Summary: Top-Down Parsing

Top-Down Parsing

- Backtracking Parsing
  (“brute force”)

- Predictive Parsing
  (lookahead)

Recursive Descent Parsing

- a func for a nonterminal
- leverage call stack

LL(1) Parsing

- use explicit stack
- First and Follow sets
- parsing table-driven

Common Issues:

- left recursion
- common prefix
Example

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>$</td>
</tr>
<tr>
<td>S</td>
<td>S → A</td>
<td>S → B C</td>
<td>S → B C</td>
<td>S → B C</td>
</tr>
<tr>
<td>A</td>
<td>A → a A</td>
<td></td>
<td></td>
<td>A → ε</td>
</tr>
<tr>
<td>B</td>
<td>B → b B</td>
<td>B → ε</td>
<td>B → ε</td>
<td>B → ε</td>
</tr>
<tr>
<td>C</td>
<td>C → c C</td>
<td>C → d C</td>
<td></td>
<td>C → ε</td>
</tr>
</tbody>
</table>

Production rules:

- S → A | B C
- A → a A | ε
- B → b B | ε
- C → c C | d C | ε

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>a, b, c, d, ε</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>a, ε</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>b, ε</td>
<td>c, d, $</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>c, d, ε</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>
Bottom-up Parsing

*Basic Idea*:

- Scan the input string from left to right.
- Try to construct a parse tree starting at the bottom (i.e., the leaves) and working towards the root.

Shift-reduce parsing:

*Basic Idea*: Apply a sequence of “reductions” to transform the input string to the start symbol of the grammar.

*reduction*: replace a substring matching the RHS of a production by the LHS.
**Example**: Consider the grammar

\[
S \rightarrow aABe \\
A \rightarrow Abc \\
A \rightarrow b \\
B \rightarrow d
\]

Input: \[
\text{abbcde} \\
\Rightarrow a\text{Abcde} \\
\Rightarrow aA\text{de} \\
\Rightarrow aA\text{Be} \\
\Rightarrow S
\]

**Handles**

**Intuition**: A handle of a string \(s\) is a substring \(\alpha\) s.t.:

1. \(\alpha\) matches the RHS of a production \(A \rightarrow \alpha\); and
2. replacing \(\alpha\) by the LHS \(A\) represents a step in the reverse of a rightmost derivation of \(s\).
**Example**: Consider the grammar

\[
S \rightarrow aABe \\
A \rightarrow Abc \mid b \\
B \rightarrow d
\]

The rightmost derivation for the input `abbcde` is:

\[
S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \\
\Rightarrow abbcde.
\]

The string `aAbcde` can be reduced in two ways:

1. `aAbcde \Rightarrow aAde`; and
2. `aAbcde \Rightarrow aAbcBe`.

But (2) is not in a rightmost derivation, so `Abc` is the only handle.
Handles: cont’d

Definition: A *handle* of a right-sentential form \( \gamma \) is

1. a production \( A \rightarrow \beta \), and

2. a position in \( \gamma \) where \( \beta \) may be found and replaced by \( A \) to produce the *previous* sentential form in a rightmost derivation of \( \gamma \).

\[
\begin{array}{c}
S \\
\downarrow \\
\alpha \quad A \quad \omega \\
\downarrow \\
\beta \\
(a \text{ string of terminals})
\end{array}
\]

The handle \( A \rightarrow \beta \) in \( \alpha \beta \omega \)
Shift Reduce Parsing

Stack

Tokens

S → a b c

Shifts

Reduce

abc$
Stack Implementation of Shift-Reduce Parsing:

Data Structures:

- *the stack*, its bottom marked by $\$, initially empty.
- *the input string*, its right end marked by $\$, initially $w$.

Action:

```
repeat
  1. *Shift* zero or more input symbols onto the stack, until a handle $\beta$ is on the top of the stack.
  2. *Reduce* $\beta$ to the LHS of the appropriate production.
until ready to accept.
```

Acceptance: When the stack contains the start symbol and the input is empty.
Example: Consider the grammar

\[ S \rightarrow aABe \]
\[ A \rightarrow Abc \]
\[ A \rightarrow b \]
\[ B \rightarrow d \]

Input: \text{abcde} \quad b \quad d \quad B
\[ \Rightarrow \text{abcde} \quad b \quad A \quad A \quad A \]
\[ \Rightarrow \text{Ade} \quad a \quad a \quad a \quad a \quad S \]
\[ \Rightarrow \text{ABe} \quad \$ \quad \$ \quad \$ \quad \$ \quad \$ \]
\[ \Rightarrow S \]
**Example:**

Grammar:  
\[ S \rightarrow aABe \]
\[ A \rightarrow Abc | b \]
\[ B \rightarrow d \]

Input string: \textit{abbcde}

<table>
<thead>
<tr>
<th>Stack (→)</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>abbcde$</td>
<td>shift</td>
</tr>
<tr>
<td>$a</td>
<td>bbcded$</td>
<td>shift</td>
</tr>
<tr>
<td>$ab</td>
<td>bcde$d</td>
<td>reduce by ( A \rightarrow b )</td>
</tr>
<tr>
<td>$aA</td>
<td>bcde$d</td>
<td>shift</td>
</tr>
<tr>
<td>$aAb</td>
<td>cde$</td>
<td>shift</td>
</tr>
<tr>
<td>$aAbc</td>
<td>de$</td>
<td>reduce by ( A \rightarrow Abc )</td>
</tr>
<tr>
<td>$aA</td>
<td>de$</td>
<td>shift</td>
</tr>
<tr>
<td>$aAd</td>
<td>e$</td>
<td>reduce by ( B \rightarrow d )</td>
</tr>
<tr>
<td>$aAB</td>
<td>e$</td>
<td>shift</td>
</tr>
<tr>
<td>$aABe</td>
<td>$</td>
<td>reduce by ( S \rightarrow aABe )</td>
</tr>
<tr>
<td>$S</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
Conflicts during Shift-Reduce Parsing:

1. Can’t decide whether to shift or to reduce ("shift-reduce conflict").
   
   Example: “dangling else”:
   
   $$Stmt \rightarrow \text{if Expr then Stmt} \mid \text{if Expr then Stmt else Stmt} \mid \cdots$$

2. Can’t decide which of several possible reductions to make ("reduce-reduce conflict").

   Example:
   
   $$Stmt \rightarrow \text{id (params)} \mid Expr := Expr \mid \cdots$$
   $$Expr \rightarrow \text{id (params)}$$

   Given the input A(I, J) the parser doesn’t know whether it’s a procedure call or an array reference.
LR Parsing

- Bottom-up.

- LR($k$) parser:
  - Scans the input L-to-R.
  - Produces a Rightmost derivation.
  - Uses $k$-symbol lookahead.
Advantages:

- Very general and flexible.
- Efficiently implemented.
- Parses a large class of grammars.

Disadvantages:

- Difficult to implement by hand for typical programming language grammars.
  (Use tools such as yacc or bison.)
The driver program is the same for all LR parsers (SLR(1), LALR(1), LR(1), ...): only the parsing table changes.
• The stack holds strings of the form
  \[ s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m \]
  where \( s_m \) is on top, the \( s_i \) are "states", and \( X_i \) are grammar symbols.

• The \textbf{configuration} of an LR parser is given by a pair
  \textit{<stack contents, unexpended input>.

  A configuration \( \langle s_0 X_1 s_1 \cdots X_m s_m, a_i a_{i+1} \cdots a_n \rangle \)
  represents the right-sentential form
  \[ X_1 \cdots X_m a_i a_{i+1} \cdots a_n \]
  The sequence of symbols \( X_1 \cdots X_m \) on the parser stack is called a \textbf{viable prefix} of the right sentential form.
LR Parse Tables

- The parsing table consists of two parts: a parsing action function, and a goto function.

- For a given configuration of the parser, the next move is determined by the parse table entry
  
  \[ \text{action}[s_m, a_i]. \]

  where \( s_m \) is the topmost state on the stack, and \( a_i \) is the next input symbol.

- An action table entry can be of four types:
  1. **shift** \( s \), where \( s \) is a state.
  2. **reduce** by a grammar production \( A \rightarrow \beta \).
  3. **accept**
  4. **error**
LR Parsing: cont’d

Suppose the parser configuration is

\[ \langle s_0 X_1 s_1 \cdots X_m s_m, \quad a_i \cdots a_n \$ \rangle. \]

- if \( \text{action}[s_m, a_i] = \text{shift } s \) then the parser executes a \textit{shift} move. The new configuration is

\[ \langle s_0 X_1 s_1 \cdots X_m s_m, \underbrace{a_i \ s}, \quad a_{i+1} \cdots a_n \$ \rangle. \]

\textit{pushed}
• if \( \text{action}[s_m, a_i] = \text{reduce} \ A \rightarrow \beta \) then the parser does a \text{reduce} move. The new configuration is

\[
\langle s_0X_1s_1 \cdots X_{m-r}s_{m-r} \underbrace{A \overset{\text{new}}{s}}_{a_i, \cdots a_n}\$\rangle.
\]

where

- \( r = \) length of \( \beta \); and

- \( s = \text{goto}[s_{m-r}, A] \).

• if \( \text{action}[s_m, a_i] = \text{accept} \) then parsing is done.

• if \( \text{action}[s_m, a_i] = \text{error} \) the parser calls an error recovery routine.
5.2. Finite Automata to recognize Viable Prefixes

**Definition**: An LR(0) item of a grammar $G$ is a production of $G$ with a dot `·` added at some position in the RHS.

**Example**: The production $A \rightarrow aAb$ gives the items

\[
\begin{aligned}
A & \rightarrow aAb \\
A & \rightarrow a·Ab \\
A & \rightarrow aA·b \\
A & \rightarrow aAb·
\end{aligned}
\]

**Intuition**: An item $A \rightarrow α·β$ denotes:

- we have seen a string derivable from $α$; and
- we hope to see a string derivable from $β$. 
**Overall Goal**: Given a grammar with start symbol $S'$,

- Construct an *augmented grammar* by adding a new start symbol $S''$ and production $S'' \rightarrow S'$;

- Starting with the item $S'' \rightarrow \cdot S$, recognize the viable prefix $S'' \rightarrow S^\cdot$. 
Viable Prefix DFA

1. closure :

**Definition**: If $I$ is a set of items for a grammar $G$, then $\text{closure}(I)$ is the set of items constructed as follows:

repeat

1. add every item in $I$ to $\text{closure}(I)$;

2. if $A \rightarrow \alpha \cdot B \beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production of $G$, then add $B \rightarrow \gamma \cdot \gamma$ to $\text{closure}(I)$.

until no new item can be added to $\text{closure}(I)$.

**Intuition**: If $A \rightarrow \alpha \cdot B \beta$ is in $\text{closure}(I)$ then we hope to see a string derivable from $B$ in the input. So if $B \rightarrow \gamma$ is a production of $G$, then we should hope to see a string derivable from $\gamma$ in the input. Hence, $B \rightarrow \gamma \cdot \gamma$ is in $\text{closure}(I)$.
Viable Prefix DFA – cont’d:

2. *goto* :

**Definition**: If $I$ is a set of items for a grammar $G$ and $X$ a grammar symbol, then $\text{goto}(I, X)$ is the set of items $\text{closure}([A \to \alpha X \beta | A \to \alpha \cdot X \beta \in I])$.

**Intuition** :

- A set of items $I$ corresponds to a state.

- If $A \to \alpha \cdot X \beta \in I$ then
  - we’ve seen a string derivable from $\alpha$; and
  - we hope to see a string derivable from $X \beta$;
• now suppose we see a string derivable from $X$: the resulting state should be one in which:
  – we’ve seen a string derivable from $\alpha X$; and
  – we hope to see a string derivable from $\beta$;

• The item corresponding to this is $A \rightarrow \alpha X \cdot \beta$. 
Constructing the Viable Prefix DFA for LR(0) Items

- Given a grammar $G$ with start symbol $S$, construct the augmented grammar by adding a special production

$$S' \rightarrow S$$

where $S'$ does not appear in $G$.

- Algorithm for constructing the canonical collection of LR(0) items for an augmented grammar $G'$:

begin
  $C := \{\text{closure}(\{S' \rightarrow \cdot S\})\}$
repeat
  for each set of items $I \in C$ do
    for each grammar symbol $X$ do
      if $\text{goto}(I, X) \neq \emptyset$ then
        add $\text{goto}(I, X)$ to $C$;
      fi
    until no new set of items can be added to $C$;
return $C$;
end
### Example

<table>
<thead>
<tr>
<th>Original Grammar</th>
<th>Augmented Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E + T \mid T$</td>
<td>$S' \rightarrow E$</td>
</tr>
<tr>
<td>$T \rightarrow id \mid (E)$</td>
<td>$E \rightarrow E + T \mid T$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow id \mid (E)$</td>
</tr>
</tbody>
</table>
Augmented Grammar

S’ → E
E → E + T | T
T → id | ( E )

Kernel items are
Marked with *

Rest of the items
added by closure

. Tells where we are
in the production
Augmented Grammar

\[
S' \rightarrow E \\
E \rightarrow E + T | T \\
T \rightarrow \text{id} | ( E )
\]

Kernel items are Marked with *

Rest of the items added by closure

. Tells where we are in the production
5.3. Constructing an SLR(1) Parse Table

1. Given a grammar $G$, construct the augmented grammar $G''$ by adding the production $S'' \rightarrow S$.

2. Construct $G' = \{I_0, \ldots, I_n\}$, the set of states of the viable prefix DFA for $G''$.

3. State $i$ is constructed from $I_i$, with parsing action determined as follows:
   
   (a) $A \rightarrow \alpha \cdot a \beta \in I_i$, $a$ a terminal, $\text{goto}(I_i, a) = I_j$: set $\text{action}[i, a] = \text{shift } j$.
   
   (b) $A \rightarrow \alpha \cdot \in I_i, A \neq S'$: for each $a \in \text{FOLLOW}(A)$, set $\text{action}[i, a] = \text{reduce } A \rightarrow \alpha$.
   
   (c) $S' \rightarrow S \ast \in I_i$: set $\text{action}[i, \$] = \text{accept}$. 

4. goto transitions are constructed as follows: for each nonterminal \( A \), if \( \text{goto}(I_i, A) = I_j \) then \( \text{goto}[i, A] = j \).

5. All entries not defined by the above steps are made error.
   If there are any multiply defined entries, then \( G \) is not SLR.

6. Initial state of the parser: that constructed from \( I_0 \sim S' \rightarrow \ast S \).
### ACTION

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>Id</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>E</th>
<th>T</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td></td>
<td>S,S3</td>
<td>S,S4</td>
<td></td>
<td>E</td>
<td>T</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>S,S7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>R,#3</td>
<td>R,#3</td>
<td>R,#3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>R,#4</td>
<td>R,#4</td>
<td>R,#4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>S,S3</td>
<td>S,S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>S,S7</td>
<td>S,S6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>R,#5</td>
<td>R,#5</td>
<td>R,#5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>S,S3</td>
<td>S,S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>R,#2</td>
<td>R,#2</td>
<td>R,#2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### GOTO

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>E</th>
<th>T</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>S’</td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#2</td>
<td>E</td>
<td>E + T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#3</td>
<td>E</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#4</td>
<td>T</td>
<td>id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#5</td>
<td>T</td>
<td>( E )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Production Rules
- Follow(S’) → { $ }
- Follow(E) → { +, ), $ }
- Follow(T) → { +, ), $ }

---

**S** - SHIFT  
**R** - REDUCE  
**S#** - Next State  
**#n** - Production Rule Number
The LR Parsing Algorithm

begin
  set $ip$ to point to the first symbol of the input $w$;

while TRUE do
  let $s$ be the state on top of the stack, $a$ the symbol pointed at by $ip$;

  if $\text{action}[s,a] = \text{shift } s'$ then
    push $a$ then $s'$ on top of the stack;
    advance $ip$ to the next input symbol;

  else if $\text{action}[s,a] = \text{reduce } A \rightarrow \beta$ then
    pop $2*|\beta|$ symbols off the stack;
    let $s'$ be the state now on top of the stack;
    push $A$ then goto$[A,s']$ on top of the stack;

  else if $\text{action}[s,a] = \text{accept}$ then return;

  else $\text{error}()$;
  fi
od
end
<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ S0</td>
<td>id + id $</td>
<td>shift S3</td>
</tr>
<tr>
<td>$ S0 id S3</td>
<td>+ id $</td>
<td>red. T→id GOTO[S0,T]=S2</td>
</tr>
<tr>
<td>$ S0 T S2</td>
<td>+ id $</td>
<td>red E→T GOTO[S0,E]=S1</td>
</tr>
<tr>
<td>$ S0 E S1</td>
<td>+ id $</td>
<td>shift S7</td>
</tr>
<tr>
<td>$ S0 E S1 + S7</td>
<td>id $</td>
<td>shift S3</td>
</tr>
<tr>
<td>$ S0 E S1 + S7 id S3</td>
<td>$</td>
<td>red T→id GOTO[S7,T]=S8</td>
</tr>
<tr>
<td>$ S0 E S1 + S7 T S8</td>
<td>$</td>
<td>red E→E+T GOTO[S0,E]=S1</td>
</tr>
<tr>
<td>$ S0 E S1</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
Limitations of SLR Parsing

Cannot handle many “reasonable” grammars, e.g.:

\[
S \rightarrow R \mid L=R \\
L \rightarrow * R \mid id \\
R \rightarrow L
\]

The SLR parse table contains a state

\[
I = \{S \rightarrow L*R, R \rightarrow L*\}
\]

which causes a shift/reduce conflict on ‘=’, since ‘=’ is in FOLLOW(L).

**Problem**: For an input

\[
*id = id
\]

we want to remember enough “left context” after seeing * to make the right shift/reduce decision. SLR cannot do this adequately.
\[ S \rightarrow L = R \mid R \]
\[ R \rightarrow L \]
\[ L \rightarrow * R \mid id \]

Shift on $=$

Since $=$ is in Follow(R)

so Reduce on $=$
5.4. LR(1) Parsing

**Idea**: Extend SLR parsing to incorporate lookahead.

**LR(1) Item**:

- Of the form \([A \rightarrow \alpha \cdot \beta, a]\), where \(a\) is a terminal or is the endmarker $\$\$.
- The lookahead has no effect on items of the form \([A \rightarrow \alpha \cdot \beta, a]\), where \(\beta \neq \epsilon\).
- For items of the form \([A \rightarrow \alpha \cdot, a]\), reduce only if the next symbol is \(a\).

**Note**: For an item of the form \([A \rightarrow \alpha \cdot \beta, a]\), \(a \in \text{FOLLOW}(A)\). But there may be \(b \in \text{FOLLOW}(A)\) for which there is no item \([A \rightarrow \alpha \cdot \beta, b]\).
LR(1) Parsing: closure and goto Functions

1. closure(I):
   begin
   S := I;
   repeat
      for( each item [A → α·Bβ, a] ∈ I,
            each production B → γ,
            each terminal b ∈ FIRST(βa)) do
         add [B → γ, b] to S;
      until no new item can be added to S;
   return S;
   end

2. goto(I, X):
   begin
      let J = { [A → αX·β, a] | [A → α·Xβ, a] ∈ I };
      return closure(J);
   end
Constructing the Viable Prefix DFA for LR(1) Items

- **Given**: An augmented grammar $G''$.

- **Algorithm**:

  ```
  begin
  $C := \{\text{closure}([S'' \rightarrow \ast S, \$])\}$;
  repeat
  for each set of items $I \in C$ do
    for each grammar symbol $X$ do
      if $\text{goto}(I, X) \neq \emptyset$ then
        add $\text{goto}(I, X)$ to $C$;
      until no new set of items can be added to $C$;
  return $C$;
  end
  ```

- **Note**: The set of items construction is essentially the same as for the SLR(1) case.
<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>id</td>
</tr>
<tr>
<td>S0</td>
<td>$,S3</td>
</tr>
<tr>
<td>S1</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>$,S5</td>
</tr>
<tr>
<td>S3</td>
<td>$,S3</td>
</tr>
<tr>
<td>S4</td>
<td>R,L→id</td>
</tr>
<tr>
<td>S5</td>
<td>$,S11</td>
</tr>
<tr>
<td>S6</td>
<td>R,R→L</td>
</tr>
<tr>
<td>S7</td>
<td>R,L→*R</td>
</tr>
<tr>
<td>S8</td>
<td></td>
</tr>
<tr>
<td>S9</td>
<td>R,R→L</td>
</tr>
<tr>
<td>S10</td>
<td>R,L→id</td>
</tr>
<tr>
<td>S11</td>
<td>$,S11</td>
</tr>
<tr>
<td>S12</td>
<td></td>
</tr>
</tbody>
</table>

*S* - SHIFT  
*R* - REDUCE  
S# - Next State  
#n – Production Number
$ S0 * S3 * id = id $ 
$ S0 * S3 id S4 id = id $ 
$ S0 * S3 L S6 = id $ 
$ S0 * S3 R S7 = id $ 
$ S0 L S2 = id $ 
$ S0 L S2 = S5 id $ 
$ S0 L S2 = S5 id S10 $ 
$ S0 L S2 = S5 L S9 $ 
$ S0 L S2 = S5 R S8 $ 
accept
Constructing an LR(1) Parse Table

1. Given a grammar $G$, construct the augmented grammar $G''$ by adding the production $S'' \rightarrow S$.

2. Construct $C = \{I_0, \ldots, I_n\}$, the viable prefix DFA for $G''$.

3. State $i$ is constructed from $I_i$, with parsing action determined as follows:

   (a) $[A \rightarrow \alpha \cdot a \beta, b] \in I_i$, $a$ a terminal, $\text{goto}(I_i, a) = I_j$: set $\text{action}[i, a] = \text{shift } j$.

   (b) $[A \rightarrow \alpha \cdot, a] \in I_i, A \neq S'':$ set $\text{action}[i, a] = \text{reduce } A \rightarrow \alpha$.

   (c) $[S' \rightarrow S' \cdot, \$] \in I_i :$ set $\text{action}[i, \$] = \text{accept}$.
4. goto transitions are constructed as follows: for each nonterminal $A$, if $\text{goto}(I_i, A) = I_j$ then $\text{goto}[i, A] = j$.

5. All entries not defined by the above steps are made error.
   If there are any multiply defined entries, then $G$ is not LR(1).

6. Initial state of the parser: that constructed from $I_0 \sim [S' \rightarrow *S, \$]$. 
LR(1) vs. SLR(1):

- LR(1) more powerful, can handle a strictly larger class of grammars than SLR(1).

- The parse tables for LR(1) become very large — may be impractical for realistic grammars.

- A compromise between parsing power and table size that is commonly used is seen in LALR parsers. An LALR parser can be thought of as an LR(1) parser, some of whose states have been merged into a single state. This can be done in many (but not all) cases without causing problems. The parsers generated by tools such as yacc and bison are LALR.
5.4.3. LALR(1) Parsing

Observation: Every SLR grammar is an LR(1) grammar, but the LR(1) parser usually has many more states than the SLR parser.

Many of these states differ only on the lookahead token. But the lookahead token does not play any role except on reductions.

Definition: The core of a set of LR(1) items $I$ is

$$\text{core}(I) = \{J \mid [J, a] \in I \text{ for some } a\}$$

I.e., $\text{core}(I)$ is the set of first components of $I$.

Example: Suppose

$$I = \{[A \rightarrow c_*, a],
\quad [A \rightarrow c_*, b],
\quad [B \rightarrow c_*, c]\}$$

Then,

$$\text{core}(I) = \{A \rightarrow c_*, B \rightarrow c_*\}$$
Merging sets of LR(1) Items

- If sets of items with the same core are merged, the parser behaves essentially as before. However, some redundant reductions might be done before an error is detected.

- $\text{core}(\text{goto}(I,X))$ depends only on $\text{core}(I)$, so goto’s of merged sets may themselves be merged.

- Suppose we take a set $C_0$ of sets of LR(1) items for a given grammar, and merge those sets of items that have the same core to get a set $C_1$ of sets of LR(1) items.
LR(1) parse table construction using $C_1$ will not introduce any new shift/reduce conflicts compared to $C_0$.
However, this can introduce new reduce/reduce conflicts.
Example of reduce/reduce conflicts due to merging:

Consider the grammar given by

\[ S \rightarrow aAd | bBd | aBe | bAe \]
\[ A \rightarrow c \]
\[ B \rightarrow c \]
$ \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

$B \rightarrow c$

No Conflict
$\Rightarrow$ LR(1) Grammar
\[
S \rightarrow aAd \mid bBd \mid aBe \mid bAe \\
A \rightarrow c \\
B \rightarrow c \\
\]

A \rightarrow c . , d \\
B \rightarrow c . , e \\
A \rightarrow c . , e \\
B \rightarrow c . , d \\

Merge \\

A \rightarrow c . , d/e \\
B \rightarrow c . , d/e \\

Contains \\
reduce-reduce conflict \\
\rightarrow \text{not LALR(1)}
SAMPLE PROBLEMS
$S \to A$
$A \to a A a | \epsilon$

Shift-Reduce Conflicts when input token is $a \in \text{FOLLOW} (A)$
$ \Rightarrow A$
$A \Rightarrow a A a | \epsilon$

**Shift-Reduce Conflicts when input token is a**

* $S \Rightarrow A ., $*

* $S \Rightarrow . A, $*
  * $A \Rightarrow . a A a, $*
  * $A \Rightarrow ., $*

* $A \Rightarrow a A ., $*
  * $A \Rightarrow a A a, $*
  * $A \Rightarrow . a A a, $*
  * $A \Rightarrow ., a$
S \rightarrow C\ C
C \rightarrow c\ C \mid d
No Conflicts!
\[
S \to E \\
E \to (L) \mid a \\
L \to EL \mid E
\]

* S \to E 
* E \to (L). 
* E \to (L.) 
* E \to (L) 
L \to E L 
L \to E 
E \to (L) 
E \to a 

* L \to E L 
L \to E L 
L \to E 
E \to (L) 
E \to a 

L \to E L.
S → E
E → (L) | a
L → E L | E

* S → E , $
E → (L) , $
E → E , $
E → a , $
E → a , $

* S → E , $