Umeå University

Efficient Algorithms

Assignment Step 2

Runtime analysis of the Java implementation of the CYK-algorithm

Pina Kolling

October 25, 2022

Contents

1 Introduction

Parsing in Computer Science is the process of reading and processing an input. 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*), which is explained in Section 2.3. [6, 9]

The task for this assignment is to code three different parsing methods 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 grammer 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.

In Section 4 the runtimes and experiments of the different algorithms are compared and differences in efficiency are shown.

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) and recursion (Section 2.6) 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, 7]

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 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, 7]

[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`. [8]

A formal grammar defines which words over the alphabet T/Sigma 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:

$$S \rightarrow SS \mid aSb \mid ab$$

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 formated 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 C.*

$$S \rightarrow SS \mid LA \mid LR$$

$$A \rightarrow SR$$

$$L \rightarrow a$$

$$R \rightarrow b$$

2.4 Dynamic programming

The technique of dynamic programming can be used to solve problems, which can be 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

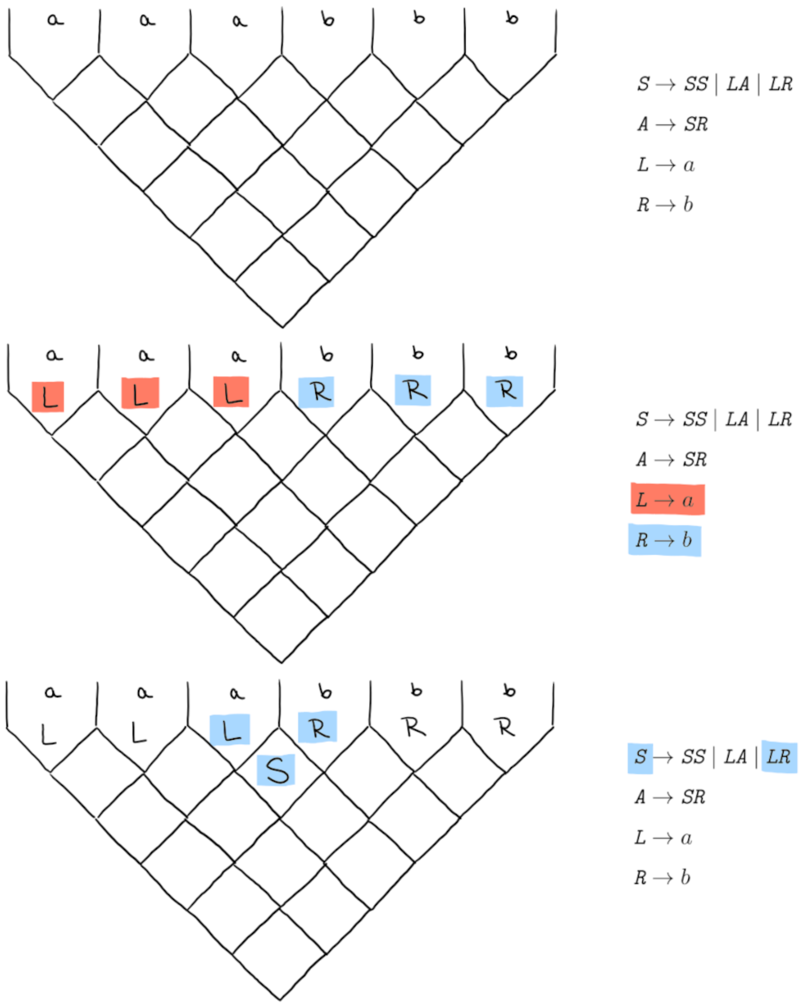
5	0	1	3	4	6	7	9	10
4	0	1	3	4	6	7	9	10
3	0	1	3	4	5	7	8	8
2	0	1	1	4	5	5	5	5
1	0	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7

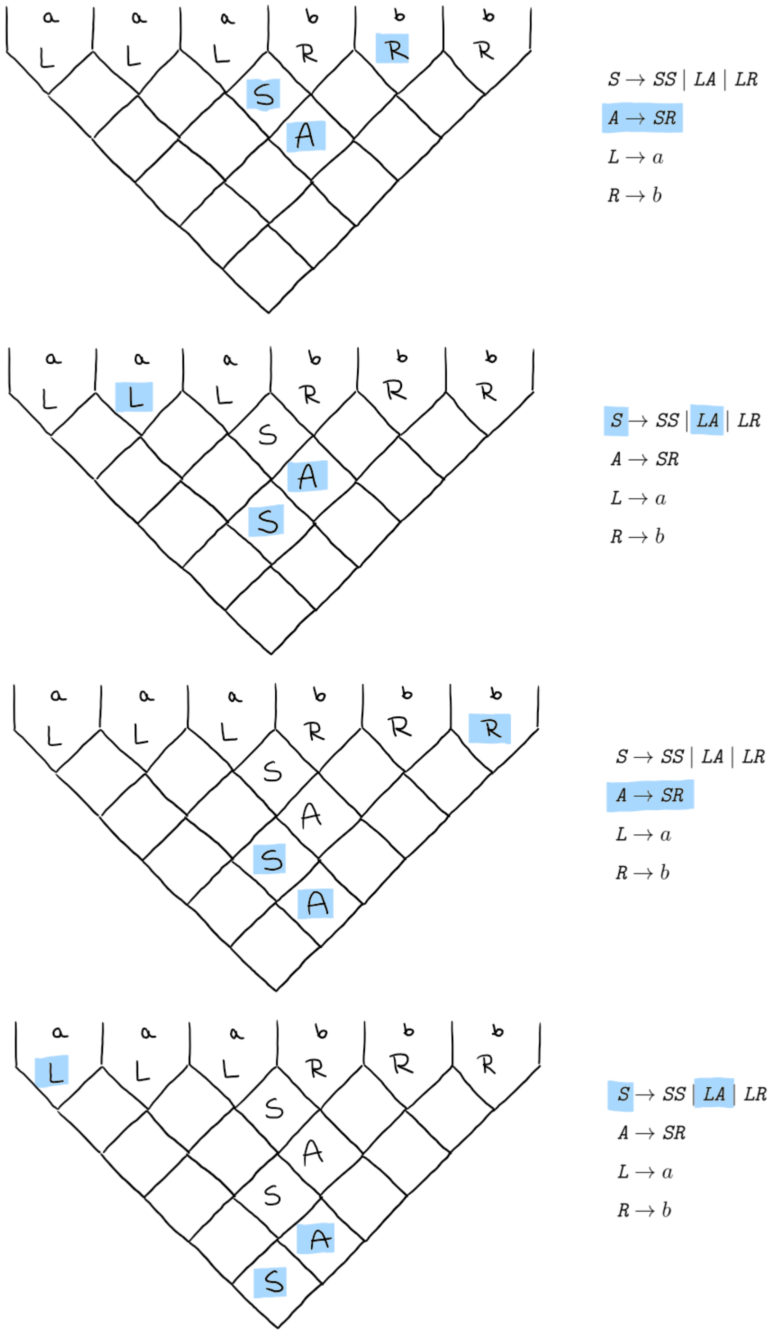
The first column in the table represent the objects and the last row the weight. The entries in the table show the maximum value. In this example the maximum value for the size 7 is 10.

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. [9]

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 intiating 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. [3]*





2.6 Recursion

A function that calls itself is called recursive function. In computer science recursive function use base cases to terminate the programm and stopping it from going on forever. Base cases are problems that can be solved without any more recursive calls. [4]

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

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

Nonrecursive approach:

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

Recursive approach:

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

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 to array – for each symbol of the body one array.

For faster and easier access, three arrays are built: One array that only contains only non-terminal 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 3.3.7). 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 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. 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 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` $\rightarrow$ `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 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][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
18:         end if
19:     end for
20: end if
21: 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.

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	$n!$	Recusrsive call
15	$1 \cdot n \cdot k$	Return statement
21	1	Return statement

Calculating the runtime:

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

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

3.3.3 Top-Down description

The top-down method is an improved 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()

```
counter ← 0
for i ← 0 to table.length do
    for j ← 0 to table[i].length do
        for k ← 0 to table[i][j].length do
            table[i][j][k] ← null
        end for
    end for
end for
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 Boolean parseTD(int indexNT, int i, int j)

```
if table[indexNT][i][j] != null then
    return table[indexNT][i][j]
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 returnes (lime 20). The general topdown approach is the same approach as the naive method that is described in Section 3.3.1.

Algorithm 5 Boolean 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] == true 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

In this part the runtime of the topdown method is calculated. The variable n represents the length of the input word and k the dimension of the rule array. The first part of the calculation shows the recursion call method.

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:

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

The runtime analysis for the upper bound of the `parseTD` method is analog to the `parseNaive` runtime analysis. According to this the topdown algorithm has an upper bound runtime in $O(2^n)$, too.

3.3.5 Top-Down modified for linear grammars

In the modified version for linear grammars, the grammar gets changed into CNF first. The modified algorithm can now work more efficient, because for each rule is only one recursive call of the method instead of two neccessary.

For each rule it is checked if the first or the second element of the body is the head of a terminal rule. Then it is checked if the head of the terminal rule is the head of the current position on the input word. For the other symbol the methos gets called recursively. This part is shown in the pseudo code below:

$first \leftarrow ruleset[indexNT][bodyIndex][0]$
$second \leftarrow ruleset[indexNT][bodyIndex][1]$
for $k \leftarrow i + 1$ **to** j **do**
  if *second* is head of a terminal rule **then**
    $table[indexNT][i][j] = parseLinearTD(first, i, k)$ **and** *second* is head of terminal rule with body $inputAsInt[k]$
  end if
  if *first* is head of a terminal rule **then**
    $table[indexNT][i][j] = parseLinearTD(second, k, j)$ **and** *first* is head of terminal rule with body $inputAsInt[i]$
  end if
end for

3.3.6 Top-Down modified for linear grammars runtime

The runtime of the modified version is $O(n^3)$, because the recursion is not in $O(2^n)$ anymore like in Section 3.3.2. Instead the method is in $O(n^3)$.

3.3.7 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 6 Boolean parseBU()

```
1:  wordlength ← word.length
2:  Integer[][][]DP ← newInteger[wordLength][wordLength][wordlength]
3:  for i ← 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 excecuted. 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.

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

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.

3.3.8 Bottom-Up runtime

In the following part the upper bound runtime of the bottom up algorithm gets analysed. 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 are compared.

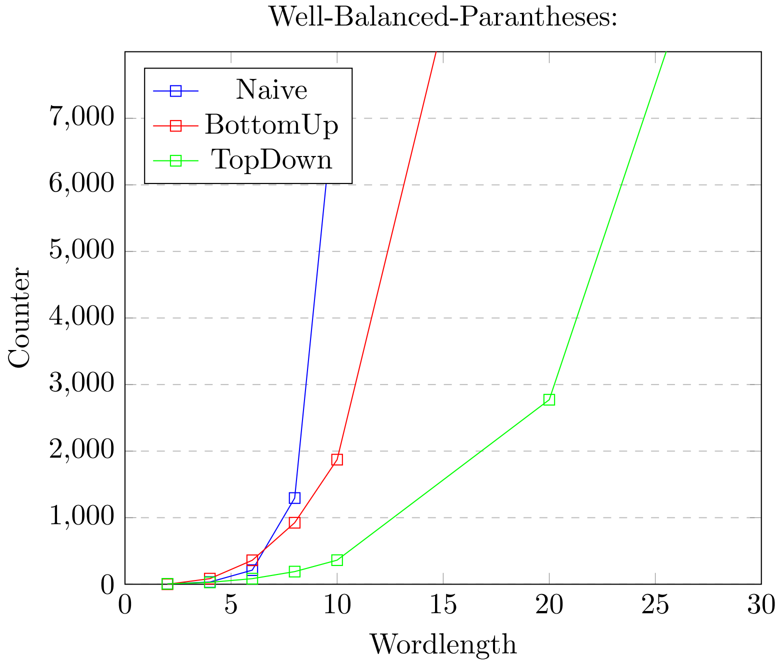
4.1 Experiments

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

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 graph to the data in the table is shown here:



Linear grammar example (abc-grammar):
`"SAc" "Sb" "AaS" "AaB" "BbS" "a*b*c*"`

Word	Length	NC	NT	BC	BT	TC	TT	LC	LT
abc	3	6	1ms	60	5ms	6	3ms	5	0ms
aabbcc	6	51	1ms	660	5ms	49	3ms	51	1ms
aaaabbbbcccc	12	1.457	1ms	6.072	9ms	518	3ms	650	1ms
a...b...c...	24	481.557	9ms	51.888	20ms	4.312	4ms	6.732	3ms

4.2 Runtimes

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

Method	Runtime
`parseNaive`	$O(n!)$
`ParseTD`	$O(2^n)$
`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 bounds 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.

To confirm the calculated runtimes from Section 3, the results of the experiments will be calculated considering the O-notation runtime.

Naive approach ($O(n!)$):
Well-Balanced-Parantheses:

n	$n!$	counter
2	2	6
4	24	33
6	720	212
8	40.320	1295
10	3.628.800	7666
20	2.432.902.008.176.640.000	51.863.993
30	265.252.859.812.191.058.636.308.480.000.000	348.838.486.712

Abc-grammar:

n	$n!$	counter
3	6	6
4	24	51
12	479.001.600	1.457
24	620.448.401.733.239.439.360.000	481.557

The results of the experiments confirm the runtime calculation of $O(n!)$ in Section 3.3.2. Small deviation for smaller input numbers can happen, because the constants are not considered.

5 Errorcorrection

6 Conclusion and Future Work

6.1 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.

NOOO !

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.

6.2 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. [5]

The future work for this assignment is to change the types of the arrays from `String[][]` to `int[][][]`. In addition to this there still exists a problem in the bottomup method, which needs to be fixed. Some specific grammars and words do not return the right value. After these things work correctly, more experiments can be run and the pseudo code will be edited. Then the upper bound runtime can be calculated again and the plotted graphs can be compared to the according O-notation runtime.

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 "SAc" "Sb" "AaS" "AaB" "BbS" "abc"`

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 will show 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 programm 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 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:

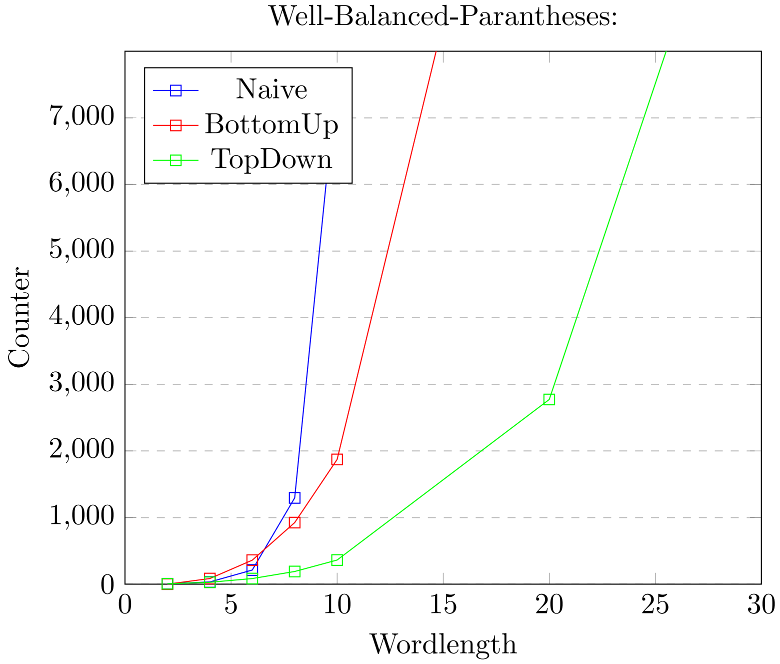
Word	Length	N	NC	NT	B	BC	BT	T	TC	TT
()	2	T	6	5ms	T	0	2ms	T	6	1ms
(())	4	T	33	4ms	T	84	3ms	T	28	1ms
((()))	6	T	212	6ms	T	360	6ms	T	84	1ms
(((())))	8	T	1295	6ms	T	924	9ms	T	190	1ms
((((()))))	10	T	7666	9ms	T	1872	11ms	T	362	1ms
((...))	20	T	51.863.993	1.049ms	T	15.732	23ms	T	2.772	3ms
((...))	30	T	348.838.486.712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

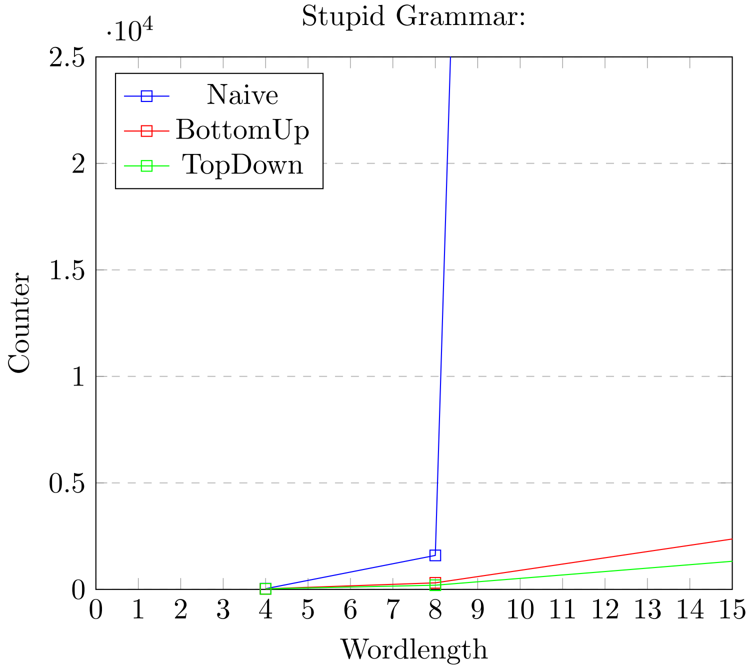
The input word with a length of 40 did not terminate over night for the naive approach.

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
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
Counter
0
5
10
15
20
25
30
Wordlength


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

C CNF Algorithm on an example

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

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

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:

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

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

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

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:

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

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

[5] 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.

[6] 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.

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

[8] 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.

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