Computing if a token can follow

first(B_1 ... B_p) = {a$\in\Sigma$ | B_1...B_p $\Rightarrow$... $\Rightarrow$ aw }

follow(X) = {a$\in\Sigma$ | S $\Rightarrow$... $\Rightarrow$ ...Xa... }

There exists a derivation from the start symbol that produces a sequence of terminals and nonterminals of the form ...Xa...
(the token a follows the non-terminal X)

Rule for Computing Follow

Given X ::= YZ (for reachable X)

then **first**(Z) ⊆ **follow**(Y)

and **follow**(X) ⊆ **follow**(Z)

S => Xa => YZa => Yba

S ::= Xa
X ::= YZ
Y ::= b
Z ::= c

Now take care of nullable ones as well:

For each rule $X ::= Y_1 \ldots Y_p \ldots Y_q \ldots Y_r$

follow(Y_p) should contain:

- **first(**$Y_{p+1}Y_{p+2}\ldots Y_r$)
- also **follow**(X) if **nullable**($Y_{p+1}Y_{p+2}Y_r$)

Compute nullable, first, follow

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

assign ::= **ID = ID ;**

block ::= **beginof ID** stmtList **ID ends**

Compute **follow** (for that we need **nullable**,**first**)

Conclusion of the Solution

The grammar is not LL(1) because we have

- **nullable**(stmtList)
- **first**(stmt) ∩ **follow**(stmtList) = {**ID**}

- If a recursive-descent parser sees **ID**, it does not know if it should
  - finish parsing stmtList or
  - parse another stmt

LL(1) Grammar - good for building recursive descent parsers

- Grammar is LL(1) if for each nonterminal X
  - first sets of different alternatives of X are disjoint
  - if nullable(X), first(X) must be disjoint from follow(X)
- For each LL(1) grammar we can build recursive-descent parser
- Each LL(1) grammar is unambiguous
- If a grammar is not LL(1), we can sometimes transform it into equivalent LL(1) grammar

Table for LL(1) Parser: Example

S ::= B **EOF**
(1)

B ::= ε | B **(**B**)**
(1) **(2)**

nullable: B

first(S) = { **(** }
follow(S) = {}

first(B) = { **(** }
follow(B) = { **)**, **(**, **EOF** }

empty entry:
when parsing S,
if we see) ,
report error

Parsing table:

	EOF	**(**	**)**
S	{1}	{1}	{}
B	{1}	{1,2}	{1}

parse conflict - choice ambiguity:
grammar not LL(1)

1 is in entry because **(** is in follow(B)
2 is in entry because **(** is in first(B(B))

Table for LL(1) Parsing

Tells which alternative to take, given current token:

choice : Nonterminal x Token -> Set[Int]

A ::= **(1)** $B_1 \ldots B_p$
| **(2)** $C_1 \ldots C_q$
| **(3)** $D_1 \ldots D_r$

if $t \in$ first($C_1 \ldots C_q$) add 2 to choice(A,t)

if $t \in$ follow(A) add K to choice(A,t)
where K is nullable alternative

For example, when parsing A and seeing token t

choice(A,t) = {2} means: parse alternative 2 ($C_1 \ldots C_q$)

choice(A,t) = {1} means: parse alternative 3 ($D_1 \ldots D_r$)

choice(A,t) = {} means: report syntax error

choice(A,t) = {2,3} : not LL(1) grammar

Transform Grammar for LL(1)

S ::= B **EOF**

B ::= ε | B **(**B**)**

(1) **(2)**

Transform the grammar so that parsing table has no conflicts.

Old parsing table:

	EOF	**(**	**)**
S	{1}	{1}	{}
B	{1}	{1,2}	{1}

conflict - choice ambiguity: grammar not LL(1)

1 is in entry because **(** is in follow(B)

2 is in entry because **(** is in first(B(B))

S ::= B **EOF**

B ::= ε | **(**B**)** B

(1) **(2)**

Left recursion is bad for LL(1)

	EOF	**(**	**)**
S			
B			

choice(A,t)

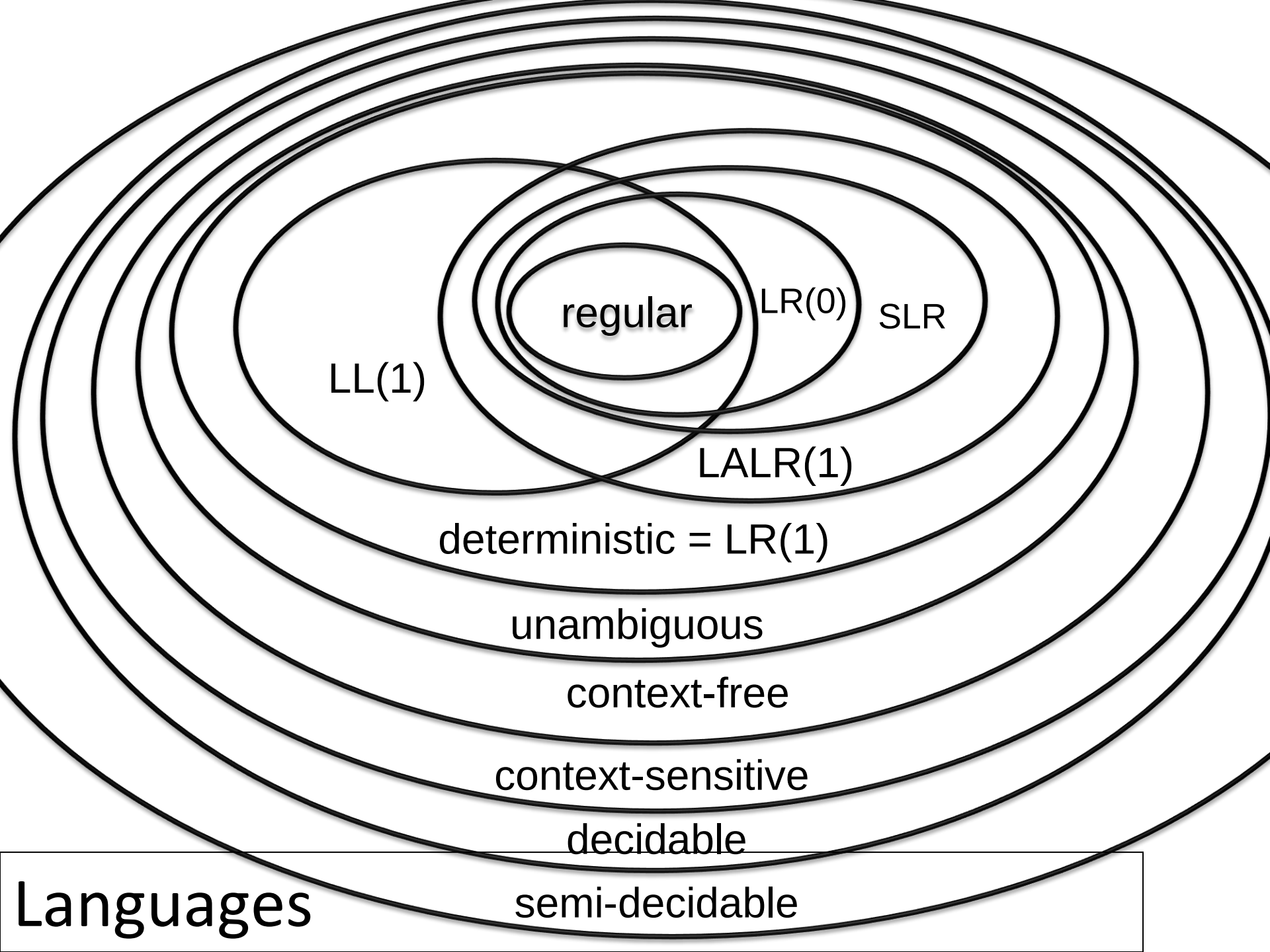
Parse Table is Code for Generic Parser

```
var stack : Stack[GrammarSymbol] // terminal or non-terminal
stack.push(EOF);
stack.push(StartNonterminal);
var lex = new Lexer(inputFile)
while (true) {
 X = stack.pop
  t = lex.curent
  if (isTerminal(X))
     if (t==X)   if (X==EOF) return success
                 else lex.next // eat token t
    else parseError("Expected " + X)
  else { // non-terminal
    cs = choice(X)(t) // look up parsing table
    cs match { // result is a set
    case {i} => { // exactly one choice
      rhs = p(X,i) // choose correct right-hand side
      stack.push(reverse(rhs)) }
    case {} => parseError("Parser expected an element of " + unionOfAll(choice(X)))
    case _ => crash(“parse table with conflicts - grammar was not LL(1)")
  }
}
```

What if we cannot transform the grammar into LL(1)?

1) Redesign your language

2) Use a more powerful parsing technique


regular
LR(0)
SLR
LL(1)
LALR(1)
deterministic = LR(1)
unambiguous
context-free
context-sensitive
decidable
semi-decidable
Languages

Remark: Grammars and Languages

- Language S is a set of words
- For each language S, there can be multiple possible grammars G such that S=L(G)
- Language S is
  - Non-ambiguous if there exists a non-ambiguous grammar for it
  - LL(1) if there is an LL(1) grammar for it
- Even if a language has ambiguous grammar, it can still be non-ambiguous if it also has a non-ambiguous grammar

Parsing General Grammars: Why

- Can be difficult or impossible to make grammar unambiguous

- Some inputs are more complex than simple programming languages
  - mathematical formulas:
    x = y /\ z ? (x=y) /\ z x = (y /\ z)
  - future programming languages
  - natural language:

    I saw the man with the telescope.

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)

Chomsky Normal Form

- Essentially, only binary rules
- Concretely, these kinds of rules:

X ::= Y Z	binary rule X,Y,Z - non-terminals
X ::= a	non-terminal as a name for token
S ::= ε	only for top-level symbol S

Balanced Parentheses Grammar

Original grammar G

$S \rightarrow \varepsilon \mid (S) \mid S\ S$

Modified grammar in Chomsky Normal Form:

$S \rightarrow \varepsilon \mid S'$ ← if "" ∈ L(G)

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

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

$N_(\rightarrow ($ } Rules

$N_) \rightarrow)$

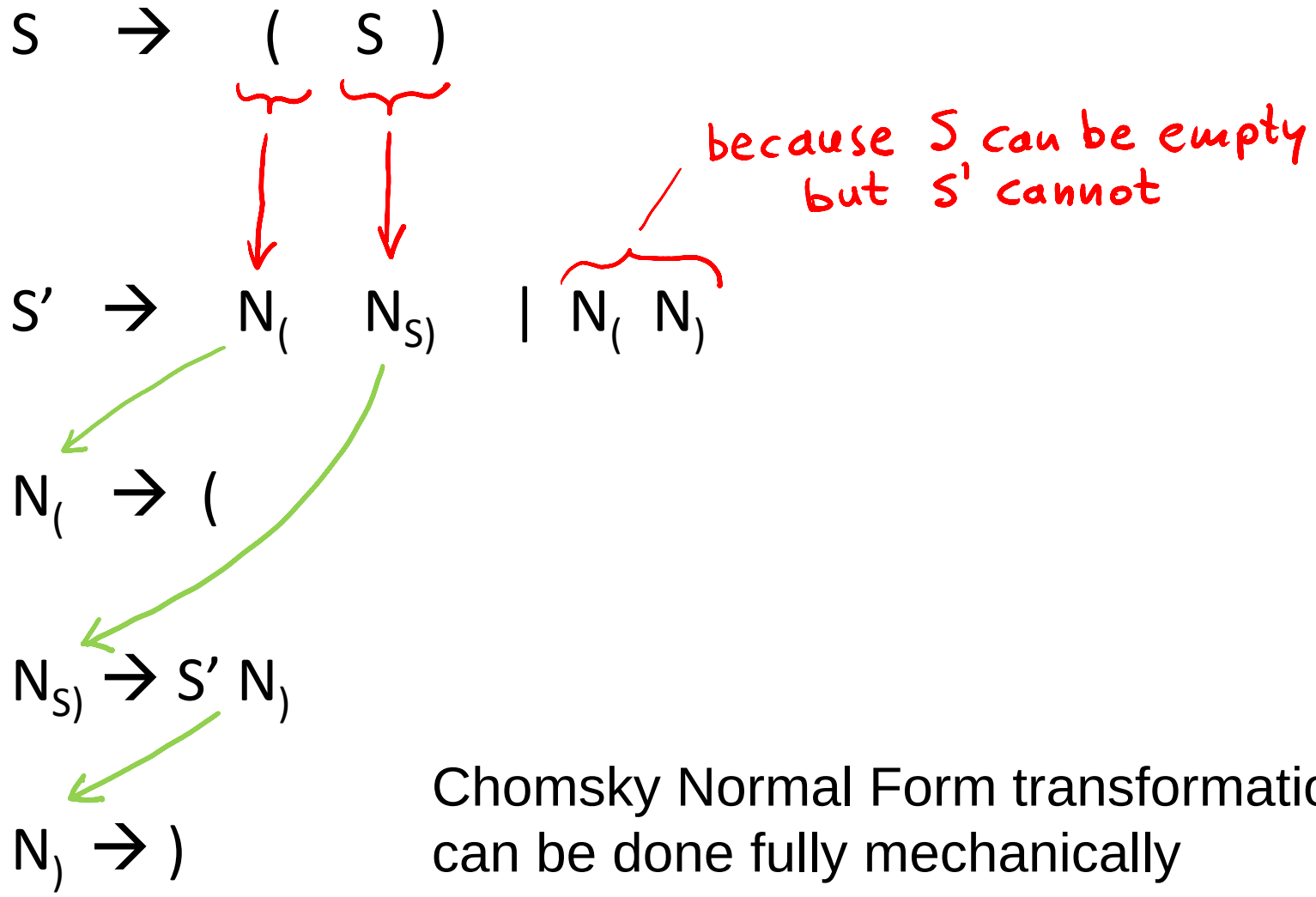
$N \rightarrow N_1 N_2$ (nonterminals)

$N \rightarrow t$ (nonterminal, terminal)

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

nonterminal with funny name

Idea How We Obtained the Grammar



Chomsky Normal Form transformation can be done fully mechanically

Transforming Grammars into Chomsky Normal 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 w/ more than 3 on RHS
6. make terminals occur alone on right-hand side

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
```

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 … s_n then add rule

  X ::= s_1 s_2 … s_n

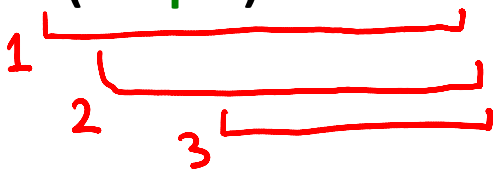
At the end, remove all single productions.

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



stmt ::= while $stmt_1$

$stmt_1$ ::= ($stmt_2$

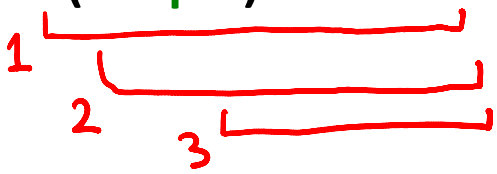
$stmt_2$ ::= expr $stmt_3$

$stmt_3$ ::=) stmt

6) A non-terminal for each terminal

stmt ::= while (expr) stmt

becomes



$stmt ::= N_{while}\ stmt_1$

$stmt_1 ::= N_{(}\ stmt_2$

$stmt_2 ::= expr\ stmt_3$

$stmt_3 ::= N_{)}\ stmt$

$N_{while} ::= while$

$N_{(} ::= ($

$N_{)} ::=)$

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

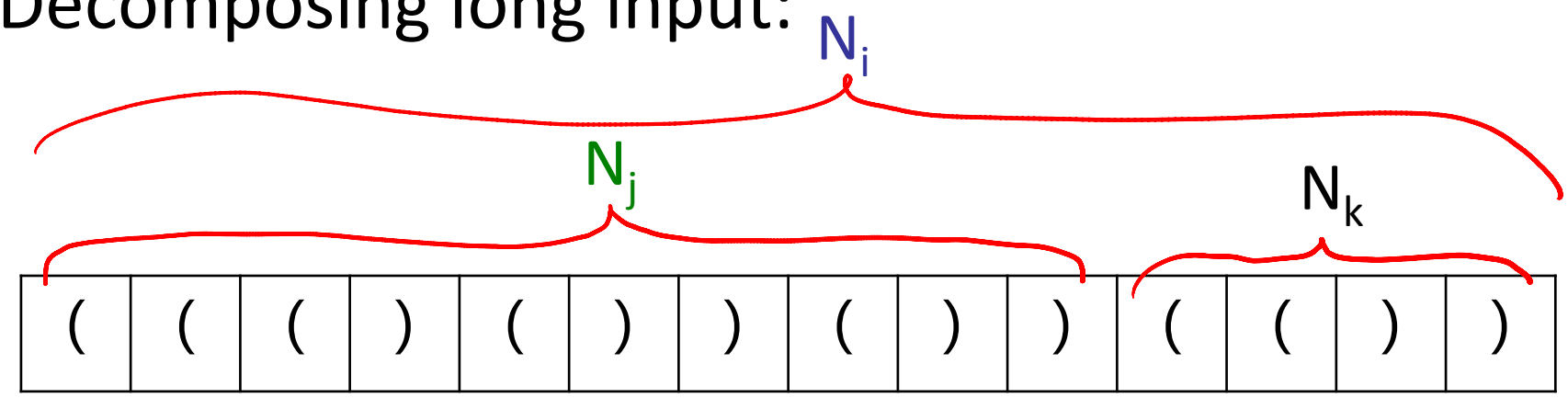
  Have only rules X ::= Y Z, X ::= t, and possibly S ::= “”
- Apply CYK dynamic programming algorithm

Dynamic Programming to Parse Input

Assume Chomsky Normal Form, 3 types of rules:

$S \rightarrow \varepsilon \mid S'$	(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

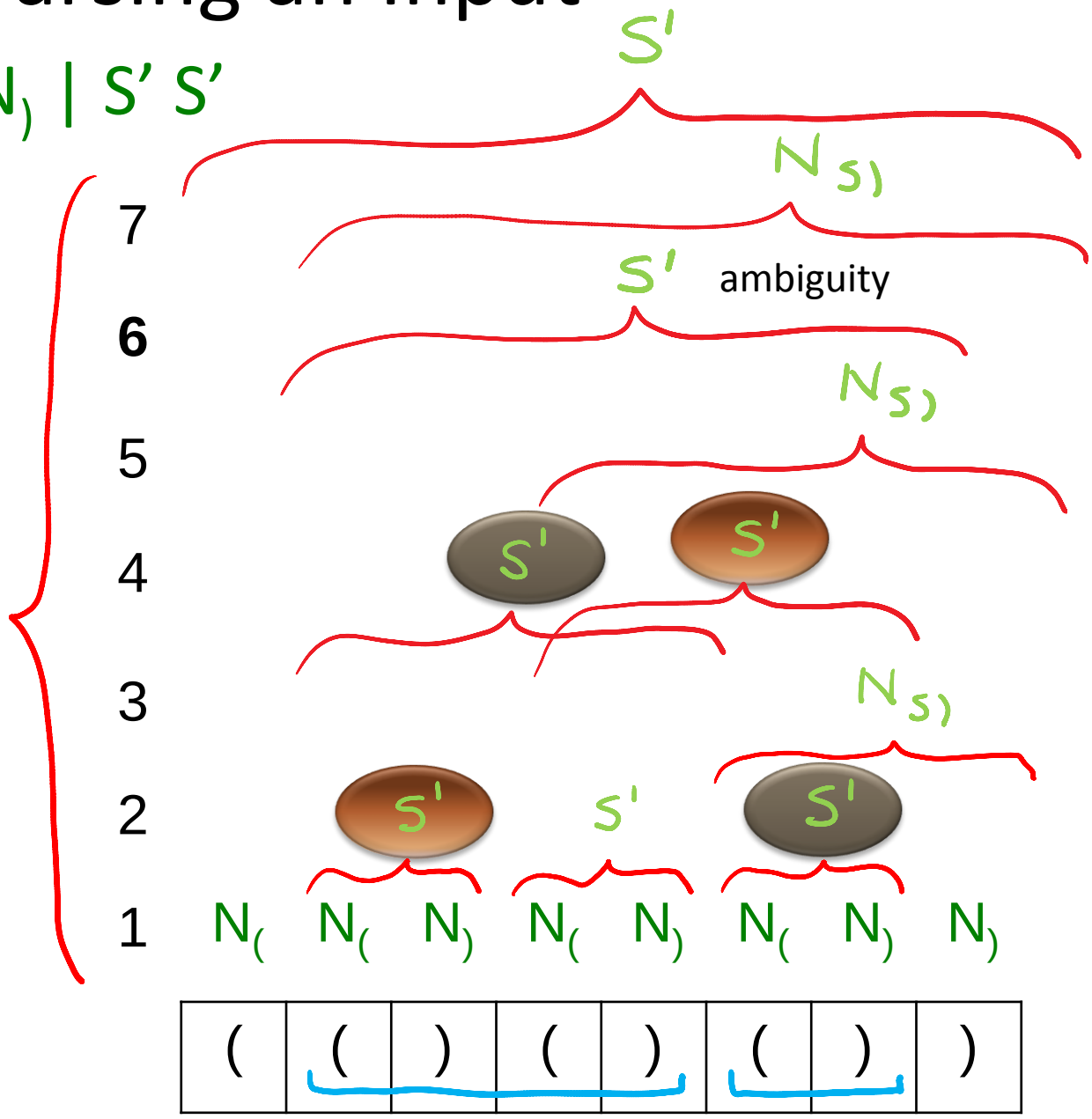
$S' \rightarrow N_(\; N_{S)} \mid N_(\; N_) \mid S' \; S'$

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

$N_(\rightarrow ($

$N_) \rightarrow)$

substring length



Algorithm Idea

S' → 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 → Y Z is a rule,

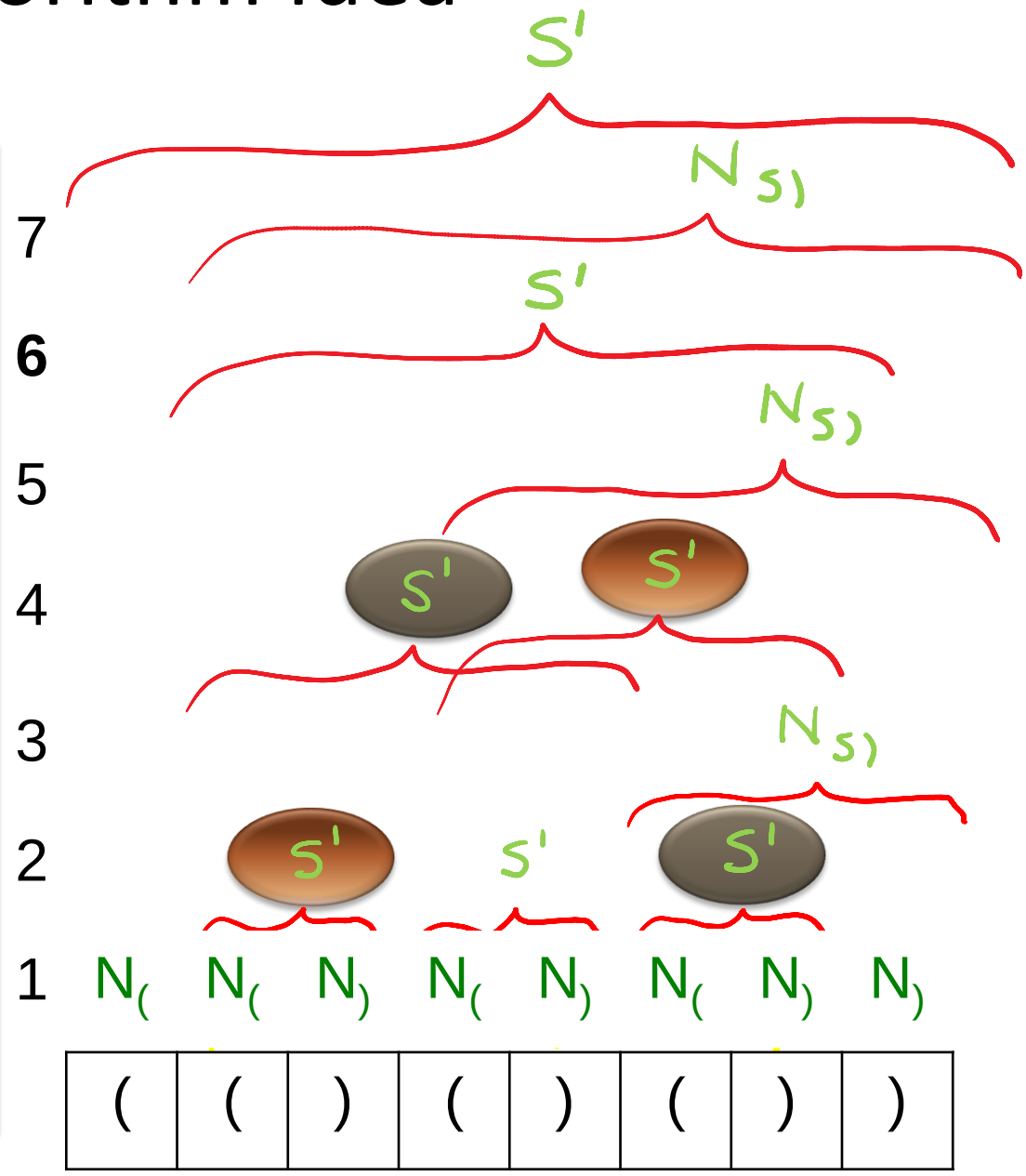
Y is in $d_{p\,r}$, and

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

then put X into d_{pq}

($p \leq r < q$),

in increasing value of (q-p)



Algorithm

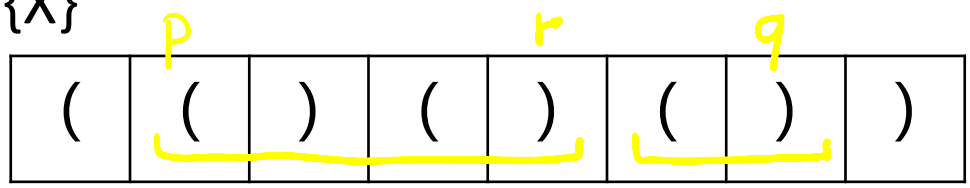
INPUT: grammar G in Chomsky normal form
word w to parse using G

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( )



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

7

6

5

4

3

2

1

$N_($	$N_)$	$N_($	$N_)$	$N_($	$N_)$	$N_($	$N_)$
(	)	(	)	(	)	(	)

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

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

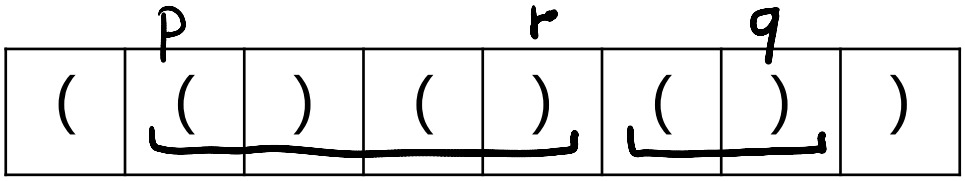
Earley's Algorithm
also parser arbitrary grammars

J. Earley, "An efficient context-free parsing algorithm", *Communications of the Association for Computing Machinery*, **13**:2:94-102, 1970.

CYK vs Earley's Parser Comparison

Z ::= X Y Z parses w_{pq}

- CYK: if d_{pr} parses X and $d_{(r+1)q}$ parses Y, then in d_{pq} stores symbol Z
- Earley's parser:
  in set S_q stores *item* (Z ::= XY. , p)
- Move forward, similar to top-down parsers
- Use **dotted rules** to avoid binary rules



Example: expressions

D ::= e EOF

e ::= ID | e – e | e == e

Rules with a dot inside

D ::= . e EOF | e . EOF | e EOF .

e ::= . ID | ID .

| . e – e | e . – e | e – . e | e – e .

| . e == e | e . == e | e == . e | e == e .

		ID	-	ID	==	ID	EOF
	ε	ID	ID-	ID-ID	ID-ID==	ID-ID==ID	
ID		ε	-	-ID	-ID==	-ID==ID	
-			ε	ID	ID==	ID==ID	
ID				ε	==	==ID	
==					ε	ID	
ID						ε	
EOF							ε

S ::= . e EOF | e . EOF | e EOF .

e ::= . ID | ID .
| . e – e | e . – e | e – . e | e – e .
| . e == e | e . == e | e == . e | e == e .

		ID s_1	- s_2	ID s_3	==	ID	EOF
	ε . e EOF . ID . e – e . e = e	ID ID . e . EOF e . – e e . = e	ID- e – . e	ID-ID e – e . e . EOF e . – e e . = e	ID-ID== e = . e	ID-ID==ID e = e . e – e .	e . EOF
ID		ε	-	-ID	-ID==	-ID==ID	
-			ε . ID . e – e . e = e	ID ID . e . – e e . = e	ID== e = . e	ID==ID e = e .	
ID				ε	==	==ID	
==					ε . ID . e – e . e = e	ID ID . e . – e e . = e	
ID						ε	
EOF							ε

S ::= . e EOF | e . EOF | e EOF .

e ::= . ID | ID .

| . e – e | e . – e | e – . e | e – e .

| . e == e | e . == e | e == . e | e == e .