Configuration Sets for CSX-Lite

State	Cofiguration Set
s_0	**Prog → •{ Stmts } Eof**
s_1	**Prog → { • Stmts } Eof** **Stmts → •Stmt Stmts** **Stmts → λ •** **Stmt → • id = Expr ;** **Stmt → • if (Expr) Stmt**
s_2	**Prog → { Stmts •} Eof**
s_3	**Stmts → Stmt • Stmts** **Stmts → •Stmt Stmts** **Stmts → λ •** **Stmt → • id = Expr ;** **Stmt → • if (Expr) Stmt**
s_4	**Stmt → id • = Expr ;**
s_5	**Stmt → if • (Expr) Stmt**

State	Cofiguration Set
s_6	**Prog → { Stmts } •Eof**
s_7	**Stmts → Stmt Stmts •**
s_8	**Stmt → id = • Expr ;** **Expr → • Expr + id** **Expr → • Expr - id** **Expr → • id**
s_9	**Stmt → if (• Expr) Stmt** **Expr → • Expr + id** **Expr → • Expr - id** **Expr → • id**
s_{10}	**Prog → { Stmts } Eof •**
s_{11}	**Stmt → id = Expr • ;** **Expr → Expr • + id** **Expr → Expr • - id**
s_{12}	**Expr → id •**
s_{13}	**Stmt → if (Expr •) Stmt** **Expr → Expr • + id** **Expr → Expr • - id**

State	Cofiguration Set
s_{14}	**Stmt → id = Expr ; •**
s_{15}	**Expr → Expr + • id**
s_{16}	**Expr → Expr - • id**
s_{17}	**Stmt → if (Expr) • Stmt** **Stmt → • id = Expr ;** **Stmt → • if (Expr) Stmt**
s_{18}	**Expr → Expr + id •**
s_{19}	**Expr → Expr - id •**
s_{20}	**Stmt → if (Expr) Stmt •**

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] =
 {Reduce i | i-th production is $\mathbf{A} \rightarrow \alpha$
 and $\mathbf{A} \rightarrow \alpha \bullet$ is in C
 and T in Follow(A) }
 U (If ($\mathbf{B} \rightarrow \beta \bullet \mathbf{T} \gamma$ is in C)
 {Shift} else ϕ)

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:

```
void LALRDriver(){
  Push(S0);
 while(true){
  //Let S = Top state on parse stack
  //Let CT = current token to match
    switch (A[S][CT]) {
     case error:
        SyntaxError(CT);return;
     case accept:
        return;
     case shift:
        push(GoTo[S][CT]);
        CT= Scanner();
        break;
     case reduce i:
        //Let prod i = A→Y1...Ym
        pop m states;
        //Let S' = new top state
        push(GoTo[S'][A]);
        break;
} } }
```

Action Table for CSX-Lite

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
{	S																				
}		R3	S	R3				R2							R4						R5
if		S		S											R4			S			R5
(						S															
)													R8	S					R6	R7	
id		S		S					S	S					R4	S	S	S			
=					S																
+												S	R8	S					R6	R7	
-												S	R8	S					R6	R7	
;												S	R8						R6	R7	R5
eof							A														

GoTo Table for CSX-Lite

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
{	1																				
}			6																		
if		5		5														5			
(						9															
)														17							
id		4		4					12	12						18	19	4			
=					8																
+												15		15							
-												16		16							
;												14									
eof							10														
stmts		2		7																	
stmt		3		3																	
expr									11	13											

Example of LALR(1) Parsing

We'll again parse

`{ a = b + c; } Eof`

We start by pushing state 0 on the parse stack.

Parse Stack	Top State	Action	Remaining Input
0	Prog → •{ Stmts } Eof	Shift	`{ a = b + c; } Eof`
1 0	Prog → { • Stmts } Eof Stmts → • Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr)	Shift	`a = b + c; } Eof`
4 1 0	Stmt → id • = Expr ;		`= b + c; } Eof`
8 4 1 0	Stmt → id = • Expr ; Expr → • Expr + id Expr → • Expr - id Expr → • id	Shift	`b + c; } Eof`

Parse Stack	Top State	Action	Remaining Input
12 8 4 1 0	Expr $\rightarrow$ id •	Reduce 8	`+ c; } Eof`
11 8 4 1 0	Stmt $\rightarrow$ id = Expr • ; Expr $\rightarrow$ Expr • + id Expr $\rightarrow$ Expr • - id	Shift	`+ c; } Eof`
15 11 8 4 1 0	Expr $\rightarrow$ Expr + • id	Shift	`c; } Eof`

Parse Stack	Top State	Action	Remaining Input
18 15 11 8 4 1 0	Expr $\rightarrow$ Expr + id $\bullet$	Reduce 6	`; } Eof`
11 8 4 1 0	Stmt $\rightarrow$ id = Expr $\bullet$; Expr $\rightarrow$ Expr $\bullet$ + id Expr $\rightarrow$ Expr $\bullet$ - id	Shift	`; } Eof`
14 11 8 4 1 0	Stmt $\rightarrow$ id = Expr ; $\bullet$	Reduce 4	`} Eof`

Parse Stack	Top State	Action	Remaining Input
3 1 0	Stmts $\rightarrow$ Stmt $\bullet$ Stmts Stmts $\rightarrow$ $\bullet$Stmt Stmts Stmts $\rightarrow$ λ $\bullet$ Stmt $\rightarrow$ $\bullet$ id = Expr ; Stmt $\rightarrow$ $\bullet$ if (Expr) Stmt	Reduce 3	`} Eof`
7 3 1 0	Stmts $\rightarrow$ Stmt Stmts $\bullet$	Reduce 2	`} Eof`
2 1 0	Prog $\rightarrow$ { Stmts $\bullet$} Eof	Shift	`} Eof`
6 2 1 0	Prog $\rightarrow$ { Stmts } $\bullet$Eof	Accept	`Eof`

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:

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

Parse Stack	Top State	Action	Remaining Input
0	Prog $\to$ •{ Stmts } Eof	Shift	`{ b + c = a; } Eof`
1 0	Prog $\to$ { • Stmts } Eof Stmts $\to$ • Stmt Stmts Stmts $\to$ λ • Stmt $\to$ • id = Expr ; Stmt $\to$ • if (Expr)	Shift	`b + c = a; } Eof`
4 1 0	Stmt $\to$ id • = Expr ;	Error (blank)	`+ 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

  Stmt $\rightarrow$ **id = Expr ;**
  Stmt $\rightarrow$ **id (Args) ;**

  after we match an id we have

  Stmt $\rightarrow$ **id · = Expr ;**
  Stmt $\rightarrow$ **id · (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

$$\textbf{Expr} \to \cdot\ \textbf{Expr}\ \ \textbf{+ id}$$
$$\textbf{Expr} \to\ \cdot\ \textbf{id}$$

we can first match an **id**:

$$\textbf{Expr} \to\ \textbf{id}\ \cdot$$

Then the **Expr** is recognized:

$$\textbf{Expr} \to\ \textbf{Expr}\ \cdot\ \textbf{+ id}$$

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:

  Stmt $\rightarrow$ if (Expr) Stmt $\bullet$

  Stmt $\rightarrow$ if (Expr) Stmt $\bullet$ else Stmt

  Since **else** can follow **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

MemberDecls $\rightarrow$ **FieldDecls MethodDecls**
FieldDecls $\rightarrow$ **FieldDecl FieldDecls**
FieldDecls $\rightarrow$ λ
MethodDecls $\rightarrow$ **MethodDecl MethodDecls**
MethodDecls $\rightarrow$ λ
FieldDecl $\rightarrow$ **int id ;**
MethodDecl $\rightarrow$ **int id ( ) ; Body**

When we predict **MemberDecls** we get:

MemberDecls $\to \bullet$ **FieldDecls MethodDecls**
FieldDecls $\to \bullet$ **FieldDecl FieldDecls**
FieldDecls $\to \lambda\bullet$
FieldDecl $\to \bullet$ **int id ;**

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

Thus an unresolvable shift/reduce conflict exists.

The problem is that **int** is derivable from both **FieldDecls** and **MethodDecls**, so when we see an **int**, we can't tell which way to parse it (and **FieldDecls** $\to \lambda$ 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.