Top-Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

\[
\begin{align*}
E & \rightarrow T + E \mid T \\
T & \rightarrow (E) \mid \text{int} \mid \text{int} \ast T
\end{align*}
\]

The leaves at any point form a string $\beta A\gamma$
- $\beta$ contains only terminals
- The input string is $\beta b\delta$
- The prefix $\beta$ matches
- The next token is $b$
Top-Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

The leaves at any point form a string $\beta A\gamma$
- $\beta$ contains only terminals
- The input string is $\beta b\delta$
- The prefix $\beta$ matches
- The next token is $b$

Predictive Parsing: Review

- A predictive parser is described by a table
  - For each non-terminal $A$ and for each token $b$ we specify a production $A \rightarrow \alpha$
  - When trying to expand $A$ we use $A \rightarrow \alpha$ if $b$ follows next

Once we have the table
- The parsing algorithm is simple and fast
- No backtracking is necessary

Constructing Predictive Parsing Tables

Consider the state $S \rightarrow^* \beta A\gamma$
- With $b$ the next token
- Trying to match $\beta b\delta$

There are two possibilities:
1. Token $b$ belongs to an expansion of $A$
   - Any $A \rightarrow \alpha$ can be used if $b$ can start a string derived from $\alpha$
   - We say that $b \in \text{First}(\alpha)$

Or...
Constructing Predictive Parsing Tables (Cont.)

2. Token \( b \) does not belong to an expansion of \( A \)
   - The expansion of \( A \) is empty and \( b \) belongs to an expansion of \( \gamma \)
   - Means that \( b \) can appear after \( A \) in a derivation of the form \( S \rightarrow^* \beta Ab \)
   - We say that \( b \in \text{Follow}(A) \) in this case

- What productions can we use in this case?
  - Any \( A \rightarrow \alpha \) can be used if \( \alpha \) can expand to \( \epsilon \)
  - We say that \( \epsilon \in \text{First}(A) \) in this case

Computing First Sets

Definition
\[
\text{First}(X) = \{ b \mid X \rightarrow^* b\alpha \} \cup \{ \epsilon \mid X \rightarrow^* \epsilon \}
\]

Algorithm sketch
1. \( \text{First}(b) = \{ b \} \)
2. \( \epsilon \in \text{First}(X) \) if \( X \rightarrow \epsilon \) is a production
3. \( \epsilon \in \text{First}(X) \) if \( X \rightarrow A_1 \ldots A_n \)
   and \( \epsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)
4. \( \text{First}(\alpha) \subseteq \text{First}(X) \) if \( X \rightarrow A_1 \ldots A_n \alpha \)
   and \( \epsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)

First Sets: Example

- Recall the grammar
  \[
  \begin{align*}
  E & \rightarrow T X & X & \rightarrow + E | \epsilon \\
  T & \rightarrow ( E ) | \text{int} & Y & \rightarrow * T | \epsilon
  \end{align*}
  \]

- First sets
  \[
  \begin{align*}
  \text{First}( ( ) ) & = \{ ( ) \} \\
  \text{First}( ( ) ) & = \{ ( ) \} \\
  \text{First}( \text{int} ) & = \{ \text{int} \} \\
  \text{First}( + ) & = \{ + \} \\
  \text{First}( * ) & = \{ * \}
  \end{align*}
  \]

Computing Follow Sets

- Definition
  \[
  \text{Follow}(X) = \{ b \mid S \rightarrow^* \beta X b \delta \}
  \]

- Intuition
  - If \( X \rightarrow A B \) then \( \text{First}(B) \subseteq \text{Follow}(A) \)
    and \( \text{Follow}(X) \subseteq \text{Follow}(B) \)
  - Also if \( B \rightarrow^* \epsilon \) then \( \text{Follow}(X) \subseteq \text{Follow}(A) \)
  - If \( S \) is the start symbol then \( \$ \in \text{Follow}(S) \)
Computing Follow Sets (Cont.)

Algorithm sketch
1. $ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
   - For each production $A \rightarrow \alpha X \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
   - For each production $A \rightarrow \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$

Follow Sets: Example
• Recall the grammar
  
  $E \rightarrow T X$
  $X \rightarrow + E \mid \varepsilon$
  $T \rightarrow ( E ) \mid \text{int} \ Y$
  $Y \rightarrow * T \mid \varepsilon$

  • Follow sets
    $\text{Follow}(+)=\{\text{int, (}\}$
    $\text{Follow}(*)=\{\text{int, (}\}$
    $\text{Follow}(()=\{\text{int, (}\}$
    $\text{Follow}(E)=\{)\}$
    $\text{Follow}(X)=\{+, )\}$
    $\text{Follow}(Y)=\{+,$
    $\text{Follow}(\text{int})=\{*, +,$

Constructing LL(1) Parsing Tables
• Construct a parsing table $T$ for CFG $G$

• For each production $A \rightarrow \alpha$ in $G$ do:
  - For each terminal $b \in \text{First}(\alpha)$ do
    $T[A, b] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $b \in \text{Follow}(A)$ do
    $T[A, b] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    $T[A, \$] = \alpha$

Constructing LL(1) Tables: Example
• Recall the grammar
  
  $E \rightarrow T X$
  $X \rightarrow + E \mid \varepsilon$
  $T \rightarrow ( E ) \mid \text{int} \ Y$
  $Y \rightarrow * T \mid \varepsilon$

  • Where in the line of $Y$ we put $Y \rightarrow* T$?
    - In the lines of $\text{First}(\ast T) = \{\ast\}$

  • Where in the line of $Y$ we put $Y \rightarrow \varepsilon$?
    - In the lines of $\text{Follow}(Y) = \{\$, +,$\}$
Notes on LL(1) Parsing Tables

• If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - And in other cases as well

• For some grammars there is a simple parsing strategy: Predictive parsing
• Most programming language grammars are not LL(1)
• Thus, we need more powerful parsing strategies

Bottom-Up Parsing

• Bottom-up parsing is 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 don't need left-factored grammars and can also handle left-recursive grammars

  • Consider the following grammar:

    \[ E \rightarrow E + (E) | \text{int} \]

  • Consider the string: \text{int} + ( \text{int} ) + ( \text{int} )
The Idea

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

str w input string of terminals
repeat
  - Identify \( \beta \) in str such that \( A \rightarrow \beta \) is a production (i.e., \( \text{str} = \alpha \beta \gamma \))
  - Replace \( \beta \) by \( A \) in str (i.e., \( \text{str} w = \alpha A \gamma \))
until \( \text{str} = S \) (the start symbol)
  OR all possibilities are exhausted

A Bottom-up Parse in Detail (1)

\[
E \rightarrow E + (E) | \text{int}
\]

\[
\text{int + (int) + (int)}
\]

A Bottom-up Parse in Detail (2)

\[
E \rightarrow E + (E) | \text{int}
\]

\[
\text{int + (int) + (int)}
\]

\[
E + (\text{int}) + (\text{int})
\]

A Bottom-up Parse in Detail (3)

\[
E \rightarrow E + (E) | \text{int}
\]

\[
\text{int + (int) + (int)}
\]

\[
E + (\text{int}) + (\text{int})
\]

\[
E + (E) + (\text{int})
\]
Important Fact #1 about Bottom-up Parsing

An LR parser traces a rightmost derivation in reverse
Where Do Reductions Happen

Fact #1 has an interesting consequence:
- Let $\alpha \beta \gamma$ be a step of a bottom-up parse
- Assume the next reduction is by using $A \rightarrow \beta$
- Then $\gamma$ is a string of terminals

Why?
Because $\alpha A \gamma \rightarrow \alpha \beta \gamma$ is a step in a right-most derivation

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Notation

- Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - 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: $I x_1 x_2 \ldots x_n$

Shift

Shift: Move $I$ one place to the right
- Shifts a terminal to the left string

$E + (I \text{ int}) \Rightarrow E + (\text{int } I)$

In general:
$ABC I xyz \Rightarrow ABCx I yz$
**Reduce**

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

- If \( E \rightarrow E + (E) \) is a production, then

\[
E + (E + (E)) \Rightarrow E + (E)
\]

In general, given \( A \rightarrow xy \), then:

\[
Cbxy \Leftrightarrow CbA\]

---

**Shift-Reduce Example**

- \( E \rightarrow E + (E) \mid \text{int} \)
  - \( \text{int} + (\text{int}) + (\text{int}) \) shift
  - \( \text{int} + (\text{int}) + (\text{int}) \) reduce \( E \rightarrow \text{int} \)
  - \( \text{int} + (\text{int}) + (\text{int}) \) shift 3 times

---

**Shift-Reduce Example**

- \( E \rightarrow E + (E) \mid \text{int} \)
  - \( \text{int} + (\text{int}) + (\text{int}) \) shift
  - \( \text{int} + (\text{int}) + (\text{int}) \) reduce \( E \rightarrow \text{int} \)
  - \( \text{int} + (\text{int}) + (\text{int}) \) shift 3 times
Shift-Reduce Example

I int + (int) + (int)$ shift
int I + (int) + (int)$ reduce E \rightarrow int
E I + (int) + (int)$ shift 3 times
E + (int I) + (int)$ reduce E \rightarrow int
E + (E I) + (int)$ shift
E + (E I) + (int)$ reduce E \rightarrow E + (E)
E I + (int)$ shift 3 times

Shift-Reduce Example

E \rightarrow E + (E) | int

\[
\begin{align*}
\text{E} & \rightarrow \text{E} \rightarrow \text{E} + (\text{E}) \\
\text{E} & \rightarrow \text{E} + \text{E} \\
\text{E} & \rightarrow \text{E} + \text{E} \rightarrow \text{E} + \text{E} \\
\text{E} & \rightarrow \text{E} + \text{E} \rightarrow \text{E} + \text{E} \rightarrow \text{E} + \text{E} \\
\end{align*}
\]
Shift-Reduce Example

1. int + (int) + (int)
   - shift
2. int + (int) + (int)
   - reduce E → int
3. E + (int) + (int)
   - shift 3 times
4. E + (int) + (int)
   - reduce E → int
5. E + (E) + (int)
   - shift
6. E + (E) + (int)
   - reduce E → E + (E)
7. E + (E) + (int)
   - shift
8. E + (E) + (int)
   - reduce E → E + (E)
9. E + (int) + (int)
   - shift
10. E + (int) + (int)
    - reduce E → int
11. E + (E) + (int)
    - shift
12. E + (E) + (int)
    - reduce E → E + (E)
13. E + (E) + (int)
    - shift
14. E + (E) + (int)
    - reduce E → E + (E)
15. E + (E) + (int)
    - shift
16. E + (E) + (int)
    - reduce E → E + (E)
17. E + (E) + (int)
    - shift
18. E + (E) + (int)
    - reduce E → E + (E)
19. E + (E) + (int)
    - shift
20. E + (E) + (int)
    - reduce E → E + (E)
21. E + (E) + (int)
    - shift
22. E + (E) + (int)
    - reduce E → E + (E)
23. E + (E) + (int)
    - shift
24. E + (E) + (int)
    - reduce E → E + (E)
25. E + (E) + (int)
    - shift
26. E + (E) + (int)
    - reduce E → E + (E)
27. E + (E) + (int)
    - shift
28. E + (E) + (int)
    - reduce E → E + (E)
29. E + (E) + (int)
    - shift
30. E + (E) + (int)
    - reduce E → E + (E)
31. E + (E) + (int)
    - shift
32. E + (E) + (int)
    - reduce E → E + (E)
33. E + (E) + (int)
    - shift
34. E + (E) + (int)
    - reduce E → E + (E)
35. E + (E) + (int)
    - shift
36. E + (E) + (int)
    - reduce E → E + (E)
37. E + (E) + (int)
    - shift
38. E + (E) + (int)
    - reduce E → E + (E)
39. E + (E) + (int)
    - shift
40. E + (E) + (int)
    - reduce E → E + (E)
41. E + (E) + (int)
    - shift
42. E + (E) + (int)
    - reduce E → E + (E)
43. E + (E) + (int)
    - shift
44. E + (E) + (int)
    - reduce E → E + (E)
45. E + (E) + (int)
    - shift
46. E + (E) + (int)
    - reduce E → E + (E)
47. E + (E) + (int)
    - shift
48. E + (E) + (int)
    - reduce E → E + (E)
49. E + (E) + (int)
    - shift
50. E + (E) + (int)
    - reduce E → E + (E)
51. E + (E) + (int)
    - shift
52. E + (E) + (int)
    - reduce E → E + (E)
53. E + (E) + (int)
    - shift
54. E + (E) + (int)
    - reduce E → E + (E)
55. E + (E) + (int)
    - shift
56. E + (E) + (int)
    - reduce E → E + (E)
57. E + (E) + (int)
    - shift
58. E + (E) + (int)
    - reduce E → E + (E)
59. E + (E) + (int)
    - shift
60. E + (E) + (int)
    - reduce E → E + (E)
61. E + (E) + (int)
    - shift
62. E + (E) + (int)
    - reduce E → E + (E)
63. E + (E) + (int)
    - shift
64. E + (E) + (int)
    - reduce E → E + (E)
65. E + (E) + (int)
    - shift
66. E + (E) + (int)
    - reduce E → E + (E)
67. E + (E) + (int)
    - shift
68. E + (E) + (int)
    - reduce E → E + (E)
69. E + (E) + (int)
    - shift
70. E + (E) + (int)
    - reduce E → E + (E)
71. E + (E) + (int)
    - shift
72. E + (E) + (int)
    - reduce E → E + (E)
73. E + (E) + (int)
    - shift
74. E + (E) + (int)
    - reduce E → E + (E)
75. E + (E) + (int)
    - shift
76. E + (E) + (int)
    - reduce E → E + (E)
77. E + (E) + (int)
    - shift
78. E + (E) + (int)
    - reduce E → E + (E)
79. E + (E) + (int)
    - shift
80. E + (E) + (int)
    - reduce E → E + (E)
81. E + (E) + (int)
    - shift
82. E + (E) + (int)
    - reduce E → E + (E)
83. E + (E) + (int)
    - shift
84. E + (E) + (int)
    - reduce E → E + (E)
85. E + (E) + (int)
    - shift
86. E + (E) + (int)
    - reduce E → E + (E)
87. E + (E) + (int)
    - shift
88. E + (E) + (int)
    - reduce E → E + (E)
89. E + (E) + (int)
    - shift
90. E + (E) + (int)
    - reduce E → E + (E)
91. E + (E) + (int)
    - shift
92. E + (E) + (int)
    - reduce E → E + (E)
93. E + (E) + (int)
    - shift
94. E + (E) + (int)
    - reduce E → E + (E)
95. E + (E) + (int)
    - shift
96. E + (E) + (int)
    - reduce E → E + (E)
97. E + (E) + (int)
    - shift
98. E + (E) + (int)
    - reduce E → E + (E)
99. E + (E) + (int)
    - shift
100. E + (E) + (int)
    - reduce E → E + (E)
The Stack

- Left string can be implemented by a stack
  - Top of the stack is the $I$

- Shift pushes a terminal on the stack

- Reduce pops 0 or more symbols off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS)

Key Question: To Shift or to Reduce?

Idea: use a finite automaton (DFA) to decide when to shift or reduce

- The input is the stack
- The language consists of terminals and non-terminals

- We run the DFA on the stack and we examine the resulting state $X$ and the token $tok$ after $I$
  - If $X$ has a transition labeled $tok$ then shift
  - If $X$ is labeled with “$A \rightarrow \beta$ on tok” then reduce

LR(1) Parsing: An Example

Representing the DFA

- Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for terminals: action table
  - Those for non-terminals: goto table
Representing the DFA: Example

- The table for a fragment of our DFA:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>+</th>
<th>( )</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>s4</td>
<td>g6</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>rE</td>
<td></td>
<td></td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>s8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>rE→E+(E)</td>
<td>rE→int</td>
</tr>
</tbody>
</table>

E → E + (E) on $, +
E

The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated

- Remember for each stack element on which state it brings the DFA

- LR parser maintains a stack
  
  \[ (\text{sym}_1, \text{state}_1) \ldots (\text{sym}_n, \text{state}_n) \]

  \text{state}_k is the final state of the DFA on sym_1 \ldots sym_k

The LR Parsing Algorithm

let I = w$ be initial input
let j = 0
let DFA state 0 be the start state
let stack = \langle \text{dummy, 0} \rangle

repeat
  case action[top_state(stack), I[j]] of
    shift k: push \langle I[j++], k \rangle
    reduce X → A:
      pop |A| pairs,
      push \langle X, \text{Goto}[top_state(stack), X] \rangle
    accept: halt normally
    error: halt and report error