Fixing Grammars

Fixing shift/reduce conflicts
A typical case where lots of shift/reduce conflicts occur in the definition of the syntax of most programming languages is in the expression language portion of the language. For example:

\[ \text{Expr ::= Expr T_MINUS Expr} \]

can result in shift/reduce conflicts, as when given the input:

\[ a - b - c \]

It is ambiguous what to do when seeing the second “-”—whether to reduce the \( a - b \) or to shift. By declaring the various infix operators to have left associative (where appropriate for the semantics of the language), the shift/reduce conflicts can be removed.

Fixing reduce/reduce conflicts
Parser generators typically only accept LALR(1) grammars. The transformation techniques discussed earlier in the semester take an EBNF to a BNF form. These are equivalent expressive power, as both are context-free grammars (CFGs). LALR(1) grammars are a subset of CFGs, and almost all programming language grammars can be expressed in an LALR(1) form. Once we submit a grammar to a parser generator, if the submitted grammar is not LALR(1), then the tool will report shift/reduce or reduce/reduce conflicts. The goal then is to remove the conflicts while still accepting the same set of input strings, also known as accepting the same language. One of the more common problems that comes up are reduce/reduce conflicts where the use of the productions is very similar. As an example, consider the following reduce/reduce conflict:

Reduce/Reduce conflict on \( \text{T_LEFTSQUARE} \)
\[ \text{NameType ::= T_ID(*)} \]
\[ \text{Var ::= T_ID (*)} \]

where “(*)” indicates the point where the conflict occurs. Recall that a reduce/reduce conflict occurs when the parser cannot make a decision between two productions to reduce by. The technique we will use to remove this reduce/reduce conflict has several components:

1. Examine the context: find all the productions where the non-terminals on the left-hand side occur in a right-hand side
2. Generate examples: construct concrete instances of input strings that illustrate the reduce/reduce conflict
3. Determine if too specific: identify whether the problem is that the amount of look-ahead (1 token) makes it impossible for the parser to recognize the difference between the two alternatives in the same context
4. Inline the smaller of the two conflicting productions into all the places where it is used.

In our example, \( \text{Var} \) has the following other productions and is used in two productions:

\[ \text{Var ::= T_ID} \]
\[ \quad \text{Exp T_Period T_ID} \]
\[ \quad \text{Exp T_LEFTSQUARE Exp T_RIGHTSQUARE ;} \]

\[ \text{Exp ::= Var} \]
\[ \text{AssignExp ::= Var T_EQ Exp} \]

In our example, \( \text{NameType} \) has the following other productions and is used in one production:
Fixing Grammars

NameType ::= T_INT
| T_BOOLEAN
| T_ID ;

Type ::= NameType T_LEFTSQUARE T_RIGHTSQUARE

The basic problem here is that the parser cannot distinguish between array assignment and array usage. Here are concrete examples of the problem:

\[
a[i] = 5
\]

vs.

\[
a[] \text{ foo}
\]

The first parses into (AssignExp (Var (a)) (T_EQ) (Exp (T_INT (5)))) and the second into (Type (NameType (T_ID (a))) (T_LEFTSQUARE) (T_RIGHTSQUARE)). Inlining NameType into its single use, in Type, results in the following:

Type ::= T_INT T_LEFTSQUARE T_RIGHTSQUARE
| T_BOOLEAN T_LEFTSQUARE T_RIGHTSQUARE
| T_ID T_LEFTSQUARE T_RIGHTSQUARE ;

and the removal of the NameType production. These transformation is a simplified form of what happens in the real grammar for Java, the discussion of which is included below.

19.1.4 Problem #4: Array Type versus Array Access

Consider the productions (shown after problem #1 has been corrected):

\[
ArrayType:
Type [ ]
\]

and:

\[
ArrayAccess:
Name [ Expression ]
PrimaryNoNewArray [ Expression ]
\]

Now consider the partial input:

\[
\text{class Problem4} \{ \text{Problem4() \{} \text{peter[}
\]

When the parser is considering the token Peter, with one-token lookahead to symbol [, it cannot yet tell whether Peter will be part of a type name, as in:

\[
\text{peter[]} \text{ team;}
\]

or part of an array access, as in:

\[
\text{peter[3]} = 12;
\]

Therefore, after the parser reduces Peter to the nonterminal Name, it cannot tell with only one-token lookahead whether Name should be reduced ultimately to Type (for an array type) or left
Fixing Grammars

alone (for an array access). Therefore, the productions shown above result in a grammar that is not LALR(1). The solution is to have separate alternatives for ArrayType:

\[
\text{ArrayType:} \\
\text{PrimitiveType [ ]} \\
\text{Name [ ]} \\
\text{ArrayType [ ]}
\]

This allows the parser to reduce \texttt{peter} to Name and then leave it as is, delaying the decision as to whether an array type or array access is in progress.

Syntax Errors With Generated Grammar

Once the grammar has successfully passed through the parser generator, one has to test it. The first thing to remember is that you can’t test productions of a grammar in isolation, given most parser-generator tools based on LALR(1). This implies that the example input must be a complete “piece”. For example, given a Java-like grammar, the following might be the shortest possible program:

\[
\text{class A { } }
\]

Second, once this file is created, you might still get syntax errors. To catch these, add debugging print statements as the action code for all your grammars. If you are using Java_cup, then this will look something like this:

\[
\text{Expr ::= Expr T_MINUS Expr} \\
\{: \text{System.out.println(“Got Expr T_MINUS Expr”);} \\
\text{System.out.flush(); :}
\]

There are several things to note. One is that action code is surrounded by “curly-colons”—curly braces with colons on the inside. Another is that \texttt{System.out} is being explicitly flushed. Otherwise, one can spend way too much time trying to figure out why a particular production isn’t being reached, when in fact, it is.