Agenda

- LR(0) state construction
- FIRST, FOLLOW, and nullable
- Variations: SLR, LR(1), LALR
LR State Machine

- Idea: Build a DFA that recognizes handles
  - Language generated by a CFG is generally not regular, but
  - Language of handles for a CFG is regular
    - So a DFA can be used to recognize handles
  - Parser reduces when DFA accepts
Prefixes, Handles, &c (review)

- If $S$ is the start symbol of a grammar $G$,
  - If $S \Rightarrow^* \alpha$ then $\alpha$ is a **sentential form** of $G$
  - $\gamma$ is a **viable prefix** of $G$ if there is some derivation $S \Rightarrow^*_r \alpha Aw \Rightarrow^*_r \alpha \beta w$ and $\gamma$ is a prefix of $\alpha \beta$.
  - The occurrence of $\beta$ in $\alpha \beta w$ is a **handle** of $\alpha \beta w$

- An **item** is a marked production (at some position in the right hand side)
  - $[A ::= . XY]$  $[A ::= X . Y]$  $[A ::= XY .]$
Building the LR(0) States

- Example grammar

  - $S' ::= S$  
  - $S ::= ( L )$  
  - $S ::= x$  
  - $L ::= S$  
  - $L ::= L, S$

- We add a production $S'$ with the original start symbol followed by end of file ($$)

- Question: What language does this grammar generate?
Start of LR Parse

Initially
- Stack is empty
- Input is the right hand side of $S'$, i.e., $S$
- Initial configuration is $[S' ::= . S]$ 
- But, since position is just before $S$, we are also just before anything that can be derived from $S$
Initial state

A state is just a set of items

- Start: an initial set of items
- Completion (or closure): additional productions whose left hand side appears to the right of the dot in some item already in the state

0. $S'::= S$
1. $S::= ( L )$
2. $S::= x$
3. $L::= S$
4. $L::= L , S$
Shift Actions (1)

- To shift past the $x$, add a new state with the appropriate item(s)
  - In this case, a single item; the closure adds nothing
  - This state will lead to a reduction since no further shift is possible
Shift Actions (2)

- If we shift past the (, we are at the beginning of \( L \)
- the closure adds all productions that start with \( L \), which requires adding all productions starting with \( S \)
Once we reduce $S$, we’ll pop the rhs from the stack exposing the first state. Add a *goto* transition on $S$ for this.
Basic Operations

- **Closure** ($S$)
  - Adds all items implied by items already in $S$

- **Goto** ($I, X$)
  - $I$ is a set of items
  - $X$ is a grammar symbol (terminal or non-terminal)
  - **Goto** moves the dot past the symbol $X$ in all appropriate items in set $I$
Closure Algorithm

\[ Closure(S) = \]

repeat

for any item \([A ::= \alpha \cdot X \beta] \text{ in } S\]

for all productions \(X ::= \gamma\)

add \([X ::= \cdot \gamma]\) to \(S\)

until \(S\) does not change

return \(S\)
Goto Algorithm

- \( \text{Goto} (I, X) = \)
  - set \( new \) to the empty set
  - for each item \([A ::= \alpha . \ X \ \beta]\) in \( I\)
    - add \([A ::= \alpha \ X . \ \beta]\) to \( new \)
  - return \( \text{Closure}(new) \)

- This may create a new state, or may return an existing one
LR(0) Construction

- First, augment the grammar with an extra start production $S'::= S$.
- Let $T$ be the set of states.
- Let $E$ be the set of edges.
- Initialize $T$ to $\text{Closure}( [S'::= S \cdot S ] )$.
- Initialize $E$ to empty.
LR(0) Construction Algorithm

repeat
  for each state $I$ in $T$
    for each item $[A ::= \alpha \cdot X \beta]$ in $I$
      Let new be $\text{Goto}(I, X)$
      Add new to $T$ if not present
      Add $I \xrightarrow{X} \text{new}$ to $E$ if not present
  until $E$ and $T$ do not change in this iteration

- Footnote: For symbol $\$$, we don’t compute $\text{goto}(I, \$$); instead, we make this an accept action.
LR(0) Reduce Actions

Algorithm:
  Initialize $R$ to empty
  for each state $I$ in $T$
    for each item $[A ::= \alpha .]$ in $I$
      add $(I, A ::= \alpha)$ to $R$
Building the Parse Tables (1)

- For each edge $I \xrightarrow{x} J$
  - if $X$ is a terminal, put $s_j$ in column $X$, row $I$ of the action table (shift to state $j$)
  - If $X$ is a non-terminal, put $g_j$ in column $X$, row $I$ of the goto table
Building the Parse Tables (2)

- For each state $I$ containing an item $[S' ::= S \cdot \ ]$, put accept in column $\$ of row $I$

- Finally, for any state containing $[A ::= \gamma \ ]$ put action $rn$ in every column of row $I$ in the table, where $n$ is the production number
Example: States for

1. $S ::= (L)$
2. $S ::= x$
3. $L ::= S$
4. $L ::= L, S$

Diagram:

- $S ::= S$
- $S ::= .(L)$
- $S ::= .x$
- $L ::= .S$
- $L ::= .L, S$
- $S ::= .(L)$
- $S ::= .x$
- $L ::= L, S$
- $L ::= L, S$
**Example: Tables for**

<table>
<thead>
<tr>
<th></th>
<th>( )</th>
<th>x</th>
<th>,</th>
<th>8</th>
<th>s</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td></td>
<td></td>
<td>s2</td>
<td></td>
<td>s4</td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>g9</td>
</tr>
<tr>
<td>8</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

0. $S' ::= S$
1. $S ::= ( S )$
2. $S ::= x$
3. $L ::= S$
4. $L ::= L, S$
Where Do We Stand?

- We have built the LR(0) state machine and parser tables
  - No lookahead yet
  - Different variations of LR parsers add lookahead information, but basic idea of states, closures, and edges remains the same
A Grammar that is not LR(0)

- Build the state machine and parse tables for a simple expression grammar

\[
S ::= E$
\]
\[
E ::= T + E$
\]
\[
E ::= T$
\]
\[
T ::= x$
\]
LR(0) Parser for

0. $S ::= E$
1. $E ::= T + E$
2. $E ::= T$
3. $T ::= x$

- State 3 is has two possible actions on +
  - shift 4, or reduce 2
- $::$ Grammar is not LR(0)
SLR Parsers

- Idea: Use information about what can follow a non-terminal to decide if we should perform a reduction
- Easiest form is SLR – Simple LR
- So we need to be able to compute \( \text{FOLLOW}(A) \) – the set of symbols that can follow \( A \) in any possible derivation
  - But to do this, we need to compute \( \text{FIRST}(\gamma) \) for strings \( \gamma \) that can follow \( A \)
Calculating FIRST(γ)

- Sounds easy... If γ = XYZ, then FIRST(γ) is FIRST(X), right?

- But what if we have the rule X ::= ε?
- In that case, FIRST(γ) includes anything that can follow an X – i.e. FOLLOW(X)
FIRST, FOLLOW, and nullable

- `nullable(\(X\))` is true if \(X\) can derive the empty string.
- Given a string \(\gamma\) of terminals and non-terminals, `FIRST(\(\gamma\))` is the set of terminals that can begin strings derived from \(\gamma\).
- `FOLLOW(\(X\))` is the set of terminals that can immediately follow \(X\) in some derivation.
- All three of these are computed together.
Computing FIRST, FOLLOW, and nullable (1)

- Initialization
  - set FIRST and FOLLOW to be empty sets
  - set nullable to false for all non-terminals
  - set FIRST[a] to a for all terminal symbols a
Computing FIRST, FOLLOW, and nullable (2)

repeat
  for each production \( X ::= Y_1 Y_2 \ldots Y_k \)
    if \( Y_1 \ldots Y_k \) are all nullable (or if \( k = 0 \))
      set nullable[\( X \)] = true
    for each \( i \) from 1 to \( k \) and each \( j \) from \( i + 1 \) to \( k \)
      if \( Y_1 \ldots Y_{i-1} \) are all nullable (or if \( i = 1 \))
        add FIRST[\( Y_i \)] to FIRST[\( X \)]
      if \( Y_{i+1} \ldots Y_k \) are all nullable (or if \( i = k \))
        add FOLLOW[\( X \)] to FOLLOW[\( Y_i \)]
      if \( Y_{i+1} \ldots Y_{i+1} \) are all nullable (or if \( i+1=j \))
        add FIRST[\( Y_j \)] to FOLLOW[\( Y_i \)]
  Until FIRST, FOLLOW, and nullable do not change
Example

Grammar

- $Z ::= d$
- $Z ::= X Y Z$
- $Y ::= \varepsilon$
- $Y ::= c$
- $X ::= Y$
- $X ::= a$

<table>
<thead>
<tr>
<th>nullable</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>${a, c}$</td>
<td>${c, d, a}$</td>
</tr>
<tr>
<td>true</td>
<td>${c}$</td>
<td>${d, a, c}$</td>
</tr>
<tr>
<td>true</td>
<td>${d, a, c}$</td>
<td>${}$</td>
</tr>
</tbody>
</table>
SLR Construction

- This is identical to LR(0) – states, etc., except for the calculation of reduce actions

- Algorithm:
  Initialize \( R \) to empty
  for each state \( I \) in \( T \)
    for each item \([A ::= \alpha .]\) in \( I \)
      for each terminal \( a \) in \text{FOLLOW}(A)
        add \((I, a, A ::= \alpha)\) to \( R \)
  i.e., reduce \( \alpha \) to \( A \) in state \( I \) only on lookahead \( a \)
SLR Parser for

0. $S ::= E$
1. $E ::= T + E$
2. $E ::= T$
3. $T ::= x$

1. $S ::= E$
2. $S ::= E$
3. $E ::= T + E$
4. $E ::= T$
5. $T ::= x$
6. $E ::= T$

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>+</th>
<th>$</th>
<th>E</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s5</td>
<td></td>
<td></td>
<td>g2</td>
<td>g3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r2</td>
<td>s4</td>
<td>r2</td>
<td>g6</td>
<td>g3</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td></td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>6</td>
<td>r1</td>
<td></td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
</tbody>
</table>
On To LR(1)

- Many practical grammars are SLR
- LR(1) is more powerful yet
- Similar construction, but notion of an item is more complex, incorporating lookahead information
LR(1) Items

- An LR(1) item \([A ::= \alpha . \beta, a]\) is
  - A grammar production \((A ::= \alpha\beta)\)
  - A right hand side position (the dot)
  - A lookahead symbol (a)

- Idea: This item indicates that \(\alpha\) is the top of the stack and the next input is derivable from \(\beta a\).

- Full construction: see the book
LR(1) Tradeoffs

- LR(1)
  - Pro: extremely precise; largest set of grammars
  - Con: potentially very large parse tables with many states
LALR(1)

- Variation of LR(1), but merge any two states that differ only in lookahead
- Example: these two would be merged

\[
\begin{align*}
[A ::= x \ . \ , a] \\
[A ::= x \ . \ , b]
\end{align*}
\]
LALR(1) vs LR(1)

- LALR(1) tables can have many fewer states than LR(1)
- LALR(1) may have reduce conflicts where LR(1) would not (but in practice this doesn’t happen often)
Language Heirarchies
Coming Attractions

- LL(k) Parsing – Top-Down
- Recursive Descent Parsers
  - What you can do if you need a parser in a hurry