Umeå University
Efficient Algorithms

Assignment Step 2

Runtime analysis of the Java implementation of the CYK-algorithm

Pina Kolling

September 27, 2022

Contents

1 Introduction

Parsing in Computer Science is the process of analysing a string of characters to examine if the string is built according to the rules of a formal grammar.
A formal grammar describes how to form strings with correct syntax from a language's alphabet (explained in section 2.2 and 2.1). To examine if such a string follows the rules of a grammar the *Cocke-Younger-Kasami*-algorithm (short: *CYK*) can be used. This algorithm is described in section 2.4. To use the *CYK*-algorithm the grammar needs to be in a specific format, that is called *Chomsky-Normal-Form* (short: *CNF*), which is explained in section 2.3. [5, 7]
The task for this assignment was to code three different parsing methods to execute the *CYK*-algorithm in *Java*. The different parsing methods will be described and presented as pseudo code in section 3. For the implementation three different classes are implemented: `main.java, grammar.java` and `parser.java`. The `main`-class calls the methods and the `grammar`-class parses the input grammer ans string into a format that then can be processed in the `parser`-class. This `parser`-class has three different parsing methods, which will be tested and compared against each other. The function and implementation will be former described in section 3. Also in this section the runtimes in O-notation are calculated for each approach.
In section 4 the runtimes and experiments of the different algorithms are compared and differences in efficiency are shown.
In the final part (section 5) the results and future possibilities are discussed.

2 Background

In this section background information on formal languages, alphabets and Kleene star (section 2.1), formal grammars (section 2.2), Chomsky-Normal-Form (section 2.3), CYK-algorithm (section 2.4) and dynmaic programming (section 2.5) will be presented.

2.1 Formal language

Formal languages are abstract languages which define the syntax of the words that get accepted by that language. It is a set of words that get accepted by the language and has a set of symbols that is called alphabet and contains all the possible characters of the words. Those characters are calles nonterminal symbols. [1, 6]

Definition (Formal language)**.** *A formal language L over an alphabet Σ is a subset of the Kleene star of the alphabet: $L \subseteq \Sigma^*$*

Definition (Kleene star)**.** *The Kleene star Σ^* of an alphabet Σ is the set of all words that can be created through concatenation of the symbols of the alphabet Σ. The empty word ϵ is included.*

Example (Formal language)**.** [1] *The language accepts words that contain the same number of as and bs, while the a has to be left of the b. The alphabet Σ of this language looks like this:*

$$\Sigma = \{a, b\}$$

The Kleene star Σ^* of the alphabet Σ looks like this:

$$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, abb, bbb, bba, baa, aba, bab, ...\}$$

The language definition L is the following one:

$$L = \{(a^n b^n)^m\} \text{ with } n, m \in \mathbb{N}$$

2.2 Formal grammar

A formal grammar describes how to form strings with correct syntax from a language's alphabet. A grammar does not describe the meaning of the strings or any semantics — only their syntax is defined. The grammar is a set of rules which define which words are accepted by a formal language. Those rules consist of terminal and nonterminal symbols. The terminalsymbols are the characters of the alphabet of the language and the nonterminalsymbols are used to build the rules of the language – they get replaced by terminalsymbols. [1, 6]

[1] The following examples show the *Well-Balanced Parantheses* example from the assignment task sheet with the alphabet $\{a, b\}$ instead of $\{(,)\}$.

Definition (Formal grammar). *A formal grammar G is defined as a 4-tuple:*

$$G = (V, T, P, S)$$

The set V contains the nonterminal symbols of the grammar and the set T the terminal symbols, which is the alphabet of a language. This concludes to $V \cap T = \emptyset$. The set P is the set of production rules, where an element of V points to an element of $V^* \times T^*$. The symbol $S \in V \setminus T$ represents the start symbol.

Example (Formal grammar). *The example grammar G for the previous example language L over the alphabet Σ is the following:*

$$G = (\{S\}, \{a, b\}, \{S \to SS, S \to aSb, S \to ab\}, \{S\})$$

The rules of the grammar G over the alphabet $\{a, b\}$ with the start symbol S and the nonterminal symbols $\{S\}$ can also be written in the following form:

$$S \ \to \ SS \ | \ aSb \ | \ ab$$

2.3 Chomsky-Normal-Form

The *Chomsky-Normal-Form* (short: *CNF*) is a grammar which is formated in a specific way. If the startsymbol (nonterminal symbol) is not generating the empty word ($S \to \epsilon$) it either generates two nonterminal symbols or one terminal symbol for the grammar to be in *CNF*. [1]

Example (Chomsky-Normal-Form). *This is the previous example in CNF. How it was edited can be seen in appendix C.1.*

$$S \to SS \mid LA \mid LR$$

$$A \to SR$$

$$L \to a$$

$$R \to b$$

2.4 CYK-algorithm

Cocke-Younger-Kasami-algorithm (short: *CYK*) takes a grammar $G = (V, T, P, S)$ in *CNF* and a word $w = w_1, w_2, ..., w_n \in T^*$ as an input. It then examines if the word follows the rules of the grammar. Then for every substring $w_{i,j} = w_i, ..., w_{i+j-1}$ (begins at index i and has the length j) of the word w the set of nonterminal symbols that lead to $w_{i,j}$ gets calculated and saved as $V_{i,j}$ to access it in later steps. [7]

Example (Chomsky-Normal-Form). *The following table shows the CYK-algorithm with the previous grammar example and the input word aaabbb. [3]*

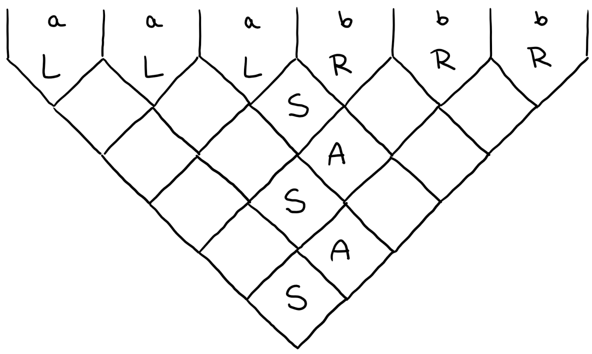


Figure 1: CYK-algorithm table for the word *aaabbb*

The word is accepted by the grammar rules, because the intiating nonterminal symbol S *can be filled into the lowest field. In appendix C.2 the table is filled in step by step and the rules that are used in each step are marked.*

2.5 Dynamic programming

The technique of dynamic programming can be used to solve problems, which can be devided into smaller subproblems. The solutions of the subproblems are saved (for example in a multi-dimensional array) and referenced later. [2]

Example (Knapsack problem)**.** *As an example the table for the knapsack problem is filled out, because it is a really common example for the concept of dynamic programming.*

Given is a set of items with a weight and a value. The task is to choose which items to include so that the total weight is less than the given limit of the knapsack and the total value is as large as possible.

The formula with which the dynamic programming works is the follwing:

$$Opt(i,j) = \begin{cases} 0, & for\ 0 \leq j \leq size \\ Opt(i-1,j), & for\ j < w[i] \\ max\{Opt(i-1,j), v[i] + Opt(i-1, j-w[i])\}, & else \end{cases}$$

If $0 \leq j \leq w$ *then there are no objects which could be put into the bag. The second case acts when object* i *does not fit into the bag and the optimal solution is found with the objects from index* 1 *to* $i-1$*. Else the object* i *is either part of the optimal solution or it consists out of the objects* 1 *to* $i-1$*.*

The table of the values and weight of each item with index i is shown on the left and on the right is the table that gets filled in with a dynamic programming approach of the formula above. The maximum bag size is 7.

i	$w[i]$	$v[i]$
1	1	1
2	3	4
3	2	3
4	4	6
5	6	8

5	0	1	3	4	6	7	9	10
4	0	1	3	4	6	7	9	10
3	0	1	3	4	5	7	8	8
2	0	1	1	4	5	5	5	5
1	0	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7

The first column in the table represent the objects and the last row the weight. The entries in the table show the maximum value. In this example the maximum value for the size 7 is 10.

3 System Design

The implementation was done in Java and three different classes were implemented: `main.java` (described in section 3.1), `grammar.java` (described in section 3.2) and `parser.java` (described in section 3.3). The `main`-class calls the methods and the `grammar`-class parses the input grammer ans string into a format that then can be processed in the `parser`-class.

3.1 Main

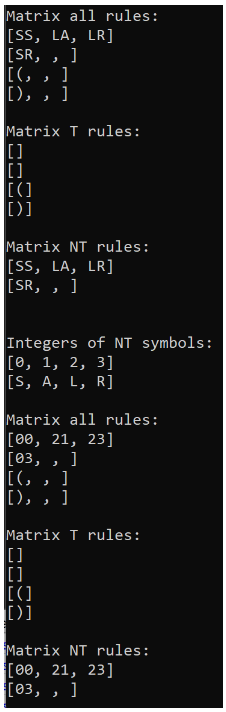
The `main`-class takes the input grammar and word and parses them into `String[]` and `String`. The arguments have to follow the following rules:

- The Grammar needs to be in *CNF*.
- The first rule begins with the start symbol of the grammar.
- The rules are put in wothout arrows, one rule body is represented by one string, beginning with the rule head.
- The last argument is the input word.

Input example (*Well-Balanced-Parantheses*):
`java Main "SSS" "SLA" "SLR" "ASR" "L(" "R)" "(())"`
for the grammar `S` $\rightarrow$ `SS | LA | LR, A` $\rightarrow$ `SR, L` $\rightarrow$ `(, R` $\rightarrow$ `)` and the input word `(())`.
A more detailed description on how to run the code can be found in appendix A.

3.2 Grammar

The `grammar`-class assigns the nonterminal symbols to integers and builds arrays with them. The start symbol for example is then assigned with the integer zero and the bodies of that rule are at the index zero of an two-dimensional array. One array that only contains nonterminal symbols is build, one array that contains terminal and nonterminal symbold and one array that represents the integers that represent each nonterminal symbol is built.
For the input `java Main "SSS" "SLA" "SLR" "ASR" "L(" "R)" "(())"` the two-dimensional arrays shown on the screenshot on the right side are built.
(Right now the arrays still have the type `String[][]`, that will be changed later to `int[][][]` to minimize the access time in the parsing methods.)



3.3 Parser

The `parser`-class contains three different parsing methods. Each methods has a counter as `long` which counts the number of interations and a timer as `long` in ms which measures the runtime of the method.

3.3.1 Naive

The naive algorithm is a recursive algorithm which returns a boolean and has an initial method to start the recursion with the start values:

- The start symbol of the grammar is first parameter. It is an integer called `indexNT` and initialized with 0, because the start symbol is assigned to the index 0.
- The second parameter is the integer 0 as a start index of the input word.
- The third parameter is the integer n, which is the length of the input word.

The start values get assignened in the recursion call which can be seen in the following pseudo code:

Algorithm 1 Recursion call: Boolean parseNaive()

```
counter ← 0
return parseNaive(0, 0, inputWord.length)
```

The naive approach does not use dynamic programming. Instead it checks for each call `parseNaive(indexNT, i, j)` first if i = j-1 and checks if the nonterminal symbol head of `indexNT` leads to a body of a rule with `s[i]`. This is the base case of the recursion which then returns true or false depending if the rule `indexNT` → `s[i]` exists. This base case can be seen in line 2-7 in the pseudo code below.

If i is not equal to j-1 it loops for the integer k from i+1 to j-1 and checks for all rules `A` → `BC` if both calls `parseNaive(B, i, k)` and `parseNaive(C, k, j)` return true. This recursive call applies the function on the substrings of the input word. If such a pair of substrings is found, the function returns true, because the recursive call in combination with the "`and`" leads to the result of the complete word. If such a pair cannot be found the function returns false after looping through all rule bodies and values for k. This part of the recursion can be seen in the pseudo code below in line 9-21.

Algorithm 2 Boolean parseNaive(int indexNT, int i, int j)

```
1:  counter ← counter + 1
2:  if i == (j − 1) then
3:      for l ← 0 to ruleset[0].length do
4:          if ruleset[indexNT][1] == inputWord[i] then
5:              return true
6:          end if
7:      end for
8:  else
9:      for bodyIndex ← 0 to ruleset[indexNT].length do
10:         if ruleset[indexNT][bodyIndex].length >= 2 then
11:             first ← ruleset[indexNT][bodyIndex].charAt(0)
12:             second ← ruleset[indexNT][bodyIndex].charAt(1)
13:             for k ← i + 1 to j do
14:                 if parseNaive(first, i, k) and parseNaive(second, k, j) then
15:                     return true
16:                 end if
17:             end for
18:         end if
19:     end for
20: end if
21: return false
```

In the following table the upper boundary runtime of the second method (algorithm 2) is listed for each line. The variable n represents the length of the input word and k the dimension of the rule array.

Line	Runtime	Type
1	1	Assignment
2	1	Comparison
3	k	Loop
4	1 · k	Comparison
5	1 · k	Return statement
9	k	Loop
10	1 · k	comparison
11	1 · k	Assignment
12	1 · k	Assignment
13	n · k	Loop
14	n · n · k	Recusrsive call
15	1 · n · k	Return statement
21	1	Return statement

Calculating the runtime:

$$\begin{aligned}&1 + 1 + k + k + k + k + k + k + k\\&+ n \cdot k + n \cdot n \cdot k + n \cdot k\\&= 2 + 7k + 2(n \cdot k) + n \cdot n \cdot k\\&= 2 + 7k + 2n \cdot 2k + 2n^2 \cdot k\\&\in O(n^2)\end{aligned}$$

The runtime of the naive method is in $O(n^2)$.

3.3.2 Top-Down

The top-down method is an improves version of the naive method (section 3.3.1). This algorithm works recursive, too. In this algorithm another global array is used, which contains the values `true`, `false` and `null`. The array gets intialized with `null` in each field. That happens in the recursion call method below.

Algorithm 3 Recursion call: Boolean parseTD()

```
1: counter ← 0
2: for i ← 0 to table.length  do
3:     for j ← 0 to table[i].length do
4:         for k ← 0 to table[i][j].length do
5:             table[i][j][k] ← null
6:         end for
7:     end for
8: end for
9: return parseTD(0, 0, inputWord.length)
```

Additional to the naive algorithm, the recursion in the topdown approach starts with another condition: If one of the values in the global table is not `null`, the next recursive call is not executed and this value gets returned. This if-condition can be seen in the following part of the pseudo code. The rest of the topdown approach is the same as the naive method.

Algorithm 4 Additional condition in Boolean parseTD(int indexNT, int i, int j)

```
if table[indexNT][i][j]  != null then
    return table[indexNT][i][j]
end if
```

The variable n represents the length of the input word and k the dimension of the rule array.

Line	Runtime	Type
1	1	Assignment
2	k	Loop
3	k · n	Loop
4	k · n · n	Loop
5	k · n · n · 1	Assignment
9	...	Function call

Calculating the runtime of the method call:

$$1 + k + k \cdot n + k \cdot n \cdot n + k \cdot n \cdot n$$
$$= 1 + k + k \cdot n + 2(n^2 \cdot k)$$
$$\in O(n^2)$$

Algorithm 5 Boolean parseTD(int indexNT, int i, int j)

```
counter ← counter + 1
rulesetLength ← ruleset[0].length
if table[indexNT][i][j] != null then
    return table[indexNT][i][j]
end if
if i == (j − 1) then
    for l ← 0 to ruleset[0].length do
        if ruleset[indexNT][1] == inputWord[i] then
            return true
        end if
    end for
else
    for bodyIndex ← 0 to ruleset[indexNT].length do
        if ruleset[indexNT][bodyIndex].length >= 2 then
            first ← ruleset[indexNT][bodyIndex].charAt(0)
            second ← ruleset[indexNT][bodyIndex].charAt(1)
            for k ← i + 1 to j do
                if parseNaive(first, i, k) and parseNaive(second, k, j) then
                    return true
                end if
            end for
        end if
    end for
end if
return false
```

The runtime analysis of the `parseTD` method is analog to the `parseNaive` runtime analysis. Both algorithms have a runtime in $O(n^2)$.

3.3.3 Bottom-Up

The `parseBU` method works with dynamic programming.

Algorithm 6 Boolean parseBU()

$wordlength \leftarrow word.length$
$String[][]DP \leftarrow newString[wordLength][wordLength]$
for $i \leftarrow 0$ **to** $wordlength$ **do**
  if $ruleset\ contains\ word[i]$ **then**
    $temp \leftarrow indexofword[i]\ in\ ruleset$
    if $DP[i][i]\ != null$ **then**
      $DP[i][i] \leftarrow DP[i][i] + temp$
    else
      $DP[i][i] \leftarrow +temp$
    end if
  end if
end for
for $l \leftarrow 0$ **to** $wordlength$ **do**
  for $i \leftarrow 0$ **to** $wordlength - l$ **do**
    $j \leftarrow i + l$
    for $k \leftarrow 0$ **to** j **do**
      for $head \leftarrow 0$ **to** $ruleset.length$ **do**
        for $body \leftarrow 0$ **to** $ruleset[head].length$ **do**
          $conter \leftarrow counter + 1$
          if $ruleset[head][body].length >= 2$ **then**
            $first \leftarrow ruleset[head][body].charAt(0)$
            $second \leftarrow ruleset[head][body].charAt(1)$
            if $DP[i][k]\ contains\ first$ **and** $DP[k+1][j]\ contains\ second$ **then**
              $temp \leftarrow head$
              $DP[i][j] \leftarrow DP[i][j] + temp$
            end if
          end if
        end for
      end for
    end for
  end for
end for
if $DP[0][wordlength - 1]\ contains\ 0$ **then**
  return true
end if
return false

The variable n represents the length of the input word and k the dimension of the rule array.

Line	Runtime	Type
1	1	Assignment
2	1	Assignment
3	n	Loop
4	n	Comparison
5	n	Assignment
6	n	Comparison
7	n	Assignment
9	n	Assignment
13	n	Loop
14	n · n	Loop
15	n · n	Assignment
16	n · n · n	Loop
17	n · n · n · k	Loop
18	n · n · n · k · k	Loop
19	n · n · n · k · k	Assignment
20	n · n · n · k · k	Comparison
21	n · n · n · k · k	Assignment
22	n · n · n · k · k	Assignment
23	n · n · n · k · k	Comparison
24	n · n · n · k · k	Assignment
25	n · n · n · k · k	Assignment
33	1	Comparison
34	1	Return statement
36	1	Return statement

Calculating the runtime:

$$5 + 7n + 2n^2 + n^3 + n^3 \cdot k + 8 \cdot n^3 \cdot k^2$$
$$5 + 7n + 2n^2 + (k+1)n^3 + 8 \cdot n^3 \cdot k^2$$
$$\in O(n^3)$$

The runtime of the `parseBU` method is in $O(n^3)$.

4 Evaluation

In this section the runtimes and experiments will be compared.

4.1 Experiments

The following tables show some tests that were run with the different parsing methods and grammars. The first of each column for the method shows the truth value that is returned (as T for true and F for false), C represents the counter and T the time in ms.

Well-Balanced-Parantheses:

Word	Length	N	NC	NT	B	BC	BT	T	TC	TT
()	2	T	6	5ms	F	0	2ms	T	6	1ms
(())	4	T	33	4ms	F	84	3ms	T	28	1ms
((()))	6	T	212	6ms	F	360	6ms	T	84	1ms
(((())))	8	T	1295	6ms	F	924	9ms	T	190	1ms
((((()))))	10	T	7666	9ms	F	1872	11ms	T	362	1ms
((((((((((()))))))))))	20	T	51.863.993	1049ms	F	15.732	23ms	T	2.772	3ms
((...))	40				F	127.452	50ms	T	21.743	4ms

The input word with a length of 40 did not terminate for the naive parsing in 60 minutes.

4.2 Runtimes

The runtimes of the methods are calculated in section 3. The results are the following:

Method	Runtime
`parseNaive`	$O(n^2)$
`ParseTD`	$O(n^2)$
`ParseBU`	$O(n^3)$

Seeing those runtimes one could think that the naive and the topdown approach are equally efficient. Considering the differences in the algorithms and the results of the experiments it gets shown that this is not the case. The O-notation runtimes are the upper boundary for the runtime.
Regarding the experiments that were run on the code, the topdown method seems to be the most efficient, followed by the bottomup method. The least efficient method is in these cases the naive approach.

5 Conclusion and Future Work

5.1 Conclusion

Regarding the experiments from section 4, the topdown method seems to be the most efficient, followed by the bottomup method. The least efficient method is in these cases the naive approach.
Depending on the input word and the grammar thenaive and the topdown approach have the same worst case runtime with $O(n^2)$. The bottomup method has the highest worst case runtime with $O(n^3)$.
These algorithms show differences in efficiency but also limits for the examination of words with the CYK-algorithm. The amount of calls raised especially with the Well-Balanced-Parantheses language exponentially. This can also be seen in the graphs in appendix B.

5.2 Future work

Formal languages and grammars can be used in different use cases. For exampel for AI learning methods. Real languages like English can be defined as a formal language, too. But it is important to consider that formal languages only define the syntax and not the semantics. In addition to this has english a lot of rules and many exceptions. [4]

A How to use the code?

The code can be run in the terminal and input is expected as Strings in quotation marks. The grammar needs to be in CNF. The first rule begins with the startsymbol of the grammar.

First: Rules without arrows (one rule as one String)
Last: The last argument is the input word

Input example (*Well-Balanced-Parantheses*):
`java Main "SSS" "SLA" "SLR" "ASR" "L(" "R)" "(())"`
for the grammar `S` $\rightarrow$ `SS | LA | LR, A` $\rightarrow$ `SR, L` $\rightarrow$ `(, R` $\rightarrow$ `)` and the input word `(())`.

Output example:
The first part of the output shows the arrays, which get generated in the `Grammar.java` class.

The first array contains all rules.

The second array contains only the terminal rules.

The third array contains only the nonterminal rules.

Then it is shown which nonterminal symbols are represented by which integers. Later the nonterminal symbols can be referred to with those integers.

After this the mentioned arrays are shown again but the nonterminal symbols got replaced with the according integers.

```
Matrix all rules:
[SS, LA, LR]
[SR, , ]
[(, , ]
[), , ]

Matrix T rules:
[]
[]
[(]
[)]

Matrix NT rules:
[SS, LA, LR]
[SR, , ]


Integers of NT symbols:
[0, 1, 2, 3]
[S, A, L, R]

Matrix all rules:
[00, 21, 23]
[03, , ]
[(, , ]
[), , ]

Matrix T rules:
[]
[]
[(]
[)]

Matrix NT rules:
[00, 21, 23]
[03, , ]
```

```
Input word: (())

Naive: true   Amount of calls: 33
Naive runtime: 9ms

CYK-Table (Bottom Up):
[2, , , ]
[, 2, 0, 1]
[, , 3, ]
[, , , 3]

BottomUp: false   Amount of calls: 84
Naive runtime: 4ms

TopDown: true   Amount of calls: 28
Naive runtime: 1ms
```

Then the results, counter and runtime in ms is shown for each parsing method.
For the `BottomUp` method is the CYK algorithm table printed.

B Graphs and additional results

The following tables show some tests that were run with the different parsing methods and grammars. The first of each column for the method shows the truth value that is returned (as T for true and F for false), C represents the counter and T the time in ms. Those tests were run before the BottomUp method was working correctly.

Well-Balanced-Parantheses:

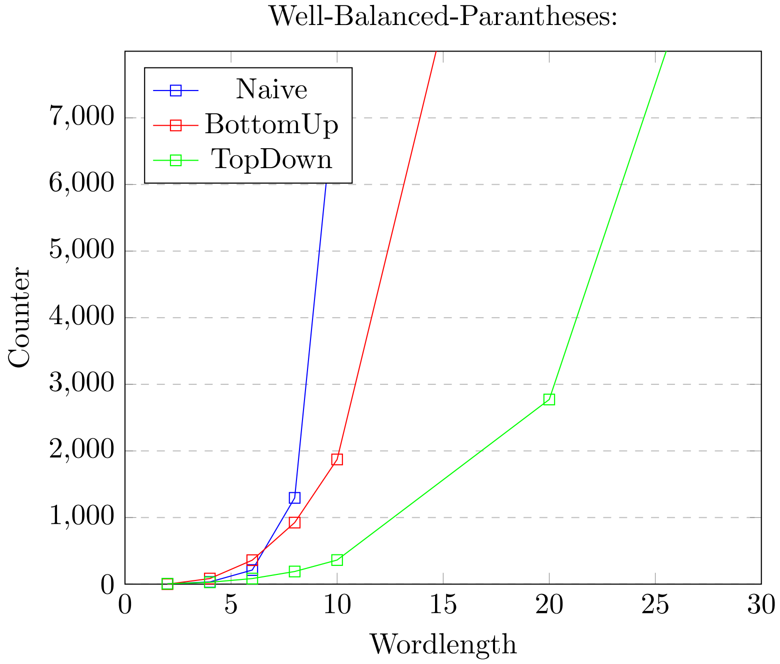
Word	Length	N	NC	NT	B	BC	BT	T	TC	TT
()	2	T	6	5ms	F	0	2ms	T	6	1ms
(())	4	T	33	4ms	F	84	3ms	T	28	1ms
((()))	6	T	212	6ms	F	360	6ms	T	84	1ms
(((())))	8	T	1295	6ms	F	924	9ms	T	190	1ms
((((()))))	10	T	7666	9ms	F	1872	11ms	T	362	1ms
(((((((((())))))))))	20	T	51.863.993	1049ms	F	15.732	23ms	T	2.772	3ms
((...))	40				F	127.452	50ms	T	21.743	4ms

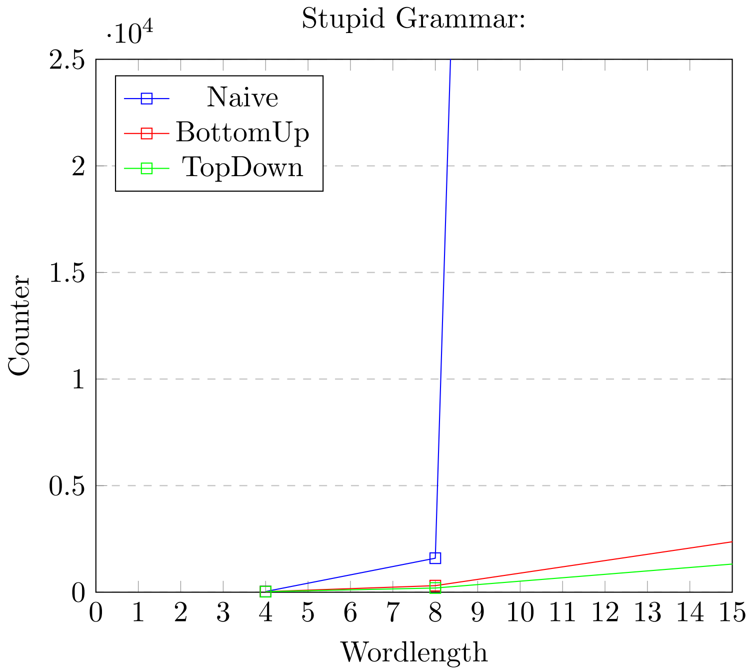
The input word with a length of 40 did not terminate for the naive parsing in 60 minutes.

Stupid Grammar:

Word	Length	N	NC	NT	B	BC	BT	T	TC	TT
aaaa	4	F	33	5ms	F	28	6ms	F	27	1ms
aaaaaaaa	8	F	1.596	8ms	F	308	5ms	F	197	1ms
aaaaaaaaaaaaaaaa	16	F	3.524.577	89ms	F	2.660	15ms	F	1.481	1ms

Graphs:


Well-Balanced-Parantheses:
Naive
BottomUp
TopDown
Counter
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
0
5
10
15
20
25
30
Wordlength


Stupid Grammar:
·10^4
Naive
BottomUp
TopDown
Counter
0
0.5
1
1.5
2
2.5
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Wordlength

C Calculations

C.1 CNF Algorithm on an example

In this part, the following ruleset of a grammar will be translated into *CNF*.

$$\texttt{S} \rightarrow \texttt{SS} \mid \texttt{aSb} \mid \texttt{ab}$$

1. Remove every nonterminal symbol that cannot be reached or is not generating another symbol:
   `S` is the only nonterminal symbol and does not need to be removed.

2. Remove all symbols that cannot be reached. (All symbols can be reached.)

3. Replace the terminal symbols in the body of other rules with new nonterminal symbols to not have bodies which contain terminal and nonterminal symbols:

$$\begin{aligned}
&\texttt{S} \rightarrow \texttt{SS} \mid \texttt{LSR} \mid \texttt{LR}\\
&\texttt{L} \rightarrow \texttt{a}\\
&\texttt{R} \rightarrow \texttt{b}
\end{aligned}$$

4. On the right side of the rules are only two nonterminal symbols allowed:

$$\begin{aligned}
&\texttt{S} \rightarrow \texttt{SS} \mid \texttt{LA} \mid \texttt{LR}\\
&\texttt{A} \rightarrow \texttt{SR}\\
&\texttt{L} \rightarrow \texttt{a}\\
&\texttt{R} \rightarrow \texttt{b}
\end{aligned}$$

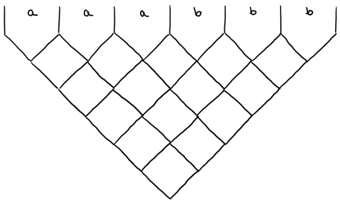
5. Remove all ϵ-rules and paste in the start symbol what can be generated by them. (This grammar does not have ϵ-rules.)

6. Check on transitivity and remove those dependencies. (There are no transitive rules here.)

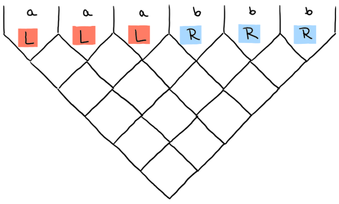
This is the grammar in CNF:

$$\begin{aligned}
&\texttt{S} \rightarrow \texttt{SS} \mid \texttt{LA} \mid \texttt{LR}\\
&\texttt{A} \rightarrow \texttt{SR}\\
&\texttt{L} \rightarrow \texttt{a}\\
&\texttt{R} \rightarrow \texttt{b}
\end{aligned}$$

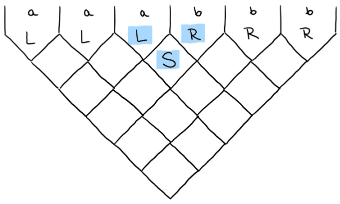
C.2 CYK Algorithm on an example



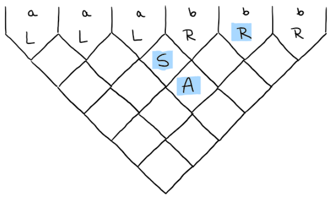
$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$



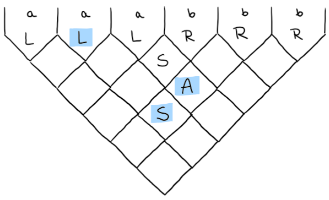
$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$



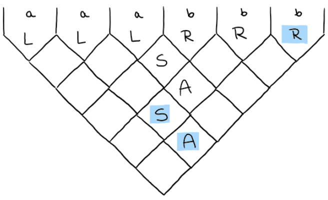
$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$



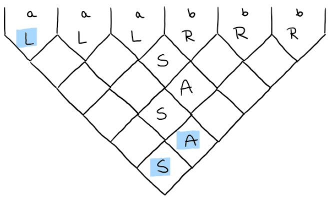
$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$



$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$



$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$



$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$

References

[1] *Chomsky's Normal Form (CNF)*. Website. `https://www.javatpoint.com/automata-chomskys-normal-form`, opened on 26.09.2022.

[2] *Dynamic Programming*. Website. `https://www.programiz.com/dsa/dynamic-programming`, opened on 26.09.2022.

[3] Robert Eisele. *A CYK Algorithm Visualization*. May 2018. URL: `https://www.xarg.org/tools/cyk-algorithm/`.

[4] Maggie Johnson and Julie Zelenski. *Formal Grammars*. Website. `https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts`, opened on 26.09.2022. 2012.

[5] Glenn K. Manacher. "An improved version of the Cocke-Younger-Kasami algorithm". In: *Computer Languages* 3.2 (1978), pp. 127–133. ISSN: 0096-0551. DOI: `https://doi.org/10.1016/0096-0551(78)90029-2`. URL: `https://www.sciencedirect.com/science/article/pii/0096055178900292`.

[6] A.J. Kfoury Robert N. Moll Michael A. Arbib. *An Introduction to Formal Language Theory*. Springer-Verlag, 1988.

[7] Fabio Massimo Zanzotto, Giorgio Satta, and Giordano Cristini. "CYK Parsing over Distributed Representations". In: *Algorithms* 13 (Oct. 2020), p. 262. DOI: `10.3390/a13100262`.