Example:
1) $E \rightarrow E + T$
2) $E \rightarrow T$
3) $T \rightarrow T \cdot T$
4) $T \rightarrow T$
5) $T \rightarrow ( E )$
6) $T \rightarrow \text{id}$

Let $I = \{ [ E \rightarrow . E ] \}$


$\text{given a set of items } I$

$\forall x \in \text{set of symbols}$, allowing dot

$GOTO(I, x) = \text{closure}(\{ [ A \rightarrow x \cdot x ] , [ A \rightarrow x , x ] \} \cup I)$

$I_1 = GOTO(I_0, E) = \{ [ E \rightarrow E ] , \{ [ E \rightarrow E + T ] , \{ [ E \rightarrow E , T ] , \{ [ T \rightarrow T + T ] , \{ [ T \rightarrow . T ] , \{ [ T \rightarrow ( E ) ] , \{ [ T \rightarrow \text{id} ] \} \}

$I_2 = GOTO(I_0, T) = \{ [ E \rightarrow E + T ] , \{ [ E \rightarrow E , T ] , \{ [ T \rightarrow T + T ] , \{ [ T \rightarrow . T ] , \{ [ T \rightarrow ( E ) ] , \{ [ T \rightarrow \text{id} ] \} \}


$I_4 = GOTO(I_0, \text{id}) = \{ [ E \rightarrow \text{id} ] , \{ [ E \rightarrow E + T ] , \{ [ E \rightarrow E , T ] , \{ [ T \rightarrow . T ] , \{ [ T \rightarrow ( E ) ] , \{ [ T \rightarrow \text{id} ] \} \}

Algorithm for constructing canonical LR(0) set of items:

1. Initialize $C = \{ \text{closure}(I_0) \}$
2. Repeat for all $I \in C, \forall x \in \text{set of symbols}$
   - If $GOTO(I, x) \neq \emptyset$ and $GOTO(I, x) \in C$
     - Add $GOTO(I, x)$ to $C$

In practice, find repeating item, e.g.

$GOTO(I_4, \text{id})$ happens to be $I_2$

A final item is an item with the 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 $LR(0)$

To fix, use follow sets to decide what to reduce

Doing that yields an SLR(1) table

Even in SLR(1), can still have conflicts, since follow sets may overlap

See hand-outs for graphical and table representations of the running example
LR parsing

(a_1, a_2, ... , a_n, $)

(input)

end of file

Action Goto

(parsing table A - F)

Stack = summary of a path in the automaton

LR parser configuration

S_0 S_1 ... S_m

a_1 a_2 ... a_n $

If next move is

a) shift a : move past symbol a in input, push S_j into stack

b) reduce A->B : pop |B| states from stack, push GOTO (S_m-H(B)) in stack

c) accept : Done!

d) error : I Quit!

Example (see table from hand-out)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_0</td>
<td>a+b+c $</td>
</tr>
<tr>
<td>S_0</td>
<td>S_5</td>
</tr>
<tr>
<td>S_0</td>
<td>S_5</td>
</tr>
<tr>
<td>S_0</td>
<td>S_5</td>
</tr>
<tr>
<td>S_0</td>
<td>S_1</td>
</tr>
<tr>
<td>S_0</td>
<td>S_1</td>
</tr>
<tr>
<td>S_0</td>
<td>S_1</td>
</tr>
<tr>
<td>S_0</td>
<td>S_1</td>
</tr>
<tr>
<td>S_0</td>
<td>S_1</td>
</tr>
<tr>
<td>S_0</td>
<td>S_1</td>
</tr>
<tr>
<td>S_0</td>
<td>S_1</td>
</tr>
<tr>
<td>S_0</td>
<td>S_1</td>
</tr>
</tbody>
</table>

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

The LR driver:

push S_0 onto empty stack

read first input token

repeat:

get s = the state at top of stack

if action (s, a) = shift t:

push t onto stack

let a be next symbol

else if action (s, a) = reduce A->B:

pop |B| states off stack

else:

if b is else clause

let t be new top of stack

push GOTO (E, T) on stack

synthesis attributes for A->B

else if action (s, a) = accept:

break

else:

call error recovery

The same LR driver works for all the variants of LR parse tables

IV LRC(0) LALR(1)

An LRC(0) item is a pair consisting of

an LRC(0) item and a look-ahead,

e.g., [A -> x . px b]

Starting point: [E] -> S_0, S_1, $]

Example: [E] -> E * T, S_2

Closure:

[E] -> E + T, S_3

[E] -> T S_4

[T] -> F S_5

[T] -> F S_6

[F] -> (E), S_7

General rule for an LRC(1) canonical collection:

Closure (I):

repeat for all [A -> x . px b], in I,

for all y in S_3, if for some x:

add [E] -> x_y b to set I

until no more items can be added

goto (I, x):

J = 0

for all [A -> x . px b, a] in I

add [A -> x_y px b] to J

return closure J

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

For LALR, systematically merge certain LRC(1) steps, to get more compact table

To build LALR, given LRC(0) state, consider path that led to it: more efficient parser table construction
Fig. 4.36. Transition diagram of DFA D for viable prefixes.


<table>
<thead>
<tr>
<th>State</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td>+</td>
</tr>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>s7</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>5</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>6</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>7</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>8</td>
<td>s6</td>
<td>s11</td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>s7</td>
</tr>
<tr>
<td>10</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>11</td>
<td>r5</td>
<td>r5</td>
</tr>
</tbody>
</table>

**Fig. 4.31.** Parsing table for expression grammar.

1. *s* means shift and stack state *i*,
2. *rj* means reduce by production numbered *j*,
3. *acc* means accept,