COMP 181

Lecture 8

Bottom-up Parsing

September 28, 2006

Homework

- 1. Extended regular expressions
  - `[s]`
    - For all c_i in s, c_1 | c_2 | c_3 | … | c_n
  - `[^s]`
    - For all c_i in Σ, not in s
  - r+
    - rr*
  - `r{n,m}`
    - r×n | … | r×m
    - e.g., `r{2,4}` <=> `rr | rrr | rrrr`

Homework

- 2. Regular expressions
  - All strings with the five vowels in order
    - `co = [^aeiou]`
    - `co* a co* e co* i co* o co* u co*`
  - C block comments
    - `'/\*' .* '\*/'`
    - `'/\*' ([^\*] | \*[^/])* '\*/'`
    - `/* foo */ bar /* baz */`
  - IP addresses
    - num = `0 | [0-9] | [0-9][0-9] 1 [0-9][0-9] | 2[0-4][0-9] | 25[0-5]`
    - `num \. num \. num \. num`

Homework

- All strings of 0,1 with no 011
  - `1* (0+ 1)* 0*`
  - `1* (01 | 0)*`
- 3. Language of parens
  - Limit to length 10?
    - Yes, enumerate all strings
  - Limit to depth 3?
    - Yes, enumerate cases:
    - `( () | (()) | ((())) )*`

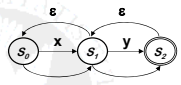
Homework

- Single nest, unlimited depth
- No,
  - Intuition: need to count
  - How to make sure numbers match?

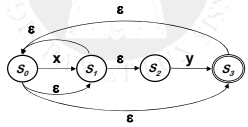
  - Not a ***finite*** automaton
  - Proof: *Pumping lemma*

Homework

- 4. Thompson's construction
  - Problem: consider `x*y*`:



  - Depending on concatenation rule…
  - `(x*y)*`



Homework

- 5. Reading numbers in different bases

Where are we

- Finished top-down parsing
- Start bottom-up parsing

Bottom-Up Parsing

- More general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
  - Preferred method in practice
- Also called LR parsing
  - L means that tokens are read left to right
  - R means that it constructs a rightmost derivation

An Introductory Example

- LR parsers:
  - Can handle left-recursion
  - Don't need left factoring
- Consider the following grammar:

  $E \rightarrow E + (E) \mid int$

  - Why is this not LL(1)?
- Consider the string: int + (int) + (int)

The Idea

- LR parsing ***reduces*** a string to the start symbol by inverting productions:

  str = input string of terminals
  repeat
  - Identify β in str such that $A \rightarrow \beta$ is a production (i.e., str = $\alpha\ \beta\ \gamma$)
  - Replace β by A in str (i.e., str becomes $\alpha\ A\ \gamma$)

  until str = G

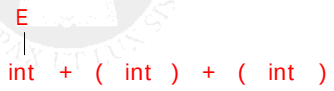
A Bottom-up Parse in Detail (1)

int + (int) + (int)

int + (int) + (int)

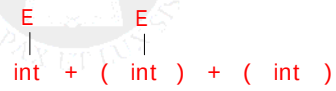
A Bottom-up Parse in Detail (2)

int + (int) + (int)
E + (int) + (int)



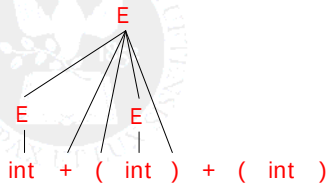
A Bottom-up Parse in Detail (3)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)



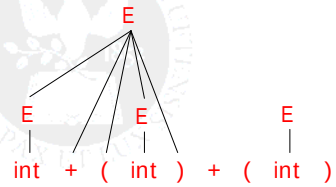
A Bottom-up Parse in Detail (4)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)



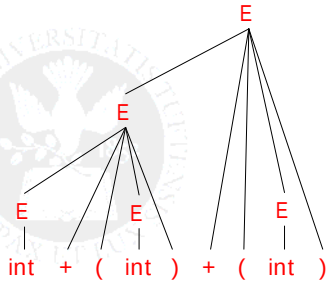
A Bottom-up Parse in Detail (5)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)



A Bottom-up Parse in Detail (6)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E



Another example

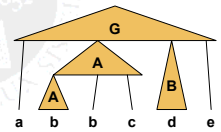
- Grammar:

#	Production rule
1	*G* → *a A B e*
2	*A* → A b c
3	| b
4	B → *d*

- Is "**abbcde**" in L(G)?
- Yes

"Reverse" derivation:

Rule	Sentential form
-	*abbcde*
3	*aAbcde*
2	*aAde*
4	*aABe*
1	*G*



Choosing reductions

- Basic algorithm:
  - Search for right sides of productions, reduce
  - Does this work?
- Not always:

Rule	Sentential form
-	*abbcde*
3	*aAbcde*
2	*aAAcde*
?	*...now what?*

- Problem:

  "aAAcde" is not part of any sentential form

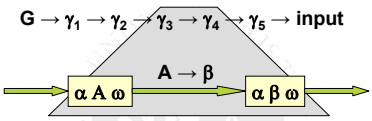
How do we choose?

Important Fact #1 about bottom-up parsing:

An LR parser traces a rightmost derivation in reverse

Why does this help?

- Right-most derivation



- A is the right-most non-terminal in γ_3
- ω contains only terminal symbols
- Unambiguous grammar
  - Right-most derivation is unique
  - ⇨ At each step, reduction is unique

Notation

- Split input into two substrings
  - Right substring (a string of terminals) is as yet unexamined by parser
  - Left substring has terminals and non-terminals
- The dividing point is marked by a I
  - The I is not part of the string
- Initially, all input is unexamined: $Ix_1x_2 \ldots x_n$

Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

Shift: Move I one place to the right

- Shifts a terminal to the left string

E + (I int) ⇒ E + (int I)

Reduce

Reduce: Apply an inverse production at the right end of the left string

- If E → E + (E) is a production, then

E + (E + (E) I) ⇒ E +(E I)

Shift-Reduce Example

I int + (int) + (int)$ shift

int + (int) + (int)

Shift-Reduce Example

I int + (int) + (int)$ shift
int I + (int) + (int)$ red. E → int

int + (int) + (int)

Shift-Reduce Example

I int + (int) + (int)$ shift
int I + (int) + (int)$ red. E --> int
E I + (int) + (int)$ shift 3 times

E
int + (int) + (int)

Shift-Reduce Example

I int + (int) + (int)$ shift
int I + (int) + (int)$ red. E --> int
E I + (int) + (int)$ shift 3 times
E + (int I) + (int)$ red. E --> int

E
int + (int) + (int)

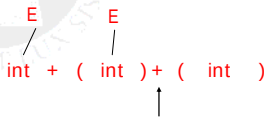
Shift-Reduce Example

I int + (int) + (int)$ shift
int I + (int) + (int)$ red. E --> int
E I + (int) + (int)$ shift 3 times
E + (int I) + (int)$ red. E --> int
E + (E I) + (int)$ shift

E E
int + (int) + (int)

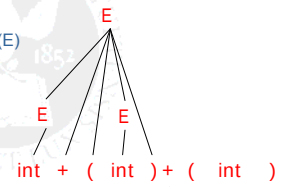
Shift-Reduce Example

I int + (int) + (int)$	shift
int I + (int) + (int)$	red. E --> int
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	red. E --> int
E + (E I) + (int)$	shift
E + (E) I + (int)$	red. E --> E + (E)



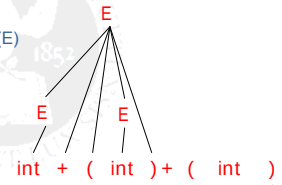
Shift-Reduce Example

I int + (int) + (int)$	shift
int I + (int) + (int)$	red. E --> int
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	red. E --> int
E + (E I) + (int)$	shift
E + (E) I + (int)$	red. E --> E + (E)
E I + (int)$	shift 3 times



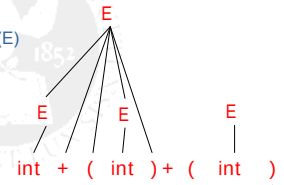
Shift-Reduce Example

I int + (int) + (int)$	shift
int I + (int) + (int)$	red. E --> int
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	red. E --> int
E + (E I) + (int)$	shift
E + (E) I + (int)$	red. E --> E + (E)
E I + (int)$	shift 3 times
E + (int I)$	red. E --> int



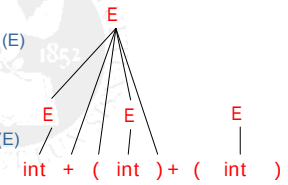
Shift-Reduce Example

I int + (int) + (int)$	shift
int I + (int) + (int)$	red. E --> int
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	red. E --> int
E + (E I) + (int)$	shift
E + (E) I + (int)$	red. E --> E + (E)
E I + (int)$	shift 3 times
E + (int I)$	red. E --> int
E + (E I)$	shift



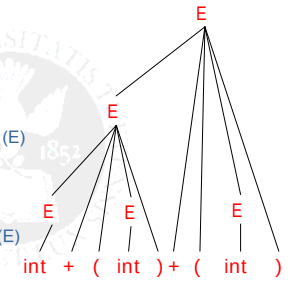
Shift-Reduce Example

I int + (int) + (int)$	shift
int I + (int) + (int)$	red. E --> int
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	red. E --> int
E + (E I) + (int)$	shift
E + (E) I + (int)$	red. E --> E + (E)
E I + (int)$	shift 3 times
E + (int I)$	red. E --> int
E + (E I)$	shift
E + (E) I $	red. E --> E + (E)



Shift-Reduce Example

I int + (int) + (int)$	shift
int I + (int) + (int)$	red. E --> int
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	red. E --> int
E + (E I) + (int)$	shift
E + (E) I + (int)$	red. E --> E + (E)
E I + (int)$	shift 3 times
E + (int I)$	red. E --> int
E + (E I)$	shift
E + (E) I $	red. E --> E + (E)
E I $	accept



How do we keep track?

- Left part string implemented as a **stack**
  - Top of the stack is the |
  - Shift:
    - Pushes a terminal on the stack
  - Reduce:
    - Pops 0 or more symbols off of the stack
    - Symbols are right-hand side of a production
    - Pushes a non-terminal on the stack (production LHS)
  - Terminology
    - We refer to the top set of symbols as a ***handle***