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

1. $S ::= E$
2. $E ::= T + E$
3. $E ::= T$
4. $T ::= x$
5. $E ::= T + \cdot E$
6. $E ::= \cdot T + E$
7. $E ::= \cdot T$
8. $T ::= \cdot x$

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
- We need to be able to compute FOLLOW(A) – the set of symbols that can follow A in any possible derivation
  - i.e., t is in FOLLOW(A) if any derivation contains At
  - To compute this, we need to compute FIRST(γ) for strings γ that can follow A
Calculating FIRST(\(\gamma\))

- Sounds easy... If \(\gamma = XYZ\), then \(\text{FIRST}(\gamma)\) is \(\text{FIRST}(X)\), right?

- But what if we have the rule \(X ::= \varepsilon\)?
- In that case, \(\text{FIRST}(\gamma)\) includes anything that can follow an \(X\) – i.e. \(\text{FOLLOW}(X)\)
FIRST, FOLLOW, and nullable

- **nullable(χ)** is true if χ 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(χ)** is the set of terminals that can immediately follow χ 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(X) false for all non-terminals X
  - 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 $Y_{i+1} \ldots Y_{j-1}$ 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

1) $Z ::= d$
2) $Z ::= XYZ$
3) $Y ::= \varepsilon$
4) $Y ::= c$
5) $X ::= Y$
6) $X ::= a$

<table>
<thead>
<tr>
<th>Nullable</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>no</td>
<td>$a_{(6)}^{(5)}$</td>
</tr>
<tr>
<td>yes</td>
<td>$c_{(1)}^{(5)}$</td>
<td>$a_{(2)}^{(2)}$</td>
</tr>
</tbody>
</table>
LR(0) Reduce Actions

- In an LR(0) parser, if a state contains a reduction, it is unconditional regardless of the next input symbol
- Algorithm:
  
  Initialize $R$ to empty
  
  for each state $I$ in $T$
    
    for each item $[A ::= \alpha.]$ in $I$
      
      add $(I, A ::= \alpha)$ to $R$
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 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

[Diagram of LL(1) parser with production rules and LR(1) action and goto tables]
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 \(\beta a\).

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

- LR(1)
  - Pro: extremely precise; largest set of grammars / languages
  - 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
    \[ A ::= x . , a \]
    \[ A ::= x . , b \]
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)
- Most practical bottom-up parser tools are LALR(1) (e.g., yacc, bison, CUP)
Language Heirarchies

Diagram showing the relationship between different types of grammars, with unambiguous grammars on the left and ambiguous grammars on the right. The types of grammars include LL(k), LR(k), LL(1), LR(1), LALR(1), SLR, LL(0), and LR(0).