Mini PASCAL Grammar

program →
  program id ( identifier_list ) ;
  declarations
  subprogram_declarations
  compound_statement

identifier_list →
  id
  | identifier_list , id

declarations →
  declarations var identifier_list : type ;
  | ε

type →
  standard_type
  | array [ num .. num ] of standard_type

standard_type →
  integer
  | real

subprogram_declarations →
  subprogram_declarations subprogram_declaration ;
  | ε

subprogram_declaration →
  subprogram_head declarations compound_statement

subprogram_head →
  function id arguments : standard_type ;
  | procedure id arguments ;

arguments →
  ( parameter_list )
  | ε
parameter_list →
    identifier_list ; type
    | parameter_list ; identifier_list : type

compound_statement →
    begin
    optional_statements
    end

optional_statements →
    statement_list
    | ε

statement_list →
    statement
    | statement_list ; statement

statement →
    variable assignop expression
    | procedure_statement
    | compound_statement
    | if expression then statement else statement
    | while expression do statement

variable →
    id
    | id [ expression ]

procedure_statement →
    id
    | id ( expression_list )

eexpression_list →
    expression
    | expression_list , expression

eexpression →
    simple_expression
    | simple_expression relop simple_expression

simple_expression →
    term
\[
\begin{align*}
| \text{sign term} \\
| \text{simple-expression addop term}
\end{align*}
\]

term \rightarrow 
\begin{align*}
| & \text{factor} \\
| & \text{term mulop factor}
\end{align*}

factor \rightarrow 
\begin{align*}
| & \text{id} \\
| & \text{id ( expression_list )} \\
| & \text{num} \\
| & \text{( expression )} \\
| & \text{not factor}
\end{align*}

sign \rightarrow 
\begin{align*}
| & + \\
| & –
\end{align*}
Examples of Grammars

\( G_0: \quad E \rightarrow E + E \mid E \ast E \mid id \)

\( G_1: \quad E \rightarrow TE' \\
E' \rightarrow +TE' \mid \epsilon \\
T \rightarrow FT' \\
T' \rightarrow *FT' \mid \epsilon \\
F \rightarrow (E) \mid id \)

\( G_2: \quad E \rightarrow E + T \mid T \\
T \rightarrow T \ast F \mid F \\
F \rightarrow (E) \mid id \)

\( G_3: \quad E' \rightarrow E \\
E \rightarrow E + T \mid T \\
T \rightarrow T \ast F \mid F \\
F \rightarrow (E) \mid id \)

\( G_4: \quad S' \rightarrow S \\
S \rightarrow L = R \\
S \rightarrow R \\
L \rightarrow *R \\
L \rightarrow id \\
R \rightarrow L \)

\( G_5: \quad S' \rightarrow S \\
S \rightarrow aAd \mid bBd \mid aBe \mid bAe \\
A \rightarrow c \\
B \rightarrow c \)
1 Introduction

2 Lexical Analysis
3 Top-Down Parsing

3.1 Transform to Unambiguous Grammar

A grammar is called *ambiguous* if there is some sentence in its language for which there is more than one parse tree.

Example: \[ E \rightarrow E + E \mid E \ast E \mid id; \]

\[ w = id + id \ast id. \]

In general, we may not be able to determine which tree to use. In fact, determining whether a given arbitrary CFG is ambiguous or not is undecidable.

*Solution:*

(a) Transform the grammar to an equivalent unambiguous one.

(b) Use *disambiguating rule* with the ambiguous grammar to specify, for ambiguous cases, which parse tree to use.

3.1.1 if then else statement

For an input “if \( E_1 \) then if \( E_2 \) then \( S_1 \) else \( S_2 \),” two parse trees can be constructed; hence, \( G_1 \) is ambiguous. An unambiguous grammar \( G_2 \) which is equivalent to \( G_1 \) can be constructed as follows:

\[
G_2: \quad stmt \rightarrow matched_stmt \mid \text{unmatched_stmt} \\
matched_stmt \rightarrow \text{if exp then matched_stmt else matched_stmt} \\
\text{unmatched_stmt} \rightarrow \text{if exp then stmt else unmatched_stmt}
\]
3.2 Left-factoring and Removing left recursions

Consider the following grammar $G_1$ and a token string $w = bede$.

\[
G_1: \quad S \rightarrow ee \mid bAc \mid bAe \\
A \rightarrow d \mid eA
\]

Since the initial $b$ is in two production rules, $S \rightarrow bAc$ and $S \rightarrow bAe$, the parser cannot make a correct decision without backtracking. This problem may be solved to redesign the grammar as shown in $G_2$.

\[
G_2: \quad S \rightarrow ee \mid bAQ \\
Q \rightarrow c \mid e \\
A \rightarrow d \mid eA
\]

In $G_2$, we have factored out the common prefix $bA$ and used another non-terminal symbol $Q$ to permit the choice between the final $c$ and $a$. Such a transformation is called as left factorization or left factoring.

Now, consider the following grammar $G_3$ and consider a token string $w = id + id + id$.

\[
G_3: \quad E \rightarrow E + T \mid T \\
T \rightarrow T \ast F \mid F \\
F \rightarrow id \mid (E)
\]

A top-down parser for this grammar will start by expanding $E$ with the production $E \rightarrow E + T$. It will then expand $E$ in the same way. In the next step, the parser should expand $E$ by $E \rightarrow T$ instead of $E \rightarrow E + T$. But there is no way for the parser to know which choice it should make. In general, there is no solution to this problem as long as the grammar has productions of the form $A \rightarrow A\alpha$, called left-recursive productions. The solution to this problem is to rewrite the grammar in such a way to eliminate the left recursions. There are two types of left recursions: immediate left recursions, where the productions are of the form $A \rightarrow A\alpha$, and non-immediate left recursions, where the productions are of the form $A \rightarrow B\alpha; B \rightarrow A\beta$. In the latter case, $A$ will use $B\alpha$, and $B$ will use $A\beta$, resulting in the same problem as the immediate left recursions have.

We now have the following formal definition: “A grammar is left-recursive if it has a nonterminal $A$ such that there is a derivation $A \Rightarrow A\alpha$ for some string $\alpha$.”

Removing immediate left recursions:
Consider the above example $G_3$ in which two productions have left recursions. Applying the above algorithm to remove immediate left recursions, we have

(i) $E \rightarrow E + T \mid T$
   $\Rightarrow E \rightarrow TE'$
   $E' \rightarrow +TE' \mid \epsilon$

(ii) $T \rightarrow T * F \mid F$
    $\Rightarrow T \rightarrow FT'$
    $T' \rightarrow *FT' \mid \epsilon$

Now, we have following grammar $G_4$ which is equivalent to $G_3$:

$G_4: E \rightarrow TE'$
    $E' \rightarrow +TE' \mid \epsilon$
    $T \rightarrow FT'$
    $T' \rightarrow *FT' \mid \epsilon$
    $F \rightarrow (E) \mid id$

The following is an algorithm for eliminating all left recursions including non-immediate left recursions.
Algorithm: Eliminating left recursion.

Input: Grammar $G$ with no cycles or $\epsilon$-productions.

Output: An equivalent grammar with no left recursion.

1. Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2. for $i = 1$ to $n$ begin
   for $j = 1$ to $i - 1$ do begin
     replace each production of the form $A_i \rightarrow A_j \gamma$
     by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \cdots \mid \delta_k \gamma$.
     where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all the current $A_j$-productions;
   end
   eliminate the immediate left recursion among the $A_i$-productions
end
end.

3.2.1 Examples

EXAMPLE 1: Consider the following example:

\[ G : \begin{align*}
S & \rightarrow Ba \mid b \\
B & \rightarrow Bc \mid Sd \mid e
\end{align*} \]

Let $A_1 = S$ and $A_2 = B$. We then have,

\[ G : \begin{align*}
A_1 & \rightarrow A_2a \mid b \\
A_2 & \rightarrow A_2c \mid A_1d \mid e
\end{align*} \]

(i) i=1:
\[ A_1 \rightarrow A_2a \mid b, \text{ OK} \]

(ii) i=2:
\[ A_2 \rightarrow A_1d \text{ is replace by } A_2 \rightarrow A_2ad \mid bd \]
Now, $G$ becomes

\[ G : \begin{align*}
A_1 & \rightarrow A_2a \mid b \\
A_2 & \rightarrow A_2c \mid A_2ad \mid bd \mid e
\end{align*} \]
By eliminating immediate recursions in $A_2$-productions, we have

(i) $A_2 \rightarrow A_2c \mid bd \mid e$ are replaced by
$A_2 \rightarrow bdA_3$
$A_2 \rightarrow eA_3$
$A_3 \rightarrow cA_3 \mid \epsilon$

(ii) $A_2 \rightarrow A_2ad \mid bd \mid e$ are replaced by
$A_2 \rightarrow bdA_4$
$A_2 \rightarrow eA_4$
$A_3 \rightarrow adA_4 \mid \epsilon$

(i) and (ii) can be combined as

$A_2 \rightarrow bdA_3 \mid eA_3$
$A_3 \rightarrow cA_3 \mid adA_3 \mid \epsilon$

Therefore, we have

$S \rightarrow Ba \mid b$
$B \rightarrow bdD \mid eD$
$D \rightarrow cD \mid adD \mid \epsilon$

3.3 First and Follow Sets

Consider every string derivable from some sentential form $\alpha$ by a leftmost derivation. If $\alpha \Rightarrow^* \beta$, where $\beta$ begins with some terminal, then that terminal is in $FIRST(\alpha)$. 
Algorithm: Computing $FIRST(A)$.

1. If $A$ is a terminal, $FIRST(A) = \{A\}$.
2. If $A \to \epsilon$, add $\epsilon$ to $FIRST(A)$.
3. if $A \to Y_1Y_2 \cdots Y_k$, then
   \[
   \text{for } i = 1 \text{ to } k - 1 \text{ do}
   \]
   \[
   \text{if } [\epsilon \in FIRST(Y_1) \cap FIRST(Y_2) \cap \cdots \cap FIRST(Y_{i-1})] \text{ (i.e., } Y_1Y_2 \cdots Y_{i-1} \Rightarrow \epsilon) \text{ and }
   a \in FIRST(Y_i), \text{ then add } a \text{ to } FIRST(A).
   \]
   \[
   \text{end}
   \]
   \[
   \text{if } \epsilon \in FIRST(Y_1) \cap \cdots \cap FIRST(Y_k), \text{ then add } \epsilon \text{ to } FIRST(A).
   \]
   \[
   \text{end.}
   \]

Now, we define $FOLLOW(A)$ as the set of terminals that can come right after $A$ in any sentential form of $L(G)$. If $A$ comes at the end, then $FOLLOW(A)$ includes the end marker $\$).

Algorithm: Computing $FOLLOW(A)$.

1. $\$$ is in $FOLLOW(S)$.
2. if $A \to \alpha B \beta$, then $FIRST(\beta) - \{\epsilon\} \subseteq FOLLOW(B)$.
3. if $A \to \alpha B$ or $A \to \alpha B \beta$ where $\epsilon \in FIRST(\beta)$ (i.e., $\beta \Rightarrow \epsilon$),
   \[
   FOLLOW(A) \subseteq FOLLOW(B)
   \]
   \[
   \text{end.}
   \]

Note: In Step 3, $FOLLOW(B) \not\subseteq FOLLOW(A)$. To prove this, consider the following example:
$S \to Ab \mid Bc; A \to aB; B \to c$. Clearly, $c \in FOLLOW(B)$ but $c \not\in FOLLOW(A)$.

3.3.1 EXAMPLE:

For the grammar $G_4$,

$G_4:$
\[
E \to TE'
\]
\[
E' \to +TE' | \epsilon
\]
\[
T \to FT'
\]
\[
T' \to *FT' | \epsilon
\]
\[
F \to (E) | id
\]
\[ \text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \text{id}\} \]
\[ \text{FIRST}(E') = \{+, \epsilon\} \]
\[ \text{FIRST}(T') = \{*, \epsilon\} \]
\[ \text{FOLLOW}(E) = \text{FOLLOW}(E') = \{\}, \$\} \]
\[ \text{FOLLOW}(T) = \text{FOLLOW}(T') = \{+, \$\} \]
\[ \text{FOLLOW}(F) = \{+, *, \}, \$\} \]

### 3.4 Constructing a predictive parser

**Algorithm:** Predictive parser construction.

*Input:* Grammar \( G \).

*Output:* Parsing table \( M \).

1. for each \( A \rightarrow \alpha \), do Steps 2 & 3.
2. for each terminal \( a \in \text{FIRST}(\alpha) \),
   add \( A \rightarrow \alpha \) to \( M[A, a] \).
3. 3.1 if \( \epsilon \in \text{FIRST}(\alpha) \),
   add \( A \rightarrow \alpha \) to \( M[A, b] \) for each terminal \( b \in \text{FOLLOW}(A) \).
   3.2 if \( \epsilon \in \text{FIRST}(\alpha) \) and \( \$ \in \text{FOLLOW}(A) \),
   add \( A \rightarrow \alpha \) to \( M[A, \$] \).

end.

### 3.4.1 Example:

\( G_4 \):

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' | \epsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' | \epsilon \\
F & \rightarrow (E) | \text{id}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Input symbol</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>( E \rightarrow TE' )</td>
<td>( E \rightarrow TE' )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E' )</td>
<td>( E' \rightarrow +TE' )</td>
<td>( E' \rightarrow \epsilon )</td>
<td>( E' \rightarrow \epsilon )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T )</td>
<td>( T \rightarrow FT' )</td>
<td>( T \rightarrow FT' )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T' )</td>
<td>( T' \rightarrow \epsilon )</td>
<td>( T' \rightarrow *FT' )</td>
<td>( T' \rightarrow \epsilon )</td>
<td>( T' \epsilon )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F )</td>
<td>( F \rightarrow \text{id} )</td>
<td>( F \rightarrow (E) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.4.2 Stack Operation

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$id + id * id$</td>
<td>$E \rightarrow TE'$</td>
</tr>
<tr>
<td>$E'T$</td>
<td>$id + id * id$</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>$E'TF$</td>
<td>$id + id * id$</td>
<td>$F \rightarrow id$</td>
</tr>
<tr>
<td>$E'id$</td>
<td>$id + id * id$</td>
<td>match</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>$id * id$</td>
<td>$T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$id * id$</td>
<td>$E' \rightarrow +TE'$</td>
</tr>
<tr>
<td>$E'T+$</td>
<td>$id * id$</td>
<td>match</td>
</tr>
<tr>
<td>$E'T$</td>
<td>$id * id$</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>$E'TF$</td>
<td>$id * id$</td>
<td>$F \rightarrow id$</td>
</tr>
<tr>
<td>$E'id$</td>
<td>$id * id$</td>
<td>match</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>$id * id$</td>
<td>$T' \rightarrow *FT'$</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>$id * id$</td>
<td>match</td>
</tr>
<tr>
<td>$E'$</td>
<td>$id$</td>
<td>$E' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E$</td>
<td>$$</td>
<td>accept</td>
</tr>
</tbody>
</table>

3.5 Properties of LL(1) Grammars

A grammar whose parsing table has no multiply-defined entries is said to be LL(1).

Properties:

1. No ambiguous or left-recursive grammar can be LL(1).

2. A grammar $G$ is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions, the following conditions hold:

   2.1 For any terminal $a$, there exist no derivations that $\alpha \Rightarrow \ast a\alpha'$ and $\beta \Rightarrow \ast a\beta'$.

   2.2 Either $\alpha$ or $\beta$, but not both, can derive $\epsilon$.

   2.3 If $\beta \Rightarrow \epsilon$, then $\alpha$ does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$.

Proof of Condition 2.2: Suppose $\alpha \Rightarrow \epsilon$ and $\beta \Rightarrow \epsilon$. Consider $S \Rightarrow \gamma_1A\gamma_2$. Then, two possibilities exist: $S \Rightarrow \gamma_1A\gamma_2 \Rightarrow \gamma_1\alpha\gamma_2 \Rightarrow \gamma_1\gamma_2$ and $S \Rightarrow \gamma_1A\gamma_2 \Rightarrow \gamma_1\beta\gamma_2 \Rightarrow \gamma_1\gamma_2$. $G$ must be then ambiguous.

Proof of Condition 2.3: Suppose $\beta \Rightarrow \epsilon$ and $\alpha \Rightarrow \ast aa'$, where $a \in \text{FOLLOW}(A)$. Also, assume that $\gamma_2 \Rightarrow a\gamma_2'$. We then have two possibilities: (i) $S \Rightarrow \gamma_1A\gamma_2 \Rightarrow \gamma_1\alpha\gamma_2 \Rightarrow \gamma_1aa'\gamma_2$, and (ii) $S \Rightarrow \gamma_1A\gamma_2 \Rightarrow \gamma_1\beta\gamma_2 \Rightarrow \gamma_1\gamma_2 \Rightarrow \gamma_1a\gamma_2$. Hence, after taking care of the input tokens corresponding to $\gamma_1$, the parser cannot make a clear choice between the two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$. 


4 Bottom-Up Parsing

4.1 SLR Parser

4.1.1 Computation of Closure

If $I$ is a set of items for a grammar $G$, then $\text{closure}(I)$ is the set of items constructed from $I$ by the two rules.

1. Initially, every item in $I$ is added to $\text{closure}(I)$.

2. If $A \rightarrow \alpha \cdot B\beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot \gamma$ to $I$, if it is not already in $I$. We apply this rule until no more new items can be added to $\text{closure}(I)$.

function $\text{closure}(I)$:
begin
$J = I$;
repeat
for each item $A \rightarrow \alpha \cdot B\beta$ in $J$ and each production $B \rightarrow \gamma$ of $G$ such that $B \rightarrow \cdot \gamma$ is not in $J$ do
add $B \rightarrow \cdot \gamma$ to $J$
until no more items can be added to $J$
return
end

We are now ready to give the algorithm to construct $C$, the canonical collection of sets of $LR(0)$ items for an augmenting grammar $G'$.

procedure $\text{items}(G')$:
begin
$C = \{\text{closure}([S' \rightarrow \cdot S])\}$;
repeat
for each set of items $I$ in $C$ and each grammar symbol $X$ such that $\text{goto}(I, X)$ is not empty and not in $C$ do
add $\text{goto}(I, X)$ to $C$
until no more sets of items can be added to $C$
end
4.1.2 Constructing SLR Parsing Table

**Algorithm:** Constructing an SLR parsing table.

**Input:** An augmenting grammar \( G' \).

**Output:** The SLR parsing table functions \textit{action} and \textit{goto} for \( G' \).

1. Construct \( C = \{ I_0, \cdots, I_n \} \), the collection of sets of LR(0) items for \( G' \).
2. State \( i \) constructed from \( I_i \). The parsing actions for state \( i \) are determined as follows:
   a) If \([ A \rightarrow \alpha \cdot a\beta ] \) is in \( I_i \) and \( \text{goto}(I_i, a) = I_j \), then set \text{action}[i, a] to “shift j.”
      Here \( a \) must be a terminal.
   b) If \([ A \rightarrow \alpha \cdot ] \) is in \( I_i \), then set \text{action}[i, a] to “reduce \( A \rightarrow \alpha \)” for all \( a \) in \textit{FOLLOW}(A);
      here \( A \) may not be \( S' \).
   c) If \([ S' \rightarrow \cdot S ] \) is in \( I_i \), then set \text{action}[i, $] to “accept.”

If any conflicting actions are generated by the above rules, we say the grammar is not SLR(0).

The algorithm fails to produce a parser in this case.

3. The \textit{goto} transitions for state \( i \) are constructed for all nonterminals \( A \) using the rule:
   If \( \text{goto}(I_i, A) = I_j \), then \( \text{goto}[i, A] = j \).
4. All entries not defined by rules (2) and (3) are made “error.”
5. The initial state of the parser is the one constructed from the set of items containing \([ S' \rightarrow \cdot S ] \).

end.

4.1.3 Example

Consider the following grammar \( G \):

\[
\begin{align*}
(0) & \quad E' \rightarrow E \\
(1) & \quad E \rightarrow E + T \\
(2) & \quad E \rightarrow T \\
(3) & \quad T \rightarrow T \ast F \\
(4) & \quad T \rightarrow F \\
(5) & \quad F \rightarrow (E) \\
(6) & \quad F \rightarrow id
\end{align*}
\]
The canonical LR(0) collection for $G$ is:

<table>
<thead>
<tr>
<th>$I_0$</th>
<th>$E' \rightarrow \cdot E$</th>
<th>$I_5$</th>
<th>$F \rightarrow \text{id}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E \rightarrow \cdot E + T$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow \cdot T$</td>
<td>$I_6$</td>
<td>$E \rightarrow \cdot E + T$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow \cdot T * F$</td>
<td></td>
<td>$T \rightarrow \cdot T * F$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow \cdot F$</td>
<td></td>
<td>$T \rightarrow \cdot F$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow \cdot (E)$</td>
<td></td>
<td>$F \rightarrow \cdot (E)$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow \cdot \text{id}$</td>
<td></td>
<td>$F \rightarrow \cdot \text{id}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$I_1$</th>
<th>$E' \rightarrow E\cdot$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E \rightarrow E \cdot + T$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$I_2$</th>
<th>$E \rightarrow T\cdot$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T \rightarrow T \cdot * F$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$I_3$</th>
<th>$T \rightarrow F\cdot$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T \rightarrow T \cdot * F$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$I_4$</th>
<th>$F \rightarrow \cdot (E)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E \rightarrow \cdot E + T$</td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow \cdot T$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow \cdot T * F$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow \cdot F$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow \cdot (E)$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow \cdot \text{id}$</td>
</tr>
</tbody>
</table>

| $I_5$ | $F \rightarrow \text{id}$ |

| $I_6$ | $E \rightarrow \cdot E + T$ |

<table>
<thead>
<tr>
<th>$I_7$</th>
<th>$T \rightarrow T \cdot * F$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$F \rightarrow \cdot (E)$</td>
</tr>
<tr>
<td></td>
<td>$F \rightarrow \cdot \text{id}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$I_8$</th>
<th>$F \rightarrow \cdot (E)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E \rightarrow E \cdot + T$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$I_9$</th>
<th>$E \rightarrow E + T\cdot$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T \rightarrow T \cdot * F$</td>
</tr>
</tbody>
</table>

| $I_{10}$ | $T \rightarrow T \cdot * F$ |

| $I_{11}$ | $F \rightarrow \cdot (E)$ |

The transition for viable prefixes is:

$I_0$: $goto(I_0, E) = I_1$; $goto(I_0, T) = I_2$; $goto(I_0, F) = I_3$; $goto(I_0, () = I_4$; $goto(I_0, \text{id}) = I_5$;

$I_1$: $goto(I_1, +) = I_6$;

$I_2$: $goto(I_2, *) = I_7$;

$I_3$: $goto(I_3, E) = I_8$; $goto(I_3, T) = I_2$; $goto(I_3, F) = I_3$; $goto(I_3, () = I_4$; $goto(I_3, \text{id}) = I_5$;

$I_4$: $goto(I_4, E) = I_8$; $goto(I_4, T) = I_2$; $goto(I_4, F) = I_3$; $goto(I_4, () = I_4$; $goto(I_4, \text{id}) = I_5$;

$I_5$: $goto(I_5, () = I_4$; $goto(I_5, \text{id}) = I_5$;

$I_6$: $goto(I_6, T) = I_9$; $goto(I_6, F) = I_3$; $goto(I_6, () = I_4$; $goto(I_6, \text{id}) = I_5$;

$I_7$: $goto(I_7, F) = I_{10}$; $goto(I_7, () = I_4$; $goto(I_7, \text{id}) = I_5$;

$I_8$: $goto(I_8, () = I_{11}$; $goto(I_8, +) = I_6$;

$I_9$: $goto(I_9, *) = I_7$;
The FOLLOW set is: \( \text{FOLLOW}(E') = \{\$\}; \text{FOLLOW}(E) = \{+, \}, \$\}; \text{FOLLOW}(T) = \text{FOLLOW}(F) = \{+, \}, \$, \ast\}.

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

The moves of the SLR parser on input \( id \ast id + id \) is:

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>0</td>
<td>id * id + id $</td>
<td>shift</td>
</tr>
<tr>
<td>(2)</td>
<td>0id5</td>
<td>*id+id$</td>
<td>reduce by ( F \rightarrow id )</td>
</tr>
<tr>
<td>(3)</td>
<td>0F3</td>
<td>*id+id$</td>
<td>reduce by ( T \rightarrow F )</td>
</tr>
<tr>
<td>(4)</td>
<td>0T2</td>
<td>*id+id$</td>
<td>shift</td>
</tr>
<tr>
<td>(5)</td>
<td>0T2*7</td>
<td>id+id$</td>
<td>shift</td>
</tr>
<tr>
<td>(6)</td>
<td>0T2*7id5</td>
<td>+id$</td>
<td>reduce by ( F \rightarrow id )</td>
</tr>
<tr>
<td>(7)</td>
<td>0T2*7F10</td>
<td>+id$</td>
<td>reduce by ( T \rightarrow T * F )</td>
</tr>
<tr>
<td>(8)</td>
<td>0T2</td>
<td>+id$</td>
<td>reduce by ( E \rightarrow T )</td>
</tr>
<tr>
<td>(9)</td>
<td>0E1</td>
<td>+id$</td>
<td>shift</td>
</tr>
<tr>
<td>(10)</td>
<td>0E1+6</td>
<td>id$</td>
<td>shift</td>
</tr>
<tr>
<td>(11)</td>
<td>0E1+6id5</td>
<td>$</td>
<td>reduce by ( F \rightarrow id )</td>
</tr>
<tr>
<td>(12)</td>
<td>0E1+6F3</td>
<td>$</td>
<td>reduce by ( T \rightarrow F )</td>
</tr>
<tr>
<td>(13)</td>
<td>0E1+6T9</td>
<td>$</td>
<td>reduce by ( E \rightarrow E + T )</td>
</tr>
<tr>
<td>(14)</td>
<td>0E1</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
4.2 Canonical LR(1) Parser

Consider the following grammar $G$ with productions:

$$
S' \rightarrow S \\
S \rightarrow L = R \\
S \rightarrow R \\
L \rightarrow \ast R \\
L \rightarrow id \\
R \rightarrow L
$$

Let’s construct the canonical sets of LR(0) items for $G$:

$$
I_0: S' \rightarrow \cdot S \\
S \rightarrow \cdot L = R \\
S \rightarrow \cdot R \\
L \rightarrow \cdot \ast R \\
L \rightarrow \cdot id \\
R \rightarrow \cdot L
$$

$$
I_5: L \rightarrow id \cdot \\
I_6: S \rightarrow L = \cdot R \\
R \rightarrow \cdot L \\
L \rightarrow \cdot \ast R \\
L \rightarrow \cdot id
$$

$$
I_1: S' \rightarrow S \cdot \\
I_7: L \rightarrow \ast R \cdot \\
I_2: S \rightarrow L \cdot = R \\
R \rightarrow L \cdot \\
I_3: S \rightarrow R \cdot \\
I_8: R \rightarrow L \cdot \\
I_4: L \rightarrow \ast \cdot R \\
R \rightarrow \cdot L \\
L \rightarrow \cdot \ast L \\
L \rightarrow \cdot id
$$

Note that $\ast \in \text{FOLLOW}(R)$ since $S \Rightarrow L = R \Rightarrow \ast R = R$. Consider the state $I_2$ and the input symbol is “=.” From $[R \rightarrow L \cdot]$, the parser will reduce by $R \rightarrow L$ since $\ast \in \text{FOLLOW}(R)$. But due to $[S \rightarrow L \cdot = R]$, it will try to shift the input as well, a conflict. Therefore, this grammar $G$ cannot be handled by the SLR(0) parser. In fact, $G$ can be parsed using the canonical-LR(1) parser that will be discussed next.
4.2.1 Construction of LR(1) Items

Let $G'$ be an augmented grammar of $G$.

**function** closure($I$):
**begin**
  repeat
    for each item $[A \to \alpha \cdot B \beta, a]$ in $I$,
    each production $B \to \gamma$ in $G'$,
    and each terminal $b$ in FIRST($\beta a$)
    such that $[B \to \cdot \gamma, b]$ is not in $I$ do
      add $[B \to \cdot \gamma, b]$ to $I$;
  until no more items can be added to $I$
  **return** $I$
**end**

**function** goto($I, X$):
**begin**
  let $J$ be the set of items $[A \to \alpha X \beta, a]$ such that
  $[A \to \alpha \cdot X \beta, a]$ is in $I$;
  **return** closure($J$)
**end**

**procedure** items($G'$):
**begin**
  $C = \{\text{closure}([S' \to \cdot S, \$])\}$;
  repeat
    for each set of items $I$ in $C$ and each grammar symbol $X$
    such that goto($I, X$) is not empty and not in $C$ do
      add goto($I, X$) to $C$
    until no more sets of items can be added to $C$
**end**
4.2.2 Construction of canonical-LR(1) parser

Algorithm: Constructing a canonical LR(1) parsing table.

Input: An augmenting grammar $G'$.

Output: The canonical LR(1) parsing table functions action and goto for $G'$.

1. Construct $C = \{I_0, \cdots, I_n\}$, the collection of sets of LR(1) items for $G'$.
2. State $i$ constructed from $I_i$. The parsing actions for state $i$ are determined as follows:
   a) If $[A \rightarrow \alpha \cdot a \beta, b]$ is in $I_i$ and $\text{goto}(I_i, a) = I_j$, then set $\text{action}[i, a]$ to “shift $j$.”
      Here $a$ must be a terminal.
   b) If $[A \rightarrow \alpha \cdot, a]$ is in $I_i$, then set $\text{action}[i, a]$ to “reduce $A \rightarrow \alpha$”;
      here $A$ may not be $S'$.
   c) If $[S' \rightarrow S, \]$ is in $I_i$, then set $\text{action}[i, \]$ to “accept.”

   If any conflicting actions are generated by the above rules, we say the grammar is not to be LR(1).
   The algorithm fails to produce a parser in this case.
3. The goto transitions for state $i$ are constructed for all nonterminals $A$ using the rule:
   If $\text{goto}(I_i, A) = I_j$, then $\text{goto}[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made “error.”
5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot S, \]$.

end.

4.2.3 Construction of LALR Parsing Table

Algorithm: Constructing an LALR parsing table.

Input: A grammar $G$.

Output: The LALR parsing table for $G$.

1. Construct $C = \{I_0, \cdots, I_n\}$, the collection of sets of LR(1) items for $G$.
2. Final all sets having the same core, and replace these sets by their union.
3. Let $C' = \{J_1, J_2, \cdots, J_m\}$ be the resulting sets of LR(1) items.
   Action table is constructed in the same manner as in Algorithm for Canonical LR(1) parsing table.
4. goto table is constructed as follows.
   Note that if $J_q = I_1 \cup I_2 \cup \cdots \cup I_k$, and for a non-terminal $X$,
   $\text{goto}(I_1, X) = J_{p_1}$, $\text{goto}(I_2, X) = J_{p_2}$, $\cdots$, $\text{goto}(I_k, X) = J_{p_k}$,
   then make $\text{goto}(J_q, X) = s$ where $s = J_{p_1} \cup J_{p_2} \cup \cdots \cup J_{p_k}$.
   (Note that $J_{p_1}, \cdots, J_{p_k}$ all have the same core.) end.
4.2.4 Example

Consider the following grammar $G'$:

(0) $S' \rightarrow S$
(1) $S \rightarrow CC$
(2) $C \rightarrow cC$
(3) $C \rightarrow d$

The canonical LR(1) collection for $G'$ is:

$I_0 :$ $S' \rightarrow \cdot S, \$  
     $S \rightarrow \cdot CC, \$  
     $C \rightarrow \cdot cC, c/d$  
     $C \rightarrow \cdot d, c/d$

$I_1 :$ $S' \rightarrow \cdot S, \$

$I_2 :$ $S \rightarrow C \cdot C, \$  
     $C \rightarrow \cdot cC, \$  
     $C \rightarrow \cdot d, \$

$I_3 :$ $C \rightarrow \cdot C, c/d$  
     $C \rightarrow \cdot cC, c/d$  
     $C \rightarrow \cdot d, c/d$

$I_4 :$ $C \rightarrow d \cdot , c/d$

$I_5 :$ $S \rightarrow CC \cdot , \$

$I_6 :$ $C \rightarrow \cdot C, \$  
     $C \rightarrow \cdot cC, \$  
     $C \rightarrow \cdot d, \$

$I_7 :$ $C \rightarrow d \cdot , \$

$I_8 :$ $C \rightarrow cC \cdot, c/d$

$I_9 :$ $C \rightarrow cC \cdot, \$
The transition for viable prefixes is:

\( I_0: \) goto\((I_0, S) = I_1; \) goto\((I_0, C) = I_2; \) goto\((I_0, c) = I_3; \) goto\((I_0, d) = I_4; \)

\( I_2: \) goto\((I_2, C) = I_5; \) goto\((I_2, c) = I_6; \) goto\((I_2, d) = I_7; \)

\( I_3: \) goto\((I_3, c) = I_3; \) goto\((I_3, d) = I_4; \) goto\((I_3, C) = I_8; \)

\( I_6: \) goto\((I_6, C) = I_9; \)

### A. Canonical-LR(1) parsing table

<table>
<thead>
<tr>
<th>State</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>s3 s4</td>
<td>1 2</td>
</tr>
<tr>
<td>( 1 )</td>
<td>acc</td>
<td></td>
</tr>
<tr>
<td>( 2 )</td>
<td>s6 s7</td>
<td>5</td>
</tr>
<tr>
<td>( 3 )</td>
<td>s3 s4</td>
<td>8</td>
</tr>
<tr>
<td>( 4 )</td>
<td>r3 r3</td>
<td></td>
</tr>
<tr>
<td>( 5 )</td>
<td></td>
<td>r1</td>
</tr>
<tr>
<td>( 6 )</td>
<td>s6 s7</td>
<td>9</td>
</tr>
<tr>
<td>( 7 )</td>
<td></td>
<td>r3</td>
</tr>
<tr>
<td>( 8 )</td>
<td>r2 r2</td>
<td></td>
</tr>
<tr>
<td>( 9 )</td>
<td></td>
<td>r2</td>
</tr>
</tbody>
</table>

### B. LALR(1) parsing table

<table>
<thead>
<tr>
<th>State</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>s36 s47</td>
<td>1 2</td>
</tr>
<tr>
<td>( 1 )</td>
<td>acc</td>
<td></td>
</tr>
<tr>
<td>( 2 )</td>
<td>s36 s47</td>
<td>5</td>
</tr>
<tr>
<td>( 36 )</td>
<td>s36 s47</td>
<td>89</td>
</tr>
<tr>
<td>( 47 )</td>
<td>r3 r3 r3</td>
<td></td>
</tr>
<tr>
<td>( 5 )</td>
<td></td>
<td>r1</td>
</tr>
<tr>
<td>( 89 )</td>
<td>r2 r2 r2</td>
<td></td>
</tr>
</tbody>
</table>

### 4.2.5 Note on LALR Parsing Table

Suppose we have an LR(1) grammar, that is, one whose sets of LR(1) items produce no parsing action conflicts. If we replace all states having the same core with their union, it is possible that the resulting union will have a conflict, but it is unlikely for the following reasons.

Suppose in the union there is a conflict on lookahead \( a \) because there is an item \([B \rightarrow \beta \cdot a\gamma, b]\) calling for a reduction by \( A \rightarrow \alpha \), and there is another item \([B \rightarrow \beta \cdot a\gamma, b]\) calling for a shift. Then,
some set of items from which the union was formed has item \([A \rightarrow \alpha \cdot a]\), and since the cores of all these states are the same, it must have an item \([B \rightarrow \beta \cdot a\gamma \cdot c]\) for some \(c\). But then this state has the same shift/reduce conflict on \(a\), and the grammar was not LR(1) as we assumed. Thus, the merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states, because shift actions depend only on core, not the lookahead.

It is possible, however, that a merger will produce a reduce/reduce conflict as the following example shows.

**Example:**

\[
\begin{align*}
S' & \rightarrow S \\
S & \rightarrow aAd \mid bBd \mid aBe \mid bAe \\
A & \rightarrow c \\
B & \rightarrow c
\end{align*}
\]

which generates the four strings \(acd, ace, bcd, bce\). This grammar can be checked to be LR(1) by constructing the sets of items. Upon doing so, we find the set of items \{\([A \rightarrow \cdot c \cdot d]\), \([B \rightarrow \cdot c \cdot e]\)\} valid for viable prefix \(ac\) and \{\([A \rightarrow \cdot c \cdot e]\), \([B \rightarrow \cdot c \cdot d]\)\} valid for \(bc\). Neither of these sets generates a conflict, and their cores are the same. However, their union, which is

\[
\begin{align*}
A & \rightarrow \cdot c \cdot d/e \\
B & \rightarrow \cdot c \cdot d/e
\end{align*}
\]

generates a reduce/reduce conflict, since reduction by both \(A \rightarrow c\) and \(B \rightarrow c\) are called for on input \(d\) and \(e\).