LR Parsing

- Read tokens left to right (L)
- Create a rightmost derivation (R)
- How is this possible?
- Start at the bottom (left) and work your way up
- Last step has only one non-terminal to be replaced so is right-most
- Working backwards, replace mixed strings by non-terminals
- Always proceed so that there are no non-terminals to the right of the string to be replaced

Example: \[ \text{<Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>} \]

\[ \text{<Sum> = >} \]

\[ = \cdot (0 + 1) + 0 \quad \text{shift} \]

Example: \[ \text{<Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>} \]

\[ \text{<Sum> = >} \]

\[ = (0 + 1) + 0 \quad \text{shift} \]
\[ = (0 + 1) + 0 \quad \text{shift} \]
Example: \( \text{<Sum>} = 0 \mid 1 \mid (<\text{Sum}>), \text{<Sum>} + \text{<Sum>} \)

\[
\text{<Sum>} =>
\]
\[
= (\text{<Sum>} + \bullet 1) + 0 \quad \text{shift}
\]
\[
= (\text{<Sum>} \bullet + 1) + 0 \quad \text{shift}
\]
\[
=> (0 \bullet + 1) + 0 \quad \text{reduce}
\]
\[
= (0 + 1) + 0 \quad \text{shift}
\]
\[
= \bullet (0 + 1) + 0 \quad \text{shift}
\]

Example: \( \text{<Sum>} = 0 \mid 1 \mid (<\text{Sum}>), \text{<Sum>} + \text{<Sum>} \)

\[
\text{<Sum>} =>
\]
\[
= (\text{<Sum>} + \bullet 1) + 0 \quad \text{shift}
\]
\[
= (\text{<Sum>} \bullet + 1) + 0 \quad \text{shift}
\]
\[
=> (0 \bullet + 1) + 0 \quad \text{reduce}
\]
\[
= (0 + 1) + 0 \quad \text{shift}
\]
\[
= \bullet (0 + 1) + 0 \quad \text{shift}
\]

Example: \( \text{<Sum>} = 0 \mid 1 \mid (<\text{Sum}>), \text{<Sum>} + \text{<Sum>} \)

\[
\text{<Sum>} =>
\]
\[
= (\text{<Sum>} \bullet + 1) + 0 \quad \text{reduce}
\]
\[
= (\text{<Sum>} + \bullet 1) + 0 \quad \text{reduce}
\]
\[
=> (\text{<Sum>} + 1 \bullet) + 0 \quad \text{reduce}
\]
\[
= (\text{<Sum>} \bullet + 1) + 0 \quad \text{shift}
\]
\[
= (\text{<Sum>} \bullet + 1) + 0 \quad \text{shift}
\]
\[
=> (0 \bullet + 1) + 0 \quad \text{reduce}
\]
\[
= (0 + 1) + 0 \quad \text{shift}
\]
\[
= \bullet (0 + 1) + 0 \quad \text{shift}
\]
Example: \( \langle \text{Sum} \rangle = 0 | 1 | (\langle \text{Sum} \rangle) \)
| \( \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \)

\[
\begin{align*}
\langle \text{Sum} \rangle & \Rightarrow \\
& = \langle \text{Sum} \rangle + 0 \quad \text{shift} \\
& = \langle \text{Sum} \rangle + 0 \quad \text{shift} \\
& = (\langle \text{Sum} \rangle + 0) \quad \text{reduce} \\
& = (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle \cdot 1) + 0 \quad \text{reduce} \\
& = (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle + 1) + 0 \quad \text{shift} \\
& = ((0 + 1) + 0) \quad \text{reduce} \\
& = (0 + 1) + 0 \quad \text{shift} \\
& = (0 + 1) + 0 \quad \text{shift}
\end{align*}
\]
Example

(0 + 1) + 0

Example

Example

Example

Example

Example

Example

Example

Example
LR Parsing Tables

- Build a pair of tables, Action and Goto, from the grammar
  - This is the hardest part, we omit here
  - Rows labeled by states
  - For Action, columns labeled by terminals and “end-of-token” marker
    - (more generally strings of terminals of fixed length)
  - For Goto, columns labeled by non-terminals

Action and Goto Tables

- Given a state and the next input, Action table says either
  - **shift** and go to state \( n \), or
  - **reduce** by production \( k \) (explained in a bit)
  - **accept** or **error**
- Given a state and a non-terminal, Goto table says
  - go to state \( m \)

LR(i) Parsing Algorithm

- Based on push-down automata
- Uses states and transitions (as recorded in Action and Goto tables)
- Uses a stack containing states, terminals and non-terminals

- 0. Insure token stream ends in special “end-of-token” symbol
- 1. Start in state 1 with an empty stack
- 2. Push **state**(1) onto stack
- 3. Look at next \( i \) tokens from token stream (\( toks \)) (don’t remove yet)
- 4. If top symbol on stack is **state**(\( n \)), look up action in Action table at \( (n, toks) \)
- 5. If action = **shift** \( m \),
  - a) Remove the top token from token stream and push it onto the stack
  - b) Push **state**(\( m \)) onto stack
  - c) Go to step 3
LR(i) Parsing Algorithm

6. If action = reduce \( k \) where production \( k \) is \( E ::= u \)
   a) Remove \( 2 \times \text{length}(u) \) symbols from stack (\( u \) and all the interleaved states)
   b) If new top symbol on stack is \( \text{state}(m) \), look up new state \( p \) in \( \text{Goto}(m,E) \)
   c) Push \( E \) onto the stack, then push \( \text{state}(p) \) onto the stack
   d) Go to step 3

7. If action = accept
   - Stop parsing, return success
8. If action = error,
   - Stop parsing, return failure

Adding Synthesized Attributes

- Add to each reduce a rule for calculating the new synthesized attribute from the component attributes
- Add to each non-terminal pushed onto the stack, the attribute calculated for it
- When performing a reduce,
  - gather the recorded attributes from each non-terminal popped from stack
  - Compute new attribute for non-terminal pushed onto stack

Example: \(<\text{Sum}> = 0 | 1 | (<\text{Sum}> | \text{<Sum>} + \text{<Sum>}\)

\[
\begin{array}{ll}
\text{0 + 1 + 0} & \text{shift} \\
\rightarrow 0 & \text{+ 1 + 0} & \text{reduce} \\
\rightarrow <\text{Sum}> & \text{● + 1 + 0} & \text{shift} \\
\rightarrow <\text{Sum}> & + \text{● + 0} & \text{shift} \\
\rightarrow <\text{Sum}> + 1 & \text{● + 0} & \text{reduce} \\
\rightarrow <\text{Sum}> + <\text{Sum}> & \text{● + 0} \\
\end{array}
\]

Shift-Reduce Conflicts

- **Problem**: can’t decide whether the action for a state and input character should be **shift** or **reduce**
- Caused by ambiguity in grammar
- Usually caused by lack of associativity or precedence information in grammar

Example - cont

- **Problem**: shift or reduce?
  - You can shift-shift-reduce-reduce or reduce-shift-shift-reduce
  - Shift first - right associative
  - Reduce first- left associative
Reduce - Reduce Conflicts

- **Problem**: can’t decide between two different rules to reduce by
- Again caused by ambiguity in grammar
- **Symptom**: RHS of one production suffix of another
- Requires examining grammar and rewriting it
- Harder to solve than shift-reduce errors

---

Example

- **S ::= A | aB**
  - A ::= abc
  - B ::= bc

  - abc  shift
  - a bc  shift
  - ab c  shift
  - abc  

- Problem: reduce by B ::= bc then by S ::= aB, or by A ::= abc then S::A?