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 18, 2017

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

ABC

Abstract

The abstract of this thesis will be found here.

Zusammenfassung

Hier steht die Zusammenfassung dieser Bachelorarbeit.

Contents

1 Introduction

1.1 Motivation

The starting point of this thesis is to get a tool to automatically generate a suitable 4-tuple $exercise = (grammar,\ word,\ parse\ table,\ derivation\ tree)$, that is used to test if the students have properly understood how the Cocke-Younger-Kasami (CYK) algorithm works.
Various implementations and small online tools of the CYK algorithm can be found [1] [2] [3]. Nevertheless it is required to automatically generate suitable *exercise*s, that afterwards can be modified as wanted. This is the reason an own implementation has been made. It is also a task to find a more clever algorithm to automatically generate *exercise*s with a high chance of being suitable as an exam exercise.

1.2 Context Free Grammar

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

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

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

Definition 2. CFG with restrictions
A CFG $G = (V,\ \Sigma,\ S,\ P)$ has the following restrictions:

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

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

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

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.

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

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

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

1.3 General approaches

Now two basic approaches that may help finding a good algorithm are explained informally like in [4]. At first the difference between derivation and parsing and secondly the two different ways of parsing are explained shortly.

1.3.1 Forward Problem & Backward Problem

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

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

It is called Forward Problem, if you are given a grammar G and form a derivation from its root node to a final word w. The derived word w is always part of $L(G)$.

Definition 5. Backward Problem = Parsing ($w \overset{?}{\subseteq} L(G)$)
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 Backward Problem or parsing. The Cocke-Younger-Kasami algorithm does parsing and is the focus here.

1.3.2 Parsing Bottom-Up & Top-Down

After knowing what parsing in general is, it is important to know that there are again two ways to classify the approach of parsing itself.

Bottom-Up parsing

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

"Bottom-Up parsing is the general method used in the Cocke-Younger-Kasami algorithm, which fills a parse table from the "bottom up"" [4].

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." [4] (The so called test sentence is synonymous to an word w.) [4].

1.4 Data Structure Pyramid

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

Definition 6. $[i, j]$

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

Definition 7. $Cell_{i,j}$

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

Now *Pyramid* can be defined as following:

Definition 8. $Pyramid$

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

The following is the visual representation of a *Pyramid*.

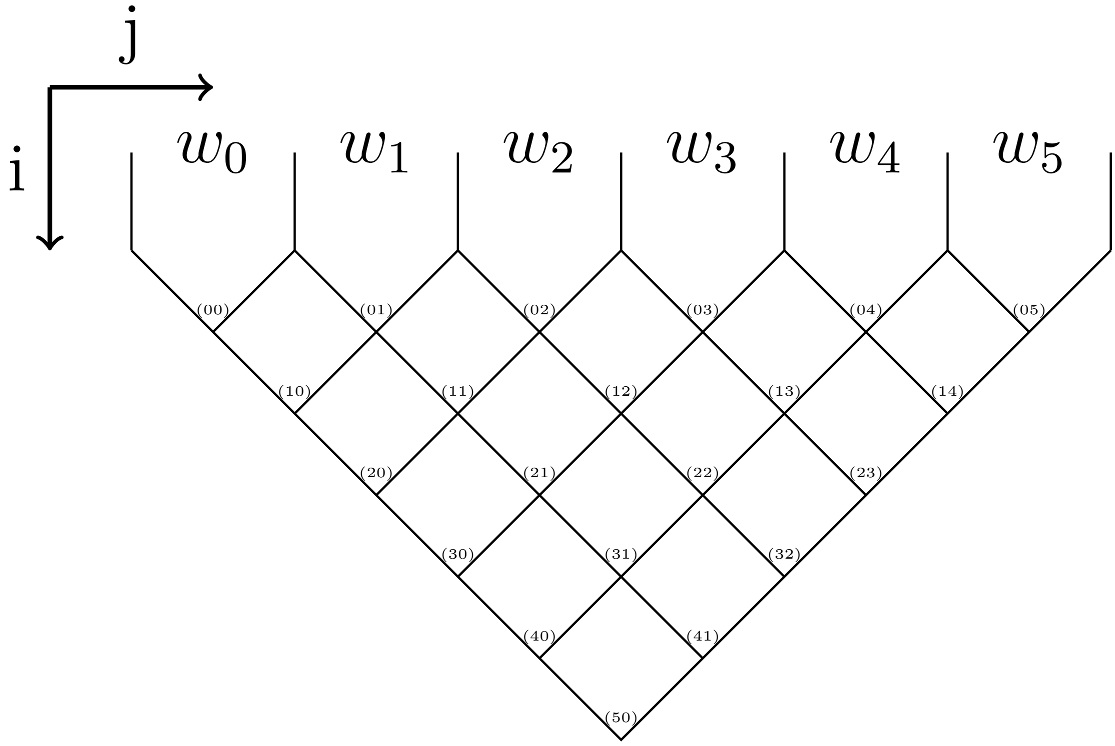


Figure 1: Visual representation of a *Pyramid* with the word w written above it.

Lastly we define $CellBottom$, $CellUpperLeft$ and $CellUpperright$ that are used for description in Algorithm 11.

Definition 9. *CellDown, CellUpperLeft and CellUpperRight*
For every $Cell_{i,j}$ with $i > 0$ the following is true:

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

1.5 Cocke-Younger-Kasami Algorithm

The Cocke-Younger-Kasami Algorithm (CYK) has been developed independently in the 1960s by Itiroo Sakai [5], John Cocke and Jacob Schwartz [6], Tadao Kasami [7] and Daniel Younger [8] [9].
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 if $w \in L(G)$ through the presence of the start variable in the tip of the pyramid..
The description of the algorithm follows the source [10] adjusted to the data structure *Pyramid*. While describing Algorithm 11 later on a back reference to the CYK-

Algorithm here will be made that points out a similarity.

Algorithm 1: CYK

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

Output: true $\Leftrightarrow w \in L(G)$

1 $Pyramid = \emptyset$;

2 **for** $j := 0 \rightarrow i_{max}$ **do**

3 $\quad Pyramid \cup = \{(X, j+1) \mid X \longrightarrow w_j\}$; `// Fills cells` $Cell_{0,j}$

4 **end**

5 **for** $i := 1 \rightarrow i_{max}$ **do**

6 $\quad$ **for** $j := 0 \rightarrow j_{max,i}$ **do**

7 $\quad\quad$ **for** $k := i-1 \rightarrow 0$ **do**

8 $\quad\quad\quad Pyramid\ \cup \{(X, k) \mid X \longrightarrow YZ,\ Y \in Cell_{k,j},\ Z \in Cell_{i-k-1,k+j+1}\}$;

$\quad\quad\quad\quad$ `// Fills cells` $Cell_{i,j}$ `??k?? XXX`

9 $\quad\quad$ **end**

10 $\quad$ **end**

11 **end**

12 **if** $(S, i) \in Cell_{i_{max},0}$ **then**

13 $\quad$ **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 Algorithm 1 the parsing table is filled as shown in Figure 2. An example output of this definition of the CYK could look like this:

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

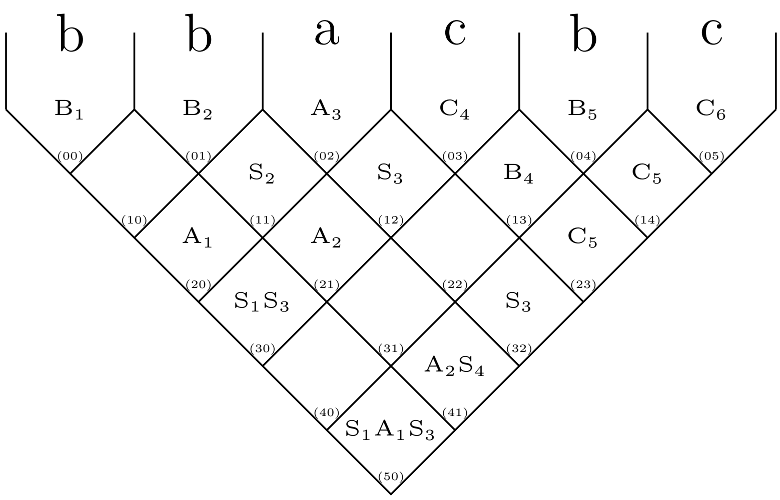


Figure 2: The CYK algorithm fills the cells of the pyramid during execution (Line 3 and Line 8).

2 Algorithms

2.1 Sub modules

Sub modules are exchangeable parts of the algorithms and are denoted circled with A, B, C, D and E. They are procedures that should be explained in more detail for a better understanding of the way of working of algorithms in Chapter 2.2ff.

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

The difference between A and B is that one time Σ and the other time V^2 are distributed. 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: $Rhse \subseteq\ (V^2 \cup \Sigma),\ V$

Output: Set of productions $P \subseteq V \times (V^2 \cup \Sigma)$

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

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

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

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

end

return P;

Stopping Criteria C:

Two kinds of stopping criteria have been used to determine if a algorithm terminates early on. One is that it stops iff more than half of the pyramid cells are not empty any more and the other one is that there is at least one variable in the tip of the pyramid. Both stopping criteria are compared in short in Chapter 2.7.

CalculateSubsetForCell(Pyramid, i, j) 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). If now for one $Cell_{i,j}$ a rule like $lhse \rightarrow cs$ with $cs \in CellSet$ (*Line* 3) is added then automatically $Cell_{i,j}$ won't be empty any more.

Algorithm 3: CalculateSubsetForCell

Input: $Pyramid,\ i \in \mathbb{N},\ j \in \mathbb{N}$

Output: $CellSet \subseteq V^2$

$CellSet = \emptyset$;

for $k := i-1 \to 0$ **do**

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

end

return $CellSet$;

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

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

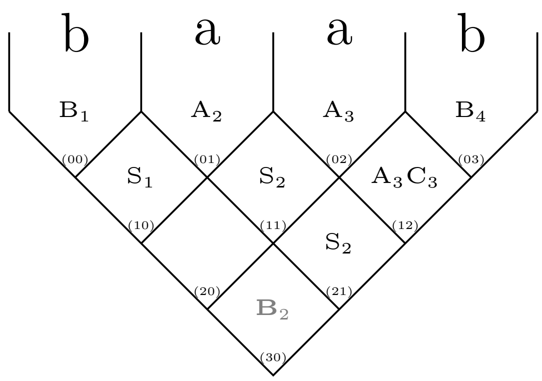


Figure 3: Example of Algorithm 3 while applying it on $Cell_{3,0}$ via adding the rule $B \to SC$.

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 is 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 $\mathrm{B} \to \mathrm{SC}$ has been added and now $(B, 2)$ is element of $Cell_{3,0}$.

Choose one xy from (xy,i) $\in$ RowSet uniform randomly with probability depending on row i 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 change in one of the first rows it is more likely that the entire pyramid will be more filled. 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 E.

The approach here only makes sense together with D in which all possible compound variables are calculated that help to fill one specific cell. If you use this sub module on every cell of the pyramid to calculate the different variable compounds xy and additionally store the row number i then you get the set $RowSet \subseteq \{(xy, i) \mid x, y \in V \land i \in \mathbb{N}\}$.

Using 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 uniform randomly picked out of this weighted $RowMultiSet$ example wise $xy = BC$.

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

Figure 4: Shortened example of the procedure E as before in the text.

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. Each future algorithm should perform better than this algorithm or otherwise it would be worse, than simple dice rolling the distribution of terminals (Line 2) and compound variables (Line 3). Note that later on in Chapter 2.7.1 it is described how algorithm can perform better than another.

Algorithm 4: DiceRollOnlyCYK

```
   Input: Word w ∈ Σ*
   Output: Set of productions P
1  P = ∅;  // P ⊆ V × (V² ∪ Σ)
2  P = Distribute(Σ, V); A
3  P ∪ Distribute(V², V); B
4  return P;
```

A terminal Σ is distributed to at least one $lhse$, but a compound variable V^2 must not be distributed at all. This means that for each terminal of $\Sigma = \{a, b\}$ there exists at least one rule like $lhse \to a$ and $lhse \to b$ and for each possible compound variable $V^2 = \{AA,\ AB,\ AC,\ AS,\ BB,\ BC,\ BS,\ CC,\ CS, SS\}$ it is possible that only a smaller subset like $\{AA,\ BA,\ CC,\ SC\}$ is distributed so that only rules like $lhse \to AA$, $lhse \to BA$, $lhse \to CC$ 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 5: Shortend overview of an example of Algorithm 4 as described before.

2.3 Dice rolling and Bottom-Up approach variant 1

Another approach to design an algorithm will be Bottom-Up (Chapter 1.3) whereby the parsing table is filled starting from the leaves in direction to the root node.
The basic idea is to guide the choice of rules while distributing the compound variables V^2. In Algorithm 4 it can happen 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.

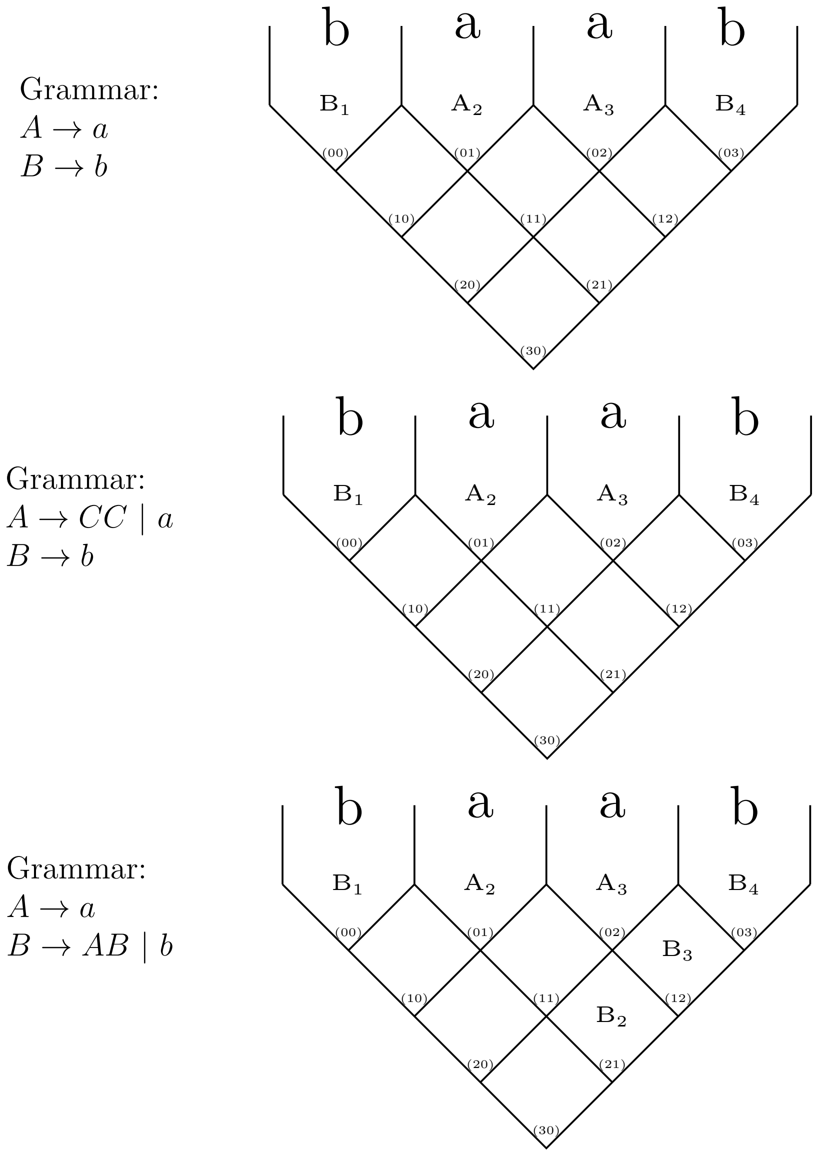


Figure 6: Example of disregarding the already added rules. Top: starting situation. Middle: Unfortunate adding of rules that doesn't help to fill the parsing table and can happen in Algorithm 4. Bottom: Good adding of rules as intended in Algorithm 5 that helps filling.

If rules like $lhse \rightarrow CC$ or $lhse \rightarrow SC$ are added they don't directly help to fill the parsing table and bloat the grammar with useless rules. More reasonably rules to add would be $lhse \rightarrow BA$, $lhse \rightarrow AA$ and $lhse \rightarrow AB$ (for an example see Figure 6). Algorithm 5 takes this up: 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. I.e. 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 = ∅; // P ⊆ V × (V² ∪ Σ)
2  P = Distribute(Σ, V); A
3  Pyramid = CYK(G, w);
4  for i := 1 to i_max do
5      J = {0, ... , j_max − 1}; // J ⊆ ℕ
6      CellSet = ∅; // CellSet ⊆ V²
7      while |J| > 0 do
8          choose one j ∈ J uniform randomly;
9          J = J \ {j};
10         CellSet = CalculateSubsetForCell(Pyramid, i, j); D
11         P ∪ Distribute(CellSet, V); B
12         Pyramid = CYK(G, w);
13         if stopping criteria met C then
14             return P;
15         end
16     end
17 end
18 return P;
```

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

2.4 Dice rolling and Bottom-Up approach variant 2

While examining Algorithm 5 via its log file (Figure 7) it can be seen (for the default values described in Chapter 1.2) that already a very small number of rules in the grammar is sufficient so that the stopping criteria 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 7: Digest of log files of Algorithm 5.

This again leads to further improvement idea to introduce a row dependent $threshold_i$ (Line 9) 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 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 (See explanation in Figure 6). 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 E (Line 10). Variable compounds that already have been used in a row with low index i are at a disadvantage to be picked again as described in Algorithm 3.
As seen in Figure 8 rules with BA and AA have been added to the variables B and A in Grammar1. For Grammar2 instead the rule $B \to SS$ was added that contributes to a better diversity 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 8: Example for better diversity. Starting point is Grammar0. Grammar2 is of better diversity than Grammar1.

Algorithm 6: BottomUpDiceRollVar2

```
   Input: Word w ∈ Σ*
   Output: Set of productions P
1  P = ∅;  // P ⊆ V × (V² ∪ Σ)
2  RowSet = ∅;  // RowSet ⊆ {(xy, i) | x, y ∈ V ∧ i ∈ ℕ}
3  P = Distribute(Σ, V); (A)
4  Pyramid = CYK(G, w) ;
5  for i := 1 to i_max do
6      for j := 0 to j_max − i do
7          RowSet ∪ {(xy, i) | xy ∈ CalculateSubsetForCell(Pyramid, i, j)(D)};
8      end
9      while threshold_i not reached do
10         choose one xy from (xy, i) ∈ RowSet uniform randomly with
            probability depending on i; (E)
11         P ∪ Distribute(xy, V); (B)
12         Pyramid = CYK(G, w);
13         if stopping criteria met (C) then
14             return P;
15         end
16     end
17 end
18 return P;
```

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

2.5 Split Top-Down then fill Bottom-Up

Up till now we have only discussed algorithms that purely use the Bottom-Up approach, so another way is to make use of the Top-Down approach in combination with the Bottom-Up approach.
The idea here is to first distribute the terminals (Line 2 of Algorithm 7) and then to uniformly randomly generate a predefined structure of the derivation tree (Line 4 of Algorithm 2 and in general Algorithm 8) 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:

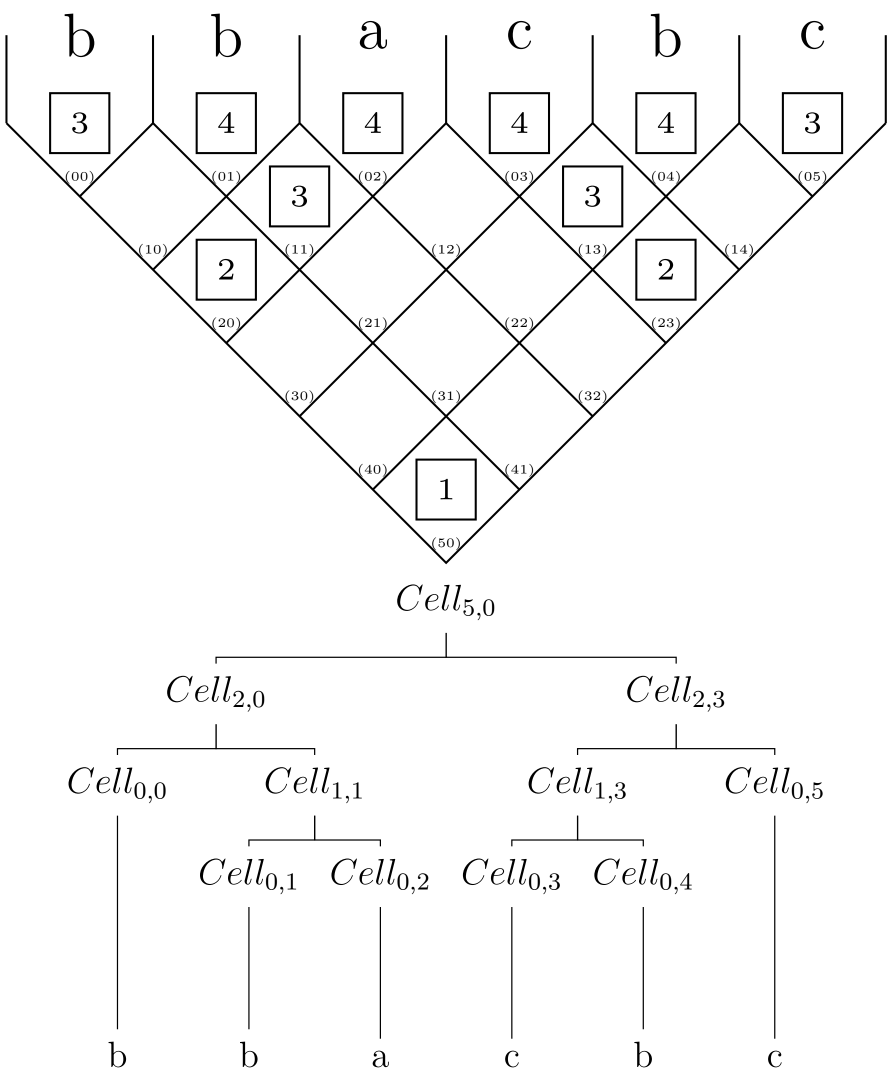


Figure 9: Example derivation tree structure. Top: Shown in the pyramid, the numbers correspond to the depth in the tree. Down: Shown as a derivation tree.

As the name of the algorithms implies only after completely generating the structure of the derivation tree (splitting of the word in subwords) then 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 Line 14) the already available

information regarding the other rules is used to determine if a new rule is needed to fill this node of the derivation tree (Line 12 of Algorithm 8).

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

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 B (Line 14 of Algorithm 8) a variable compound is added to at least one *lhse*. For every element of $vc \in VarComp$ (Line 13 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* C **then**

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

11 **end**

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

13     $VarComp = uniform\ random\ subset\ from\ \{vc \mid v \in Cell_l\ \wedge\ c \in Cell_r\}\ with\ |VarComp| \geq 1$;

14     $P \cup Distribute(VarComp,\ V)$; B

15 **end**

16 **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:

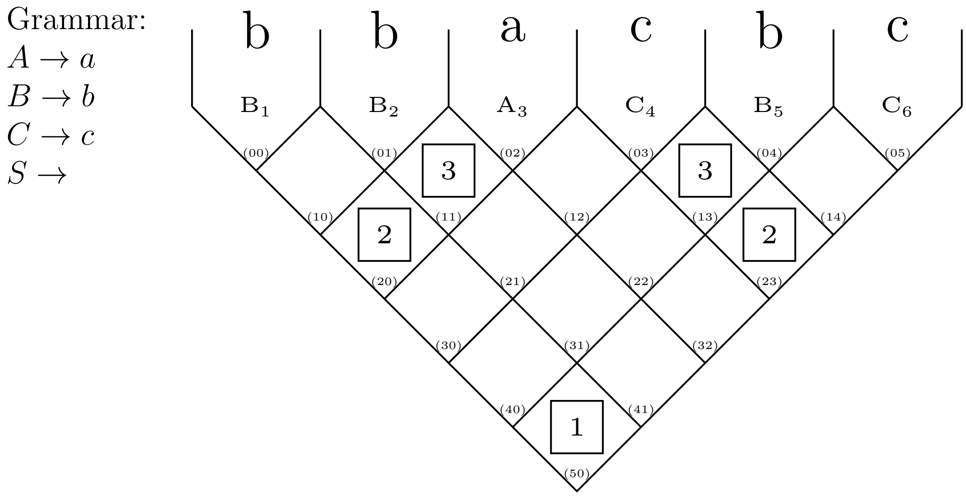


Figure 10: Illustration of Algorithm 7 part 1 after adding $A \rightarrow a$, $B \rightarrow b$ and $C \rightarrow c$.

After adding the terminals to the grammar (Line 2 in Algorithm 7) now one must take on the recursion step at $Cell_{1,1}$. Now $Cell_l = \{B_2\}$ and $Cell_r = \{A_3\}$ and therefore $VarComp = \{BA\}$. Adding the rule $S \rightarrow BA$ leads to the following changes:

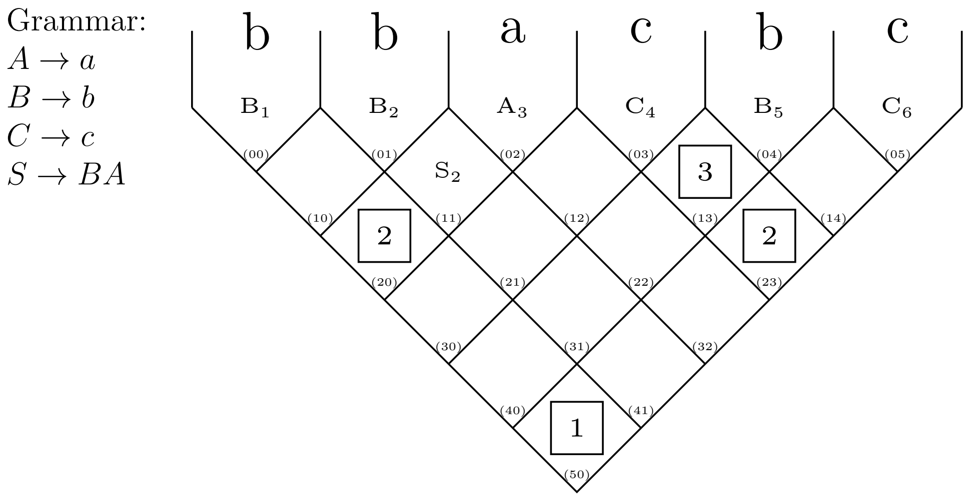


Figure 11: Illustration of Algorithm 7 part 2 after adding $S \rightarrow BA$.

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

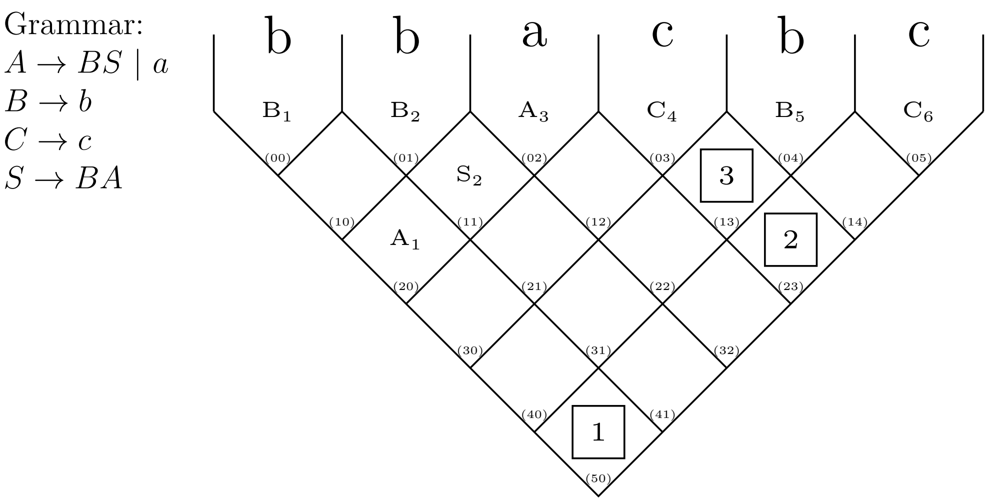


Figure 12: Illustration of Algorithm 7 part 3 after adding the rule $A \rightarrow BS$.

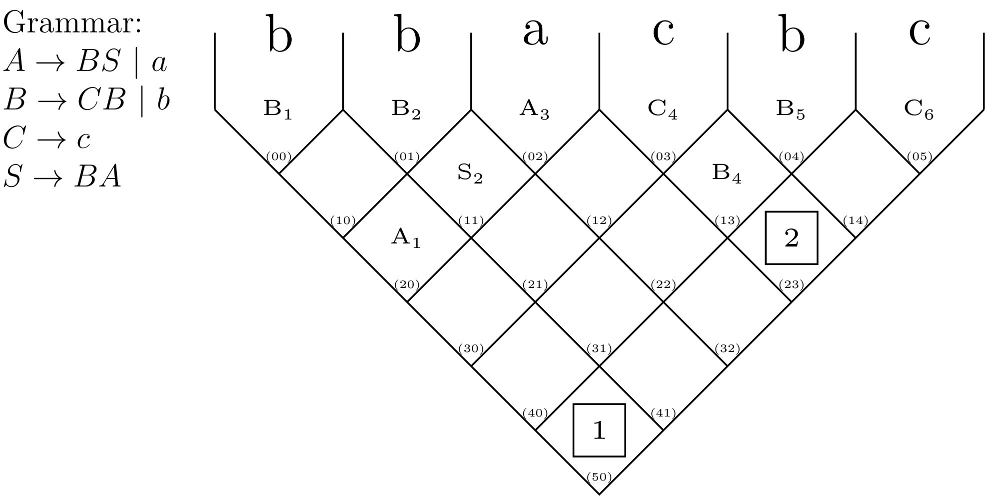


Figure 13: Illustration of Algorithm 7 part 4. The recursion step in $Cell_{1,3}$ is resolved by adding the rule $B \rightarrow CB$.

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

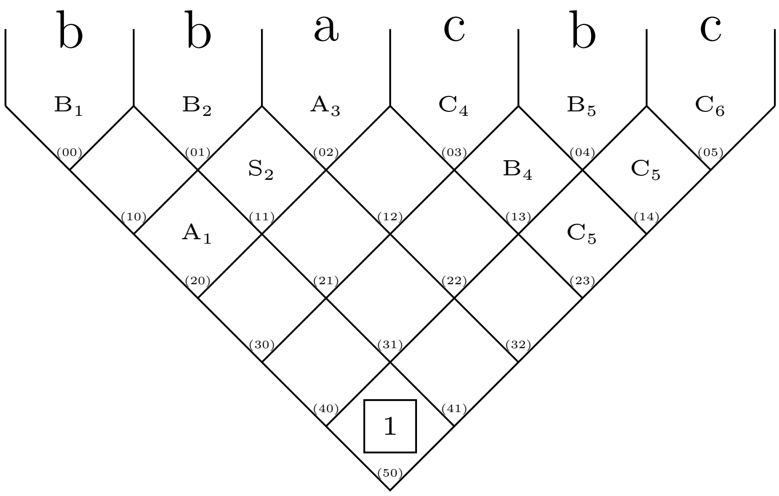


Figure 14: Illustration of Algorithm 7 part 5. The recursion step in $Cell_{2,3}$ is resolved by adding the rule $C \to BC$.

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

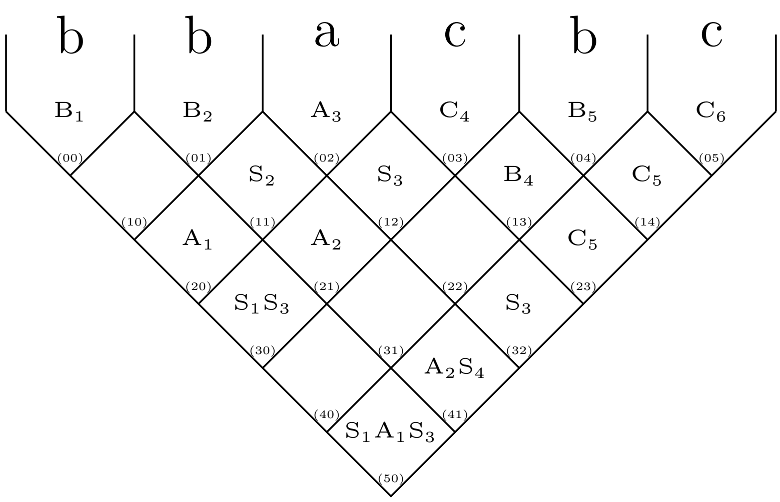


Figure 15: Illustration of Algorithm 7 part 6. The recursion step in $Cell_{5,0}$ is resolved by adding the rule $S \to AC$.

2.6 SplitAndFill

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.
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 the start variable will be in the tip in 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 $(P,\ v_l) = \mathit{SplitAndFillRec}(P,\ w,\ (m-j-1),\ j)$;
10 $(P,\ v_r) = \mathit{SplitAndFillRec}(P,\ w,\ (j+i-m),\ m)$;
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 $v_{lhse} \rightarrow w_j$, 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 $v \in V$.

Looking at this algorithm, only productions according to the tree structure are added to the grammar. For illustration purposes, the pyramid here is also shown to reflect the immediate changes of the added rules to the pyramid. Again the predefined derivation tree structure of Figure 2.5 is used.

Grammar:
$A \rightarrow$
$B \rightarrow b$
$C \rightarrow$
$S \rightarrow$

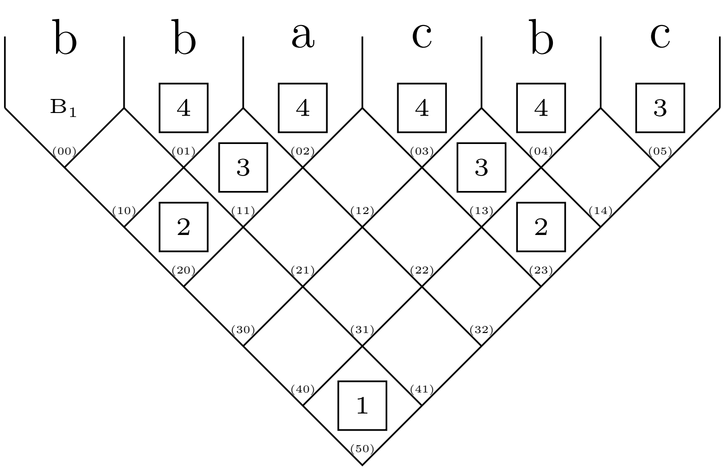


Figure 16: Illustration of Algorithm 9 part 1. To resolve the recursion step that fills $Cell_{0,0}$ the rule $B \rightarrow b$ is added.

Grammar:
$A \rightarrow a$
$B \rightarrow b$
$C \rightarrow$
$S \rightarrow BA$

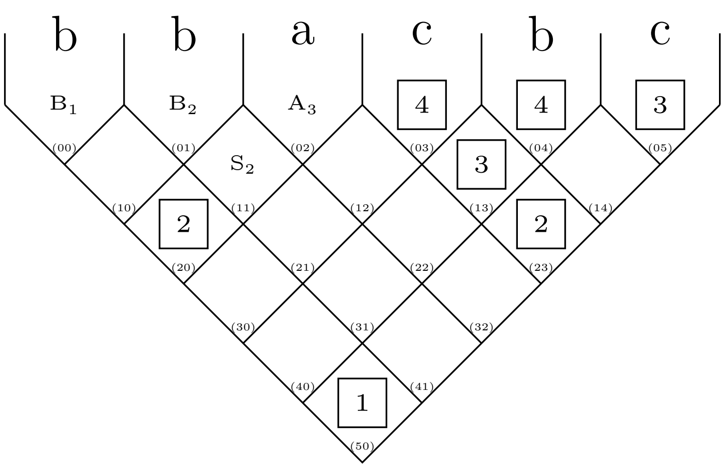


Figure 17: Illustration of Algorithm 9 part 2. Resolving the recursion step that fills $Cell_{0,1}$ no rule is added because a rule $lhse \rightarrow b$ already exists. To fill $Cell_{0,2}$ the rule $A \rightarrow a$ is added. Regarding $Cell_{1,1}$ the rule $S \rightarrow BA$ is added.

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

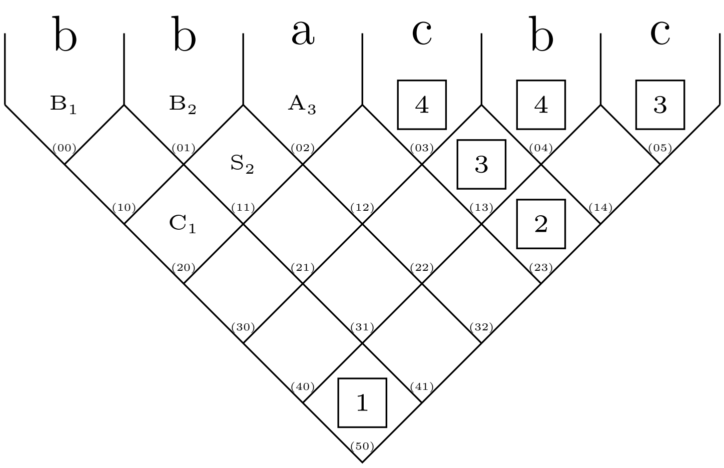


Figure 18: Illustration of Algorithm 9 part 3. Filling the $Cell_{2,0}$ the rule $C \rightarrow BS$ is added.

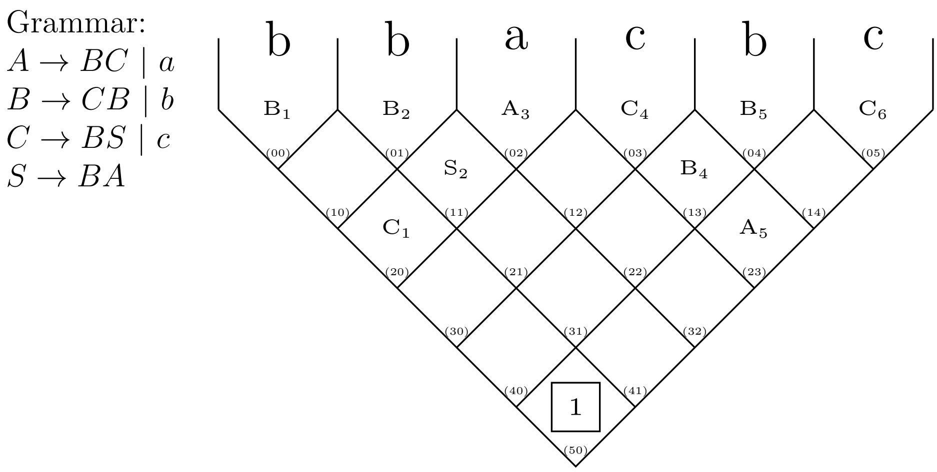


Figure 19: Illustration of Algorithm 9 part 4. Analogously the other cells are filled. $Cell_{0,3}$ is responsible for the rule $C \to c$, $Cell_{0,4}$ doesn't cause a rule because again there already is the rule $B \to b$, $Cell_{1,3}$ contributes for the rule $B \to CB$, $Cell_{0,5}$ does not add a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

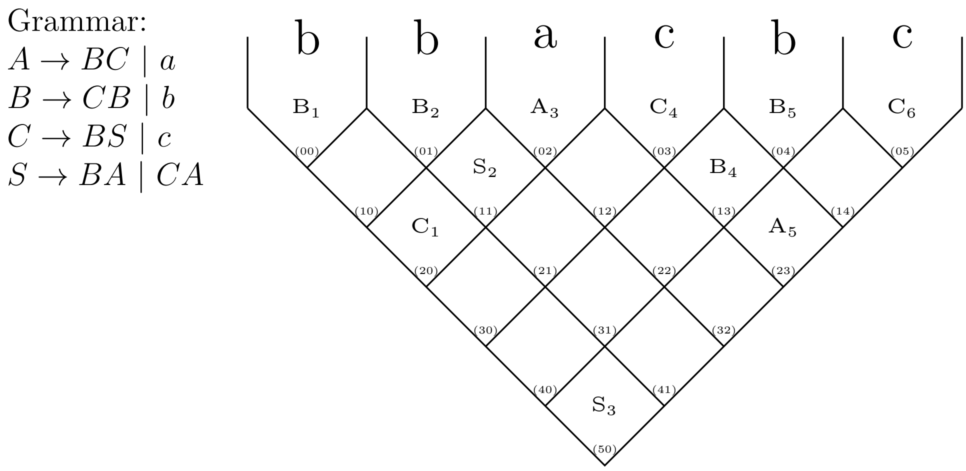


Figure 20: Illustration of Algorithm 9 part 5. Finally, to fill the cell in the tip a rule must be added that has the start variable as its *lhse* that guarantees $w \in L(G)$. Here the rule $S \to CA$ is added.

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

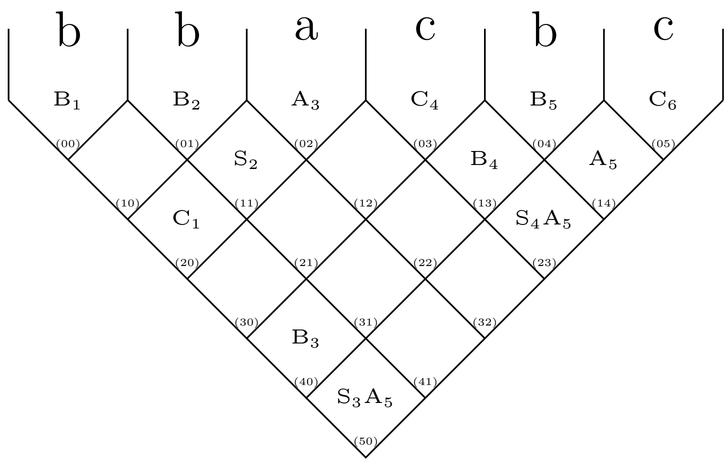


Figure 21: Illustration of Algorithm 9 part 6. With part 5 of the example the algorithm is finished. In comparison to Figure 20 the complete parsing table looks like above.

2.7 Evaluation of Algorithms

2.7.1 Success Rates

Now that different algorithms have been described to generate exam *exercises* it is of interest to compare them. Therefore different Success Rates are used on the algorithms according to their performance of the different requirements for an exam *exercise*. Here $N \in \mathbb{N}^+$ is the sample size of all generated grammars of the examined algorithm. Before defining a overall Success Rate (SR) three other Success Rates set the basis for it.

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

Success Rate Cardinality-Rules: A generated *exercise* contributes to the SR-Cardinality-Rules iff the grammar has got less than a certain amount of productions. SR-Cardinality-Rules $= cr/N$, whereas cr is the count of *exercise*s.

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

1. At least one cell forces to do a correct cell combination.
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 $= p/N$, whereas 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:
$Cell_{i,j} \subseteq \{(V, k) \mid k \in \mathbb{N}\} \longrightarrow Cell_{i,j} \subseteq V$, see Figure 22.

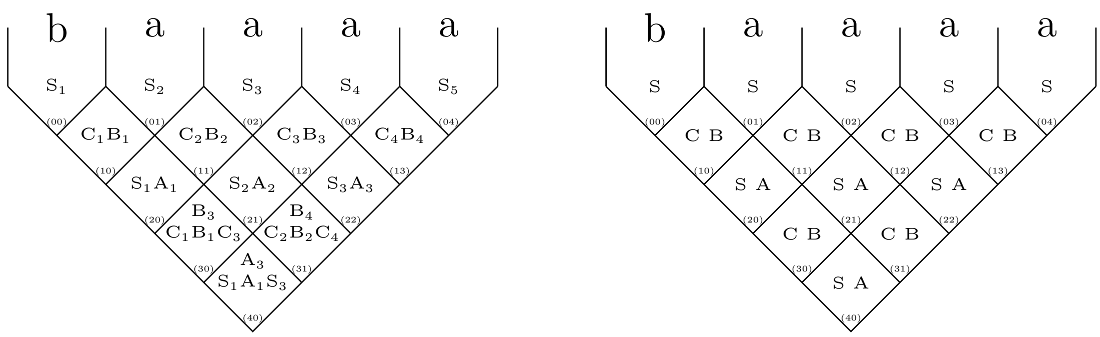


Figure 22: The simplification of cells in a pyramid.

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. Students often don't know what cell combinations need to be considered while filling one specific cell of the pyramid. They simply take the $UpperLeftCell$ and the $UpperRightCell$ and try to find rules in the grammar that match the resulting compound variables. More detail on how to force a correct cell combination (1.) see Algorithm 11. But note that a right cell combination can only be forced of cells with index $i > 1$.

Algorithm 11: checkForceCombinationPerCell

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

Output: $true \iff |VarsForcing| > 0$

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

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

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

4     $Rhses = \{rhse \mid p \in P \ \wedge \ p = (v, rhse)\}$;

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

6         $VarsForcing = VarsForcing \cup v$;

7   **end**

8 **end**

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

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.

As seen 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. $Cell_{3,0}$ doesn't force because $VarComp = \{CC\}$ and the variable $v = S$ has CC as its rhse. Note again, that cells with index $i \leq 1$ can't force at all.

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

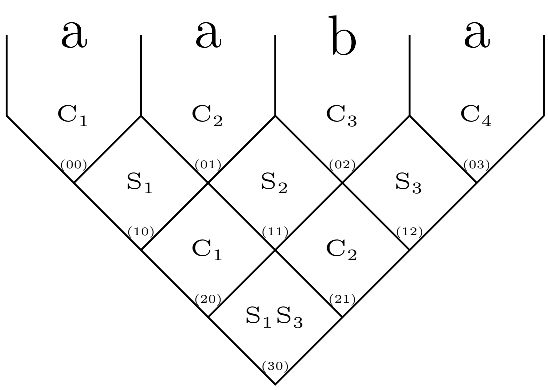


Figure 23: Application of Algorithm 11 onto an entire pyramid.

Now that the three Success Rates are known the overall Success Rate can be specified.

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

2.7.2 Problem space exploration

For the problem space exploration the following settings need to be considered while each shown value represents the default value for its parameter.

Input Values of the program:

- variables = [A, B, S, C]
- terminals = [a, b]
- sizeOfWord = 10

Specified parameters that decide over the SR:

- maxSumOfProductions = 10
- minRightCellCombinationsForced = 1
- maxNumberOfVarsPerCell = 3
- maxSumOfVarsInPyramid = 100

Intern algorithm dependent parameters:

- minValueCompoundVariablesAreAddedTo = 0
  (=1 in case of Algorithm SplitThenFill and Algorithm SplitAndFill)
- minValueTerminalsAreAddedTo = 1
- maxValueCompoundVariablesAreAddedTo = 2
  (=1 in case of Algorithm SplitThenFill and Algorithm SplitAndFill)
- maxValueTerminalsAreAddedTo = 1

The problem space or input parameter space in itself is a n-dimensional hypercube with dimension n equalling the count of parameters, i.e. n = #parametersAbove = 3 + 4 + 4 = 11.
But now the evaluation area of our problem space is restricted to exam *exercise* relevant parameters, that have been decided upon before.
Within that predefined area now one can find the optimal settings for the algorithm dependent parameters and can use these to finally compare them.
During the comparison of the algorithms the n-dimensional problem space is reduced to a single scalar value, the Success Rate.

Because of the restricted problem space brute forcing is used to find the best parameter settings.

2.7.3 Comparison of Success Rates

By comparing Table 1 and Table 2 it is seen that the stopping criteria doesn't have that much of an influence on the algorithms SR with the used input configuration. In

Algorithm	SR	Producibility	Cardinality-Rules	Pyramid			
					Force-Right	Vars-PerCell	VarsIn-Pyramid
DiceRollOnly	04%	24%	59%	37%	50%	88%	94%
BottomUpVar1	15%	51%	89%	42%	73%	76%	67%
BottomUpVar2	19%	46%	92%	54%	80%	79%	77%
SplitThenFill	23%	39%	97%	68%	78%	91%	93%
SplitAndFill	11%	100%	70%	14%	79%	34%	22%

Table 1: SRs of the algorithms with stopping criteria that the root is not empty (N = 10000).

both cases the SplitThenFill algorithm is the one with the highest SR.

Algorithm	SR	Producibility	Cardinality-Rules	Pyramid			
					Force-Right	Vars-PerCell	VarsIn-Pyramid
DiceRollOnly	04%	24%	59%	37%	50%	88%	94%
BottomUpVar1	11%	29%	99%	56%	69%	91%	87%
BottomUpVar2	12%	25%	99%	66%	78%	90%	93%
SplitThenFill	23%	40%	97%	66%	78%	90%	92%
SplitAndFill	10%	100%	69%	14%	78%	36%	20%

Table 2: SRs of the algorithms with stopping criteria that half of the parsing table is not empty (N = 10000).

2.8 Conclusion

3 GUI Tool: CYK Instances Generator

One of the goals of the thesis is to get a small tool that assists in creating exam exercises to test if the students have understood the CYK algorithm.

3.1 Overview GUI

The developed tool consists out of four major elements as marked in Figure 24.

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

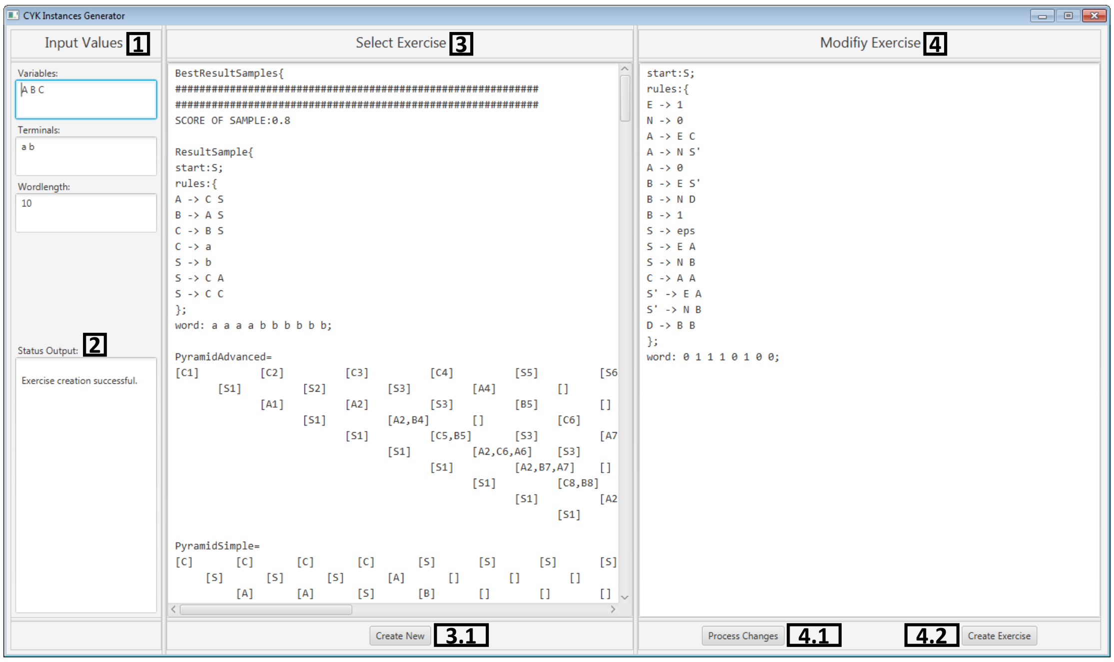


Figure 24: CYK Instances Generator.

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 output for this *exercise* is done through a LaTeX-code-file and a pdf-file.

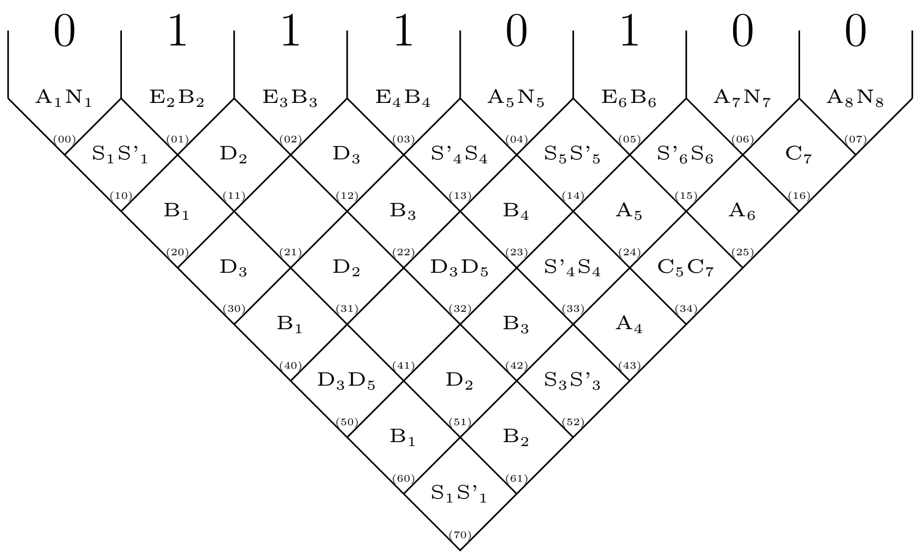
3.1.1 Working with the program

There is the folder BachelorThesisCYK that contains the executable "bachelor_thesis_cyk.jar" file and four other folders. One of these folders is named "exercise". After clicking button 4.2 "Create Exercise" a new "exerciseLatex.tex"-file and the corresponding "exerciseLatex.pdf"-file will be generated within it.

3.2 Exam Exercises

A exam *exercise* is a 4-tuple $exercise = (grammar,\ word,\ parse\,table,\ derivation\,tree)$. The pdf-file output of button 4.2 of the tool looks as following:

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



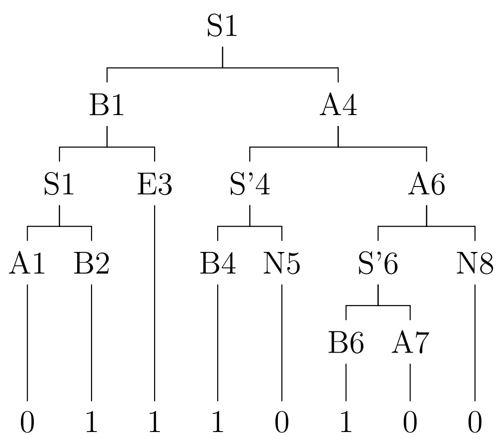


Figure 25: Example output for a *exercise*.

3.3 Scoring Model

To be able to find suitable exercises that can be displayed in area 3 of the tool a scoring model is needed. The exercises are given a score according to Table 3. 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]	≤2

Table 3: Scoring of the different parameter values.

The score of each *exercise* is normalized to the maximum possible points so that the maximum score is 1.0.

$score = (\#Parameter \cdot 10)^{-1} \cdot \sum_{parameter} points$

A high negative score is used to avoid examples in area 3 with undesired properties.

3.4 Parsing input with ANTLR

The first step here is the tokenization of the input. After that with the help of the Grammar seen below a abstract syntax tree is generated out of which intern Java objects can parsed.
The used grammar is a LL(k) grammar whereas each derivation step can be distinctly identified through the next k tokens.
ANTLR has been used because it enables a clear separation between the language definition and the object handling in the code.
In Figure 26 the rules of the grammar are seen and in Figure 27 its used 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 26: Formal definition of the used ANTLR grammar rules.

```
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 27: Formal definition of the used ANTLR grammar tokens.

3.5 Other matters

Technologies that have been used for programming are Github [11] with Sourcetree [12] for version control, Maven [13] for build management, IntelliJ [14] as the IDE, ANTLR [15] with ANTLRWorks for parsing input and JavaFX Scene Builder [16] to create the GUI.

Important used frameworks are: JUnit [17] for testing and Project Lombok [18] to greatly reduce boilerplate code.

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

References

[1] http://lxmls.it.pt/2015/cky.html.

[2] http://jflap.org/tutorial/grammar/cyk/index.html.

[3] https://github.com/ajh17/CYK Java.

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

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

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

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

[8] D. H. YOUNGER. Recognition and parsing of context-free languages in time n3. *INFORMATION AND CONTROL*, 10(2):189–&, 1967.

[9] https://de.wikipedia.org/wiki/Cocke Younger-Kasami-Algorithmus.

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

[11] https://github.com/.

[12] https://www.sourcetreeapp.com/.

[13] https://maven.apache.org/.

[14] https://www.jetbrains.com/idea/.

[15] http://www.antlr.org/.

[16] http://gluonhq.com/products/scene builder/.

[17] http://junit.org/junit4/.

[18] https://projectlombok.org/features/all.