Umeå University
Department of Computing Science

Efficient algorithms
5DV182

Cocke–Younger–Kasami algorithm analysis

Thomas Ranvier

supervised by
Franck Drewes

November 04, 2018

Abstract

This paper presents six different implementations of the Cocke-Younger-Kasami algorithm (CYK algorithm for short). The first one is a naive version, it uses the Divide and conquer approach and has a complexity of $O(3^n)$. The other implementations make use of Dynamic programming, there are two versions using bottom-up approach and the others use the top-down method, they all have a complexity of $O(n^3)$.

The four first parsers are classic implementations of the CYK algorithm using different programming methods, the last two have some specialities. After the presentation of the implementations and the analysis of the different complexities a chapter is focused on experimentations with the four first parsers using diverse context-free grammars in the Chomsky normal-form.

The implementation of a parser able to make use of linear grammars with a running time of $O(n^3)$ and based on the top-down implementation is presented. Experimentations and comparisons with the results of the four previous parsers are made.

A parser that automatically corrects the input string in order to make it match the given grammar (when possible) in $O(n^3)$ is presented. This parser is based on the top-down initial implementation, it suggests to the user a correction of the initial string and indicates the number of character modifications and character deletions that were needed to reach total correctness.

Contents

Introduction

This is the report of the efficient algorithms assignment on the CYK parser.

The point of the assignment was firstly to discover and implement the CYK algorithm using several development approaches. Then the goal was to experiment with the implementations and compare their efficiencies for diverse context-free grammars in the Chomsky normal-form.

After that the task was to design a parser that can work with linear grammars, to implement it and then to experiment with it in order to compare its efficency to the other parsers.

Then the final goal was to design and implement a parser that does not simply return true if the string is part of the language and false otherwise, but corrects the input string and returns a proposed correction with the needed number of character modifications and deletions.

Chapter 1

Context

1.1 Context-free grammar

Context-free grammars are part of the Chomsky hierarchy [1, 2] and can be defined as a quadruple $G = (N, \Sigma, P, S)$ that consists of the following components:

- N is a finite set of *non-terminal* variables.
- Σ is a finite set of *terminal* variables.
- P is a finite set of *production rules*, also called *rewrite rules*, it specifies a symbol substitution that can be recursively performed to generate new symbol sequences. In an unrestricted grammar, which is the most general form of the Chomsky hierarchy, a production is of the form $u \to v$ where u and v are arbitrary strings of *terminals* and *non-terminals*.
- $S \in N$ is a distinguished symbol that is the *start symbol*, also called the *sentence symbol*.

The language $L(G)$ is the language generated by the grammar G [3]. To generate a string of the language, one begins with a string consisting of only a single start symbol, and then successively applies the rules (any number of times, in any order) to rewrite this string. This stops when we obtain a string containing only terminals. The language consists of all the strings that can be generated in this manner.

The same process can be applied backward to check weither a given string can be generated by the grammar or not.

1.1.1 Description of the Chomsky normal-form

Here is an example of a context-free grammar in the Chomsky normal-form:

$$S \rightarrow AB|BC$$
$$A \rightarrow BA|a$$
$$B \rightarrow CC|b$$
$$C \rightarrow AB|a$$

The grammar can be described as a finite set of production rules, in those productions we can find four types of symbols:

- The '$\rightarrow$' symbol is what seperates the left-hand from the right-hand(s) of the rules.
- In this grammar every single uppercase letter is a *non-terminal* variable.
- The *terminals* are primitive symbols, they can only be found on the right-hand of a production.
- A left-hand is constitued of one *non-terminal*, it can relate to one or several right-hands seperated by the '|' symbol. A right-hand is constitued either by one *terminal* or a pair of *non-terminals*.

Here S, A, B, and C are *non-terminals*, a and b are *terminals* and S is the *start symbol*.

1.1.2 Convert grammar to Chomsky normal-form

Sometimes it is needed to convert a context-free grammar to the Chomsky normal-form. Later in this report is presented an algorithm that follows the next pseudo-code in order to automaticly convert a given context-free grammar into the Chomsky normal-form.

For example this is how to convert the following grammar:

$$S \rightarrow S + P|P$$
$$P \rightarrow P * C|C$$
$$C \rightarrow (S)|0|1$$

1. The first thing to do is to eliminate the *start symbol* from the right-hands of the grammar by replacing the *start symbol* by a new one (some labels

have been modified):

$$
\begin{aligned}
&S \to A \\
&A \to A + B | B \\
&B \to B * C | C \\
&C \to (A) | 0 | 1
\end{aligned}
$$

2. Then the goal is to replace the *terminals* that are not the only symbols on the right-hand by a new *non-terminal*:

$$
\begin{aligned}
&S \to A && P \to + \\
&A \to APB | B && M \to * \\
&B \to BMC | C && L \to (\\
&C \to LAR | 0 | 1 && R \to)
\end{aligned}
$$

3. The next step is to delete the right-hands with more than 2 *non-terminals* by adding *non-terminals* that will relate to a pair:

$$
\begin{aligned}
&S \to A && F \to AR \\
&A \to AD | B && P \to + \\
&B \to BE | C && M \to * \\
&C \to LF | 0 | 1 && L \to (\\
&D \to PB && R \to) \\
&E \to MC
\end{aligned}
$$

4. At last it is needed to eliminate the unit rules, those are the right-hands in which there is one *non-terminal* alone:

$$
\begin{aligned}
&S \to AD | BE | LG | 0 | 1 && F \to AR \\
&A \to AD | BE | LG | 0 | 1 && P \to + \\
&B \to BE | LG | 0 | 1 && M \to * \\
&C \to LF | 0 | 1 && L \to (\\
&D \to PB && R \to) \\
&E \to MC
\end{aligned}
$$

1.2 CYK algorithm

The CYK algorithm is a parser for context-free grammar, it is named after its inventors, John Cocke, Daniel Younger and Tadao Kasami. The algorithm takes

a string and a context-free grammar as input and determines if the string is part of the language of the grammar or not. It uses the backward process explained in the presentation of section 1.1.

Context-free grammar parsers can be used for example in computer sciences to check code structure in compilers or in biology for DNA and RNA strings analysis.

1.3 Diverse development approaches

The goal of the assignment is to implement CYK parsers using different development approaches in order to compare the efficiency of those. In this section the three used development methods are briefly presented.

1.3.1 Divide and conquer

Divide and conquer is an algorithmic strategy that consists in recursively breaking down a problem into several others, and do so until the problems become easy enough to solve. The obtained results are then combined in order to solve the initial problem.

Those are the steps of the general structure of that strategy:

1. Divide the problem instance given as input into smaller instances.
2. Recursively compute the results for those smaller instances.
3. Assemble the recursively obtained results into a result for the entire input and return it.

1.3.1.1 Recurrence relation

It is possible to find a function that satisfies the recurrence relation of the divide and conquer strategy. Let's suppose that a recursive algorithm divides a problem of size n into a sub-problems, where each sub-problem is of size $\frac{n}{b}$ Suppose also that $g(n)$ is the total time needed to create the sub-problems and combine their results.

Then we can define $f(n)$, being the number of operations needed to solve the problem of size n, as:

$$f(n) = a * f(\frac{n}{b}) + g(n)$$

1.3.1.2 Master theorem

The Master theorem for the divide and conquer recurrences provides an asymptotic analysis of the complexity, using the Big O notation. It allows one to solve $f(n)$ depending on three different cases.

Now that we have the function $f(n)$ we suppose that $g(n) = \Theta(n^d)$.

Then the master theorem is:

$$f(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d log(n)) & \text{if } a = b^d \\ \Theta(n^{log_b(a)}) & \text{if } a > b^d \end{cases}$$

Here the first case is when it is possible to ignore the solving of the subproblems before the creation and combination of the sub-problems. The running time is then the time needed to create and combine the sub-problems, $g(n) = \Theta(n^d)$.

The second case is when it is not possible to ignore anything.

The final case is when it is possible to ignore the creation and combination of the sub-problems. The running time is then the time needed to solve $a^{log_b(n)}$ subproblems of size 1.

1.3.2 Dynamic programing

Dynamic programing is an algorithmic paradigm that solves a given complex problem by breaking it down into subproblems and storing the results of those subproblems to avoid computing the same results again.

1.3.2.1 Top-down approach

The top-down approach is similar to the divide and conquer approach, the initial problem is recursively broken down into several sub-problems. The difference is that in this approach a global variable memorize every computed result. In that way when the program runs into a known problem the memorized result is directly returned. The program generally computes less operations and goes less deep in the recursion than the divide and conquer method.

1.3.2.2 Bottom-up approach

The bottom-up approach consists in solving the simplest sub-problems first, memorizing every computed result, and then going up solving bigger and bigger sub-problems until it can solve the initial problem. This approach is a way to avoid recursion, which saves some memory, but on the other hand it can sometimes solve sub-problems that will never be used to solve the initial problem, which is a waste of time.

A good analogy for the bottom-up design are building blocks, indeed you have to start assembling the most little pieces together in order to construct bigger ones that you will then assemble to get the final result.

Chapter 2

Implementations of the different parsers

2.1 Grammar

This is the description of the Grammar class as it has been used for the general experimentations that can be found in the chapter 3. A more efficient version that can handle linear and Chomsky normal-form grammars has been implemented and is presented later in this report.

The different productions of the context-free grammar are given by the user through a file following the same pattern as the example of section 1.1.2.

The program uses the following conventions:

- Every *non-terminal* is a single uppercase letter.
- Every *terminal* is a single other ASCII character.
- The right-hand is composed of one or several groups, each group is either a pair of *non-terminals* or a single *terminal*.

When the program comes accross an uppercase letter it automaticly transorms it into an index beginning from 0, so 'A' becomes 0, 'B' becomes 1, etc.

My program then fills three variables:

- The variable 'non_terminals' is a string, when a new *non-terminal* is read from the file it is converted as described above and then added as a character to the variable. This variable will then make it easy and efficient to access the rules.
- The variable 'non_terminal_rules' is a three dimensional array of integers.

1. The first dimension has the size of the latin alphabet: 26, each line of this dimension will allow access to the non-terminal rules of the corresponding *non-terminal*, using the converted values stored in 'non_terminals'.
2. The second dimension allows access to the different pairs of *non-terminals*.
3. The third dimension has a size of 2, it represents the pair of *non-terminals*.

– The variable 'terminal_rules' is a two dimensional array of integers.

1. The first dimension has the size of the latin alphabet, each line of this dimension allows access to the terminal rules of the corresponding *non-terminal*.
2. The second dimension allows access to the different *terminals* related to that *non-terminal*.

The point of seperating the *non-terminal* rules from the *terminal* ones in two variables is that they are always accessed on different cases. It would then slow down the processus to always have to differenciate them from one another.

2.2 Naive parser

The naive implementation uses the divide and conquer approach,

Algorithm 1 Naive parser

```
1: s ← input_string
2: procedure PARSE(var, i, j)                    ▷ Recursive parser function
3:     if i == j − 1 then
4:         for all t ∈ terminal_rule[var] do
5:             if t == s[i] then
6:                 return true
7:             end if
8:         end for
9:     else
10:        for all nt ∈ non_termial_rules[var] do
11:            for k ← i + 1; k < j do
12:                if parse(nt[0], i, k) ∧ parse(nt[1], k, j) then
13:                    return true
14:                end if
15:            end for
16:        end for
17:    end if
18:    return false
19: end procedure
```

2.2.1 Complexity of the algorithm

To find the complexity of this algorithm one can draw recursion trees for small values of n and find the following sequence: $0, 2, 8, 26, 80\ldots$ After some research it is possible to find out that this sequence corresponds to the formula $3^{n-1} - 1$ [4].

The resulting complexity of this algorithm is then $O(3^n)$.

2.3 Top-down parser

The top-down implementation is almost the same as the naive one, except that there is here a global variable that memorize every computed results.

Algorithm 2 Top-down parser

```
1: table is a 3d array of 0
2: s ← input_string
3: procedure PARSE(var, i, j)                          ▷ Recursive parser function
4:     if table[var, i, j]! = 0 then
5:         return (table[var, i, j] == 1)
6:     end if
7:     if i == j − 1 then
8:         for all t ∈ terminal_rule[var] do
9:             if t == s[i] then
10:                table[var, i, j] ← 1
11:                return true
12:            end if
13:        end for
14:    else
15:        for all nt ∈ non_terminal_rules[var] do
16:            for k ← i + 1; k < j do
17:                if parse(nt[0], i, k) ∧ parse(nt[1], k, j) then
18:                    table[var, i, j] ← 1
19:                    return true
20:                end if
21:            end for
22:        end for
23:    end if
24:    table[var, i, j] ← 2
25:    return false
26: end procedure
```

2.3.1 Complexity of the algorithm

To compute the running-time of the top-down algorithm we can count the number of cells in the memoization table and multiply it by the complexity of the inner loops of the algorithm. This will give an upper bound of the complexity. The loops that go through the grammar are ignored, the size of the table is n^2, with n the size of the input string, the complexity of the parse function without any recursion is $O(n)$, so the complexity of the top-down parser is $O(n * n^2) = O(n^3)$.

2.4 Bottom-up parser

For this parser there are two possible implementation versions, one uses a boolean table and the other one uses a string table, their process are slightly different and they give different results in term of iterations and running time.

In the first approach the goal is to complete a triangular matrix, when the matrix is fully completed it watches the last cell and if it finds the *start symbol* in it it returns true, otherwise it returns false. That version is called the 'string' bottom-up parser in this report.

Algorithm 3 'String' bottom-up parser

```
1: procedure PARSE(s)                                   ▷ Iterative parser function
2:     matrix is a 2d array of empty strings
3:     step ← 0
4:     for i ← 0; i < s.length() do
5:         for all var ∈ non_terminals do
6:             for all t ∈ terminal_rules[var] do
7:                 if t == s[i] then
8:                     Add var to matrix[step, i] if not already inside
9:                 end if
10:            end for
11:        end for
12:    end for
13:    step ← step + 1
14:    while s.length() − step > 0 do
15:        for i ← 0; i < s.length() − step do
16:            for j ← 0; j < step do
17:                r ← check_combs(matrix[j, i], matrix[step − (j + 1), i + j + 1])
18:                Add r to matrix[step, i] if not already inside
19:            end for
20:        end for
21:        step ← step + 1
22:    end while
23:    return is the start symbol in matrix[s.length() − 1, 0] ?
24: end procedure
25: procedure CHECK_COMBS(cell_1, cell_2)
26:    result is an empty string
27:    for i ← 0; i < cell_1.length() do
28:        for j ← 0; j < cell_2.length() do
29:            production ← cell_1[i].cell_2[j]
30:            if production is in the non − terminal_rules then
31:                Add the left hand of the matching rule to result
32:            end if
33:        end for
34:    end for
35:    return result
36: end procedure
```

In the second approach the goal is to complete a boolean table, then it looks in a specific cell and returns the result. That parser is called the 'boolean' bottom-up parser in this report.

Algorithm 4 'Boolean' bottom-up parser

```
1: procedure PARSE(s)                                   ▷ Iterative parser function
2:     matrix is a 3d array of false booleans
3:     step ← 0
4:     for i ← 0; i < s.length() do
5:         for all var ∈ non_terminals do
6:             for all t ∈ terminal_rules[var] do
7:                 if t == s[i] then
8:                     matrix[0, i, var] ← true
9:                 end if
10:            end for
11:        end for
12:    end for
13:    step ← step + 1
14:    while s.length() − step > 0 do
15:        for i ← 0; i < s.length() − step do
16:            for j ← 0; j < step do
17:                for all var ∈ non_terminals do
18:                    for all t ∈ terminal_rules[var] do
19:                        bool_1 ← matrix[j, i, t[0]]
20:                        bool_2 ← matrix[step − j − 1, i + j + 1, t[1]]
21:                        if bool_1 ∧ bool_2 then
22:                            matrix[step, i, var] ← true
23:                        end if
24:                    end for
25:                end for
26:            end for
27:        end for
28:        step ← step + 1
29:    end while
30:    return matrix[s.length() − 1, 0, start_symbol_index]
31: end procedure
```

The 'boolean' implementation needs a predictable amount of iterations for a given grammar to parse a string, no matter what pattern this string follows. The 'string' one on the other hand needs a various number of iterations depending on the pattern of the input string.

2.4.1 Complexity of the algorithms

The two algorithms above have the same complexity, to determine it it is needed to look at the inner loops of the algorithms. For both of them if the grammar loops are ignored the complexity is $O(n + n^3) = O(n^3)$.

2.4.2 Expression of the number of iterations

For the 'boolean' bottom-up parser it is possible to predict the exact number of iterations that it will compute, indeed unlike the naive and top-down versions there is no recursion that denies us to anticipate it.

It is possible to translate the 'for' and 'while' loops into a mathematical expression:

$$iterations = n \cdot gt + \sum_{k=1}^{n} (n - k) \cdot k \cdot gnt \tag{1}$$

With:

- n being the size of the input string.
- gt being the number of terminal rules: the number of *terminals* on the right-hands of the grammar.
- gnt being the number of non terminal rules: the number of *non-terminals* pairs on the right-hands of the grammar.

With such an expression it will be easier to justify the observed results in the 'boolean' bottom-up parser case.

But the equation 1 has an issue, it is not a polynomial function, which denies one from displaying it in a plot for example. To get a real mathematical expression it is possible to use the Faulhaber's formulas that allows one to translate a sum into a polinomial function:

$$\begin{aligned}
iterations &= n \cdot gt + \sum_{k=1}^{n} (n - k) \cdot k \cdot gnt \\
&\Leftrightarrow n \cdot gt + gnt \cdot \sum_{k=1}^{n} n \cdot k - k^2 \\
&\Leftrightarrow n \cdot gt + gnt \cdot (n \cdot \sum_{k=1}^{n} k - \sum_{k=1}^{n} k^2) \\
&\Leftrightarrow n \cdot gt + gnt \cdot (\frac{n^2 \cdot (n+1)}{2} - \frac{n \cdot (n+1) \cdot (2 \cdot n + 1)}{6})
\end{aligned}$$

The final expression is the following:

$$iterations = n \cdot gt + \frac{gnt}{6} \cdot (3 \cdot n^2 \cdot (n+1) - n \cdot (n+1) \cdot (2 \cdot n + 1)) \tag{2}$$

The function above is a general expression of the number of iterations that the boolean bottom-up parser will have to do in order to parse a string of size n for a grammar possessing gt *terminals* and gnt pairs of *non-terminals* on its right-hands.

With such function one can represent the behaviour of the number of iterations when the number of *non-terminals* gnt increases and the strings sizes n too.

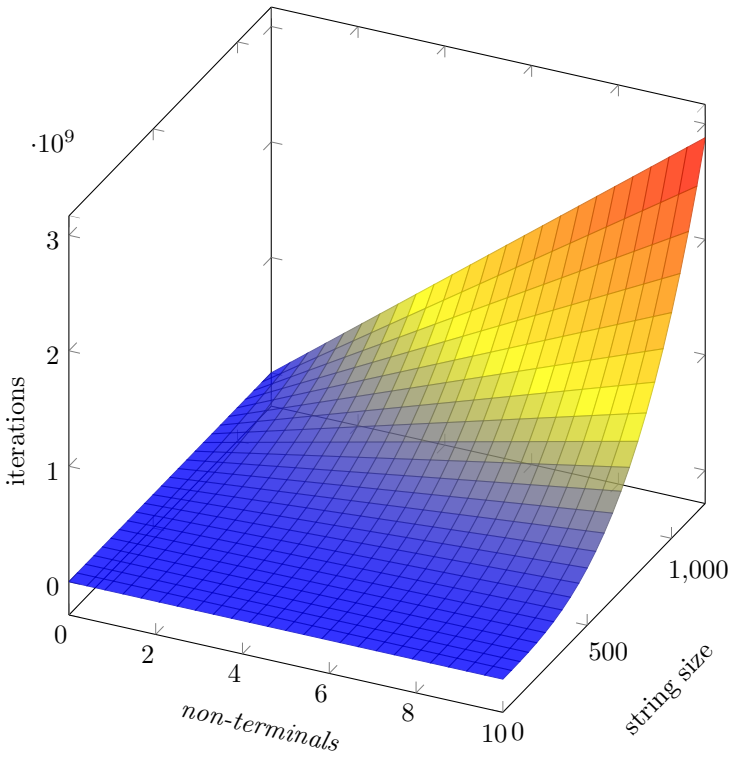


Figure 1: 3D plot of the function 2

It is interesting to see on the 3d plot that for a fixed gnt when n increases the number of iterations increases following a power function which should be of the form $y = a \cdot x^3$, since the complexity of the bottom-up parser is $O(n^3)$.

On the other hand when n is fixed and that gnt increases the number of iterations increases following a linear function. That can be explained by the fact that the *non-terminals* represent only on loop in the parser, so if the amount of *non-terminals* is changed the impact on the number of iterations is linear.

Chapter 3

Experimentations, comparisons and obtained results

3.1 Generation of the strings

In order to make the experimentations easier the input strings are automaticaly generated using patterns. Those patterns are put in a file by the user which allows him to quickly be able to experiment with different cases. Then when the Enumeration class is instanciated it reads the patterns from the file and all the strings are generated and stocked in a vector, then the strings can be accessed easily from anywhere.

The patterns uses the following expression:

$$c\hat{}x\_d\hat{}y$$

Where c is an ASCII character and x is the number of time we add the character, a space separates each characters. It is also possible not to renseignate $\hat{}x$, then the character is added 1 time.

That method allows the user to generate new strings for different grammars very quickly.

3.2 Experimentations

In some of the experimentations the results of the naive parser will not be displayed, which means that it was not efficient enough to be interesting.

The results of the 'boolean' bottom-up parser will be displayed only once per grammar since no matter the pattern of the input string it does not change the behaviour of this parser. However its results might be used several times for a same grammar in order to use them as reference for the other parsers.

3.2.1 Well balanced parentheses

The grammar is the following:

$$
\begin{aligned}
&S \to SS|LA|LR \\
&A \to SR \\
&L \to (\\
&R \to)
\end{aligned}
$$

With this grammar it is possible to demonstrate that the top-down parser needs less recursive calls when it can 'detect' that the string does not match the grammar very quickly.

3.2.1.1 Number of iterations/recursive calls for each parsers

With this grammar two interesting cases will be considered:

1. Any pattern starting with a right parenthese:

$$
patterns = \begin{cases})\hat{}n \\ )\ (\hat{}n-1 \\ )\hat{}n-1\ (\end{cases}
$$

   Many other patterns corresponding to the same case could be found, as long as the first character is a right parenthese. This case is interesting because it is the case in which the top-down parser needs to solve the less sub-problems in order to parse the strings.

   For example if it parses a string of size n based on one of the patterns above it will need to solve $2n-1$ sub-problems to give the result. That information, while interesting, is not relevant to anticipate the running time of the parser since even if we know how many sub-problems the parser will solve it can need a very different amount of recursive calls.

2. The other interesting case is the pattern '$(\hat{}n$', that pattern is the worst that can be generated with this grammar for the top-down parser.

   With that pattern the top-down parser will need to solve $n^2 + \lfloor \frac{n}{2} \rfloor$ sub-problems to parse a string of size n.

Of course as demonstrated in the section 2.4 with equation 2 the 'boolean' bottom-up parser always needs the same amount of iterations for a string of size n and a given grammar:

- $n = 500$
- $gt = 2$

– $gnt = 4$

$$500 \cdot 2 + \frac{4}{6} \cdot (3 \cdot 500^2 \cdot (500+1) - 500 \cdot (500+1) \cdot (2 \cdot 500+1)) = 83{,}334{,}000 \text{ iterations}$$

Which is indeed the number of iterations obtained when running the code.

The 'string' bottom-up parser will also follow the same behaviour for any string pattern.

It is now possible to represent the anticiped behaviour that the 'boolean' bottom-up parser will follow for both cases.

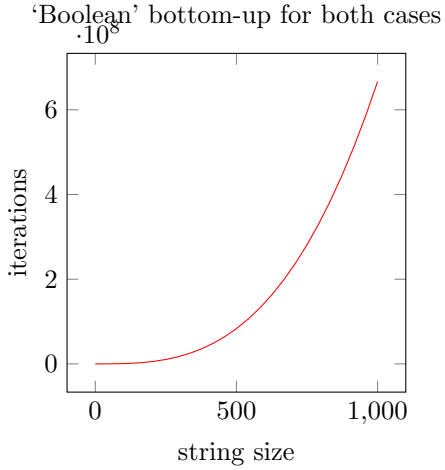


Figure 2: Anticipation of the 'boolean' bottom-up parser, grammar 1

3.2.1.2 Comparing the efficiency

The first experimentation uses the case in which the strings are starting with a right patenthese:)^n.

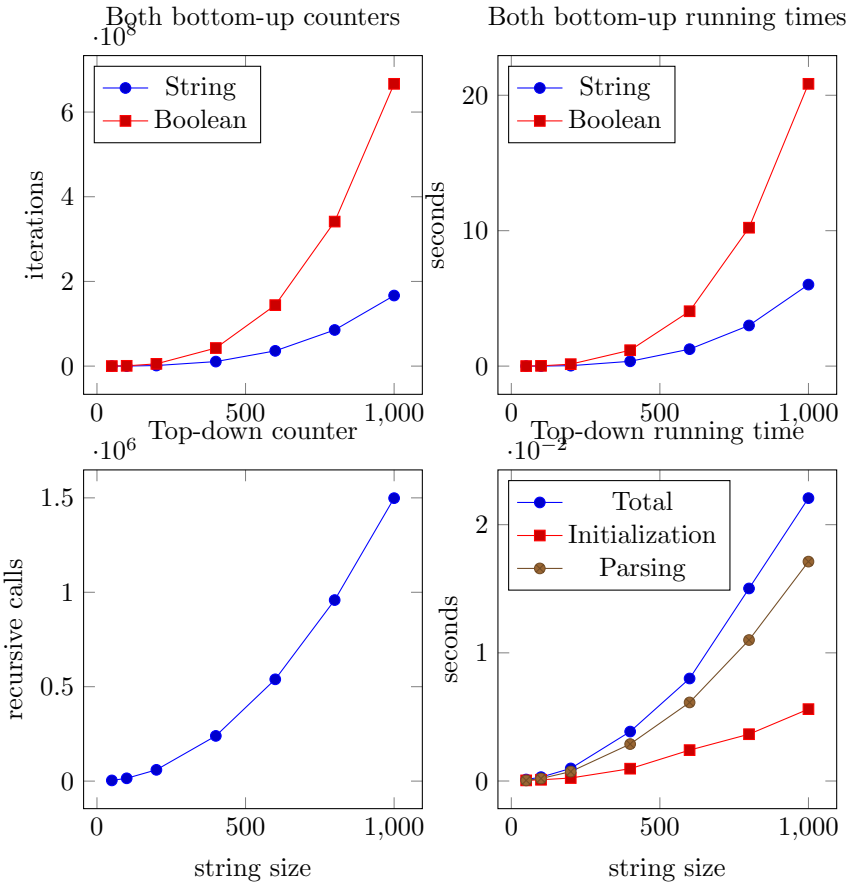


Figure 3: Both bottom-up and top-down parsers behaviours, grammar 1, case 1

The 'boolean' bottom-up parser follows exactly the predicted behaviour, the number of iterations going up following the equation 2 and its running time follows exactly the same curve since for each iteration the parser does exactly the same amount of loops. For this grammar it is slower than the 'string' version.

Here the top-down parser is faster than both bottom-up parsers. To get a more precise result the initialization and the parsing process have been timed apart from each other. We can see that the initialization time scales up a power function. The parsing time seems to follow the curve of the recursive calls counter.

It is very easy to check the complexity of the two bottom-up parsers here by resolving for both an equation of the form $y = a \cdot x^3$ by replacing x and y by the coordinates of a point of their graphs.

Those are respectively the theorical expressions of the 'boolean' and 'string' bottom-up parsers:

$$y = 2.0844 \cdot 10^{-8} \cdot x^3 \qquad y = 6.0186 \cdot 10^{-9} \cdot x^3$$

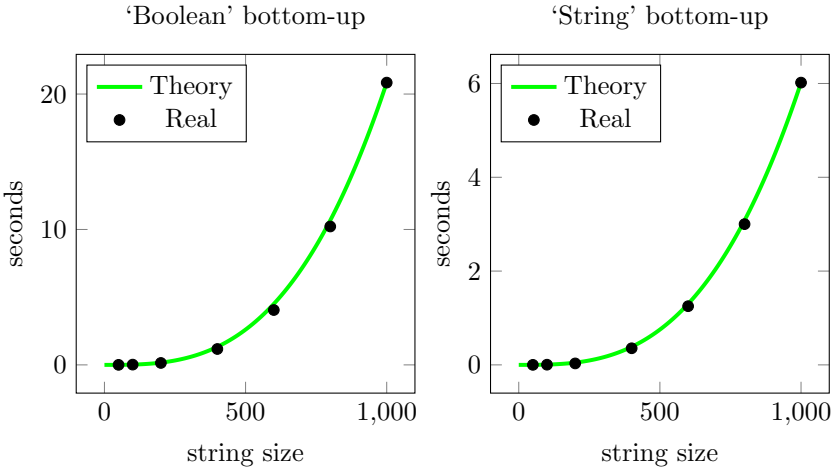


Figure 4: Checking theorical fit, both bottom-up parsers

The two functions fit almost perfectly to the points which confirms the $O(n^3)$ complexity for both bottom-up parsers.

The naive parser is very slow with that grammar, but it is possible to check its complexity too by resolving the equation $y = a \cdot 3^x$.

$$y = 2.146422 \cdot 10^{-10} \cdot 3^x$$

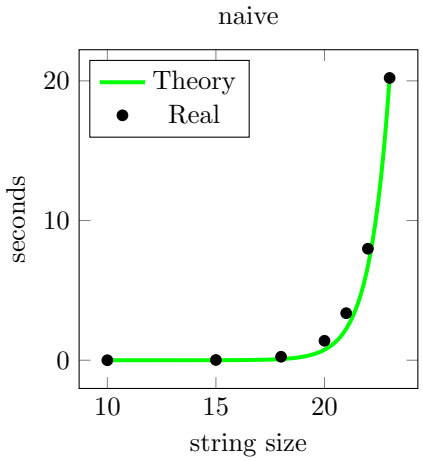


Figure 5: Checking theorical fit, naive parser

The theorical curve fits perfectly the running time of the naive parser which proves that the complexity of the naive parser is $O(3^n)$, it also shows that with that grammar this parser is very bad compared to the others.

The second pattern is the pattern only constituated by left parentheses: '(^n', what makes it interesting is the fact that it is the worst case that we can generate with this grammar for the top-down parser. The bottom-up parsers results will not be displayed in that case since they have the same behaviour as with the first case.

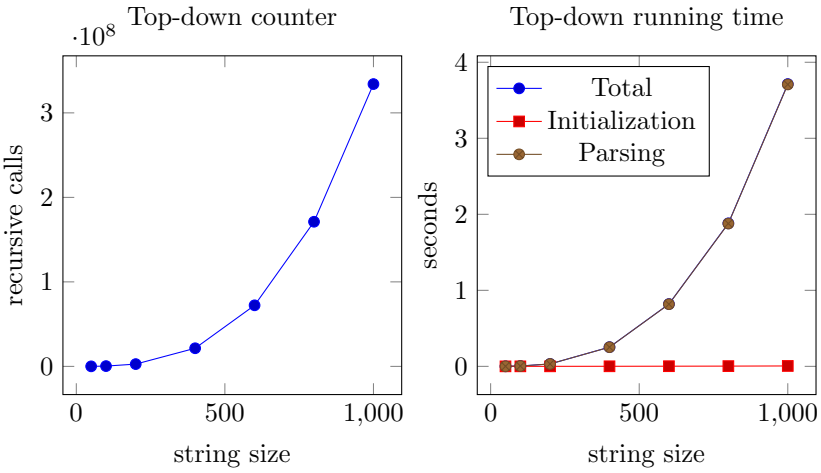


Figure 6: Top-down parser behaviour, grammar 1, case 2

As we can see on the graph above in that case the initialization time of the top-down parser is negligible, the parsing time is the one that matters. The parsing time follows exactly the same curve as the number of recursive calls. Despite the fact that this case is the worst one for the top-down parser its running time remains lower than the other parsers.

We can conclude that for this grammar the top-down parser is more efficient than both bottom-up algorithms.

Here we can verify the complexity of the top-down parser by resolving the same equation as for the two bottom-up parsers.

$$y = 3.10797 \cdot 10^{-9} \cdot x^3$$

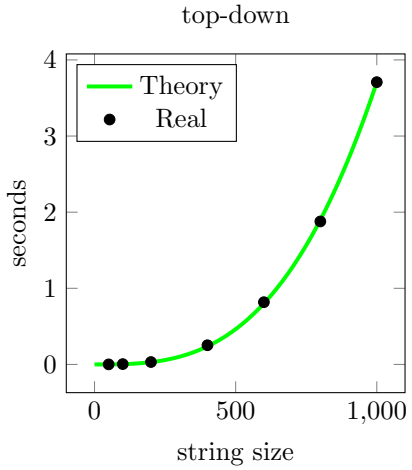


Figure 7: Checking theorical fit, grammar 1, case 2

The curve fits perfectly the points of the top-down parser which proves that the complexity of the algorithm is $O(n^3)$.

3.2.2 String starting with an 'a'

The grammar used in this section is the following:

$$S \to AB$$
$$A \to a$$
$$B \to BB|a|b$$

This grammar matches every string that begins with an 'a', no matter what comes next.

3.2.2.1 Number of iterations/recusive calls for each parser

It is very interesting because with this grammar the top-down algorithm never uses the stored values in its table, no matter the pattern used. At first one could think that this is a very bad thing, indeed since the top-down without the table is exactly the same as the naive parser it means that using this grammar they will both have the same behaviour. But what is interesting is that the naive parser with this grammar and a string of size n reacts like so:

$$recursive\ calls = \begin{cases} 2n-1 & \text{if string matches the grammar} \\ n & \text{otherwise} \end{cases}$$

It is very easy to demonstrate those two results:

1. In every pattern that does start with an 'a' the number of recursive calls is $2n-1$ because:
   - The initial call is $parse(S, 0, n)$.
   - The function will the recursively call $parse(A, 0, 1)$, which will return true since the *terminal* 'a' is equal to the character 0 of the string which is also 'a'.
   - Then the function will call a second time $parse(B, 1, n)$.
   - That function will call $parse(B, 1, 2)$. That will return true no matter the character since 'B' possesses both *terminals* 'a' and 'b'.
   - Then it will call $parse(B, 2, n)$. This will repeat itself until $parse(B, n-1, n)$ is called. At that moment it will return true.
   - At that moment the number of recursive calls is $(n-1) * 2$ plus the initial call 1, for a total of $2n-1$ recursive calls. Note that every 2 steps this algorithm goes 1 step deeper into the recursive stack, this is an important fact for the experimentations.
2. In every pattern that does not start with an 'a' the number of recursive calls is the size of the string n because:
   - The initial call is $parse(S, 0, n)$. That function will enter in a loop for k going from 1 to $n-1$.
   - The function will then recursively call $parse(A, 0, 1)$, which returns false since the *terminal* 'a' is not equal to the character 0 of the string, which is 'b'.
   - Then the function will call $parse(A, 0, k)$ until the end of the for loop. The calls will return false each time since 'A' does not possess any *non-terminals* rules.
   - So there is 1 call plus $n-1$ calls made by the first called function, for a total of n recursive calls. Note that all the sub calls are made by the first called function, which limits the maximum reached recursive depth to only 2, it will be interesting to see in the experimentations how that has a very important impact.

With a behaviour like this one the naive parser is very efficient with that grammar. Also in this section only the naive parser and both bottom-up parsers will be considered, since the naive and top-down one are the same.

With this grammar the best and worst case scenarios for the 'string' bottom-up parser are:

- The worst case is a string following this pattern: '$b\hat{}n$'.
- The best case is a string following this pattern: '$a\hat{}n$'.
- Every case that is a mix between 'a's and 'b's will have a running time situated between the best and worst case. The more 'b's there is in the string the closer the running time will be from the worst case and conversely with 'a's.

It is now poosible to compare the anticipated behaviours of the 'boolean' bottom-up and the naive parser, in both cases, using the function 2 and the two expressions from above.

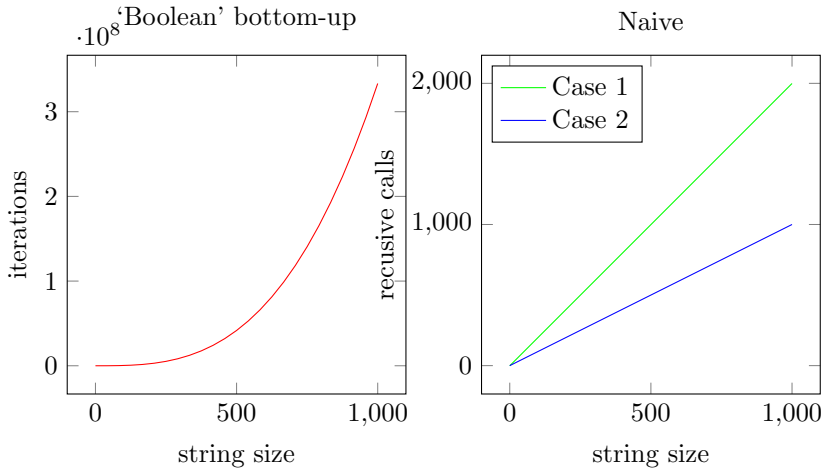


Figure 8: Anticipation of the behaviours of the parsers, grammar 2

Since the number of recursive calls of the naive parser with that grammar is linear it should be very effective compared to the 'boolean' bottom-up one. The naive parser should also be more effective than the 'string' bottom-up parser since this one also follows a power function.

3.2.2.2 Comparing the efficiency

With that grammar there is only two cases to compare, the case in which the string matches the grammar and the case when it doesn't, except for the 'string' bottom-up which has a behaviour depending on the pattern but for which those two patterns are the extrem cases.

The first used pattern is a case in which the strings match the grammar: '$a\hat{}n$'.

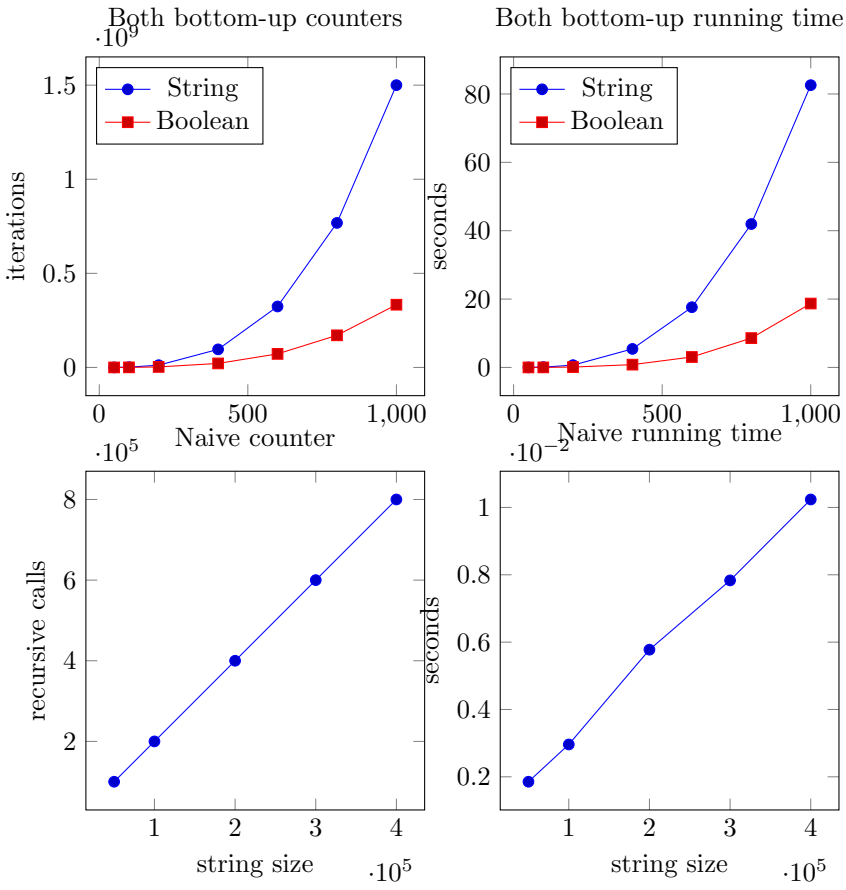


Figure 9: Both bottom-up and naive parsers behaviours, grammar 2, case 1

The running time of the naive parser compared to the two bottom-up ones with this grammar is so small that it is needed to use different string sizes in order to get relevant results.

With this grammar and that case the 'boolean' bottom-up parser is faster than the 'string' version.

A limitation of the top-down parser is that it needs to allocate memory for its table, but with strings of the size that are used in this experimentation it is not possible to allocate the memory. So even if the behaviour is theoricaly exactly the same as with the naive parser in practice with a classic computer it is not possible to parse strings that long with the top-down parser.

The running time of the naive parser is linear, which makes sense since its number of recursive calls is linear too.

An interesting thing is that even if the running time of the naive parser remains very small with big string sizes the maximum size of the recursive stack was reached for strings bigger than 410000 characters. This can be explained by the fact that when the string size n increases the algorithm goes deeper and deeper in the stack: 1 more level of depth every 2 recursive calls. This is not an issue that we will be able to see with a pattern that does not starts with a 'b', indeed in that case as explained in section 3.2.2.1, case 2, the maximum recursive depth accessed is 2, which will allow the parser to parse longer strings.

The second used pattern is a case in which the strings do not match the grammar: '$b\hat{}n$'.

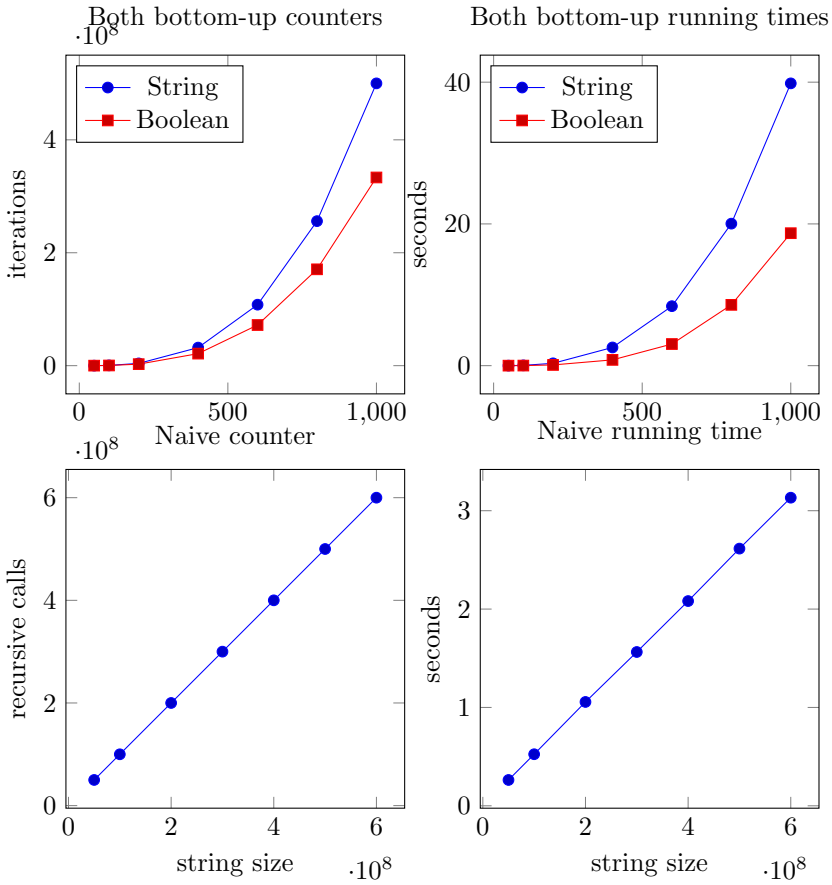


Figure 10: Both bottom-up and naive parsers behaviour, grammar 2, case 2

The curves of the 'boolean' are the same as with the last case since no matter the pattern used that parser will always have the same behaviour with that grammar. This case is the best case for the 'string' bottom-up parser and the running time is indeed lower than with the first case, however it still takes

more time than the 'boolean' version.

With that case it was possible to parse a word of 600 millions character in about 3 seconds with the naive parser, which is very impressive. The parser was maybe able to parse the word in some seconds but the longest part was in fact the time to generate the string and also the time to pass such string in parameter to some functions. The running time is once again linear as expected.

With both the best and the worst case scenarios for the 'string' bottom-up parser it is possible to display the area between the two curves. The running time of that parser will always be contained in the blue area and the more 'b's there is in the given string the more the running time will follow a close curve from the worst case and conversely.

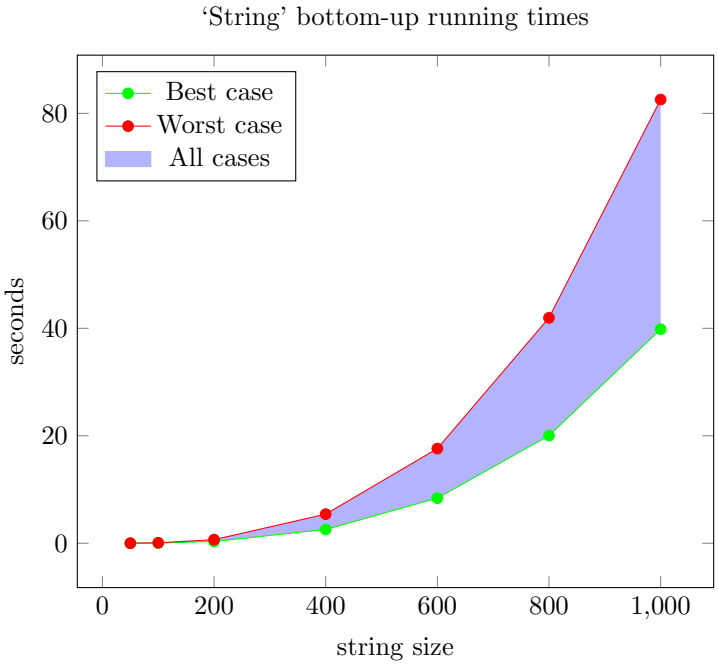


Figure 11: Running time of the 'string' bottom-up parser, grammar 3

3.2.3 String ending with an 'a'

The grammar used in this section is the following:

$$S \to BA$$
$$A \to a$$
$$B \to BB|a|b$$

3.2.3.1 Number of iterations/recursive calls for each parser

An interesting thing to notice is that with that grammar and a string of size n, no matter the used pattern, the amount of needed iterations or recursive calls will be the same for the naive, top-down and 'boolean' bottom-up parsers.

The 'string' bottom-up parser follows exactly the same behaviour as with the previous grammar.

By decomposing the behaviour of the naive parser one can find out that for any pattern it evolves following that sequence:

$$n + (n-1)^2 \tag{3}$$

It can be demonstrated like so:

- The initial call is $parse(S, 0, n)$
- Then it calls $parse(B, 0, 1)$, which returns true no matter the character.
- Then it calls $parse(A, 1, n)$, which returns false since A doesn't possess any *non-terminals*.
- Then it calls $parse(B, 0, 2)$, which returns true by doing 2 more recursive calls, $parse(B, 0, n)$ will always return true and do two more recursive calls each time it is called.
- The process ends up when $parse(A, n-1, n)$ is called, then it returns true or false depending on if the last character is an 'a' or a 'b'. No matter the result, and so the pattern, the algorithm will always make the same amount of recursive calls.
- When it ends up the number of recursive calls is 1 for the initial call, plus $n-1$ for each time it has called parse(A, x, n), plus $(n-1)^2$ for each time it has called parse(B, 0, x) and all the recursive calls it made. So the total number of recursive calls is $n + (n-1)^2$ for the naive parser.

The top-down parser follows the exact same behaviour as the naive parser, to parse a string of size n it will also need $n + (n-1)^2$ recursive calls.

However some sub-problems will not need to be recomputed by the top-down parser, it is possible to show that, for a string of size n and whatever pattern is used, it needs to solve that many sub-problems:

$$n + (n-1)^2 - \frac{(n-2) \cdot (n-1)}{2} \tag{4}$$

Here is the demonstration:

- The running process is exactly the same as for the naive one, the difference is that some results have already been computed and can be directly returned.

– The amount of results already stored in the table and then asked again increases by one for each level of recursive depth when $parse(B, 0, x)$ is called.

– This amount corresponds to a triangular number sequence: $\frac{n \cdot (n+1)}{2}$. The formula needs to be twisted a little to match the string size n and the number of asked already known results is $\frac{(n-1) \cdot (n-2)}{2}$.

– Then the total amount of solved sub-problems is just the number of recursive calls minus the expression above: $n + (n-1)^2 - \frac{(n-2) \cdot (n-1)}{2}$.

It is now possible to compare the anticipated behaviours of the naive, top-down and bottom-up parsers, using functions 3 and 2.

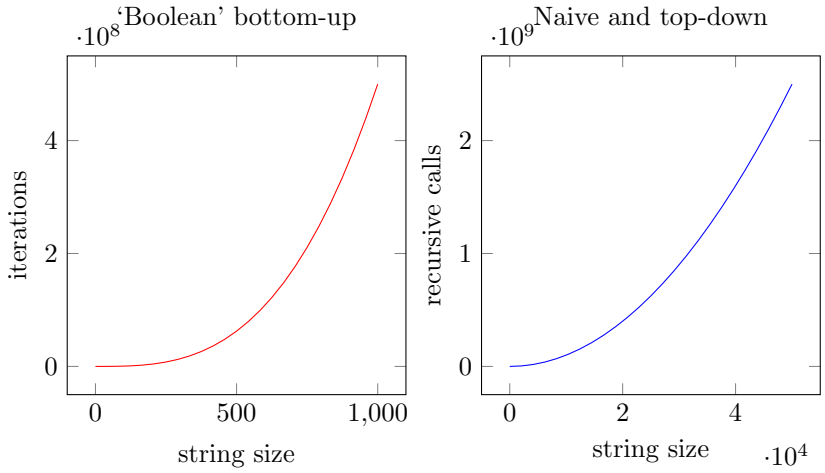


Figure 12: Anticipation of the behaviours of the parsers, grammar 3

The 'boolean' bottom-up parser follows a power function with an exponent of three as usual, but the top-down and naive parsers will follow a power function with an exponent of two with that grammar.

3.2.3.2 Comparing the efficiency

Except for the 'string' bottom-up parser, with this grammar there is no particular cases, any pattern will be as long to parse as another by any of the three other parsers, the first used pattern is a^n.

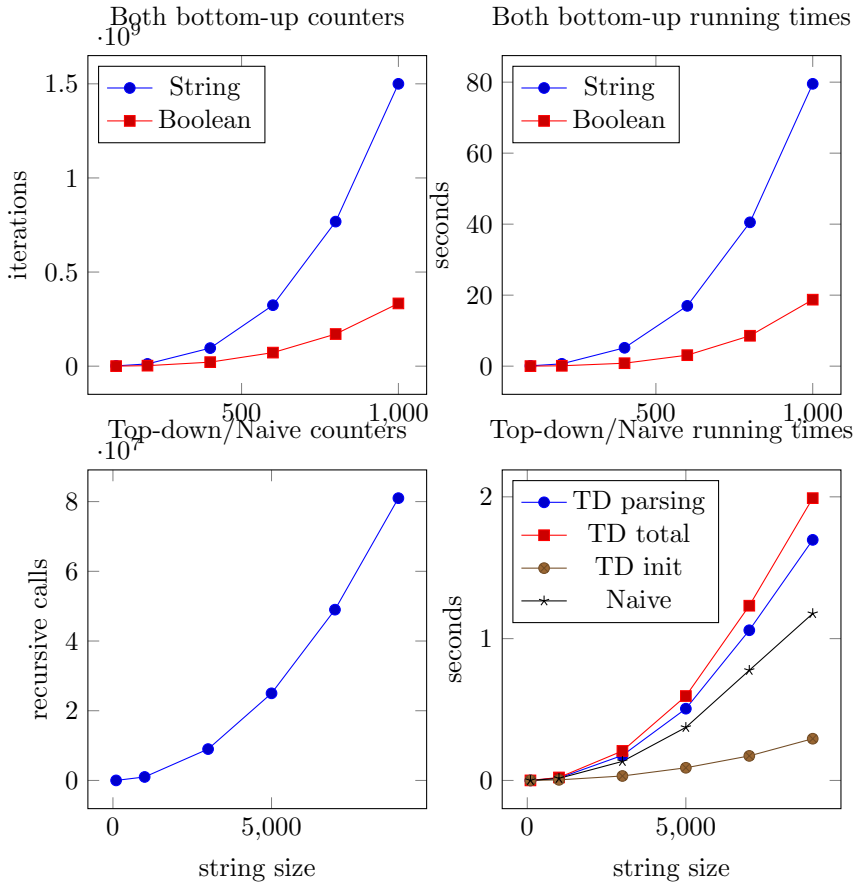


Figure 13: The 4 parsers behaviours, grammar 3

With strings of sizes greater than $9,000$ the top-down parser could not allocate the table so the used strings did not exceed that size.

The same string sizes are used for the top-down and naive parsers, it is possible to see on the graphs above that despite the fact that the naive parser and the top-down parsers need the same amount of recursive calls the naive parser has a lower running time.

The only parser that needs a various number of iterations is the 'string' bottom-up parser, the pattern '$a\hat{}n$' was as previously its worst case, its best case with that grammar is the pattern '$b\hat{}n$' as previously. It is not a suprise to find out that the 'string' bottom-up parser is slower than the 'boolean' version with that grammar, since it follows the same behaviour as before.

Next are the results obtained with the 'string' bottom-up parser, using the last results for the 'boolean' bottom-up parser as reference.

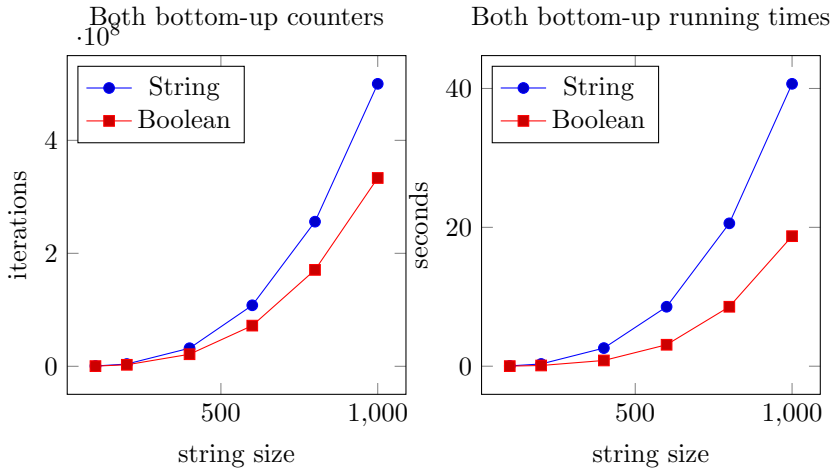


Figure 14: The 'string' bottom-up parser worst case, grammar 3

As with the previous grammar the best case of the 'string' bottom-up parser is once again slower than the 'boolean' version. The running time follows exactly the same curve as the number of iterations for the 'string' bottom-up parser.

It would be possible to plot the area containing all the possible cases of the 'string' bottom-up parser once again but it would be the exact same figure as before since the running time of the parser for each pattern remains the same with both grammars.

Chapter 4

Linear grammar

4.1 Grammar preprocessing

The goal was to implement an algorithm that would convert a context-free grammar into a grammar in the Chomsky normal-form, the most simple choice was to develop a new version of the Grammar class.

On its initialization the class is given a file that contains the grammar, which can be in any form.

The first thing that the class does is to replace all the *non-terminal* variables by an index, starting at 0 for the *start symbol* and increasing by 1 for every variable, the *terminals* are also put in negative so there is no chance of confusion between the variables.

Then the grammar is converted to the Chomsky normal-form, following the same process as manually explained is section 1.1.2.

Finally the converted grammar is splitted into a set of *terminal* rules and a set of *non-terminal* rules.

The final grammar is stocked and can be accessed through the following variables:

- A 3D vector of integers named 'non_terminal_rules'.
    1. The first dimension represents the *non-terminal* variable, converted as an index starting at 0 for the *start symbol*.
    2. Each element of the second dimension is a right hand of the variable from the first dimension.
    3. The pairs of *non-terminals* are stocked in the third dimension.
- A 2D vector of integers named 'terminal_rules' that allows access to the *terminal* rules.
- Since there are no *non-terminal* rules for every variable, when it is the case the variable is added to the string 'nt_variables'.

– The string ‘t_variables’ is exactly the same as the one above but for the *terminal* rules.

For example if one gives the following grammar to the program:

$$S \to Ac|b$$
$$A \to aS|aB$$
$$B \to bS$$

This is how the grammar will be stocked:

– ‘non_terminal_rules’ =

$$0 \to 0\ 0|2\ 1|2\ 3 \qquad 2 \to$$
$$1 \to 0\ 3 \qquad 3 \to$$

– ‘terminal_rules’ =

$$0 \to \qquad 2 \to ($$
$$1 \to \qquad 3 \to)$$

– ‘nt_variables’ = ”0 1”

– ‘t_variables’ = ”2 3”

The symbol ‘|’ indicates the separation of the second dimension of the vector. As we can see in the variables content above the *non-terminals* are converted in indices.

That preprocessing allows one to give a linear grammar to the program without bothering to convert anything by advance.

Since the conversion of a linear grammar in the Chomsky normal-form often add new rules there is no reason to think that it could in any way fasten the parsing process to automaticly convert a linear grammar instead of directly giving the algorithm a Chomsky normal-form grammar.

4.2 Implementation of a linear grammar parser

Converting a grammar to the Chomsky normal-form cannot improve the running time of the parsers but a parser that directly parses strings using a linear grammar can be faster.

To implement a linear grammar parser the easiest choice was to use the actual top-down parser as basis for the new parser. The program now contains a LinearGrammar class which simply stocks a linear grammar in the *rules* variable without doing any modifications to the characters, the variable *non_terminals* is a string linking a *non-terminal* to its index in the rules. The parser is concived in such a way that it can parse a string using a linear grammar but also a grammar in the Chomsky normal-form. Since the algorithm is based on the top-down parser if it is given a grammar in the Chomsky normal-form it will need the same number of resursive calls as the top-down parser would.

Algorithm 5 Linear grammar parser

```
1: table is a 3d array of 0
2: s ← input_string
3: procedure PARSE(var, i, j)                          ▷ Recursive parser function
4:     if table[var, i, j]! = 0 then
5:         return (table[var, i, j] == 1)
6:     end if
7:     for all right_hand ∈ rules[var] do
8:         if right_hand.size() == j − i ∧ s.size() − i ≥ j − i then
9:             bool is_equal ← true
10:            for k ← 0 : k < right_hand.size() do
11:                if s[i + k] ≠ right_hand[k] then
12:                    is_equal ← false
13:                    break
14:                end if
15:            end for
16:            if is_equal then
17:                table[var, i, j] ← 1
18:                return true
19:            end if
20:        end if
21:        var_1, var_2, var_3 are characters
22:        if right_hand.size() == 1 then
23:            var_1 ← right_hand[0]
24:            if var_1 ≥ 'A' ∧ var_1 ≤ 'Z' then
25:                if parse(non_terminals.find(var_1), i, j) then
26:                    table[var, i, j] ← 1
27:                    return true
28:                end if
29:            end if
30:        else if j − i ≥ 2 ∧ right_hand.size() == 2 then
31:            var_1 ← right_hand[0]
32:            var_2 ← right_hand[1]
33:            if var_1 ≥ 'A' ∧ var_1 ≤ 'Z' ∧ var_2 ≥ 'A' ∧ var_2 ≤ 'Z' then
34:                for k ← i + 1 : k < j do
35:                    if parse(non_terminals.find(var_1), i, k) ∧
   parse(non_terminals.find(var_2), k, j) then
36:                        table[var, i, j] ← 1
37:                        return true
38:                    end if
39:                end for
40:            else if (var_1 < 'A' ∨ var_1 > 'Z') ∧ var_2 ≥ 'A' ∧ var_2 ≤ 'Z'
   then
41:                if s[i] == var_1 then
42:                    if parse(non_terminals.find(var_2), i + 1, j) then
43:                        table[var, i, j] ← 1
44:                        return true
45:                    end if
46:                end if
```

```
47:             else if var_1 ≥ 'A' ∧ var_1 ≤ 'Z' ∧ (var_2 < 'A' ∨ var_2 > 'Z') then
48:                 if s[j − 1] == var_2 then
49:                     if parse(non_terminals.find(var_1), i, j − 1) then
50:                         table[var, i, j] ← 1
51:                         return true
52:                     end if
53:                 end if
54:             end if
55:         else if j − i ≥ 3 ∧ right_hand.size() == 3 then
56:             var_1 ← right_hand[0]
57:             var_2 ← right_hand[1]
58:             var_3 ← right_hand[2]
59:             if var_1 ≥ 'A'∧var_1 ≤ 'Z'∧(var_2 < 'A'∨var_2 > 'Z')∧var_3 ≥ 'A' ∧ var_3 ≤ 'Z' then
60:                 for k ← i + 1; k < j − 1 do
61:                     if parse(non_terminals.find(var_1), i, k) ∧ var_2 == s[k] ∧ parse(non_terminals.find(var_3), k + 1, j) then
62:                         table[var, i, j] ← 1
63:                         return true
64:                     end if
65:                 end for
66:             else if (var_1 < 'A' ∨ var_1 > 'Z') ∧ var_2 ≥ 'A' ∧ var_2 ≤ 'Z') ∧ (var_3 < 'A' ∨ var_3 > 'Z') then
67:                 if var_1 == s[i]∧parse(non_terminals.find(var_2), i+1, j − 1) ∧ var_3 == s[j − 1] then
68:                     table[var, i, j] ← 1
69:                     return true
70:                 end if
71:             end if
72:         end if
73:     end for
74:     table[var, i, j] ← 2
75:     return false
76: end procedure
```

Since this parser allows one to parse strings using a linear grammar instead of the Chomsky normal-form version it should be more efficient, indeed the linear grammar possesses less production rules and so the parser should need less recursive calls.

On the other hand since this parser is based on the same logic as the top-down parser its complexity is the same as that last one: $O(n^3)$. That means that this parser will be more efficient in most practical cases but the same as the top-down parser in some other cases.

4.3 Experimentation and comparison with previous parsers

For this experimentation the first used linear grammar is the following.

$$S \to Ac|b$$
$$A \to aS|aB$$
$$B \to bS$$

The linear parser is given the grammar above as it appears and the other parsers are given the automatically converted grammar as showed in section 4.1.

The used pattern to obtain the following results is '$a\hat{}x\ b\ c\hat{}x$', the strings generated by this pattern always match the grammar.

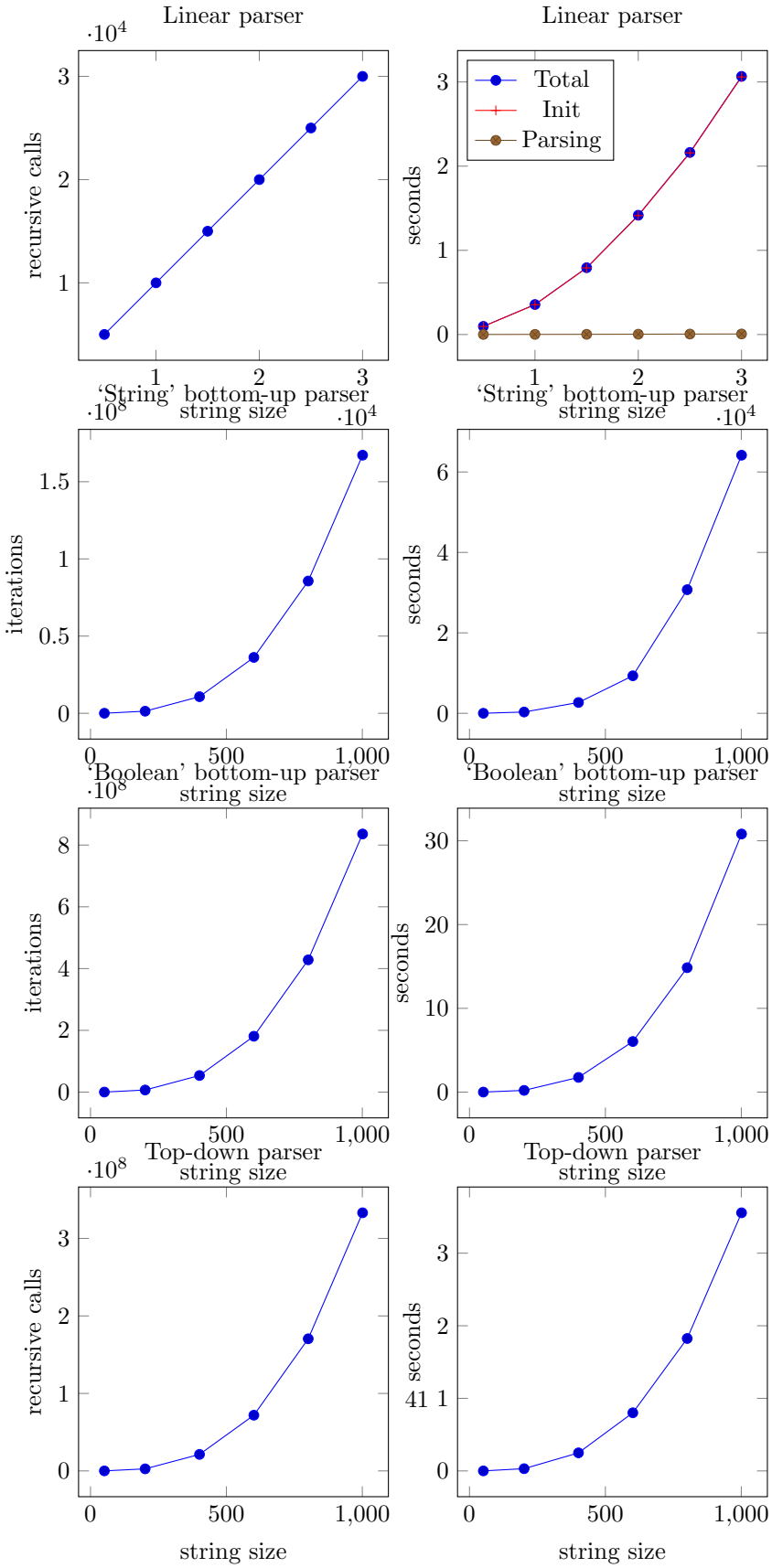


Figure 15: Results with linear grammar 1, case 1

The results above about the 'String' and 'boolean' bottom-up and top-down parsers are similar to results obtained before with various grammars.

Is is interesting to see that the linear parser needs n recursive calls to parse a string of size n which matches this grammar, the expected running time is expected to be linear too. However the initialization process of the memoization table takes basicaly all the running time, the actual parsing time is too small to be relevant. This shows a limitation of the linear parser in that case, the memoization table limits the input strings sizes that can be parsed. Even with input strings bigger than 30000 characters the parsing time is so low that even if it seems to follow a linear behaviour it is not relevant.

The next results are obtained using a set of strings that does not match the grammar, the pattern is '$a\hat{}n$'.

The 'boolean' bottom-up parser having the same behaviour for a given grammar the results are exactly the same for this pattern so they are not displayed.

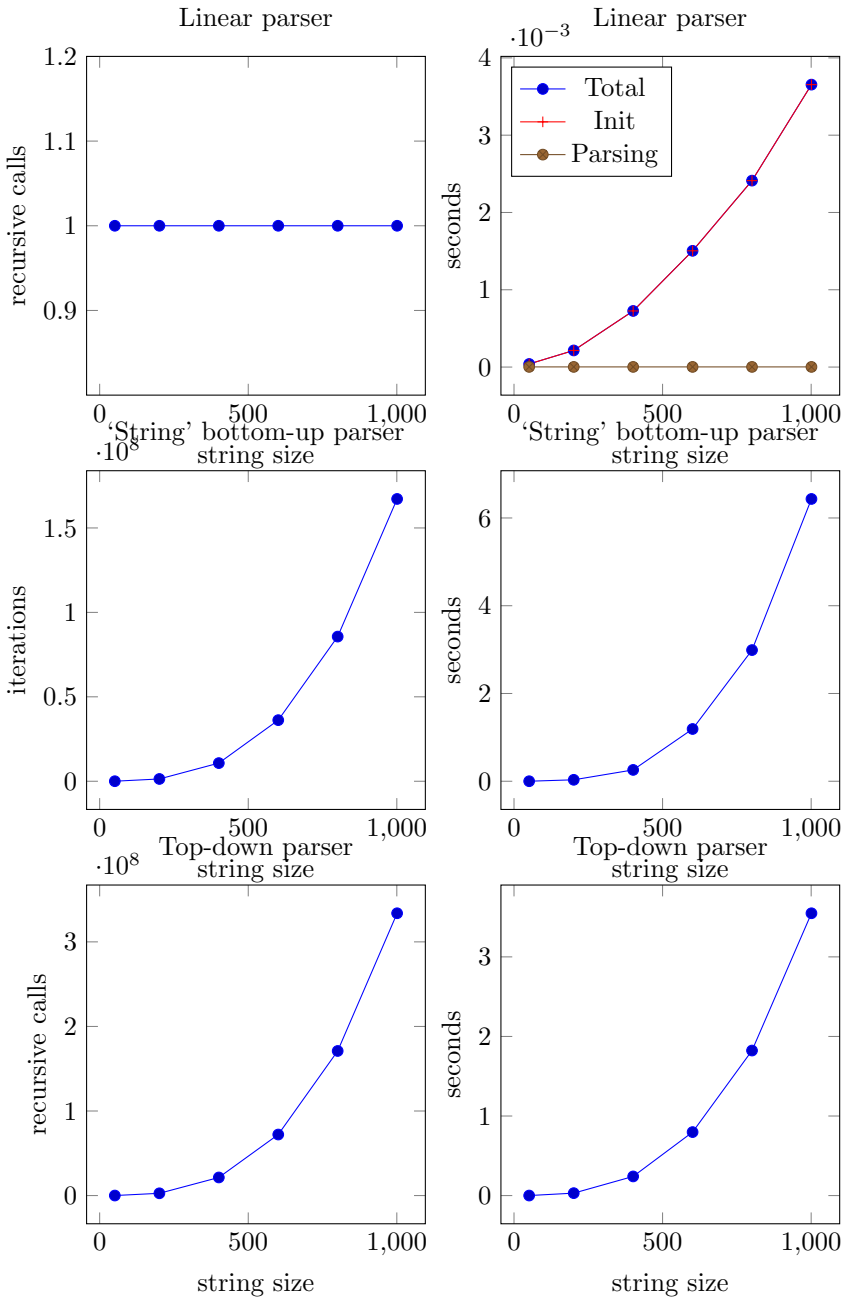


Figure 16: Results with linear grammar 1, case 2

With this pattern the top-down needs a few more recursive calls than before, the change is so little that its running time is not touched by it.

With that pattern the number of recursive calls for the linear parser is always 1 since it just goes through the right-hands of 'S' and then return false. Once again the running time is high because of the initialization process, with that case the parsing time is constant and so small that it is not possible to represent it properly.

A limitation of the linear parser for this grammar in this form is the memoization table, indeed the parsing process is so short that it would be better to just use a naive version of this parser by not using the table. This for two reason, the first obvious reason is that with this grammar the long part is to allocate the memory for the table. The second reason is that once again with string longer than 40000 characters the used machine is not able to allocate enough memory for the table.

4.3.1 Well balanced parentheses

To really compare the efficiency of the linear parser with the other ones the best thing is to compare how they deal with the well balanced parenthese grammar.

The linear form of this grammar is the following.

$$S \to SS|(S)|()$$

The linear parser results will be compared with the top-down parser results in section 3.2.1.

With this grammar the linear parser behaves differently depending on the pattern of the strings. The first considered case is the pattern '$)\hat{}n$'.

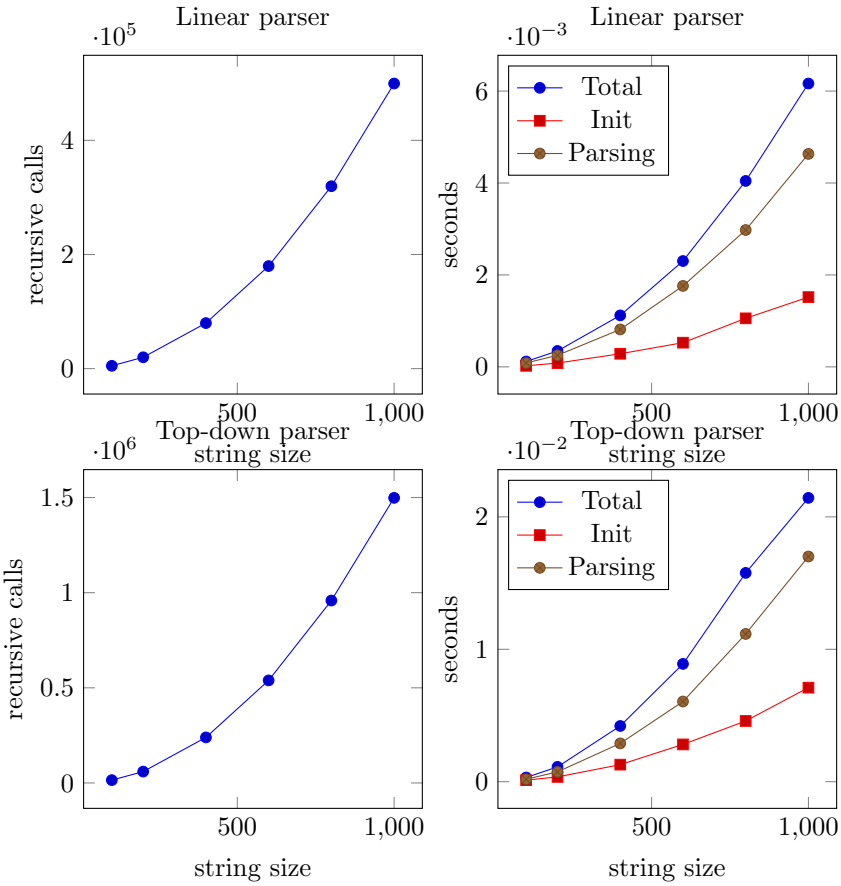


Figure 17: Results with linear grammar 2, case 1

Here the two parsers running times follow the same behaviour as their number of recursive calls, the linear parser needs less recursive calls and so has a lower running time than the top-down parser.

For the second case the same parsers are used and the used pattern is '($\hat{}n$'.

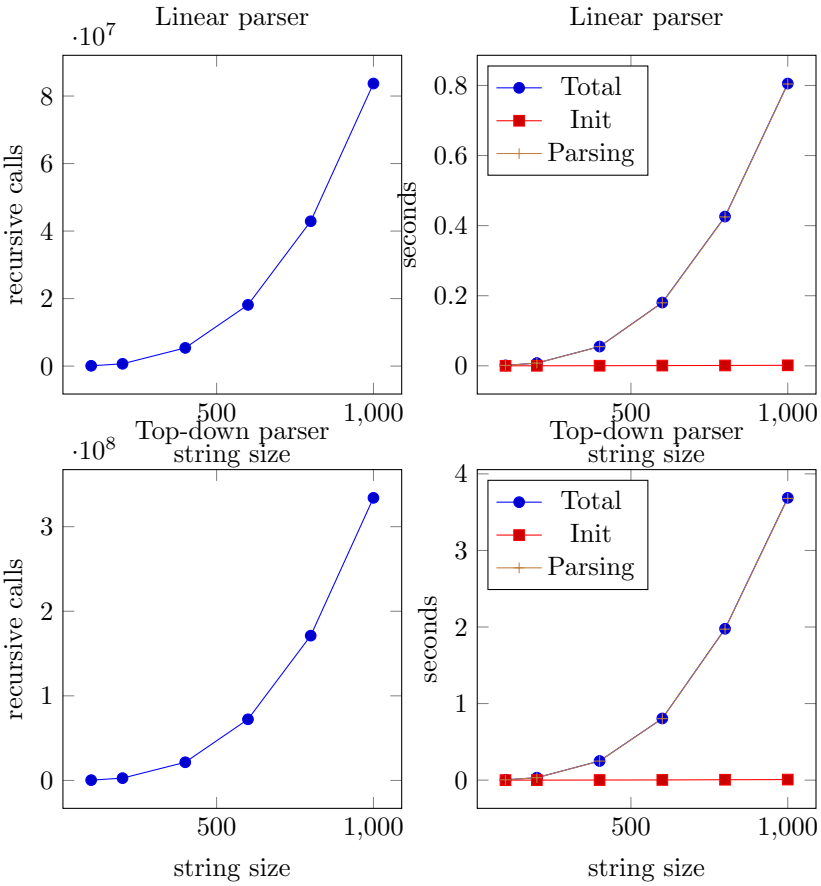


Figure 18: Results with linear grammar 2, case 2

Here the initialization time is negligible before the parsing time for both parsers.

With that grammar the linear parser is more efficient than the top-down parser which was by far the most efficient compared to the other parser in section 3.2.1.

Here is the verification of the complexity of the linear parser.

$$y = 8.04034 \cdot 10^{-10} \cdot x^3$$

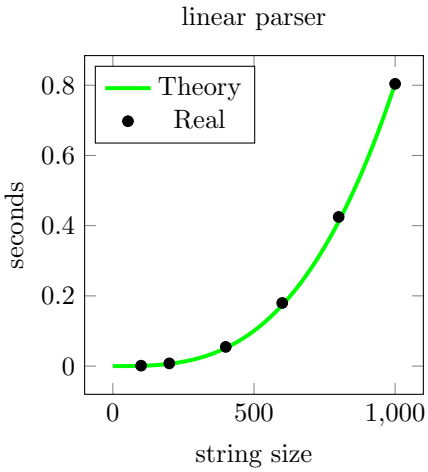


Figure 19: Checking theorical fit, linear parser

The real points fit perfectly the theorical curve which proves that the complexity of the linear parser really is $O(n^3)$.

Chapter 5

Error correction

Since the corrections are directly effectued on the *primitive symbols* the choice that seemed to be the simplest was to use the top-down parser as basis for the implementation of the error correction parser.

5.1 Implementation of the character replacement

The first step was to implement the character replacement, this is a modified version of the previous code of the top-down parser:

Algorithm 6 Top-down parser character modification

```
1: table is a 3d array of 0
2: s ← input_string
3: procedure PARSE(var, i, j)                          ▷ Recursive parser function
4:     if table[var, i, j]! = 0 then
5:         return (table[var, i, j] == 1)
6:     end if
7:     if i == j − 1 then
8:         for all t ∈ terminal_rule[var] do
9:             if t == s[i] then
10:                table[var, i, j] ← 1
11:                return table[var, i, j]
12:            end if
13:        end for
14:        if terminal_rule[var].size() > 0 then
15:            table[var, i, j] ← 2
16:            return table[var, i, j]
17:        end if
18:    else
19:        min ← INT_MAX
20:        for all nt ∈ non_terminal_rules[var] do
21:            for k ← i + 1; k < j do
22:                declare res_1 and res_2
23:                if (res_1 ← parse(nt[0], i, k)) < INT_MAX ∧ (res_2 ←
   parse(nt[1], k, j)) < INT_MAX then
24:                    if res_1 + res_2 − 1 < min then
25:                        min ← res_1 + res_2 − 1
26:                    end if
27:                end if
28:                end for
29:            end for
30:            table[var, i, j] ← min
31:            return table[var, i, j]
32:        end if
33:        table[var, i, j] ← INT_MAX
34:        return table[var, i, j]
35:    end procedure
```

All the differences with the initial top-down parser are in red on this pseudo-code.

With those modifications the top-down parser now returns an integer when it parses a string:

- If the value is 1 it means that the string can be parsed without any modifications.
- If the value is $x > 1$ it means that the string can be parsed with $x - 1$ modifications.
- If the value is INT_MAX, the maximum value of an integer, it means that the string cannot be parsed, no matter how much modifications could be made.

5.2 Implementation of the character deletion

Once the character modification is done the next step is to implement character deletion.

Algorithm 7 Top-down parser character deletion

```
1: table is a 3d array of 0
2: s ← input_string
3: procedure PARSE(var, i, j)                          ▷ Recursive parser function
4:     if table[var, i, j]! = 0 then
5:         return (table[var, i, j] == 1)
6:     end if
7:     if i == j − 1 then
8:         for all t ∈ terminal_rule[var] do
9:             if t == s[i] then
10:                table[var, i, j] ← 1
11:                return table[var, i, j]
12:            end if
13:        end for
14:        if terminal_rule[var].size() > 0 then
15:            table[var, i, j] ← 2
16:            return table[var, i, j]
17:        end if
18:    else
19:        min ← INT_MAX
20:        del_1 ← parse(var, i + 1, j)
21:        del_2 ← parse(var, i, j − 1)
22:        for all nt ∈ non_terminal_rules[var] do
23:            for k ← i + 1; k < j do
24:                declare res_1 and res_2
25:                if (res_1 ← parse(nt[0], i, k)) < INT_MAX ∧ (res_2 ← parse(nt[1], k, j)) <
                       INT_MAX then
26:                    if res_1 + res_2 − 1 < min then
27:                        min ← res_1 + res_2 − 1
28:                    end if
29:                end if
30:            end for
31:        end for
32:        if del_2 < del_1 then
33:            del_1 ← del_2
34:        end if
35:        if del_1 < min then
36:            min ← del_1
37:        end if
38:        table[var, i, j] ← min
39:        return table[var, i, j]
40:    end if
41:    table[var, i, j] ← INT_MAX
42:    return table[var, i, j]
43: end procedure
```

That algorithm returns the number of modifications plus deletions, the next step is to be able to return the corrected version of the input string. Once such corrected string can be generated it will be trivial to know how much deletions and modifications were both needed.

5.3 Recuperation of the corrected string

In order to proceed to the recuperation of the corrected string a structure is used to contain the result of the parse function plus the corresponding corrected string. The memoization table of the top-down parser will now contains datas of this structure. The parse function will also now return a *parse_result* structure, it will also take the corrected string at its actual state as parameter.

This is is the pseudo-code with the modifications needed to be able to retrieve the corrected string.

Algorithm 8 Top-down parser correction and string recuperation

```
1: table is a 3d array of parse_result
2: s ← input_string
3: procedure PARSE(var, i, j, sc)                    ▷ Recursive parser function
4:     if table[var, i, j]! = 0 then
5:         return (table[var, i, j] == 1)
6:     end if
7:     if i == j − 1 then
8:         for all t ∈ terminal_rule[var] do
9:             if t == s[i] then
10:                table[var, i, j].result ← 1
11:                table[var, i, j].string ← sc
12:                return table[var, i, j]
13:            end if
14:        end for
15:        if terminal_rule[var].size() > 0 then
16:            table[var, i, j].result ← 2
17:            table[var, i, j].string ← terminal_rule[var][0]
18:            return table[var, i, j]
19:        end if
20:    else
21:        min ← INT_MAX
22:        new_s ← ""
23:        del_1 ← parse(var, i + 1, j, sc.substr(1, sc.size() − 1))
24:        del_2 ← parse(var, i, j − 1, sc.substr(0, sc.size() − 1))
25:        for all nt ∈ non_terminal_rules[var] do
26:            for k ← i + 1; k < j do
27:                res_1 ← parse(nt[0], i, k, sc.substr(0, k − i))
28:                if res_1.result < INT_MAX then
29:                    res_2 ← parse(nt[1], k, j, sc.substr(k − i, j − k))
30:                    if res_2.result < INT_MAX then
31:                        if res_1.result + res_2.result − 1 < min then
32:                            min ← res_1.result + res_2.result − 1
33:                            new_s ← res_1.string + res_2.string
34:                        end if
35:                    end if
36:                end if
37:            end for
38:        end for
39:        table[var, i, j] ← min_parse_result(del_1, del_2, min, new_string)
40:        return table[var, i, j]
41:    end if
42:    table[var, i, j].result ← INT_MAX
43:    table[var, i, j].string ← sc
44:    return table[var, i, j]
45: end procedure
```

Such an algorithm will end by returning the corrected string plus the sum of all needed corrections and deletions to arrive to this result. It is then trivial to separate the number of needed deletions from the number of modifications, just by comparing the size from the initial string to the size of the corrected string.

The running time of this algorithm will always be longer than for the top-down parser since the algorithm has to test the two possible deletions before entering the main loop, which the top-down parser does not do. However the complexity of this algorithm remains the same as for the top-down parser, $O(n^3)$.

5.4 Obtained results

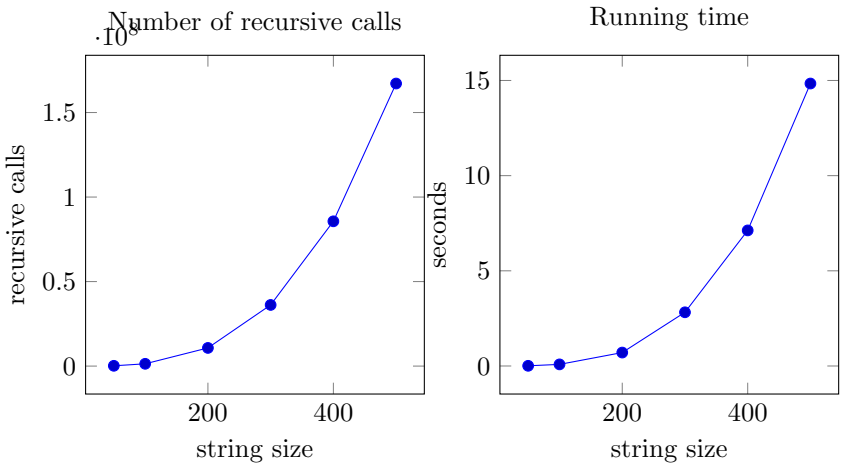
5.4.1 Well balanced parenthesis

The goal of the new parser is to correct an input string to make it match the given grammar if it is possible, the first experimentation is realised on the well balanced parenthesis.

Initially it was planned to check the parser's performances on the following cases:

- Strings in which the parser must modify half of the characters, that case corresponds to whatever pattern gives a string that does not contain any substring that would match the grammar, for example '$)\hat{}n$', or '$(\hat{}n$', or '$)\hat{}n/2$ $(\hat{}n/2$'.
- Strings that match the grammar, then the parser does not have to modify the input string.
- Strings that match the grammar or not, containing an odd number of characters so the parser needs to delete one.

It was interesting to notice that the correction top-down parser followed exactly the same behaviour no matter what pattern the given string follows.



The running time plot follows exactly the same curve as the number of recursive calls, which was expected.

Here is the verification of the complexity of the correction parser.

$$y = 1.187152 \cdot 10^{-7} \cdot x^3$$

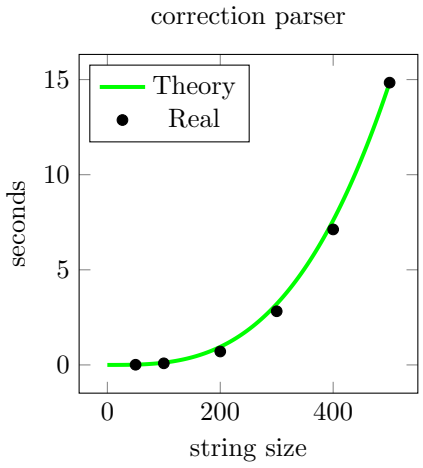
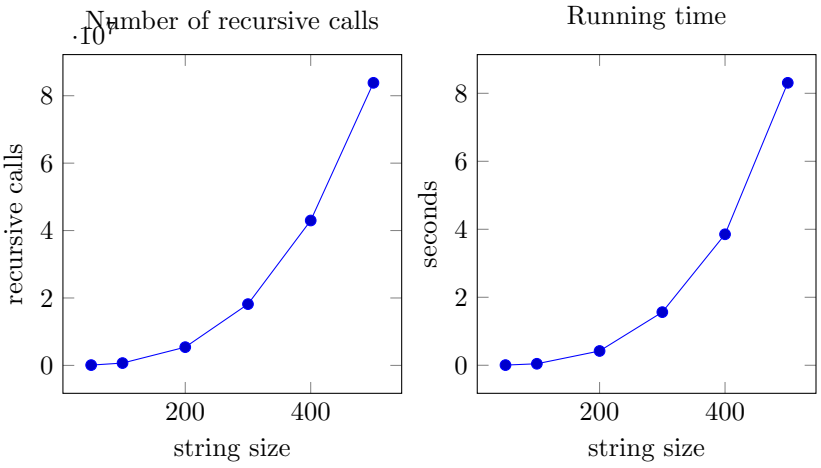


Figure 20: Checking theorical fit, correction parser

The running time behaviour of the correction parser follows exactly the theorical fit, which proves that its complexity is $O(n^3)$.

5.4.2 Strings starting with an 'a'

The next expimentation is on the grammar that matches strings that start with an 'a'. With that grammar the parser follows exactly the same behaviour as before, it always need the same amount of recursive calls for a string of size n, no matter the pattern.



Here the parser does not need as much recursive calls as with the previous grammar but it still follows exactly the same behaviour.

The number of solved subproblems solved for this grammar follows the next sequence, with n the size of the input string.

$$1 + \frac{(n-1) \cdot (3 \cdot (n-1) + 1)}{2} + (n-1) \cdot 2$$

It has no use to keep going with the experimentation since no matter the grammar or the given string this parser always follows the same behaviour.

Conclusion

The efficiency of the different implementations of the Cocke-Younger-Kasami algorithm depends on the used development approach for a big part but also of the given grammar.

The naive parser with its complexity of $O(3^n)$ is by far the less efficient of the implementations, there are only a few cases for which this parser can be used and compete with the others and those cases are not the cases usually met in real life.

All the other parsers are implemented using Dynamic programming and have a complexity of $O(n^3)$. However, if with grammars in the Chomsky normal-form the two bottom-up, the top-down and the linear parsers can be more or less equivalent depending on the cases, when a linear grammar is used the linear parser is in most cases more efficient than the others.

The correction parser also has a complexity of $O(n^3)$ since it uses the top-down paradigm, but in practice it is less efficient than all the other implementations since it does more recursive calls to test different modifications. Even if this last parser is slower than the others it can be used in way more useful ways, those kind of correction parsers are especially used in DNA and RNA analysis for example.

List of Algorithms

List of Figures

Bibliography

[1] Noam Chomsky. Three models for the description of language, 1956. https://chomsky.info/wp-content/uploads/195609-.pdf.

[2] Noam Chomsky. On certain formal properties of grammars, 1959. https://www.sciencedirect.com/science/article/pii/S0019995859903626.

[3] Jeffrey D Ullman John E Hopcroft. Introduction to automata theory, languages and computation, 1939. https://mcdtu.files.wordpress.com/2017/03/introduction-to-automata-theory.pdf.

[4] Neil James Alexander Sloane. The online encyclopedia of integer sequences, 2018. https://oeis.org/A024023.

[5] Christian N. S. Pedersen James W. J. Anderson, Zsuzsanna Sukosd and Jotun Hein. Technical report: An n-free-passes cyk algorithm for error-correction and the prediction of non-canonical base-pairs in rna secondary structure, 2013. https://pdfs.semanticscholar.org/8eca/9d5ae7ca732ec6537dc26c0a5e24311fd3e4.pdf.

[6] Donald E. Knuth. Semantics of context-free languages, 1968. https://www.csee.umbc.edu/courses/331/fall16/01/resources/papers/Knuth67AG.pdf.

[7] Marius Nicolae Sanguthevar Rajasekaran. An error correcting parser for context free grammar that take less than cubic time, 2014. https://arxiv.org/pdf/1406.3405.pdf.

Appendix

Grammar

Grammar.h

```
#ifndef CYK_ALGORITHM_NEWGRAMMAR_H
#define CYK_ALGORITHM_NEWGRAMMAR_H

#include <iostream>
#include <fstream>
#include <vector>
#include <iomanip>

class Grammar {
private:
    std::vector<std::vector<std::vector<int>>> raw_grammar;
    std::vector<std::vector<std::vector<int>>> converted_grammar;
    std::string non_terminals;
    bool start_symbol_in_raw_right_hand;

    void load_raw_grammar(std::string basic_string);
    void convert_to_chomsky();
    void replace_raw_non_terminals();
    void eliminate_start_symbol();
    void display_grammar(std::vector<std::vector<std::vector<int>>> grammar);
    void replace_terminals_by_new_non_terminals();
    void eliminate_right_hands_with_too_much_nt();
    void eliminate_unit_rules();
    void fill_final_variables();
public:
    std::vector<std::vector<std::vector<int>>> non_terminal_rules;
    std::vector<std::vector<int>> terminal_rules;
    std::string nt_variables, t_variables;

    explicit Grammar(std::string, bool = true);

    void display_raw_grammar();
    void display_converted_grammar();
    void display_final_grammar();
};


#endif
```

Grammar.cpp

```
#include "Grammar.h"

Grammar::Grammar(std::string filename, bool convert) {
```

```
4       this->non_terminals = "";
5       this->nt_variables = "";
6       this->t_variables = "";
7       this->start_symbol_in_raw_right_hand = false;
8       this->load_raw_grammar(filename);
9       this->converted_grammar = this->raw_grammar;
10      if (convert)
11          this->convert_to_chomsky();
12      this->fill_final_variables();
13  }
14
15  void Grammar::display_raw_grammar() {
16      std::cout << "\nRaw grammar :\n";
17      this->display_grammar(this->raw_grammar);
18  }
19
20  void Grammar::display_converted_grammar() {
21      std::cout << "\nConverted grammar :\n";
22      this->display_grammar(this->converted_grammar);
23  }
24
25  void Grammar::display_final_grammar() {
26      std::cout << "\nFinal grammar :\n";
27      std::cout << "Non-terminal rules :\n";
28      for (int var: this->nt_variables) {
29          std::cout << var << " -> ";
30          for (int right_hand(0); right_hand < this->non_terminal_rules[var].
                size(); right_hand++) {
31              std::cout << std::setw(5);
32              std::cout << this->non_terminal_rules[var][right_hand][0]
33                        << " " << this->non_terminal_rules[var][right_hand][1];
34              std::cout << " ";
35          }
36          std::cout << "\n";
37      }
38      std::cout << "\nTerminal rules :\n";
39      for (int var: this->t_variables) {
40          std::cout << var << " -> ";
41          for (int right_hand(0); right_hand < this->terminal_rules[var].size()
                ; right_hand++) {
42              std::cout << std::setw(3);
43              std::cout << char(this->terminal_rules[var][right_hand]) << " ";
44          }
45          std::cout << "\n";
46      }
47  }
48
49  void Grammar::display_grammar(std::vector<std::vector<std::vector<int>>>
        grammar) {
50      for (int i(0); i < grammar.size(); i++) {
51          std::cout << i << " -> ";
52          for (int j(0); j < grammar[i].size(); j++) {
53              std::cout << std::setw(5);
54              for (int k(0); k < grammar[i][j].size(); k++) {
55                  std::cout << grammar[i][j][k] << " ";
56              }
57              std::cout << " ";
58          }
59          std::cout << "\n";
60      }
61  }
62
63  void Grammar::load_raw_grammar(std::string filename) {
64      std::ifstream file(filename);
65      if (file.is_open()) {
66          std::string line;
67          while (std::getline(file, line)) {
68              this->non_terminals += line[0];
```

```
69              line = line.substr(3, line.length());
70              std::vector<std::vector<int>> right_hands;
71              while(line.length() > 0) {
72                  std::vector<int> right_hand;
73                  int separator = line.find("|");
74                  if(separator > line.length()) {
75                      for (int i(0); i < line.size(); i++) {
76                          right_hand.push_back((line[i] < 'A' || line[i] > 'Z')
                                ? -(int(line[i])) : line[i]);
77                      }
78                      line = "";
79                  } else {
80                      std::string sub(line.substr(0, separator));
81                      for (int i(0); i < sub.size(); i++) {
82                          right_hand.push_back((sub[i] < 'A' || sub[i] > 'Z') ?
                                -(int(sub[i])) : sub[i]);
83                      }
84                      line = line.substr(separator + 1, line.length());
85                  }
86                  right_hands.push_back(right_hand);
87              }
88              this->raw_grammar.push_back(right_hands);
89          }
90          file.close();
91      }
92      this->replace_raw_non_terminals();
93  }
94
95  void Grammar::replace_raw_non_terminals() {
96      for (int i(0); i < this->raw_grammar.size(); i++) {
97          for (int j(0); j < this->raw_grammar[i].size(); j++) {
98              for (int k(0); k < this->raw_grammar[i][j].size(); k++) {
99                  int index(this->non_terminals.find(this->raw_grammar[i][j][k
                        ]));
100                 if (index <= this->non_terminals.size())
101                     this->raw_grammar[i][j][k] = index;
102                 if (index == 0)
103                     this->start_symbol_in_raw_right_hand = true;
104             }
105         }
106     }
107 }
108
109 void Grammar::convert_to_chomsky() {
110     if (this->start_symbol_in_raw_right_hand)
111         this->eliminate_start_symbol();
112     this->replace_terminals_by_new_non_terminals();
113     this->eliminate_right_hands_with_too_much_nt();
114     this->eliminate_unit_rules();
115 }
116
117 void Grammar::eliminate_start_symbol() {
118     for (int i(0); i < this->converted_grammar.size(); i++) {
119         for (int j(0); j < this->converted_grammar[i].size(); j++) {
120             for (int k(0); k < this->converted_grammar[i][j].size(); k++) {
121                 if (this->converted_grammar[i][j][k] >= 0)
122                     this->converted_grammar[i][j][k]++;
123             }
124         }
125     }
126     std::vector<std::vector<int>> start_symbol_right_hand = {{1}};
127     this->converted_grammar.insert(this->converted_grammar.begin(),
          start_symbol_right_hand);
128 }
129
130 void Grammar::replace_terminals_by_new_non_terminals() {
131     int grammar_size(this->converted_grammar.size());
132     for (int i(0); i < grammar_size; i++) {
```

```
133         for (int j(0); j < this->converted_grammar[i].size(); j++) {
134             if (this->converted_grammar[i][j].size() > 1) {
135                 for (int k(0); k < this->converted_grammar[i][j].size(); k++)
                        {
136                     if (this->converted_grammar[i][j][k] < 0) {
137                         std::vector<std::vector<int>> new_right_hand = {{this
                                ->converted_grammar[i][j][k]}};
138                         this->converted_grammar.push_back(new_right_hand);
139 
140                         int terminal_to_replace(this->converted_grammar[i][j
                                ][k]);
141                         int replace_value(int(this->converted_grammar.size())
                                 - 1);
142                         for (int l(0); l < grammar_size; l++) {
143                             for (int m(0); m < this->converted_grammar[l].
                                    size(); m++) {
144                                 for (int n(0); n < this->converted_grammar[l
                                        ][m].size(); n++) {
145                                     if (this->converted_grammar[l][m][n] ==
                                            terminal_to_replace)
146                                         this->converted_grammar[l][m][n] =
                                                replace_value;
147                                 }
148                             }
149                         }
150                     }
151                 }
152             }
153         }
154     }
155 }
156 
157 void Grammar::eliminate_right_hands_with_too_much_nt() {
158     for (int i(0); i < this->converted_grammar.size(); i++) {
159         for (int j(0); j < this->converted_grammar[i].size(); j++) {
160             if (this->converted_grammar[i][j].size() > 2) {
161                 std::vector<std::vector<int>> new_right_hand_parent;
162                 std::vector<int> new_right_hand_child;
163                 for (int k(1); k < this->converted_grammar[i][j].size(); k++)
                        {
164                     new_right_hand_child.push_back(this->converted_grammar[i
                            ][j][k]);
165                 }
166                 new_right_hand_parent.push_back(new_right_hand_child);
167                 this->converted_grammar.push_back(new_right_hand_parent);
168                 this->converted_grammar[i][j] = {this->converted_grammar[i][j
                        ][0],
169                                                  int(this->converted_grammar.
                                                         size()) - 1};
170             }
171         }
172     }
173 }
174 
175 void Grammar::eliminate_unit_rules() {
176     for (int i(this->converted_grammar.size() - 1); i >= 0; i--) {
177         for (int j(0); j < this->converted_grammar[i].size(); j++) {
178             if (this->converted_grammar[i][j].size() == 1 && this->
                    converted_grammar[i][j][0] >= 0) {
179                 int nt(this->converted_grammar[i][j][0]);
180                 this->converted_grammar[i].erase(this->converted_grammar[i].
                        begin() + j);
181                 for (int k(0); k < this->converted_grammar[nt].size(); k++) {
182                     this->converted_grammar[i].push_back(this->
                            converted_grammar[nt][k]);
183                 }
184             }
185         }
```

```
    }
}

void Grammar::fill_final_variables() {
    for (int i(0); i < this->converted_grammar.size(); i++) {
        std::vector<std::vector<int>> non_terminal_right_hands;
        std::vector<int> terminal_right_hands;
        bool nt(false), t(false);
        for (int j(0); j < this->converted_grammar[i].size(); j++) {
            if (this->converted_grammar[i][j].size() == 1) {
                terminal_right_hands.push_back(-this->converted_grammar[i][j
                    ][0]);
                t = true;
            } else {
                std::vector<int> non_terminal_right_hand = {this->
                    converted_grammar[i][j][0],
                                                            this->
                                                                converted_grammar
                                                                [i][j][1]};
                non_terminal_right_hands.push_back(non_terminal_right_hand);
                nt = true;
            }
        }
        this->non_terminal_rules.push_back(non_terminal_right_hands);
        this->terminal_rules.push_back(terminal_right_hands);
        if (nt)
            this->nt_variables += i;
        if (t)
            this->t_variables += i;
    }
}
```

Abstract class parser

Parser.h

```
#ifndef CYK_ALGORITHM_PARSER_H
#define CYK_ALGORITHM_PARSER_H

#include "Grammar.h"

typedef struct parser_result {
    unsigned long long int counter;
    int result;
    std::string corrected_string;
} parser_result;

class Parser {
public:
    Grammar grammar;
    static unsigned long long int counter;

    Parser(Grammar);

    virtual parser_result parse(std::string) = 0;
    virtual std::string get_name() = 0;
};


#endif
```

Parser.cpp

```
#include "Parser.h"

```

```
3  Parser::Parser(Grammar grammar) : grammar(grammar) {}
4
5  unsigned long long int Parser::counter = 0;
```

Naive parser

NaiveParser.h

```
1  #ifndef CYK_ALGORITHM_RECURSIVEPARSER_H
2  #define CYK_ALGORITHM_RECURSIVEPARSER_H
3
4  #include "Parser.h"
5
6  class NaiveParser : public Parser {
7  private:
8      std::string string;
9      bool parse(int, int, int);
10
11 public:
12     explicit NaiveParser(Grammar);
13     parser_result parse(std::string) override;
14     std::string get_name() override;
15 };
16
17
18 #endif
```

NaiveParser.cpp

```
1  #include "NaiveParser.h"
2
3  NaiveParser::NaiveParser(Grammar grammar) : Parser(grammar) {}
4
5  parser_result NaiveParser::parse(std::string string) {
6      this->counter = 0;
7      this->string = string;
8
9      parser_result parser_result;
10     parser_result.result = parse(0, 0, string.length());
11     parser_result.counter = this->counter;
12
13     return parser_result;
14 }
15
16 bool NaiveParser::parse(int variable, int i, int j) {
17     this->counter++;
18     if (i == j - 1) {
19         for (int right_hand(0); right_hand < this->grammar.terminal_rules[
               variable].size(); right_hand++) {
20             if (this->grammar.terminal_rules[variable][right_hand] == this->
                   string[i]) {
21                 return true;
22             }
23         }
24     } else {
25         for (int right_hand(0); right_hand < this->grammar.non_terminal_rules
               [variable].size(); right_hand++) {
26             for (int k(i + 1); k < j; k++) {
27                 if (this->parse(this->grammar.non_terminal_rules[variable][
                       right_hand][0], i, k) &&
28                     this->parse(this->grammar.non_terminal_rules[variable][
                           right_hand][1], k, j))
29                 {
30                     return true;
31                 }
```

```
            }
        }
    }
    return false;
}

std::string NaiveParser::get_name() {
    return "Naive parser";
}
```

Top-down parser

TopDownParser.h

```
#ifndef CYK_ALGORITHM_TOPDOWNPARSER_H
#define CYK_ALGORITHM_TOPDOWNPARSER_H

#include "Parser.h"

class TopDownParser : public Parser {
private:
    int ***table;
    std::string string;
    bool parse(int, int, int);

public:
    explicit TopDownParser(Grammar);
    parser_result parse(std::string) override;
    std::string get_name() override;
};


#endif
```

TopDownParser.cpp

```
#include "TopDownParser.h"

TopDownParser::TopDownParser(Grammar grammar) : Parser(grammar) {}

parser_result TopDownParser::parse(std::string string) {
    this->counter = 0;
    this->string = string;

    this->table = new int **[this->grammar.terminal_rules.size()];
    for(int i = 0; i < this->grammar.terminal_rules.size(); i++) {
        this->table[i] = new int *[string.length()];
        for(int j = 0; j < string.length(); j++)
            this->table[i][j] = new int [string.length()]();
    }

    parser_result parser_result;
    parser_result.result = parse(0, 0, string.length());
    parser_result.counter = this->counter;
    return parser_result;
}

bool TopDownParser::parse(int variable, int i, int j) {
    this->counter++;

    if (this->table[variable][i][j - 1] != 0)
        return (this->table[variable][i][j - 1] == 1);

    if (i == j - 1) {
```

```
29          for (int right_hand(0); right_hand < this->grammar.terminal_rules[
                variable].size(); right_hand++) {
30              if (this->grammar.terminal_rules[variable][right_hand] == this->
                    string[i]) {
31                  this->table[variable][i][j - 1] = 1;
32                  return true;
33              }
34          }
35      } else {
36          for (int right_hand(0); right_hand < this->grammar.non_terminal_rules
                [variable].size(); right_hand++) {
37              for (int k(i + 1); k < j; k++) {
38                  if (this->parse(this->grammar.non_terminal_rules[variable][
                        right_hand][0], i, k) &&
39                      this->parse(this->grammar.non_terminal_rules[variable][
                            right_hand][1], k, j))
40                  {
41                      this->table[variable][i][j - 1] = 1;
42                      return true;
43                  }
44              }
45          }
46      }
47      this->table[variable][i][j - 1] = 2;
48      return false;
49  }
50
51  std::string TopDownParser::get_name() {
52      return "Top down parser";
53  }
```

‘Boolean’ bottom-up parser

BottomUpBoolParser.h

```
1   #ifndef CYK_ALGORITHM_BOTTOMUPBOOLPARSER_H
2   #define CYK_ALGORITHM_BOTTOMUPBOOLPARSER_H
3
4   #include "Parser.h"
5
6   class BottomUpBoolParser : public Parser {
7   public:
8       explicit BottomUpBoolParser(Grammar);
9
10      parser_result parse(std::string) override;
11      std::string get_name() override;
12  };
13
14
15  #endif
```

BottomUpBoolParser.cpp

```
1   #include "BottomUpBoolParser.h"
2
3   BottomUpBoolParser::BottomUpBoolParser(Grammar grammar) : Parser(grammar) {}
4
5   parser_result BottomUpBoolParser::parse(std::string string) {
6       this->counter = 0;
7
8       bool ***matrix;
9       matrix = new bool **[string.length()];
10      for(int i = 0; i < string.length(); i++) {
11          matrix[i] = new bool *[string.length()];
12          for(int j = 0; j < string.length(); j++)
```

```
13              matrix[i][j] = new bool [this->grammar.terminal_rules.size()]();
14      }
15
16      int step(0);
17      for (int i(0); i < string.length(); i++) {
18          for (int var: this->grammar.t_variables) {
19              for (int right_hand(0); right_hand < this->grammar.terminal_rules
                    [var].size(); right_hand++) {
20                  if (this->grammar.terminal_rules[var][right_hand] == string[i
                        ]) {
21                      matrix[0][i][var] = true;
22                  }
23                  this->counter++;
24              }
25          }
26      }
27
28      step++;
29      while (string.length() - step > 0) {
30          for (int i(0); i < string.length() - step; i++) {
31              for (int j(0); j < step; j++) {
32                  for (int var: this->grammar.nt_variables) {
33                      for (int right_hand(0); right_hand < this->grammar.
                            non_terminal_rules[var].size(); right_hand++) {
34                          int var_1(this->grammar.non_terminal_rules[var][
                                right_hand][0]);
35                          int var_2(this->grammar.non_terminal_rules[var][
                                right_hand][1]);
36
37                          bool bool_1(matrix[j][i][var_1]);
38                          bool bool_2(matrix[step - j - 1][i + j + 1][var_2]);
39
40                          if (bool_1 && bool_2)
41                              matrix[step][i][var] = true;
42
43                          this->counter++;
44                      }
45                  }
46              }
47          }
48          step++;
49      }
50
51      parser_result parser_result;
52      parser_result.result = matrix[string.length() - 1][0][0];
53      parser_result.counter = this->counter;
54      return parser_result;
55  }
56
57  std::string BottomUpBoolParser::get_name() {
58      return "Boolean bottom up parser";
59  }
```

‘String’ bottom-up parser

BottomUpParser.h

```
1   #ifndef CYK_ALGORITHM_ITERATIVEPARSER_H
2   #define CYK_ALGORITHM_ITERATIVEPARSER_H
3
4   #include "Parser.h"
5   #include <iomanip>
6
7   class BottomUpParser : public Parser {
8   private:
9       std::string concatenate(std::string, std::string);
```

```
    std::string generate_combinations(std::string, std::string);
    std::string search_production(int char_1, int char_2);
    void display_matrix(std::string **, int);

public:
    explicit BottomUpParser(Grammar);

    parser_result parse(std::string) override;
    std::string get_name() override;
};


#endif
```

BottomUpParser.cpp

```
#include "BottomUpParser.h"

BottomUpParser::BottomUpParser(Grammar grammar) : Parser(grammar) {}

parser_result BottomUpParser::parse(std::string string) {
    this->counter = 0;

    std::string **matrix;
    matrix = new std::string *[string.length()];
    for(int i = 0; i < string.length(); i++)
        matrix[i] = new std::string[string.length()];

    int step(0);
    for (int i(0); i < string.length(); i++) {
        for (int var: this->grammar.t_variables) {
            for (int right_hand(0); right_hand < this->grammar.terminal_rules
                 [var].size(); right_hand++) {
                if (this->grammar.terminal_rules[var][right_hand] == string[i
                    ]) {
                    std::string variable("");
                    variable = var;
                    matrix[step][i] = concatenate(matrix[step][i], variable);
                }
                this->counter++;
            }
        }
    }

    step++;
    while (string.length() - step > 0) {
        for (int i(0); i < string.length() - step; i++) {
            for (int j(0); j < step; j++) {
                std::string cell_1(matrix[j][i]), cell_2(matrix[step - (j +
                    1)][i + j + 1]);
                if (cell_1 != "" && cell_2 != "")
                    matrix[step][i] = concatenate(matrix[step][i],
                        generate_combinations(cell_1, cell_2));
                this->counter++;
            }
        }
        step++;
    }

    parser_result parser_result;
    parser_result.result = (matrix[string.length() - 1][0].find(char(0)) <=
         string.length());
    parser_result.counter = this->counter;

    //display_matrix(matrix, string.length());

    return parser_result;
```

```
}

std::string BottomUpParser::concatenate(std::string a, std::string b) {
    std::string result = a;
    for(int i(0); i < b.length(); i++) {
        if(result.find(b[i]) > result.length()) {
            result += b[i];
        }
    }
    return (result);
}

std::string BottomUpParser::generate_combinations(std::string cell_1, std::
    string cell_2) {
    std::string result("");
    for(int i(0); i < cell_1.length(); i++) {
        for(int j(0); j < cell_2.length(); j++) {
            result += search_production(cell_1[i], cell_2[j]);
        }
    }

    return result;
}

std::string BottomUpParser::search_production(int char_1, int char_2) {
    std::string result("");
    for (int var: this->grammar.nt_variables) {
        for (int right_hand(0); right_hand < this->grammar.non_terminal_rules
            [var].size(); right_hand++) {
            if (this->grammar.non_terminal_rules[var][right_hand][0] ==
                char_1 &&
                this->grammar.non_terminal_rules[var][right_hand][1] ==
                    char_2)
            {
                std::string variable("");
                variable = var;
                result = concatenate(result, variable);
            }
            this->counter++;
        }
    }
    return result;
}

void BottomUpParser::display_matrix(std::string ** matrix, int size) {
    for(int i(0); i < size; i++) {
        std::cout << "  ";
        for(int j(0); j < size; j++) {
            std::cout << std::setw(8);
            for (int k(0); k < matrix[i][j].length(); k++) {
                std::cout << int(matrix[i][j][k]) << " ";
            }
        }
        std::cout << "\n";
    }
}

std::string BottomUpParser::get_name() {
    return "String bottom up parser";
}
```

Top-down correction parser

TopDownParserCorrection.h

```
#ifndef CYK_ALGORITHM_TOPDOWNPARSERCORRECTION_H
```

```
#define CYK_ALGORITHM_TOPDOWNPARSERCORRECTION_H

#include <climits>
#include "Parser.h"

typedef struct parse_result {
    int result;
    std::string corrected_string;
} parse_result;

class TopDownParserCorrection : public Parser {
private:
    parse_result ***table;
    std::string string;
    parse_result parse(int, int, int, std::string);
    parse_result get_min_parse_result(parse_result, parse_result, int, std::
        string);

public:
    explicit TopDownParserCorrection(Grammar);
    parser_result parse(std::string) override;
    std::string get_name() override;
};


#endif
```

TopDownParserCorrection.cpp

```
#include "TopDownParserCorrection.h"

TopDownParserCorrection::TopDownParserCorrection(Grammar grammar) : Parser(
    grammar) {}

parser_result TopDownParserCorrection::parse(std::string string) {
    this->counter = 0;
    this->string = string;

    this->table = new parse_result **[this->grammar.terminal_rules.size()];
    for(int i = 0; i < this->grammar.terminal_rules.size(); i++) {
        this->table[i] = new parse_result *[string.length()];
        for(int j = 0; j < string.length(); j++)
            this->table[i][j] = new parse_result [string.length()]();
    }

    parser_result parser_result;
    parse_result parse_result(this->parse(0, 0, string.length(), string));
    parser_result.result = parse_result.result;
    parser_result.corrected_string = parse_result.corrected_string;
    parser_result.counter = this->counter;

    return parser_result;
}

parse_result TopDownParserCorrection::parse(int variable, int i, int j, std::
    string string_to_correct) {
    this->counter++;

    if (this->table[variable][i][j - 1].result != 0)
        return this->table[variable][i][j - 1];

    if (i == j - 1) {
        for (int right_hand(0); right_hand < this->grammar.terminal_rules[
            variable].size(); right_hand++) {
            if (this->grammar.terminal_rules[variable][right_hand] == this->
                string[i]) {
                this->table[variable][i][j - 1].result = 1;
```

```
35                  this->table[variable][i][j - 1].corrected_string =
                        string_to_correct;
36                  return this->table[variable][i][j - 1];
37              }
38          }
39          if (this->grammar.terminal_rules[variable].size() > 0) {
40              this->table[variable][i][j - 1].result = 2;
41              string_to_correct[0] = char(this->grammar.terminal_rules[variable
                    ][0]);
42              this->table[variable][i][j - 1].corrected_string =
                    string_to_correct;
43              return this->table[variable][i][j - 1];
44          }
45      } else {
46          int min(INT_MAX);
47          std::string new_string("");
48
49          parse_result del_1(this->parse(variable, i + 1, j,
50                                         string_to_correct.substr(1,
                                               string_to_correct.size() -
                                               1)));
51          if (del_1.result < INT_MAX)
52              del_1.result++;
53          parse_result del_2(this->parse(variable, i, j - 1,
54                                         string_to_correct.substr(0,
                                               string_to_correct.size() -
                                               1)));
55          if (del_2.result < INT_MAX)
56              del_2.result++;
57
58          for (int right_hand(0); right_hand < this->grammar.non_terminal_rules
                    [variable].size(); right_hand++) {
59              for (int k(i + 1); k < j; k++) {
60                  int var_1(this->grammar.non_terminal_rules[variable][
                        right_hand][0]);
61                  parse_result res_1(this->parse(var_1, i, k, string_to_correct
                        .substr(0, k - i)));
62
63                  if (res_1.result < INT_MAX) {
64                      int var_2(this->grammar.non_terminal_rules[variable][
                            right_hand][1]);
65                      parse_result res_2(this->parse(var_2, k, j,
                            string_to_correct.substr(k - i, j - k)));
66                      if (res_2.result < INT_MAX) {
67                          if (min > res_1.result + res_2.result - 1) {
68                              min = res_1.result + res_2.result - 1;
69                              new_string = res_1.corrected_string + res_2.
                                    corrected_string;
70                          }
71                      }
72                  }
73              }
74          }
75
76          this->table[variable][i][j - 1] = get_min_parse_result(del_1, del_2,
                min, new_string);
77          return this->table[variable][i][j - 1];
78      }
79      this->table[variable][i][j - 1].result = INT_MAX;
80      this->table[variable][i][j - 1].corrected_string = string_to_correct;
81      return this->table[variable][i][j - 1];
82  }
83
84  std::string TopDownParserCorrection::get_name() {
85      return "Top down error correction parser";
86  }
87
88  parse_result TopDownParserCorrection::get_min_parse_result(parse_result res_1
```

```
    , parse_result res_2, int min,
                                                        std::string string
                                                            ) {
    if (res_2.result < res_1.result)
        res_1 = res_2;

    if (res_1.result > min) {
        res_1.result = min;
        res_1.corrected_string = string;
    }

    return res_1;
}
```

Linear grammar

LinearGrammar.h

```
#ifndef CYK_ALGORITHM_LINEARGRAMMAR_H
#define CYK_ALGORITHM_LINEARGRAMMAR_H

#include <iomanip>
#include <vector>
#include <fstream>
#include <iostream>

class LinearGrammar {
public:
    std::string non_terminals;
    std::vector<std::vector<std::string>> rules;

    explicit LinearGrammar(std::string);
    void display_rules();
};


#endif
```

LinearGrammar.cpp

```
#include "LinearGrammar.h"

LinearGrammar::LinearGrammar(std::string filename) {
    std::ifstream file(filename);
    if (file.is_open()) {
        std::string line;
        while (std::getline(file, line)) {
            this->non_terminals += line[0];
            line = line.substr(3, line.length());
            std::vector<std::string> right_hands;
            while(line.length() > 0) {
                int separator = line.find("|");
                if(separator > line.length()) {
                    right_hands.push_back(line);
                    line = "";
                } else {
                    std::string sub(line.substr(0, separator));
                    right_hands.push_back(sub);
                    line = line.substr(separator + 1, line.length());
                }
            }
            this->rules.push_back(right_hands);
        }
        file.close();
    }
```

```
}

void LinearGrammar::display_rules() {
    std::cout << "Linear grammar :\n";
    for (int i(0); i < this->rules.size(); i++) {
        std::cout << i << " -> ";
        for (int j(0); j < this->rules[i].size(); j++) {
            std::cout << std::setw(5);
            std::cout << this->rules[i][j];
            std::cout << " ";
        }
        std::cout << "\n";
    }
}
```

Linear parser

LinearParser.h

```
#ifndef CYK_ALGORITHM_LINEARPARSER_H
#define CYK_ALGORITHM_LINEARPARSER_H

#include "Parser.h"
#include "LinearGrammar.h"

class LinearParser {
private:
    int ***table;
    LinearGrammar grammar;
    unsigned long long int counter;
    std::string string;

public:
    explicit LinearParser(LinearGrammar);

    parser_result parse(std::string);
    std::string get_name();
    bool parse(int, int, int);
};


#endif
```

LinearParser.cpp

```
#include "LinearParser.h"

LinearParser::LinearParser(LinearGrammar grammar) : grammar(grammar) {}

std::string LinearParser::get_name() {
    return "Linear parser";
}

parser_result LinearParser::parse(std::string string) {
    this->counter = 0;
    this->string = string;

    clock_t begin, end;
    begin = clock();
    this->table = new int **[this->grammar.rules.size()];
    for(int i = 0; i < this->grammar.rules.size(); i++) {
        this->table[i] = new int *[string.length()];
        for(int j = 0; j < string.length(); j++)
            this->table[i][j] = new int [string.length()]();
    }
```

```
21      end = clock();
22      std::cout << "  " << "Time elapsed init (s) : " << double(end - begin) /
            CLOCKS_PER_SEC << "\n";
23
24      parser_result parser_result;
25
26      begin = clock();
27      parser_result.result = this->parse(0, 0, string.size());
28      end = clock();
29      std::cout << "  " << "Time elapsed trt (s) : " << double(end - begin) /
            CLOCKS_PER_SEC << "\n";
30
31      parser_result.counter = this->counter;
32      return parser_result;
33  }
34
35  bool LinearParser::parse(int var, int i, int j) {
36      this->counter++;
37
38      if (this->table[var][i][j - 1] == 0)
39          return (this->table[var][i][j - 1] == 1);
40
41      for (int right_hand(0); right_hand < this->grammar.rules[var].size();
            right_hand++) {
42          if (this->grammar.rules[var][right_hand].size() == j - i && this->
                string.size() - i >= j - i) {
43              bool is_equal(true);
44              for (int k(0); k < this->grammar.rules[var][right_hand].size(); k
                    ++) {
45                  if (this->string[i + k] != this->grammar.rules[var][
                        right_hand][k]) {
46                      is_equal = false;
47                      break;
48                  }
49              }
50              if (is_equal) {
51                  this->table[var][i][j - 1] = 1;
52                  return true;
53              }
54          }
55          char var_1, var_2, var_3;
56          if (this->grammar.rules[var][right_hand].size() == 1) {
57              var_1 = this->grammar.rules[var][right_hand][0];
58              if (var_1 >= 'A' && var_1 <= 'Z') {
59                  if (this->parse(this->grammar.non_terminals.find(var_1), i, j
                        )) {
60                      this->table[var][i][j - 1] = 1;
61                      return true;
62                  }
63              }
64          } else if (j - i >= 2 && this->grammar.rules[var][right_hand].size()
                == 2) {
65              var_1 = this->grammar.rules[var][right_hand][0];
66              var_2 = this->grammar.rules[var][right_hand][1];
67              if (var_1 >= 'A' && var_1 <= 'Z' && var_2 >= 'A' && var_2 <= 'Z')
                    {
68                  for (int k(i + 1); k < j; k++) {
69                      if (this->parse(this->grammar.non_terminals.find(var_1),
                            i, k) &&
70                          this->parse(this->grammar.non_terminals.find(var_2),
                                k, j))
71                      {
72                          this->table[var][i][j - 1] = 1;
73                          return true;
74                      }
75                  }
76              } else if ((var_1 < 'A' || var_1 > 'Z') && var_2 >= 'A' && var_2
                    <= 'Z') {
```

```
77                  if (this->string[i] == var_1) {
78                      if (this->parse(this->grammar.non_terminals.find(var_2),
                            i + 1, j)) {
79                          this->table[var][i][j - 1] = 1;
80                          return true;
81                      }
82                  }
83              } else if (var_1 >= 'A' && var_1 <= 'Z' && (var_2 < 'A' || var_2
                    > 'Z')) {
84                  if (this->string[j - 1] == var_2) {
85                      if (this->parse(this->grammar.non_terminals.find(var_1),
                            i, j - 1)) {
86                          this->table[var][i][j - 1] = 1;
87                          return true;
88                      }
89                  }
90              }
91          } else if (j - i >= 3 && this->grammar.rules[var][right_hand].size()
                == 3) {
92              var_1 = this->grammar.rules[var][right_hand][0];
93              var_2 = this->grammar.rules[var][right_hand][1];
94              var_3 = this->grammar.rules[var][right_hand][2];
95              if (var_1 >= 'A' && var_1 <= 'Z' && (var_2 < 'A' || var_2 > 'Z')
                    && var_3 >= 'A' && var_3 <= 'Z') {
96                  for (int k(i + 1); k < j - 1; k++) {
97                      if (this->parse(this->grammar.non_terminals.find(var_1),
                            i, k) &&
98                          var_2 == this->string[k] &&
99                          this->parse(this->grammar.non_terminals.find(var_3),
                                k + 1, j))
100                     {
101                         this->table[var][i][j - 1] = 1;
102                         return true;
103                     }
104                 }
105             } else if ((var_1 < 'A' || var_1 > 'Z') && var_2 >= 'A' && var_2
                    <= 'Z' && (var_3 < 'A' || var_3 > 'Z')) {
106                 if (var_1 == this->string[i] &&
107                     this->parse(this->grammar.non_terminals.find(var_2), i +
                            1, j - 1) &&
108                     var_3 == this->string[j - 1])
109                 {
110                     this->table[var][i][j - 1] = 1;
111                     return true;
112                 }
113             }
114         }
115     }
116
117     this->table[var][i][j - 1] = 2;
118     return false;
119 }
```