Bottom Up (LR) Parsing

Michal Young

April 12, 2006

Introduction to Bottom-Up Parsing

Building Parsing Tables
“Bottom Up” Parsing

Also known as “LR parsing” and “shift reduce parsing”

**Bottom-up:** Recognize all children before recognizing parent node. (Vs. top-down prediction after seeing one leaf)

**Shift-reduce:** We’ll *shift* some symbols onto a stack before *reducing* them to a non-terminal.

**LR:** Left-to-right scan of input, rightmost derivation in reverse.

*Ummm ... huh?*
Shift-Reduce parsing

Intuitive view:
▶ *Shift* the input cursor past matching symbols, keeping a marker in *each* partially matched production
▶ *Reduce* matching symbols to non-terminal when end of production is reached; slide the input cursor back to match non-terminals and terminals

Implementation:
▶ Build state machine for *all* possible paths through productions
▶ Interpret as push-down automaton: *reduce* action pops earlier position
A trivial grammar

Comma-separated lists (just one level)

\[ \langle S \rangle ::= \langle L \rangle \; \$ \]

\[ \langle L \rangle ::= \langle L \rangle \; , \; i \]

\[ \langle L \rangle ::= i \]

Note left recursion — not a problem

Examples:
a, b, c \$

a \$

a \$
LR(0) parsing table

Don’t worry about how we built it (yet)
Initial Configuration
Step 1: Shift 1 (i)
Step 2: Reduce $L$
Step 3: Shift 1 (L)

1. $S ::= \cdot L \cdot$
   - $L ::= \cdot L, i$
   - $L ::= \cdot i$

2. $S ::= L \cdot$
   - $L ::= L, i$
   - $S ::= L \cdot$

3. $S ::= L \cdot$
   - $S ::= L \cdot$

4. $L ::= i \cdot$

5. $L ::= L, i$

6. $L ::= L, i \cdot$

Input: $a, b, c \cdot$
Step 4: Shift 2 (,)

```
S ::= • L $
L ::= • L , i
L ::= • i
S ::= L • $
L ::= L • , i
S ::= L $ •
L ::= i • L ::= L , • i L ::= L , i •
L $
i
i
,
1 2 3
4 5 6
a
, b , c $L
1
2
```
Step 5: Shift 5 (i)
Step 6: Reduce L

\[
S ::= \cdot L \, S \\
L ::= \cdot L, \, i \\
L ::= \cdot i \\
S ::= L \cdot S \\
L ::= L \cdot i \\
L ::= \cdot S \\
L ::= L, \cdot i \\
S ::= L \cdot S, \, i \\
L ::= \cdot L, \, i \\
L ::= L, \, i \\
\]

\[
\begin{align*}
S &::= \cdot L \\
L &::= \cdot L, \, i \\
L &::= \cdot i \\
S &::= L \cdot S \\
L &::= L \cdot i \\
L &::= \cdot S \\
L &::= L, \cdot i \\
S &::= L \cdot S, \, i \\
L &::= \cdot L, \, i \\
L &::= L, \, i \\
\end{align*}
\]
Step 7: Shift 2 (,)

\[
S ::= \bullet L \$
\]

\[
L ::= \bullet L, i
\]

\[
L ::= \bullet i
\]

\[
S ::= L \bullet$
\]

\[
L ::= L, \bullet i
\]

\[
L ::= \bullet, i
\]

\[
L ::= L, i \bullet
\]
Step 8: Shift 2 (,)

```
S ::= • L $
L ::= • L , i
L ::= • i
S ::= L • $
L ::= L • , i
S ::= L $ •
L ::= i •
L ::= L , • i
L ::= L , i •
```

Diagram:

1. $S ::= • L$
2. $S ::= L •$
3. $S ::= L $ •
4. $L ::= i •$
5. $L ::= L , i$
6. $L ::= L , i •$
7. $L ::= • L , i$
8. $L ::= • i$
9. $L ::= • L$
10. $S ::= • L$
11. $S ::= L •$
12. $S ::= L $ •
13. $L ::= i •$
14. $L ::= L , i$
15. $L ::= L , i •$
16. $L ::= • L , i$
17. $L ::= • i$
18. $L ::= • L$
19. $S ::= • L$
20. $S ::= L •$
21. $S ::= L $ •
22. $L ::= i •$
23. $L ::= L , i$
24. $L ::= L , i •$
25. $L ::= • L , i$
26. $L ::= • i$
27. $L ::= • L$
28. $S ::= • L$
29. $S ::= L •$
30. $S ::= L $ •
31. $L ::= i •$
32. $L ::= L , i$
33. $L ::= L , i •$
34. $L ::= • L , i$
35. $L ::= • i$
36. $L ::= • L$
37. $S ::= • L$
38. $S ::= L •$
39. $S ::= L $ •
40. $L ::= i •$
41. $L ::= L , i$
42. $L ::= L , i •$
43. $L ::= • L , i$
44. $L ::= • i$
45. $L ::= • L$
46. $S ::= • L$
47. $S ::= L •$
48. $S ::= L $ •
49. $L ::= i •$
50. $L ::= L , i$
51. $L ::= L , i •$
52. $L ::= • L , i$
53. $L ::= • i$
54. $L ::= • L$
55. $S ::= • L$
56. $S ::= L •$
57. $S ::= L $ •
58. $L ::= i •$
59. $L ::= L , i$
60. $L ::= L , i •$
61. $L ::= • L , i$
62. $L ::= • i$
63. $L ::= • L$
64. $S ::= • L$
65. $S ::= L •$
66. $S ::= L $ •
67. $L ::= i •$
68. $L ::= L , i$
69. $L ::= L , i •$
70. $L ::= • L , i$
71. $L ::= • i$
72. $L ::= • L$
73. $S ::= • L$
74. $S ::= L •$
75. $S ::= L $ •
76. $L ::= i •$
77. $L ::= L , i$
78. $L ::= L , i •$
79. $L ::= • L , i$
80. $L ::= • i$
81. $L ::= • L$
82. $S ::= • L$
83. $S ::= L •$
84. $S ::= L $ •
85. $L ::= i •$
86. $L ::= L , i$
87. $L ::= L , i •$
88. $L ::= • L , i$
89. $L ::= • i$
90. $L ::= • L$
91. $S ::= • L$
92. $S ::= L •$
93. $S ::= L $ •
94. $L ::= i •$
95. $L ::= L , i$
96. $L ::= L , i •$
Step 9: Shift 5 (i)
Step 10: Reduce L

\[
S ::= \cdot L$
\]

\[
L ::= \cdot L, i$
\]

\[
L ::= \cdot i$
\]

\[
S ::= L \cdot$
\]

\[
S ::= L \cdot, i$
\]

\[
S ::= L \cdot, i$
\]

\[
S ::= L \cdot$
\]

\[
S ::= L \cdot$
\]
Step 11: Shift 1 (L)

1. $S ::= \cdot L \$ 
$L ::= \cdot L , i$
$L ::= \cdot i$
2. $S ::= L \cdot \$ 
$L ::= L \cdot , i$
3. $S ::= L \$ \cdot$

4. $L ::= \cdot i \cdot$
5. $L ::= L , \cdot i$
6. $L ::= L , i \cdot$

$S ::= \cdot L$
$L ::= \cdot L , i$
$L ::= \cdot i$
$S ::= L \cdot$
$L ::= L \cdot , i$
$L ::= \cdot i \cdot$
$L \ni$
$i

1 2 3
4 5 6
$a
, b L
L \ni
L$
Step 12: Shift 2 ($)

```
S ::= • L $
L ::= • L , i
L ::= • i
S ::= L •$
L ::= L • , i
S ::= L $ •
L ::= i • L ::= L , • i
L ::= L , i •

i
1
2
3
4
5
6

i

L

L

L

L

L

L

S

L

L

L

L

L

$
Step 13: Reduce $S$

```
S ::= \cdot L \$
L ::= \cdot L, i
L ::= \cdot i

L ::= \cdot L \$
L ::= L \cdot, i

L ::= L \cdot, i
L ::= L, i \$
S ::= L \cdot$
L ::= L \cdot
L ::= i \cdot L
L ::= L, i
```
Core representation used by all LR parsing techniques:

- Position in production — called LR(0) “item”
- State = set of items (multiple partial matches)
- Pre-built state machine for all possible paths

Difference: Use of lookahead to allow reduction

- LR(0): No lookahead (too weak)
- SLR(1): Use “first” and “follow” as in LL(1)
- LR(1): Compute lookaheads\(^1\) and split states (too expensive)
- LALR(1): Compute lookaheads, but don’t split states

\(^1\)Method described later
Building LR parsing tables

- For SLR(1) and LALR(1), we start with LR(0) "items"
- Treat productions as NFA, LR(0) items as NFA states, sets of items as DFA states.
  (The subset construction, again)
Comma list grammar as NFA

\[
\langle S \rangle ::= \langle L \rangle \ \$ \\
\langle L \rangle ::= \langle L \rangle , \ i \\
\langle L \rangle ::= \langle i \rangle
\]

- \( \epsilon \) transitions for each non-terminal
- “Final” state means: reduce
LR(0) machine (CFSM)

Similar to subset construction (NFA to DFA), except

- State is a set of *dotted items*
  Item is a production with a marker
- Transitions on each possible next symbol
  *shift* (for tokens on RHS)
  *goto* (for non-terminals on RHS)
- *Reduce* action when marker reaches end of production
LR(0) CFSM, Comma lists

\[ S ::= L \$ \]
\[ L ::= L , i \]
\[ L ::= i \]

(Build cfsm on chalkboard)
LR(0) CFSM, Comma lists

\[ S ::= \langle L \rangle \$
\]

\[ L ::= \langle L \rangle , i \]

\[ L ::= \langle i \rangle \]
“Goto” entries

- Goto entries short-circuit *reduce, shift* sequence
  - Instead of shifting non-terminal after reduction, use goto function directly.
- CFSM terminal edges are “shift”, non-terminal edges are “goto”
  - Example: Goto[1,L] is 2
Ambiguity in shift-reduce parsing

Consider:
\[ \langle S \rangle ::= \langle E \rangle \ \$ \]
\[ \langle E \rangle ::= \langle E \rangle + \langle E \rangle \]
\[ \langle E \rangle ::= i \]

(Build cfsm on chalkboard)
Shift-reduce conflict

Conflict (shift-reduce): What do we do on +?

It's a real ambiguity: two different derivations of i + i + i
Removing ambiguity

\[
\langle S \rangle ::= \langle E \rangle \ \$ \\
\langle E \rangle ::= \langle E \rangle + \langle i \rangle \\
\langle E \rangle ::= \ i \\
\]

Note: left recursion is not a problem for LR parsing

(*Build cfsm on chalkboard*)
Choosing productions

- LL(1) parsers must make choice with the *first* token in a RHS.
- LR(0) parsers look at the whole RHS, then decide.
- xLR(1) look at the whole RHS, and one more token beyond.
  - SLR(1): Compute lookahead as *follow* set from the grammar
  - LALR(1): Compute refined *follow* set from a *path through* the grammar (path to current state).
  - LR(1): Compute refined *follow* set from a path, and distinguish states by their *follow* set.

Question: Why is lookahead used only for *reduce* actions?
Adding SLR(1) lookahead

- Comma list grammar is LR(0) (no shift/reduce conflicts)
- In general, lookahead is necessary, even for non-ambiguous grammars.
- SLR solution: Reduce if next token in “follow” set of non-terminal.
SLR(1) lookahead for comma lists

Follow(L) is comma or $
Follow(S)$ is $
Capturing precedence in SLR(1) lookahead

\[ \langle S \rangle ::= \langle E \rangle \ \$ \]
\[ \langle E \rangle ::= \langle E \rangle \ + \ \langle T \rangle \]
\[ \langle E \rangle ::= \ T \]
\[ \langle T \rangle ::= \langle T \rangle \ * \ \ i \]
\[ \langle T \rangle ::= \ i \]

(Build cfsm on chalkboard, tracing \( E + T \times i \)
Use “follow” set to resolve shift/reduce conflict)
LR(1) lookahead

LR(0) items construction \textit{plus}

- Lookahead of start symbol is $\$
- Lookahead propagates \textit{except} for closure
- In closure, we pick up the \textit{following} symbol

Note that LR(1) lookahead for $A$ will always be a subset of \texttt{Follow}(A)
A (simplified) problem from C

\[
\langle \text{decl} \rangle ::= \langle \text{func-decl} \rangle \mid \langle \text{var-decl} \rangle
\]

\[
\langle \text{var-decl} \rangle ::= \langle \text{var-type} \rangle \langle \text{var-list} \rangle ;
\]

\[
\langle \text{var-type} \rangle ::= \text{id}
\]

\[
\langle \text{var-list} \rangle ::= \langle \text{var-list} \rangle , \text{id} \mid \text{id}
\]

\[
\langle \text{func-decl} \rangle ::= \langle \text{func-type} \rangle \langle \text{func-name} \rangle
\]

\[
\text{func-decl} : ( \langle \text{args} \rangle ) ;
\]

\[
\langle \text{func-type} \rangle ::= \text{id} \mid \text{void}
\]

\[
\langle \text{func-name} \rangle ::= \text{id}
\]

(Option: Build cfsm on chalkboard; requires more than one token of lookahead. Fix grammar to use ; , (, and , to resolve reduce/reduce conflict.)
History of Shift-Reduce Parsing

- Until early 70’s, precedence parsing was common.
  - LR(0) was too weak
  - LR(1) was too expensive
- In early 70’s, most switched to SLR(1) and LALR(1)
  (but a few stayed with LL(1) for performance and error recovery)
- LR(k) may eventually become practical, as LL(k) has,
  but don’t hold your breath.