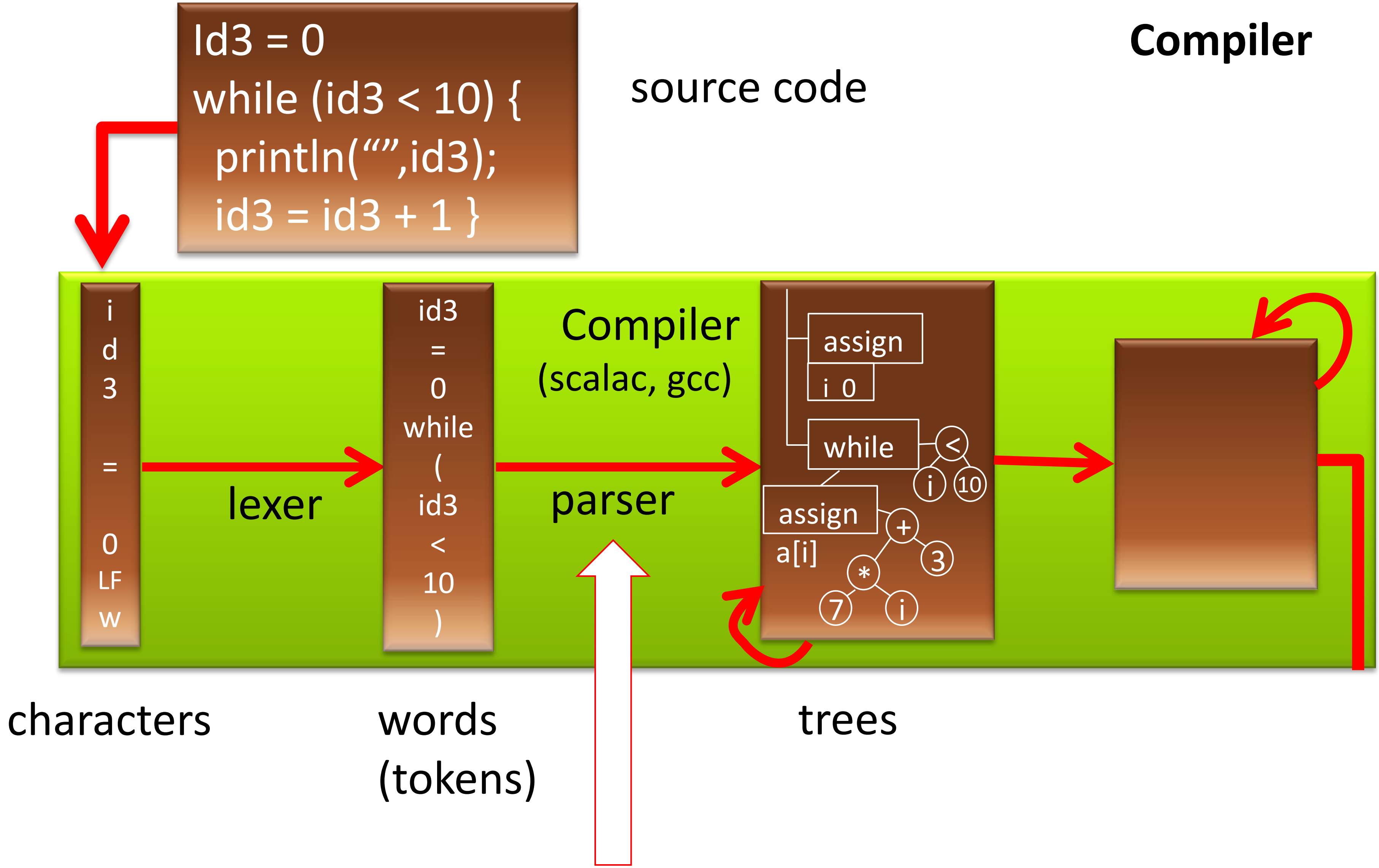
# Syntax Trees

# Compiler



Id3 = 0
while (id3 < 10) {
println(“”,id3);
id3 = id3 + 1 }
source code
i
d
3
=
0
LF
w
lexer
id3
=
0
while
(
id3
<
10
)
Compiler
(scalac, gcc)
parser
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
characters
words
(tokens)
trees

# Trees for Statements

```
statmt ::= println ( stringConst , ident )
         | ident = expr
         | if ( expr ) statmt (else statmt)?
         | while ( expr ) statmt
         | { statmt* }
```

```
abstract class Statmt
case class PrintlnS(msg : String, var : Identifier) extends Statmt
case class Assignment(left : Identifier, right : Expr) extends Statmt
case class If(cond : Expr, trueBr : Statmt,
                    falseBr : Option[Statmt]) extends Statmt
case class While(cond : Expr, body : Expr) extends Statmt
case class Block(sts : List[Statmt]) extends Statmt
```

# Recursive Descent: Grammar -> Parser

statmt ::= ... | **while** ( expr ) statmt | ... grammar

**case class** While(cond : Expr, body : Expr) **extends** Statmt tree

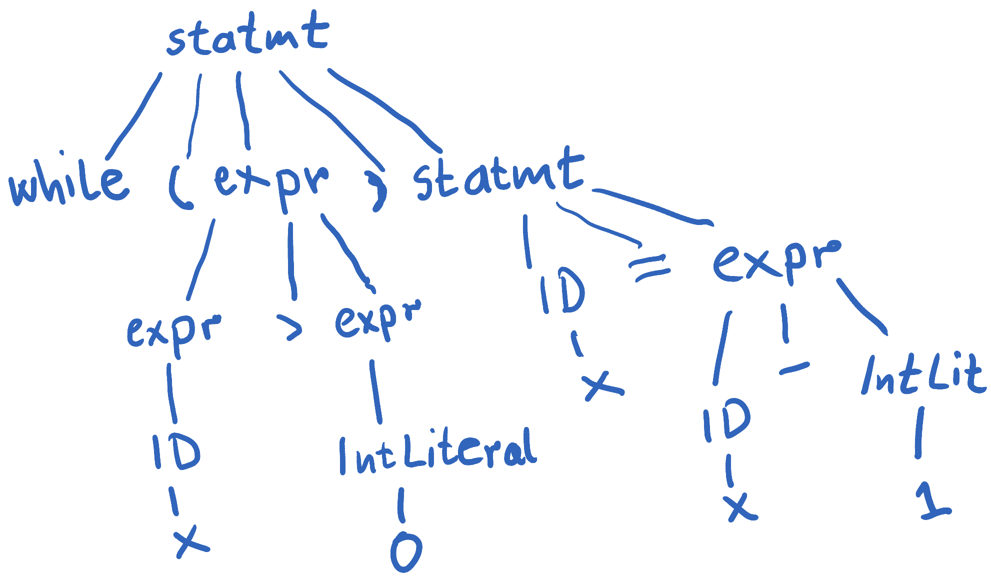
```
def statmt : Statmt = {
  // println ( stringConst , ident )
  if (lexer.token == Println) {
  ...
  } else if (lexer.token == WhileKeyword) {   // fill in missing parts

    val cond =

    val body =
    While(cond,body)
```

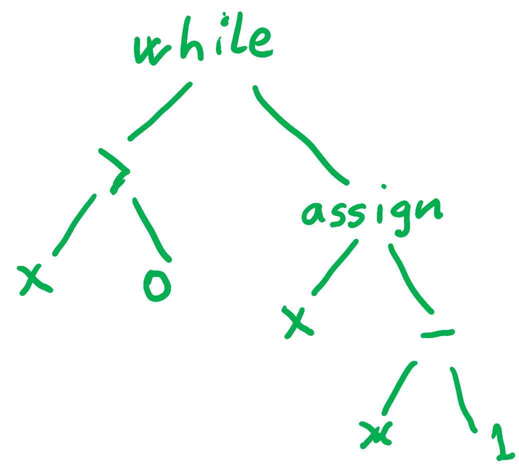
# Hint: Constructing Tree for 'if'

```
def expr : Expr = { ... }
// statmt ::=
def statmt : Statmt = {
  ...
// | while ( expr ) statmt
// case class If(cond : Expr, trueBr: Statmt, falseBr: Option[Statmt])
  } else if (lexer.token == ifKeyword) { lexer.next;
    skip(openParen); val c = expr; skip(closedParen);
    val trueBr = statmt
    val elseBr = if (lexer.token == elseKeyword) {
      lexer.next; Some(statmt) } else Nothing
    If(c, trueBr, elseBr)
```

# Parse Tree vs Abstract Syntax Tree (AST)

**while** (x > 0) x = x - 1




**Pretty printer:** takes abstract syntax tree (AST) and outputs the leaves of one possible (concrete) parse tree.

parse(prettyPrint(ast)) $\approx$ ast

# Beyond Statements:
# Parsing Expressions

# While Language with Simple Expressions

statmt ::=

println ( stringConst , ident )

| ident = expr

| if ( expr ) statmt (else statmt)?

| while ( expr ) statmt

| { statmt* }

expr ::= intLiteral | ident

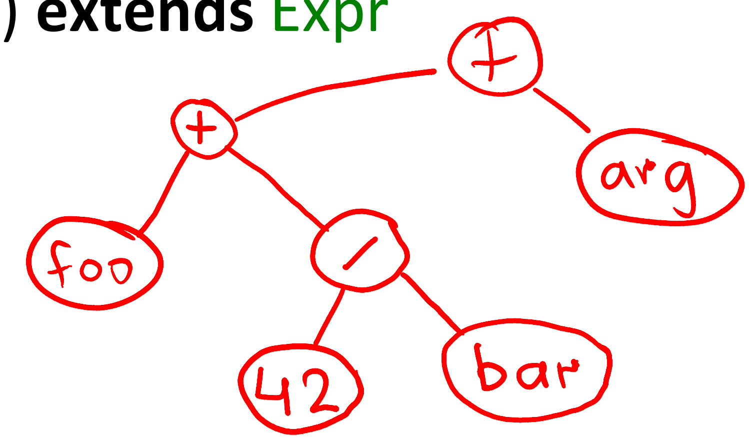
| expr ( + | / ) expr

# Abstract Syntax Trees for Expressions

expr ::= intLiteral | ident
| expr + expr | expr / expr

**abstract class** Expr
**case class** IntLiteral(x : Int) **extends** Expr
**case class** Variable(id : Identifier) **extends** Expr
**case class** Plus(e1 : Expr, e2 : Expr) **extends** Expr
**case class** Divide(e1 : Expr, e2 : Expr) **extends** Expr

foo + 42 / bar + arg



# Parser That Follows the Grammar?

```
expr ::= intLiteral | ident
       | expr + expr | expr / expr
```

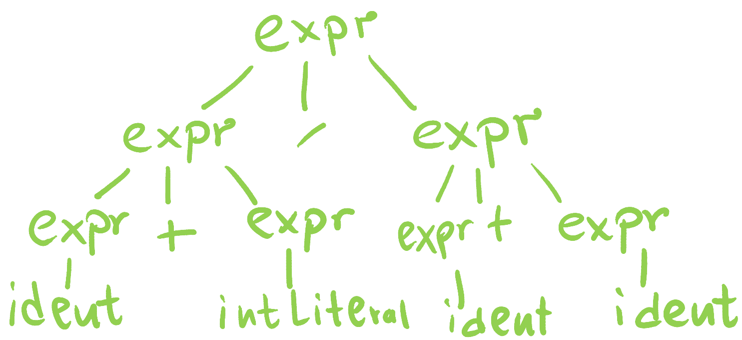
input:
foo + 42 / bar + arg

```
def expr : Expr = {
  if (??) IntLiteral(getInt(lexer.token))
  else if (??) Variable(getIdent(lexer.token))
  else if (??) {
    val e1 = expr; val op = lexer.token; val e2 = expr
    op match Plus {
     case PlusToken => Plus(e1, e2)
     case DividesToken => Divides(e1, e2)
    } }
```

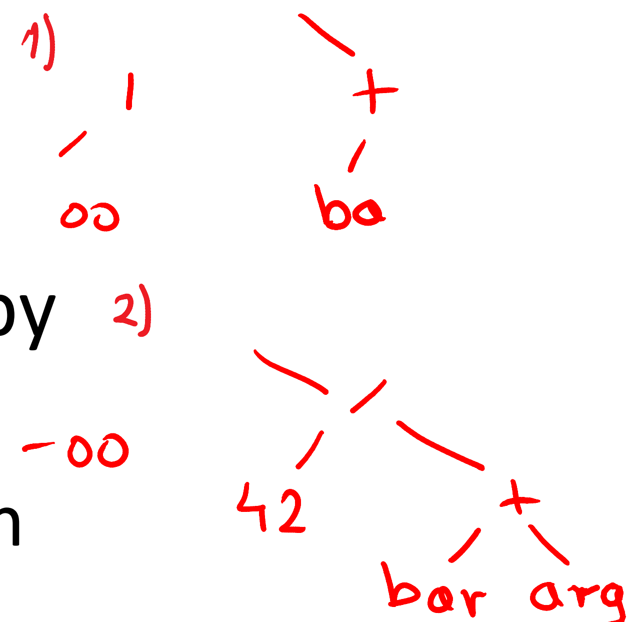
When should parser enter the recursive case?!

# Ambiguous Grammars

expr ::= intLiteral | ident
| expr + expr | expr / expr




foo + 42 / bar + arg




Each node in parse tree is given by one grammar alternative.

Ambiguous grammar: if some token sequence has multiple parse trees (then it is has multiple abstract trees).

# An attempt to rewrite the grammar

expr ::= simpleExpr (( + | / ) simpleExpr)*

simpleExpr ::= intLiteral | ident

```
def simpleExpr : Expr = { … }
def expr : Expr = {
  var e = simpleExpr
  while (lexer.token == PlusToken ||
           lexer.token == DividesToken)) {
    val op = lexer.token
    val eNew = simpleExpr
    op match {
      case TokenPlus => { e = Plus(e, eNew) }
      case TokenDiv => { e = Divide(e, eNew) }
    }
  }
  e }
```

foo + 42 / bar + arg


Not ambiguous, but gives wrong tree.

# **Solution:** Layer the grammar to express priorities

```
expr ::= term (+ term)*
term ::= simpleExpr (/ simpleExpr)*
simpleExpr ::= intLiteral | ident | ( expr )
```

```
def expr : Expr = {
  var e = term
  while (lexer.token == PlusToken) {
    lexer.next;   e = Plus(e, term)
  }
  e }
def term : Expr = {
 var e = simpleExpr
 ...
```

Decompose first by the least-priority operator (+)

# Using recursion instead of *

expr ::= term (+ term)*

expr ::= term (+ expr)?

```
def expr : Expr = {
  val e = term
  if (lexer.token == PlusToken) {
    lexer.next
    Plus(e, expr)
  } else e
}
def term : Expr = {
  val e = simpleExpr
  if (lexer.token == DivideToken) {
    lexer.next
    Divide(e, term)
  } else e
}
```

# Another Example for Building Trees

expr ::= ident | expr - expr | expr ^ expr | (expr)

where:

- "-" is left associative
- "^" is right associative
- "^" has higher priority (binds stronger) than "-"

Draw parentheses and a tree for token sequence:

**a – b – c ^ d ^ e – f**

**((a – b) – (c ^ (d ^ e)) ) – f**

left associative: x o y o z -> (x o y) o z (common case)

right associative: x o y o z -> x o (y o z)

# Goal: Build Expressions

**abstract class** Expr

**case class** Variable(id : Identifier) **extends** Expr

**case class** Minus(e1 : Expr, e2 : Expr) **extends** Expr

**case class** Exp(e1 : Expr, e2 : Expr) **extends** Expr

# 1) Layer the grammar by priorities

expr ::= ident | expr - expr | expr ^ expr | (expr)

expr ::= term (- term)*

term ::= factor (^ factor)*

factor ::= id | (expr)

lower priority binds weaker, so it goes outside

# 2) Building trees: left-associative "-"

**LEFT-associative** operator

x – y – z ➔ (x – y) – z

Minus(Minus(Var(“x”),Var(“y”)), Var(“z”))

```
def expr : Expr = {
  var e = term
  while (lexer.token == MinusToken) {
    lexer.next
    e = Minus(e, term)
  }
  e
}
```

# 3) Building trees: right-associative "^"

**RIGHT-associative** operator – using recursion
(or also loop and then reverse a list)

x ^ y ^ z ➔ x ^ (y ^ z)

Exp(Var(“x”), Exp(Var(“y”), Var(“z”)) )

```
def expr : Expr = {
  val e = factor
  if (lexer.token == ExpToken) {
    lexer.next
    Exp(e, expr)
  } else e
}
```

# Exercise: Unary Minus

**1)** Show that the grammar

$$A ::= -A$$
$$A ::= A - id$$
$$A ::= id$$

is ambiguous by finding a string that has two different syntax trees.

**2)** Make two different unambiguous grammars for the same language:

**a)** One where prefix minus binds stronger than infix minus.

**b)** One where infix minus binds stronger than prefix minus.

3) Show the syntax trees using the new grammars for the string you used to prove the original grammar ambiguous.

# Exercise: Balanced Parentheses

Show that the following balanced parentheses grammar is ambiguous (by finding two parse trees for some input sequence) and find unambiguous grammar for the same language.

B ::= $\varepsilon$ | **(** B **)** | B B

# Dangling Else

The dangling-else problem happens when the conditional statements are parsed using the following grammar.

S ::= S **;** S

S ::= id **:=** E

S ::= **if** E **then** S

S ::= **if** E **then** S else S

Find an unambiguous grammar that accepts the same conditional statements and matches the else statement with the nearest unmatched if.

# Left Recursive and Right Recursive

We call a production rule “left recursive” if it is of the form

A ::= A p

for some sequence of symbols p. Similarly, a "right-recursive" rule is of a form

A ::= q A

Is every context free grammar that contains both left and right recursive rule for a some nonterminal A ambiguous?

# 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

$S \rightarrow$ “” | ( S ) | S S

Modified grammar in Chomsky Normal Form:

$S \rightarrow$ “” | S’ $\leftarrow$ if "" $\in L(G)$

$$\left.\begin{array}{l} S' \rightarrow N_{(}^{2} N_{S)} \mid N_{(}^{2} N_{)} \mid S'^{3} S' \\ N_{S)} \rightarrow S'_{2} N_{)} \end{array}\right\} \text{Rules } N \rightarrow N_1 N_2$$

nonterminals

$$\left.\begin{array}{l} N_{(} \rightarrow ( \\ N_{)} \rightarrow ) \end{array}\right\} \text{Rules } N \rightarrow t$$

nonterminal terminal

- Terminals: ( ) Nonterminals: S S’ $N_{S)}$ $N_{)}$ $N_{(}$

nonterminal with funny name

# Idea How We Obtained the Grammar

$S \rightarrow ( \; S \; )$

because S can be empty
but S' cannot

$S' \rightarrow N_( \; N_{S)} \quad | \quad N_( \; N_)$

$N_( \rightarrow ($

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

$N_) \rightarrow )$

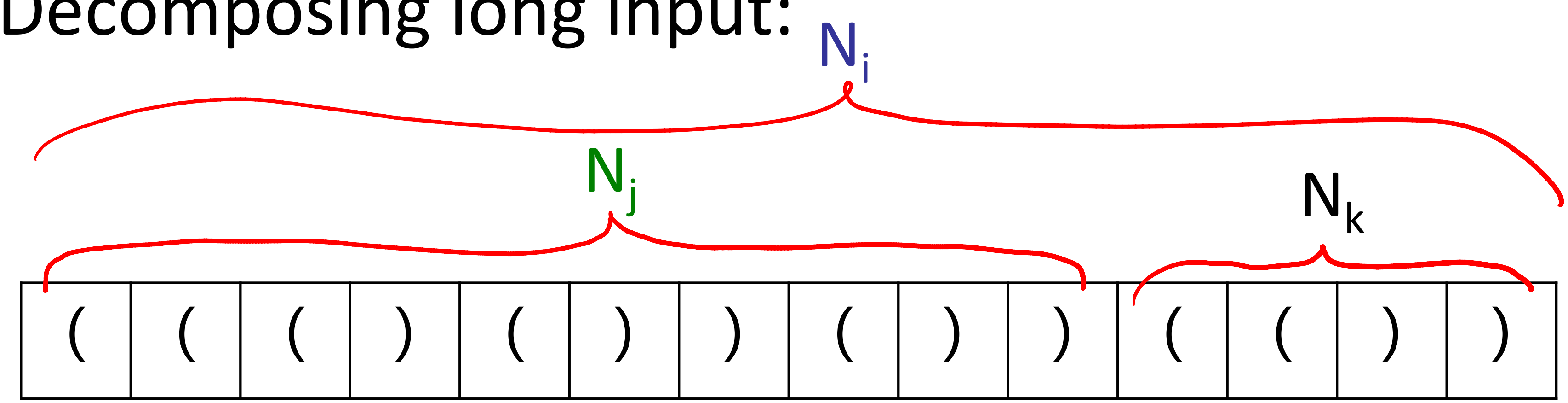
Chomsky Normal Form transformation
can be done fully mechanically

# Dynamic Programming to Parse Input

Assume Chomsky Normal Form, 3 types of rules:

| | |
|---|---|
| $S \to$ “” \| $S'$ | (only for the start non-terminal) |
| $N_j \to t$ | (names for terminals) |
| $N_i \to N_j\ N_k$ | (just **2** non-terminals on RHS) |

Decomposing long input:




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

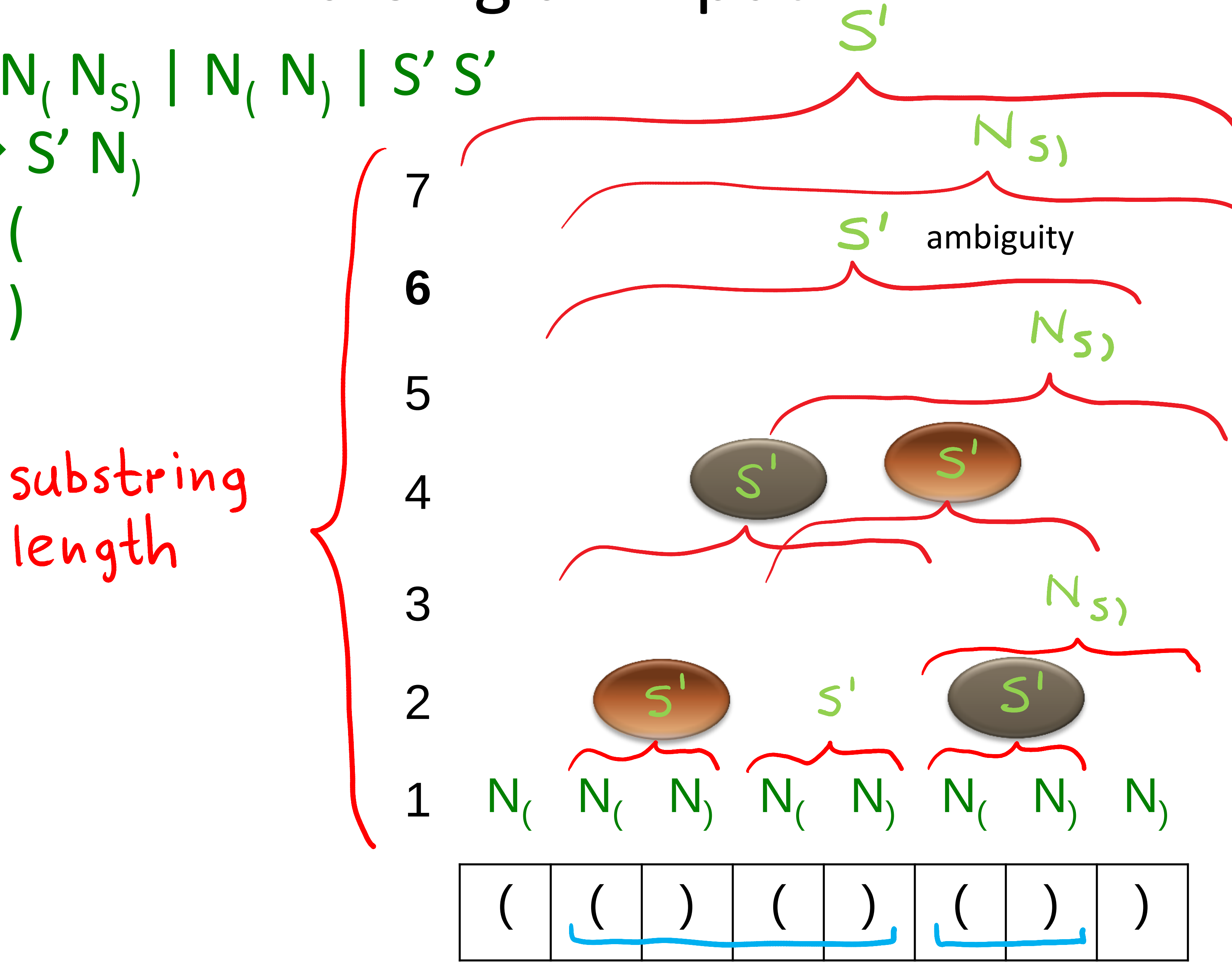
# Parsing an Input

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

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

$N_( \rightarrow ($

$N_) \rightarrow )$



# Parsing an Input

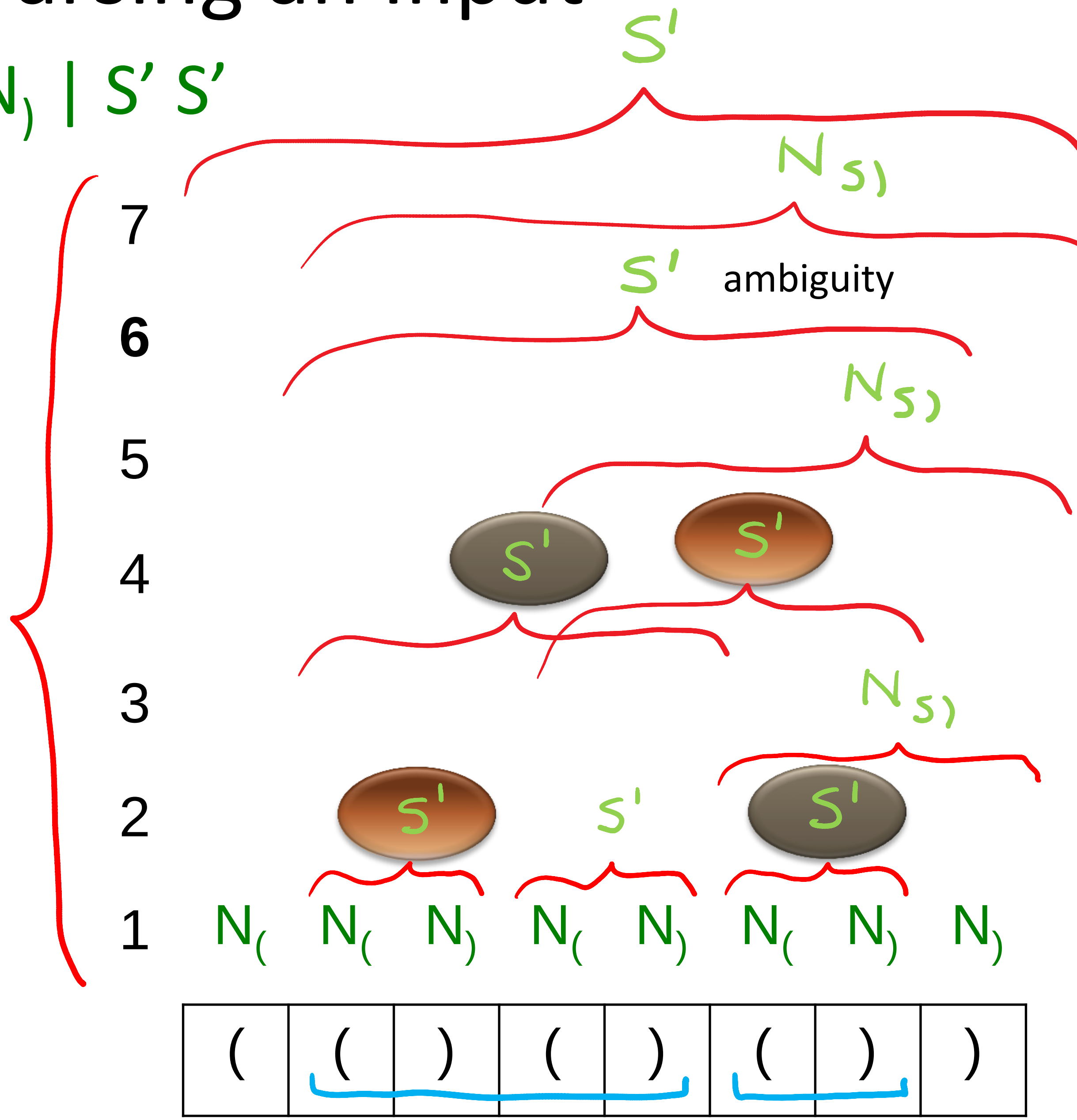
$S' \rightarrow N_( \, N_{S)} \mid N_( \, N_) \mid S' \, S'$

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

$N_( \rightarrow ($

$N_) \rightarrow )$

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 $X \rightarrow Y \; Z$ is a rule,

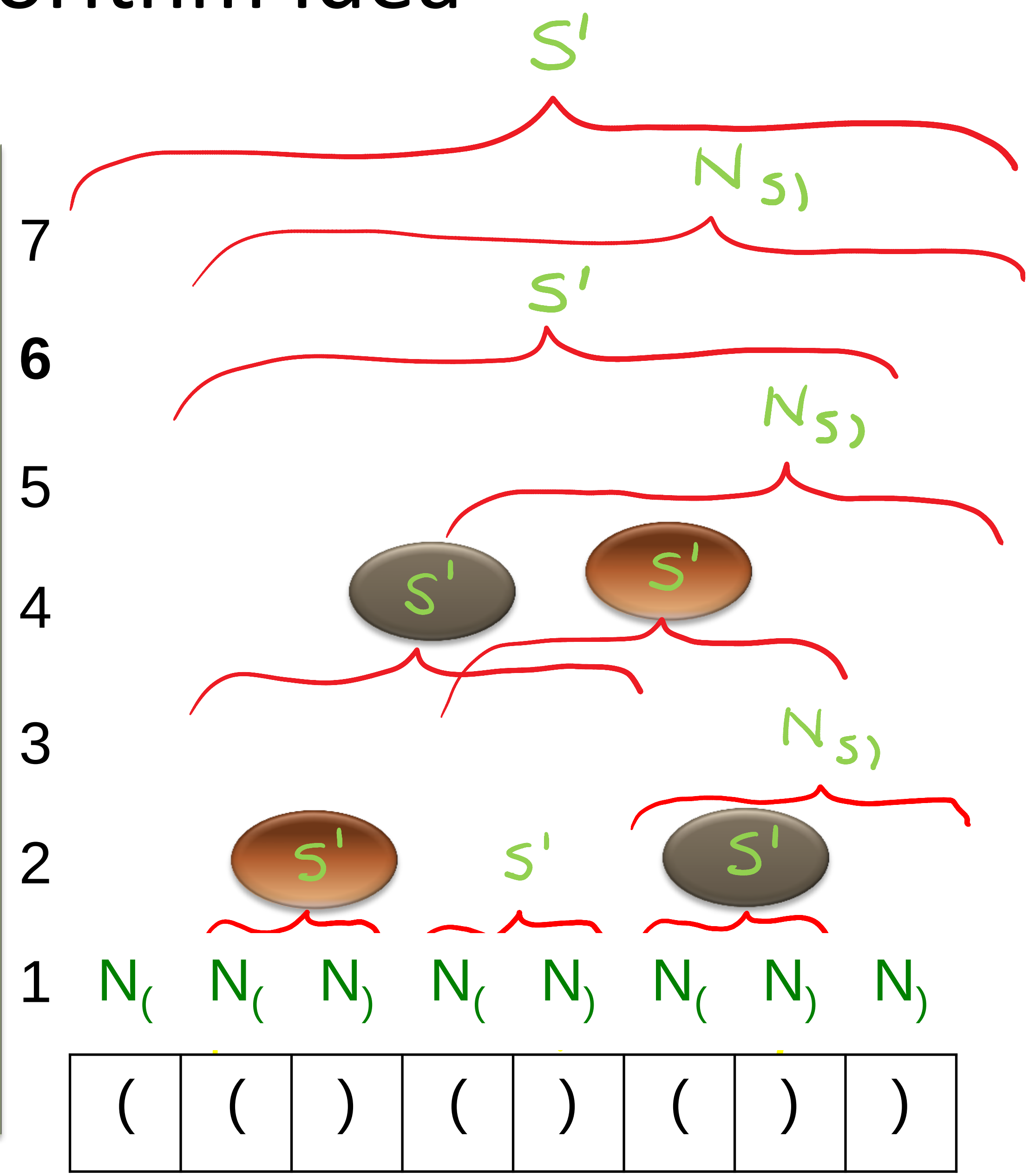
Y is in $d_{p\,r}$ , and

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 | G contains X->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) union {X}
return  S in d(0)(N-1)
```

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

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | p | | | r | | q | |
| ( | ( | ) | ( | ) | ( | ) | ) |

# Parsing another Input

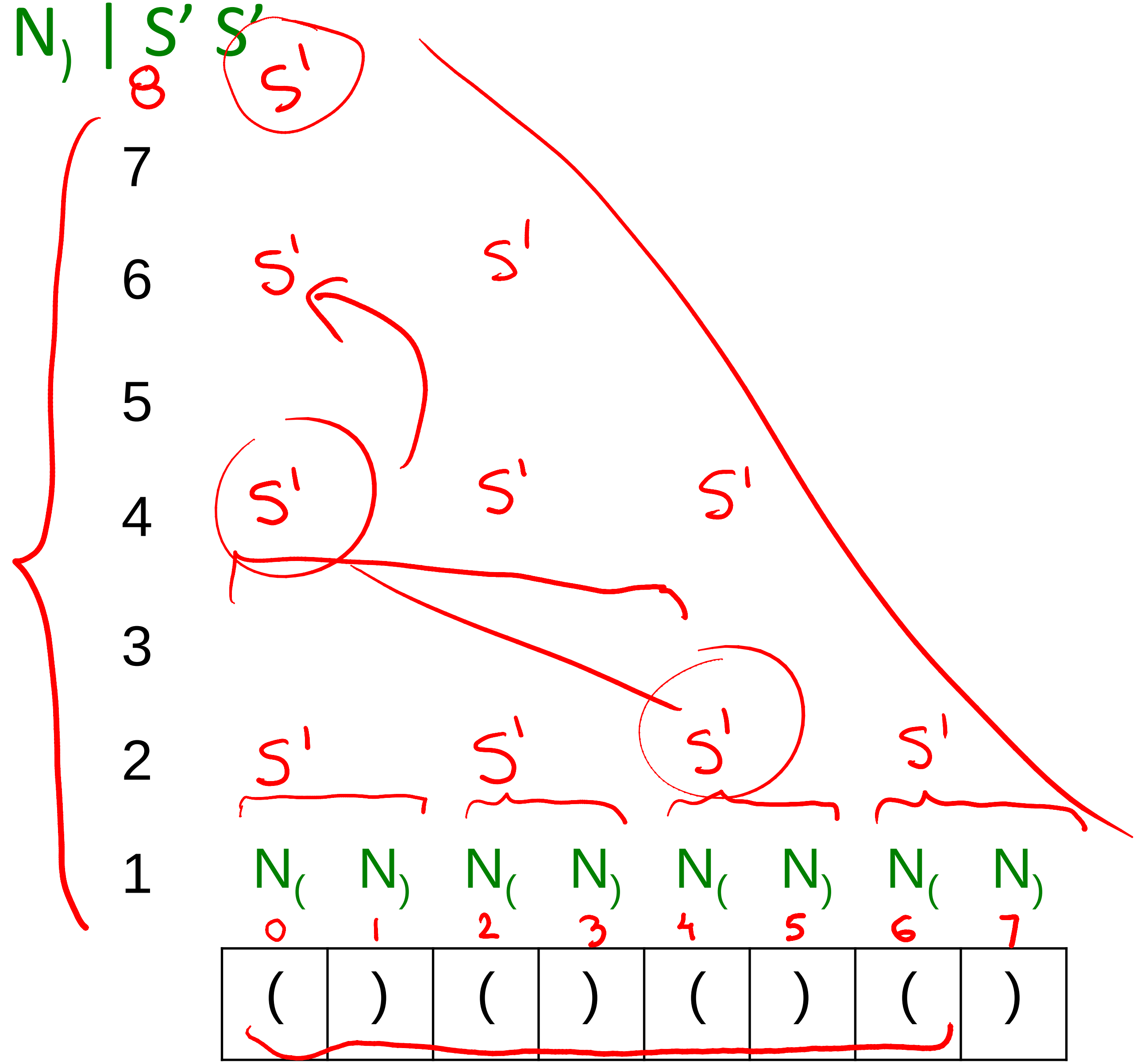
$S' \rightarrow N_( \, N_{S)} \mid N_( \, N_) \mid S' \, S'$

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

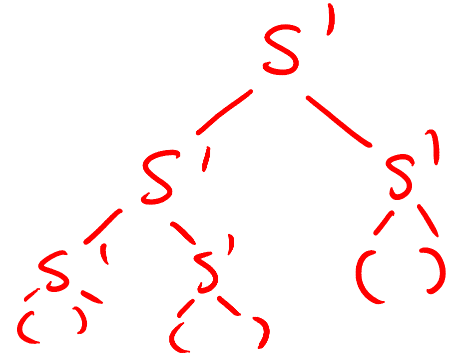
$N_( \rightarrow ($

$N_) \rightarrow )$

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

# 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 $X \rightarrow Y Z$ is a rule,

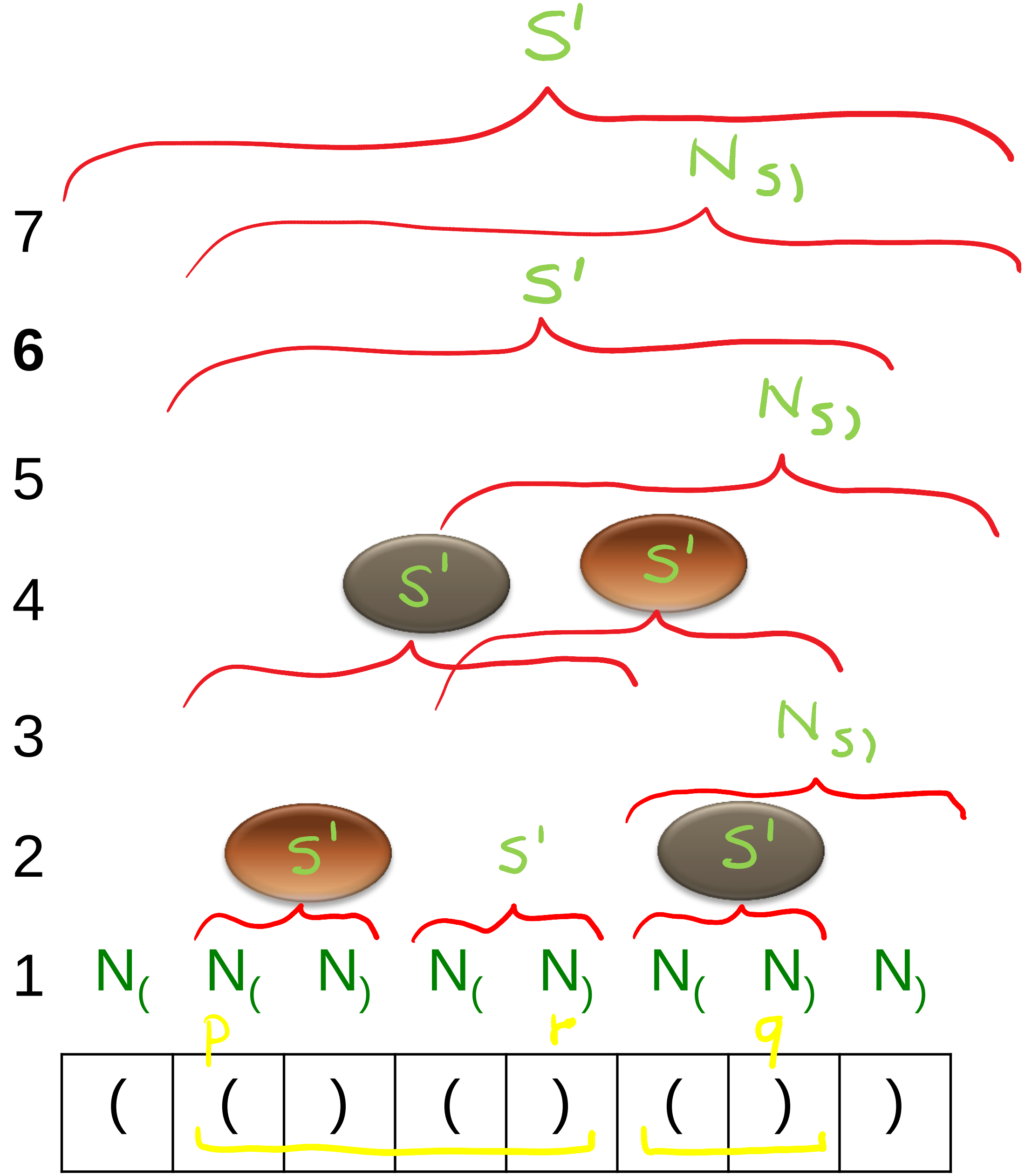
$Y$ is in $d_{p\,r}$ , and

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

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

($p \leq r < q$),

in increasing value of (q-p)



# 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

$X \rightarrow S_1 \ldots S_n$

# 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::= $s_1$ $s_2$ … $s_n$ is rule and each $s_i$ is productive then X is productive

stmt ::= identifier := identifier<br>
~~| while (expr) stmt~~<br>
~~| if (expr) stmt else stmt~~<br>
~~expr ::= term + term | term – term~~<br>
~~term ::= factor * factor~~<br>
~~factor ::= ( expr )~~<br>
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'

# 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

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 $X ::= s_1\ s_2 \ldots s_n$ is rule and X is reachable then each non-terminal among $s_1\ s_2 \ldots s_n$ is reachable

Delete unreachable 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

# 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 $X ::= s_1\ s_2\ \ldots\ s_n$ is rule then add new rules of form

    $X ::= r_1\ r_2\ \ldots\ r_n$ $2^n$

  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 | ""

# 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

```
program ::= stmtSeq
stmtSeq ::= stmt
          | stmt ; stmtSeq
stmt ::= assignment | whileStmt
assignment ::= expr = expr
whileStmt ::= while (expr) 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 $Z ::= s_1\ s_2 \ldots s_n$ then add rule

$$X ::= s_1\ s_2 \ldots s_n$$

At the end, remove all single productions.

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

# 5) No more than 2 symbols on RHS

stmt ::= while (expr) stmt

becomes

1

2

3

stmt ::= while $stmt_1$

$stmt_1$ ::= ( $stmt_2$

$stmt_2$ ::= expr $stmt_3$

$stmt_3$ ::= ) stmt

# 6) A non-terminal for each terminal

$$\text{stmt ::= while } \underbrace{\text{(}\underbrace{\text{expr}\underbrace{\text{) stmt}}_{3}}_{2}}_{1}$$

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 side

  Have only rules X ::= Y Z, X ::= t, and possibly S ::= “”
- Apply CYK dynamic programming algorithm