Chomsky’s Classification of Grammars

On Certain Formal Properties of Grammars

(N. Chomsky, INFORMATION AND CONTROL 9., 137-167 (1959)

type 0: arbitrary string-rewrite rules

equivalent to Turing machines!

e X b => e X   e X => Y

type 1: context sensitive, RHS always larger

O(n)-space Turing machines

a X b => a c X b

type 2: context free - one LHS nonterminal

type 3: regular grammars (regular languages)

Parsing Context-Free Grammars

Decidable even for type 1 grammars,
(by eliminating epsilons - Chomsky 1959)

We choose $O(n^3)$ CYK algorithm - simple

Better complexity possible:

General Context-Free Recognition in Less than Cubic Time, JOURNAL OF COMPUTER AND SYSTE M SCIENCES 10, 308--315 (1975)

- problem reduced to matrix multiplication - n^k for k between 2 and 3

More practical algorithms known:

J. Earley **An efficient context-free parsing algorithm,** Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA (1968)

can be adapted so that it automatically works in quadratic or linear time for better-behaved grammars

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.

CYK Algorithm Can Handle Ambiguity

Why Parse General Grammars

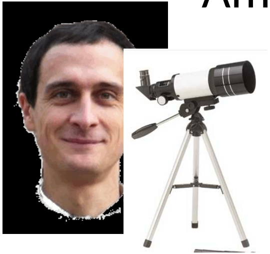
- General grammars can be ambiguous: for some strings, there are multiple parser trees
- Can be impossible to make grammar unambiguous
- Some languages are more complex than simple programming languages
  - mathematical formulas:

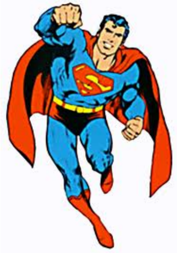
    $x = y \land z$? $(x=y) \land z$ $x = (y \land z)$
  - natural language:

    I saw the man with the telescope.
  - future programming languages

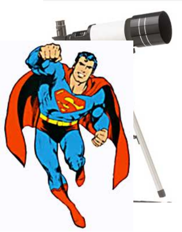
Ambiguity 1

1)





2)



I saw the man with the telescope.

Ambiguity 2

Time flies like an arrow.

Indeed, time passes by quickly.

Those special “time flies” have an “arrow” as their favorite food.

You should regularly measure how fast the flies are flying, using a process that is much like an arrow.

…

Two Steps in the Algorithm

1) Transform grammar to normal form
   called Chomsky Normal Form

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”

Dynamic Programming to Parse Input

Assume Chomsky Normal Form, 3 types of rules:

$S' \to \varepsilon \mid S$ (only for the start non-terminal)

$N_i \to t$ (names for terminals)

$N_i \to N_j\ N_k$ (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,...

Balanced Parentheses Grammar

Original grammar G

B → ε | B B | (B)

Modified grammar in Chomsky Normal Form:

B1 → ε | B B | O M | O C

B → B B | O M | O C

M → B C

O → '('

C → ')'

Terminals: ( )

Nonterminals: B, B1, O, C, M, B

Parsing an Input

B1 → ε | B B | O M | O C

B → B B | O M | O C

M → B C

O → '('

C → ')'

6								
5								
4								
3								
2								
1	O	O	C	O	C	O	C	C
	(	(	)	(	)	(	)	)
	1	2	3	4	5	6	8	9

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 $X \rightarrow Y Z$ is a rule,

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

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

then put X into d_{pq}

($p <= r < q$),

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(   )

(	(	)	(	)	(	)	)

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: (not in the optimal order)

- remove unproductive symbols
- remove unreachable symbols
- remove epsilons (no non-start nullable symbols)
- remove single non-terminal productions
  (unit productions)   X::=Y
- reduce arity of every production to less than two
- 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

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

$O(2^n)$

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) 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 … s_n then add rule

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

At the end, remove all unit productions.

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

5) Reducing Arity: 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

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

becomes

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

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

$stmt_2 ::= expr\ stmt_3$

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

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

$N_{(} ::= ($

$N_{)} ::=)$

Order of steps in conversion to CNF

1. remove unproductive symbols (optional)
2. remove unreachable symbols (optional)
3. make terminals occur alone on right-hand side
4. Reduce arity of every production to <= 2
5. remove epsilons
6. remove unit productions X::=Y
7. unproductive symbols
8. unreachable symbols

- What if we swap the steps 4 and 5 ?
  - Potentially exponential blow-up in the # of productions

Ordering of Unreachable / Unproductive symbols

First Unreachable then Unproductive

S := B C | “”	S := B C | “”	S := “”
C := D	C := D	C := D
D := a	D := a	D := a
R := r		

First Unproductive then Unreachable

S := B C | “”	S := “”	S := “”
C := D	C := D	
D := C	D := a	
R := r	R := r	

Alternative to Chomsky form

We need not go all the way to Chomsky form

it is possible to directly parse arbitrary grammar

Key steps: (not in the optimal order)

- reduce arity of every production to less than two (otherwise, worse than cubic in string input size)

Can be less efficient in grammar size, but still works

More algorithms for arbitrary grammars are variations:

Earley's parsing algorithm (Earley, CACM 1970)

GLR parsing algorithm (Lang, ICALP 1974, Deterministic Techniques for Efficient Non-Deterministic Parsers)

GLL algorithm