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"

Balanced Parentheses Grammar

Original grammar G

$S \rightarrow$ “” | (S) | S S

Modified grammar in Chomsky Normal Form:

$S \rightarrow$ “” | S’

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

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

$N_(\rightarrow$ (

$N_) \rightarrow$)

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

Idea How We Obtained the Grammar

$S \rightarrow (\quad S \quad)$

Because S can be empty but S' cannot

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

$N_{(} \rightarrow ($

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

$N_{)} \rightarrow)$

Chomsky Normal Form transformation can be done fully mechanically

Dynamic Programming to Parse Input

Assume Chomsky Normal Form, 3 types of rules:

$S \rightarrow$ “” | 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: N_i

N_j N_k

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

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

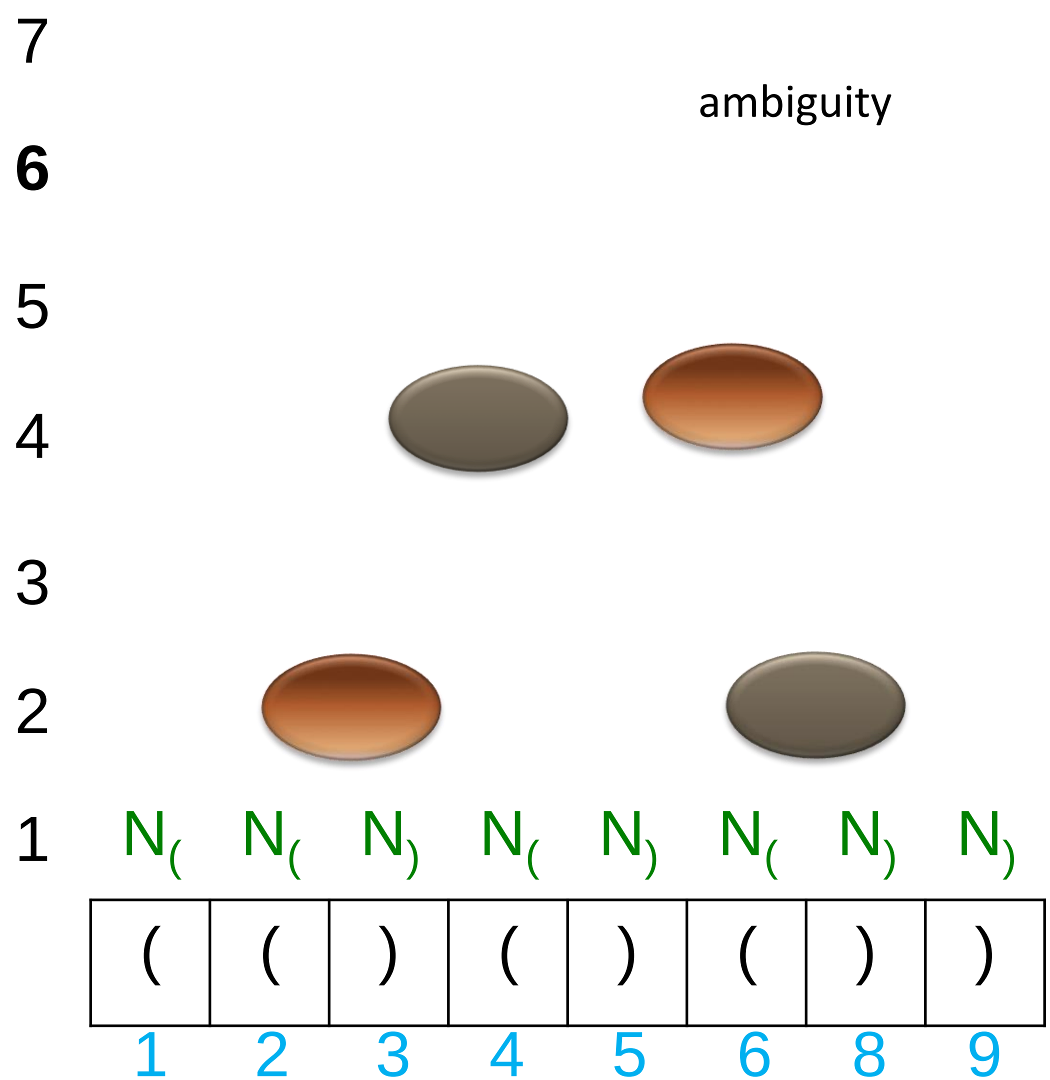
Parsing an Input

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

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

$N_{(} \rightarrow ($

$N_{)} \rightarrow)$



Algorithm Idea

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,

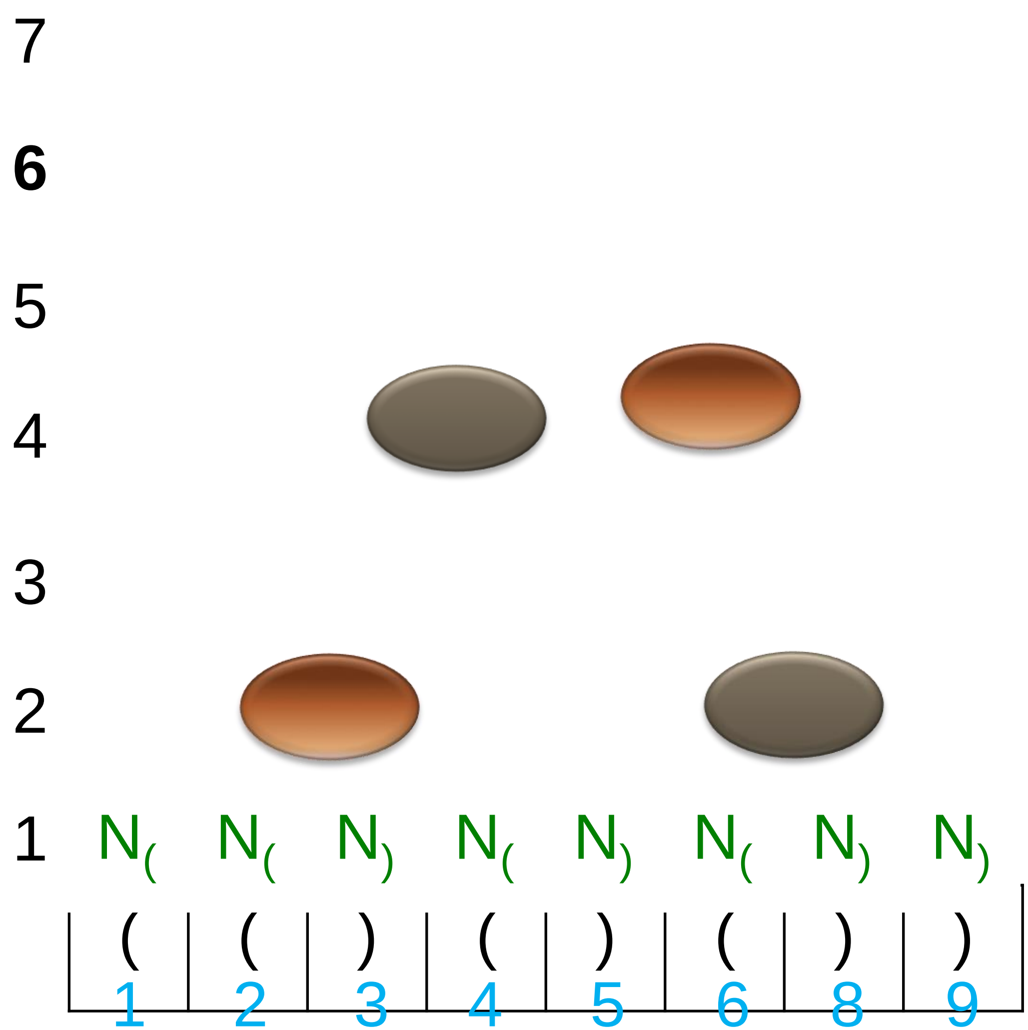
Y is in $d_{p\,r}$, and

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

then put X into d_{pq}

($p <= r < q$),

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

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

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

$N_(\rightarrow ($

$N_) \rightarrow)$

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

Conversion to Chomsky Normal Form (CNF)

- Steps:
  1. remove unproductive symbols
  2. remove unreachable symbols
  3. remove epsilons (no non-start nullable symbols)
  4. remove single non-terminal productions (unit Productions): X::=Y
  5. reduce arity of every production to less than two
  6. make terminals occur alone on right-hand side

1) Unproductive non-terminals

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

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 a 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.

The language recognized by the grammar will not change

2) Unreachable non-terminals

What is funny about this grammar with start symbol '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'

Can we formulate rules for reachable symbols ?

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 … s_n is rule and X is reachable then every non-terminal in s_1 s_2 … s_n is reachable
- Delete unreachable nonterminals and their productions

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
- If X::= $s_1 \cdots s_n$ is a production and s_i is nullable then add new rule
  - X::= $s_1 \cdots s_{i-1} s_{i+1} \cdots s_n \mid s_1 \cdots s_n$

- 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 unit 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 unit productions - Result

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) Unit Production Elimination Algorithm

- If there is a unit 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 unit 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

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

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

$stmt_2 ::= expr\ stmt_3$

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

$N_{while} ::= while$

$N_{(} ::= ($

$N_{)} ::=)$

Order of steps in conversion to CNF

1. remove unproductive symbols
2. remove unreachable symbols
3. Reduce arity of every production to <= 2
4. remove epsilons (no non-start nullable symbols)
5. remove unit productions X::=Y
6. make terminals occur alone on right-hand side

– What if we swap the steps 3 and 4 ?
  - Potentially exponential blow-up in the # of productions

– What if we swap the steps 4 and 5 ?
  - Epsilon removal can introduce unit productions

Parsing using CYK Algorithm

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