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 $\rightarrow \cdot$ **Expr + id**
  Expr $\rightarrow \cdot$ **id**

  we can first match an **id**:

  Expr $\rightarrow$ **id** $\cdot$

  Then the **Expr** is recognized:

  Expr $\rightarrow$ **Expr** $\cdot$ **+ 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 $\rightarrow$ **FieldDecls MethodDecls**
FieldDecls $\rightarrow$ **FieldDecl FieldDecls**
FieldDecls $\rightarrow \lambda$
MethodDecls $\rightarrow$ **MethodDecl MethodDecls**
MethodDecls $\rightarrow \lambda$
FieldDecl $\rightarrow$ **int id ;**
MethodDecl $\rightarrow$ **int id ( ) ; Body**

When we predict **MemberDecls** we get:

MemberDecls $\rightarrow$ • **FieldDecls MethodDecls**
FieldDecls $\rightarrow$ • **FieldDecl FieldDecls**
FieldDecls $\rightarrow$ λ•
FieldDecl $\rightarrow$ • **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** $\rightarrow$ 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.

Classes, Structs and Records

The fields and methods declared within a class, struct or record are stored within a individual symbol table allocated for its declarations.

Member names must be unique within the class, record or struct, but may clash with other visible declarations. This is allowed because member names are qualified by the object they occur in.

Hence the reference **`x.a`** means look up **`x`**, using normal scoping rules. Object **`x`** should have a type that includes local fields. The type of **`x`** will include a pointer to the symbol table containing the field declarations. Field **`a`** is looked up in that symbol table.

Chains of field references are no problem.

For example, in Java

```
System.out.println
```

is commonly used.

`System` is looked up and found to be a class in one of the standard Java packages (**`java.lang`**). Class **`System`** has a static member **`out`** (of type **`PrintStream`**) and **`PrintStream`** has a member **`println`**.