Configuration Sets for CSX-Lite

State | Configuration Set
--- | ---
$s_0$ | Prog → *(Stmts)* Eof

### State $s_1$
- **Prog → *(Stmts)* Eof**
- **Stmts → ·StmtStmts**
- **Stmt → λ·**
- **Stmt → *id = Expr·**
- **Stmt → if (Expr)Stmt**

### State $s_2$
- **Prog → *(Stmts) Eof**

### State $s_3$
- **Stmts → StmtStmts**
- **Stmts → ·StmtStmts**
- **Stmts → λ·**
- **Stmt → *id = Expr·**
- **Stmt → if (Expr)Stmt**

### State $s_4$
- **Stmt → *id = Expr·**

### State $s_5$
- **Stmt → if * (Expr) Stmt**

### Parser Action Table

<table>
<thead>
<tr>
<th>State</th>
<th>Configuration Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_{14}$</td>
<td>Stmt → *id = Expr·</td>
</tr>
<tr>
<td>$s_{15}$</td>
<td>Expr → Expr + ·id</td>
</tr>
<tr>
<td>$s_{16}$</td>
<td>Expr → Expr - ·id</td>
</tr>
<tr>
<td>$s_{17}$</td>
<td>Stmt → if (Expr)·Stmt</td>
</tr>
<tr>
<td>$s_{18}$</td>
<td>Expr → Expr + ·id</td>
</tr>
<tr>
<td>$s_{19}$</td>
<td>Expr → Expr - ·id</td>
</tr>
<tr>
<td>$s_{20}$</td>
<td>Stmt → if (Expr) Stmt·</td>
</tr>
</tbody>
</table>

### Parser Action Table

We will table possible parser actions based on the current state (configuration set) and token. Given configuration set $C$ and input token $T$ four actions are possible:

- **Reduce i**: The $i$-th production has been matched.
- **Shift**: Match the current token.
- **Accept**: Parse is correct and complete.
- **Error**: A syntax error has been discovered.
We will let $A[C][T]$ represent the possible parser actions given configuration set $C$ and input token $T$.

$$A[C][T] =
\begin{cases}
\text{Reduce } i & \text{i-th production is } A \rightarrow \alpha \\
\text{and } A \rightarrow \alpha \ast \text{ is in } C \\
\text{and } T \text{ in Follow}(A) \\
\end{cases}
\bigcup
\begin{cases}
\text{Shift} & \text{if } (B \rightarrow \beta \ast T \gamma \text{ is in } C) \\
\text{else } \phi \\
\end{cases}$$

This rule simply collects all the actions that a parser might do given $C$ and $T$.

But we want parser actions to be unique so we require that the parser action always be unique for any $C$ and $T$.

If the parser action isn’t unique, then we have a shift/reduce error or reduce/reduce error. The grammar is then rejected as unparsable.

If parser actions are always unique then we will consider a shift of EOF to be an accept action.

An empty (or undefined) action for $C$ and $T$ will signify that token $T$ is illegal given configuration set $C$.

A syntax error will be signaled.

### LALR Parser Driver

Given the GoTo and parser action tables, a Shift/Reduce (LALR) parser is fairly simple:

```c
void LALRDriver(){
Push(S0);
while(true){
//Let S = Top state on parse stack
//Let CT = current token to match
switch (A[S][CT]) {
\text{case error:}
\text{SyntaxError(CT);return;}
\text{case accept:}
\text{return;}
\text{case shift:}
\text{push(GoTo[S][CT]);}
\text{CT= Scanner();}
\text{break;}
\text{case reduce i:}
\text{//Let prod i = A\rightarrow Y_1...Y_m}
\text{pop m states;}
\text{//Let S’ = new top state}
\text{push(GoTo[S’][A]);}
\text{break;}
} } }
```

### Action Table for CSX-Lite

```
0  1  2  3  4  5  6  7  8  9  1 0  1 1  1 2  1 3  1 4  1 5  1 6  1 7  1 8  1 9  2 0
[   S
} R3 S R3 R2 R4 R5 R4 S R6 R5
if S S S R8 S R6 R7
{ ( S
} R8 S R6 R7
id S S S S S R4 S S S
= S
+ S R8 S R6 R7
- S R8 S R6 R7
; S R8 S R6 R7 R5
eof A
```
### Example of LALR(1) Parsing

We’ll again parse

\{ a = b + c; \} Eof

We start by pushing state 0 on the parse stack.

<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Top State</th>
<th>Action</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Prog \rightarrow { Stmts } Eof</td>
<td>Shift</td>
<td>(a = b + c;) Eof</td>
</tr>
<tr>
<td>1 0</td>
<td>Prog \rightarrow { \ast Stmts } Eof</td>
<td>Shift</td>
<td>a = b + c; Eof</td>
</tr>
<tr>
<td></td>
<td>Stmts \rightarrow \ast Stmts Stmts \ast Stmts \ast id = Expr ; Stmts \ast if (Expr)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1 0</td>
<td>Stmts \rightarrow \ast id = Expr ;</td>
<td>Shift</td>
<td>b + c; Eof</td>
</tr>
<tr>
<td></td>
<td>Expr \rightarrow \ast Expr ; Expr \rightarrow \ast Expr + id Expr \rightarrow \ast Expr - id Expr \rightarrow \ast id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 8 4 1 0</td>
<td>Expr \rightarrow id \ast</td>
<td>Reduce 8</td>
<td>+ c;) Eof</td>
</tr>
<tr>
<td>11 8 4 1 0</td>
<td>Stmt \rightarrow id = Expr \ast ; Expr \rightarrow Expr \ast + id Expr \rightarrow Expr \ast - id</td>
<td>Shift</td>
<td>+ c;) Eof</td>
</tr>
<tr>
<td>15 11 8 4 1 0</td>
<td>Expr \rightarrow Expr \ast + id</td>
<td>Shift</td>
<td>c;) Eof</td>
</tr>
<tr>
<td>18 15 11 8 4 1 0</td>
<td>Expr \rightarrow Expr + id \ast</td>
<td>Reduce 6</td>
<td>; Eof</td>
</tr>
<tr>
<td>11 8 4 1 0</td>
<td>Stmt \rightarrow id = Expr \ast ; Expr \rightarrow Expr \ast + id Expr \rightarrow Expr \ast - id</td>
<td>Shift</td>
<td>; Eof</td>
</tr>
<tr>
<td>14 11 8 4 1 0</td>
<td>Stmt \rightarrow id = Expr \ast ;</td>
<td>Reduce 4</td>
<td>Eof</td>
</tr>
</tbody>
</table>
Error Detection in LALR Parsers

In bottom-up, LALR parsers syntax errors are discovered when a blank (error) entry is fetched from the parser action table.

Let's again trace how the following illegal CSX-lite program is parsed:

```plaintext
{ b + c = a; } Eof
```

LALR is More Powerful

Essentially all LL(1) grammars are LALR(1) plus many more. Grammar constructs that confuse LL(1) are readily handled.

- Common prefixes are no problem. Since sets of configurations are tracked, more than one prefix can be followed. For example, in

```plaintext
Stmt \rightarrow id = Expr ;
Stmt \rightarrow id ( Args ) ;
```

after we match an id we have

```plaintext
Stmt \rightarrow id \cdot = Expr ;
Stmt \rightarrow id \cdot ( Args ) ;
```

The next token will tell us which production to use.
Left recursion is also not a problem. Since sets of configurations are tracked, we can follow a left-recursive production and all others it might use. For example, in

\[
\begin{align*}
\text{Expr} &\rightarrow \cdot \text{Expr} + \text{id} \\
\text{Expr} &\rightarrow \cdot \text{id}
\end{align*}
\]

we can first match an \text{id}:

\[
\text{Expr} \rightarrow \text{id} \cdot
\]

Then the \text{Expr} is recognized:

\[
\text{Expr} \rightarrow \text{Expr} \cdot + \text{id}
\]

The left-recursion is handled!

But ambiguity will still block construction of an LALR parser. Some shift/reduce or reduce/reduce conflict must appear. (Since two or more distinct parses are possible for some input). Consider our original productions for if-then and if-then-else statements:

\[
\begin{align*}
\text{Stmt} &\rightarrow \text{if} \left( \text{Expr} \right) \text{Stmt} \cdot \\
\text{Stmt} &\rightarrow \text{if} \left( \text{Expr} \right) \text{Stmt} \cdot \text{else} \text{Stmt}
\end{align*}
\]

Since \text{else} can follow \text{Stmt}, we have an unresolvable shift/reduce conflict.

Grammar Engineering

Though LALR grammars are very general and inclusive, sometimes a reasonable set of productions is rejected due to shift/reduce or reduce/reduce conflicts.

In such cases, the grammar may need to be “engineered” to allow the parser to operate.

A good example of this is the definition of MemberDecls in CSX.

A straightforward definition is

\[
\begin{align*}
\text{MemberDecls} &\rightarrow \cdot \text{FieldDecls} \text{MethodDecls} \\
\text{FieldDecls} &\rightarrow \cdot \text{FieldDecl} \text{FieldDecls} \\
\text{FieldDecls} &\rightarrow \lambda \\
\text{MethodDecls} &\rightarrow \cdot \text{MethodDecl} \text{MethodDecls} \\
\text{MethodDecls} &\rightarrow \lambda \\
\text{FieldDecl} &\rightarrow \text{int} \text{id} ; \\
\text{MethodDecl} &\rightarrow \text{int} \text{id} ( ) ; \text{Body}
\end{align*}
\]

When we predict MemberDecls we get:

\[
\begin{align*}
\text{MemberDecls} &\rightarrow \cdot \text{FieldDecls} \text{MethodDecls} \\
\text{FieldDecls} &\rightarrow \cdot \text{FieldDecl} \text{FieldDecls} \\
\text{FieldDecls} &\rightarrow \lambda \\
\text{MethodDecls} &\rightarrow \cdot \text{MethodDecl} \text{MethodDecls} \\
\text{MethodDecls} &\rightarrow \lambda \\
\text{FieldDecl} &\rightarrow \text{int} \text{id} ; \\
\text{MethodDecl} &\rightarrow \text{int} \text{id} ( ) ; \text{Body}
\end{align*}
\]

Now \text{int} follows FieldDecls since MethodDecls \Rightarrow^* \text{int} ...

Thus an unresolvable shift/reduce conflict exists.

The problem is that \text{int} is derivable from both FieldDecls and MethodDecls, so when we see an \text{int}, we can’t tell which way to parse it (and FieldDecls \rightarrow \lambda requires we make an immediate decision!).
If we rewrite the grammar so that we can delay deciding from where the int was generated, a valid LALR parser can be built:

- `MemberDecls → FieldDecl MemberDecls`
- `MemberDecls → MethodDecls`
- `MethodDecls → MethodDecl MethodDecls`
- `MethodDecls → λ`
- `FieldDecl → int id ;`
- `MethodDecl → int id ( ) ; Body`

When `MemberDecls` is predicted we have:

- `MemberDecls → • FieldDecl MemberDecls`
- `MemberDecls → • MethodDecls`
- `MethodDecls → • MethodDecl MethodDecls`
- `MethodDecls → λ •`
- `FieldDecl → • int id ;`
- `MethodDecl → • int id ( ) ; Body`

Now `Follow(MethodDecls) = Follow(MemberDecls) = “}”,` so we have no shift/reduce conflict. After `int id` is matched, the next token (a “;” or a “(“) will tell us whether a `FieldDecl` or a `MethodDecl` is being matched.