Warm Up Problem

Recall from last class:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Nullable</th>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S' \rightarrow \epsilon )</td>
<td>F</td>
<td>{\epsilon}</td>
<td>{}</td>
</tr>
<tr>
<td>( S \rightarrow c )                   (1)</td>
<td>F</td>
<td>{b, c, d}</td>
<td>{}</td>
</tr>
<tr>
<td>( S \rightarrow QRS )                  (2)</td>
<td>F</td>
<td>{b, d}</td>
<td>{b, c, d}</td>
</tr>
<tr>
<td>( R \rightarrow \epsilon )                   (3)</td>
<td>T</td>
<td>{b}</td>
<td>{b, c, d}</td>
</tr>
<tr>
<td>( R \rightarrow b )                  (4)</td>
<td>T</td>
<td>{b}</td>
<td>{b, c, d}</td>
</tr>
<tr>
<td>( Q \rightarrow R )                  (5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Q \rightarrow d )                  (6)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compute the predict table for the above grammar.
Predict Table

**Definition**

\[
\text{Predict}(A, a) = \{ A \to \beta : a \in \text{First}(\beta) \} \\
\cup \{ A \to \beta : \beta \text{ is nullable and } a \in \text{Follow}(A) \}
\]

\[
\begin{array}{c|cccc}
S' & \leftarrow S & \leftarrow & \leftarrow & \leftarrow \\
S \rightarrow c & (1) & & & \\
S \rightarrow QRS & (2) & & & \\
R \rightarrow \epsilon & (3) & & & \\
R \rightarrow b & (4) & & & \\
Q \rightarrow R & (5) & & & \\
Q \rightarrow d & (6) & & & \\
\end{array}
\]

\[
\begin{array}{c|cccc}
\text{ } & \leftarrow \quad b \quad c \quad d \quad \leftarrow \\
S' & \{0\} & & & \\
S & \{2\} & \{1, 2\} & \{2\} & \\
Q & \{5\} & \{5\} & \{5, 6\} & \\
R & \{3, 4\} & \{3\} & \{3\} & \\
\end{array}
\]
CS 241 Lecture 13
Bottom Up Parsing
With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot
A grammar is $LL(1)$ if and only if:

- no two distinct productions with the same LHS can generate the same first terminal symbol
- no nullable symbol $A$ has the same terminal symbol $a$ in both its first and follow sets.
- there is only one way to send a nullable symbol to $\epsilon$. 
A Classical Example

- The previous example has shown us that not all examples are $LL(1)$.
- Suppose we have a grammar that is not $LL(1)$. Can we convert it to become $LL(1)$?
A Classical Example

• The previous example has shown us that not all examples are $LL(1)$.

• Suppose we have a grammar that is not $LL(1)$. Can we convert it to become $LL(1)$?

• Sometimes. Let’s see an example:

\[
\begin{align*}
S & \rightarrow S+T \hspace{1cm} (1) \\
S & \rightarrow T \hspace{1cm} (2) \\
T & \rightarrow T*F \hspace{1cm} (3) \\
T & \rightarrow F \hspace{1cm} (4) \\
F & \rightarrow a \mid b \mid c \mid ( S ) \hspace{1cm} (5, 6, 7, 8)
\end{align*}
\]

This grammar is not $LL(1)$. Why?
Primary Issue

With this grammar (Recall: This respected BEDMAS):

\[
egin{align*}
S & \rightarrow S+T & (1) \\
S & \rightarrow T & (2) \\
T & \rightarrow T*F & (3) \\
T & \rightarrow F & (4) \\
F & \rightarrow a \mid b \mid c \mid (S) & (5, 6, 7, 8)
\end{align*}
\]

The primary issue is that left recursion is at odds with LL(1). In fact, left recursive grammars are always not LL(1). For example, Examine the derivations for \(a\) and \(a + b\) below:

\[
\begin{align*}
S & \Rightarrow S + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + b \\
S & \Rightarrow T \Rightarrow F \Rightarrow a
\end{align*}
\]

Notice that they have the same first character but required different starting rules from \(S\). That is \(\{1, 2\} \subseteq \text{Predict}(S, a)\). Our first step is to at least make this right recursive instead.
To make a [direct] left recursive grammar right recursive; say

\[ A \rightarrow A\alpha | \beta \]

where \( \beta \) does not begin with the non-terminal \( A \), we remove this rule from our grammar and replace it with:

\[
A \rightarrow \beta A' \\
A' \rightarrow \alpha A' | \epsilon
\]
Right Recursive

The above solves our issue

\[
S \rightarrow TZ' \quad (1)
\]

\[
Z' \rightarrow +TZ' \mid \epsilon \quad (2,3)
\]

\[
T \rightarrow FT' \quad (4)
\]

\[
T' \rightarrow \ast FT' \mid \epsilon \quad (5,6)
\]

\[
F \rightarrow a \mid b \mid c \mid (S) \quad (7,8,9,10)
\]

we get a right recursive grammar. This is \( LL(1) \) (Exercise).
Right Recursive Grammars are Still Mortal!

However not all right recursive grammars are $LL(1)!$. Consider

$$
\begin{align*}
S & \rightarrow T+S \\
S & \rightarrow T \\
T & \rightarrow F*T \\
T & \rightarrow F \\
F & \rightarrow a | b | c | (S)
\end{align*}
$$

we get a right recursive grammar. However this one is not $LL(1)!$

$$
\begin{align*}
S \Rightarrow T + S \Rightarrow F + S \Rightarrow a + S \Rightarrow a + T \Rightarrow a + b \\
S \Rightarrow T \Rightarrow F \Rightarrow a
\end{align*}
$$

Again, we have $\{1, 2\} \subseteq \text{Predict}(S, a)$. However with this there is still hope. We can apply a process known as $\text{factoring}$. 
Left Factoring

Idea: If $A \rightarrow \alpha \beta_1 | ... | \alpha \beta_n | \gamma$ where $\alpha \neq \epsilon$ and $\gamma$ is representative of other productions that do not begin with $\alpha$, then we can change this to the following equivalent grammar by left factoring:

$$
A \rightarrow \alpha B | \gamma \\
B \rightarrow \beta_1 | \ldots | \beta_n
$$

In this way, we remove the issues on the previous slide. We can factor to get the following grammar:
Applying this technique to the previous example:

\[
\begin{align*}
S & \rightarrow TZ' \\
Z' & \rightarrow \epsilon | +S \\
T & \rightarrow FT' \\
T' & \rightarrow \epsilon | \star T \\
F & \rightarrow a | b | c | (S)
\end{align*}
\]

we can get an \textit{LL}(1) grammar. The take away from these last few slides is that \textit{LL}(1) is not compatible with a left-associative grammar.

There is one other situation we can attempt to resolve and that is the situation where a rule of the form \( A \rightarrow \epsilon \) causes the ambiguity. I will leave this as an example to consider.
Cheat Sheet and Examples

Nullable:
- $A \rightarrow \epsilon$ implies that Nullable($A$) = true.
- Further Nullable($\epsilon$) = true.
- If $A \rightarrow B_1...B_n$ and each of Nullable($B_i$) = true then Nullable($A$) = true.

First:
- $A \rightarrow a\alpha$ then $a \in$ First($A$)
- $A \rightarrow B_1...B_n$ then First($A$) = First($A$) $\cup$ First($B_i$) for each $i \in \{1,..,n\}$ until Nullable($B_i$) is false.

Follow:
- $A \rightarrow \alpha B \beta$ then Follow($B$) = Follow($B$) $\cup$ First($\beta$)
- $A \rightarrow \alpha B \beta$ and Nullable($\beta$) = true, then Follow($B$) = Follow($B$) $\cup$ Follow($A$)

$$\text{Predict}(A, a) = \{ A \rightarrow \beta : a \in \text{First}(\beta) \} \cup \{ A \rightarrow \beta : \beta \text{ is nullable and } a \in \text{Follow}(A) \}$$
Just For Fun (Thanks to Troy Vasiga and Kevin Lanctot for this!)

Is there a grammar that is not $LL(k)$ for any $k$?
Just For Fun (Thanks to Troy Vasiga and Kevin Lanctot for this!)

Is there a grammar that is not $LL(k)$ for any $k$?

Consider $L = \{a^n b^m : n \geq m \geq 0\}$ (language with number of $a$s is greater than number of $b$s).

This is not $LL(k)$ for any $k$! (Consider $w = a^{k+1} b^k$ which would need at least a $k + 1$ look ahead). In fact there are ambiguous and unambiguous examples!

Create two CFGs that recognize this language; one ambiguous and one not.
Grammars for $L$

Ambiguous

\[
S \rightarrow \epsilon \\
S \rightarrow aS \\
S \rightarrow aSb
\]

Unambiguous

\[
S \rightarrow aS \\
S \rightarrow B \\
B \rightarrow aBb \\
B \rightarrow \epsilon
\]
Bottom Up Parsing

Recall: Determining the $\alpha_i$ in $S \Rightarrow \alpha_1 \Rightarrow ... \Rightarrow w$

- Idea: Instead of going from $S$ to $w$, let's try to go from $w$ to $S$.
- Our stack this time will store the partially reduced information thus far. (Contrast to top-down which stores the $\alpha_i$ in reverse order!)
- Our invariant here will be Stack + Unread Input = $\alpha_i$. (Contrast to top-down where invariant was unread input + reversed Stack contents = $\alpha_i$.)
Recall our grammar:

\[
\begin{align*}
S' & \rightarrow \vdash S \dashv & (0) \\
S & \rightarrow AcB & (1) \\
A & \rightarrow ab & (2) \\
A & \rightarrow ff & (3) \\
B & \rightarrow def & (4) \\
B & \rightarrow ef & (5)
\end{align*}
\]

We wish to process \( w = \vdash abcdef \dashv \) using this bottom up technique.
# Parsing Bottom-Up

<table>
<thead>
<tr>
<th>Stack</th>
<th>Read</th>
<th>Processing</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢</td>
<td>⊢</td>
<td>⊢ abcdef</td>
<td>Shift ⊢</td>
</tr>
<tr>
<td>⊢</td>
<td>⊢</td>
<td>abcdef ⊢</td>
<td>Shift a</td>
</tr>
<tr>
<td>⊢ a</td>
<td>⊢ a</td>
<td>bcdef ⊢</td>
<td>Shift b</td>
</tr>
<tr>
<td>⊢ ab</td>
<td>⊢ ab</td>
<td>cdef ⊢</td>
<td>Reduce (2); pop b, a, push A</td>
</tr>
<tr>
<td>⊢ A</td>
<td>⊢ ab</td>
<td>cdef ⊢</td>
<td>Shift c</td>
</tr>
<tr>
<td>⊢ Ac</td>
<td>⊢ abc</td>
<td>def ⊢</td>
<td>Shift d</td>
</tr>
<tr>
<td>⊢ Acd</td>
<td>⊢ abcd</td>
<td>ef ⊢</td>
<td>Shift e</td>
</tr>
<tr>
<td>⊢ Acde</td>
<td>⊢ abcde</td>
<td>f ⊢</td>
<td>Shift f</td>
</tr>
<tr>
<td>⊢ Acdef</td>
<td>⊢ abcdef</td>
<td></td>
<td>Reduce (4); pop f, d, e push B</td>
</tr>
<tr>
<td>⊢ AcB</td>
<td>⊢ abcdef</td>
<td></td>
<td>Reduce (1); pop B, c, A push S</td>
</tr>
<tr>
<td>⊢ S</td>
<td>⊢ abcdef</td>
<td></td>
<td>Shift ⊢</td>
</tr>
<tr>
<td>⊢ S ⊢</td>
<td>⊢ abcdef</td>
<td>ε</td>
<td>Reduce (0); pop ⊢, S, ⊢ push S’</td>
</tr>
<tr>
<td>S’</td>
<td>⊢ abcdef</td>
<td>ε</td>
<td>Accept</td>
</tr>
</tbody>
</table>


Notes On Bottom-Up Parsing

- Accept if and only if stack contains $S'$ and input is $\epsilon$.
- At each step, need to either shift a character from the stack OR reduce if the top of the stack is the right hand side of a grammar rule (then replace top of stack elements with the left hand side of the aforementioned rule).
- When should we shift vs reduce?
Ideas

Could try to use the next character of input to help decide when to shift or reduce like with LL(1) but the problem is still tricky. However...
Ideas

Could try to use the next character of input to help decide when to shift or reduce like with LL(1) but the problem is still tricky. However...

**Theorem (Knuth 1965)**

The set

\[ \{ wa : \exists x \text{ such that } S \Rightarrow^* wax \} \]

where \( w \) is the stack and \( a \) is the next character is a regular language!

Note: This is actually a DFA that also write output which are called *finite state transducers*. 
Summary

• Creating the associated DFA can help us to make shift or reduce decisions.
• The resulting method is called an LR scan
  • Left to right scanning
  • Do rightmost derivations.
• We will start off with an LR(0) automaton.
An Example (Thanks to Brad and Kevin for this example)

Consider the following context-free grammar:

\[
\begin{align*}
S' & \rightarrow \mid S \mid & (0) \\
S & \rightarrow S + T & (1) \\
S & \rightarrow T & (2) \\
T & \rightarrow d & (3)
\end{align*}
\]

We construct the DFA associated with this grammar.
Notation and Procedure

**Definition**

An **item** is a production with a dot • somewhere on the right hand side of a rule.

- Items indicate a partially completed rule.
- We will begin in a state labelled by the rule $S' \rightarrow \bullet \vdash S \dashv$. 
LR(0) Construction

- From a state, for each rule in the state, move the dot forward by one character. The transition function is given by the symbol you jumped over.
- For example, with $S' \rightarrow \bullet \vdash S \dashv$, we move the $\bullet$ over $\vdash$. Thus, the transition function will consume the symbol $\vdash$.
- The state we end up in will contain the transition $S' \rightarrow \vdash \bullet S \dashv$. It also contains more!
- In the new state, if in the set of rules we have $\bullet A$ for some non-terminal $A$, we then add all rules with $A$ in the left hand side of a production with a dot preceding the right hand side!
- In this case, this state will include the rules $S \rightarrow \bullet S + T$ and $S \rightarrow \bullet T$.
- Notice now we also have $\bullet T$ and so we also need to include the rules where $T$ is the left hand side adding the rule $T \rightarrow \bullet d$.
- We continue on with these rules to get the final DFA.
Transducer for our LR(0) grammar

(With thanks to Kevin Lanctot)
Using the Automaton

- Start in the start state with an empty stack.
- **Shift:**
  - Shift a character from input to the stack.
  - Follow any transition that follows with the character as a label.
  - If none, reduce or give an error.
- **Reduce:**
  - Reduce states have only one item and the • is in the rightmost position.
  - Reduce the rule in the state: Pop the RHS off the stack and backtrack in your DFA the number of states corresponding to the number of elements in the RHS. Then follow the transition for the LHS and push the LHS on your stack.
- Accept if $S'$ is on the stack and the input is empty.

**NOTE:** Because we need to backtrack, we also need to push the DFA states on the stack as well! (so really accept when stack has only the start state).
Example

Consider processing $d + d + d$.