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
  - 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 sentence form of $G$
  - $\gamma$ is a viable prefix of $G$ if there is some derivation $S \Rightarrow^* \alpha$, $\alpha \Rightarrow^* \alpha \beta w$ 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 (a . at some position in the right hand side)
    - $[A ::= . X Y]$ $[A ::= X . Y]$ $[A ::= X Y .]$

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 $S$
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

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 \( S \)

Goto Actions
- 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 \([A ::= \alpha \cdot X\beta] \) in \( S \)
      - for all productions \(X ::= \gamma\)
        - add \([X ::= \gamma]\) to \( S \)
    - until \( S \) does not change
  - return \( S \)
**Goto Algorithm**

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

  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 \(Closure ( [S' ::= . S \, \$] )\)
- Initialize \(E\) to empty

**LR(0) Algorithm**

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

  Footnote: For symbol \(\$\), we don’t compute \(goto (I, \$)\); 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 \(I \rightarrow X \rightarrow J\)
  - 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 \([S' ::= S . \$]\), put \(accept\) in column \(S\) of row \(I\)
- Finally, for any state containing \([A ::= \gamma . ]\) put action \(r_n\) in every column of row \(n\) in the table, where \(n\) is the production number
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
  \[ S ::= E \$
  E ::= T + E \\
  E ::= T \\
  T ::= x \]

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 = X Y Z, 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 – FOLLOW(X)

FIRST, FOLLOW, and nullable

- nullable(X) is true if X can derive the empty string
- Given a string γ of terminals and non-terminals, FIRST(γ) is the set of terminals that can begin strings derived from γ.
- 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 ::= Y1 Y2 … Yk
    if Y1 … Yk 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 Y1 … Yi-1 are all nullable (or if i = 1)
        add FIRST[Yi] to FIRST[X]
      if Yi+1 … Yk are all nullable (or if i = k)
        add FOLLOW[X] to FOLLOW[Yi]
      if Yi+1 … Yj-1 are all nullable (or if i+1=j)
        add FIRST[Yj] to FOLLOW[Yi]
  Until FIRST, FOLLOW, and nullable do not change

Example

- Grammar

<table>
<thead>
<tr>
<th>Terminal</th>
<th>Nullable</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z ::= d</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z ::= X Y Z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y ::= ε</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y ::= c</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X ::= Y</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X ::= a</td>
<td>z</td>
<td></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 ::= α.] in I
      for each terminal a in FOLLOW(A)
        add (I, a, A ::= α.) to R
  i.e., reduce α to A in state I only on lookahead a
SLR Parser for

0. \[ S ::= E \]
1. \[ E ::= T + E \]
2. \[ E ::= T \]
3. \[ T ::= x \]

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 \([A ::= \alpha \cdot \beta, a]\) 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 \(\beta\).
- Full construction: see the book

LR(1) Tradeoffs

- LR(1)
  - Pro: extremely precise; largest class 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
  - \([A ::= x \cdot a]\)
  - \([A ::= x \cdot b]\)

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

ambiguous grammars

unambiguous grammars

LL(k)

LR(k)

LR(1)

LALR(1)

SLR

LR(0)

LR(0)

LL(0)

LL(k) Parsing – Top-Down

Recursive Descent Parsers

What to do if you need a parser in a hurry

Coming Attractions