A Quick Introduction to Handling Conflicts in Yacc Parsers

Saumya Debray
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721

September 10, 2008

1 How Yacc-generated Parsers Work

Yacc is a tool that takes a context-free grammar as input and (where possible) produces a parser for that grammar as output. A message from yacc indicating a "conflict" means that at some point in the course of generating a parser for its input grammar, it faced a choice between multiple possible actions for the parser to take. While yacc uses default rules to choose one of these actions in such cases, the very presence of a conflict means that, potentially, something may be wrong and deserves careful examination.

Yacc produces parsers that belong to a family known as "LR parsers." Intuitively, such parsers can be thought of as similar to finite state machines, in that they have states and transitions between states; one big difference, however, is that unlike the finite automata used in lexical analysis, the parser also has a stack that it uses for pushing and popping grammar symbols (it is this stack that allows the parser to go beyond regular languages and handle context-free languages). Broadly speaking, such parsers repeatedly perform two actions: *shift* and *reduce*. The general idea of the parsing algorithm is to match the input that has been seen with the right hand sides of grammar productions until an appropriate match is found, to replace it with the corresponding left hand side. To "match the input ... until an appropriate match is found," the parser saves the grammar symbols it has found so far on its stack. A *shift* action takes the next token from the input and pushes it onto the stack. A *reduce* action happens when the sequence of grammar symbols on top of the parser's stack matches the right hand side $x_1x_2 \cdots x_n$ of an appropriate grammar production

$$A \rightarrow x_1x_2 \cdots x_n$$

and the parser decides to "reduce" this to the left hand side of the production, i.e., pop the string $x_1x_2 \cdots x_n$ off the stack and push A instead.

The details of why and how these actions are chosen, and how they help with parsing, will be discussed in class, or may be found in a compiler text. *For our purposes, it suffices to understand the main intuition here: a* shift *action means that the parser is waiting to see more of the input before it decides that a match has been found; a* reduce *action means that a match has been found.*

There are two types of conflics we might encounter: *shift-reduce* and *reduce-reduce*:

- *Shift-reduce conflict.* This occurs when the parser is faced with a choice of a shift action and a reduce action. (Yacc's default action in the case of a shift-reduce conflict is to choose the *shift* action.)

- *Reduce-reduce conflict.* This occurs when the parser is faced with a choice of two different productions that could be used for a reduce action. (Yacc's default action in the case of a reduce-reduce conflict is to reduce using the production that comes first, textually, in the input grammar specification.)

2 Parser States and the y.output File

In order to understand the source of a parser conflict, we need to know a little more about parser states. Intuitively, the parser processes a whole bunch of grammar rules "in parallel," using the next input token to guide this processing at each stage. Each parser state, therefore, corresponds to one or more grammar rules in different stages of being processed. We'll use a '$\bullet$' to mark the point upto which a given rule has been processed: e.g., given a rule '$A \rightarrow \mathbf{a}\ \mathbf{b}\ \mathbf{c}$' where the first two symbols in the right hand side have been processed, but not the third, we'll write

$$A \rightarrow \mathbf{a}\ \mathbf{b} \bullet \mathbf{c}$$

Each parser state can be thought of as being labelled with a number of such "dotted rules" (usually called "items"). In order to figure out the reason for a conflict, we have to find out (1) which state has a conflict; and (2) the reason for the conflict.

The first step is to use the '`-v`' option for yacc, which causes it to create a file `y.output` containing verbose information about the parser's states. As a concrete example, consider the following simple grammar for generating strings that contain an even number of **0**s:

Grammar 1:

$$S \rightarrow \mathbf{0}\ S\ \mathbf{0}$$
$$S \rightarrow \varepsilon$$

The corresponding Yacc rules are:

```
S : '0' S '0'
  | /* epsilon */
  ;
```

On invoking yacc, we get the following:

```
% yacc -d -v grammar.y
yacc: 1 shift/reduce conflict
```

The `-v` option causes yacc to generate the file `y.output`, which contains, among other things, the following:

```
   0  $accept : S $end
   1  S : '0' S '0'
   2    |
...
1: shift/reduce conflict (shift 1, reduce 2) on '0'
state 1
        S : '0' . S '0'  (1)
        S : .  (2)

        '0'  shift 1
        S  goto 3
...
```

The first three lines shown, which appear at the beginning of the `y.output` file, simply lists the grammar rules (rule 0 is an artificial one introduced by yacc for convenience; rules 1 and 2 are those appearing in the input to yacc). After this, each parser state is shown together with the corresponding set of items ("dotted grammar rules"); here we show the one state that happens to have a conflict.

The following line tells us the location and cause of the conflict, with the relevant phrases shown in boxes:

$\boxed{\texttt{1}}$: `shift/reduce conflict (`$\boxed{\texttt{shift 1, reduce 2}}$`)` $\boxed{\texttt{on '0'}}$

The different parts of this line are to be understood as follows:

1. The '`1`' at the beginning tells us which state has the conflict (state 1).

2. The next item specifies the conflicting actions:

   (a) The '`shift 1`' means that one action is "*shift, go to (i.e., stay in) state 1.*"

   (b) The '`reduce 2`' means that the other action is "*reduce using rule #2*" where the rule number is with respect to the list at the beginning of the `y.output` file, i.e., the rule '$S \to \varepsilon$'.

3. The '`on '0'`' at the end indicates the input token on which this conflict arises: in this case, '**0**'.

This is followed by information about state 1. The first two lines after this tell us about the items in this state:

Yacc *output*	*"Dotted" notation*
`S : '0' . S '0'`	$S \to \mathbf{0} \bullet S\ \mathbf{0}$
`S : .`	$S \to \bullet$

The final two lines refer to the parser's actions in this state. The line '`'0' shift 1`' means "*on input* **0**, *shift and stay in state 1.*" The line '`S goto 3`' refers to the state transition on a reduce action, and are not relevant for this discussion.

3 Diagnosing Conflicts

3.1 Shift/Reduce Conflicts

As shown above, the `y.output` file shows us the cause and location of the shift/reduce conflict. In the example above, we have:

```
1: shift/reduce conflict (shift 1, reduce 2) on '0'
```

Given our basic intuition about what items in a state as well as shift and reduce actions mean, this means that yacc is considering the following possibilities at this point If the next input token is `'0'`:

1. The `'0'` in the input comes from the 'S' that this parser state is about to process (i.e., which is to the immediate right of the '$\bullet$'), so the parser should shift this input token (and perhaps others) waiting to eventually reduce using the grammar rule '$S \to \mathbf{0}\ S\ \mathbf{0}$.'

2. The `'0'` in the input is actually that at the end of the item '$S \to \mathbf{0} \bullet S\ \mathbf{0}$', which means that the 'S' that the parser state is about to process should "disappear" via the production '$S \to \varepsilon$'. In this case, the parser should carry out a reduce action using this rule.

3.2 Reduce/Reduce Conflicts

As the name suggests, reduce/reduce conflict arises when there are (at least) two different grammar rules that can be used for a reduce action in some parser state. As an example, consider the following variation on the grammar shown above:

Grammar 2:

$S \to T$
$S \to \varepsilon$
$T \to \mathbf{0}\ \mathbf{0}\ T$
$T \to \varepsilon$

Running yacc on this grammar produces a `y.output` file that contains the following:

```
   0  $accept : S $end
   1  S : T
   2    |
   3  T : '0' '0' T
   4    |
...
0: reduce/reduce conflict (reduce 2, reduce 4) on $end
state 0
        $accept : . S $end  (0)
        S : .  (2)
        T : .  (4)

        '0'  shift 1
        $end  reduce 2

        S  goto 2
        T  goto 3
```

The following line gives the source of the conflict:

```
0: reduce/reduce conflict (reduce 2, reduce 4) on $end
```

Here, '`reduce 2`' refers to reducing using grammar rule 2, '$S \to \varepsilon$'; '`reduce 4`' refers to reducing using grammar rule 4, '$T \to \varepsilon$'; and `$end` refers to EOF. The reason for the conflict, therefore, is that if the next input "token" is EOF (i.e., the input string is 'ε') then we can either derive this string directly as $S \Rightarrow \varepsilon$ (which would then have us reduce using the rule '$S \to \varepsilon$' at this point), or we can derive it as $S \Rightarrow T \Rightarrow \varepsilon$ (which would have us first reduce using '$T \to \varepsilon$').

4 Removing Conflicts

Once we have figured out the actual reason for a conflict, we can try and modify the grammar to remove the conflict without changing the overall language. We can do this either by changing the grammar to remove the source of the conflict, or give yacc additional information to allow it to figure out which of the conflicting actions is the right one to choose, or (if we know what we're doing) we can leave the conflict alone and rely on yacc's default action.

It's important to realize that conflicts cannot always be removed: there are some grammars for which, no matter what we do, we cannot remove all conflicts while preserving equivalence of grammars. The approaches outlined below are therefore heuristics that often work, though they may not always do so.

4.1 Grammar Restructuring

In the case of Grammar 1 above, the problem arises due to the rule '$S \to \varepsilon$' in the recursion. Given this understanding, we can try to either get rid of this rule, or move it out of the context where the conflict

(a) *shift* action (b) *reduce* action

Figure 1: Parsing choices for expressions

arises. In this particular example, the problem arises because in parser state 1, we can't tell, in the item '$S \rightarrow \mathbf{0} \bullet S\ \mathbf{0}$', whether an input `'0'` comes from the S ($\Rightarrow$ shift action) or is the '**0**' at the end of the item ($\Rightarrow$ reduce action), so we can restructure the grammar as follows to eliminate this source of confusion:

$S \rightarrow \mathbf{0}\ \mathbf{0}\ S$
$S \rightarrow \varepsilon$

With Grammar 2, the problem is that the string 'ε' can be derived in two ways. One of these, using '$S \rightarrow \varepsilon$', is unnecessary. Getting rid of this rule removes the conflict.

4.2 Providing Additional Information to Yacc

Ambiguities resulting from a lack of information about operator precedences and associativities are a common source of shift/reduce conflicts. These typically arise in situations involving expressions. They can be eliminated using declarations such as `%left` and `%right` (and sometimes `%prec`) to operator precedences and associativities.

As an example, consider the following simple grammar for expressions:

Grammar 3:
$E \rightarrow E + E$
$E \rightarrow \mathbf{id}$

Running yacc on this gives the following:

```
   0  $accept : E $end
   1  E : E '+' E
   2    | id
...
4: shift/reduce conflict (shift 3, reduce 1) on '+'
state 4
        E : E . '+' E  (1)
        E : E '+' E .  (1)

        '+'  shift 3
        $end  reduce 1
```

Here, when the parser sees the token `'+'`, it can't decide between the following choices:

1. Assume that the `'+'` comes from the item '$E \rightarrow E \bullet + E$', as indicated by the arrow in Figure 1(a). In this case the parser needs to see the rest of the right hand side of this production before it can

reduce, so the appropriate action is to shift. This choice has the effect of making the operator '+' right-associative.

2. Assume that the `'+'` comes from a different recursive level of the variable E, as indicated by the arrow shown in Figure 1(b). This means that the current recursive level of E is finished, corresponding to the item '$E \rightarrow E + E \bullet$ ', and that the parser should carry out a reduce action using rule 1, i.e., '$E \rightarrow E + E$'.

It is possible to restructure the grammar to remove this conflict (the details can be found in compiler texts under the heading of *Top-down Parsing*). A simpler alternative is to simply add a declaration to the `yacc` input file indicating the desired associativity of the '+' operator.

5 Conflicts Introduced by Actions

Yacc rules can have actions embedded in them, and the way in which Yacc handles such actions can give rise to conflicts.

An action in a rule consists of a sequence of C statements enclosed within braces. An action can appear anywhere in the right side of a rule, e.g., the following illustrates a rule '$X \rightarrow Y\ Z$' with three actions (shown underlined) added to it:

$$X \rightarrow \underline{\{action_1\}}\ Y\ \underline{\{action_2\}}\ Z\ \underline{\{action_3\}}$$

For technical reasons, it turns out to be convenient to transform the grammar so that actions always appear at the very end (i.e., all the way on the right) of a rule. This can be done by introducing new nonterminals, called *marker nonterminals*, in a way that does not change the language of the grammar but has the effect of moving actions so that they only occur at the ends of rules. Each action A_i that is not already at the end of a rule is replaced by a new (marker) nonterminal M_i, which is defined by the rule '$M_i \rightarrow \varepsilon$' together with the action A_i at the end. Thus, the rule shown above is transformed to the following, where M_1 and M_2 are marker nonterminals:

$$\begin{array}{l} X \rightarrow M_1\ Y\ M_2\ Z\ \underline{\{action_3\}} \\ M_1 \rightarrow \varepsilon\ \underline{\{action_1\}} \\ M_2 \rightarrow \varepsilon\ \underline{\{action_2\}} \end{array}$$

The ϵ-productions introduced by this transformation can give rise to conflicts that were absent in the grammar without the actions. This is illustrated by the following grammar fragment for a very simple programming language, where a program consists of a single function prototype or a single function definition:

$$\begin{array}{lcl} \textit{prog} & \rightarrow & \textit{prototype} \mid \textit{function} \\ \textit{prototype} & \rightarrow & \textbf{int id} \text{ '(' } \textit{formals} \text{ ')'} \\ \textit{function} & \rightarrow & \textbf{int id} \text{ '(' } \textit{formals} \text{ ')' } \textit{func\_body} \end{array}$$

Yacc does not find any conflicts in this grammar fragment. However, suppose that, before processing the formal parameters, we want to change the value of a variable, *scope*, to the value `Local`. Our first attempt to do this might be the following:

```
prog : prototype  | function ;
prototype  : INT ID '(' formals ')' ;
function : INT ID { scope = Local; } '(' formals ')' func_body ;
```

The action added to the last rule of this grammar produces the following output from yacc:

```
yacc: 1 rule never reduced
yacc: 1 shift/reduce conflict
```

An examination of the `y.output` file shows the following:

```
   0  $accept : prog $end
   1  prog : prototype
   2       | function
   3  prototype : INT ID '(' params ')'
   4  $$1 :
   5  function : INT ID $$1 '(' params ')' func_body
...
5: shift/reduce conflict (shift 6, reduce 4) on '('
state 5
        prototype : INT ID . '(' params ')'  (3)
        function : INT ID . $$1 '(' params ')' func_body  (5)
        $$1 : .  (4)

        '('  shift 6
        $$1  goto 7
...
Rules never reduced:
        $$1 :  (4)

State 5 contains 1 shift/reduce conflict.
```

Notice that the marker nonterminal '`$$1`', together with the rule defining it (rule 4), are shown explicitly at the beginning of `y.output` (however, the action associated with the marker nonterminal is not shown).

State 5, which has the conflict, contains the following three items:

$$\begin{array}{lll} \textit{prototype} & \rightarrow & \textbf{int id} \bullet \text{ `('}\ \textit{params}\ \text{`)'} \\ \textit{function} & \rightarrow & \textbf{int id} \bullet \texttt{\$\$1}\ \text{`('}\ \textit{params}\ \text{`)'}\ \textit{func\_body} \\ \texttt{\$\$1} & \rightarrow & \bullet \end{array}$$

The first item, where the token '**(**' appears immediately after the $\bullet$, gives rise to a shift action; it indicates that the parser may be processing the rule for the nonterminal *prototype* and be at the point where it has just seen the token **id**, which means that if the token '**(**' appears in the input it should be shifted onto its stack. The second item, where the $\bullet$ is followed by the marker nonterminal `$$1`, indicates that in this state the parser may also be processing the rule for the nonterminal *function* and be at the point where it has just seen the token **id**, which means that if the token '**(**' appears in the input the next action should be to reduce using the rule '`$$1` $\rightarrow \varepsilon$' (and carry out the associated action). Stated differently, when the parser is in this state and sees the token '**(**' in the input, it cannot tell whether this corresponds to the '**(**' in the rule for *prototype*, in which case there is no semantic action to be carried out, or to that in the rule for *function*, in which case a semantic action has to be carried out.

One might think that we can deal with this by putting the same semantic action in the same place in both the grammar rules, so that there would presumably be no question about carrying out the action. The resulting grammar is as follows:

```
prog : prototype  | function ;
prototype  : INT ID { scope = Local; } '(' formals ')' ;
function : INT ID { scope = Local; } '(' formals ')' func_body ;
```

When we run yacc on this grammar, we get the following output:

```
yacc: 1 rule never reduced
yacc: 1 reduce/reduce conflict
```

The problem is that, unfortunately, yacc doesn't recognize that the two actions are identical, but treats them as different actions and generates a different marker nonterminal for each action. The resulting `y.output` is as follows:

```
   0  $accept : prog $end
   1  prog : prototype
   2       | function
   3  $$1 :
   4  prototype : INT ID $$1 '(' params ')'
   5  $$2 :
   6  function : INT ID $$2 '(' params ')' func_body
...
5: reduce/reduce conflict (reduce 3, reduce 5) on '('
state 5
        prototype : INT ID . $$1 '(' params ')'  (4)
        function : INT ID . $$2 '(' params ')' func_body  (6)
        $$1 : .  (3)
        $$2 : .  (5)

        .  reduce 3

        $$1  goto 6
        $$2  goto 7
...
Rules never reduced:
        $$2 :  (5)

State 5 contains 1 reduce/reduce conflict.
```

We can deal with this situation by explicitly introducing our own "marker nonterminal":

```
prog : prototype  | function ;
prototype  : INT ID UpdateScope2Local '(' formals ')' ;
function : INT ID UpdateScope2Local '(' formals ')' func_body ;
UpdateScope2Local : /* epsilon */ { scope = Local; } ;
```

This grammar goes through yacc without any conflicts.