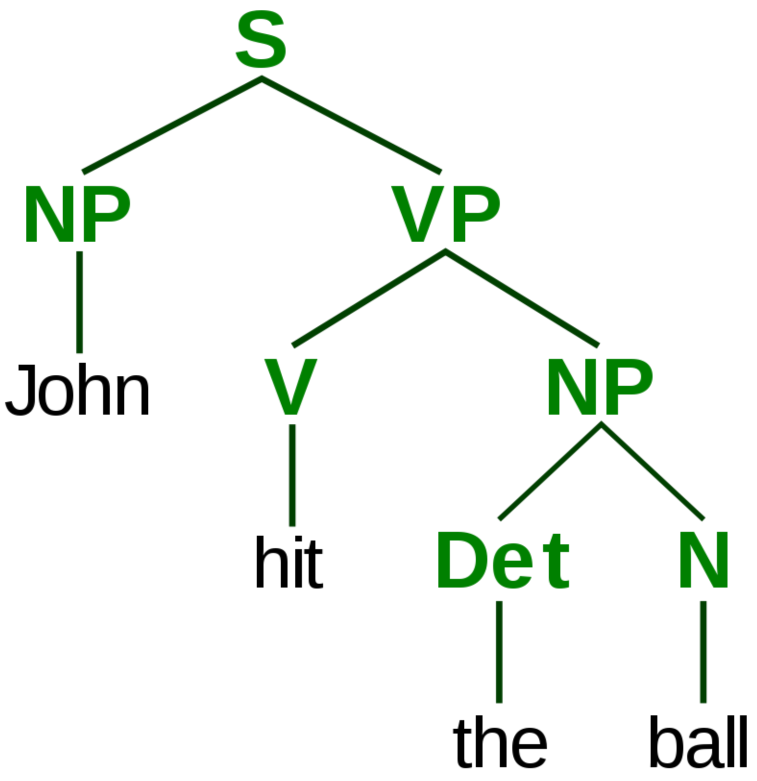
Cocke
Younger
Kasami

CYK Algorithm for CFL Parsing

Nagasai
Rohit Jorige
Sacchit Kale
Sahil Chaudhary

CYK ALGORITHM
FOR CFL PARSING

Key Features:
> Bottom-up parsing strategy
> Employs dynamic programming for efficiency



Setting

- Suppose we are trying to make a compiler for a programming language. One of the tasks of compiler is to make sure that the given code is syntactically correct.

Setting

- Suppose we are trying to make a compiler for a programming language. One of the tasks of compiler is to make sure that the given code is syntactically correct.
- How do compilers achieve that?
- How to formulate this problem mathematically?

Setting

- Let's take an example. Suppose your programming language accepts only valid arithmetic expressions of numbers in base 2.
- For simplicity, let's assume that the only arithmetic operation allowed is Addition.

So, the program accepts strings like

$$((10)+(1+1))$$
$$(((10))+(((101))))$$

But doesn't accept strings like

$$((10+101)$$
$$(01+(10+01))$$

Setting

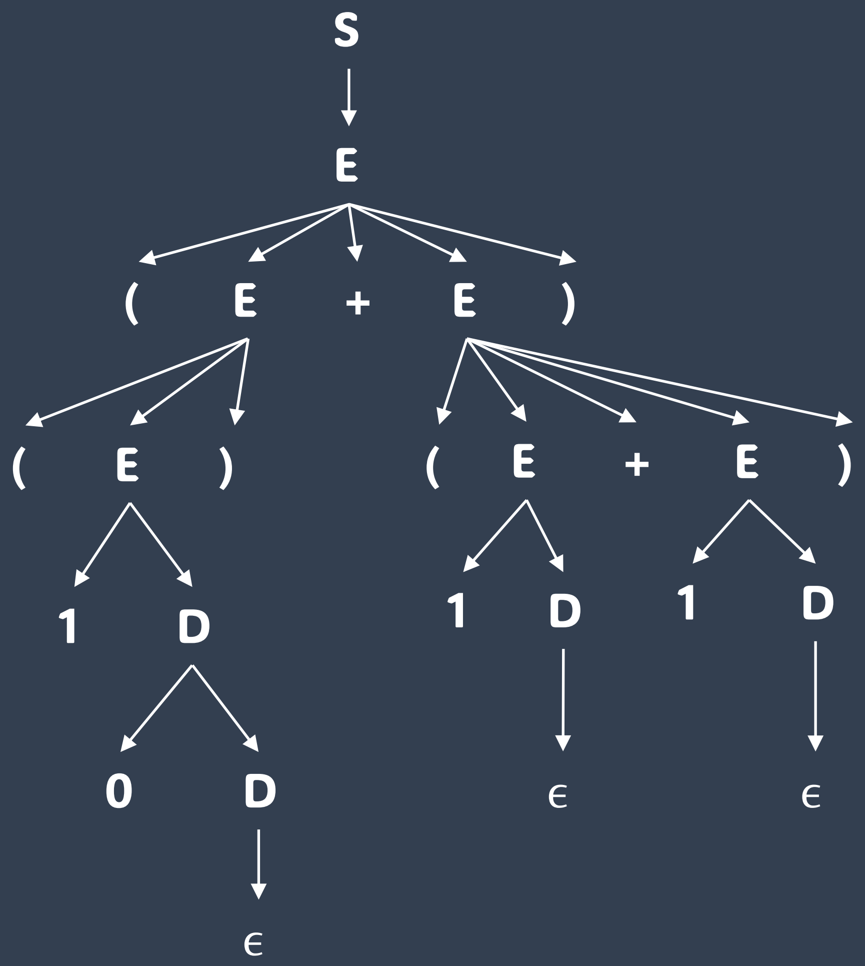
- Consider the following Context-Free Grammar
  - $S \rightarrow E$
  - $E \rightarrow (E) \mid (E + E) \mid 1D \mid 0$
  - $D \rightarrow \varepsilon \mid 1D \mid 0D$
- This Grammar generates all the valid expressions as discussed before.

Setting

- Consider the following Context-Free Grammar
  - **S $\rightarrow$E**
  - **E $\rightarrow$(E) | (E + E) | 1D | 0**
  - **D $\rightarrow$ε | 1D | 0D**
- This Grammar generates all the valid expressions as discussed before.
- We can visualize the strings generated by the grammar using parse trees. They encode all the information needed to derive the string from the starting non-terminal.

Setting

- Example of a parse tree: Consider the base 2 arithmetic expression shown before. $((10) + (1 + 1))$
- The corresponding parse tree for the string is:



- $S \rightarrow E$
- $E \rightarrow (E) \mid (E + E) \mid 1D \mid 0$
- $D \rightarrow \varepsilon \mid 1D \mid 0D$

Setting

- Then, our original problem boils down to checking whether a given string (our code) is a part of language of that Context Free Grammar.
- This is also called **Parsing Problem.**

Setting

- Can we come up with an algorithm which determines whether a given string is in language of a Context-Free Grammar?

Problem:

Given a string ‘w’ and Context Free Grammer(G), can we determine **if**

$$w \in L(G)?$$

Fundamental Idea that comes to mind from looking at this problem is,

Divide and Conquer

Given a Context Free Grammer, there can multiple sub-problem for a given problem.

$$\underbrace{w_1 w_2 w_3 \ldots w_i}_{A}\quad \underbrace{w_{i+1} w_{i+2} \ldots w_n}_{B}$$

$$\underbrace{w_1 w_2 \ldots w_j}_{A}\quad \underbrace{w_{j+1} \ldots w_i}_{B}\quad \underbrace{w_{i+1} w_{i+2} \ldots w_n}_{C}$$

$$\underbrace{w_1 w_2 \ldots w_j}_{A}\quad \underbrace{w_{j+1} \ldots w_k}_{B}\quad \underbrace{w_{k+1} \ldots w_i}_{C}\quad \underbrace{w_{i+1} w_{i+2} \ldots w_n}_{D}$$

Problem of reducing our parsing problem into sub-problem and solving it later for CFG is very **time consuming!**

Every string in CNF is produced from a basic rule

$S \rightarrow AB$

$$\begin{tikzcd} & S \arrow[dl] \arrow[dr] & \\ A & & B \end{tikzcd}$$

&

$A \rightarrow a$

$$w = w_1 w_2 w_3 \dots w_n$$

Intuitively, It makes more sense to use CNF to determine if w belongs to language of the grammar

Let's Try Brute Force !

For a String of Length n ,Let's go through Every Possible Parse Tree Whose yield is a String of length n.

But How Many Such Parse Trees?
In General Could be Infinitely many, But if we consider our Grammar to be in Chomsky normal form.
We could give an Exact Number.

But How Many Such Parse Trees?

In General Could be Infinitely many, But if we consider our Grammar to be in Chomsky normal form.

We could give an Exact Number.

We Know that The Parse tree for CNF Grammar is a Binary Tree.

Let P(n) be the number of Parse Trees which generate a Sequence a Length n.

The Root of the Binary tree has Two Subtrees which generate the String.

Let's Say One tree Generates K words and the other generates remaining n-k words .

Then the number of Parse Trees P(n) is Given By:

$$P(n) = \sum_{k=1}^{n-1} P(n-k)P(k)$$

$$P(n) = \frac{\binom{2n}{n}}{(n+1)!}$$

Which is an exponential time algorithm!
Implying very large number of computations.

Brute Force is bad even in CNF. But can we utilize the property of CNF to reduce the time complexity somehow?

Every string in CNF is produced from a basic rule

$$S \rightarrow AB$$

$$\begin{tikzcd} & S \arrow[dl] \arrow[dr] & \\ A & & B \end{tikzcd}$$

&

$$A \rightarrow a$$

$$w = w_1 w_2 w_3 \ldots w_n$$

Intuitively, makes sense to **split w** such that we know first part is produced from one rule and the second from the other

Can we split w such that we know A yields the first half and B yields the other?

$$w = \underbrace{w_1 w_2 w_3 \dots w_i}_{A} \;\Big|\; \underbrace{w_{i+1} w_{i+2} \dots w_n}_{B}$$

$$w = \underbrace{w_1 w_2 w_3 \dots w_i}_{A} \;\Big|\; \underbrace{w_{i+1} w_{i+2} \dots w_n}_{B}$$

$$\underbrace{w_1 w_2 w_3 \dots w_{i_1}}_{A_1} \;\Big|\; \underbrace{w_{i_1+1} \dots w_i}_{B_1} \qquad \underbrace{w_{i+1} w_{i+2} \dots w_{i_2}}_{A_1} \;\Big|\; \underbrace{w_{i_2+1} \dots w_n}_{B_1}$$

And Keep on Splitting!

Until we reach a point, where

$$w = w_1 \mid w_2 \mid w_3 \mid \ldots \mid w_n$$

$$\downarrow \quad \downarrow \quad \downarrow \qquad \downarrow$$

$$A_1 \quad A_2 \quad A_3 \qquad A_n$$

If $A_i's$ **are non terminal symbols which produce terminals wi's, we are done!**

Else, w doesn't belong to language of the grammar

The idea seems correct but how to identify the position of

Split ?

Does there exist a Substructure to the problem ?

Since the Grammar is in CNF form, Then We Can Say That
The Grammar Generates W_{1n} if and only if there exists a pair of non−terminals A_1, A_2 such that A_1 Produces W_{1i} and A_2 produces $W_{(i+1)n}$ for some k=1 to n−1 such that there exist a production Rule X gives $X \to A_1 A_2$ for some Non−terminal X.

Base Case for this problem is for n=1 , grammar generates w if and only if there exists a Non−Terminal X which gives w.

Let's try thinking Bottom Up!

INPUT AND PROCESSING

Prerequisite

The CFG must be in Chomsky Normal Form (CNF) to use CYK. CNF Rules:

A ⟶ BC	(Two non-terminals on the right-hand side)
A ⟶ a	(A single terminal on the right-hand side)

CYK Table

> A $n \times n$ triangular matrix (where n is string length)

> Each cell holds non-terminals that could generate the corresponding substring of the input string.

> Fill bottom-up using dynamic programming.

KEY IDEA – DYNAMIC PROGRAMMING

- WE BREAK DOWN THE PROBLEM TO SUBPROBLEMS AND REMEMBER TO STORE THE RESULT OF EACH PROBLEM. WE THEN CONSTRUCT A BIGGER PROBLEM USING THE SOLUTIONS OF THE PREVIOUSLY STORED AND SOLVED SUBPROBLEMS.
- THIS AVOIDS REPEATED RECURSION AND REDUCES THE TIME COMPLEXITY.
- BY MAINTAINING BACK POINTER WE CAN BACKTRACK TO CHECK WHETHER THE STRING IS IN THE LANGUAGE OR NOT.

Algorithm 1 CYK Algorithm

Require: Grammar G, string w

```
n ← length of w
matrix[n][n]
for i = 1 to n do
    if w[i] is in RHS of some production rule then
        matrix[i][i].append(Non-terminal that produces w[i])
    end if
end for
for k = 1 to n do
    for i = 1 to n − k + 1 do
        j ← i + k
        for b = i to j do
            for x · y in cross(matrix[i][b], matrix[b + 1][j]) do
                if A produces x · y then
                    matrix[i][j].append(A)
                end if
            end for
        end for
    end for
end for
if S in matrix[1][n] then
    return True
else
    return False
end if
```

- i represents row number
- j represents column number
- k represents diagonal number
- b varies from i to $j - 1$

EXAMPLE 1

$$W = (()())$$

$$S \to LX|SS|LR$$
$$X \to SR$$
$$L \to ($$
$$R \to)$$

We break it down into subproblems

	1	2	3	4	5	6
1	(	((	(()	(()(	(()()	(()())
2		(	()	()(	()()	()())
3			)	)(	)()	)())
4				(	()	())
5					)	))
6						)

EXAMPLE 1

$$W = (()())$$

$$S \to LX|SS|LR$$
$$X \to SR$$
$$L \to ($$
$$R \to)$$

We break it down into subproblems

	1	2	3	4	5	6
1	(	((	(()	(()(	(()()	(()())
2		(	()	()(	()()	()())
3			)	)(	)()	)())
4				(	()	())
5					)	))
6						)

EXAMPLE 1

$$W = (( ) ( ))$$

$$S \rightarrow LX|SS|LR$$
$$X \rightarrow SR$$
$$L \rightarrow ($$
$$R \rightarrow)$$

THIS REPRESENTS THE BASE CASE

	1	2	3	4	5	6
1	L	((	(()	(()(	(()()	(()())
2		L	()	()(	()()	()())
3			R	)(	)()	)())
4				L	()	())
5					R	))
6						R

EXAMPLE 1

$$W = (\,(\,)\,(\,)\,)$$

$$S \to LX|SS|LR$$
$$X \to SR$$
$$L \to ($$
$$R \to)$$

WE NOW CONSTRUCT BIGGER CASES STRINGS USING ALREADY CONSTRUCTED SMALLER STRINGS

$$((= (\cdot (= L \cdot L$$
$$(\,) = (\cdot) = L \cdot R = S$$

	1	2	3	4	5	6
1	L	((	(()	(()(	(()()	(()())
2		L	()	()(	()()	()())
3			R	)(	)()	)())
4				L	()	())
5					R	))
6						R

EXAMPLE 1

$$W = (()())$$

$$S \to LX|SS|LR$$
$$X \to SR$$
$$L \to ($$
$$R \to)$$

similarly we define for the remaining symbols for the 2nd diagonal

	1	2	3	4	5	6
1	L	Ø	(()	(()(	(()()	(()())
2		L	S	()(	()()	()())
3			R	Ø	)()	)())
4				L	S	())
5					R	Ø
6						R

EXAMPLE 1

$$W = (( ) ( ))$$

$$S \rightarrow LX|SS|LR$$
$$X \rightarrow SR$$
$$L \rightarrow ($$
$$R \rightarrow)$$

$$(() = (\cdot () + ((\cdot)$$
$$= L \cdot S \cup \emptyset \cdot R$$
$$= \emptyset$$

(since $L \cdot S$ is not in production rules)

$$()(= (\cdot)(+ () \cdot ($$
$$= L \cdot \emptyset \cup S \cdot L$$
$$= \emptyset \cup S \cdot L$$
$$= S \cdot L$$
$$= \emptyset$$

(SINCE $S \cdot L$ IS NOT THERE IN PRODUCTION RULES

	1	2	3	4	5	6
1	L	$\emptyset$	(()	(()(	(()()	(()())
2		L	S	()(	()()	()())
3			R	$\emptyset$	)()	)())
4				L	S	())
5					R	$\emptyset$
6						R

EXAMPLE 1

$W = (\,(\,)\,(\,)\,)$

$$S \rightarrow LX|SS|LR$$
$$X \rightarrow SR$$
$$L \rightarrow ($$
$$R \rightarrow)$$

We break it down into subproblems

	1	2	3	4	5	6
1	L	Ø	Ø	(()(	(()()	(()())
2		L	S	Ø	()()	()())
3			R	Ø	Ø	)())
4				L	S	X
5					R	Ø
6						R

EXAMPLE 1

$$W = (()())$$

$$S \rightarrow LX|SS|LR$$
$$X \rightarrow SR$$
$$L \rightarrow ($$
$$R \rightarrow)$$

We break it down into subproblems

	1	2	3	4	5	6
1	L	∅	∅	(()(	(()()	(()())
2		L	S	∅	()()	()())
3			R	∅	∅	)())
4				L	S	X
5					R	∅
6						R

EXAMPLE 1

$$W = (\,(\,)\,(\,)\,)$$

$$S \rightarrow LX|SS|LR$$
$$X \rightarrow SR$$
$$L \rightarrow ($$
$$R \rightarrow)$$

We continue to break down the strings and see if they can be represented in terms of already solved subproblems in the production rules.

	1	2	3	4	5	6
1	L	Ø	Ø	Ø	(()()	(()())
2		L	S	Ø	S	()())
3			R	Ø	Ø	Ø
4				L	S	X
5					R	Ø
6						R

EXAMPLE 1

$$W = (\,(\,)\,(\,)\,)$$

$$S \rightarrow LX|SS|LR$$
$$X \rightarrow SR$$
$$L \rightarrow ($$
$$R \rightarrow)$$

We continue to break down the strings and see if they can be represented in terms of already solved subproblems in the production rules.

	1	2	3	4	5	6
1	L	Ø	Ø	Ø	Ø	(()())
2		L	S	Ø	S	X
3			R	Ø	Ø	Ø
4				L	S	X
5					R	Ø
6						R

EXAMPLE 1

$$W = (\,(\,)\,(\,)\,)$$

$$S \to LX|SS|LR$$
$$X \to SR$$
$$L \to ($$
$$R \to)$$

Notice that we have completely filled the table .
We have that the top-most corner has S .
Therefore by the algorithm we can conclude that the given sequence will be accepted by the CFG.

	1	2	3	4	5	6
1	L	Ø	Ø	Ø	Ø	S
2		L	S	Ø	S	X
3			R	Ø	Ø	Ø
4				L	S	X
5					R	Ø
6						R

EXAMPLE 1

$$W = (\,(\,)\,(\,)\,)$$

$$S \rightarrow LX|SS|LR$$
$$X \rightarrow SR$$
$$L \rightarrow ($$
$$R \rightarrow)$$

BUT CAN WE WRITE A PARSE TREE FROM THE TABLE WE HAVE CONSTRUCTED?

	1	2	3	4	5	6
1	L	∅	∅	∅	∅	S
2		L	S	∅	S	X
3			R	∅	∅	∅
4				L	S	X
5					R	∅
6						R

EXAMPLE 1

$$W = (\,(\,)\,(\,)\,)$$

$$S \rightarrow LX|SS|LR$$
$$X \rightarrow SR$$
$$L \rightarrow ($$
$$R \rightarrow)$$

YES, THROUGH BACK POINTERS

	1	2	3	4	5	6
1	L					S
2		L	S		S	X
3			R			
4				L	S	
5					R	
6						R


(( ) ( ))
You shall Parse!

EXAMPLE 2

Let us consider the CFG of Valid Parenthesis:

$$W = (()()$$

$$S \rightarrow LX|SS|LR$$
$$X \rightarrow SR$$
$$L \rightarrow ($$
$$R \rightarrow)$$

We break it down into subproblems and fill the table.

(	((	(()	(()(	(()()
	(	()	()(	()()
		)	)(	)()
			(	()
				)

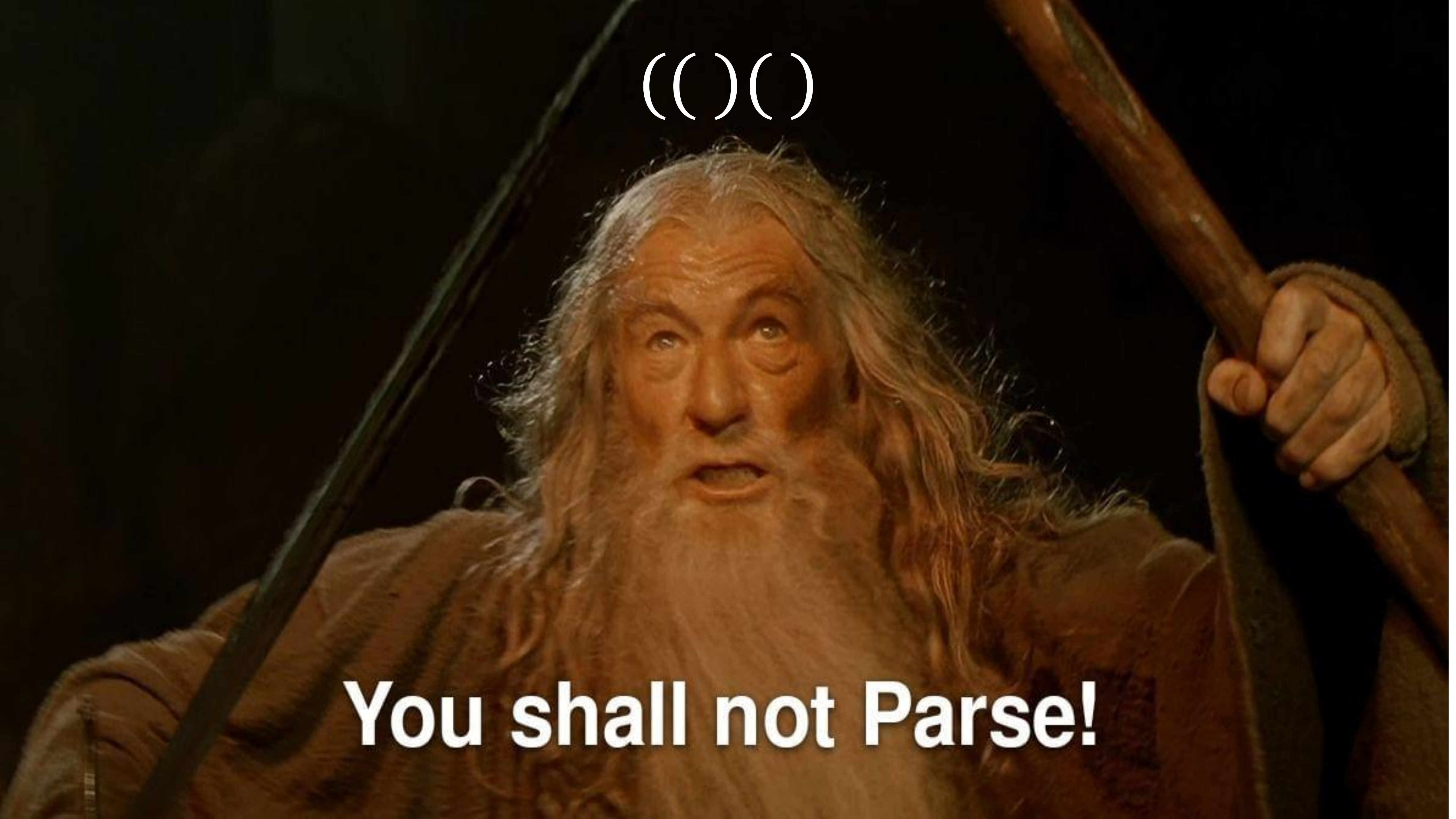
EXAMPLE 2

$$W = (()()$$

$$S \rightarrow LX|SS|LR$$
$$X \rightarrow SR$$
$$L \rightarrow ($$
$$R \rightarrow)$$

Notice that we have completely filled the table .
The top most corner does not have S .
Given sequence will NOT be accepted by the CFG.

	1	2	3	4	5
1	L	Ø	Ø	Ø	Ø
2		L	S	Ø	S
3			R	Ø	Ø
4				L	S
5					R


(()()
You shall not Parse!

Proof of correctness

Let’s take a closer look at the entries of the table for $w = w_1 w_2 w_3 \ldots w_n$

$X_{1,1}$	$X_{1,2}$	…	…	…	…	…	…	…	$X_{1,n}$
	$X_{2,2}$	$X_{2,3}$	…	…	…	…	…	…	$X_{2,n}$
		$X_{3,3}$	$X_{3,4}$	…	…	…	…	…	$X_{3,n}$
			$X_{4,4}$	…	…	…	…	…	$X_{4,n}$
				$X_{5,5}$	…	…	…	…	$X_{5,n}$
					$X_{6,6}$	…	…	…	$X_{6,n}$
									$X_{n,n}$

What does $X_{i,j}$ represent in our table?

Claim: Table entry $X_{i,j}$ is the set of variable A such that

$$\boldsymbol{A \overset{*}{\Rightarrow} w_i w_{i+1} \dots \mathrm{w_j}}$$

What does $X_{i,j}$ represent in our table?

Claim: Table entry $X_{i,j}$ is the set of variable A such that

$$\boldsymbol{A \overset{*}{\Rightarrow} w_i w_{i+1} \ldots \mathrm{w_j}}$$

Algorithm returns 1 when $S \in X_{1n}$

So, If we can prove our claim, its equivalent to saying

$$\mathbf{S} \overset{*}{\Rightarrow} \boldsymbol{w_1 w_2 \ldots \mathrm{w_n}}$$

i.e.

$$\boldsymbol{w_1 w_2 \ldots \mathrm{w_n} \in L(G)}$$

Idea to Prove

Induct on **k=(j-i)** which represents diagonal element entries of matrix row has the form $X_{i,j}$
where, $X_{i,j}$ is the set of variable A such that

$$A \overset{*}{\Rightarrow} w_i w_{i+1} \ldots w_j$$

Base Case: k=0

The entries of first diagonal is only of the form $X_{i,i}$

String beginning at i and ending at i is just w_i

According to our algorithm, we fill the cells, if there is a production of the form $\boldsymbol{A \rightarrow w_i}$

Thus, table entries at first diagonal of the form $X_{i,i}$ has entries of production of the form

$$\boldsymbol{A \rightarrow w_i}$$

$$\boldsymbol{A \Rightarrow w_i}$$

Assume the claim holds for all the Kth diagonal. $(0 \leq k < n\text{-}1)$
i.e.
Information about all strings of length shorter than k in w is known!

According to our algorithm, we add variable A to set $\boldsymbol{X_{ij}}$, if we can find variables B and C and integer k such that:

1. $i \leq k < j$
2. $B \in X_{i,k}$
3. $C \in X_{k+1,j}$
4. $A \rightarrow BC$ is a production of G

According to our algorithm, we add variable A to set $\boldsymbol{X_{ij}}$, if we can find variables B and C and integer k such that:

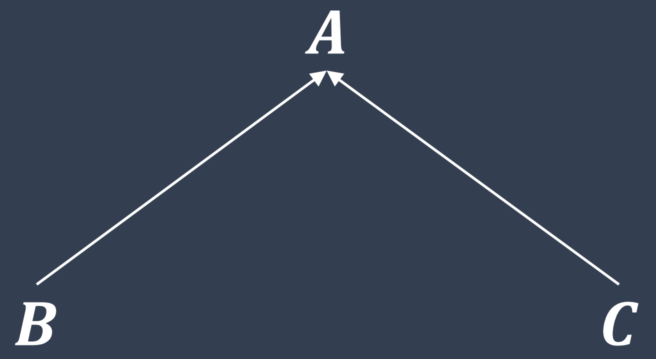
1. $i \leq k < j$
2. $B \in X_{i,k}$
3. $C \in X_{k+1,j}$
4. $A \rightarrow BC$ is a production of G

Thus, we compare exactly j-i pairs of previously computed sets!

$$(X_{i,i},\ X_{i+1,j}),(X_{i,i+1},\ X_{i+2,j}) \ldots (X_{i,j-1},\ X_{j,j})$$

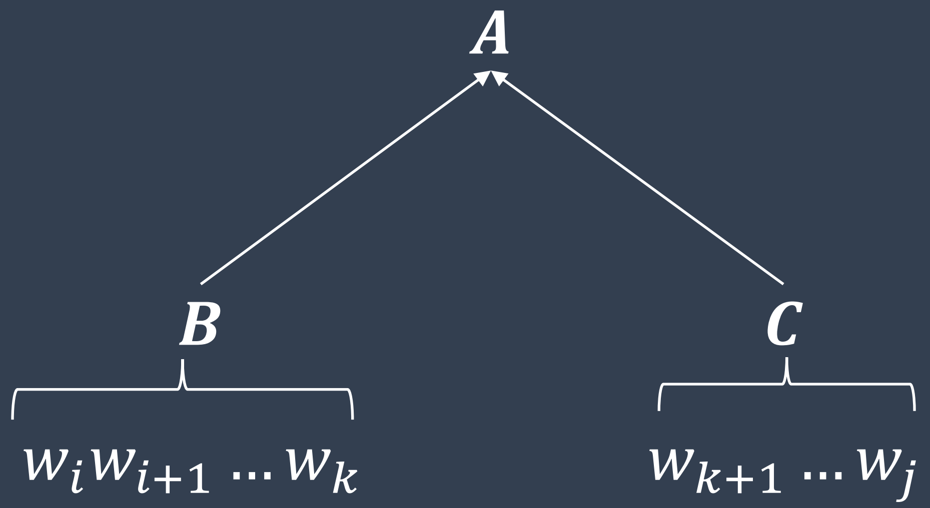
Inductive Step: $\boldsymbol{k = K + 1 \leq n}$

$\{\boldsymbol{X}_{i,j}\}: j - i + 1 = k + 1$



Inductive Step: $\boldsymbol{k = K + 1 \leq n}$

$\{\boldsymbol{X_{i,j}}\}: j - i + 1 = k + 1$



$$\boldsymbol{B} \overset{*}{\Rightarrow} w_i w_{i+1} \dots w_k$$

$$C \overset{*}{\Rightarrow} w_{k+1} w_{k+2} \dots w_j$$

We add A if $A \rightarrow BC$ *is a production*!

$$A \rightarrow BC \Rightarrow w_i w_{i+1} \dots w_k w_{k+1} w_{k+2} \dots w_j$$

$$A \Rightarrow w_i w_{i+1} \dots w_k w_{k+1} w_{k+2} \dots w_j$$

$$A \overset{*}{\Rightarrow} w_i w_{i+1} \dots \dots w_j$$

Thus, Our claim holds true by principle of Induction hypothesis

By our previous claim,

Table entry $X_{i,j}$ is the set of variable A such that $\boldsymbol{A \overset{*}{\Rightarrow} w_i w_{i+1} \ldots w_j}$

Thus, $X_{1,n}$ is the set of variable A such that $\boldsymbol{A \overset{*}{\Rightarrow} w_1 w_2 \ldots w_n}$

If $S \in X_{1,n}$, we are done because saying w belongs to language the grammar is equivalent to saying,

$$\mathbf{S} \overset{*}{\Rightarrow} \boldsymbol{w_1 w_2 \ldots w_n}$$

If $S \notin X_{1,n}$, S doesn’t belong to set of symbols that can derive w.
Thus, Our algorithm correctly identifies if a string $w \in L(G)$.

Time Complexity Analysis of CYK

Time Complexity Analysis of CYK

Note that there are $O(n^2)$ entries to compute and each involves comparing and computing n pairs of entries.
Remember, although there can be many variables in each set X_{ij}, the grammar G is fixed and the number of variables only depend on number of production.

Thus, the time to compare two entries $X_{i,k}$ and $\mathrm{X}_{\mathrm{k+1,j}}$ and find variables that go into X_{ij} is O(1).
As there are at most n such pairs for each X_{ij}, thus the total work is $O(n^3)$.

Thank You!