Non LL(1) Grammars

1. $S \rightarrow ab$
2. $S \rightarrow acb$
A grammar $G$ is non-LL(1) if there exists $A \in N, x \in T$ s.t. $|\text{Predict}(A, x)| > 1$

- There might be another grammar $G'$ s.t. $L(G) = L(G')$ but $G'$ is LL(1)
- Factoring might help
A non LL(1) language

Or it might not.

\[ L = \{ a^n b^m | n \geq m \geq 0 \} \]

Grammar (ambiguous)

Grammar (unambiguous)
Bottom-Up Parsing

- The Bottom-Up version is LR parsing.
- The language class LR(1) is not the same as LL(1)
Recall that a stack in LL/top-down parsing is used in the following way:

\[ \text{input processed} + \text{stack} = \text{current derivation} \]
(Note that the stack here is read from the top to bottom)

For LR/bottom-up parsing, we have

\[ \text{stack} + \text{input to be read} = \text{current derivation} \]
(stack is read from bottom to top here)
A trace

<table>
<thead>
<tr>
<th>Derivation</th>
<th>Stack</th>
<th>Input read</th>
<th>Unread Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ abywz ⊥</td>
<td>ε</td>
<td>ε</td>
<td>⊢ abywz ⊥</td>
<td>Shift ⊥</td>
</tr>
<tr>
<td>⊢ abywz ⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊢ abywz ⊥</td>
<td>Shift a</td>
</tr>
<tr>
<td>⊢ abywz ⊥</td>
<td>⊥ a</td>
<td>⊥ a</td>
<td>bywz ⊥</td>
<td>Shift b</td>
</tr>
<tr>
<td>⊢ abywz ⊥</td>
<td>⊥ a b</td>
<td>⊥ ab</td>
<td>ywz ⊥</td>
<td>Reduce A → ab</td>
</tr>
<tr>
<td>⊢ Aywz ⊥</td>
<td>⊥ A</td>
<td>⊥ ab</td>
<td>ywz ⊥</td>
<td>Shift y</td>
</tr>
<tr>
<td>⊢ Aywz ⊥</td>
<td>⊥ A y</td>
<td>⊥ aby</td>
<td>wz ⊥</td>
<td>Shift w</td>
</tr>
<tr>
<td>⊢ Aywz ⊥</td>
<td>⊥ A y w</td>
<td>⊥ abyw</td>
<td>z ⊥</td>
<td>Shift z</td>
</tr>
<tr>
<td>⊢ Aywz ⊥</td>
<td>⊥ A y w z</td>
<td>⊥ abywz</td>
<td>⊥</td>
<td>Reduce B → w z</td>
</tr>
<tr>
<td>⊢ AyB ⊥</td>
<td>⊥ A y B</td>
<td>⊥ abywz</td>
<td>⊥</td>
<td>Reduce S → AyB</td>
</tr>
<tr>
<td>⊢ S ⊥</td>
<td>⊥ S</td>
<td>⊥ abywz</td>
<td>⊥</td>
<td>Shift ⊥</td>
</tr>
<tr>
<td>⊢ S ⊥</td>
<td>⊥ S ⊥</td>
<td>⊥ abywz ⊥</td>
<td>ε</td>
<td>Reduce S → ⊥ S ⊥</td>
</tr>
<tr>
<td>S’</td>
<td>S’</td>
<td>⊥ abywz ⊥</td>
<td>ε</td>
<td>Accept</td>
</tr>
</tbody>
</table>

Shift: shifting a token from one place to another (push)

Reduce: size of the stack may be reduced (pop RHS, push LHS)
Somehow, we shifted at just the right time, and reduced just at the right time

How did we know this?
  - Recall that for LL(1) parsing, we had a predictor table
  - For LR(1) parsing, we have an oracle, in the form of a DFA
Constructing DFA oracle for LR(1) grammars

- This is difficult to do
  - Donald Knuth proved a theorem that we can construct a DFA (really, a transducer) for LR(1) grammars (1965)
  - This transducer tells us when to shift or reduce.
- We will build and use the transducer
Building an LR(0) automaton

Definition: An item is a production with a dot (●) somewhere on the RHS (which indicates a partially completed rule)

How to construct the automaton:

- make the start state the first rule, with the dot (●) in front of the left-most symbol of the RHS
- for each state, label an arc with the symbol that follows ● and advance the ● one position to the right in the next state.
- If the ● precedes a non-terminal (e.g., A) add all productions with that non-terminal A on the LHS to the current state, with the ● in the leftmost position