Bottom Up Parsing

- Builds syntax tree as rightmost derivation
  - Like postorder traversal (leaves visited before nodes)
- Bottom up parsing algorithms more powerful than top down methods
  - Left recursion not a problem
  - But constructions much more complex
  - Methods are suitable for parser generators
- "Real" grammars too complex to construct tables by hand, but we will look at very simple examples
- LR(0), SLR(1), LALR(1), LR(1) parsing

Bottom Up Parsing

- Parsing is table driven (no recursion)
- Uses a stack (like LL(1) parser)
  - Stack has terminals and non-terminals, and states
  - Extra token $ added for bottom of stack / end of input
- Goal is to end up with stack containing just start symbol, and all input consumed (accept)
- Two other actions
  - Shift – move a terminal from input to top of stack
  - Reduce – use a rule to replace a string at top of stack by a non-terminal
- Called shift-reduce parsers
Bottom Up Example

Balanced parentheses:

\[ S' \rightarrow S \]
\[ S \rightarrow ( S ) \mid \varepsilon \]

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>( ) $</td>
<td>shift</td>
</tr>
<tr>
<td>$(</td>
<td>) $</td>
<td>reduce $ S \rightarrow \varepsilon</td>
</tr>
<tr>
<td>$( S )</td>
<td>) $</td>
<td>shift</td>
</tr>
<tr>
<td>$( S ) S</td>
<td>$</td>
<td>reduce $ S \rightarrow \varepsilon</td>
</tr>
<tr>
<td>$ S</td>
<td>$</td>
<td>reduce $ S \rightarrow ( S ) S</td>
</tr>
<tr>
<td>$ S'</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Notes on Bottom Up Parsing

- Grammar augmented with new start symbol
- Lookahead is less of a problem
  - Input shifted onto stack, and we can look through stack
  - Uses DFA to decide (but the parser is not a DFA)
  - Some lookahead may be needed
- Rightmost derivation traced in reverse order
- Mechanism is to shift input to stack until it can be reduced
  - Choice for reduction based on right hand side of rules
- Right recursion makes stack grow larger
**LR(0) Items**

- An **LR(0) item** is a grammar rule along with a position in the right hand side (indicated by a dot).

  Rule: \( S \rightarrow ( \ S \ ) \ S \)

  Items: \( S \rightarrow \bullet ( \ S \ ) \ S \)
  
  \( S \rightarrow ( \bullet \ S \ ) \ S \)
  
  \( S \rightarrow ( \ S \bullet ) \ S \)
  
  \( S \rightarrow ( \ S \ ) \bullet \ S \)
  
  An item records an intermediate step, i.e., the division between what is on the stack and the input left to process.

---

**NFA of LR(0) Items**

- Transition for each movement of dot

  \[ A \rightarrow \alpha \bullet X \beta \xrightarrow{X} A \rightarrow \alpha X \bullet \beta \]

  - Corresponds to shift
  - Pushes symbol on stack
  
  - \( \varepsilon \) transition to initial item for each non-terminal

  \[ A \rightarrow \alpha \bullet X \beta \xrightarrow{\varepsilon} X \rightarrow \bullet \gamma \]

  - Corresponds to reduce, pushing non-terminal on stack
More on the NFA

- We need a unique start state, so augment with rule that has single production
- Purpose of NFA is to track the state of the parse, not recognize the string, so no need for accept states
  - Parser decides when to accept
- The NFA tells parser what to do next
- Convert NFA to DFA using subset construction
Computing the DFA

- Add the augmentation item $S' \rightarrow \cdot S$ to the start state of the DFA. Then add all the initial items for $S$ to the state (i.e., $S \rightarrow \cdot \alpha$). Continue by adding all the initial items for those non-terminals which appear right after the dot in any previous item (this is called the closure of the set of items).
- Every symbol that comes immediately after the dot gives rise to a transition to a state generated by adding closure items to the item with the dot moved past that symbol.

Constructing the DFA
LR(0) Parsing Algorithm

- Add states to parsing stack
  - Push state number on stack after symbol is pushed
  - Start with $ and state 0
- Parser chooses action based on current state
  - If state contains any item with dot before terminal, then shift. If token has such item, then push state that token transitions to. If no such item, error.
  - If state has complete item (dot at end), reduce by that rule. Pop string of rule, uncovering state which contains item leading to reduction. Push reduced symbol and state it transitions to. Reduction by $S' \rightarrow S$ means accept.

Conflicts

- Algorithm seems fuzzy. What if several items?
- If a state contains a complete item and a shift item, it is a shift-reduce conflict
  - Can't decide between shift and reduce
- If a state contains two complete items, it is a reduce-reduce conflict
  - Can't decide which way to reduce
- No shift-shift conflicts – input decides
Definition of LR(0)

- A grammar is LR(0) if the rules of the algorithm are unambiguous.
  - If a state contains a complete item, then it cannot contain any other items
- Our example is not LR(0)!

DFA for $S \rightarrow (S)S / \varepsilon$

<table>
<thead>
<tr>
<th>State</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>$S \rightarrow S$</td>
</tr>
<tr>
<td>S</td>
<td>$S \rightarrow S \cdot 1$</td>
</tr>
<tr>
<td>S</td>
<td>$S \rightarrow S \cdot 2$</td>
</tr>
<tr>
<td>S</td>
<td>$S \rightarrow S \cdot 3$</td>
</tr>
<tr>
<td>S</td>
<td>$S \rightarrow S \cdot 4$</td>
</tr>
<tr>
<td>S</td>
<td>$S \rightarrow S \cdot 5$</td>
</tr>
</tbody>
</table>

Shift-reduce conflict
Another Example

- Let's try again with simpler grammar:
  \[ A \rightarrow (A) \mid a \]
- (Note this requires a PDA)
- First augment and number rules:
  1. \( A' \rightarrow A \)
  2. \( A \rightarrow (A) \)
  3. \( A \rightarrow a \)

Construct the DFA

```
0
A' → • A
A → •(A)
A → • a

1
A' → A •

2
A → a •

3
A → (• A)
A → •(A)
A → • a

4
A → (A •)

5
A → (A) •
```
### Parsing Actions

Parse the string: \(( (a) )\)

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>( (a) ) $</td>
<td>shift</td>
</tr>
<tr>
<td>$0(3)$</td>
<td>(a) ) $</td>
<td>shift</td>
</tr>
<tr>
<td>$0(3a)$</td>
<td>) $</td>
<td>shift</td>
</tr>
<tr>
<td>$0(A)$</td>
<td>) $</td>
<td>reduce A → a</td>
</tr>
<tr>
<td>$0(A)$</td>
<td>) $</td>
<td>shift</td>
</tr>
<tr>
<td>$0(A)$</td>
<td>$</td>
<td>reduce A → (A)</td>
</tr>
<tr>
<td>$0A$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

### LR(0) Parsing Table

- Each state is reduce or shift
  - If Reduce, give rule
  - If Shift, give new state for input (or non-terminal)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Rule</th>
<th>Input</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>shift</td>
<td>3</td>
<td>(a)</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>reduce</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>reduce</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>shift</td>
<td>3</td>
<td>(a)</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>shift</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>reduce</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parsing Conflicts

- **Shift-reduce** conflicts: almost always come from ambiguities, and almost always the right disambiguating rule is to shift (dangling-else).
- **Reduce-reduce** conflicts are more difficult; bottom-up parsers try to resolve them using Follow contexts.

Lookahead

- LR(1) uses lookahead, but first we look at a simpler form: SLR(1)
- Uses DFA of items for LR(0)
  - But looking at next token increases power
  - Checks next token before shift
  - Checks Follow set to decide to reduce
- Lookahead allows most languages to parse
SLR(1) Parsing Algorithm

1. If state contains any item with dot before terminal, and next token matches, then push state that token transitions to.
2. If state has complete item and next token is in Follow set, reduce by that rule. Pop string of rule, uncovering state which contains item leading to reduction. Push reduced symbol and state it transitions to.
3. Otherwise error
   - That is, use lookahead and
     - Choose shift over reduce
     - Use Follow set to choose among reductions

Definition of SLR(1)

- SLR(1) if no ambiguity in algorithm
- Equivalent conditions
  1. For any item $A \rightarrow \alpha \bullet x \beta$ in a state, there can be no complete item $B \rightarrow \gamma \bullet$ in the state where $x$ is in Follow(B)
  2. Intersection of Follow sets of any two complete items in a state is empty
- Otherwise we have (1) shift-reduce or (2) reduce-reduce conflicts
DFA for $S \rightarrow ( S ) S / \varepsilon$

Follow($S'$) = \{ $ \}$, Follow($S$) = \{ $, ) \}$

SLR(1) Parsing Table

- Grammar Rules  1. $S' \rightarrow S$  2. $S \rightarrow ( S ) S$  3. $S \rightarrow \varepsilon$

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(</td>
<td>$S$</td>
</tr>
<tr>
<td>1</td>
<td>$S'$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>s2</td>
<td>r3</td>
</tr>
<tr>
<td>3</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>4</td>
<td>s2</td>
<td>r3</td>
</tr>
<tr>
<td>5</td>
<td>r2</td>
<td>r2</td>
</tr>
</tbody>
</table>
SLR(1) Example

Parse the string: () ()

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>()</td>
<td>shift to state 2</td>
</tr>
<tr>
<td>$0 ( 2 S 3 )$</td>
<td>()</td>
<td>reduce by rule 3</td>
</tr>
<tr>
<td>$0 ( 2 S 3 )$</td>
<td>()</td>
<td>shift to state 4</td>
</tr>
<tr>
<td>$0 ( 2 S 3 )$</td>
<td>()</td>
<td>reduce by rule 3</td>
</tr>
<tr>
<td>$0 ( 2 S 3 )$</td>
<td>()</td>
<td>shift to state 4</td>
</tr>
<tr>
<td>$0 ( 2 S 3 )$</td>
<td>()</td>
<td>reduce by rule 3</td>
</tr>
<tr>
<td>$0 ( 2 S 3 )$</td>
<td>()</td>
<td>reduce by rule 2</td>
</tr>
<tr>
<td>$0 ( 2 S 3 )$</td>
<td>()</td>
<td>reduce by rule 2</td>
</tr>
<tr>
<td>$0 ( 2 S 3 )$</td>
<td>()</td>
<td>reduce by rule 2</td>
</tr>
<tr>
<td>$0 ( 2 S 3 )$</td>
<td>()</td>
<td>accept</td>
</tr>
</tbody>
</table>

Notes on SLR(1) Parsing

- Stack grows with right recursion
  - May cause stack overflow
- Natural disambiguating rule for shift-reduce
  - Prefer shift over reduce
  - Produces correct results for if-else
- Reduce-reduce usually an error in grammar
- Generalize to SLR(k)
Another example

- Expressions and pointer dereference in C
  \[ S \rightarrow V = E \]
  \[ S \rightarrow E \]
  \[ E \rightarrow V \]
  \[ V \rightarrow \text{id} \]
  \[ V \rightarrow *E \]
  \[ \text{Follow}(V) = \text{Follow}(E) = \{ $, = \} \]

LR(1) and LALR(1) Parsing

- SLR not powerful enough for some grammars
- LR(1) parsing
  - Increased complexity (large tables)
- LALR(1) – lookahead LR parsing
  - Efficient like SLR, works almost as well as general LR(1)
  - Used by yacc and other parser generators
LR(1) Items

- SLR uses DFA of LR(0) items
  - lookahead used after DFA created
- LR(1) items add lookahead
  - usual LR(0) item plus a lookahead token
- Transitions same as LR(0) except
  - \( \epsilon \) transitions from item \([A \rightarrow \alpha \cdot B \gamma, a]\) to all items \([B \rightarrow \beta \cdot x]\) where \(x\) is in \( \text{First}(\gamma a) \)
  - this keeps track of recognition context for \(B\)

Partial LR(1) example

\[
\begin{align*}
S' & \rightarrow S \cdot, \$ \\
S & \rightarrow V = E, \$ \\
S & \rightarrow \cdot E, \$ \\
V & \rightarrow \cdot id, \$, \$ \\
V & \rightarrow \cdot \cdot E, \$, \$ \\
E & \rightarrow \cdot V, \$ \\
V & \rightarrow \cdot id, \$, \$ \\
V & \rightarrow \cdot \cdot E, \$, \$ \\
V & \rightarrow \cdot \cdot \cdot E, \$, \$ \\
E & \rightarrow \cdot V, \$ \\
E & \rightarrow \cdot id, \$, \$ \\
E & \rightarrow \cdot \cdot E, \$, \$ \\
V & \rightarrow \cdot \cdot \cdot E, \$, \$ \\
S & \rightarrow V = E, \$ \\
S & \rightarrow E \\
E & \rightarrow V \\
V & \rightarrow \cdot id \\
V & \rightarrow \cdot \cdot E \\
S & \rightarrow V = E \\
S & \rightarrow E \\
E & \rightarrow V \\
V & \rightarrow \cdot id \\
V & \rightarrow \cdot \cdot E \\
E & \rightarrow \cdot V \\
V & \rightarrow \cdot id \\
V & \rightarrow \cdot \cdot E \\
V & \rightarrow \cdot \cdot \cdot E
\end{align*}
\]

First(V)=First(E)=First(S)= \{id, \*\}
Follow(V)=Follow(E)=\{\$, =\}
**LR(1) Notes**

- DFA can be very large
  - Lots of states because of lookahead
- Parsing (and parse table construction) is the same as for SLR except
  - Use lookahead in state instead of Follow sets
- A grammar is LR(1) if for each state
  - No shift of terminal and complete item with terminal as lookahead
  - No two complete items with same lookahead

---

**What happens to SLR conflict?**

No conflict:

- `'='` indicates shift
- `'$'` indicates reduce

First(V)=First(E)=First(S)= \{id, *\}
Follow(V)=Follow(E)=\{$, =\}
LALR(1)

- Observe that many states differ only by lookahead in LR(1) items
- Combine all lookaheads and fold states (example: 3 & 5)
- DFA will "look like" the LR(0) DFA – but have lookaheads
- Use same parsing algorithm as for LR(1)
- Much smaller DFA, much smaller table
- Most LR(1) languages are LALR(1)
- LALR used for parser generators – yacc & bison

Folding LR(1) States

- Differ only by lookahead, so fold