Lecture 16+17

Bottom-Up Parsing

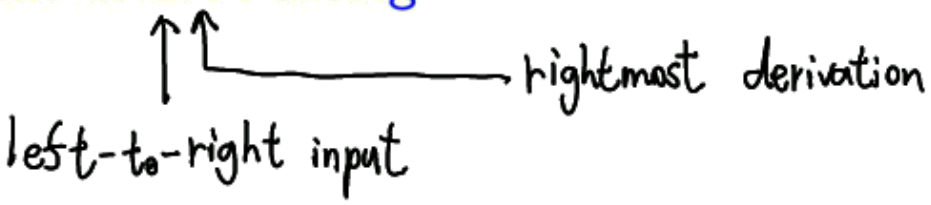
CS 241: Foundations of Sequential Programs
Fall 2014

Troy Vasiga et al
University of Waterloo

Example 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

Stacks in LR Parsing



- Recall that a stack in LL/top-down parsing is used in the following way:

  input processed + stack = current derivation

  (Note that the stack here is read from the top to bottom)
- For LR/bottom-up parsing, we have

  stack+input to be read = current derivation

  (stack is read from bottom to top here)

A trace

top of the stack

Derivation	Stack	Input read	Unread Input	Action
$\vdash$ abywz $\dashv$	$\vdash$	$\vdash$	abywz $\dashv$	initialize (Shift $\vdash$)
$\vdash$ abywz $\dashv$	$\vdash$ a	$\vdash$ a	bywz $\dashv$	Shift a
$\vdash$ abywz $\dashv$	$\vdash$ a b	$\vdash$ ab	ywz $\dashv$	Shift b
$\vdash$ Aywz $\dashv$	$\vdash$ A	$\vdash$ ab	ywz $\dashv$	Reduce A $\rightarrow$ ab
$\vdash$ Aywz $\dashv$	$\vdash$ A y	$\vdash$ aby	wz $\dashv$	Shift y
$\vdash$ Aywz $\dashv$	$\vdash$ A y w	$\vdash$ abyw	z $\dashv$	Shift w
$\vdash$ Aywz $\dashv$	$\vdash$ A y w z	$\vdash$ abywz	$\dashv$	Shift z
$\vdash$ AyB $\dashv$	$\vdash$ A y B	$\vdash$ abywz	$\dashv$	Reduce B $\rightarrow$ w z
$\vdash$ S $\dashv$	$\vdash$ S	$\vdash$ abywz	$\dashv$	Reduce S $\rightarrow$ AyB
$\vdash$ S $\dashv$	$\vdash$ S $\dashv$	$\vdash$ abywz $\dashv$	ϵ	Shift $\dashv$

Shift: shifting a token from one place to another (push)

Reduce: size of the stack may be reduced (pop RHS, push LHS)

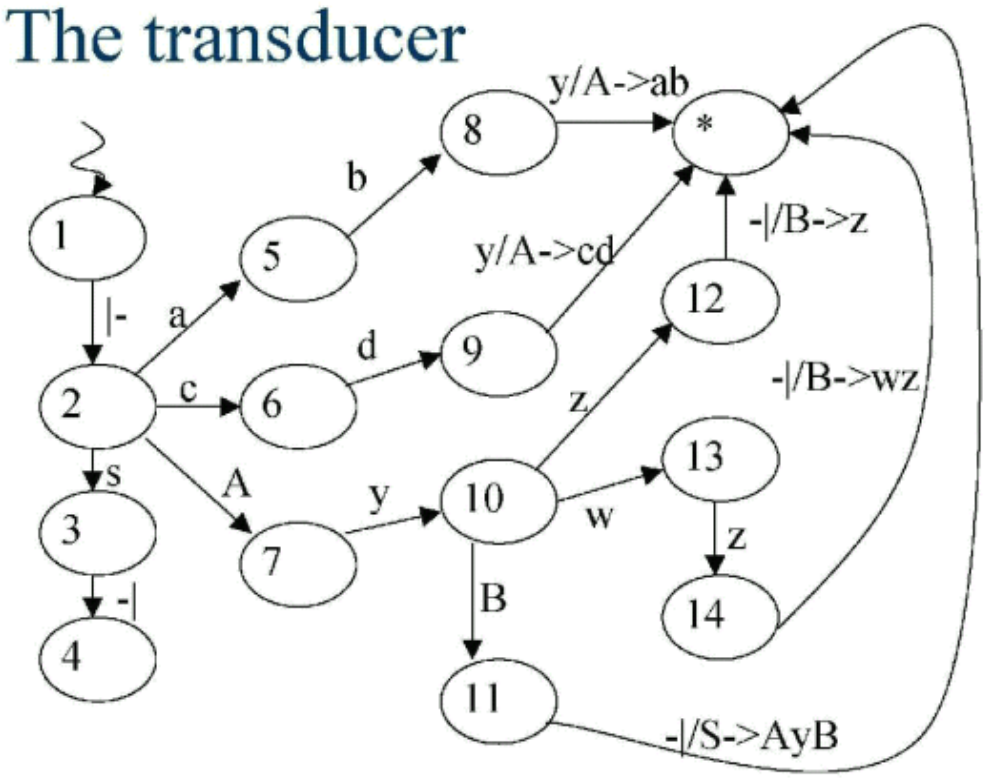
Shift/Reduce

- Somehow, we shifted at just the right time, and reduced just at the right time
- How did we know this?
  - Recall that for LL(1) parsing, we had a predictor table
  - For LR(1) parsing, we have an oracle, in the form of a DFA

Constructing DFA oracle for LR(1) grammars

- This is difficult to do
  - Donald Knuth proved a theorem that we can construct a DFA (really, a transducer) for LR(1) grammars (1965)
- You should know how to use the transducer

The transducer



Understanding the transducer

Indicates whether to:

- shift (default)
- reduce (indicates rule)
- reject
  - All states except * and ERROR are final states
- Let's use this on input ⊢abywz⊣

Using the transducer

top

Stack	States visited	Input read	Unread Input	Action
ϵ	1	$\vdash$	abywz$\dashv$	shift
$\vdash$	1 2	$\vdash$a	bywz$\dashv$	shift a
$\vdash$ a	1 2 5	$\vdash$ab	ywz$\dashv$	shift b
$\vdash$ a b	1 2 5 8	$\vdash$ab	ywz$\dashv$	Reduce A$\rightarrow$ ab
$\vdash$ A	1 2 7	$\vdash$aby	wz$\dashv$	shift y
$\vdash$ A y	1 2 7 10	$\vdash$abyw	z$\dashv$	shift w
$\vdash$ A y w	1 2 7 10 13	$\vdash$abywz	$\dashv$	shift z
$\vdash$ A y w z	1 2 7 10 13 14	$\vdash$abywz	$\dashv$	Reduce B $\rightarrow$ wz
$\vdash$ A y B	1 2 7 10 11	$\vdash$abywz	$\dashv$	Reduce S $\rightarrow$ AyB
$\vdash$ S	1 2 3	$\vdash$abywz	$\dashv$	shift
$\vdash$ S $\dashv$	1 2 3 4	$\vdash$abywz$\dashv$	ϵ	ACCEPT

read stack first

4 is not error not ϵ

pop B
pop y
pop A
push S

A basic algorithm

- for each input token
  - Start in the start state
  - Read the stack (from the bottom up) and read the current input, and do the action indicated for the current input
    - If at any point, we hit the error state, reject input
- At the end, if we read $\vdash S \dashv$ on the stack, accept

Making this more efficient

Current running time of this algorithm:

suppose N tokens: $O(n^2)$

Instead of scanning the stack each time...

push (input, state)

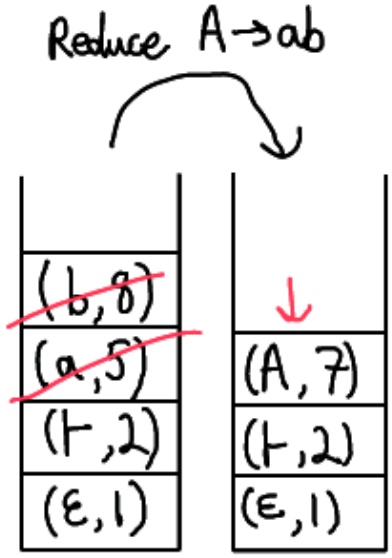
Start the transducer in....

the state at the top of the stack

Running time:

top: $O(1)$

overall $\Rightarrow O(N)$



Outputting a derivation

→ "proof"

- Easy: each time we do a reduction, output the rule
- But, this isn't quite right. Derivations should start with the start symbol. Bottom-up parsing doesn't.

$A \to ab$

$B \to wz$

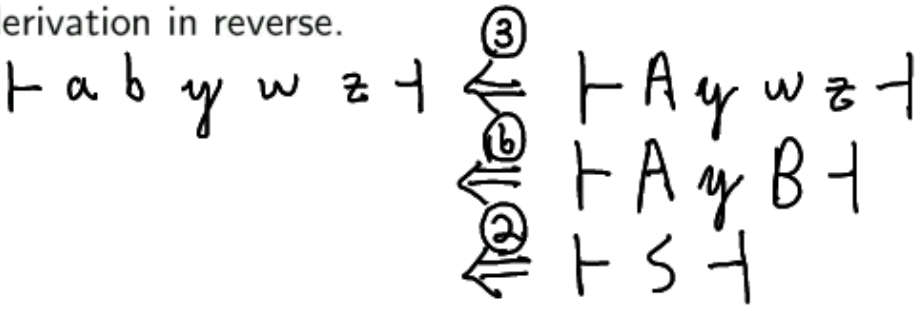
$S \to AyB$

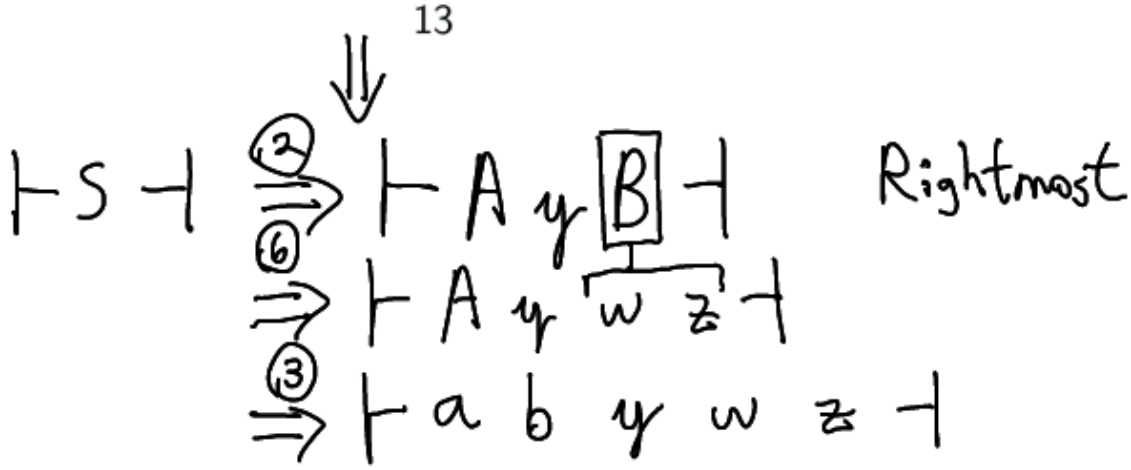
A simple observation

- Didn't we say that this was LR(1) parsing? yup
- Doesn't the "R" mean rightmost derivation? yep
- Aren't we always reducing the leftmost nonterminal?

  Uh, yeah: $A \to ab$, $B \to wz$ $\qquad S \to AyB$

- But notice the direction we are creating the derivation. Write the derivation in reverse.



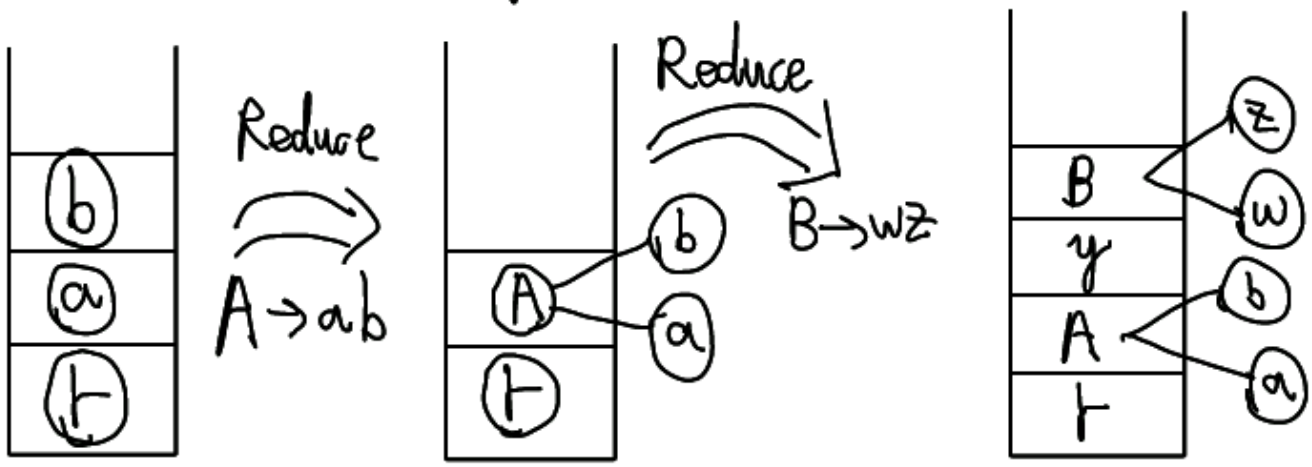


Outputting the parse tree

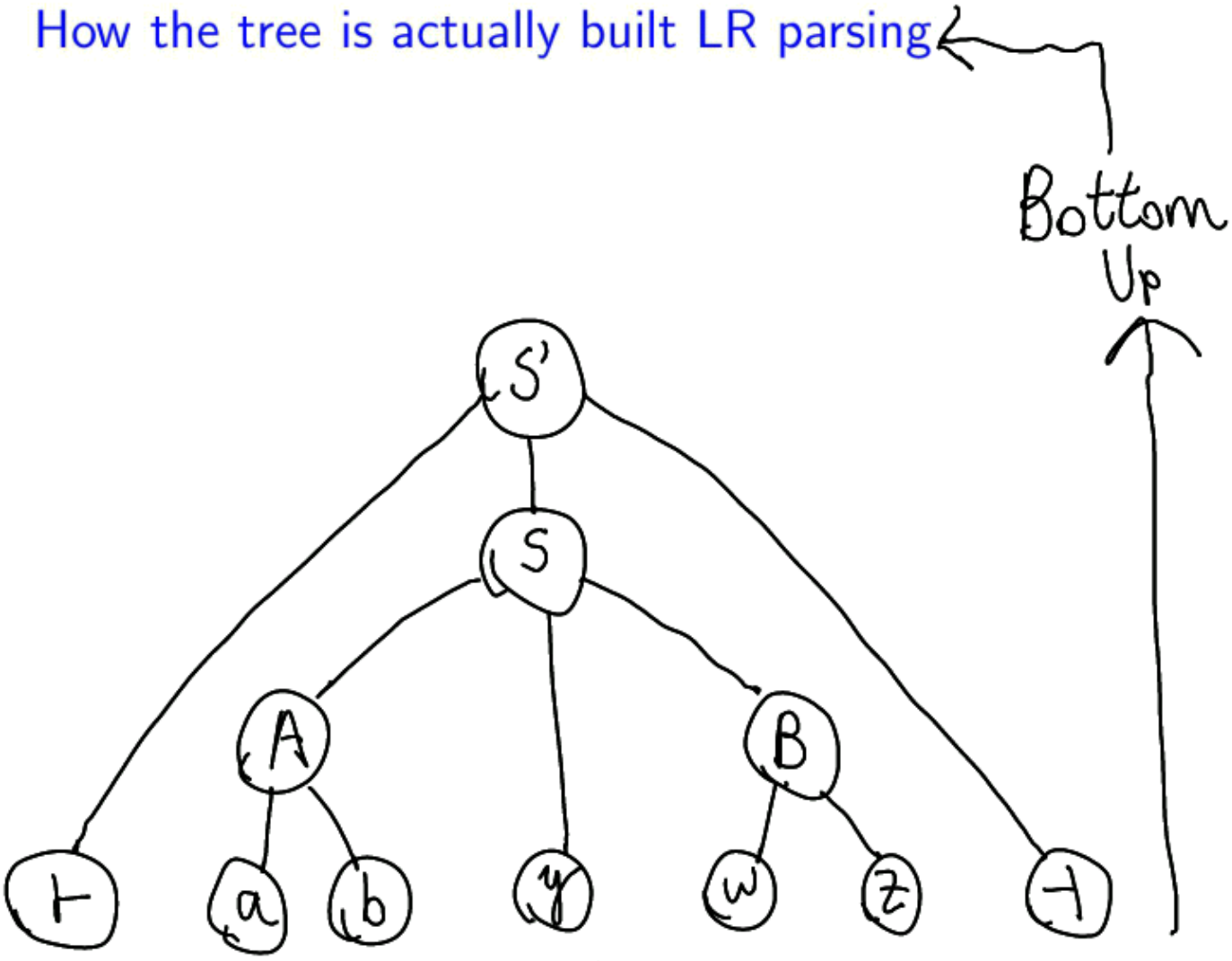
Algorithm

In sync with the (input, state) stack.

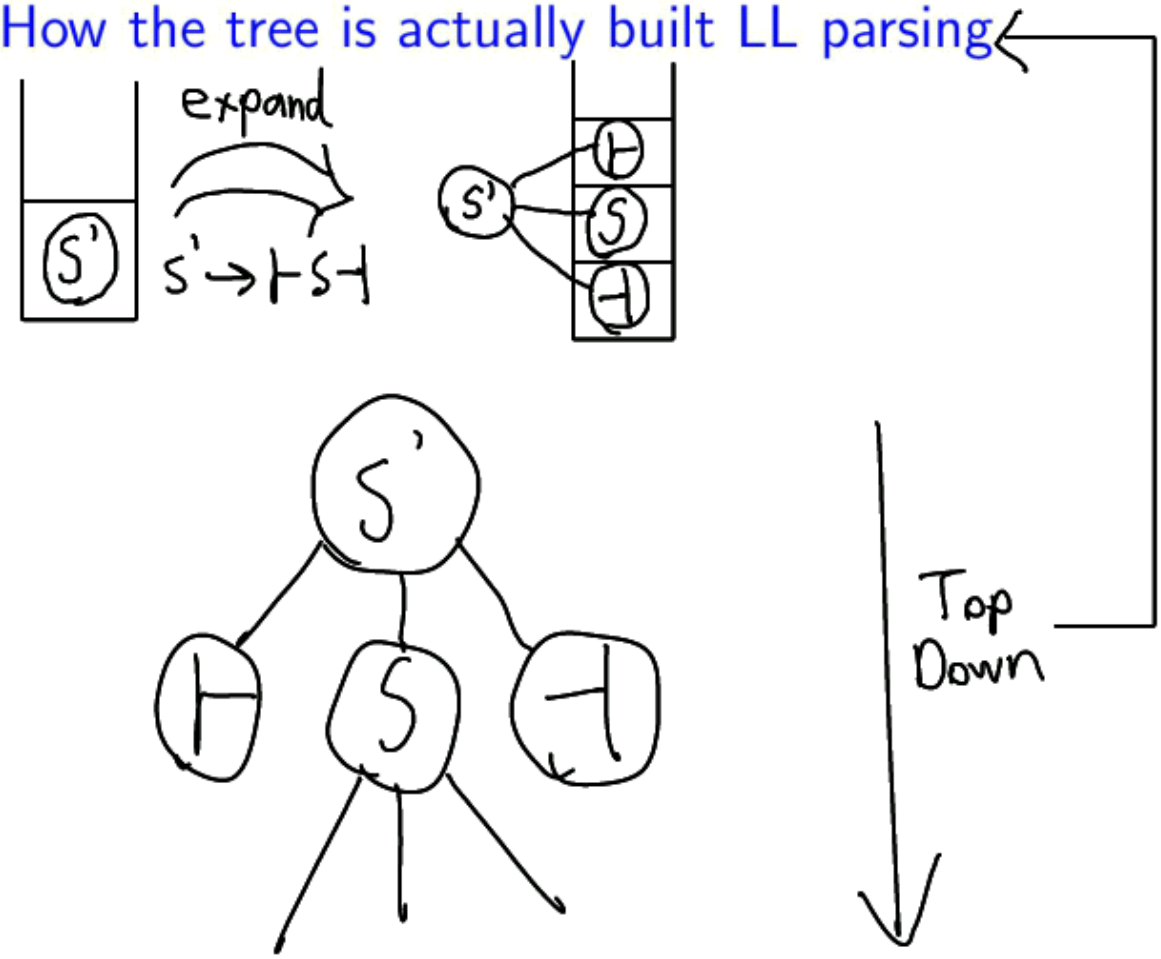
- Create a "tree stack"
- Each time we reduce, pop the right hand side nodes from tree stack
- Push the left hand side node and make its children the nodes we just popped
- Example: ⊢ a b y w z ⊣



How the tree is actually built LR parsing



How the tree is actually built LL parsing



Assignment hints

A7

P1, P2: write a cfg-r derivation by hand RTFS

spec

P3: Write a parser

- Read a CFG, the DFA and input
- Output cfg-r (derivation) → the oracle

P4: Write a parser for ~~WLPP~~ WLP4

- Your scanner reads characters as input and outputs tokens } A6
- Your parser will read tokens, builds a parse tree and outputs a left-most derivation WLP4
- Don't read the ~~WLPP~~ grammar from a separate file. Find a way to embed it in your program. Don't do this by typing it into your program directly!

Going back

Looking at: $L = \{a^n b^m : n \geq m \geq 0\}$ (non-LL(1) language)

and non-LL(k)

1. S' $\rightarrow \vdash$ S $\dashv$
2. S $\rightarrow$ a S
3. S $\rightarrow$ T
4. T $\rightarrow$ a T b
5. T $\rightarrow \epsilon$

} this is LR(1)

What to do when you see the symbol:

- $\vdash$ $\Rightarrow$ shift
- a $\Rightarrow$ shift
- b $\Rightarrow$ reduce by rule 5 if not on stack; shift, reduce by rule 4
- $\dashv$ $\Rightarrow$ reduce by rule 3, repeatedly reduce by rule 2 until reach $\vdash$, shift

|L(G)|=1 S⇒T T⇒Q Q⇒hi

only one rule for each non terminal

Notice: LL(0) grammars are useless

Final fun facts

zero tokens of lookahead!!

S'⇒⊢S⊣

- Theorem: For any augmented LR(1) grammar, there is an equivalent LR(0) grammar. See CS 462
- Theorem: The class of languages that can be parsed deterministically with a stack can be represented with an LR(1) grammar.
- Comparing LL(1) vs. LR(1)