Lecture 16
Bottom-Up Parsing

CS 241: Foundations of Sequential Programs
Fall 2009

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:

  $$\text{input processed} + \text{stack} = \text{current derivation}$$

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

  $$\text{stack}+\text{input to be read} = \text{current derivation}$$

  (stack is read from bottom to top here)

A trace

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

Shift: shifting a token from one place to another

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

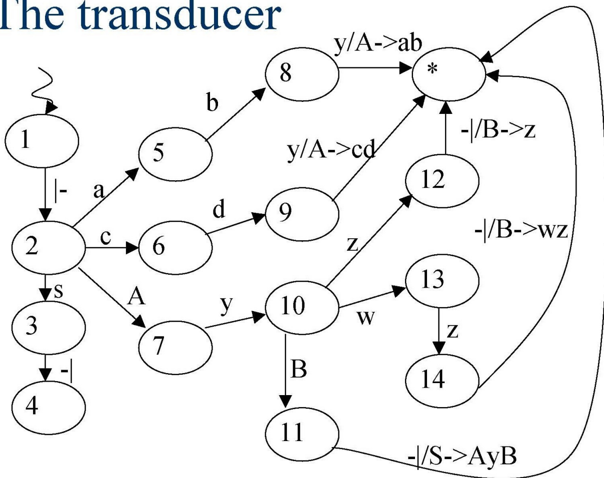
Shift/Reduce

- 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 $\vdash$abywz$\dashv$

Using the transducer

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

A basic algorithm

- Read the stack (from the bottom up) and read the current input, and do the action indicated
  - 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:

Instead of scanning the stack each time...

Start the transducer in....

Running time:

Outputting a derivation

- 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 simple observation

- Didn't we say that this was LR(1) parsing?
- Does the "R" mean rightmost derivation?
- Aren't we always reducing the leftmost nonterminal?
- But notice the direction we are creating the derivation. Write the derivation in reverse.
-

Outputting the parse tree

Algorithm

- 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

A8 hints

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

P3: Write a parser

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

P4: Write a parser for WL

- Your scanner reads characters as input and outputs tokens
- Your parser will read tokens, builds a parse tree and outputs a left-most derivation
- Don't read the WL 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!

Final fun facts

- Theorem: For any augmented LR(1) grammar, there is an equivalent LR(0) grammar.
- 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)