CS412/413

Introduction to Compilers
Radu Rugina

Lecture 8: Bottom-UpParsing
8 Feb 06

Bottom-Up Parsing

- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - Construct right-most derivation of program
  - 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 ↑

(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

- (1+2+(3+4))+5 ⇐
  (E+2+(3+4))+5 ⇐
  (S+2+(3+4))+5 ⇐
  (S+E+(3+4))+5 ...
- Advantage of bottom-up parsing: can postpone the selection of productions until more of the input is scanned

S → S + E | E
E → num | (S)

Top-Down Parsing

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

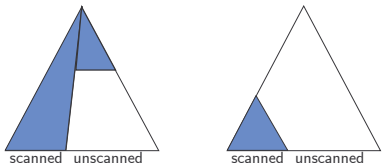
S → S + E | E
E → num | (S)

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 ...

- In left-most derivation, entire tree above a token (2) has been expanded when 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 stack: contains terminals and non-terminals
  - Also contains state numbers, will discuss them later
- Current derivation step = 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 X, 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

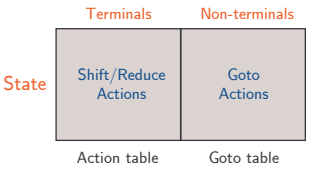
Problem

- How to figure out the actions:
  - Shift or reduce?
  - Which production?
- Issues:
  - Sometimes can reduce but shouldn't
  - Sometimes can reduce in multiple ways

LR Parsing Engine

- Basic mechanism:
  - Compute a set of parser states
  - Use a parsing stack
  - Build 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 S and input terminal c

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

  If Table[S,c] = A→α then reduce:
  pop(2|α|); S'=top(); push(A); push(Table[S',A])

LR(1) Parsing Table Example

	(	)	id	,	$	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

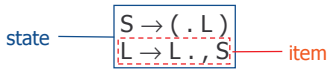
- LR(k) = Left-to-right scanning, Right-most derivation, k look-ahead characters
- Main cases: LR(0), LR(1), and some variations (SLR and LALR(1))
- Parsers for LR(0) Grammars:
  - Determine the actions without any lookahead symbol
  - will help us understand shift-reduce parsing

Building LR(0) Parsing Tables

- To build the parsing table:
  - Compute parser states
  - Build a DFA to describe the transitions between states
  - Use the DFA to build the parsing table
- Each LR(0) state is a set of LR(0) items:
  - An LR(0) item: $X \rightarrow \alpha \,.\, \beta$, where $X \rightarrow \alpha\,\beta$ is a production in the grammar
  - The LR(0) items keep track of the progress on all of the possible upcoming productions
  - The item $X \rightarrow \alpha \,.\, \beta$ abstracts the fact that the parser already matched the string α at the top of the stack

Example LR(0) State

- An LR(0) item is a production from the language with a separator "." somewhere in the RHS of the production



- Sub-string before "." is already on stack
- Sub-string after "." : what we might see next

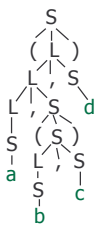
LR(0) Grammar

- Nested lists:

  S → (L) | id
  L → S | L , S

- Examples
  - (a, b, c)
  - ((a,b), (c,d), (e,f))
  - (a, (b,c,d), ((f,g)))

Parse tree for (a, (b,c), d)



Start State & Closure

- Start state
  - Augment grammar with production S ' → S $
  - Start state of DFA has empty stack: S ' → . S $
- Closure of a parser state:
  - Start with Closure(S) = S
  - Then for each item in S:

    $X \rightarrow \alpha \,.\, Y\,\beta$

    add the items for all the productions $Y \rightarrow \gamma$ to the closure of S:

    $Y \rightarrow \,.\, \gamma$

Closure Example

S → (L) | id
L → S | L, S

DFA start state

S ' → . S $ —closure→ S ' → . S $
S → . (L)
S → . id

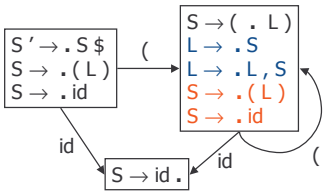
– set of possible productions to be reduced next
– Added items have the "." located at the beginning: no symbols for these items on the stack yet

The Goto Operation

- Goto operation = describes transitions between parser states, which are sets of items
- Algorithm: for a state S and a symbol Y
  – S' = {X → α Y . β | X → α . Y β ∈ S}
  – Goto(S, Y) = Closure(S')

S ' → . S $
S → . (L)
S → . id —Goto(S, '(')→ Closure({S → (. L)})

Goto: Terminal Symbols

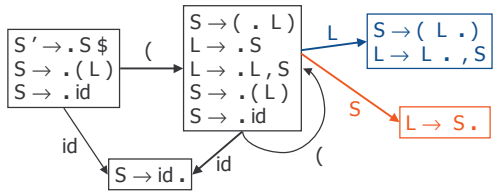


Grammar:

S → (L) | id
L → S | L , S

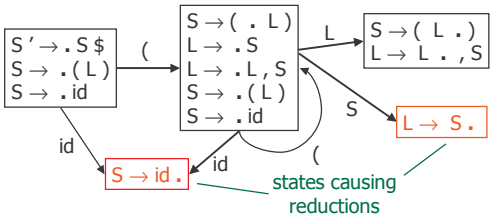
In new state, include all items that have appropriate input symbol just after dot, advance dot in those items, and take closure.

Goto: Non-terminal Symbols



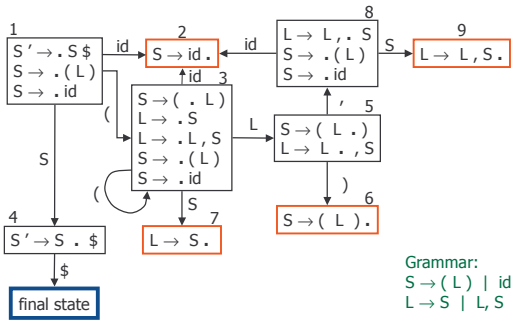
(same algorithm for transitions on non-terminals)

Applying Reduce Actions



- Pop RHS off stack, replace with LHS X (X→γ), then rerun DFA (e.g. (x))

Full DFA



Parsing Example: ((a),b)

S → (L) | id
L → S | L, S

derivation	stack	input	action
((a),b) ←	$_1$	((a),b)	shift, goto 3
((a),b) ←	$_1$ ($_3$	(a),b)	shift, goto 3
((a),b) ←	$_1$ ($_3$ ($_3$	a),b)	shift, goto 2
((a),b) ←	$_1$ ($_3$ a$_2$	),b)	reduce S→id
((S),b) ←	$_1$ ($_3$ S$_7$	),b)	reduce L→S
((L),b) ←	$_1$ ($_3$ L$_5$	),b)	shift, goto 6
((L),b) ←	$_1$ ($_3$ L$_5$)$_6$	,b)	reduce S→(L)
(S,b) ←	$_1$ ($_3$ S$_7$	,b)	reduce L→S
(L,b) ←	$_1$ ($_3$ L$_5$	,b)	shift, goto 8
(L,b) ←	$_1$ ($_3$ L$_5$, $_8$	b)	shift, goto 9
(L,b) ←	$_1$ ($_3$ L$_5$, $_8$ b$_2$	)	reduce S→id
(L,S) ←	$_1$ ($_3$ L$_5$, $_8$ S$_9$	)	reduce L→L , S
(L) ←	$_1$ ($_3$ L$_5$	)	shift, goto 6
(L) ←	$_1$ ($_3$ L$_5$)$_6$		reduce S→(L)
S	$_1$ S$_4$	$	done

Reductions

- When reducing X→γ with stack αγ:
  - pop **γ** off stack, revealing prefix α and state s
  - take single step in DFA from top state s on symbol X
  - push X onto stack with new DFA state
- Example:

((a),b) ←	$_1$ ($_3$ ($_3$	a),b)	shift, goto 2
((a),b) ←	$_1$ ($_3$ ($_3$ a$_2$	),b)	reduce S→id
((S),b) ←	$_1$ ($_3$ ($_3$ S$_7$	),b)	…

Build the Parsing Table

- States in the table = states in the DFA
- For a transition S → S' on terminal c:
  Shift(S') ⊆ Table[S,c]
- For a transition S → S' on non-terminal N:
  Goto(S') ⊆ Table[S,N]
- If S is a reduction state X → **γ** then:
  Reduce(X → **γ**) ⊆ Table[S,*]

Computed LR Parsing Table

	(	)	id	,	$	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(0) Summary

- LR(0) parsing recipe:
  Start with an LR(0) grammar
  Compute LR(0) states and build DFA:
  Build the LR(0) parsing table from the DFA
- This process can be automated, i.e. we can build parser generator tools