Lecture 15

Top-Down Parsing

CS 241: Foundations of Sequential Programs
Fall 2014

Troy Vasiga et al
University of Waterloo

Parsing

Given a grammar G and a word w, find a derivation for w.
Two strategies:

1. Top-down: find a non-terminal and replace it with a right-hand side of a rule.

Natural: humans do this

$S \overset{3}{\Rightarrow} aAbX$

2. Bottom-up: replace a right-hand side with a non-terminal.

$AbYT \Rightarrow AB$

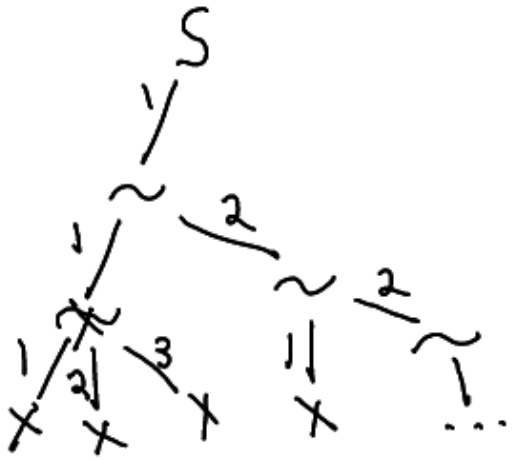
machines are good at this.

In both of the above strategies, we have to make the correct decision at each step.

which rule to use?

Parsing Algorithm

- There is a backtracking algorithm for parsing in any CFG
  - try each rule in turn
  - if we can move "forward", do so
  - if we cannot move "forward", go back a step and try the "next" rule
  - stop when we find the derivation



- Backtracking is not practical.

- exhaustive search
- exponential time

- We will look at two (linear-time) algorithms.

Stack-based Parsing

- if (push
- if) pop
- never pop empty
- make sure empty at end

e.g. (()())

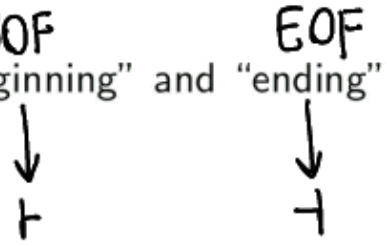
For top-down parsing, we use a stack to remember information about our derivations and/or processed input.

Recall: Context free languages are recognized by a finite control & one stack.

Augmenting Grammars

Empty words and empty stacks can cause hassles.

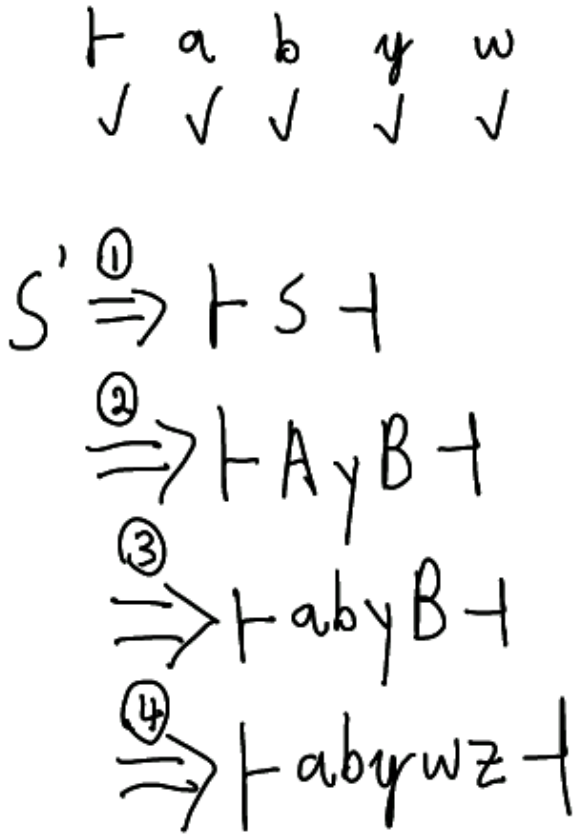
We augment our grammars by adding "beginning" and "ending" characters.



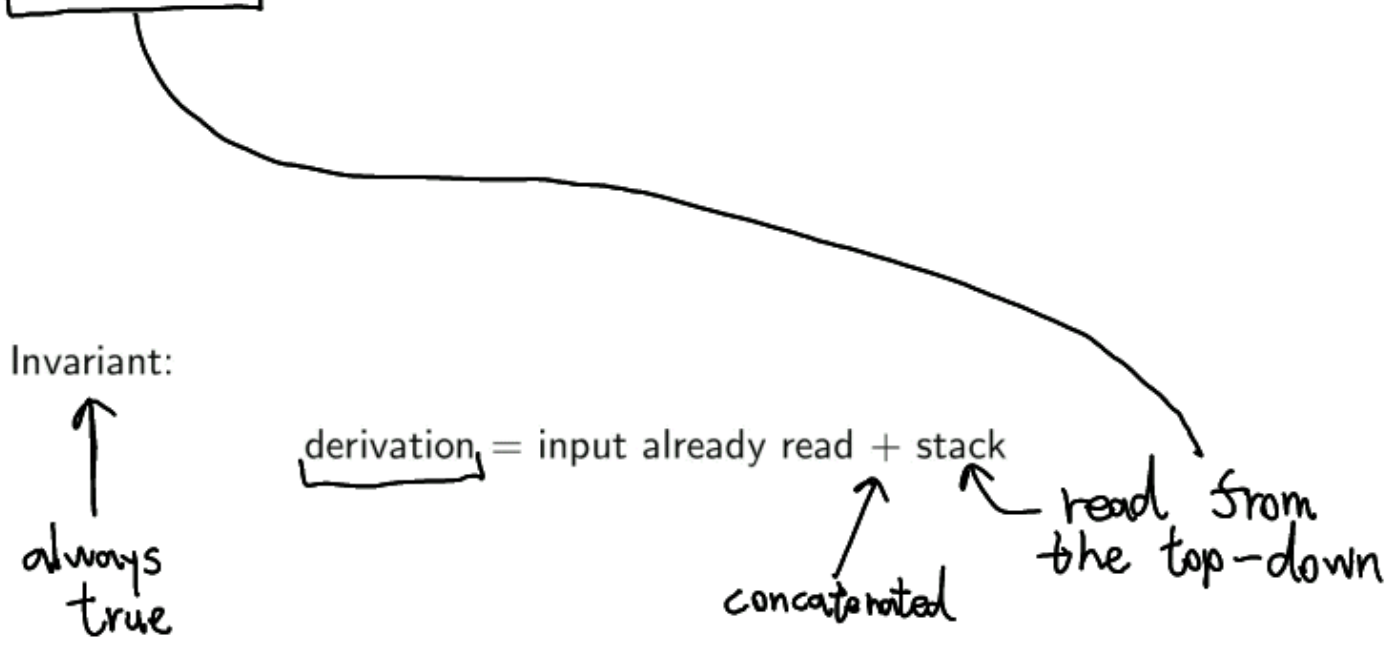
Example:

1. $S' \to \vdash S \dashv$
2. $S \to AyB$
3. $A \to ab$
4. $A \to cd$
5. $B \to z$
6. $B \to wz$

A simple parse



Top-down parsing with a stack



expand: pop LHS and push RHS in reverse.

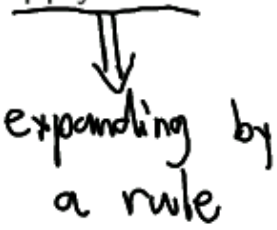
Stack Example

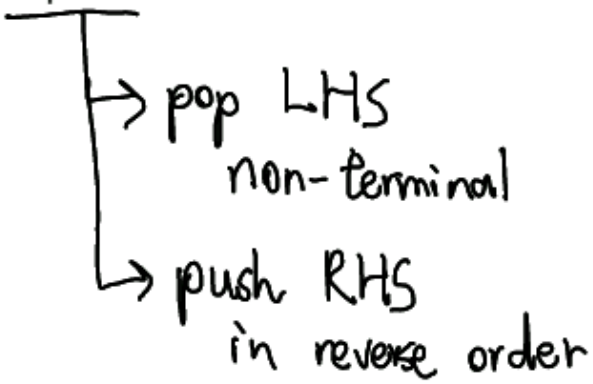
Matched Input ↓ | top ↓

Derivation	Input read	Input to be read	Stack	Actions
S'		⊢a b y w z⊣	S'	expand ①
⊢ S ⊣		⊢a b y w z⊣	⊢ S ⊣	match
⊢ S ⊣	⊢	a b y w z⊣	S ⊣	expand ②
⊢ AyB ⊣	⊢	a b y w z⊣	AyB ⊣	expand ③
⊢abyB⊣	⊢	a b y w z⊣	abyB⊣	match
⊢abyB⊣	⊢a	b y w z⊣	byB⊣	match
⊢abyB⊣	⊢ab	y w z⊣	yB⊣	match
⊢abyB⊣	⊢aby	w z⊣	B⊣	expand ⑥
⊢abywz⊣	⊢aby	w z⊣	wz⊣	match
⊢abywz⊣	⊢abyw	z⊣	z⊣	match
⊢abywz⊣	⊢abywz	⊣	⊣	ACCEPT

Observations:

How do we apply these rules? What does "expand" mean?

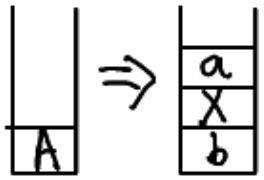




How do we know we are done?

Stack contains EOF
and input contains EOF

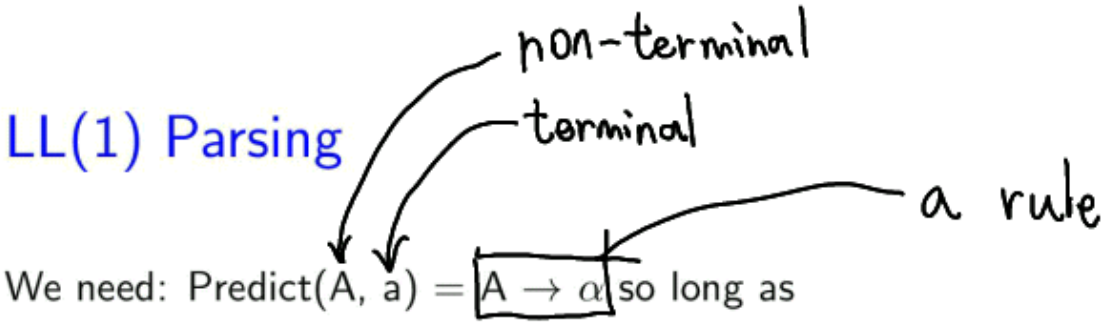
$A \rightarrow aXb$



How to know which rule to use?

predict correctly

LL(1) Parsing



We need: Predict(A, a) = $A \to \alpha$ so long as

- A is on top of the stack, and
- a is the first symbol of input to be read

Definition of an LL(1) grammar:

$$\forall A \in N, \forall a \in T, |Predict(A, a)| \leq 1$$

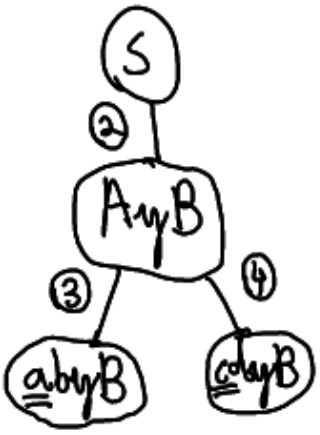
Meaning of:

- L Left-to-right input
- L Leftmost derivation
- 1 1-token of "look ahead" (input)

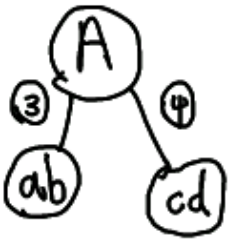
Constructing a Predictor Table

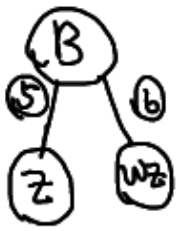
CFG:

1. S' $\rightarrow \vdash$ S $\dashv$
2. S $\rightarrow$ AyB
3. A $\rightarrow$ ab
4. A $\rightarrow$ cd
5. B $\rightarrow$ z
6. B $\rightarrow$ wz



	a	b	c	d	y	w	z	⊢	⊣
S'	E	E	E	E	E	E	E	1	E
S	2	E	2	E	E	E	E	E	E
A	3	E	4	E	E	E	E	E	E
B	E	E	E	E	E	6	5	E	E

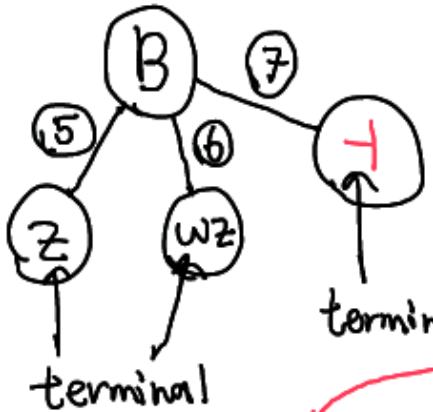




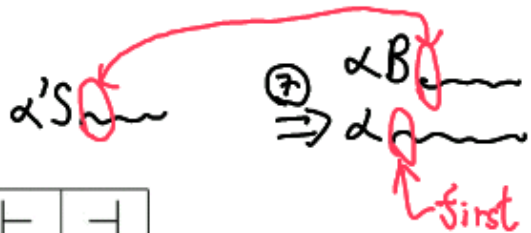
Constructing a Predictor Table (with ϵ)

CFG:

1. S' $\rightarrow \vdash$ S $\dashv$
2. S $\rightarrow$ AyB
3. A $\rightarrow$ ab
4. A $\rightarrow$ cd
5. B $\rightarrow$ z
6. B $\rightarrow$ wz
7. B $\rightarrow \epsilon$



W(hat's)
T(he)
F(ollow?)



·	a	b	c	d	y	w	z	$\vdash$	$\dashv$
S'								1	
S	2		2						
A	3		4						
B						6	5		7

Same as before

∴ Follow(B) = Follow(S)
Follow(S) = ⊣

Algorithm to construct predicator table

Below, $\alpha, \beta \in (N \cup T)^*$, $a, b \in T$, $A \in N$

Empty(α) = true if $\alpha \Rightarrow^* \epsilon$

Can α disappear?

Implement: if $x \in T$ in $\alpha \Rightarrow$ false
if no terminals:
search tree on each non-terminal.

First(α) = $\{a \mid \alpha \Rightarrow^* a\beta\}$

starting from α, what can I generate as a first terminal

Follow(A) = $\{b \mid S' \Rightarrow^* \alpha A b \beta\}$

Starting from the start symbol, does the terminal b ever immediately follow A? Look at RHS

Predict(A, a) = $\{A \to \alpha \mid a \in \text{First}(\alpha)\}$
$\cup \{A \to \beta \mid a \in \text{Follows}(A) \text{ and Empty}(\beta) = \text{true}\}$

LL(1) Parsing algorithm

Input: w
push $\vdash S \dashv$ ← in reverse
for each $a \in w$
    while (top of stack is some $A \in N$) {
        pop A
        if Predict(A, a) $= \{A \to \alpha\}$
            push α ← in reverse
        else
            reject ← no rule found
    }
    pop c ← terminal
    if c $\neq$ a reject ← no-match
end for
accept w

→ try to expand

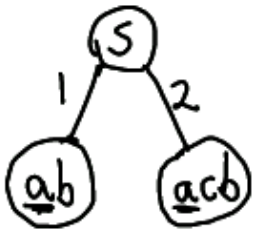
→ try to match

$O(n)$ ← $n = |G| + |w|$

Non LL(1) Grammars

G: 1. $S \rightarrow a\ b$

2. $S \rightarrow a\ c\ b$



$|L(G)| = 2$

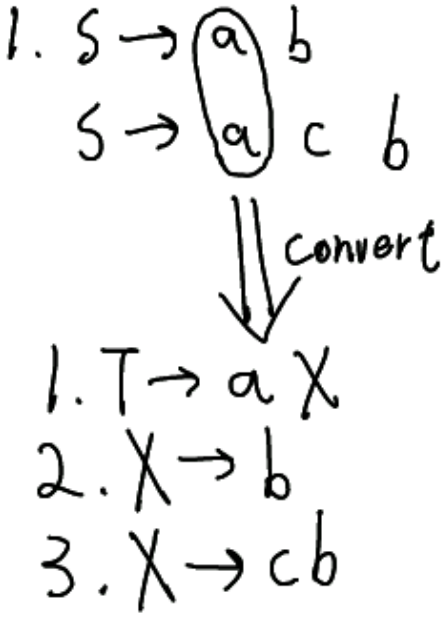
Not LL(1):

	a	b	c
S	1,2		

It is LL(2):

	aa	ab	ac	ba	bb	bc	ca	cb	cc
S		1	2						

Converting non-LL(1) grammars to LL(1) grammars



Factoring

	a	b	c
T	1		
X		2	3

A non LL(1) language

ε,
ab
aab, aaaaa

$L = \{a^n b^m | n \geq m \geq 0\}$

not LL(k) for any K

Grammar (ambiguous)

1. $S \rightarrow \varepsilon$
2. $S \rightarrow aS$
3. $S \rightarrow aSb$

$S \overset{2}{\Rightarrow} aS$
ab

Grammar (unambiguous)

$$\begin{bmatrix} S \rightarrow A & A \rightarrow \varepsilon & B \rightarrow aA \\ S \rightarrow B & A \rightarrow aAb & B \rightarrow aB \end{bmatrix}$$