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 → id •	Reduce 8	`+ c; } Eof`
11 8 4 1 0	Stmt → id = Expr • ; Expr → Expr • + id Expr → Expr • - id	Shift	`+ c; } Eof`
15 11 8 4 1 0	Expr → Expr + • id	Shift	`c; } Eof`

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

Parse Stack	Top State	Action	Remaining Input
3 1 0	Stmts → Stmt • Stmts Stmts → •Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr) Stmt	Reduce 3	`} Eof`
7 3 1 0	Stmts → Stmt Stmts •	Reduce 2	`} Eof`
2 1 0	Prog → { Stmts •} Eof	Shift	`} Eof`
6 2 1 0	Prog → { Stmts } •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 → •{ Stmts } Eof	Shift	`{ b + c = a; } Eof`
1 0	Prog → { • Stmts } Eof Stmts → • Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr)	Shift	`b + c = a; } Eof`
4 1 0	Stmt → 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 → id = Expr ;
  Stmt → id (Args) ;

  after we match an id we have

  Stmt → id · = Expr ;
  Stmt → 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

  Expr → · Expr + id
  Expr → · id

  we can first match an **id**:

  Expr → id ·

  Then the **Expr** is recognized:

  Expr → Expr · + 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 → if (Expr) Stmt •
  Stmt → if (Expr) Stmt • 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 → FieldDecls MethodDecls
FieldDecls → FieldDecl FieldDecls
FieldDecls → λ
MethodDecls → MethodDecl MethodDecls
MethodDecls → λ
FieldDecl → int id ;
MethodDecl → int id ( ) ; Body

When we predict **MemberDecls** we get:

MemberDecls → • FieldDecls MethodDecls
FieldDecls → • FieldDecl FieldDecls
FieldDecls → λ•
FieldDecl → • 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 → λ** 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.

Properties of LL and LALR Parsers

- Each prediction or reduce action is *guaranteed* correct. Hence the entire parse (built from LL predictions or LALR reductions) must be correct.

  This follows from the fact that LL parsers allow only one valid prediction per step. Similarly, an LALR parser never skips a reduction if it is consistent with the current token (and *all* possible reductions are tracked).

- LL and LALR parsers detect an syntax error as soon as the first invalid token is seen.

  Neither parser can match an invalid program prefix. If a token is matched it *must be* part of a valid program prefix. In fact, the prediction made or the stacked configuration sets *show* a possible derivation of the token accepted so far.

- All LL and LALR grammars are unambiguous.

  LL predictions are always unique and LALR shift/reduce or reduce/reduce conflicts are disallowed. Hence only one valid derivation of any token sequence is possible.

- All LL and LALR parsers require only linear time and space (in terms of the number of tokens parsed).

  The parsers do only fixed work per node of the concrete parse tree, and the size of this tree is linear in terms of the number of leaves in it (even with λ-productions included!).