Bottom-up parsing

• Bottom-up parsing builds a parse tree from the leaves (terminals) to the start symbol

\[
E \rightarrow T + E \mid T \\
T \rightarrow \text{int} \ast T \mid \text{int}
\]

Rightmost Derivation

\[
E \Rightarrow T + E \\
\Rightarrow T + T \\
\Rightarrow T + \text{int} \\
\Rightarrow \text{int} \ast T + \text{int} \\
\Rightarrow \text{int} \ast \text{int} + \text{int}
\]

Leftmost Derivation

\[
E \Rightarrow T + E \\
\Rightarrow \text{int} \ast T + E \\
\Rightarrow \text{int} \ast \text{int} + E \\
\Rightarrow \text{int} \ast \text{int} + T \\
\Rightarrow \text{int} \ast \text{int} + \text{int}
\]
### Bottom-up parsing II

- Bottom-up parsing is a series of reductions (inverses of productions), the reverse of which is the rightmost derivation

<table>
<thead>
<tr>
<th>Reductions</th>
<th>Rightmost Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) int --&gt; T</td>
<td>E --&gt; T + E</td>
</tr>
<tr>
<td>(2) int * T --&gt; T</td>
<td>T --&gt; T + E</td>
</tr>
<tr>
<td>(3) int --&gt; T</td>
<td>=&gt; E + T</td>
</tr>
<tr>
<td>(4) T --&gt; E</td>
<td>=&gt; E + int</td>
</tr>
<tr>
<td>(5) T + E --&gt; E</td>
<td>=&gt; int * T + int</td>
</tr>
</tbody>
</table>

- Explanation in terms of reverse rightmost derivations is correct but not terribly intuitive
- In particular, connection between top-down and bottom-up parsing becomes obscured
- More intuitive explanation – in terms of transition diagram of grammar

### Example

- Consider the grammar
  
<table>
<thead>
<tr>
<th>Production</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E --&gt; E</td>
<td>(1)</td>
</tr>
<tr>
<td>E --&gt; int</td>
<td>(2)</td>
</tr>
<tr>
<td>E --&gt; (E+E)</td>
<td>(3)</td>
</tr>
<tr>
<td>E --&gt; (E-E)</td>
<td>(4)</td>
</tr>
</tbody>
</table>

- This grammar is not LL(1)
  - although it can be massaged into a LL(1) grammar
  - Key problem with recursive-descent parser:
    - when we are trying to parse an "E" from the input, parser needs to decide whether to apply production (2), (3) or (4) by just looking at next terminal symbol
    - if next terminal is '+', not clear whether to apply (3) or (4)
- Bottom-up parsing
  - instead of making this decision at the start of the RHS of E productions, delay this decision till we get to end of RHS of productions
  - in this case, it is clear that delaying the decision will let us figure out the right rule unambiguously when we encounter either '+' or '-' somewhere along the way

### Intuition

- Explanation in terms of reverse rightmost derivations is correct but not terribly intuitive
- In particular, connection between top-down and bottom-up parsing becomes obscured
- More intuitive explanation – in terms of transition diagram of grammar
**Bottom-up parsing**

- Idea: think of transition diagram as the state diagram for a non-deterministic automaton
- Automaton follows “all possible paths” till it either accepts or rejects, thereby delaying decision of which production to apply
- Implementation:
  - make the dashed lines in the transition diagram into $\epsilon$ transitions for the automaton

---

**Transition diagram with $\epsilon$ transitions**

---

**Constructing an LR(0) automaton**

1. Add a dummy start symbol [$S' \rightarrow S$]
   - Distinguishes accepting reductions
2. Make an automaton for each production
3. For transitions on non-terminals, add $\epsilon$-edges to the corresponding automaton
4. Apply NFA to DFA conversion

---

**LR(0) items**

- Alternate construction of LR(0) automaton
  - it is actually easy to construct the deterministic automaton directly rather than construct the non-deterministic automaton first, and then convert it to a deterministic one
- A LR(0) item is
  - A production, $A \rightarrow X Y Z$, with a dot in the body, e.g., $A \rightarrow . X Y Z$
  - Represents state of parser
    - $A \rightarrow X : Y Z$ means that the parser has seen $X$ so far and is looking for a string derivable from $Y Z$
**LR(0) items**

- **CLOSURE(I : item set)**
  - \( I \subseteq \text{CLOSURE}(I) \)
  - **If**
    - \([A \to \alpha \cdot B \beta] \in \text{CLOSURE}(I) \) and
    - \([B \rightarrow \gamma] \)
  - **Then**
    - \([B \rightarrow \gamma] \in \text{CLOSURE}(I) \)
    - “If we’re looking for \( B \beta \) and \( B \rightarrow \gamma \) then we should also be looking for \( \gamma \)”

- **GOTO(I, X : symbol)**
  - If \([A \rightarrow \alpha \cdot X \beta] \in I \) then \([A \rightarrow \alpha X \cdot \beta] \in \text{GOTO}(I, X) \)
  - \( \text{CLOSURE}(\text{GOTO}(I, X)) \subseteq \text{GOTO}(I, X) \)
  - “If we’re in state I and see symbol X, we are now in state I’”

**Deterministic automaton (LR(0))**

**Shift-reduce parsing**

- **Shift \( S_m \):** Push \( S_m \) on stack; increment input position

**Shift-reduce parsing**

- **Reduce \( A \rightarrow \beta \):** Pop |\( \beta \) symbols; push Goto[\( S_{m-\mid \beta \mid} \cdot A \)] on stack
**LR(0) parsing table**

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>+</td>
<td>- $</td>
</tr>
<tr>
<td>1</td>
<td>S3</td>
<td>S4</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>3</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>4</td>
<td>S3</td>
<td>S4</td>
</tr>
<tr>
<td>5</td>
<td>S6</td>
<td>S9</td>
</tr>
<tr>
<td>6</td>
<td>S3</td>
<td>S4</td>
</tr>
<tr>
<td>7</td>
<td>S8</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>9</td>
<td>S3</td>
<td>S4</td>
</tr>
<tr>
<td>10</td>
<td>S11</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example**

Stack | Input | Action
--- | --- | ---
1 | (int + (int – int))$ | s4
1 | (4 | int + (int – int)$ | s3
1 | 4 int 3 | + (int – int)$ | r2
1 | 4 E 5 | + (int – int)$ | s6
1 | 4 E 5 + 6 | (int – int)$ | s4
1 | 4 E 5 + 6 (4 | int – int)$ | s3
1 | 4 E 5 + 6 (4 int 3 | - int)$ | r2
1 | 4 E 5 + 6 (4 E 5 | - int)$ | s9
1 | 4 E 5 + 6 (4 E 5 – 9 | int)$ | s3
1 | 4 E 5 + 6 (4 E 5 – 9 int 3 | )$ | r2
1 | 4 E 5 + 6 (4 E 5 – 9 E 10 | )$ | s11
1 | 4 E 5 + 6 (4 E 5 – 9 E 10) 11 | )$ | r4
1 | 4 E 5 + 6 E 7 | )$ | s8
1 | 4 E 5 + 6 E 7 | )$ | s2
1 E | 2 | )$ | acc

**Conflicts in LR(0) table**

- When the grammar is not LR(0), there are conflicts in the LR(0) parsing table
  - shift-reduce conflicts: in some state, it is possible to perform both a shift and a reduce
  - reduce-reduce conflicts: in some state, two or more reductions can be applied

  - Example: S → aB | Ac
    - A → ab
    - B → b

**Example**

E: \[
\begin{array}{ccc}
  & T & X \\
  E: & ( & E \\
\end{array}
\]

T: \[
\begin{array}{ccc}
  & \text{int} & Y \\
  E: & ( & E \\
\end{array}
\]

X: \[
\begin{array}{ccc}
  + & & E \\
  X: & + & E \\
\end{array}
\]

Y: \[
\begin{array}{ccc}
  * & & T \\
  Y: & * & T \\
\end{array}
\]

E → TX
T → (E) | int Y
X → + E | ε
Y → * T | ε
LR(0) automaton

- Add $\epsilon$-transitions to indicate possible parsing states
- Apply NFA to DFA conversion

Shift-reduce conflicts

- When do we apply $\epsilon$ reductions?

LR(k) grammars

- Recognize most programming language constructs
  - LR(k) recognizes the body of a production in right-sentential form with k symbols of lookahead
  - Determine when to apply reductions, $A \rightarrow \beta$, given string $\delta_0 \cdots \delta_n$
  - LL(k) recognizes the use of a production after seeing the first k symbols of what the body derives
  - Determine when to apply productions, $A \rightarrow a_1 \cdots a_k \beta$, given string $\delta_0 \cdots \delta_n$
- LR(k) is a proper superset of LL(k)
- However, tables can get very large, so we usually use tricks to add some look-ahead to LR(0) automaton

SLR(1) parser

- When do we apply $\epsilon$ reductions?
  - when look-ahead token is in FOLLOW set of non-terminal
SLR(1) parser

• Generate Action table from automaton
  – For each edge $S_i \rightarrow S_j$ in LR(0) automaton,
    $\text{Action}[S_i, a] = \text{shift } S_j$
  – For each "reduce" node $S_i$ with reduction $[A \rightarrow \beta]$,
    $\text{Action}[S_i, a] = \text{reduce } A \rightarrow \beta$ \text{ where } a \in \text{FOLLOW}(A)
    • Exception: If the node corresponds to the reduction $S' \rightarrow S$, $\text{Action}[S_i, $] = \text{accept}$
  – All other actions are error
  – Conflict between actions $\Rightarrow$ grammar not SLR

• Generate Goto table from automaton
  – For each edge $S_i \rightarrow S_j$ in LR(0) automaton,
    $\text{Goto}[S_i, A] = S_j$

SLR(1) parsing example

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S9</td>
<td>S6</td>
</tr>
<tr>
<td>2</td>
<td>R5</td>
<td>S4</td>
</tr>
<tr>
<td>3</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>4</td>
<td>S9</td>
<td>S6</td>
</tr>
<tr>
<td>5</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>6</td>
<td>S9</td>
<td>S6</td>
</tr>
<tr>
<td>7</td>
<td>S8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>9</td>
<td>R7</td>
<td>R7</td>
</tr>
<tr>
<td>10</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>11</td>
<td>S9</td>
<td>S6</td>
</tr>
<tr>
<td>12</td>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>13</td>
<td>acc</td>
<td></td>
</tr>
</tbody>
</table>

Problem with SLR(1)

LR(0) automaton
Finding look-aheads in LR(0) graph

Easy to see that if we are in state 1 and reduction $R \rightarrow L$ is valid, look-ahead symbol cannot be $=$.

Parser moves:
- Pop state 1 from stack
- Topmost state must be state 0
- Take $R$ transition: push state 2
- Pop state 2 and take $S$ transition to 3
- Only look-ahead symbol is $\$. You can figure this out by tracing paths in graph → intuition behind LALR(1) grammars

LALR(1) grammars

- Generate LR(0) automaton.
- If there are no conflicts, done.
- Otherwise, figure out look-ahead sets for reductions by tracing paths backwards (and forwards) in the automaton graph.
- Doing this systematically:
  - set up a system of constraints as shown in next slide
  - reason is that these sets depend on each other as in the example
  - look-ahead set for $R \rightarrow L$ depended on look-ahead set for $S \rightarrow R$
  - which in turn depended on look-ahead set for $0 \rightarrow 3$ transition
  - solve them

Rules for LALR(1) constraints

Look-ahead($q,A$) $\subseteq$ Look-ahead($p,A \rightarrow \gamma$)
FIRST($\gamma$) $\subseteq$ Look-ahead ($q,A$)
If $\gamma \rightarrow \epsilon$, Look-ahead($r,B \rightarrow \gamma A \rightarrow \epsilon$) $\subseteq$ Look-ahead($q,A$)

Example

\[ S \rightarrow SS \]
\[ S \rightarrow LR \]
\[ S \rightarrow R \]
\[ L \rightarrow \ast R \]
\[ L \rightarrow \text{id} \]
\[ R \rightarrow \text{id} \]
\[ \$ \rightarrow \$ \]
LALR(1) grammars in practice

- yacc, CUP: parser-generators for LALR(1) grammars
- Most PL constructs can be expressed reasonably in an LALR(1) grammar
- That fact and the availability of tools like yacc and CUP have made LALR(1) grammars pretty much the default
- More powerful grammars like LR(1) are described in textbooks but rarely used in practice

Dealing with ambiguity

- Commonly, parser generators allow methods for dealing with ambiguous grammars
  - Precedence and associativity rules for operators
  - Implemented by generating suitable parser table

Example

```
E → int
E → E + E
E → E * E
E → E . E
E → int
E → E + E
E → E * E
```

Grammar is ambiguous but ambiguity is resolved using operator priority and precedence.

JFlex and CUP

- JFlex, a lexer generator for Java
- CUP, a parser generator for Java
- Both take specifications and generate Java code
JFlex

```java
import java_cup.runtime.Symbol;

private Symbol symbol(int sym) { return new Symbol(sym, yyline+1, yycolumn+1); }
private Symbol symbol(int sym, Object val) { return new Symbol(sym, val); }

IntLiteral = 0 | [1-9][0-9]*
new_line = \r|\n|\r\n;
white_space = {new_line} | [\t\f]

{IntLiteral}      { return symbol(sym.INT, new Integer(Integer.parseInt(yytext()))); }
"("               { return symbol(sym.LPAREN); }
")"               { return symbol(sym.RPAREN); }
"+"               { return symbol(sym.PLUS); }
...
{white_space}     { /* ignore */ }
./|n              { error("Illegal character <" + yytext() + ">;" ); }
```

CUP

```java
/* Terminals (tokens returned by lexer). */
terminal PLUS, MINUS, SLASH, STAR, QUESTION, COLON, LPAREN, RPAREN;
terminal Integer INT;
non terminal Integer Exp;
precedence left QUESTION;
precedence left PLUS, MINUS;
precedence left STAR, SLASH;
Exp ::= INT:i               {: RESULT = i; :}
| Exp:e1 PLUS Exp:e2  {: RESULT = e1 + e2; :}
| Exp:e1 MINUS Exp:e2 {: RESULT = e1 - e2; :}
| Exp:e1 SLASH Exp:e2 {: RESULT = e1 / e2; :}
| Exp:e1 STAR Exp:e2  {: RESULT = e1 * e2; :}
| Exp:e1 QUESTION Exp:e2 COLON Exp:e3 {: RESULT = e1 == 0 ? e3 : e2; :}
| LPAREN Exp:e1 RPAREN {: RESULT = e1; :}
```

LR(1)

- Use k = 1 lookahead symbols to determine when to shift rather than reduce
  - Reduce only when we have a matching lookahead
  - The set of lookahead symbols for A is some subset of FOLLOW(A)
- Use LR(0) automaton to give intuition about LR(1)
LR(1) example

\[
\begin{align*}
S &\rightarrow S \ \$ \\
S &\rightarrow L = R \mid R \\
L &\rightarrow ^* R \mid id \\
R &\rightarrow L \rightarrow _L \\
S' &\rightarrow S \ \$
\end{align*}
\]

Constraints
- FOLLOW(S) = { $ }
- FOLLOW(R) = { =, $ }
- FOLLOW(L) = { =, $ }

Solutions
- FIRST($) ⊆ FOLLOW(S)
- FIRST(=) ⊆ FOLLOW(L)
- FOLLOW(R) ⊆ FOLLOW(L)
- FOLLOW(S) ⊆ FOLLOW(R)
- FOLLOW(L) ⊆ FOLLOW(R)

Recap
1. Use “context” of ε-moves to introduce states corresponding to the terminal(s) we expect to see after non-terminal
   - State dependent FOLLOW
   - Subset of FOLLOW
2. Propagate lookahead to reduction rules
3. Perform NFA to DFA conversion
LR(1) items

- Equivalence between LR(1) automaton and LR(1) item sets
- A LR(1) item is
  - An LR(0) item augmented with a lookahead symbol (terminal), e.g., [A → . X Y Z, a]
  - The item [A → X Y Z ., a] calls for a reduction only if the next input symbol is a

LR(1) items

- CLOSURE(I : item set)
  - I ⊆ CLOSURE(I)
  - If
    - [A → α . B β, a] ∈ CLOSURE(I),
    - [B → γ], and
    - b ∈ FIRST(β a)
  - Then
    - [B → γ ., b] ∈ CLOSURE(I)
  - “If we’re looking for B β and B → γ then we should also be looking for γ”
- GOTO(I, X : symbol)
  - If [A → α . X β, a] ∈ I then [A → α X ., β, a] ∈ GOTO(I, X)
  - CLOSURE(GOTO(I, X)) ⊆ GOTO(I, X)
  - “If we’re in state I and see symbol X, we are now in state I’”

LR(1) parser

- Generate Action table from automaton
  - For each edge S_i → a S_j in LR(1) automaton, Action[S_i,a] = shift S_j
  - For each “reduce” node S_i with reduction [A → β ., a], Action[S_i,a] = reduce A → β
    - Exception: If the node corresponds to the reduction [S → S, $], then Action[S, $] = accept
  - All other actions are error
  - If there is a conflict between actions, grammar is not in LR(1)
- Goto table generated as in SLR(1)