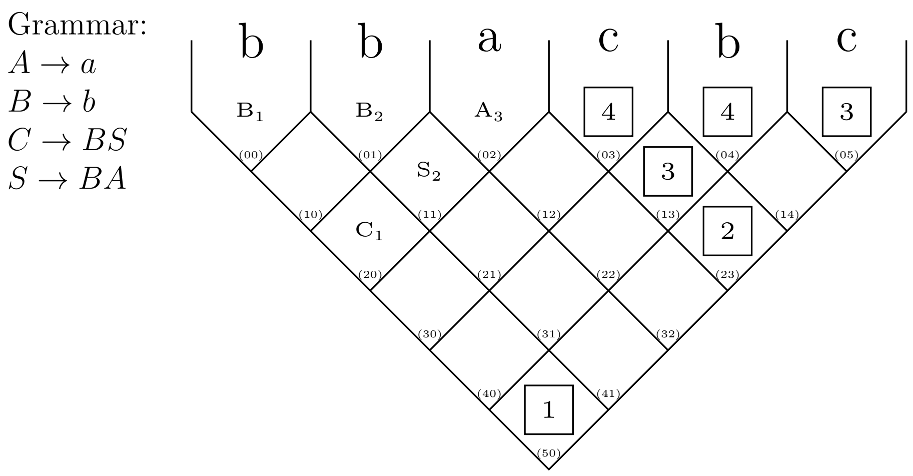


Figure 18: Illustration of Algorithm 9 part 3. Filling the $Cell_{2,0}$ the rule $C \to BS$ is added.

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

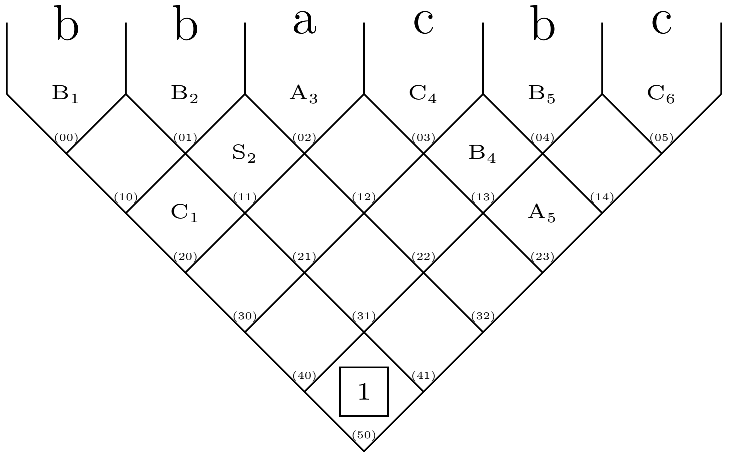


Figure 19: Illustration of Algorithm 9 part 4. Analogously the other cells are filled. $Cell_{0,3}$ is responsible for the rule $C \to c$, $Cell_{0,4}$ doesn't cause a rule because again there already is the rule $B \to b$, $Cell_{1,3}$ contributes for the rule $B \to CB$, $Cell_{0,5}$ does not add a rule because of $C \to c$ and to fill $Cell_{2,3}$ the rule $A \to BC$ is added.

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

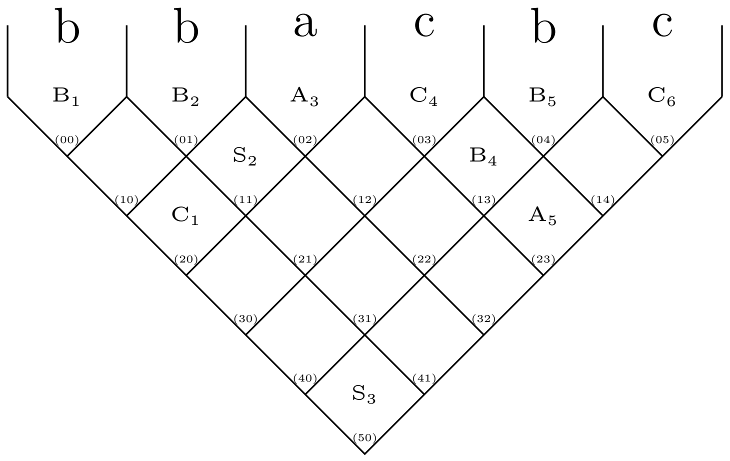


Figure 20: Illustration of Algorithm 9 part 5. Finally, to fill the cell in the root a rule must be added that has the start variable as its *lhse* that guarantees $w \in L(G)$. Here the rule $S \to CA$ is added.

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

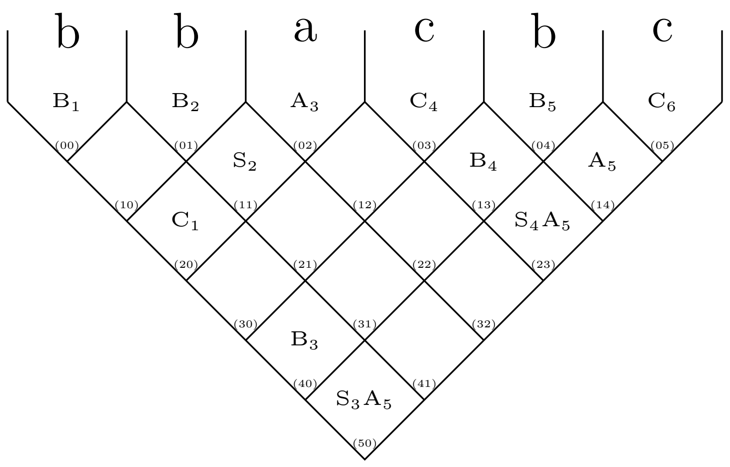


Figure 21: Illustration of Algorithm 9 part 6. With part 5 of the example the algorithm is finished. In comparison to Figure 20 the complete parsing table looks like above.

2.7 Evaluation of Algorithms

2.7.1 Success Rates

Different algorithms have been described that could be used in the application to create suitable exam *exercises*. Now it is of interest to find out which algorithm performs the best and which algorithm should actually be used in the application. Therefore a composite *Success Rate* has been 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 *exercise*s that fulfil the requirement.

Success Rate Cardinality-Rules: A generated *exercise* contributes to the SR-Cardinality-Rules if the grammar has got 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 *exercise*s.

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

1. At least one cell enforces to do 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 *exercise*s that fulfil the three requirements above.

While checking 1., 2. and 3. a simplification of $Cell_{i,j}$ is done:
$Cell_{i,j} \subseteq \{(V,k) \mid k \in \mathbb{N}\} \longrightarrow Cell_{i,j} \subseteq V$, for illustration see Figure 22.

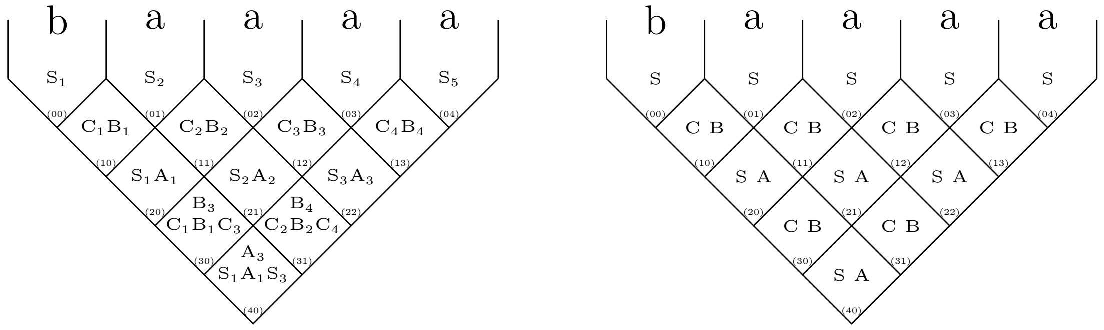


Figure 22: The simplification of cells in a pyramid.

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 $i \geq 2$. The approach of only finding patterns and not thoroughly understanding the algorithm is countered by Algorithm 11 CheckForceCombinationPerCell – it is called forcing. Students often don't 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.

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 $v \in CellDown$ 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: $CellBottom = Cell_{i,j}$, $CellUpperLeft = Cell_{i-1,j}$ and $CellUpperRight = Cell_{i-1,j-1}$
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.

In the example in Figure 23, 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 $v = C$ doesn't 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.

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

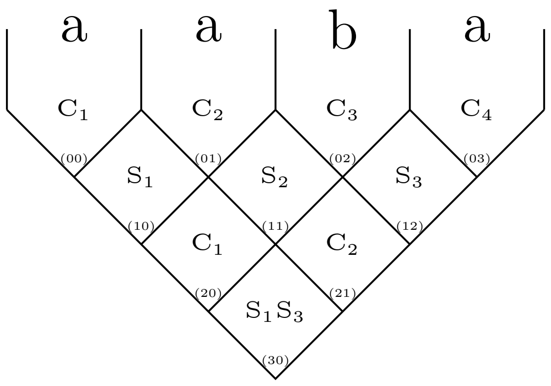


Figure 23: Application of Algorithm 11 CheckForceCombinationPerCell onto an entire pyramid.

With the help of the three now known Success Rates 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*.

2.7.2 Problem space exploration

The application allows input values to be given for the creation of new suitable *exercises* and typical ranges for this values are as following:

Input Values of the program:

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

For every algorithm and for every possible configuration of these input values the SRs are calculated to see which configuration leads to the best performance of an algorithm. The best ones of each will be used as a representative for further examination.

2.7.2.1 Comparison of stopping criteria

As described in Chapter 2.1 Sub Modules, two different stopping criteria $\textcircled{C}$ are used and it is of interest to know which one helps to generate more suitable exam *exercise*s so that the better one can be used in the application. Therefore both variants of the stopping criteria are compared in Table 1.

	MoreThanHalf	RootNotEmpty
DiceRollOnly	17%	**18%**
DiceRollVar1	18%	**35%**
DiceRollVar2	20%	**37%**
SplitThenFill	36%	**36%**
SplitAndFill	74%	**74%**

Table 1: Comparision of the two stopping criteria: Half of the cells in the pyramid are not empty and at least one variable is in the root of the pyramid. (N = 1024)

As seen in the table above, the stopping criteria RootNotEmpty performs better or equally good for every algorithm. RootNotEmpty wins and all further discussion in the following Chapter 2.7.2.2 is done only with it.
Finally a closer look onto the actual input values is given in the following Table 2. It might be interesting to see which input values of the algorithms are responsible for the good SR.

	SR	count of variables	count of terminals	size of word
DiceRollOnly	**18%**	3	3	9
DiceRollVar1	**35%**	3	3	8
DiceRollVar2	**37%**	3	3	8
SplitThenFill	**36%**	3	4	9
SplitAndFill	**74%**	3	3	8

Table 2: Comparison of the input values of each algorithm with its best SR. (N = 1024)

Mostly in every case the count of variables and the count of terminals is three and the size of the word ranges from 8 to 9. No further discussion is done, as this only intends to display the actual values.

2.7.2.2 Comparison of the algorithms

By comparing the five algorithms in Table 3 in more detail it is clearly seen that "very very good" SR are achieved. VarsInPyramid and VarsPerCell are in most cases fulfilled. ForceRight is kind of not always fulfilled. Why do I have to discuss this....

Algorithm	SR	Producibility	Cardinality-Rules	Pyramid			
					Force-Right	Vars-PerCell	VarsIn-Pyramid
DiceRollOnly	**18%**	22%	98%	48%	48%	100%	100%
BottomUpVar1	**35%**	55%	92%	66%	66%	100%	100%
BottomUpVar2	**37%**	56%	96%	69%	69%	100%	100%
SplitThenFill	**36%**	50%	93%	75%	76%	100%	99%
SplitAndFill	**74%**	100%	100%	74%	74%	100%	100%

Table 3: Comparison of the five algorithms with their best SR in more detail. (N = 1024)

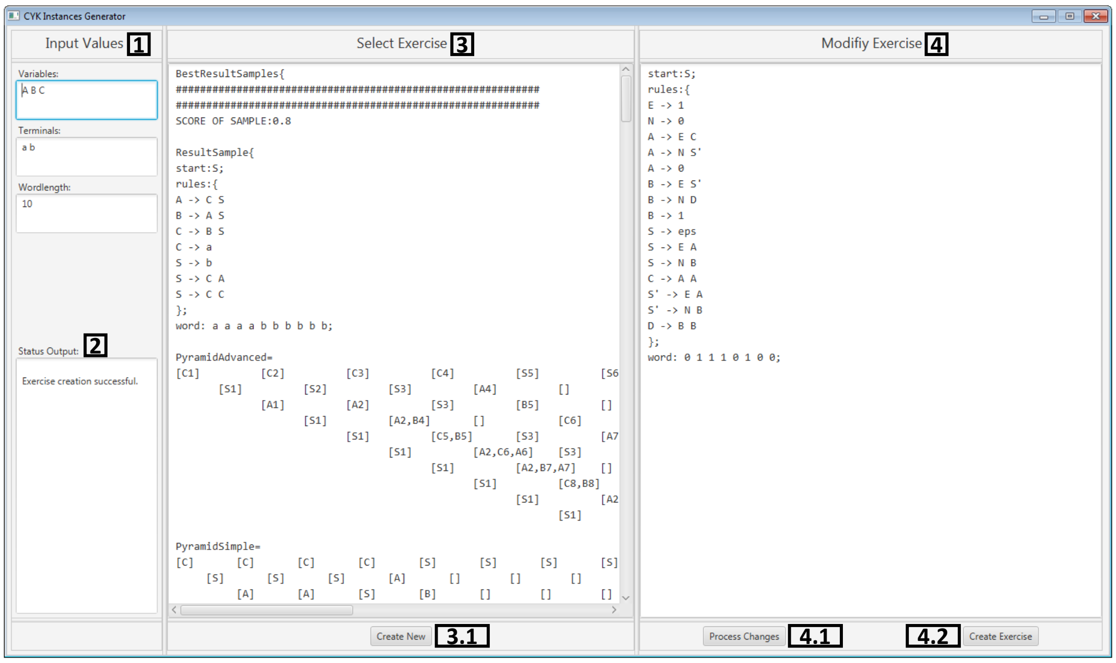
3 GUI Tool: CYK Instances Generator

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 24.

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 24: 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 Exercises

A exam *exercise* is a 4-tuple $exercise = (grammar, word, parse\,table, derivation\,tree)$. The pdf-file output of the tool looks similar to this:

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

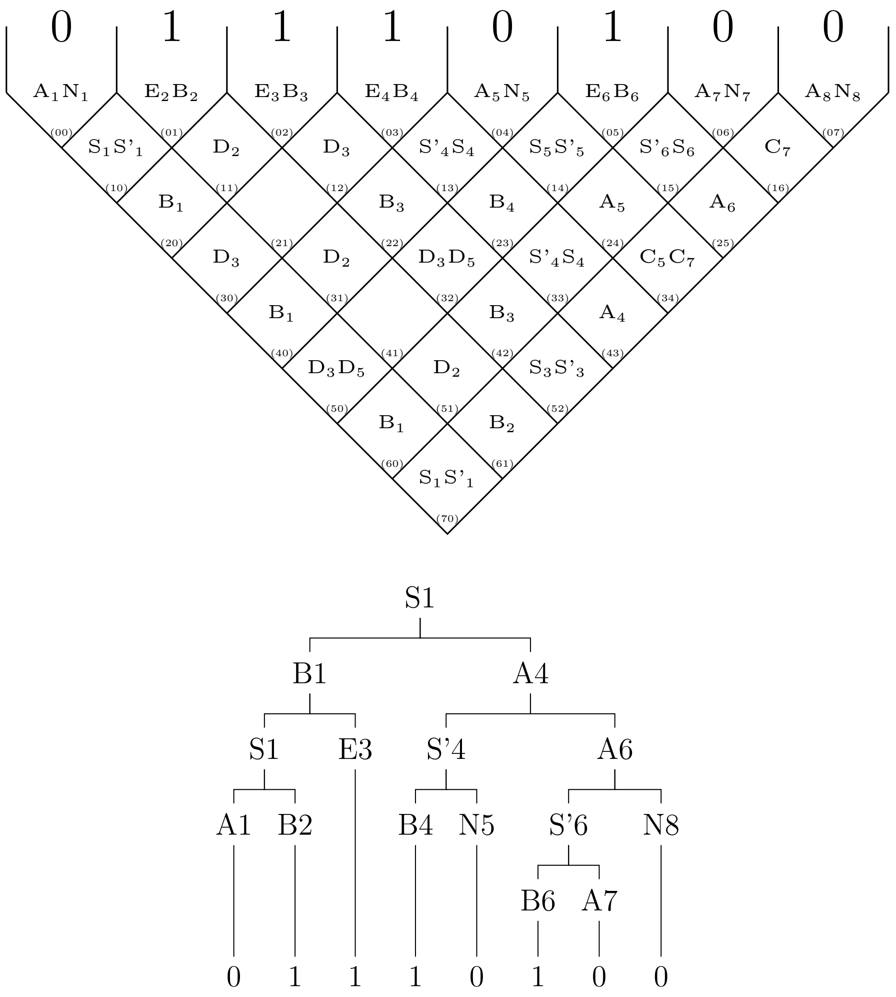


Figure 25: Example output for a *exercise*. Top: Context free grammar. Middle: A *pyramid* filled after the CYK algorithm. Bottom: Exemplary random derivation tree.

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 4 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 $i = 0$.

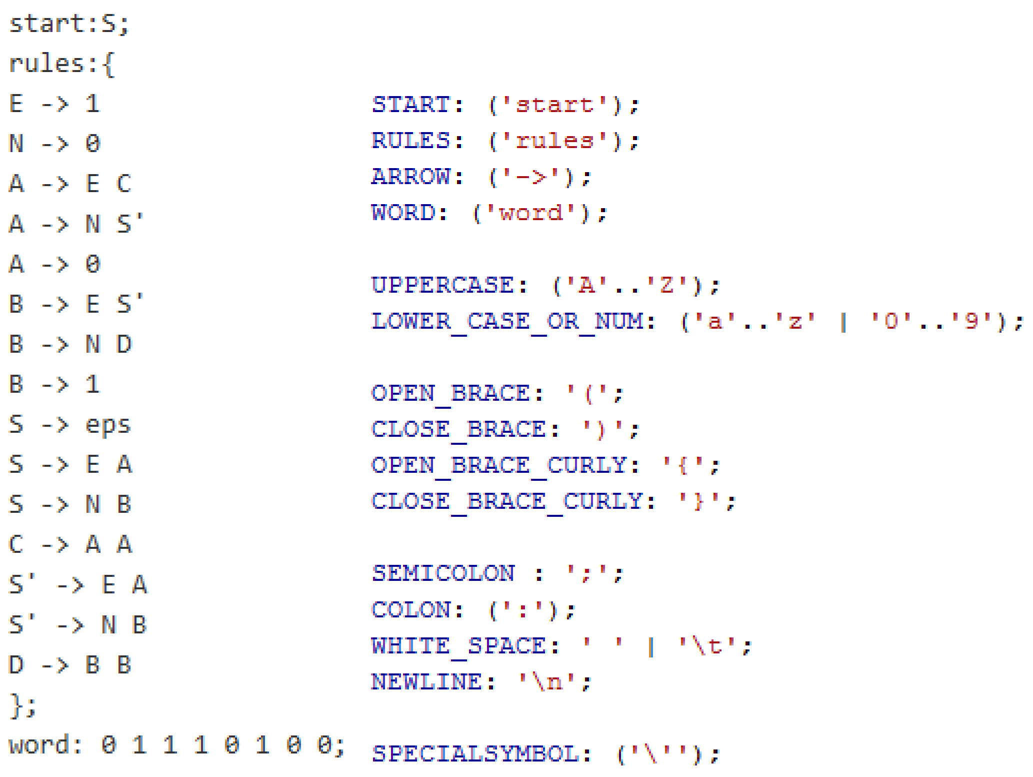
Parameter	Points					
	2	4	6	8	10	-100
#cellCombinationsForced	[0,10]	[11,20]	[21,30]	[41,50]	[31,40]	>50
sumVarsInPyramid	[0,10]	[11,20]	[21,30]	[41,50]	[31,40]	>50
#VarsPerCell	[5,5]	[4,4]	[1,1]	[3,3]	[2,2]	>5
sumOfRules	[1,2]	[3,4]	[5,6]	[9,10]	[7,8]	>10
countUniqueCells	[3,3]	[4,4]	[5,5]	[6,6]	[7,7]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

Table 4: 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: $score = (\#Parameter \cdot 10)^{-1} \cdot \sum_{parameter} points.$
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 26 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 26 on the right. After that with the help of the Grammar as seen in Figure 27 an abstract syntax tree is generated out of which intern Java objects can be parsed.

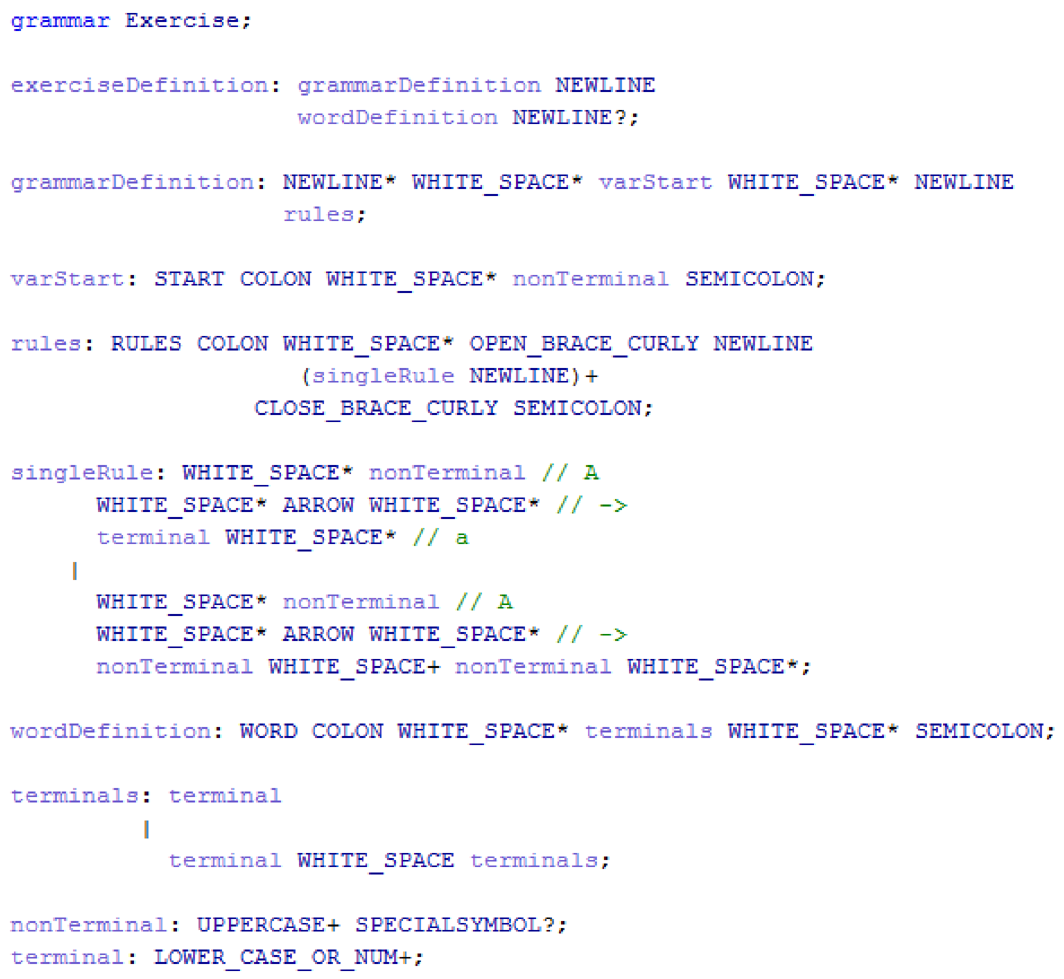


```
start:S;
rules:{
E -> 1
N -> 0
A -> E C
A -> N S'
A -> 0
B -> E S'
B -> N D
B -> 1
S -> eps
S -> E A
S -> N B
C -> A A
S' -> E A
S' -> N B
D -> B B
};
word: 0 1 1 1 0 1 0 0;
```

```
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 26: 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 27: 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

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und noch nicht veröffentlicht.

Ort, Datum Unterschrift

References

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

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

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

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

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

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