ECE251: Programming Languages & Translators Fall 2010

Lecture 13 — October 7, 2010

Patrick Lam *version 1*

Parser Generators

Let's step back a bit and put parser generation in context. Recall the big picture of a compiler:

$$\begin{tikzcd}\boxed{\text{scan}} \arrow[r] & \boxed{\text{parse}} \arrow[r] & \boxed{\text{weed}} \arrow[r] & \boxed{\text{symbol}} \arrow[r] & \boxed{\text{type}} \arrow[r] & \boxed{\text{code}} \arrow[r] & \boxed{\text{emit}}\end{tikzcd}$$

ANTLR (and other parser generators) can help with the first three stages. You provide a specification of the tokens and the tree, and provide hooks for describing the AST, and you get code that produces ASTs from inputs. It is a *domain-specific language* for describing the first three stages of compilation.

ANTLR leaves you on your own for the other four stages. After we finish our discussion of parsing, we will talk about symbol tables, type checking, and a bit about emitting code.

Alternatives. There are a lot of alternatives to ANTLR, which you might use in different situations. For instance, if you just need a lexer, then you might use `flex` instead of ANTLR, since it only lexes. There are also practical differences, like ANTLR's requirement to include its runtime `jar` in the classpath for its generated Java code. Many parser generators generate bottom-up parsers, which have different limitations than top-down parsers.

Bottom-up Parsing

Top-down parsers are also known as LL parsers. Bottom-up parsers used in practice are LALR parsers. I briefly said how bottom-up parsers worked last time: they build up parse trees starting at leaves, and put them together. Many compiler courses go on at length about how LALR parsers work. I won't; I don't think it's useful.

Fixing shift-reduce conflicts. We have, in fact, seen many of the techniques you need to solve shift-reduce conflicts. We'll now present a few techniques for solving them, in case you have to use an LALR parser generator.

First of all, the parser *reduces* when it decides that it has seen enough tokens to form a given nonterminal. On the other hand, it *shifts* when it doesn't have enough tokens and is waiting for more. For instance, say we had the simple grammar,

$$T := \texttt{LIT '+' LIT}$$

Upon reading a `LIT`, we'd shift it onto the stack, as well as the `'+'`. Finally, we'd shift the last `LIT` onto the stack, and reduce the three items on the stack to a single `T`.

We will next consider a few examples from

`http://www.cs.uiuc.edu/class/sp10/cs421/lectures/lecture%2010%20supp.pdf`

Example 1. Here is an unambiguous grammar:

$$\begin{aligned}
\mathit{Stmt} &:= \mathit{MethodCall} \mid \mathit{ArrayAsgn} \\
\mathit{MethodCall} &:= \mathit{Target}\ \texttt{'('}\ \texttt{')'}\ \texttt{';'} \\
\mathit{Target} &:= \texttt{id} \mid \texttt{id}\ \texttt{'.'}\ \texttt{id} \\
\mathit{ArrayAsgn} &:= \texttt{id}\ \texttt{'('}\ \texttt{int}\ \texttt{')'}\ \texttt{'='}\ \texttt{int}\ \texttt{';'}
\end{aligned}$$

However, `bison` will report the following error:

```
3: shift/reduce conflict (shift 8, reduce 4) on oparen
state 3
     TARGET : id . (4)
     TARGET : id . dot id (5)
     ARRAYASGN : id . oparen int cparen equal int semic (6)

     oparen shift 8
     dot shift
```

The problem here is that the parser won't know, after shifting an `id` and seeing an `oparen`, whether it has a simple `TARGET` or an `ARRAYASGN`. But it needs to decide right now.

The usual answer is to delay the decision. We inline the production:

$$\begin{aligned}
\mathit{Stmt} &:= \mathit{MethodCall} \mid \mathit{ArrayAsgn} \\
\mathit{MethodCall} &:= \texttt{id}\ \texttt{'('}\ \texttt{')'}\ \texttt{';'} \mid \mathit{Target}\ \texttt{'('}\ \texttt{')'}\ \texttt{';'} \mid \\
\mathit{Target} &:= \texttt{id}\ \texttt{'.'}\ \texttt{id} \\
\mathit{ArrayAsgn} &:= \texttt{id}\ \texttt{'('}\ \texttt{int}\ \texttt{')'}\ \texttt{'='}\ \texttt{int}\ \texttt{';'}
\end{aligned}$$

So we've moved the offending alternative from *Target* up into *MethodCall.* When the parser has an `id` and sees a `oparen` in its lookahead, it will then shift the `oparen`, and it can later decide if it is goign to have a *MethodCall* or an *ArrayAsgn.*

Example 2. Another grammar (fragment):

$$\begin{aligned}
\mathit{MethodBody} &:= \mathit{StmtList}\ \texttt{'return'}\ \texttt{id}\ \texttt{';'} \\
\mathit{StmtList} &:= \mathit{Stmt}\ \mathit{StmtList} \mid \mathit{Stmt} \\
\mathit{Stmt} &:= \texttt{'return'}\ \texttt{id}\ \texttt{';'}
\end{aligned}$$

This is right-recursive, and results in the following complaint:

```
8: shift/reduce conflict (shift 5, reduce 3) on return
state 8
     STMTLIST : STMT . STMTLIST (2)
     STMTLIST : STMT . (3)

     return shift 5
     id shift 6
     STMTLIST goto 12
     STMT goto 8
```

Here, we have shifted a *Stmt* and our lookahead is a `return`. Unfortunately, we don't know if the lookahead is the final return in the *MethodBody* (we finished our *StmtList* and we should reduce) or not (there are more *Stmt*s coming and we should shift), and we have to commit to one of these alternatives now.

A solution here is to convert the grammar to use left-recursion instead:

$$\textit{StmtList} \quad := \quad \textit{StmtList Stmt} \;\mid\; \textit{Stmt}$$

LALR supports right recursion, but here it would need more lookahead.

Example 3. The ambiguous expression grammar also produces a shift/reduce conflict:

$$E \;:=\; E \texttt{ '+' } E \mid E \texttt{ '*' } E \mid \texttt{id} \mid \texttt{'('} \; E \; \texttt{')'}$$

The shift/reduce conflict arises due to ambiguity. Here is the error:

```
10: shift/reduce conflict (shift 7, reduce 1) on plus
10: shift/reduce conflict (shift 8, reduce 1) on star
state 10
     Expr : Expr . plus Expr  (1)
     Expr : Expr plus Expr .  (1)
     Expr : Expr . star Expr  (2)
     plus shift 7
     star shift 8
     $end reduce 1
     cparen reduce 1
```

After it sees $E + E$ and gets either a $+$ or a $*$, it doesn't know if it should shift the new token or if it should reduce what it has already to an E.

Stratifying the grammar, as we've seen before, solves this problem. We tell the parser exactly which parse tree it should generate.

$$\begin{array}{lcl} E & := & E \texttt{ '+' } T \\ T & := & T \texttt{ '*' } F \mid F \\ F & := & \texttt{id} \mid \texttt{'('} \; E \; \texttt{')'} \end{array}$$

Or, you can use precedence and associativity declarations in `bison`-style parser generators.

```
%left plus
%left star
```

Example 4. We won't actually talk about this example, but it's the dangling else problem from earlier. You can also tell `bison` about the precedence here:

```
stmt:
  | IF LPAREN expression RPAREN stmt ELSE stmt
  | IF LPAREN expression RPAREN stmt %prec ELSE
```