

# CYK Algorithm for Parsing General Context-Free Grammars

# Why Parse General Grammars

- Can be difficult or impossible to make grammar unambiguous
  - thus LL(k) and LR(k) methods cannot work, for such ambiguous grammars
- Some inputs are more complex than simple programming languages
  - mathematical formulas:  
 $x = y \wedge z$       ?       $(x=y) \wedge z$        $x = (y \wedge z)$
  - natural language:  
*I saw the man with the telescope.*
  - future programming languages

# Ambiguity

1)



2)



*I saw the man with the telescope.*

# CYK Parsing Algorithm

C:

[John Cocke](#) and Jacob T. Schwartz (1970). Programming languages and their compilers: Preliminary notes. Technical report, [Courant Institute of Mathematical Sciences](#), [New York University](#).

Y:

Daniel H. **Younger** (1967). Recognition and parsing of context-free languages in time  $n^3$ . *Information and Control* 10(2): 189–208.

K:

[T. Kasami](#) (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, [Bedford, MA](#).

# Two Steps in the Algorithm

1) Transform grammar to normal form  
called Chomsky Normal Form

(Noam Chomsky, mathematical linguist)

2) Parse input using transformed grammar  
dynamic programming algorithm

“a method for solving complex problems by breaking them down into simpler steps.

It is applicable to problems exhibiting the properties of overlapping subproblems” (>WP)

# Balanced Parentheses Grammar

Original grammar G

$$S \rightarrow "" \mid (S) \mid SS$$

Modified grammar in Chomsky Normal Form:

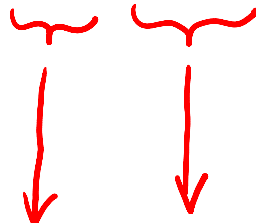
$$S \rightarrow "" \mid S' \quad \leftarrow \text{if } "" \in L(G)$$

$$\begin{array}{l}
 S' \rightarrow N_{(} N_{S)} \mid N_{(} N_{)} \mid S' S' \\
 N_{S)} \rightarrow S' N_{)} \\
 N_{(} \rightarrow ( \\
 N_{)} \rightarrow )
 \end{array}
 \left. \begin{array}{l}
 \text{Rules} \\
 \text{Rules}
 \end{array} \right\}
 \begin{array}{l}
 N \rightarrow N_1 N_2 \\
 \text{nonterminals} \\
 \\
 N \rightarrow t \\
 \begin{array}{l}
 \uparrow \text{nonterminal} \quad \nwarrow \text{terminal}
 \end{array}
 \end{array}$$

- Terminals: ( ) Nonterminals: S S' N<sub>S)</sub> N<sub>)</sub> N<sub>(</sub>  
*nonterminal with funny name*

# Idea How We Obtained the Grammar

$$S \rightarrow ( S )$$



$$S' \rightarrow N_{(} N_{)} \mid N_{(} N_{)}$$

because  $S$  can be empty  
but  $S'$  cannot

$$N_{(} \rightarrow ($$

$$N_{)} \rightarrow S' N_{)}$$

$$N_{)} \rightarrow )$$

Chomsky Normal Form transformation  
can be done fully mechanically

# Dynamic Programming to Parse Input

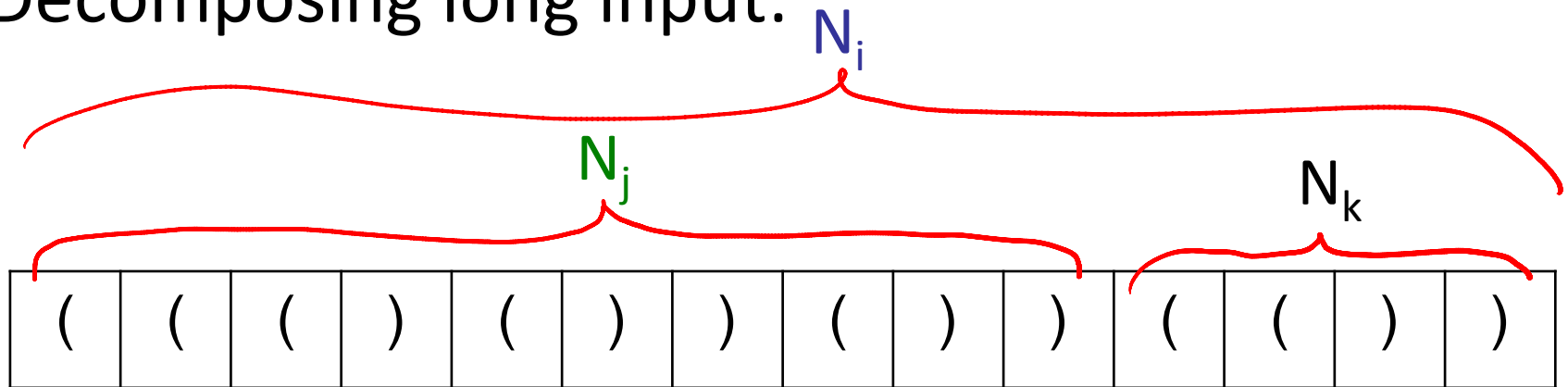
Assume Chomsky Normal Form, 3 types of rules:

$S \rightarrow "" \mid S'$  (only for the start non-terminal)

$N_j \rightarrow t$  (names for terminals)

$N_i \rightarrow N_j N_k$  (just **2** non-terminals on RHS)

Decomposing long input:



find all ways to parse substrings of length 1,2,3,...



# Parsing an Input

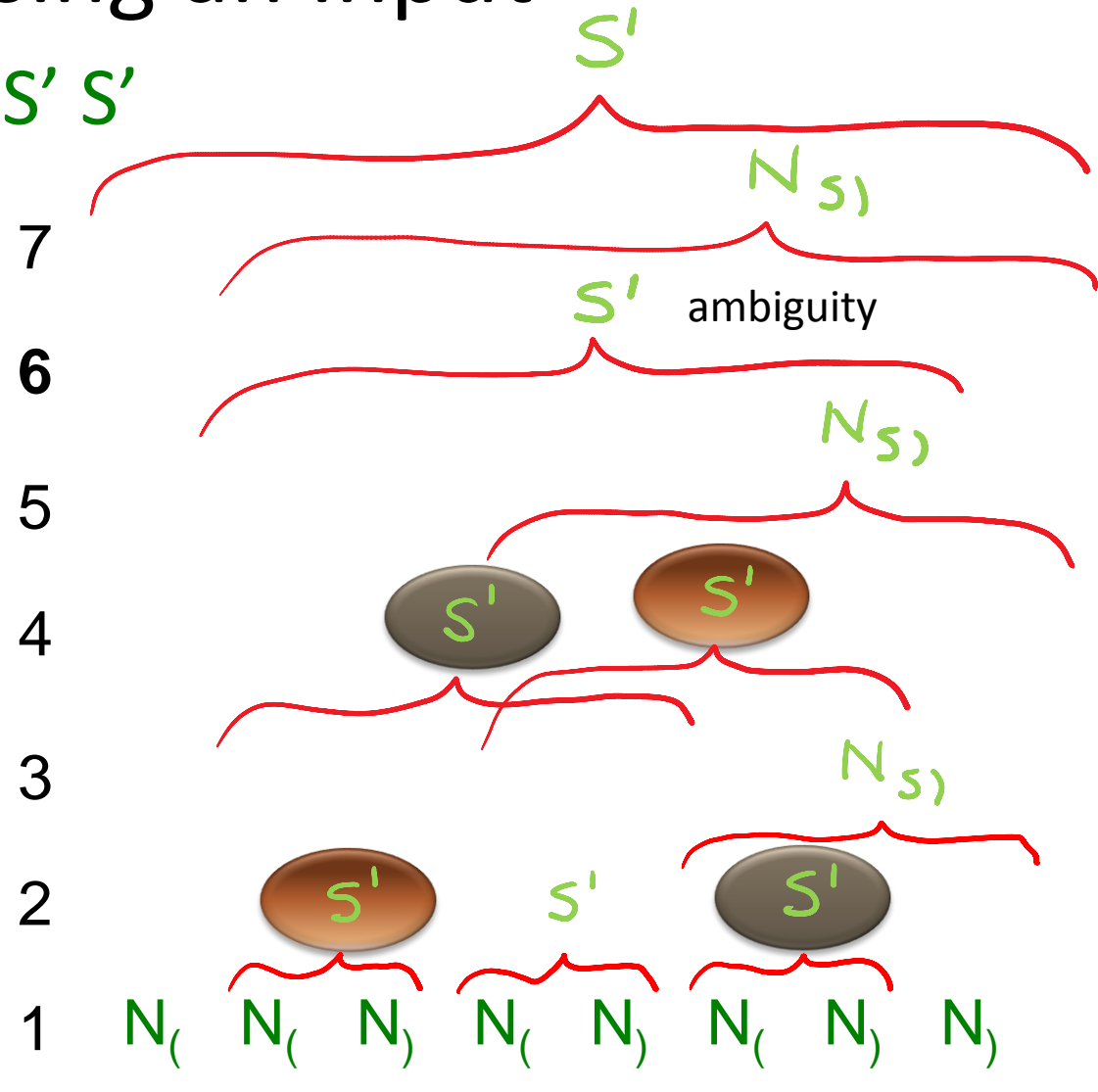
$$S' \rightarrow N_{(} N_{)} \mid N_{(} N_{)} \mid S' S'$$

$$N_{)} \rightarrow S' N_{(}$$

$$N_{(} \rightarrow ($$

$$N_{)} \rightarrow )$$

substring  
length



(	(	)	(	)	(	)	)
---	---	---	---	---	---	---	---

# Parsing an Input

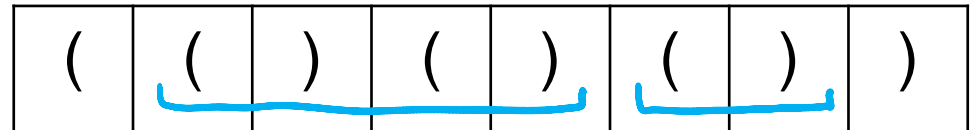
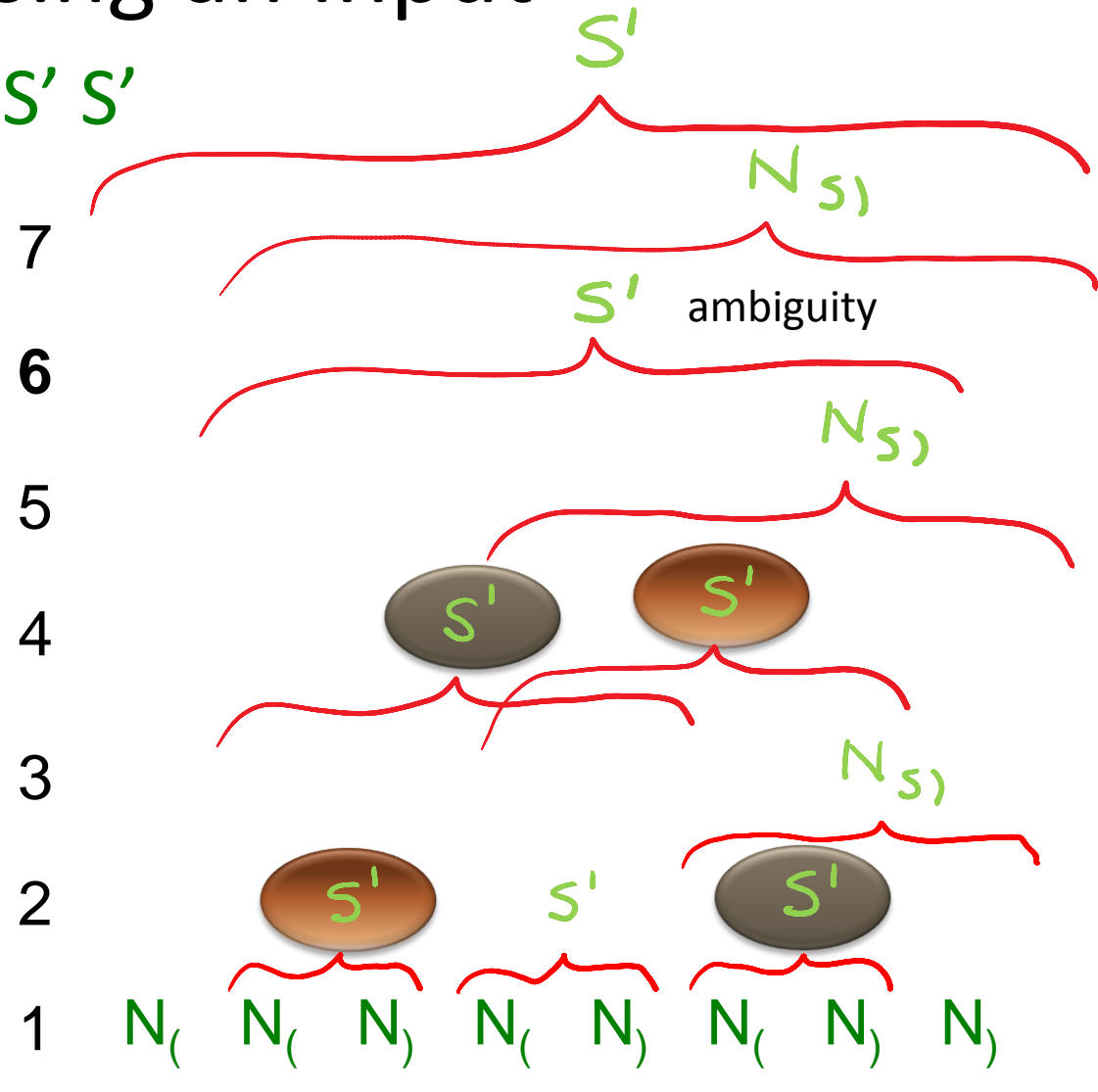
$$S' \rightarrow N_{(} N_{)} \mid N_{(} N_{)} \mid S' S'$$

$$N_{)} \rightarrow S' N_{(}$$

$$N_{(} \rightarrow ($$

$$N_{)} \rightarrow )$$

substring  
length



# Algorithm Idea

$$S' \rightarrow S' S'$$

$w_{pq}$  – substring from  $p$  to  $q$

$d_{pq}$  – all non-terminals that could expand to  $w_{pq}$

Initially  $d_{pp}$  has  $N_{w(p,p)}$

key step of the algorithm:

if  $X \rightarrow YZ$  is a rule,

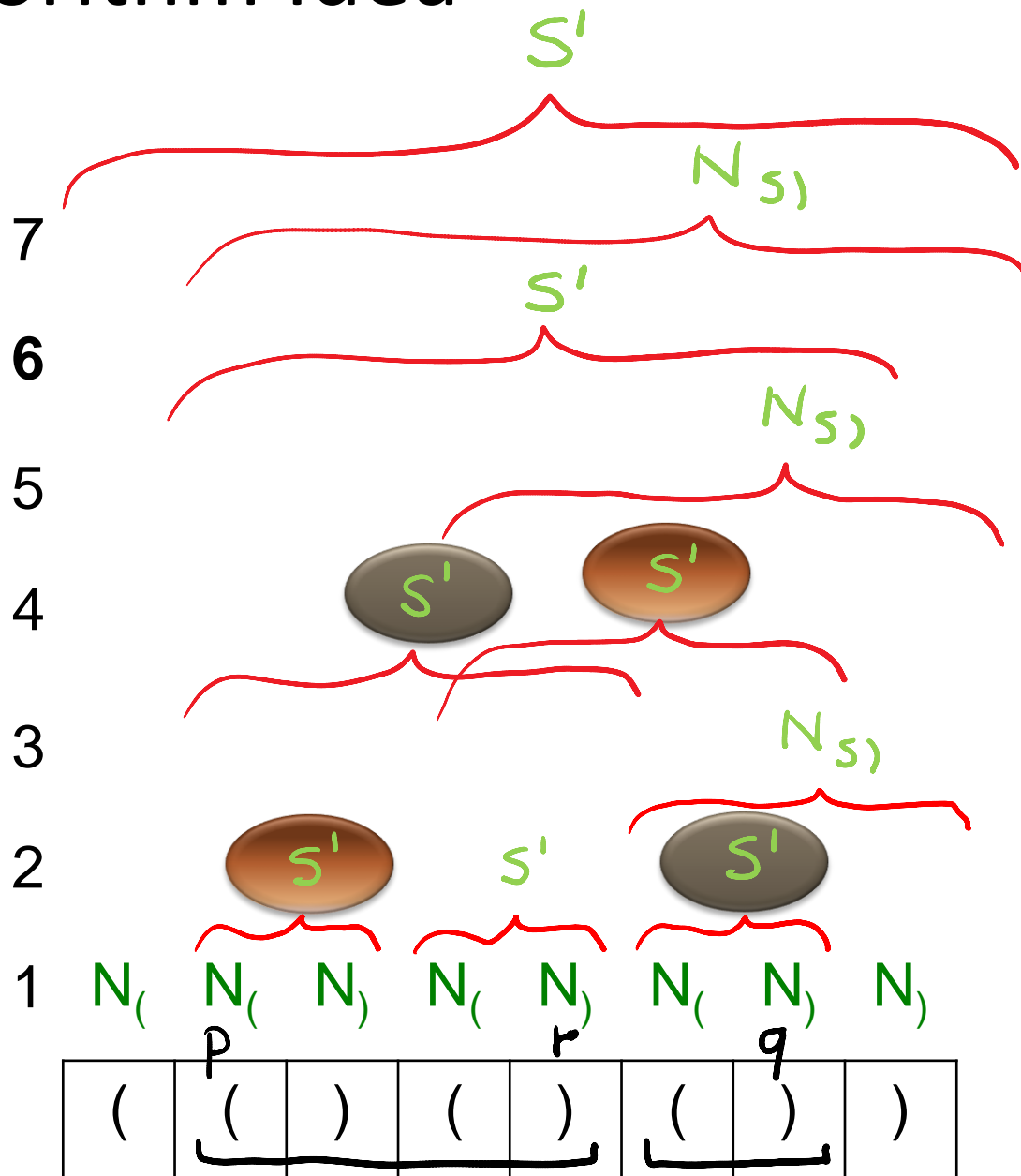
$Y$  is in  $d_{pr}$ , and

$Z$  is in  $d_{(r+1)q}$

then put  $X$  into  $d_{pq}$

( $p \leq r < q$ ),

in increasing value of  $(q-p)$



# Algorithm

INPUT: grammar  $G$  in Chomsky normal form  
word  $w$  to parse using  $G$

OUTPUT: true iff ( $w$  in  $L(G)$ )

$N = |w|$

var  $d$  : Array[ $N$ ][ $N$ ]

for  $p = 1$  to  $N$  {

$d(p)(p) = \{X \mid G \text{ contains } X \rightarrow w(p)\}$

for  $q$  in  $\{p + 1 .. N\}$   $d(p)(q) = \{\}$  }

for  $k = 2$  to  $N$  // substring length

for  $p = 0$  to  $N - k$  // initial position

for  $j = 1$  to  $k - 1$  // length of first half

val  $r = p + j - 1$ ; val  $q = p + k - 1$ ;

for  $(X ::= Y Z)$  in  $G$

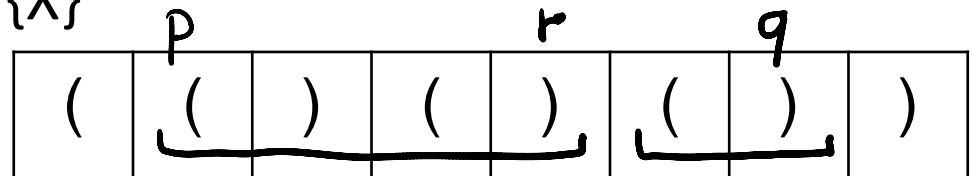
if  $Y$  in  $d(p)(r)$  and  $Z$  in  $d(r + 1)(q)$

$d(p)(q) = d(p)(q) \cup \{X\}$

return  $S$  in  $d(0)(N - 1)$

What is the running time  
as a function of grammar  
size and the size of input?

$$O(N^3 |G|)$$



# Parsing another Input

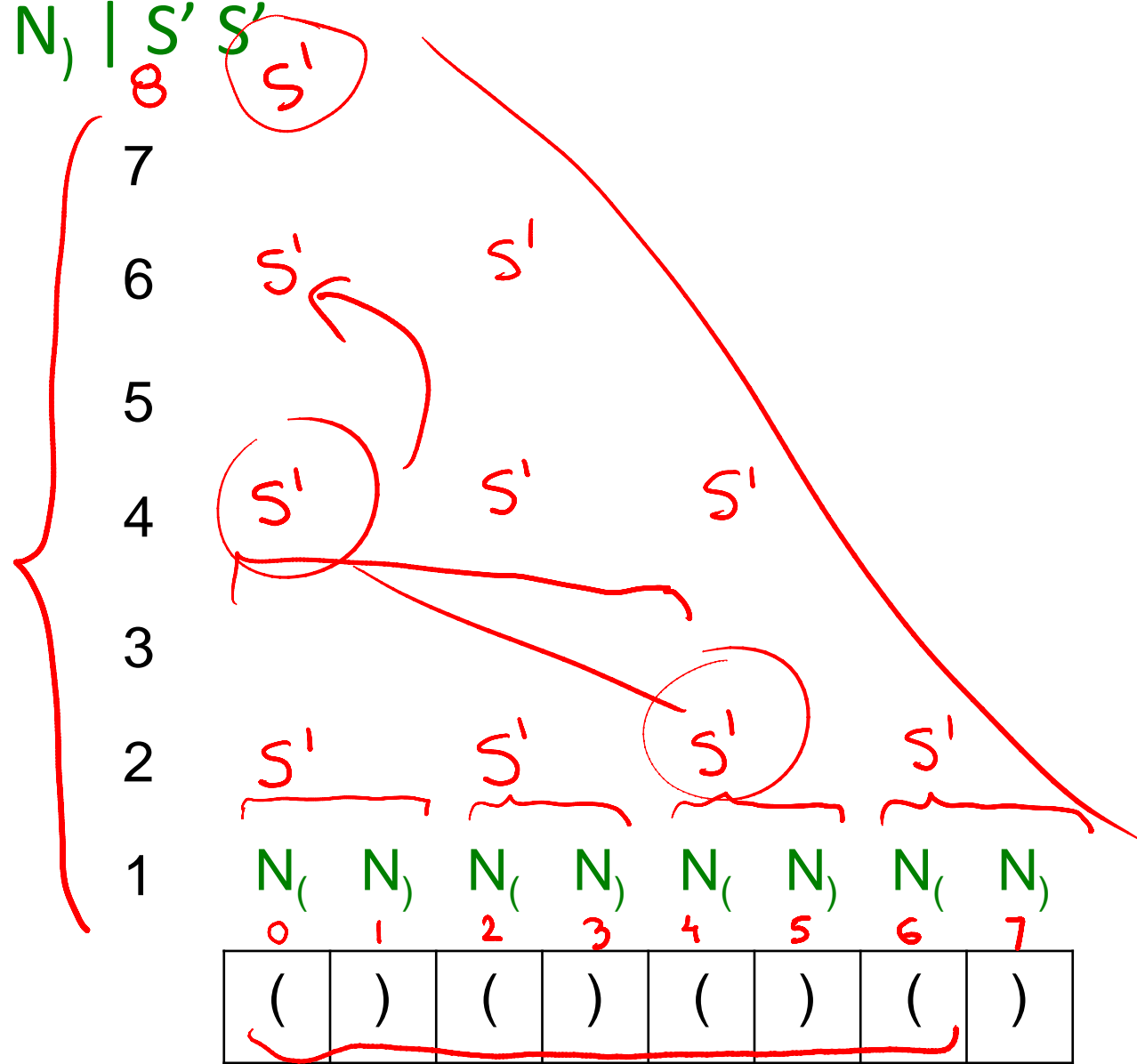
$S' \rightarrow N_{(} N_{S)} \mid N_{(} N_{)} \mid S' S'$

$N_{S)} \rightarrow S' N_{)}$

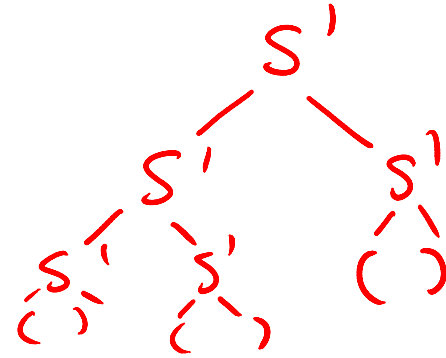
$N_{(} \rightarrow ($

$N_{)} \rightarrow )$

substring  
length



# Number of Parse Trees



- Let  $w$  denote word  $()()()$ 
  - it has two parse trees

- Give a lower bound on number of parse trees of the word  $w^n$  ( $n$  is positive integer)

$w^5$  is the word

$()()() ()()() ()()() ()()() ()()()$

$2^n$

- CYK represents all parse trees compactly
  - can re-run algorithm to extract first parse tree, or enumerate parse trees one by one

# Transforming to Chomsky Form

- Steps:
  1. remove unproductive symbols
  2. remove unreachable symbols
  3. remove epsilons (no non-start nullable symbols)
  4. remove single non-terminal productions  $X ::= Y$
  5. transform productions of arity more than two
  6. make terminals occur alone on right-hand side

$$X \rightarrow S_1 \dots S_n$$

# 1) Unproductive non-terminals

## How to compute them?

What is funny about this grammar:

$stmt ::= identifier := identifier$

$| while (expr) stmt$

$| if (expr) stmt else stmt$

$expr ::= term + term | term - term$

$term ::= factor * factor$

$factor ::= ( expr )$

There is no derivation of a sequence of tokens from  $expr$

Why? In every step will have at least one  $expr$ ,  $term$ , or  $factor$

If it cannot derive sequence of tokens we call it *unproductive*



# 1) Unproductive non-terminals

- Productive symbols are obtained using these two rules (what remains is unproductive)
  - Terminals are productive
  - If  $X ::= s_1 s_2 \dots s_n$  is rule and each  $s_i$  is productive then  $X$  is productive

**stmt** ::= identifier := identifier

| ~~while (expr) stmt~~

| ~~if (expr) stmt else stmt~~

~~expr ::= term + term | term - term~~

~~term ::= factor \* factor~~

~~factor ::= ( expr )~~

**program** ::= **stmt** | stmt program

Delete unproductive symbols.

Will the meaning of top-level symbol (program) change?

## 2) Unreachable non-terminals

What is funny about this grammar with starting terminal 'program'

program ::= stmt | stmt program

stmt ::= assignment | whileStmt

assignment ::= expr = expr

**ifStmt** ::= if (expr) stmt else stmt

whileStmt ::= while (expr) stmt

expr ::= identifier

No way to reach symbol 'ifStmt' from 'program'

What is the general algorithm?

## 2) Unreachable non-terminals

- Reachable terminals are obtained using the following rules (the rest are unreachable)
  - starting non-terminal is reachable (program)
  - If  $X ::= s_1 s_2 \dots s_n$  is rule and  $X$  is reachable then each non-terminal among  $s_1 s_2 \dots s_n$  is reachable

Delete unreachable symbols.

Will the meaning of top-level symbol (program) change?

### 3) Removing Empty Strings

Ensure only top-level symbol can be nullable

program ::= stmtSeq

stmtSeq ::= stmt | stmt ; stmtSeq

stmt ::= "" | assignment | whileStmt | blockStmt

blockStmt ::= { stmtSeq }

assignment ::= expr = expr

whileStmt ::= while (expr) stmt

expr ::= identifier

How to do it in this example?

### 3) Removing Empty Strings - Result

```
program ::= "" | stmtSeq
stmtSeq ::= stmt | stmt ; stmtSeq |
           | ; stmtSeq | stmt ; | ;
stmt ::= assignment | whileStmt | blockStmt
blockStmt ::= { stmtSeq } | { }
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
whileStmt ::= while (expr)
expr ::= identifier
```

### 3) Removing Empty Strings - Algorithm

- Compute the set of nullable non-terminals
- Add extra rules
  - If  $X ::= s_1 s_2 \dots s_n$  is rule then add new rules of form
$$X ::= r_1 r_2 \dots r_n \quad 2^n$$
where  $r_i$  is either  $s_i$  or, if  $s_i$  is nullable then  $r_i$  can also be the empty string (so it disappears)
- Remove all empty right-hand sides
- If starting symbol  $S$  was nullable, then introduce a new start symbol  $S'$  instead, and add rule  $S' ::= S \mid ""$

### 3) Removing Empty Strings

- Since `stmtSeq` is nullable, the rule

`blockStmt ::= { stmtSeq }`

gives

`blockStmt ::= { stmtSeq } | { }`

- Since `stmtSeq` and `stmt` are nullable, the rule

`stmtSeq ::= stmt | stmt ; stmtSeq`

gives

`stmtSeq ::= stmt | stmt ; stmtSeq  
| ; stmtSeq | stmt ; | ;`

# 4) Eliminating single productions

- Single production is of the form

$X ::= Y$

where  $X, Y$  are non-terminals

$\text{program} ::= \text{stmtSeq}$

$\text{stmtSeq} ::= \text{stmt}$

$\quad \quad \quad | \text{stmt} ; \text{stmtSeq}$

$\text{stmt} ::= \text{assignment} | \text{whileStmt}$

$\text{assignment} ::= \text{expr} = \text{expr}$

$\text{whileStmt} ::= \text{while} (\text{expr}) \text{stmt}$



## 4) Eliminate single productions - Result

- Generalizes removal of epsilon transitions from non-deterministic automata

program ::= expr = expr | while (expr) stmt  
          | stmt ; stmtSeq

stmtSeq ::= expr = expr | while (expr) stmt  
          | stmt ; stmtSeq

stmt ::= expr = expr | while (expr) stmt

assignment ::= expr = expr

whileStmt ::= while (expr) stmt

} now unreachable

## 4) “Single Production Terminator”

- If there is single production  
 $X ::= Y$  put an edge  $(X, Y)$  into graph
- If there is a path from  $X$  to  $Z$  in the graph, and there is rule  $Z ::= s_1 s_2 \dots s_n$  then add rule  
 $X ::= s_1 s_2 \dots s_n$

At the end, remove all single productions.

$\text{program} ::= \text{expr} = \text{expr} \mid \text{while}(\text{expr}) \text{stmt}$   
 $\quad \mid \text{stmt} ; \text{stmtSeq}$

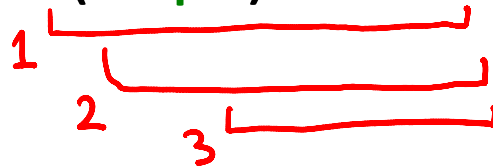
$\text{stmtSeq} ::= \text{expr} = \text{expr} \mid \text{while}(\text{expr}) \text{stmt}$   
 $\quad \mid \text{stmt} ; \text{stmtSeq}$

$\text{stmt} ::= \text{expr} = \text{expr} \mid \text{while}(\text{expr}) \text{stmt}$

## 5) No more than 2 symbols on RHS

$\text{stmt} ::= \text{while } (\text{expr}) \text{ stmt}$

becomes



$\text{stmt} ::= \text{while } \text{stmt}_1$

$\text{stmt}_1 ::= ( \text{stmt}_2$

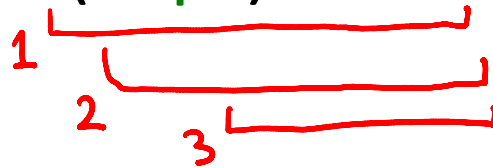
$\text{stmt}_2 ::= \text{expr } \text{stmt}_3$

$\text{stmt}_3 ::= ) \text{stmt}$

## 6) A non-terminal for each terminal

$\text{stmt} ::= \text{while } (\text{expr}) \text{ stmt}$

becomes



$\text{stmt} ::= N_{\text{while}} \text{stmt}_1$

$\text{stmt}_1 ::= N_{(} \text{stmt}_2$

$\text{stmt}_2 ::= \text{expr} \text{stmt}_3$

$\text{stmt}_3 ::= N_{)} \text{stmt}$

$N_{\text{while}} ::= \text{while}$

$N_{(} ::= ($

$N_{)} ::= )$

# Parsing using CYK Algorithm

- Transform grammar into Chomsky Form:
  1. remove unproductive symbols
  2. remove unreachable symbols
  3. remove epsilons (no non-start nullable symbols)
  4. remove single non-terminal productions  $X ::= Y$
  5. transform productions of arity more than two
  6. make terminals occur alone on right-hand sideHave only rules  $X ::= Y Z$ ,  $X ::= t$ , and possibly  $S ::= \epsilon$
- Apply CYK dynamic programming algorithm