Tutorial 7

- Bottom-Up Parsing (LR(1) & SLR(1))

**Bottom-Up Parsing**

- Top-down parsing is ill-suited for left-recursive grammars
  - Big problem: almost all programming languages are left-associative

- Bottom-Up parsing
  - Given a CFG grammar \( G = (N, \Sigma, P, S) \) & an input string \( x \in \Sigma^* \), determine if \( x \in L(G) \)
  - To show \( x \in L(G) \), show from \( x \) we can work backwards to find a derivation path \( \langle x_0, x_1, ..., x_n \rangle \) until we reach our start symbol \( S' \)
    \[ \Rightarrow \quad x \leq x_0 \leq x_{n-1} \leq \ldots \leq x_0 \leq S' \]
    \[ \uparrow \quad \text{Start at } x \quad \text{Find } x_n \ldots \quad \text{until we find } S' \]

- LR Parsing
  - Start with input string \( x' = 1 \cdot x^{-1} \), \( x \in \Sigma^* \)
  - A symbols in \( x' \) we do one of the following:
    1. **Shift**: Consume the next input symbol & push it to the stack
    2. **Reduce**: If we recognize the right hand side \( B \) of a rule \((A \rightarrow B)\), then pop the right hand side of the rule \((B)\) off the stack & push the left hand side onto the stack \((A)\)
      - Includes rules where \( A \rightarrow \varepsilon \) (pushes \( A \) onto the stack, no pop)
Consider the CFG:

0. \( S^3 \rightarrow \cdot S \cdot \)
1. \( X \rightarrow px \)
2. \( Y \rightarrow e \)
3. \( S \rightarrow S \cdot a \cdot b \)
4. \( X \rightarrow e \)
5. \( S \rightarrow x \cdot y \)
6. \( Y \rightarrow \cdot q \)

**Defn:** An item is a production with a bookmark (denoted \( \cdot \)) somewhere on the right hand side (RHS) of a rule.

- \( L \rightarrow S^3 \rightarrow \cdot S \cdot \) is a fresh item; none of the RHS is on the stack.
- If we push \( \cdot \) on the stack, the rule updates to \( S^3 \rightarrow \cdot \cdot S \cdot \) to tell us \( A \) is on the stack.
- Continue pushing symbols until \( S^3 \rightarrow \cdot S \cdot \cdot \), entire RHS of the rule is on the stack \( \rightarrow \) Reducible to \( S \)!
- We can represent positions of the bookmark as states in a DFA! Transition based on symbols being pushed to the stack.

**DFA Production Steps**

1. Create a start state with a single fresh item for the start rule \( S^3 \).

\[ \text{state label} \]

\[ S^3 \rightarrow \cdot S \cdot \] \[ \text{fresh item} \]

2. Select a state \( q_i \) that has \( \geq 1 \) non-reducible item.

- For each non-reducible item, create a transition to a new state on the symbol after \( \cdot \) in a rule.
- In the new state, any AEN that follow the \( \cdot \) should have their rules expanded into the new state as fresh items.

\[ \rightarrow \] If this adds rules where \( \cdot \) is before B\( E \cdot N \), expand for B\( E \cdot N \)'s rules.
Repeat step 2 until no new states are discovered.

Mark states containing reducible items as accept states.

\[ L \rightarrow i c : \quad 4 \rightarrow S \rightarrow S a : b \quad b \rightarrow (6) S \rightarrow S a b \rightarrow \text{reduce} \quad \text{sub to} \quad S \]

- Problem: What if a state has \( \geq 1 \) reductions or a mix of shifts & reductions?

- Shift-reduce Conflict: When a state has a shift & reduce item.

\[ L \rightarrow e g : \quad (7) A \rightarrow x : a \quad B \rightarrow B \]

- Reduce-reduce conflict: When a state has \( \geq 1 \) reducible items.

\[ L \rightarrow e g : \quad (7) A \rightarrow x : a \quad B \rightarrow B \]

- A grammar is LR(0) iff \( \not\exists \) any shift-reduce and reduce-reduce conflicts in its automaton.

- SLR(1) uses one symbol lookahead to determine if it should shift or what reduction to apply.

\[ L \rightarrow \text{With LR(0)'s DFA, uses the follow set of a reduction rule in a conflict to determine what action to take.} \]

\[ \rightarrow \text{Reduce if the lookahead is follow(A) for } A \rightarrow B^*, \text{ else shift.} \]

---

If the lookahead is follow(A), we reduce, else we shift from the next symbol from input.
**SLR(1) Diagram**

E.g. Use the shift-reduce table to parse `1pxab1`.

**LR Parsing Steps:**

1. Begin with DFA state 0 on the stack.

2. Use top of the state stack & 1st letter of input to determine next action.
   - **Shift:** move top of the input to the symbol stack & push the new state to the state stack.
   - **Reduce:** Remove a number of items equal to the length of the RHS rule then shift the LHS EN.

3. If the entry does not exist, reject (\( \vdash x \in \{\text{LGLG}\} \)).

4. If we shift -1, accept (\( \vdash x \in \{\text{LGLG}\} \)).
<table>
<thead>
<tr>
<th>State Stack</th>
<th>Symbol Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>t-pqab-1</td>
<td>Initialize, push 0</td>
</tr>
<tr>
<td>0</td>
<td>E</td>
<td>t-pqab-1</td>
<td>Shift 1, push 1</td>
</tr>
<tr>
<td>0 1</td>
<td>t</td>
<td>pqab-1</td>
<td>Shift p, push 5 shift X, push 6</td>
</tr>
<tr>
<td>0 1 5</td>
<td>t-p</td>
<td>gab-1</td>
<td>Reduce X→E shift X, push 6</td>
</tr>
<tr>
<td>0 1 5 9</td>
<td>t-p</td>
<td>gab-1</td>
<td>Reduce X→pX shift X, push 6</td>
</tr>
<tr>
<td>0 1 3</td>
<td>t-x</td>
<td>gab-1</td>
<td>Shift q, push 7</td>
</tr>
<tr>
<td>0 1 3 7</td>
<td>t-x-y</td>
<td>ab-1</td>
<td>Reduce Y→q shift Y, push 2</td>
</tr>
<tr>
<td>0 1 3 2</td>
<td>t-x-y</td>
<td>ab-1</td>
<td>Reduce S→xy shift S, push 3</td>
</tr>
<tr>
<td>0 1 8</td>
<td>t-s</td>
<td>ab-1</td>
<td>Shift a, push 4</td>
</tr>
<tr>
<td>0 1 8 4</td>
<td>t-sa</td>
<td>b-1</td>
<td>Shift b, push 5</td>
</tr>
<tr>
<td>0 1 8 4 10</td>
<td>t-sab</td>
<td>-</td>
<td>Reduce S→Sab shift S, push 6</td>
</tr>
<tr>
<td>0 1 8</td>
<td>t-s</td>
<td>-</td>
<td>Shift 1, push 6</td>
</tr>
<tr>
<td>0 1 8 6</td>
<td>t-s-1</td>
<td>E</td>
<td>Accept</td>
</tr>
</tbody>
</table>

- Rightmost derivation is obtained by reading the symbol stack concatenated with the remaining input bottom to top:

\[
S' \\
\Rightarrow tS-1 \\
\Rightarrow tSa-1 \quad S' \quad S' \\
\Rightarrow tXYab-1 \quad S' \\
\Rightarrow tXyab-1 \quad S' \\
\Rightarrow tpxqab-1 \quad t \quad S - 1 \quad S' \\
\Rightarrow tpxab-1 \quad S' \\
\Rightarrow tpx-1 \quad S' \\
\Rightarrow \epsilon
\]
For the grammar:
1. \( S' \rightarrow 1S1 \)
2. \( S \rightarrow S + T \)
3. \( S \rightarrow T \)
4. \( T \rightarrow ID \)

Show that \( x = "+" \) is \( x \notin L(G) \) (ie: input \( t+t \) is rejected)

SLR(1) DFA (Also LR(0) : there are no conflicts)

<table>
<thead>
<tr>
<th>State Stack</th>
<th>Symbol Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>Initialize</td>
</tr>
<tr>
<td>0</td>
<td>( \epsilon )</td>
<td>( t+t )</td>
<td>Shift ( t ), push 1</td>
</tr>
<tr>
<td>0 1</td>
<td>( t )</td>
<td>( +t )</td>
<td>Shift +, error!</td>
</tr>
</tbody>
</table>

\( \Rightarrow \) : there is no transition from state 0 on input "+", we error & "+" \( \notin L(G) \)
• SLR(1) can fix the limitations of LL(1) & LR(0), but also has a limitation itself.

with grammar:

\[ S' \rightarrow 1 \cdot S 1 \]

\[ S \rightarrow 1D \]

\[ S \rightarrow E = E \]

\[ E \rightarrow 1D \]

\[ \text{Follow}(S) = \{ \epsilon, 13 \} \]

\[ \text{Follow}(E) = \{ \epsilon, -13 \} \]

consider the DFA snippet:

\[ \text{(1) } S' \rightarrow 1 \cdot S 1 \]

\[ S \rightarrow 1D \]

\[ S \rightarrow E = E \]

\[ E \rightarrow 1D \]

\[ \text{(2) } S \rightarrow 1D \cdot \{ \epsilon, 13 \} \]

\[ E \rightarrow 1D \cdot \{ \epsilon, -13 \} \]

\[ \text{Both reduce on lookahead } = -1, \text{ what do I choose?} \]

• In state (2), the SLR(1)'s simple lookahead using \text{Follow}(\ldots) does not work as \[ 1 \in \text{Follow}(S) \text{ and } 1 \in \text{Follow}(E). \]

\[ \Rightarrow \text{ our SLR(1) DFA has } \geq 1 \text{ reduce-reduce or shift-reduce conflicts, our grammar cannot be SLR(1)} \]

• This can only be fixed by a more adaptive \text{Follow} set for reduction conflicts that guarantee no shared elements between sets.

\[ \Rightarrow \text{LR(1)} \text{ does this for us!} \]