Warm Up Problem

Is the following grammar LL(1)? Why or why not?

\[ S' \rightarrow \vdash S \dashv \quad (0) \]
\[ S \rightarrow cT \quad (1) \]
\[ S \rightarrow abc \quad (2) \]
\[ T \rightarrow T + U \quad (3) \]
\[ T \rightarrow d \quad (4) \]
\[ U \rightarrow a \quad (5) \]

How does one make it right recursive?

Ans: Change Rules (3) and (4) to

\[ T \rightarrow dT' \quad (3') \]
\[ T' \rightarrow + UT' \mid \epsilon \quad (4') \]
Warm Up Problem

Is the following grammar LL(1)? Why or why not?

\[ S' \rightarrow \epsilon \ S \ \epsilon \ (0) \]
\[ S \rightarrow cT \ \ (1) \]
\[ S \rightarrow \ a \ b \ c \ \ (2) \]
\[ T \rightarrow T + U \ \ (3) \]
\[ T \rightarrow d \ \ (4) \]
\[ U \rightarrow a \ \ (5) \]

How does one make it right recursive?

Ans: Change Rules (3) and (4) to
\[ T \rightarrow dT' \]
\[ T' \rightarrow + UT' \mid \epsilon \]
Is the following grammar LL(1)? Why or why not?

\[ S' \rightarrow \top \quad S \quad \downarrow \quad (0) \]
\[ S \rightarrow cT \quad (1) \]
\[ S \rightarrow abc \quad (2) \]
\[ T \rightarrow T + U \quad (3) \]
\[ T \rightarrow d \quad (4) \]
\[ U \rightarrow a \quad (5) \]

How does one make it right recursive?

Ans: Change Rules (3) and (4) to \[ T \rightarrow dT' \] and \[ T' \rightarrow +UT' \ | \ \epsilon. \]
CS 241 Lecture 14
Bottom Up Parsing Continued
With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot
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...
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.*
Consider the following context-free grammar:

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

We construct the DFA associated with this grammar.
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 $S' \rightarrow \bullet \vdash S \leftarrow$, 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 $S' \rightarrow \vdash \bullet S \leftarrow$. 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 rot 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 if not possible, 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! (Thus, accept when stack has the start state and $S'$).
Example

Consider processing ⊢ d + d + d ⊣:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Read</th>
<th>Processing</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ϵ</td>
<td>⊢ d + d + d ⊣</td>
<td>Shift ⊥, go to state 2</td>
</tr>
<tr>
<td>1 ⊢ 2</td>
<td>⊢</td>
<td>d + d + d ⊣</td>
<td>Shift d, go to state 6</td>
</tr>
<tr>
<td>1 ⊢ 2d6</td>
<td>⊢ d</td>
<td>+d + d ⊣</td>
<td>Reduce T → d. Pop one symbol and one state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Now in state 2. Push T go to state 5</td>
</tr>
<tr>
<td>1 ⊢ 2T5</td>
<td>⊢ d</td>
<td>+d + d ⊣</td>
<td>Reduce S → T. Pop one symbol and one state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Now in state 2. Push S go to state 3</td>
</tr>
<tr>
<td>1 ⊢ 2S3</td>
<td>⊢ d</td>
<td>+d + d ⊣</td>
<td>Shift +, go to state 7</td>
</tr>
<tr>
<td>1 ⊢ 2S3 + 7</td>
<td>⊢ d +</td>
<td>d + d ⊣</td>
<td>Shift d go to state 6</td>
</tr>
<tr>
<td>1 ⊢ 2S3 + 7d6</td>
<td>⊢ d + d</td>
<td>+d ⊣</td>
<td>Reduce T → d. Pop one symbol and one state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Now in state 7. Push T go to state 8</td>
</tr>
<tr>
<td>1 ⊢ 2S3 + 7T8</td>
<td>⊢ d + d</td>
<td>+d ⊣</td>
<td>Reduce S → S + T. Pop three symbols and three states.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Now in state 2. Push S go to state 3</td>
</tr>
<tr>
<td>1 ⊢ 2S3</td>
<td>⊢ d + d</td>
<td>+d ⊣</td>
<td>Shift +, go to state 7</td>
</tr>
<tr>
<td>1 ⊢ 2S3 + 7</td>
<td>⊢ d + d+</td>
<td>d ⊣</td>
<td>Shift d go to state 6</td>
</tr>
<tr>
<td>1 ⊢ 2S3 + 7d6</td>
<td>⊢ d + d + d</td>
<td>⊣</td>
<td>Reduce T → d. Pop one symbol and one state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Now in state 7. Push T go to state 8</td>
</tr>
<tr>
<td>1 ⊢ 2S3 + 7T8</td>
<td>⊢ d + d</td>
<td>+d ⊣</td>
<td>Reduce S → S + T. Pop three symbols and three states.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Now in state 2. Push S go to state 3</td>
</tr>
<tr>
<td>1 ⊢ 2S3</td>
<td>⊢ d + d + d</td>
<td>⊣</td>
<td>Shift ⊥, go to state 4</td>
</tr>
<tr>
<td>1 ⊢ 2S3 ⊣ 4</td>
<td>⊢ d + d + d ⊣</td>
<td>ε</td>
<td>Reduce S’ → ⊣ S ⊣. Pop three symbols and three states.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Now in state 1. Push S’ go to state 1</td>
</tr>
<tr>
<td>1S’</td>
<td>⊢ d + d + d ⊣</td>
<td>ε</td>
<td>Accept</td>
</tr>
</tbody>
</table>
Two Possible Issues

Issue one (Shift-Reduce):

- What if a state has two items of the form:

\[ A \rightarrow \alpha \cdot a\beta \]
\[ B \rightarrow \gamma. \]

- Should we shift or reduce?

Issue two (Reduce-Reduce)

- What if a state has two items of the form:

\[ A \rightarrow \alpha \cdot \]
\[ B \rightarrow \gamma. \]

- Which reduction should we do?
Two Possible Issues

Issue one (Shift-Reduce):

- What if a state has two items of the form:

\[
A \rightarrow \alpha \cdot a\beta
\]
\[
B \rightarrow \gamma.
\]

- Should we shift or reduce?

Issue two (Reduce-Reduce)

- What if a state has two items of the form:

\[
A \rightarrow \alpha.
\]
\[
B \rightarrow \gamma.
\]

- Which reduction should we do?

Answer: Throw out all grammars that have these situations.
We say that a grammar is LR(0) if and only if after creating the Knuth transducer, no state has one of the aforementioned conflicts.
Recall that LL(1) grammars were at odds with left recursive languages.

Are LR(0) grammars in conflict with a type of recursive language? Yes! Right recursive languages are in direct conflict with LR(0) grammars (Think $R$ from LR(0)). Consider the following grammar (changed rule 1):

- $S \rightarrow \vdash S \lhd (0)$
- $S \rightarrow T + S (1)$
- $S \rightarrow T (2)$
- $T \rightarrow d (3)$
Recall that LL(1) grammars were at odds with left recursive languages.

Are LR(0) grammars in conflict with a type of recursive language?
Recall that LL(1) grammars were at odds with left recursive languages.

Are LR(0) grammars in conflict with a type of recursive language?

Yes! Right recursive languages are in direct conflict with LR(0) grammars (Think $R$ from LR(0)). Consider the following grammar (changed rule 1):

\[
\begin{align*}
S' &\to \vdash S \vdash & (0) \\
S &\to T + S & (1) \\
S &\to T & (2) \\
T &\to d & (3)
\end{align*}
\]
New LR(0) Automaton

1. $S' \rightarrow \bullet \vdash S \dashv$

2. $S' \rightarrow \vdash \bullet S \dashv$
   $S \rightarrow \bullet T + S$
   $S \rightarrow \bullet T$
   $T \rightarrow \bullet d$

3. $S' \rightarrow \vdash S \dashv$

4. $S' \rightarrow \vdash S \dashv$

5. $S \rightarrow T \bullet + S$
   $S \rightarrow T \bullet$

6. $T \rightarrow \bullet d$

7. $S \rightarrow T + S \bullet$

8. $S \rightarrow T + \bullet S$
   $S \rightarrow \bullet T + S$
   $S \rightarrow \bullet T$
   $T \rightarrow \bullet d$
Better Picture

Source: http://smlweb.cpsc.ucalgary.ca/

(Note: We don’t have state 2 in our diagram).
Conflicts

State 5 [state 3 in the nice photo!] has a shift-reduce conflict.
  • Suppose the input began with $\vdash d$.
Conflict

State 5 [state 3 in the nice photo!] has a shift-reduce conflict.

- Suppose the input began with $\vdash d$.
- This gives a stack of $\vdash d$ and then we reduce in state 6 [state 5 in the nice photo!] so our stack changes to $\vdash T$ and we move to state 5 via state 1.
State 5 [state 3 in the nice photo!] has a shift-reduce conflict.

- Suppose the input began with $\vdash d$.
- This gives a stack of $\vdash d$ and then we reduce in state 6 [state 5 in the nice photo!] so our stack changes to $\vdash T$ and we move to state 5 via state 1.
- Should we reduce $S \rightarrow T$?
Conflict

State 5 [state 3 in the nice photo!] has a shift-reduce conflict.

• Suppose the input began with $\vdash d$.
• This gives a stack of $\vdash d$ and then we reduce in state 6 [state 5 in the nice photo!] so our stack changes to $\vdash T$ and we move to state 5 via state 1.
• Should we reduce $S \rightarrow T$?
• It depends! If the input is $\vdash d \vdash$ then absolutely!
• If instead, the input was $\vdash d + \ldots$ then no!
• How do we fix this?
We’ll add a lookahead to the automaton to fix the conflict! For every \( A \rightarrow \alpha \bullet \), attach \( \text{Follow}(A) \)! Recall:

\[
\begin{align*}
S' & \rightarrow \vdash S \dashv \\
S & \rightarrow T + S \\
S & \rightarrow T \\
T & \rightarrow d
\end{align*}
\]

What is \( \text{Follow}(S) \)? What about \( \text{Follow}(T) \)?
Follow Sets

Note that $\text{Follow}(S) = \{\rhd\}$ and $\text{Follow}(T) = \{+, \rhd\}$. So state 5 becomes

$$S \rightarrow T \bullet +S \quad \text{and} \quad S \rightarrow T \bullet \{\rhd\}$$

In other words, apply $S \rightarrow T \bullet +S$ if the next token is $+$ and apply $S \rightarrow T \bullet \{\rhd\}$ if the next token is $\rhd$.

We call these parsers SLR(1) parsers! (Simple LR with 1 character look ahead).
Wait... What is an LR(1) Parser?

- LR(1) parsing involves a more complicated procedure.
- Instead of adding all of \( \text{Follow}(S) \) to an item, you add only a subset of this set to each item.
- In this way, the number of states you get can blow up fairly badly depending on your follow sets (You might have \( 2^{|\text{Follow}(S)|} \) many states - details are suppressed).
- However, the parsing mechanism is the same (just the DFA changes). Programming an SLR parser then swapping in an LR(1) DFA gives you an LR(1) parser.
- Yacc and Bison, for example, use LALR(1) (Lookahead LR) which lies somewhere between SLR(1) and LR(1).
- LR(1) parsers are extremely powerful; Knuth proved that if you have an LR\((k)\) grammar for \( k > 1 \), then there is a corresponding LR\((1)\) grammar recognizing the same language!
Algorithm for LR(1) DFA

Elements in an LR(1) DFA are items followed by a terminal symbol. Let $S_M$ be the set of states of a DFA $M$

\begin{algorithm}
1: Make the LR(0) parser's DFA $M$
2: for each $A \rightarrow \alpha \bullet B \beta$, $t$ in a state $s$ of $S_M$ do
3: \hspace{1em} for each $B \rightarrow \gamma$ in $P$ do
4: \hspace{2em} for each $b \in \text{First}(\beta t)$ do
5: \hspace{3em} Add $B \rightarrow \bullet \gamma$, $b$ to $s$
6: \hspace{2em} end for
7: \hspace{1em} end for
8: end for
9: Repeat the above until no more changes have occurred.
\end{algorithm}
Sample LR(1) Grammar That is Not SLR(1)

Omitting $S'$ to be consistent with next pages notation.

\[
S \rightarrow Aa \quad (1) \\
S \rightarrow bAc \quad (2) \\
S \rightarrow dc \quad (3) \\
S \rightarrow bda \quad (4) \\
A \rightarrow d \quad (5)
\]

Source: https://stackoverflow.com/questions/10505717/how-is-this-grammar-lr1-but-not-slr1
SLR(1) Table

Source of next slides: http://smlweb.cpsc.ucalgary.ca/
### SLR(1) Table

<table>
<thead>
<tr>
<th></th>
<th>$</th>
<th>d</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>S</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>s4</td>
<td>s3</td>
<td>s2</td>
<td>s1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>r(A → d)</td>
<td>r(A → d)/s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r(S → d c)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>r(A → d)/s9</td>
<td>r(A → d)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r(S → A a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r(S → b d a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>r(S → b A c)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LR(1) Table

Note $ means end of file.
LR(1) Table

<table>
<thead>
<tr>
<th></th>
<th>$</th>
<th>d</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>S</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>s4</td>
<td>s3</td>
<td>s2</td>
<td>s1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r(A → d)</td>
<td></td>
<td></td>
<td></td>
<td>s5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r(S → d c)</td>
<td></td>
<td></td>
<td></td>
<td>s10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s9</td>
<td></td>
<td>r(A → d)</td>
</tr>
<tr>
<td>7</td>
<td>r(S → A a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r(S → b d a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r(S → b A c)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>