Eliminating Common Prefixes

Assume we have two of more productions with the same lefthand side and a common prefix on their righthand sides:

$\mathbf{A} \rightarrow \alpha\,\beta \mid \alpha\,\gamma \mid \ldots \mid \alpha\,\delta$

We create a new non- terminal, **X**.

We then rewrite the above productions into:

$\mathbf{A} \rightarrow \alpha\mathbf{X} \quad \mathbf{X} \rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

For example,

Stmt → id = Expr ;

Stmt → id (Args) ;

becomes

Stmt → id StmtSuffix

StmtSuffix → = Expr ;

StmtSuffix → (Args) ;

Eliminating Left Recursion

Assume we have a non- terminal that is left recursive:

$\mathbf{A} \rightarrow \mathbf{A}\alpha \quad \mathbf{A} \rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

To eliminate the left recursion, we create two new non- terminals, **N** and **T**.

We then rewrite the above productions into:

$\mathbf{A} \rightarrow \mathbf{N}\,\mathbf{T} \quad \mathbf{N} \rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

$\mathbf{T} \rightarrow \alpha\,\mathbf{T} \mid \lambda$

For example,

Expr → Expr + id

Expr → id

becomes

Expr → N T

N → id

T → + id T | λ

This simplifies to:

Expr → id T

T → + id T | λ

Reading Assignment

Read Sections 6.1 to 6.5.1 of **Crafting a Compiler**.

How does JavaCup Work?

The main limitation of LL(1) parsing is that it must predict the correct production to use when it first starts to match the production's righthand side.

An improvement to this approach is the LALR(1) parsing method that is used in JavaCUP (and Yacc and Bison too).

The LALR(1) parser is bottom- up in approach. It tracks the portion of a righthand side already matched as tokens are scanned. It may not know immediately which is the correct production to choose, so it tracks *sets* of possible matching productions.

Configurations

We'll use the notation

X → A B • C D

to represent the fact that we are trying to match the production **X → A B • C D** with **A** and **B** matched so far.

A production with a "•" somewhere in its righthand side is called a *configuration*.

Our goal is to reach a configuration with the "dot" at the extreme right:

X → A B C D •

This indicates that an entire production has just been matched.

Since we may not know which production will eventually be fully matched, we may need to track a *configuration set*. A configuration set is sometimes called a *state*.

When we predict a production, we place the "dot" at the beginning of a production:

X → • A B C D

This indicates that the production may possibly be matched, but no symbols have actually yet been matched.

We may predict a λ- production:

X → λ •

When a λ- production is predicted, it is immediately matched, since λ can be matched at any time.

Starting the Parse

At the start of the parse, we know some production with the start symbol must be used initially. We don't yet know which one, so we predict them *all*:

S → • A B C D

S → • e F g

S → • h I

...

Closure

When we encounter a configuration with the dot to the left of a non- terminal, we know we need to try to match that non-terminal.

Thus in

X → • A B C D

we need to match some production with A as its left hand side.

Which production?

We don't know, so we predict *all* possibilities:

A → • P Q R

A → • s T

...

The newly added configurations may predict other non- terminals, forcing additional productions to be included. We continue this process until no additional configurations can be added.

This process is called *closure* (of the configuration set).

Here is the closure algorithm:

```
ConfigSet Closure(ConfigSet C){
  repeat
     if (X → a •B d is in C &&
           B is a non-terminal)
       Add all configurations of
          the form   B → •g to C)
  until (no more configurations
              can be added);
  return C;
}
```


Example of Closure

Assume we have the following grammar:

S → A b

A → C D

C → D

C → c

D → d

To compute Closure(**S → • A b**) we first include all productions that rewrite A:

A → • C D

Now **C** productions are included:

C → • D

C → • c

Finally, the D production is added:

D → • d

The complete configuration set is:

S → • A b

A → • C D

C → • D

C → • c

D → • d

This set tells us that if we want to match an **A**, we will need to match a **C**, and this is done by matching a **c** or **d** token.

Shift Operations

When we match a symbol (a terminal or non- terminal), we *shift* the "dot" past the symbol just matched. Configurations that don't have a dot to the left of the matched symbol are deleted (since they didn't correctly anticipate the matched symbol).

The **GoTo** function computes an updated configuration set after a symbol is shifted:

```
ConfigSet GoTo(ConfigSet C,Symbol X){
  B=φ;
  for each configuration f in C{
     if (f is of the form A → α•X δ)
       Add A → α X •δ to B;
  }
  return Closure(B);
}
```


For example, if **C** is

S → · **A b**

A → · **C D**

C → · **D**

C → · **c**

D → · **d**

and **X** is **C**, then **GoTo** returns

A → **C** · **D**

D → · **d**

Reduce Actions

When the dot in a configuration reaches the rightmost position, we have matched an entire righthand side. We are ready to replace the righthand side symbols with the lefthand side of the production. The lefthand side symbol can now be considered matched.

If a configuration set can shift a token and also reduce a production, we have a potential *shift/reduce error*.

If we can reduce more than one production, we have a potential *reduce/reduce error*.

How do we decide whether to do a shift or reduce? How do we choose among more than one reduction?

We examine the next token to see if it is consistent with the potential reduce actions.

The simplest way to do this is to use Follow sets, as we did in LL(1) parsing.

If we have a configuration

A → α ·

we will reduce this production *only if* the current token, **CT**, is in Follow(**A**).

This makes sense since if we reduce α to **A**, we can't correctly match **CT** if **CT** can't follow **A**.

Shift/Reduce and Reduce/ Reduce Errors

If we have a parse state that contains the configurations

A $\rightarrow \alpha \bullet$

B $\rightarrow \beta \bullet$ **a** γ

and **a** in Follow(**A**) then there is an *unresolvable* shift/reduce conflict. This grammar can't be parsed.

Similarly, if we have a parse state that contains the configurations

A $\rightarrow \alpha \bullet$

B $\rightarrow \beta \bullet$

and Follow(**A**) $\cap$ Follow(**B**) $\neq \phi$, then the parser has an unresolvable reduce/reduce conflict. This grammar can't be parsed.

Building Parse States

All the manipulations needed to build and complete configuration sets suggest that parsing may be slow—configuration sets need to be updated after each token is matched.

Fortunately, all the configuration sets we ever will need can be computed and tabled *in advance*, when a tool like Java Cup builds a parser.

The idea is simple. We first compute an initial parse state, s_0, that corresponds to predicting productions that expand the start symbol. We then just compute successor states for each token that might be scanned. A complete set of states can be computed. For typical

programming language grammars, only a few hundred states are needed.

Here is the algorithm that builds a complete set of parse states for a grammar:

```
StateSet BuildStates(){
 Let s0=Closure({S → •α, S → •β, ...});
 C={s0};
 while (not all states in C are marked){
  Choose any unmarked state, s, in C
  Mark s;
  For each X in
      terminals U nonterminals {
    if (GoTo(s,X) is not in C)
       Add GoTo(s,X) to C;
  }
 }
 return C;
}
```


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 → 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

Parse Stack	Top State	Action	Remaining Input
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!).

Reading Assignment

Read Chapter 8 of **Crafting a Compiler**.

Symbol Tables in CSX

CSX is designed to make symbol tables easy to create and use.

There are three places where a new scope is opened:

- In the class that represents the program text. The scope is opened as soon as we begin processing the **`classNode`** (that roots the entire program). The scope stays open until the entire class (the whole program) is processed.
- When a **`methodDeclNode`** is processed. The name of the method is entered in the top- level (global) symbol table. Declarations of parameters and locals are placed in the method's symbol table. A method's symbol table is closed after all the statements in its body are type checked.

- When a **`blockNode`** is processed. Locals are placed in the block's symbol table. A block's symbol table is closed after all the statements in its body are type checked.

CSX Allows no Forward References

This means we can do type-checking in *one pass* over the AST. As declarations are processed, their identifiers are added to the current (innermost) symbol table. When a use of an identifier occurs, we do an ordinary block- structured lookup, always using the innermost declaration found. Hence in

```
int i = j;
int j = i;
```

the first declaration initializes **`i`** to the nearest non- local definition of **`j`**.

The second declaration initializes **`j`** to the current (local) definition of **`i`**.

Forward References Require Two Passes

If forward references are allowed, we can process declarations in two passes.

First we walk the AST to establish symbol tables entries for all local declarations. No uses (lookups) are handled in this passes.

On a second complete pass, all uses are processed, using the symbol table entries built on the first pass.

Forward references make type checking a bit trickier, as we may reference a declaration not yet fully processed.

In Java, forward references to fields within a class are allowed.

Thus in


```
class Duh {
 int i = j;
 int j = i;
}
```

a Java compiler must recognize that the initialization of **i** is to the **j** field and that the **j** declaration is incomplete (Java forbids uninitialized fields or variables).

Forward references do allow methods to be mutually recursive. That is, we can let method **a** call **b**, while **b** calls **a**.

In CSX this is impossible!

(Why?)

Incomplete Declarations

Some languages, like C++, allow incomplete declarations.

First, part of a declaration (usually the header of a procedure or method) is presented.

Later, the declaration is completed.

For example (in C++):

```
class C {
  int i;
 public:
  int f();
};
int C::f(){return i+1;}
```


Incomplete declarations solve potential forward reference problems, as you can declare method headers first, and bodies that use the headers later.

Headers support abstraction and separate compilation too.

In C and C++, it is common to use a **#include** statement to add the headers (but not bodies) of external or library routines you wish to use.

C++ also allows you to declare a class by giving its fields and method headers first, with the bodies of the methods declared later. This is good for users of the class, who don't always want to see implementation details.