Umeå University
Efficient Algorithms

Assignment Step 2

Runtime analysis of the Java implementation of the CYK-algorithm

Pina Kolling

September 26, 2022

Contents

1 Introduction

Parsing in Computer Science is the process of analysing a string of characters to examine if the string is built according to the rules of a formal grammar.
A formal grammar describes how to form strings with correct syntax from a language's alphabet (section 2.2 and 2.1). To examine if such a string follows the rules of a grammar the *Cocke-Younger-Kasami*-algorithm (short: *CYK*) can be used. This algorithm is described in section 2.4. To use the *CYK*-algorithm the grammar needs to be in a specific format, that is called *Chomsky-Normal-Form* (*CNF*), which is explained in section 2.3. [4, 6]
The task for this assignment was to code three different parsing methods to execute the *CYK*-algorithm in *Java*. The different parsing methods will be described and presented as pseudo code in section 3. For the implementation three different classes were implemented: `main.java, grammar.java` and `parser.java`. The `main`-class calls the methods and the `grammar`-class parses the input grammer ans string into a format that then can be processed in the `parser`-class. The function and implementation will be former described in section 3.

2 Background

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

2.1 Formal language

Formal languages are abstract languages which define the syntax of the words that get accepted by that language. It consits of a set of words that get accepted by the language and a set of symbols that is called alphabet and contains the characters of the words. Those characters are calles nonterminal symbols. [1, 5]

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

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

The language definition L *is the following one:*

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

2.2 Formal grammar

A formal grammar describes how to form strings with correct syntax from a language's alphabet. A grammar does not describe the meaning of the strings or any semantics — only their syntax is defined. The grammar is a set of rules which define which words are accepted by a formal language. Those rules consist of terminal and nonterminal symbols. The terminalsymbols are the characters of the alphabet of the language and the nonterminalsymbols are used to build the rules of the language – they get replaced by terminalsymbols. [1, 5]

Example (Formal grammar)**.** *The example grammar for the previous example language is the following:*

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

2.3 Chomsky-Normal-Form

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

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

Example (Chomsky-Normal-Form). *To change the previous example into CNF the rules have to be split up:*

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

2.4 CYK-algorithm

Cocke-Younger-Kasami-algorithm (short: *CYK*) takes a grammar in *CNF* and a word as an input. It then examines if the word follows the rules of the grammar. [6]

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

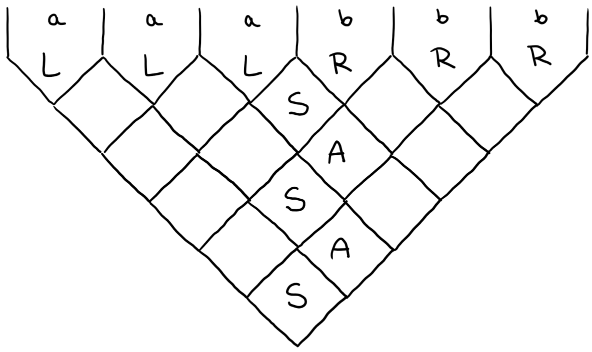


Figure 1: CYK-algorithm table for the word *aaabbb*

The word is accepted by the grammar rules, because the intiating nonterminal symbol S can be filled into the lowest field.

2.5 Dynamic programming

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

3 System Design

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

3.1 Main

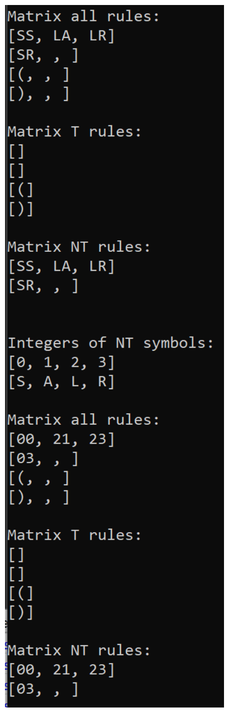
The `main`-class takes the input grammar and word and parses them into `String[]` and `String`. The arguments have to follow the following rules:

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

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

3.2 Grammar

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



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

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

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


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

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

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

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

3.3 Parser

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

3.3.1 Naive

The naive algorithm is a recursive algorithm and has an initial method to start the recursion with the start values. Then the recursive method gets called.

Algorithm 1 Recursion call: Boolean parseNaive()

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

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

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

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

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

Calculating the runtime:

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

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

3.3.2 Top-Down

The top-down method is an improves version of the naive method (section 3.3.1). This algorithm works recursive, too. In this algorithm another array is used, which contains the values `true`, `false` and `null`. If one of the values is not `null`, the next recursive call is not executed.

Algorithm 3 Recursion call: Boolean parseTD()

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

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

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

Calculating the runtime of the method call:

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

Algorithm 4 Boolean parseTD(int indexNT, int i, int j)

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

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

3.3.3 Bottom-Up

4 Evaluation

4.1 Runtimes

4.2 Experiments

5 Conclusion and Future Work

From our experiments we can conclude that ...

A How to use the code?

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

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

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

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

The first array contains all rules.

The second array contains only the terminal rules.

The third array contains only the nonterminal rules.

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

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

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

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

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


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

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

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

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

```
Input word: (())

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

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

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

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

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

References

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

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

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

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

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

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