CS412/CS413

Introduction to Compilers
Tim Teitelbaum

Lecture 8: LR parsing
February 7, 2007

Bottom-up Parsing

- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - Scan the input from Left to right and determine a Right-most derivation of program (backwards)
  - Allows left-recursive grammars, virtually all programming languages
  - Easier to express programming language syntax
- Shift-reduce parsers
  - Parsers for LR grammars
  - Automatic parser generators (*e.g.*, yacc, CUP)

Bottom-up Parsing

- Right-most derivation -- backward
  - Start with the tokens; end with the start symbol

$(\underline{1}+2+(3+4))+5 \Leftarrow$
$(\underline{E}+2+(3+4))+5 \Leftarrow$
$(S+\underline{2}+(3+4))+5 \Leftarrow$
$(\underline{S+E}+(3+4))+5 \Leftarrow$
$(S+(\underline{3}+4))+5 \Leftarrow$
$(S+(\underline{E}+4))+5 \Leftarrow$
$(S+(S+\underline{4}))+5 \Leftarrow$
$(S+(\underline{S+E}))+5 \Leftarrow$
$(S+\underline{(S)})+5 \Leftarrow$
$(\underline{S+E})+5 \Leftarrow$
$\underline{(S)}+5 \Leftarrow$
$\underline{E}+5 \Leftarrow$
$S+\underline{5} \Leftarrow$
$\underline{S+E} \Leftarrow$
S

S → S + E | E
E → num | (S)

Progress of Bottom-up Parsing

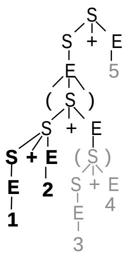
right-most derivation

(1+2+(3+4))+5 ⇐		(1+2+(3+4))+5
(E+2+(3+4))+5 ⇐	(1	+2+(3+4))+5
(S+2+(3+4))+5 ⇐	(1	+2+(3+4))+5
(S+E+(3+4))+5 ⇐	(1+2	+(3+4))+5
(S+(3+4))+5 ⇐	(1+2+(3	+4))+5
(S+(E+4))+5 ⇐	(1+2+(3	+4))+5
(S+(S+4))+5 ⇐	(1+2+(3	+4))+5
(S+(S+E))+5 ⇐	(1+2+(3+4	))+5
(S+(S))+5 ⇐	(1+2+(3+4	))+5
(S+E)+5 ⇐	(1+2+(3+4)	)+5
(S)+5 ⇐	(1+2+(3+4)	)+5
E+5 ⇐	(1+2+(3+4))	+5
S+E ⇐	(1+2+(3+4))+5	
S	(1+2+(3+4))+5	

Bottom-up Parsing

- $(\underline{1}+2+(3+4))+5 \Leftarrow$
  $(\underline{E}+2+(3+4))+5 \Leftarrow$
  $(S+\underline{2}+(3+4))+5 \Leftarrow$
  $(S+E+(3+4))+5$ ...
- If $S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta w$ then β is called a handle of $\alpha\beta w$
- Advantage of bottom-up parsing: the selection of production is postponed until after the handle has been scanned

S → S + E | E
E → num | (S)



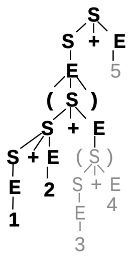
Top-down Parsing

(1+2+(3+4))+5

S ⇒ S+E ⇒ E+E ⇒ (S)+E ⇒ (S+E)+E
⇒ (S+E+E)+E ⇒ (E+E+E)+E
⇒ (1+E+E)+E ⇒ (1+2+E)+E ...

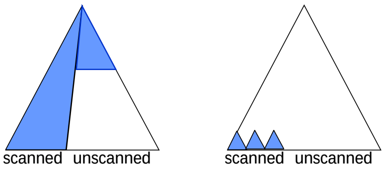
S → S + E | E
E → num | (S)

- In left-most derivation, entire tree above a token (2) has been expanded when it is encountered



Top-down vs. Bottom-up

Bottom-up: Don't need to figure out as much of the parse tree for a given amount of input



Top-down Bottom-up

Shift-reduce Parsing

- Parsing actions: a sequence of shift and reduce operations
- Parser state: a stack of terminals and non-terminals (grows to the right)
- Current derivation is always stack+input

Derivation step	stack	unconsumed input
(1+2+(3+4))+5 ⇐		(1+2+(3+4))+5
	(	1+2+(3+4))+5
	(1	+2+(3+4))+5
(E+2+(3+4))+5 ⇐	(E	+2+(3+4))+5
(S+2+(3+4))+5 ⇐	(S	+2+(3+4))+5
	(S+	2+(3+4))+5
	(S+2	+(3+4))+5
(S+E+(3+4))+5 ⇐	(S+E	+(3+4))+5

Shift-reduce Parsing

- Parsing is a sequence of shifts and reduces
- Shift: move look-ahead token to stack

stack	input	action
(	1+2+(3+4))+5	shift 1
(1	+2+(3+4))+5	

- Reduce: Replace symbols β from top of stack with non-terminal symbol A, corresponding to production A → β (pop β , push A)

stack	input	action
(S+E	+(3+4))+5	reduce S→ S+E
(S	+(3+4))+5	

Shift-reduce Parsing

S → S + E | E
E → num | (S)

derivation	stack	input stream	action
(1+2+(3+4))+5 ⇐		(1+2+(3+4))+5	shift
(1+2+(3+4))+5 ⇐	(	1+2+(3+4))+5	shift
(1+2+(3+4))+5 ⇐	(1	+2+(3+4))+5	reduce E→num
(E+2+(3+4))+5 ⇐	(E	+2+(3+4))+5	reduce S → E
(S+2+(3+4))+5 ⇐	(S	+2+(3+4))+5	shift
(S+2+(3+4))+5 ⇐	(S+	2+(3+4))+5	shift
(S+2+(3+4))+5 ⇐	(S+2	+(3+4))+5	reduce E→num
(S+E+(3+4))+5 ⇐	(S+E	+(3+4))+5	reduce S→ S+E
(S+(3+4))+5 ⇐	(S	+(3+4))+5	shift
(S+(3+4))+5 ⇐	(S+	(3+4))+5	shift
(S+(3+4))+5 ⇐	(S+(	3+4))+5	shift
(S+(3+4))+5 ⇐	(S+(3	+4))+5	reduce E→num

Problem

- How do we know which action to take: whether to shift or reduce, and which production?
- Issues:
  - Sometimes can reduce but shouldn't
  - Sometimes can reduce in different ways

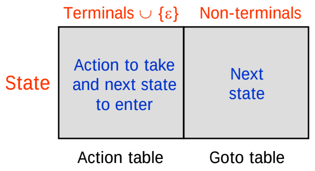
Action Selection Problem

- Given stack σ and look-ahead symbol b, should parser:
  - shift b onto the stack (making it σb)
  - reduce A→β assuming that stack has the form αβ (making it αA)

LR Parsing Engine

- Basic mechanism:
  - Use a set of parser states
  - Use a stack of states
  - Use a parsing table to:
    - Determine what action to apply (shift/reduce)
    - Determine the next state
- The parser actions can be precisely determined from the table

The LR Parsing Table



- Algorithm: look at entry for current state Q and input terminal c

  If Table[Q,c] = shift(Q') then shift:
  push(Q')

  If Table[Q,c] = A→α then reduce:
  pop(|α|); Q'=top(); push(Table[Q',A])

LR(1) Parsing Table Example

	(	)	id	,	ε	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
7	L→S	L→S	L→S	L→S	L→S		
8	s3		s2			g9	
9	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

LR(k) Grammars

- LR(k) = Left-to-right scanning, Right-most derivation, k look-ahead characters
- Main cases: LR(0), LR(1), and some variations (SLR and LALR(1))
- Parsers for LR(0) Grammars:
  - Determine the actions without any lookahead symbol
  - Will help us understand shift-reduce parsing
- Read: CUP User Manual