Outline

- Overview shift-reduce parsing
- SLR parsing
- Reading Chapter 3, Section 4,5 & 6

X_i grammar symbol with S_i state (state says it all)
each state summarizes the information contained in the stack below it - what has been seen so far

\((<S>_0, <X>_1, ... <X>_m, <a_{m-1}, a_m, ... a_1>) \rightarrow X_1, a_m, a_0)\)

State at the top of the stack and current input - index into parsing table to determine whether to shift or reduce
Parse table:  Action & Goto

\[ \text{action}[S_m, a] \rightarrow \text{only 4:} \]
\[ \text{shift}: S \text{ where } S \text{ is a state and } t \text{ is a terminal} \]
\[ \text{reduce by a grammar production } a \rightarrow \beta \]
\[ \text{accept} \]
\[ \text{error} \]
\[ \text{go to } [S_m, \text{grammar symbol}] \text{ go to another state } <S_j,X_k> \]

Assume \( S_0 X_1 S_1 X_2 \ldots X_m S_m \#a_i \ldots \$ \)
right sentential form \( X_1 X_2 \ldots X_m \#a_i \ldots \$

1. \text{Action}[S_m, a] \text{ is shift input and go to state } S
\[ (S_0 \ldots X_m \#a_i \ldots \$) \]
2. \text{Action}[S_m, a] \text{ is reduce } A \rightarrow \beta
\[ | \beta | = r, \text{ pop off } 2r \text{ symbols} \]
\[ (S_0 \ldots X_m S_m A \#a_i \ldots \$) \]
where \( S = \text{go to } [S_m, A] \)
output generated after reduce tree
3. \text{Action}[S_m, a] = \text{accept - parsing is complete}
4. \text{Action}[S_m, a] = \text{error}

Bottom up only use 2 operations:
Shift and Reduce on stack#input

Stack#Input
Shift:
ExT#abc \Rightarrow \text{ExTa#bc}
Reduce:
ExTa#bc \Rightarrow \text{ExF#bc}
LR parsers
Can tell handle by looking at stacktop (grammar symbol, state) and k input symbols - finite state automaton.
In practice k ≤ 1
How to construct LR parse table from grammar.
Only consider SLR parser
2 phases to construct table
1. Build deterministic finite state automation to go from state to state.
   x is terminal
   M is non-terminal
Each state - how do we know from grammar where we are in the parse. Production already seen.

Notion of an LR(0) item (0 look ahead)
An item is a production with a distinguished position on the right hand side - position indicates how much of the production already seen.
Example
\[ S \rightarrow a \ B \ S \]
Items for the production:
\[ S \rightarrow .\ a \ B \ S \]
\[ S \rightarrow a .\ B \ S \]
\[ S \rightarrow a \ B .\ S \]
\[ S \rightarrow a \ B \ S .\]
called LR(0) items - Basic idea - construct a DFA that recognizes the viable prefixes group items into sets - state of SLR

DFA for Grammar
* closure of state
Construction of LR(0) items & SLR parsing table for Grammar G:

1. Create augmented grammar G’

\[ G' \rightarrow 5 \]
\[ 5' \rightarrow 5 \]
\[ 5 \rightarrow \alpha | \beta \]

What else is needed?
- A -> c.d E - indicate new state by consuming symbol d:
  - need go to function
- A -> cd. E - what are all possible things to see - all possible derivations from E? Add strings derivable from E - closure function
- A -> cdE. = reduce to A and go to another state

2. Compute functions closure and go to

- will be used to determine the action and go to parts of the parsing table
  - Closure - essentially defines what is expected
  - Go to - moves from one state to another by consuming symbol

Closure (I) where I is a set of items - form states

Let N be non-terminal
- if distinguished point is in front of N then add each production for that N & put distinguished point at the beginning of the rhs
  - A -> α . B γ is in I → we expect to see a string derivable from B
  - B → γ is added to the closure, where B → γ is a production
  - Apply rule until nothing is added
Example:

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow E + T \\
E & \rightarrow T \\
T & \rightarrow id \mid (E)
\end{align*}
\]

Assume \( I = \{ S \rightarrow E \} \)

Closure \( I = \{ S \rightarrow E \}
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow T \\
T & \rightarrow id \\
T & \rightarrow (E)
\end{align*}
\]

Example:

- Go To:

  - go to(I, X), where X is a grammar symbol,
  - \( I \) = set of items
  - shift action moves from one state to another by absorbing single symbol. Successor states will contain each item with distinguished point advanced by 1 grammar symbol.
  - if \( A \rightarrow \alpha \cdot X \cdot \beta \) is in \( I \) then closure of \( A \rightarrow \alpha \cdot X \cdot \beta \) is added to go to(I, X)

Sets are viable prefixes if

- if \( \gamma \) is a viable prefix for \( I \)
  - then \( \gamma \) is a viable prefix to go to(I, X)

Example

- go to (I, ( ) = closure (T \( \rightarrow \) (E)))
Procedure items (C). C is set of items - state begin

C := closure (S ’ -> . S )
repeat
for each set of items I in C and each grammar symbol X
such that go to(I, X) is not empty and not in C do
   add go to (I, X) to C
until no more sets can added

Example: $S \rightarrow E$
$E \rightarrow E + T | T$
$T \rightarrow id | ( E )$

$S_0 = \{ \text{closure } [S \rightarrow E.] \}$
$S_1 = \{ S \rightarrow E., S \rightarrow E. + T \}$
$S_2 = \{ E \rightarrow T. \}$
$S_3 = \{ T \rightarrow id. \}$
$S_4 = \{ T \rightarrow (E), E \rightarrow .E + T, E \rightarrow .T, T \rightarrow .id, T \rightarrow .(E) \}$

DFA for Grammar
From DFA build the Parser Table

**ACTION** [state, input symbol/terminal symbol]

**GOTO** [state, non-terminal symbol]

**ACTION:**

1. If \( A \rightarrow \alpha \cdot a \cdot \beta \) is in \( S_i \) and \( a \) is a terminal and
   \( \text{goto} (S_i, a) = S_j \) then \( \text{ACTION}[S_i, a] = \text{shift} j \)

2. If \( [A \rightarrow \alpha \cdot :] \) is in \( S_i \), then
   \( \text{ACTION}[S_i, a] = \text{reduce} A \rightarrow \alpha \) for all \( a \) in \( \text{Follow} (A) \)

- **If no conflicts in 1 & 2 - then SLR(1) grammar**

3. If \( [S' \rightarrow S_0 .] \) is in \( S_i \), then
   Action \( [S_i, \] = accept

**GOTO for A non-terminal**

1. If goto \( (S_i, A) = S_j \) then \( \text{Goto} [S_i, A] = j \)

2. all entries not filled are errors

---

**Grammar**

1. \( S \rightarrow E \)

2. \( E \rightarrow E + T \)

3. \( E \rightarrow T \)

4. \( T \rightarrow \text{id} \)

5. \( T \rightarrow (E) \)

---

**Action**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
</tr>
<tr>
<td>S1</td>
<td>S7</td>
<td>S8</td>
<td>S9</td>
<td>S5</td>
</tr>
<tr>
<td>S2</td>
<td>x1</td>
<td>x2</td>
<td>x3</td>
<td>x4</td>
</tr>
<tr>
<td>S3</td>
<td>S4</td>
<td>S5</td>
<td>S6</td>
<td>S7</td>
</tr>
<tr>
<td>S4</td>
<td>x5</td>
<td>x6</td>
<td>x7</td>
<td>x8</td>
</tr>
<tr>
<td>S5</td>
<td>x9</td>
<td>x10</td>
<td>x11</td>
<td>x12</td>
</tr>
<tr>
<td>S6</td>
<td>x13</td>
<td>x14</td>
<td>x15</td>
<td>x16</td>
</tr>
<tr>
<td>S7</td>
<td>x17</td>
<td>x18</td>
<td>x19</td>
<td>x20</td>
</tr>
<tr>
<td>S8</td>
<td>x21</td>
<td>x22</td>
<td>x23</td>
<td>x24</td>
</tr>
</tbody>
</table>

**GoTo**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
</tr>
<tr>
<td>S2</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S4</td>
<td>S5</td>
<td>S6</td>
</tr>
<tr>
<td>S4</td>
<td>S5</td>
<td>S6</td>
<td>S7</td>
</tr>
<tr>
<td>S5</td>
<td>S6</td>
<td>S7</td>
<td>S8</td>
</tr>
<tr>
<td>S6</td>
<td>8</td>
<td>S7</td>
<td>S8</td>
</tr>
<tr>
<td>S7</td>
<td>S8</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>S8</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

---
LR Parsing Algorithm

Let I = w$ be initial input
Let j = 0
Let DFA state 1 have item S' ⇒ .S
Let stack = [S, 0]
repeat
  case action[top_state(stack), I[j]] of
    shift k: push (I[j++], k)
    reduce X ⇒ α:
      pop | α | pairs, (symbol,state)
      push (X, Goto[X,top_state(stack)])
    accept: halt normally
    error: halt and report error

Grammar

1. S → E
2. E → E + T
3. E → T
4. T → id
5. T → (E)

Non-term | Follow
---------|---------
S        | $ S
E        | + $ S
T        | + $ S
id       |

Action

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td>S5</td>
<td>S6</td>
<td>S7</td>
<td>S8</td>
<td>S9</td>
</tr>
<tr>
<td>2</td>
<td>S10</td>
<td>S11</td>
<td>S12</td>
<td>S13</td>
<td>S14</td>
</tr>
<tr>
<td>3</td>
<td>S15</td>
<td>S16</td>
<td>S17</td>
<td>S18</td>
<td>S19</td>
</tr>
<tr>
<td>4</td>
<td>S20</td>
<td>S21</td>
<td>S22</td>
<td>S23</td>
<td>S24</td>
</tr>
<tr>
<td>5</td>
<td>S25</td>
<td>S26</td>
<td>S27</td>
<td>S28</td>
<td>S29</td>
</tr>
</tbody>
</table>

GoTo

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td>S5</td>
<td>S6</td>
<td>S7</td>
<td>S8</td>
<td>S9</td>
</tr>
<tr>
<td>2</td>
<td>S10</td>
<td>S11</td>
<td>S12</td>
<td>S13</td>
<td>S14</td>
</tr>
<tr>
<td>3</td>
<td>S15</td>
<td>S16</td>
<td>S17</td>
<td>S18</td>
<td>S19</td>
</tr>
<tr>
<td>4</td>
<td>S20</td>
<td>S21</td>
<td>S22</td>
<td>S23</td>
<td>S24</td>
</tr>
<tr>
<td>5</td>
<td>S25</td>
<td>S26</td>
<td>S27</td>
<td>S28</td>
<td>S29</td>
</tr>
</tbody>
</table>
Power added to DFA - Return to state where non-terminal was predicted and continue - do this by counting states

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_0</td>
<td>id</td>
<td>r_3, goto[S_0, E]</td>
</tr>
<tr>
<td>S_O</td>
<td>id</td>
<td>r_3, goto[S_0, E]</td>
</tr>
<tr>
<td>S_T</td>
<td>id</td>
<td>S_7, goto[S_7, T]</td>
</tr>
<tr>
<td>S_E</td>
<td>id</td>
<td>S_7, goto[S_7, E]</td>
</tr>
<tr>
<td>S_O</td>
<td>+ id</td>
<td>S_7, goto[S_7, +]</td>
</tr>
<tr>
<td>S_E</td>
<td>+ id</td>
<td>S_7, goto[S_7, E]</td>
</tr>
<tr>
<td>S_0</td>
<td>+ id</td>
<td>accept</td>
</tr>
</tbody>
</table>

Diagram:

[Diagram showing transitions between states labeled with symbols like 'id', '+', and 'T'.]