LR(1) parsing (pg. 1)

I. Intro
   - Left-to-right
   - Right-to-left derivation
   - Look ahead usually k = 0

LR(0) - SLR, LALR
   - LR(0) easy to construct
   - More powerful, less practical

Advantages:
   - Powerful
   - Most general non-backtracking method known
   - Detect errors at earliest possible point
   - Class of grammars superset of grammars handled by predictive parsers

Drawbacks:
   - Hard to hand code
   - Hard to debug
   - Error recovery problematic

II. Table Construction

LR(k) automaton:
   - 2 basic moves: shift, reduce
   - 2 other moves: accept, error
   - Uses a stack to track parsing progress

Concepts:
   - An LR(0) item is a rule with a dot
     at some position in the RHS, e.g.:
     A → BC, A → BCD, A → BC
   - An LR(0) state is a set of items

Notation:
   - Lowercase letters: terminals
   - Uppercase letters: non-terminals
   - Greek letters: strings

Canonical LR(0) collection: basis of LR(0) deterministic finite automaton
   - Used to make parsing decisions

---

For grammar G, need:
A augmented grammar G'
B closure function
C goto function

A. For augmented grammar, just add rule S → S

B. Given a set of items I
   - closure(I):
     1. add I to closure of I
     2. if A → a. B. B. B. ε ∈ closure(I)
        then add B → a. ε to closure(I)
        keep applying until nothing can be added

Example:
1. E → E
2. E → T
3. T → T + F
4. T → F
5. F → (E)
6. F → id

Let I = {ε, E, E, E}

I₀ = closure(I₀) = \{ \{E → E, E\} \} kernel set
   \{E → T\}
   \{T → T + F\}
   \{T → F\}
   \{F → (E)\}
   \{F → id\} 3' form

Non-kernel items:
   \{A → a \}

---

G. Given a set of items I
   - In set of symbols following dot
     goto(I₀, ε) = closure(\{ \{A → a \} \})

I₁ = goto(I₀, E) = \{ \{E → E\} \}

I₂ = goto(I₁, T) = \{ \{E → T\} \}

I₃ = goto(I₂, F) = \{ \{T → F\} \}

I₄ = goto(I₃, C) = \{ \{F → (E)\} \}

I₅ = goto(I₄, id) = \{ \{E → id\} \}

I₆ = goto(I₅, id) = \{ \{E → id\} \}
Algorithm for constructing canonical LR(0) sets of items:

- Initialize \( C = \emptyset \) and queue \( E[\{s\to s\}] \}
- Repeat for all \( I \in C \), for all \( \alpha \)
  - If \( \text{goto}(I, \alpha) \neq \emptyset \) and \( \text{goto}(I, \alpha) \notin C \)
  - Add \( \text{goto}(I, \alpha) \) to \( C \)

In practice, and repeating items, e.g., \( \text{goto}(I, \alpha) \) happens to be in \( I_2 \)

A final item is an item with a dot at the end.

If all final items are in states by themselves, then the collection is LR(0); otherwise,
there is a shift-reduce conflict in LR(0) or a reduce-reduce conflict in LR(0).

Doing that yields an SLR(1) table.

Even in SRL(1), there still may be conflicts.
Since follow sets may overlap,
See handouts for graphical and table representations of the running example.

---

Example (see table from handout)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>+ 5 ( \epsilon )</td>
<td>56</td>
</tr>
<tr>
<td>55</td>
<td>+ 6 ( \epsilon )</td>
<td>56</td>
</tr>
<tr>
<td>56</td>
<td>+ 5 ( \epsilon )</td>
<td>57</td>
</tr>
<tr>
<td>57</td>
<td>+ 6 ( \epsilon )</td>
<td>57</td>
</tr>
<tr>
<td>56</td>
<td>+ 5 ( \epsilon )</td>
<td>57</td>
</tr>
<tr>
<td>57</td>
<td>+ 6 ( \epsilon )</td>
<td>57</td>
</tr>
</tbody>
</table>

---

Can use translation scheme propagating synthesized attributes during reduce actions
to build AST.

**LR Driver**

1. Push \( S_0 \) onto empty stack.
2. Read first input token.
3. Repeat:
   - Let \( S = \text{the symbol at the top of stack} \)
   - If \( \text{action}(S, \alpha) = \text{shift plus} \)
     - Push \( \alpha \) onto the stack.
     - Let \( \beta \) be next symbol.
   - Else if \( \text{action}(S, \alpha) = \text{reduce} A \Rightarrow \beta \)
     - Pop \( \beta \) states from stack.
     - Push \( \text{goto}(S_m = \beta, A) \) on stack.
     - Synthesize attributes for \( A \Rightarrow \beta \)
   - Else if \( \text{action}(S, \alpha) = \text{accept} \)
     - Break.
   - Else call error recovery.

The same LR driver works for all the variants of LR parse tables.
LR(1) parsing (p3)

IV LR(1), LALR(1)

An LR(1) item is a pair consisting of an LR(0) item and a look-ahead, e.g., \([A \to \alpha, \beta, a]\)

Starting point: \([S \to S, \$]\)

Example: \([E \to .E, E, \$]\)

\[
\text{Closure: } [E \to .E + T, \$, +]
\]
\[
[E \to .T, \$, +]
\]
\[
T \to .T + F, \$, +, *\]
\[
T \to .F, \$, +, +\]
\[
F \to (E), \$, +, *\]
\[
F \to .id, \$, +, *\]

General rule for an LR(1) canonical collection:

\[
\text{Closure(I)}:
\]
\[
\text{Repeat for all } [A \to \alpha, B \beta, a] \text{ in } I,
\]
\[
\text{for all } [B \to \gamma \text{ in } G^-],
\]
\[
\text{for all } b \text{ in } \text{first}(B, a)
\]
\[
\text{add } [B \to \gamma, b] \text{ to set I}
\]
\[
\text{until no more items can be added}
\]

\[
\text{Goto(I, \$)}:
\]
\[
J = \emptyset
\]
\[
\text{for all } [A \to \alpha, x \beta, a] \in I,
\]
\[
\text{add } [A \to \alpha x, \beta, a] \text{ to } J
\]
\[
\text{return closure}(J)
\]

Unlike with LR(0) => SLR step, no need
to deal with ambiguous states in LR(1)

For LALR, systematically merge certain
LR(1) steps, to get more compact table
to build LALR, given LR(0) state, consider
paths that lead to it: more efficient
parser table construction.