Parsing: Introduction

Context-free Grammars

- Chomsky hierarchy
  - Type 0 Grammars/Languages
    - rewrite rules $\alpha \rightarrow \beta$; α,β are any string of terminals and nonterminals
  - Context-sensitive Grammars/Languages
    - rewrite rules: $\alpha X\beta \rightarrow \alpha\gamma\beta$, where X is nonterminal, α,β,γ any string of terminals and nonterminals (γ must not be empty)
  - **Context-free Grammars/Lanuages**
    - rewrite rules: $X \rightarrow \gamma$, where X is nonterminal, γ any string of terminals and nonterminals
  - Regular Grammars/Languages
    - rewrite rules: $X \rightarrow \alpha\ Y$ where X,Y are nonterminals, α string of terminal symbols; Y might be missing

Parsing Regular Grammars

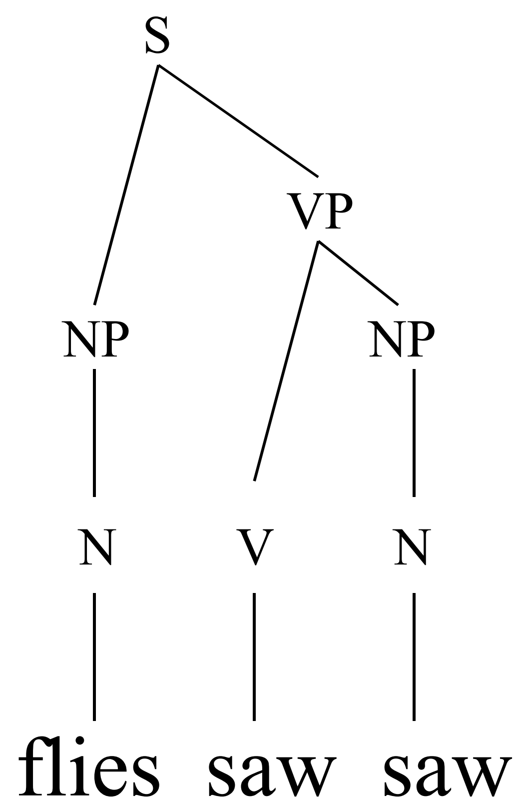
- Finite state automata
  - Grammar ↔ regular expression ↔ finite state automaton
- Space needed:
  - constant
- Time needed to parse:
  - linear (~ length of input string)
- Cannot do e.g. $a^n b^n$, embedded recursion (context-free grammars can)

Parsing Context Free Grammars

- Widely used for surface syntax description (or better to say, for correct word-order specification) of natural languages
- Space needed:
  - stack (sometimes stack of stacks)
    - in general: items ~ levels of actual (i.e. in data) recursions
- Time: in general, $O(n^3)$
- Cannot do: e.g. $a^n b^n c^n$ (Context-sensitive grammars can)

Example Toy NL Grammar

- #1 S → NP
- #2 S → NP VP
- #3 VP → V NP
- #4 NP → N
- #5 N → flies
- #6 N → saw
- #7 V → flies
- #8 V → saw



Shift-Reduce Parsing in Detail

Grammar Requirements

- Context Free Grammar with
  - no empty rules ($N \rightarrow \varepsilon$)
    - can always be made from a general CFG, except there might remain one rule $S \rightarrow \varepsilon$ (easy to handle separately)
  - recursion OK
- Idea:
  - go bottom-up (otherwise: problems with recursion)
  - construct a Push-down Automaton (non-deterministic in general, PNA)
  - delay rule acceptance until all of a (possible) rule parsed

PNA Construction - Elementary Procedures

- Initialize-Rule-In-State(q,A $\rightarrow$ α) procedure:
  - Add the rule (A $\rightarrow$ α) into a state q.
  - Insert a dot in front of the R[ight]H[and]S[ide]: A $\rightarrow$. α
- Initialize-Nonterminal-In-State(q,A) procedure:
  - Do "Initialize-Rule-In-State(q,A $\rightarrow$ α)" for all rules having the nonterminal A on the L[eft]H[and]S[ide]
- Move-Dot-In-Rule(q,A $\rightarrow$ α . Zβ) procedure:
  - Create a new rule in state q: A $\rightarrow$ αZ . β, Z term. or not

PNA Construction

- Put 0 into the (FIFO/LIFO) list of incomplete states, and do Initialize-Nonterminal-In-State(0,S)
- Until the list of incomplete states is not empty, do:
  1. Get one state, i from the list of incomplete states.
  2. Expand the state:
     - Do recursively Initialize-Nonterminal-In-State(i,A) for all nonterminals A right <u>after</u> the dot in any of the rules in state i.
  3. If the state matches exactly some other state already in the list of complete states, renumber all shift-references to it to the old state and discard the current state.

PNA Construction (Cont.)

4. Create a set T of Shift-References (or, transition/continuation links) for the current state i {(Z,x)}:
   - Suppose the highest number of a state in the incomplete state list is n.
   - For each symbol Z (regardless if terminal or nonterminal) which appears after the dot in any rule in the current state q, do:
     - increase n to n+1
     - add (Z,n) to T
       - *NB: each symbol gets only one Shift-Reference, regardless of how many times (i.e. in how many rules) it appears to the right of a dot.*
     - Add n to the list of incomplete states
     - Do Move-Dot-In-Rule(n,$A \rightarrow \alpha$. $Z\beta$)
5. Create Reduce-References for each rule in the current state i:
   - For each rule of the form ($A \rightarrow \alpha$.) (i.e. dot at the end) in the current state, attach to it the rule number r̲ of the rule $A \rightarrow \alpha$ from the grammar.

Using the PNA (Initialize)

- Maintain two stacks, the input stack I and the state stack Q.
- Maintain a stack B[acktracking] of the two stacks.
- Initialize the I stack to the input string (of terminal symbols), so that the first symbol is on top of it.
- Initialize the stack Q to contain state 0.
- Initialize the stack B to empty.

Using the PNA (Parse)

- Do until you are not stuck and/or B is empty:
  - Take the top of stack Q state ("current" state i).
  - Put all possible reductions in state i on stack B, including the contents of the current stacks I and Q.
  - Get the symbol from the top of the stack I (symbol Z).
  - If (Z,x) exists in the set T associated with the current state i, push state x onto the stack Q and remove Z from I. Continue from beginning.
  - Else pop the first possibility from B, remove n symbols from the stack Q, and push A to I, where $A \rightarrow Z_1...Z_n$ is the rule according which you are reducing.

Small Example

Grammar:

#1 S → NP VP
#2 NP → N
#3 VP → V NP
#4 N → a_cat
#5 N → a_dog
#6 V → saw

no ambiguity, no recursion

Tables: <symbol> <state>: shift
#<rule>: reduction

0 S → . NP VP	NP 1
NP → . N	N 2
N → . a_cat	a_cat 3
N → . a_dog	a_dog 4

1 S → NP . VP	VP 5
VP → . V NP	V 6
V → . saw	saw 7
2 NP → N .	#2
3 N → a_cat .	#4
4 N → a_dog .	#5
5 S → NP VP .	#1
6 VP → V . NP	NP 8
NP → . N	N 2
N → . a_cat	a_cat 3
N → . a_dog	a_dog 4
7 V → saw .	#6
8 VP → V NP .	#3

NB: dotted rules in states need not be kept

Small Example: Parsing(1)

- To parse: **a_dog saw a_cat**

Input stack (top on the left)	Rule	State stack (top on the left)	Comment(s)
• a_dog saw a_cat		0	
• saw a_cat		4 0	shift to 4 over a_dog
• N saw a_cat	#5	0	reduce #5: N → a_dog
• saw a_cat		2 0	shift to 2 over N
• NP saw a_cat	#2	0	reduce #2: NP → N
• saw a_cat		1 0	shift to 1 over NP
• a_cat		7 1 0	shift to 7 over saw
• V a_cat	#6	1 0	reduce #6: V → saw

Small Example: Parsing (2)

- ...still parsing: **a_dog saw a_cat**

• [V a_cat	#6	1 0] ←	Previous parser configuration
• a_cat		6 1 0	shift to 6 over V
•		3 6 1 0	empty input stack (not finished though!)
• N	#4	6 1 0	N inserted back
•		2 6 1 0	...again empty input stack
• NP	#2	6 1 0	
•		8 6 1 0	...and again
• VP	#3	1 0	two states removed (|RHS(#3)|=2)
•		5 1 0	
• S	#1	0	again, two items removed (RHS: NP VP)

Success: S/0 alone in input/state stack; reverse right derivation: 1,3,2,4,6,2,5

Big Example: Ambiguous and Recursive Grammar

- #1 S $\rightarrow$ NP VP
- #2 NP $\rightarrow$ NP REL VP
- #3 NP $\rightarrow$ N
- #4 NP $\rightarrow$ N PP
- #5 VP $\rightarrow$ V NP
- #6 VP $\rightarrow$ V NP PP
- #7 VP $\rightarrow$ V PP
- #8 PP $\rightarrow$ PREP NP

#9 N $\rightarrow$ a_cat
#10 N $\rightarrow$ a_dog
#11 N $\rightarrow$ a_hat
#12 PREP $\rightarrow$ in
#13 REL $\rightarrow$ that
#14 V $\rightarrow$ saw
#15 V $\rightarrow$ heard

Big Example: Tables (1)

0 S → . NP VP	NP	1
NP → . NP REL VP		
NP → . N	N	2
NP → . N PP		
N → . a_cat	a_cat	3
N → . a_dog	a_dog	4
N → . a_mirror	a_hat	5

1 S → NP . VP	VP	6
NP → NP . REL VP	REL	7
VP → . V NP	V	8
VP → . V NP PP		
VP → . V PP		
REL → . that	that	9
V → . saw	saw	10
V → . heard	heard	11

2 NP → N .	#3	
NP → N . PP	PP	12
PP → . PREP NP	PREP	13
PREP → . in	in	14

3 N → a_cat .	#9

4 N → a_dog .	#10

5 N → a_hat .	#11

6 S → NP VP .	#1

Big Example: Tables (2)

7 NP → NP REL . VP	VP	15
VP → . V NP	V	8
VP → . V NP PP		
VP → . V PP		
V → . saw	saw	10
V → . heard	heard	11

8 VP → V . NP	NP	16
VP → V . NP PP		
VP → V . PP	PP	17
NP → . NP REL VP		
NP → . N	N	2
NP → . N PP		
N → . a_cat	a_cat	3
N → . a_dog	a_dog	4
N → . a_hat	a_hat	5
PP → . PREP NP	PREP	13
PREP → . in	in	14

9 REL → that .	#13

10 V → saw .	#14

11 V → heard .	#15

12 NP → NP PP .	#4

13 PP → PREP . NP	NP	18
NP → . NP REL VP		
NP → . N	N	2
NP → . N PP		
N → . a_cat	a_cat	3
N → . a_dog	a_dog	4
N → . a_hat	a_hat	5

Big Example: Tables (3)

14 PREP → in .	#12	

15 NP → NP REL VP .	#2	

16 VP → V NP .	#5	
VP → V NP . PP	PP	19
NP → NP . REL VP	REL	7
PP → . PREP NP	PREP	13
PREP → . in	in	14
REL → . that	that	9

17 VP → V PP .	#7	

18 PP → PREP NP .	#8	
NP → NP . REL VP	REL	7
REL → . that	that 9	

19 VP → V NP PP .	#6	

Comments:

- states 2, 16, 18 have shift-reduce conflict
- no states with reduce-reduce conflict
- also, again there is no need to store the dotted rules in the states for parsing. Simply store the pair input/goto-state, or the rule number.

Big Example: Parsing (1)

- To parse: **a_dog heard a_cat in a_hat**

Input stack (top on the left) State stack (top on the left)

Input stack	Rule	State stack	Backtrack	Comment(s)
• a_dog heard a_cat in a_hat		0		shifted to 4 over a_dog
• heard a_cat in a_hat		4 0		shift to 4 over a_dog
• N heard a_cat in a_hat	#10	0		reduce #10: N $\rightarrow$ a_dog
• heard a_cat in a_hat		2 0		shift to 2 over N[1]
• NP heard a_cat in a_hat	#3	0		reduce #3: NP $\rightarrow$ N
• heard a_cat in a_hat		1 0		shift to 1 over NP
• a_cat in a_hat		11 1 0		shift to 11 over heard
• V a_cat in a_hat	#15	1 0		reduce #15: V $\rightarrow$ heard
• a_cat in a_hat		8 1 0		shift to 8 over V

[1] see also next slide, last comment

Big Example: Parsing (2)

- ...still parsing: **a_dog heard a_cat in a_hat**

Input stack (top on the left)	Rule	State stack (top on the left)	Backtrack	Comment(s)
[a_cat in a_hat		8 1 0] ← [previous parser configuration]		
in a_hat		3 8 1 0		shift to 3 over a_cat
N in a_hat	#9	8 1 0		reduce #9: N → a_cat
in a_hat		2 8 1 0	⊗	shift to 2 over N; see why we need the state stack? we are in 2 again, but after we return, we will be in 8 not 0; also save for backtrack[1]!

[1] the whole input stack, state stack, and [reversed] list of rules used for reductions so far must be saved on the backtrack stack

Big Example: Parsing (3)

- ...still parsing: **a_dog heard a_cat in a_hat**

Input stack (top on the left) State stack (top on the left)

Input stack	Rule	State stack	Backtrack	Comment(s)
[in a_hat		2 8 1 0	⊗] ← [previous parser configuration]	
a_hat		14 2 8 1 0		shift to 14 over in
PREP a_hat	#12	2 8 1 0		reduce #12: PREP → in[1]
a_hat		13 2 8 1 0		shift to 13 over PREP
		5 13 2 8 1 0		shift to 5 over a_hat
N	#11	13 2 8 1 0		reduce #11: N → a_hat
		2 13 2 8 1 0		shift to 2 over N
NP	#3	13 2 8 1 0		shift not possible; reduce #3: NP → N[1 on s.19]
		18 13 2 8 1 0		shift to 18 over NP

[1] when coming back to an ambiguous state [here: state 2] (after some reduction), reduction(s) are not considered; nothing put on backtrk stack

Big Example: Parsing (4)

- ...still parsing: **a_dog heard a_cat in a_hat**

Input stack (top on the left)	Rule	State stack (top on the left)	Backtrack	Comment(s)
[		18 13 2 8 1 0]		← [previous parser config.]
PP	#8	2 8 1 0		shift not possible; reduce #8[1 on s.19]: PP → PREP NP[1,prev.slide]
		12 2 8 1 0		shift to 12 over PP
NP	#4	8 1 0		reduce #4: NP → N PP
		16 8 1 0		shift to 16 over NP
VP	#5	1 0		shift not possible, reduce #5[1]: VP → V NP

[1] no need to keep the item on the backtrack stack; no shift possible now and there is only one reduction (#5) in state 16

Big Example: Parsing (5)

- ...still parsing: **a_dog heard a_cat in a_hat**

Input stack (top on the left)	Rule	State stack (top on the left)	Backtrack	Comment(s)
[VP	#5	1 0] ← [previous parser configuration]		
		6 1 0		shift to 6 over VP
S	#1	0		reduce #1: S → NP VP
				first solution found:
				1,5,4,8,3,11,12,9,15,3,10
				backtrack to previous ⊗ :
in a_hat		2 8 1 0		was: shift over in, now[1]:
NP in a_hat	#3	8 1 0		reduce #3: NP → N
in a_hat		16 8 1 0 ⊗		shift to 16 over NP
a_hat		14 16 8 1 0		shift, but put on backtrk

[1] no need to keep the item on the backtrack stack; no shift possible now and there is only one reduction (#3) in state 2

Big Example: Parsing (6)

- ...still parsing: **a_dog heard a_cat in a_hat**

Input stack (top on the left) State stack (top on the left)

Input stack	Rule	State stack / Backtrack	Comment(s)
[a_hat		14 16 8 1 0 ⊗] ← [previous parser config.]	
PREP a_hat	#12	16 8 1 0	reduce #12: PREP → in
a_hat		13 16 8 1 0	shift over PREP[1 on s.17]
		5 13 16 8 1 0	shift over a_hat to 5
N	#11	13 16 8 1 0	reduce #11: N → a_hat
		2 13 16 1 0	shift to 2 over N
NP	#3	13 16 1 0	shift not possible[1 on s.19]
		18 13 16 1 0	shift to 18
PP	#8	16 1 0	shift not possible[1], red.#8
		19 16 1 0	shift to 19[1 on s.17]

[1] no need to keep the item on the backtrack stack; no shift possible now and there is only one reduction (#8) in state 18

Big Example: Parsing (7)

- ...still parsing: **a_dog heard a_cat in a_hat**

Input stack (top on the left)	Rule	State stack (top on the left)	Backtrack	Comment(s)
[		19 16 8 1 0] ← [previous parser config.]		
VP	#6	1 0		red. #6: VP → V NP PP
		6 1 0		shift to 6 over VP
S	#1	0		next (2nd) solution: 1,6,8,3,11,12,3,[1]9,15,3,10
				backtrack to previous ⊗ :
in a_hat		16 8 1 0		was: shift over in[1 on s.19],
VP in a_hat	#5	1 0		now red. #5: VP → V NP
in a_hat		6 1 0		shift to 6 over VP
S in a_hat	#1	0		error[2]; backtrack empty: <u>stop</u>

[1] continue list of rules at the orig. backtrack mark (s.16,line 3) [2] S (the start symbol) not alone in input stack when state stack = (0)