Umeå University

Efficient Algorithms

Report

Runtime analysis of the Java implementation of the CYK-algorithm

Pina Kolling

March 23, 2023

Abstract

Formal languages and grammars can be used in different fields, for example for AI learning methods [3, 11]. The CYK-algorithm is a method to examine if an input word is part of a formal language. In this report, the implementation of different parsing methods for the CYK-algorithm is described. The runtimes of those methods are analysed and compared. For each method experiments with different inputs are documented and evaluated. The results of the experiments and the theoretical runtime analysis in O-notation are compared to show the differences and similarities. The results are also shown in graphs. In addition to this, one method is modified to work with linear grammars and concepts of error correction are described and presented.

Contents

1 Introduction

Parsing in Computer Science is the process of reading, analysing and processing an input string. In this context it is used for 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 out of characters from a formal 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 2.5. To use the *CYK*-algorithm, the grammar needs to be in a specific format, that is called *Chomsky-Normal-Form* (short: *CNF*) [9, 15], which is explained in Section 2.3.

Three different parsing methods are implemented and described of which one executes the *CYK*-algorithm with dynamic programming in *Java*. The different parsing methods are 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 grammar as string into a format that then can be processed in the `parser`-class. This `parser`-class has three different parsing methods, which are tested and compared against each other. The function and implementation is former described in Section 3. Also in Section 3 the runtimes in O-notation are calculated for each approach and modifications for the parsing of linear grammars are described.

In Section 5 the runtimes and experiments of the different algorithms are compared and differences in efficiency are shown. The results from the experiments are compared to the calculated runtimes in O-notation from Section 3 and those differences are explained.

Concepts, ideas and first steps for the error correction of input words with deletion and exchange are presented in Section 4.

In the final part (Section 6) the results and future possibilities are discussed.

2 Background

In this section background information, definitions and examples on formal languages, alphabets and Kleene star (Section 2.1), formal grammars (Section 2.2), Chomsky-Normal-Form (Section 2.3), CYK-algorithm (Section 2.5), dynmaic programming (Section 2.4), recursion (Section 2.6) and divide and conquer (Section 2.7) are 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, which contains all the possible characters of the words. Those characters are called nonterminal symbols [1, 10].

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.*

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

Example (Formal language)**.** [1] *The language accepts words that contain the same amount 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 terminal symbols are the characters of the alphabet of the language and the nonterminal symbols are used to build the rules of the language – they get replaced by terminal symbols [1, 10] .

[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 assumes that $V \cap T = \emptyset$. The set P is the set of production rules, where an element of V points to an element of $V^* \times T^*$. The symbol $S \in V$ represents the start symbol.

The production rules in the set P have the format `head` $\rightarrow$ `body`*. The head is always a nonterminal symbol and the body can be a combination of terminal and nonterminal symbols. A nonterminal symbol* `A` *in the body of a rule can be replaced by the body of a rule with the form* `A` $\rightarrow$ `body` *[13].*

A formal grammar defines which words over the alphabet T/Σ are contained in the associated laguage.

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

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

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` $\rightarrow$ `SS` | `aSb` | `ab`

The start symbol `S` *can for example be replaced with its body* `SS`*. If then both symbols* `S` *are replaced with the body* `ab` *the resulting word is* `abab`*. It is part of the language, because it was build with the grammar G.*

2.3 Chomsky-Normal-Form

The *Chomsky-Normal-Form* (short: *CNF*) is a grammar which is formatted in a specific way. If the head of a rule (nonterminal symbol) is not generating the empty word (`S` $\rightarrow \epsilon$) 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 built can be seen in Appendix B.*

`S` $\rightarrow$ `SS` | `LA` | `LR`

`A` $\rightarrow$ `SR`

`L` $\rightarrow a$

`R` $\rightarrow b$

2.4 Dynamic programming

The technique of dynamic programming can be used to solve problems, which can be divided into smaller subproblems. The solutions of the subproblems are saved (for example in a multi-dimensional array) and referenced later [4].

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 following:

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

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

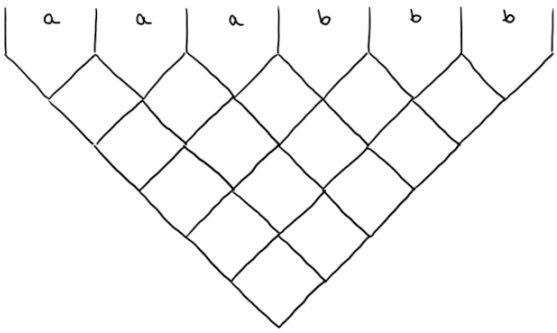
The first column in the table represent the objects and the last row the weight. The entries in the table show the maximum value. In this example the maximum value for the size 7 is 10.

2.5 CYK-algorithm

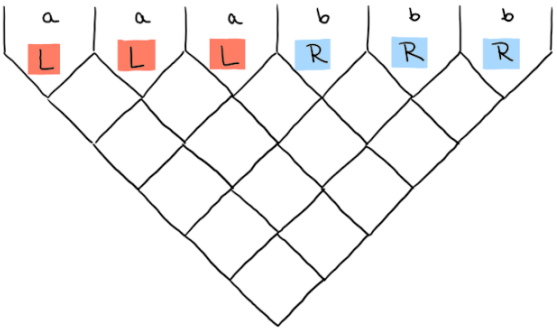
Cocke-Younger-Kasami-algorithm (short: *CYK*-algorithm) takes a grammar $G = (V, T, P, S)$ in *CNF* and a word $w = w_1, w_2, ..., w_n \in T^*$ as an input. It then examines if the word follows

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 [15].

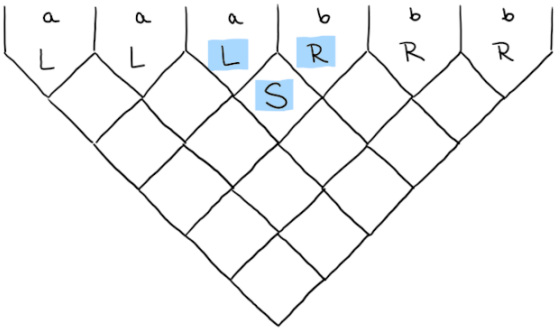
Example (Chomsky-Normal-Form)**.** *The following table shows the CYK-algorithm with the previous grammar example and the input word aaabbb. The word is accepted by the grammar rules, because the initiating nonterminal symbol* `S` *can be filled into the lowest field. The table is filled in step by step and the rules that are used in each step are marked, for a better understanding on how the algorithm works [5].*



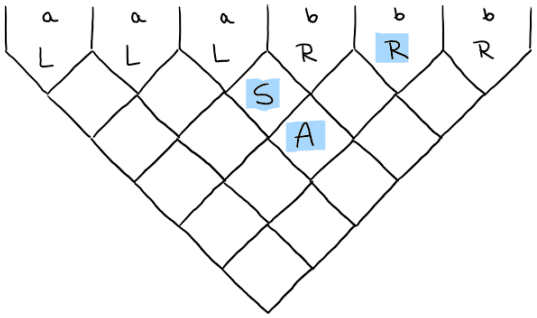
$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$



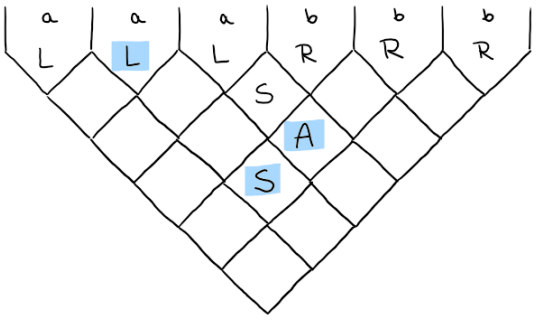
$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$



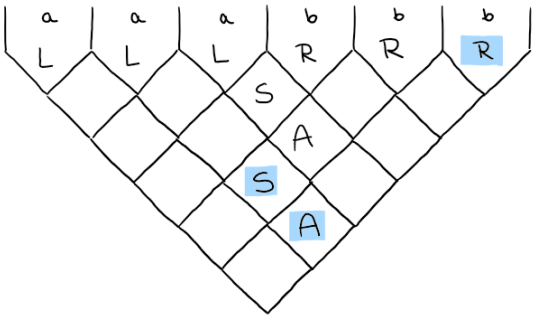
$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$



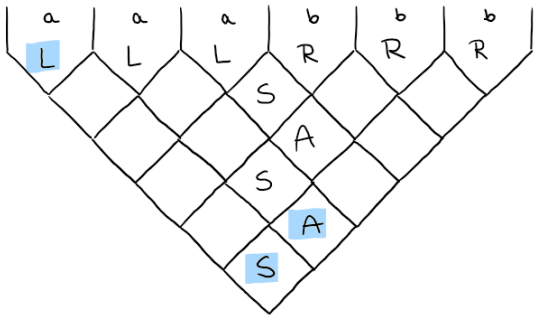
$S \to SS \mid LA \mid LR$
$A \to SR$
$L \to a$
$R \to b$



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



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



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

2.6 Recursion

A function that calls itself is called recursive function. In computer science recursive functions use base cases to terminate the program and stopping it from going on forever. Base cases are problems that can be solved without any more recursive calls [7].

Example (Recursion)**.** *For definition and examples for recursion read Section 2.6.*

Example (Recursion)**.** *Building a sum for the numbers* $1 + 2 + ... + n$.

Nonrecursive approach:

$$f(n) = 1 + 2 + ... + n$$

Recursive approach:

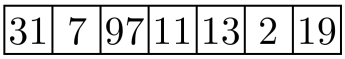
$$\begin{aligned} f(n) &= 1 && \text{if } n = 1 \\ f(n) &= n + f(n-1) && \text{if } n > 1 \end{aligned}$$

2.7 Divide and conquer

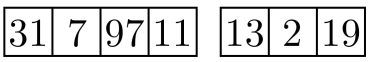
The divide and conquer method is used to design efficient algorithms. In this method, the problem is divided recursively into smaller and simpler subproblems, until those subproblems can be solved. Then those subproblems are combined again to give the solution of the original problem.

Example (Divide and conquer)**.** *A problem that can be solved with divide and conquer is the sorting of an array. The array is then divided in smaller arrays, until the array size is one. Then the subarrays are merged together in the correct order.*

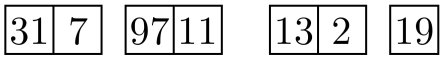
The unsorted array for the example is the following:



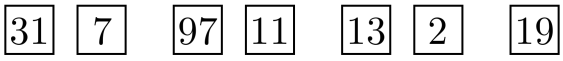
The array gets divided into two smaller subarray, which are sorted with the same technique individually and later merged together again:



Those subarrays get divided into even smaller arrays again:



The array size of the subproblems is now 1:



The subarrays get merged together in the right order:

For each merge of the subarrays only the smallest symbols of each array need to be compared. The smaller one is then sorted into the merged array and again the smallest symbols are compared:

| 7 | 11 | 31 | 97 |

| 2 | 13 | 19 |

The array is complete and sorted:

| 2 | 7 | 11 | 13 | 19 | 31 | 97 |

2.8 Master Theorem

The master theorem provides an analysis for the runtime of divide and conquer algorithms in O-notation. A problem of size n is divided into a subproblems of size $\frac{n}{b}$ each and $O(n^k)$ further computations are done to combine the solutions of the subproblems [12].

$$T(n) = a \cdot T(\frac{n}{b}) + O(n^k)$$

It solves to the following upper bounds of the runtime:

$$a < b^k \ \in O(n^k) \tag{1}$$
$$a = b^k \ \in O(n^k \log n) \tag{2}$$
$$a > b^k \ \in O(n^{\log_b a}) \tag{3}$$

For the previous sorting example in Section 2.7, the problem is divided into 2 subproblems ($a = 2$) of size $\frac{n}{2}$ each ($b = 2$). The recombining consists out of comparisons which can be expressed as $k = 1$. This leads to the following formular:

$$T(n) = 2 \cdot T(\frac{n}{2}) + n^1$$

Because a equals b^1, it can be resolved to the second case: The algorithm is in $O(n \log n)$.

3 Three variants of CYK algorithm

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 grammar and 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`. A 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 arrays have the type `Integer[][][]`. The start symbol for example is then assigned with the integer zero and the bodies of that rule are at the index zero of the three-dimensional array. The first dimension represents the head of a rule, the second dimension represents the body which is divided into two arrays – for each symbol of the body one array.

For faster and easier access, three arrays are built: One array that only contains nonterminal symbols and rules, one array that contains only terminal rules and one array that contains both.

3.3 Parser

The `parser`-class contains three different parsing methods: `parseNaive` (described in Section 3.3.1), `parseTopDown` (described in Section 3.3.3) and `parseBottomUp` (described in Section 3.3.6). Also there exists a modified method of the `parseTopDown`, which takes linear grammars (which are translated into *CNF*) and parses them faster (described in Section 3.3.5).

Each method has a counter as `long` which counts the number of iterations and a timer as `long` in ms which measures the runtime of the method. In the following sections the algorithms are described, presented as pseudo code and the runtime is analysed.

3.3.1 Naive description

The naive algorithm is a recursive divide-and-conquer algorithm. The problem gets divided into smaller subproblems and then the method is called again for each subproblem. It 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 assigned in the recursion call which can be seen in the following pseudo code:

Algorithm 1 Recursion call: parseNaive()

```
1: counter ← 0
2: 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` $\rightarrow$ `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` $\rightarrow$ `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 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][l][0] == inputWord[i]
               or ruleset[indexNT][l][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][0] != null
               and ruleset[indexNT][bodyIndex][1] != null then
11:            first ← ruleset[indexNT][bodyIndex][0]
12:            second ← ruleset[indexNT][bodyIndex][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
```

```
        end if
    end for
end if
return false
```

3.3.2 Naive runtime

In the following table the upper bound 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.

The parser has a recursive call in a for loop with $n \cdot k$ iterations. At each recursive call of the parsing method the for loop is halved and the recursive call is doubled. This can be visualized as a recursion tree in which each node has two children, which can be seen in the recursion tree in figure 1.

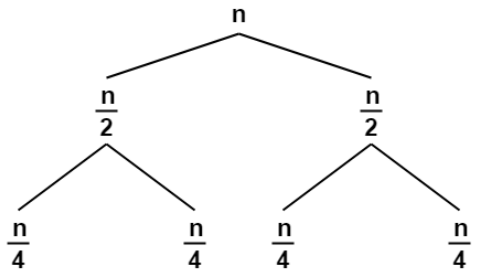


Figure 1: Recursion tree [14]

The root node of the tree represents the initial call of the parser with the entire input string of length n. Each level of the tree represents a recursive call, where the number of children is 2, because that is the number of recursive calls. Because of this, the number of nodes in the tree is 2^n. This results in a total of 2^n recursive calls.

Line	Runtime	Type
1	1	Assignment
2	1	Comparison
3	k	Loop
4	$1 \cdot k$	Comparison
5	$1 \cdot k$	Return statement
9	k	Loop
10	$1 \cdot k$	comparison
11	$1 \cdot k$	Assignment
12	$1 \cdot k$	Assignment
13	$n \cdot k$	Loop
14	2^n	Recursive call
15	$1 \cdot n \cdot k$	Return statement
21	1	Return statement

Calculating the runtime:

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

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

3.3.3 Top-Down description

The topdown method is an improved version of the naive method (Section 3.3.1). This algorithm works recursive and with the concept of divide and conquer, too. In this algorithm another global array is used, which contains the values `true`, `false` and `null`. The array gets initialized with `null` in each field. That happens in the recursion call method below.

Algorithm 3 Recursion call: 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 a similar approach to the naive method that was already described in Section 3.3.1.

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

```
1: if table[indexNT][i][j] != null then
2:     return table[indexNT][i][j]
3: end if
```

The complete topdown algorithm is shown as pseudo code below. In the inner for-loop (line 18) the boolean value of the next method calls get assigned into the global table. If there is assigned a true (line 19) the value true gets returns (line 20). The general topdown approach is the same approach as the naive method that is described in Section 3.3.1.

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

```
1: counter ← counter + 1
2: rulesetLength ← ruleset[0].length
3: if table[indexNT][i][j] != null then
4:     return table[indexNT][i][j]
5: end if
6: if i == (j − 1) then
7:     for l ← 0 to ruleset[0].length do
```

```
8:        if ruleset[indexNT][l][0] == inputWord[i]
              or ruleset[indexNT][l][1] == inputWord[i] then
9:            return true
10:       end if
11:   end for
12: else
13:   for bodyIndex ← 0 to ruleset[indexNT].length do
14:       if ruleset[indexNT][bodyIndex][0] ! = null
              and ruleset[indexNT][bodyIndex][1] ! = null then
15:           first ← ruleset[indexNT][bodyIndex][0]
16:           second ← ruleset[indexNT][bodyIndex][1]
17:           for k ← i + 1 to j do
18:               table[indexNT][i][j] ← (parseTD(first, i, k) and parseTD(second, k, j))
19:               if table[indexNT][i][j] then
20:                   return true
21:               end if
22:           end for
23:       end if
24:   end for
25: end if
26: return false
```

3.3.4 Top-Down runtime

The topdown parser performs two recursive calls in each iteration. In each recursive call, it is checked, whether the result for the current parameters is already stored in the memoization table and if so, the value is returned. If not, the parser computes the result by checking the grammar rules and recursively calling itself for each possible combination of symbols that could lead to the substring between the indices i and j in the input.

The recursive call is made twice in each iteration in the innermost loop (line 18). Therefore, the number of recursive calls made by the function is proportional to the number of iterations of the innermost loop.

The amount of work of each recursive call depends on the input grammar and the input string. In the worst case, all possible combinations of non-terminal symbols need to be explored. That leads to a time complexity of the function is in $O(n^3)$.

3.3.5 Top-Down modified for linear grammars

For the parsing of linear grammars, the topdown approach (Section 3.3.3) was modified to reduce the amounts of recursive calls. The topdown approach was chosen, because it seems to have the best performance of the different parsing methods, considering the experiments.

The modified algorithm can work more efficient, because for each rule only one recursive call is made instead of two necessary.

For each rule it is checked if the first or the second element of the body is a terminal symbol. If it is a terminal symbol, the current position on the input word gets compared to this symbol and gets set to true if they are equal. This means, that the other symbol has to be a nonterminal symbol. For this nonterminal symbol, the method gets called recursively. Both statement get connected with an `and`, to fill in the memoization table.

Algorithm 6 parseLinearTD(int indexNT, int i, int j)

1: $counter \leftarrow counter + 1$
2: $rulesetLength \leftarrow ruleset[0].length$
3: **if** $table[indexNT][i][j] != null$ **then**
4: **return** $table[indexNT][i][j]$
5: **end if**
6: **if** $i == (j - 1)$ **then**
7: **for** $l \leftarrow 0$ **to** $ruleset[0].length$ **do**
8: **if** $ruleset[indexNT][l][0] == inputWord[i]$ **and** $rulesetLinear[indexNT][l][1] == null$ **then**
9: $tableLinear[indexNT][i][j] \leftarrow true$
10: **return** true
11: **end if**
12: **end for**
13: **return** false
14: **end if**
15: **for** $bodyIndex \leftarrow 0$ **to** $ruleset[indexNT].length$ **do**
16: **if** $ruleset[indexNT][bodyIndex][0] != null$ **and** $ruleset[indexNT][bodyIndex][1] != null$ **then**
17: $first \leftarrow ruleset[indexNT][bodyIndex][0]$
18: $second \leftarrow ruleset[indexNT][bodyIndex][1]$
19: **for** $k \leftarrow i + 1$ **to** j **do**
20: **if** *second* is head of a terminal rule **then**
21: $table[indexNT][i][j] = parseLinearTD(first, i, k)$ **and** *second* is head of terminal rule with body $inputAsInt[k]$
22: **end if**
23: **if** *first* is head of a terminal rule **then**
24: $table[indexNT][i][j] = parseLinearTD(second, k, j)$ **and** *first* is head of terminal rule with body $inputAsInt[i]$
25: **end if**
26: **end for**
27: **end if**
28: **end for**

29: **if** $tableLinear[indexNT][i][j] != null$ **and** $tableLinear[indexNT][i][j] == true$ **then**
30: **return** true
31: **end if**
32: $tableLinear[indexNT][i][j] \leftarrow false$
33: **return** false

Runtime:

The runtime of the modified version is in $O(n^2)$, where n is the length of the input string.

The parser takes three input parameters: $indexNT$, i, and j, which represent the non-terminal, the starting index, and the ending index of a substring of the input string. The length of this substring is $j - i$, which is at most n.

The parser checks if the result for this substring has already been computed and stored in the memoization table. If so, it returns the stored result, which takes constant time.

If the result has not been computed, one of two operations is done: If the substring is a single terminal symbol, the parser iterates over the rules for the non-terminal and checks if there is a rule that generates this terminal symbol. This loop takes $O(k)$ time, with k as the number of rules for the non-terminal. If the substring is more than one terminal symbol, the parser iterates over the rules and checks if there is a rule which can generate the non-terminal symbol and the substring by recursively calling itself on its two subparts. This loop also takes $O(k)$ time.

The recursive calls are made on substrings that are one symbol shorter than the original substring. Therefore, the maximum number of recursive calls is proportional to the length of the input string, which is n.

The overall time complexity of the linear topdown parser is $O(n^2)$.

3.3.6 Bottom-Up description

The `parseBU` method works with dynamic programming. First a table DP with the size $n \times n$ gets constructed (line 2) – n represents the input word length. In the first step the integers that represent the nonterminal symbols that are head of a terminal rule get assigned (line 3 - 12). For the character at index i of the input word the value of $DP[i][i]$ gets the nonterminal symbol that leads to the character of the word. The approach of dynamic programming calculates the smallest substrings first and saves them for the efficient processing of the next bigger substrings. The smallest substrings are the single characters.

Algorithm 7 parseBU()

1: $wordlength \leftarrow word.length$
2: $Integer[][][]DP \leftarrow newInteger[wordLength][wordLength][wordlength]$
3: **for** $i \leftarrow 0$ **to** $wordlength$ **do**
4: **if** $ruleset\ contains\ word[i]$ **then**

```
5:        assign nonterminal symbols of inputword[i] to DP[i][i]
6:    end if
7: end for
8: for l ← 0 to wordlength do
9:    for i ← 0 to wordlength − l do
10:       j ← i + l
11:       for k ← 0 to j do
12:          for head ← 0 to ruleset.length do
13:             for body ← 0 to ruleset[head].length do
14:                conter ← counter + 1
15:                if ruleset[indexNT][bodyIndex][0] != null
                  and ruleset[indexNT][bodyIndex][1] != null then
16:                   first ← ruleset[head][body].charAt[0]
17:                   second ← ruleset[head][body][1]
18:                   if DP[i][k] contains first and DP[k + 1][j] contains second then
19:                      assign head to DP[i][j][c]²
20:                   end if
21:                end if
22:             end for
23:          end for
24:       end for
25:   end for
26: end for
27: if DP[0][wordlength − 1] contains 0 then
28:    return true
29: end if
30: return false
```

In line 13 to 32 the *CYK*-algorithm gets executed. It is described in section 2.5. For each field the two fields *leading* to it get compared, to then fill in the head of a rule that concludes into the both compared nonterminal symbols.

In the last lines (line 33-35) the last field of the DP-array gets checked. If it containes the start symbol of the grammar, the word is contained in the language and the algorithm returns true. Else false is returned.

3.3.7 Bottom-Up runtime

In the following part the upper bound runtime of the bottomup algorithm gets analysed. The variable n represents the length of the input word and k the dimension of the rule array.

[2] c is the first empty position in $DP[i][j]$

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 $O(n^3)$, because it contains three nested loops. The outer loop and the middle loop iterate n times and the inner loop iterates j times, which depends on n, too. Because of this, the number of iterations is $n \cdot n \cdot n$ which is $O(n^3)$.

4 Error correction

Apart from checking if an input word is part of a language, the amount of errors and the correction of those can be interesting. Two options for the error correction are considered in this section: A symbol can be exchanged or symbols can be deleted. The output for some examples of the error correction can be seen in Appendix A.

For the error correction the bottomup parser modified, because the CYK-table shows, which substrings lead to which nonterminal symbols, for example the start symbol.

4.1 Deletion

For the deletion the table of the CYK-algorithm is used. This is described in Section 2.5 and the parsing method for it is explained in Section 3.3.6. The CYK-table has to contain the start symbol in the entry on the right top corner to conclude that the word is part of the language. To find an accepted word with deletion, the entry with the start symbol which is closest to the right top is searched.

The concept on how to find the entry in the array that is further on the top right is described here and shown in the pseudo code of the function `findBestIndeces` in algorithm 8. The function takes two array positions as (i, j) as input and returns the better position in an int array of the length 2.

Algorithm 8 findBestIndeces(int i_a, int j_a, int i_b, int j_b)

1: $int[]\ betterIndeces \leftarrow new\ int[2]$
2: $int\ indicator_i \leftarrow i_a - i_b$
3: $int\ indicator_j \leftarrow j_b - j_a$
4: **if** $indicator_i + indicator_j) == 0$ **then**
5: ▷ both are equal far away
6: $betterIndeces[0] \leftarrow i_a$
7: $betterIndeces[1] \leftarrow j_a$
8: **end if**
9: **if** $indicator_i + indicator_j) > 0$ **then**
10: ▷ b is closer
11: $betterIndeces[0] \leftarrow i_b$
12: $betterIndeces[1] \leftarrow j_b$
13: **else**
14: ▷ a is closer
15: $betterIndeces[0] \leftarrow i_a$
16: $betterIndeces[1] \leftarrow j_a$
17: **end if**
18: **return** $betterIndeces$

The parameters for the functions are the i and j values for the two positions A and B in the array as A_i, A_j, B_i, B_j. The searched position in the array is a small i with a large j.

Two values are stored for later comparisons: $x = A_i - B_i$ and $y = B_j - A_j$. If x and y are equal, the positions are equally far away from the top right array entry. If $x + y$ is positive, B is the preferred position, else it is A.

With those indices the part of the word that is accepted and the part that needs to be deleted can be calculated. This is done in the method `solveErrorWithDeletion` which is shown as pseudo code in algorithm 9. This concludes in the longest accepted word with deletion, which is then returned.

It can happen that no accepted word is found with deletion, if no subword concluded into the start symbol in the CYK-table.

The distance of the entry in the CYK-table that contains the start symbol and the top right corner can also be returned as the amount of symbols that need to be deleted to receive an accepted word. This error counter works only for the deletion method.

Algorithm 9 solveErrorWithDeletion(Integer[][][] CYK, int amountOfErrors)

```
1:  int i ← −1
2:  int j ← −1
3:  int[] betterIndeces ← new int[2]
4:  for index ← 0 to CYK.length do
5:      for entries ← 0 to CYK[amountOfErrors][index].length do
6:          if CYK[amountOfErrors][index][entries] == 0 then
7:              betterIndeces ← findBestIndeces(i, j, amountOfErrors, index)
8:              i ← betterIndeces[0]
9:              j ← betterIndeces[1]
10:         end if
11:         temp ← CYK.length − 1 − amountOfErrors
12:         if CYK[index][temp][entries] == 0 then
13:             betterIndeces ← findBestIndeces(i, j, index, temp)
14:             i ← betterIndeces[0]
15:             j ← betterIndeces[1]
16:         end if
17:     end for
18: end for
19: if i == −1 then
20:     Integer[] noAcceptedWord ← new Integer[1]
21:     noAcceptedWord[0] ← −1
22:     return noAcceptedWord
23: end if
```

```
24: int from ← Math.min(i, j)
25: int to ← Math.max(i, j)
26: Integer[]acceptedWord ← new Integer[to − from + 1]
27: if from == to then
28:     for k ← 1 to inputAsInt.length do
29:         if inputAsInt[k]! = null and inputAsInt[k] == 0 then
30:             acceptedWord[k] ← inputAsInt[k]
31:             break
32:         end if
33:     end for
34:     return acceptedWord
35: end if
36: for k ← 1 to to − from do
37:     if inputAsInt[k + from]! = null then
38:         acceptedWord[k] ← inputAsInt[k + from]
39:     end if
40: end for
41: return acceptedWord
```

4.2 Exchange

The exchange method works iterative and iterates over every symbol of the input word and tries ever terminal symbol on each index. The method can only exchange one symbol. First, the symbol and each entry in the CYK-table that concluded from this symbol in the input word are deleted. For this, the corresponding entries are replaced with -1. Then the fields that are assigned with -1 in the CYK-table get calculated with a new input symbol, which replaces the original one. If a success is found, the accepted word is returned.

In algorithm 10 the adapted version of the bottomUp parser can be seen as pseudo code. In this methos, a symbol from the input word got replaced by a new symbol (which is the parameter `newSymbol`). All the entries, that relied on this symbol were replaced by -1 before and this version of the bottomUp parser examines the new entries of these field to produce a full CYK-table for the new input word without calculating the already known parts again.

Algorithm 10 parseBUnewSymbol(Integer[][][] DP, int newSymbol)

```
1: wordlength ← word.length
2: for i ← 0 to wordlength do
3:     if DP[i][i][0]! = null and DP[i][i][0] == −1 then
4:         DP[i][i][0] ← newSymbol
5:     end if
6: end for
7: for l ← 0 to wordlength do
```

```
    for i ← 0 to wordlength − l do
        j ← i + l
        if DP[i][j][0]! = null and DP[i][j][0] == −1 then
            for k ← 0 to j do
                for head ← 0 to ruleset.length do
                    for body ← 0 to ruleset[head].length do
                        conter ← counter + 1
                        if ruleset[indexNT][bodyIndex][0] ! = null
                and ruleset[indexNT][bodyIndex][1] ! = null then
                            first ← ruleset[head][body].charAt[0]
                            second ← ruleset[head][body][1]
                            if DP[i][k] contains first and DP[k + 1][j] contains second
    then
                                assign head to DP[i][j][c]
                            end if
                        end if
                    end for
                end for
            end for
        end if
    end for
end for
if DP[0][wordlength − 1] contains 0 then
    return true
end if
return false
```

5 Evaluation

In this section the calculated runtimes in O-notation from Section 3 and experiments are compared. First, in section 5.1, the results of different grammars and input words are presented in tables and as graphs. In section 5.2 the results of section 5.1 and the calculated runtimes in O-notation from Section 3 are compared.

More information and definitions on the O-notation can be read in *Big O Notation* from P. Danziger [2].

5.1 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:
`"SSS" "SLA" "SLR" "ASR" "L(" "R)" "(*)*"`

The Well-Balanced-Parantheses grammar accepts words, that contain a closed bracket `)` for each opened bracket `(`. The closed bracket has to appear after the accorsing opened bracket.

Different length and combinations of accepted input words were evaluated by the different parsing methods and the tables and graphs are shown below.

In the following table and graphs, variations of the input word `(*)*` were evaluated. The abbreviations starting with N stand for the naive parser, the B represents the bottom-up parser and the T for topdown. The column with these abbreviations shows the returned truth value of the parsing method. The second letter of the abbreviations represents the counter of calls/iterations with the letter C and the measured runtime is represented with the letter T.

Word	Length	N	NC	NT	B	BC	BT	T	TC	TT
()	2	T	6	5ms	T	0	2ms	T	6	1ms
(())	4	T	33	4ms	T	84	3ms	T	28	1ms
((()))	6	T	212	6ms	T	360	6ms	T	84	1ms
(((())))	8	T	1295	6ms	T	924	9ms	T	190	1ms
((((()))))	10	T	7666	9ms	T	1872	11ms	T	362	1ms
((...))	20	T	51.863.993	1.049ms	T	15.732	23ms	T	2.772	3ms
((...))	30	T	348.838.486.712	3.699.571ms	T	102.660	32ms	T	9.232	5ms
((...))	40				T	127.452	50ms	T	21.743	4ms

The input word with a length of 40 did not terminate over night for the naive approach. The corresponding graphs to the data in the table are shown below. In the experiments with input words of a length of 8 or smaller, the graphs show that the naive parser seems to perform the best. Considering longer input words, it can be observed, that the naive parser clearly

has the most growth. This observations of the performance with short input words can be disregarded, because the amounts of calls for input words with a length smaller than 8 do not have a big difference to each other. For the longer words, the results are showing more clearly.

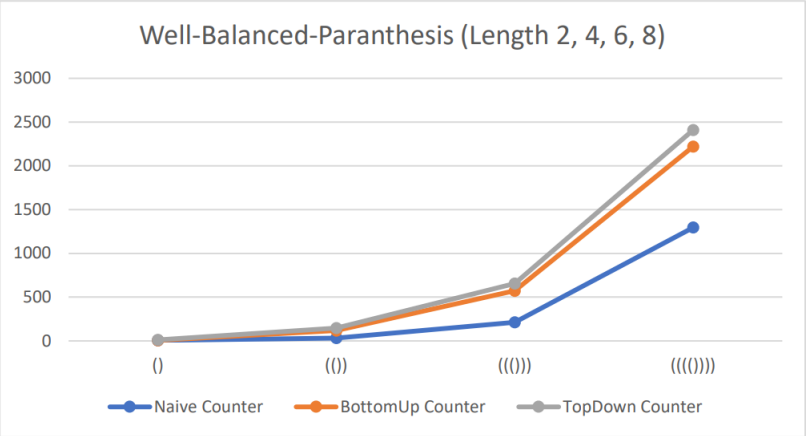


Figure 2: Well-Balanced-Parantheses with input `()`, `(())`, `((()))` and `(((())))`

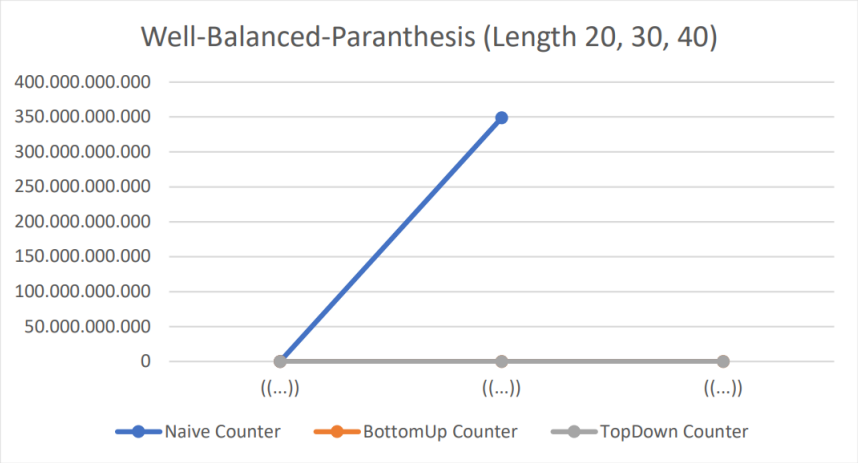


Figure 3: Well-Balanced-Parantheses with input of length 20, 30 and 40

In the following table and graphs, variations of the input word $(())^*$ were evaluated.

Word	Length	N	NC	NT	B	BC	BT	T	TC	TT
(())	4	T	33	0ms	T	168	7ms	T	28	5ms
(())(())	8	T	89	1ms	T	1680	7ms	T	60	2ms
(())(())(())	12	T	145	1ms	T	6072	14ms	T	92	3ms
(())(())(())(())	16	T	201	15ms	T	14.880	13ms	T	124	3ms
(())(())(())(())(())	20	T	257	15ms	T	29.640	21ms	T	156	4ms
(())(())(())(())(())(())	24	T	313	16ms	T	51.888	22ms	T	188	5ms

In the graphs below it can be seen, that the naive and topdown parser perform quicker than the bottomup parser for the tested input words.

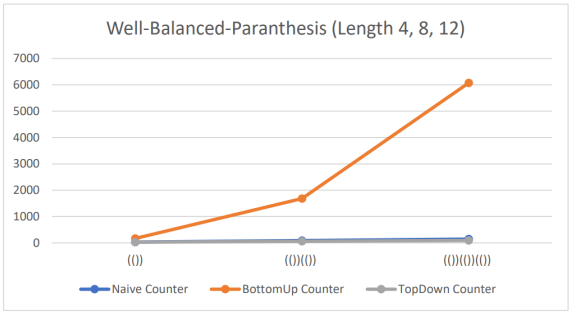


Figure 4: Well-Balanced-Parantheses with input `(())`, `(())(())` and `(())(())(())`

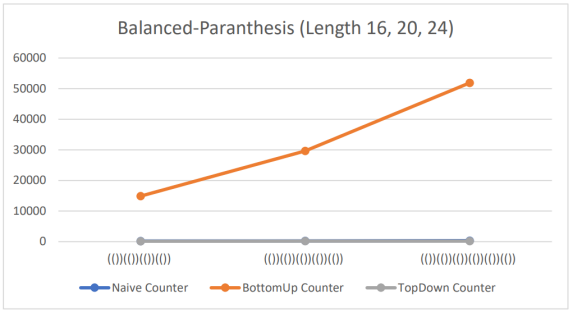


Figure 5: Well-Balanced-Parantheses with input words of length 16, 20 and 24

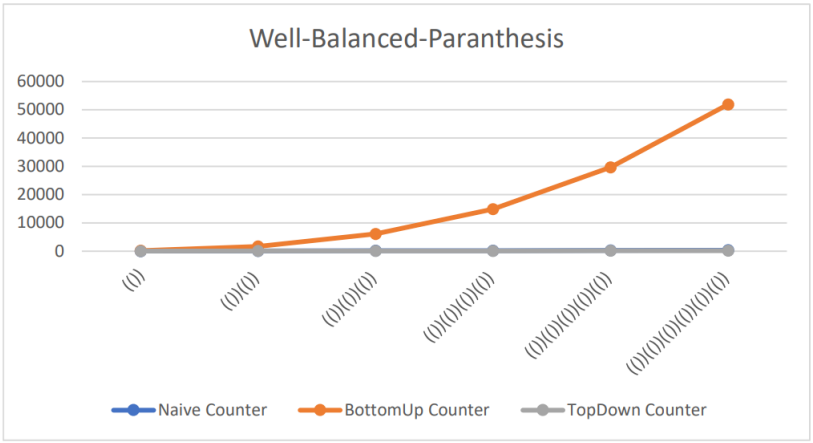


Figure 6: Well-Balanced-Parantheses with input of length 4, 8, 12, 16, 20 and 24

Stupid grammar:
`"SST" "STS" "Ta"`

The stupid grammar accepts no input word, because the nonterminal symbol `S` cannot be resolved into a terminal symbol. The only terminal symbol of this grammar is the symbol `a`.

The input word with a length of 32 did not terminate for the naive approach. The corresponding graphs to the data in the table are shown below.

Word	Length	N	NC	NT	B	BC	BT	T	TC	TT
a	1	F	1	1ms	F	0	6ms	F	1	5ms
aa	2	F	4	1ms	F	4	8ms	F	4	2ms
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
aaaaaaa...aaaaaaaaa	32				F	41.664	25ms	F	11.409	4ms

The corresponding graphs to the data in the table are shown below. In the experiments with input words of a length of 4 or smaller, the graphs show that the naive parser seems to perform the best. Considering longer input words, it can be observed, that the naive parser clearly has the most growth. This first observation of the naive parser performing the best can be disregarded, because the amounts of calls for input words with a length smaller than 4 do not have a big difference to each other. For the longer words, the results are showing more clearly.

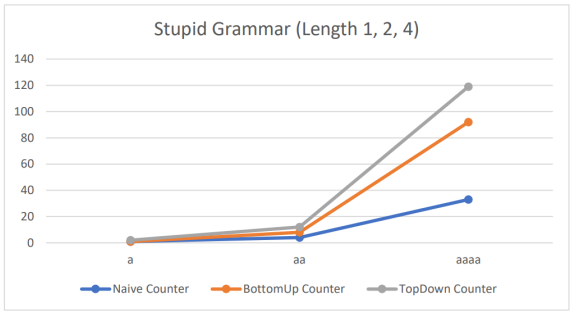


Figure 7: Stupid grammar with input `a`, `aa` and `aaaa`

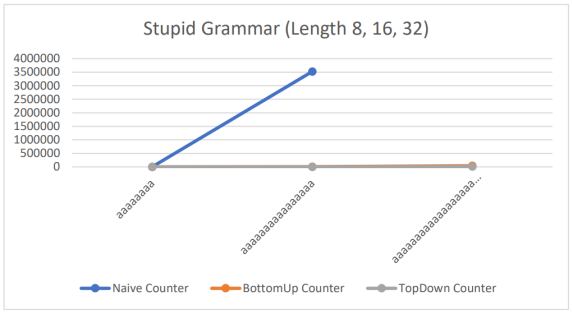


Figure 8: Stupid grammar with input of length 8, 16 and 32

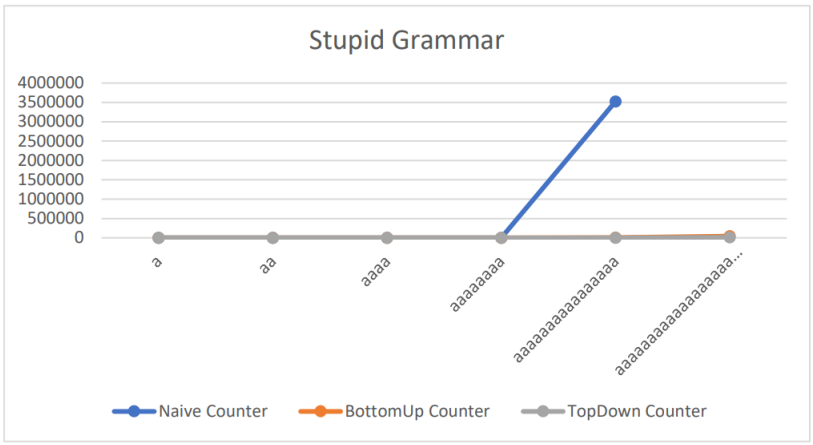


Figure 9: Stupid grammar with input of length 1, 2, 4, 8, 16, and 32

Linear grammar example (aaa-grammar): `"SAa" "ABa" "Ba"`

This grammar accepts only the input word `aaa`. Every other input gets rejected.

In the graphs below it can be seen, that the linear topdown parser uses less recursive calls than the topdown parser. For the (not linear) topdown approach, the grammar has been translated into CNF before the function is applied.

The abbreviations starting with N stand for the naive parser, the B represents the bottomup parser, the T for topdown and L for the linear topdown approach. The second letter of the

abbreviations represents the counter of calls/iterations with the letter C and the measured runtime is represented with the letter T.

Word	Length	NC	NT	BC	BT	TC	TT	LC	LT
a	1	1	1ms	0	5ms	1	2ms	1	1ms
aa	2	2	0ms	4	4ms	2	2ms	2	1ms
aaa	3	6	2ms	20	5ms	6	3ms	4	1ms
aaaa	4	10	0ms	56	5ms	10	2ms	7	1ms
aaaaaaaa	8	36	0ms	560	6ms	36	4ms	29	2ms
aaaaaaaaaaaa	12	78	1ms	2.024	15ms	78	3ms	67	1ms
aaaaa...aaaaa	16	136	1ms	4.960	13ms	136	3ms	121	2ms
aaaaa...aaaaa	20	210	2ms	9.880	13ms	210	3ms	191	2ms

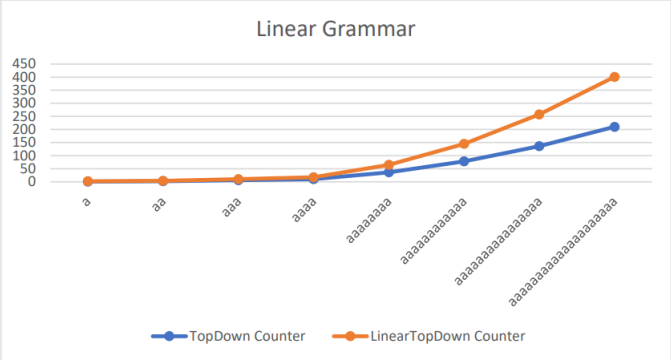


Figure 10: Linear grammar example (aaa-grammar) on topdown and linear topdown parser

5.2 *O*-notation and runtime comparison

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

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

Seeing those runtimes one could think that the bottomup and the topdown approach are equally efficient. Considering the differences in the algorithms and the measurements of the experiments in section 5.1 it gets shown that this is not the case. The *O*-notation runtimes are the upper bounds for the runtime.

Regarding the experiments that were run on the code, different approaches perform better for different grammars and input words. 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, which has an lower bound of $\Omega(2^n)$.

The linear version of the topdown performs more efficient than the original parsing method, because the amount of recursive calls was reduced from two to one.

To confirm the calculated runtimes from Section 3, the results of the experiments are calculated, considering the O-notation runtime. For the input length (n) the actual results and O-notation results as upper bound are compared.

Naive approach ($\Omega(2^n)$):

Ω-notation is defined as the lower bound of an algorithm – that is the least amount of time required [6]. The experiments show, that the calls of the function have a lower bound of 2^n.

Well-Balanced-Parantheses:

2^n	counter
4	6
16	33
64	212
256	1295
1.024	7666
1.048.576	51.863.993
1.073.741.824	348.838.486.712

The results of the experiments confirm the runtime calculation of $\Omega(2^n)$ in Section 3.3.2.

Bottomup approach ($O(n^3)$):

Well-Balanced-Parantheses:

n	n^3	counter
2	8	6
4	64	28
6	216	84
8	512	190
10	1.000	362
20	8.000	2.772
30	27.000	21.743

The results of the experiments confirm the runtime calculation of $O(n^3)$ in Section 3.3.7.

Topdown approach ($O(n^3)$):
Well-Balanced-Parantheses:

In the following table, variations of the input word `(*)*` were evaluated and compared with the runtime in O-notation.

n	n^3	counter
2	8	6
4	64	28
6	216	84
8	512	190
10	1.000	362
20	8.000	2.772
30	27.000	9.232

The results of the experiments confirm the runtime calculation of $O(n^3)$ in Section 3.3.4.

Linear topdown approach ($O(n^2)$):

For the linearapproach, a linear grammar (aaa-grammar) is evaluated with the input word $\mathtt{a}^*$.

n	n^2	counter
1	1	1
2	4	2
3	9	4
4	16	7
8	64	29
12	144	67
16	256	121
20	400	191

The results of the linear parser confirm the runtime of $O(n^2)$. The difference between the performance of the topdown approach and the linear topdown approach is shown in the table below for the aaa-grammar.

Word	Length	Topdown counter	Linear topdown counter
a	1	1	1
aa	2	2	2
aaa	3	6	4
aaaa	4	10	7
aaaaaaaa	8	36	29
aaaaaaaaaaaa	12	78	67
aaaaa...aaaaa	16	136	121
aaaaa...aaaaa	20	210	191

The linear topdown parser is restricted to linear grammars and for the comparison of the linear parsing method to the other parsing methods the linear grammar get translates into *CNF*. With the restriction to linear grammars, the recursive calls in the linear parser can be reduced from two to one in each call.

It can be seen, that the linear parser performs better than the not modified version of the topdown parser.

5.3 Results

The O-notation results from Section 3 show, that the naive approach has a runtime of $\Omega(2^n)$ and the topdown and bottomup approach have the same upper bound runtime of $O(n^3)$. This shows, that the lower bound runtime of the naive approach is still higher than the worst-case runtime for the other approaches. The experiments on the other hand show, that the topdown approach seems to perform better than the bottom up parser.

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

Considering the experiments, the topdown parser seems to be the most efficient, followed by the bottomup approach and the naive is the slowest. Regarding the experiments from section 4.2, different approaches perform better for different grammars and input words. The parser for linear grammars has different restrictions for the input and its results are discussed in Section 5.4.

The reason for the faster results of the topdown parser is the global boolean table, which saves results from previous function calls and can cause less recursive calls of the function to compare the values while the naive approach calls the recursion for every single possibility. The actual runtime of the naive approach is often a lot closer to the upper bound than the actual runtime of the topdown approach, which seems to perform more efficient in the experiments.

In the experiments, the topdown performed better than the bottomup approach, even though the bottomup and the topdown parser are in $O(n^3)$. Because the bottomup fills in the CYK-table iteratively, the upper bound runtime is similar to the actual runtime.

For these results it needs to be considered, that the O-notation states the bound from above. An algorithm can run better than the according O-notation runtime. It usually grows at most as much as the O-notation function [2].

5.4 Results for linear grammars

The upper bound runtime of the modified topdown parser for linear grammars is in $O(n^2)$ while the topdown parser for grammars in *CNF* is in $O(n^3)$. In the linear approach are one instead of two recursive call per execution made, because the linear grammars have at most one nonterminal symbol in the body of each rule. The recursive call needs only to be made for the nonterminal symbols.

The experiments in section 5.1 show, that the linear topdown approach performs faster than the not modified approach. This was also shown and compared in section 5.2.

For this comparison, both parsers took a linear grammar as input. For the topdown parser, the grammar was translated into CNF while the linear topdown parser is restricted to linear grammars as input. With the restriction to linear grammars, the recursive calls in the linear parser can be reduced from two to one in each call. This is why the performance difference can be seen in the upper bound as well as in the experiments.

6 Conclusion and Future Work

6.1 Conclusion

Regarding the experiments from section 5, 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. The O-notation showed different results considering the upper bounds of the methods. This shows, that the O-notation bound is an upper bound and an algorithm can perform faster in experiments, depending on the input stream.

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 Section 5.1.

The linear parsing method on the other hand worked with a different restriction on the input stream, which leads into even higher performances.

6.2 Future work

Formal languages and grammars can be used in different use cases, for example for AI learning methods [3, 11]. Real languages like English can not entirely be defined as a formal languages, because they evolve naturally but some rules and order can be represented and designed as formal language. It is important to consider that formal languages only define the syntax and not the semantics [8].

The future work for this code is to optimize the error correction. The combination of deletion and exchange is not implemented. It would be interesting to find a more efficient way to calculate the exchange of symbols, which would enable the exchange of more than one symbol. Then tests can be run with the error correction and the results can be compared and analyzed.

Another extension to the code would be a method, that translated every kind of grammar into *CNF*, if possible. Then input words for every grammar could be parsed with the different parsing approaches and possibilities to modify the parsing methods for different input grammars appear, similar to the modifications that were done for the parsing of linear grammars.

References

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

[2] P Danziger. *Big o notation*. cs.ryerson.ca/~mth607/Handouts/big0.pdf, opened on 17.01.2023. 2010.

[3] Jean-Paul Delahaye. *Formal methods in artificial intelligence*. Halsted Press, 1987.

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

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

[6] GeeksForGeeks. *Difference between Big Oh, Big Omega and Big Theta*. https://www.geeksforgeeks.org/difference-between-big-oh-big-omega-and-big-theta/, opened on 21.03.2023.

[7] Dietmar Herrmann. "Rekursion". In: *Effektiv Programmieren in C*. Springer, 1991, pp. 114–125.

[8] 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.

[9] 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.

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

[11] Stanley J Rosenschein et al. "Formal theories of knowledge in AI and robotics". In: (1985).

[12] Jessica Su. *Solving recurrences*. https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture3.pdf, opened on 20.03.2023.

[13] University Ulm. *Lecture slides in Formale Methoden der Informatik*. https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.040/Formale_Methoden_der_Informatik/Vorlesungsskripte/FMdI-06--2010-01-10--FormaleSprachen_Vorlesung.pdf, opened on 24.10.2022. 2011.

[14] Gate Vidyalay. *Image Source: Recursion tree*. https://www.gatevidyalay.com/recursion-tree-solving-recurrence-relations/, opened on 17.01.2023.

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

A How to use the code?

The code can be run in the terminal and input is expected as Strings in quotation marks. 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 `(())`.
Other example:
`java Main "SAa" "ABa" "Ba" "aaa"`

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 symbols are represented by which integers. Later the 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.

Then the input word is shown with the symbols replaced by their integers.

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

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

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


Symbols as Integers:
0 1 2 3 4 5
S A L R ( )

Integer-Matrix all rules:
[0 0, 2 1, 2 3]
[0 3, , ]
[4, , ]
[5, , ]

Inputword as Integers:
4 4 5 5
--------------------------------
```

```
[2, , , 0]
[, 2, 0, 1]
[, , 3, ]
[, , , 3]
BottomUp: true   Amount of calls: 168    Amount of errors: 0
BottomUp runtime: 7ms

TopDown: true   Amount of calls: 28
TopDown runtime: 3ms

Naive: true   Amount of calls: 33
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. The error counter is printed, too.

In this case the error correction shows that no errors where found and this leads to no symbols being exchanged or deleted.

```
Error correction

Error correction with exchange:
No exchange option for 1 symbol found/necessary


Error counter for deletion: 0
Error correction with deletion
amount of errors: 0
Accepted word:
( ( ) )
```

The following example shows the error correction output for the input `java Main "SSS" "SLA" "SLR" "ASR" "L(" "R)" "(((()"`.

In this case the error correction shows the solution with exchange (one symbol is exchanged) and for deletion (two symbols are deleted).

```
Error correction with exchange:
1 symbol was exchanged.
Accepted word:
( ) ( )

Error counter for deletion: 2
Error correction with deletion
amount of errors: 2
Accepted word:
( )
```

```
BottomUp: true   Amount of calls: 440    Amount of errors: 0
BottomUp runtime: 6ms

TopDown: true   Amount of calls: 21
TopDown runtime: 3ms

Naive: true   Amount of calls: 21
Naive runtime: 1ms

LinearTopDown: true   Amount of calls: 30
LinearTopDown runtime: 1ms
```

If the input grammar is a linear grammar, the program checks if it is in *CNF*, build *CNF* out of the linear grammar and runs the optimized version of the topdown parser for linear grammars.

B CNF Algorithm on an example

In this part, the following ruleset of a grammar is 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:

   $$\texttt{S} \rightarrow \texttt{SS} \mid \texttt{LSR} \mid \texttt{LR}$$
   $$\texttt{L} \rightarrow \texttt{a}$$
   $$\texttt{R} \rightarrow \texttt{b}$$

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

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

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:

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