Chapter 7. Ambiguities and Conflicts

This chapter focuses on finding and correcting *conflicts* within a bison grammar. Conflicts occur when bison reports shift/reduce and reduce/reduce errors. Bison lists any errors in the listing file `name.output`, which we will describe in this chapter, but it can still be a challenge to figure out what's wrong with the grammar and how to fix it. Before reading this chapter, you should understand the general way that bison parsers work, described in Chapter 3.

The Pointer Model and Conflicts

To describe what a conflict is in terms of the bison grammar, we introduce a model of bison's operation. In this model, a *pointer* moves through the bison grammar as each individual token is read. When you start, there is one pointer (represented here as an up arrow, ↑) at the beginning of the start rule:

```
%token A B C
%%
start:     ↑ A B C;
```

As the bison parser reads tokens, the pointer moves. Say it reads `A` and `B`:

```
%token A B C
%%
start:     A B ↑ C;
```

At times, there may be more than one pointer because of the alternatives in your bison grammar. For example, suppose with the following grammar it reads `A` and `B`:

```
%token A B C D E F
%%
start:      x
     |      y;
```

```
x:    A B ↑ C D;
y:    A B ↑ E F;
```

(For the rest of the examples in this chapter, all capital letters are tokens, so we will leave out the `%token` and the `%%`.) There are two ways for pointers to disappear. One happens when a subsequent token doesn't match a partially matched rule. If the next token that the parser reads is `C`, the second pointer will disappear, and the first pointer advances:

```
start:      x
     |      y;
x:    A B C ↑ D;
y:    A B E F;
```

The other way for pointers to disappear is for them to merge in a common subrule. In this example, `z` appears in both `x` and `y`:

```
start:      x
     |      y;
x:    A B z R;
y:    A B z S;
z:    C D
```

After reading `A`, there are two pointers:

```
start:      x
     |      y;
x:    A ↑ B z R;
y:    A ↑ B z S;
z:    C D
```

After `A B C`, there is only one pointer, in rule `z`:

```
start:      x
     |      y;
x:    A B z R;
y:    A B z S;
z:    C ↑ D;
```

And after `A B C D`, the parser has completed rule `z`, and there again are two:

```
start:      x
     |      y;
x:    A B z ↑ R;
y:    A B z ↑ S;
z:    C D;
```

When a pointer reaches the end of a rule, the rule is *reduced*. Rule `z` was reduced when the pointer got to the end of it after the parser read `D`. Then the pointer returns to the rule from which the reduced rule was called, or as in the earlier case, the pointer splits up into the rules from which the reduced rule was called.

There is a conflict if a rule is reduced when there is more than one pointer. Here is an example of reductions with only one pointer:

```
start:      x
     |      y;
x:    A ↑ ;
y:    B ;
```

After `A`, there is only one pointer—in rule `x`—and rule `x` is reduced. Similarly, after `B`, there is only one pointer—in rule `y`—and rule `y` is reduced.

Here is an example of a conflict:

```
start:     x
     |     y;
x:    A ↑ ;
y:    A ↑ ;
```

After `A`, there are two pointers, at the ends of rules `x` and `y`. They both want to reduce, so it is a *reduce/reduce* conflict.

There is no conflict if there is only one pointer, even if it is the result of merging pointers into a common subrule and even if the reduction will result in more than one pointer:

```
start:      x
     |      y;
x:    z R ;
y:    z S ;
z:    A B ↑ ;
```

After `A B`, there is one pointer, at the end of rule `z`, and that rule is reduced, resulting in two pointers:

```
start:    x
     |    y;
x:    z ↑ R;
y:    z ↑ S;
z:    A B;
```

But at the time of the reduction, there is only one pointer, so it is *not* a conflict.

Kinds of Conflicts

There are two kinds of conflicts, reduce/reduce and shift/reduce. Conflicts are categorized based upon what is happening with the other pointer when one pointer is reducing. If the other rule is also reducing, it is a reduce/reduce conflict. The following example has a *reduce/reduce* conflict in rules `x` and `y`:

```
start:     x
     |     y;
x:    A ↑ ;
y:    A ↑ ;
```

If the other pointer is not reducing, then it is shifting, and the conflict is a *shift/reduce* conflict. The following example has a shift/reduce conflict in rules `x` and `y`:

```
start:    x
     |    y R;
x:    A ↑ R;
y:    A ↑ ;
```

After the parser reads `A`, rule `y` needs to reduce to rule `start`, where `R` can then be accepted, while rule `x` can accept `R` immediately.

If there are more than two pointers at the time of a reduce, bison lists the conflicts. The following example has a reduce/reduce conflict in rules `x` and `y` and another reduce/reduce conflict in rules `x` and `z`:

```
start:     x
     |     y
     |     z;
x:    A ↑ ;
y:    A ↑ ;
z:    A ↑ ;
```

Let's define exactly when the reduction takes place with respect to token *lookahead* and pointers disappearing so we can keep our simple definition of conflicts correct. Here is a reduce/reduce conflict:

```
start:      x B
     |      y B;
x:    A ↑ ;
y:    A ↑ ;
```

But there is no conflict here:

```
start:     x B
     |     y C;
x:    A ↑ ;
y:    A ↑ ;
```

The reason the second example has no conflict is that a bison parser can look ahead one token beyond the `A`. If it sees a `B`, the pointer in rule `y` disappears before rule `x` is reduced. Similarly, if it sees a `C`, the pointer in rule `x` disappears before rule `y` is reduced.

A bison parser can look ahead only one token. The following would not be a conflict in a parser that could look ahead two tokens, but in a bison parser, it is a reduce/reduce conflict:

```
start:      x B C
     |      y B D;
x:    A ↑ ;
y:    A ↑ ;
```

A GLR parser can resolve this kind of conflict in situations where it's impractical to rewrite the grammar to avoid the conflict. See Chapter 9.

Parser States

Bison tells you about your grammar's conflicts in *name*`.output`, which is a description of the state machine it is generating. We will discuss what the states are, describe the contents of *name*`.output`, and then discuss how to find the problem in your bison grammar given a conflict described in *name*`.output`. You can generate *name*`.output` by running bison with the `-v` (verbose) option.

Each state corresponds to a unique combination of possible pointers in your bison grammar. Every nonempty bison grammar has at least three unique possible states: one at the beginning when no input has been accepted, one when a complete valid input has been accepted, and a third after the `$end` token has been accepted. The following simple example has two more states:

```
start:    A <one here> B <another here> C;
```

For future examples, we will number the states as a clear means of identification. Bison assigns a number to each state, but the particular numbers are not significant. Different versions of bison may number the states differently.

```
start:     A <state 1> B <state 2> C;
```

When a given stream of input tokens can correspond to more than one possible pointer position, then all the pointers for a given token stream correspond to one state:

```
start:        a
     |        b;
a:     X <state 1> Y <state 2> Z;
b:     X <state 1> Y <state 2> Q;
```

Different input streams can correspond to the same state when they correspond to the same pointer:

```
start:      threeAs;
threeAs: /* empty */
       | threeAs A <state 1> A <state2> A <state3>;
```

The previous grammar accepts some multiple of three `A` s. State 1 corresponds to 1, 4, 7, ... `A` s; state 2 corresponds to 2, 5, 8, ... `A` s; and state 3 corresponds to 3, 6, 9, ... `A` s. We rewrite this as a right-recursive grammar to illustrate the next point.

```
start:       threeAs;
threeAs:  /* empty */
       |  A A A threeAs;
```

A position in a rule does not necessarily correspond to only one state. A given pointer in one rule can correspond to different pointers in another rule, making several states:

```
start:      threeAs X
     |      twoAs Y;
threeAs: /* empty */
       | A A A threeAs;
twoAs: /* empty */
     | A A twoAs;
```

The grammar above accepts multiples of 2 or 3 `A` s, followed by an `X` for multiples of 3, or a `Y` for multiples of 2. Without the `X` or `Y` , the grammar would have a conflict, not knowing whether a multiple of 6 `A` s satisfied `threeAs` or `twoAs` . It would also have a conflict if we'd used left recursion, since it would have to reduce `twoAs` or `threeAs` before it saw a final `X` or `Y` . If we number the states as follows:

```
state 1: 1, 7, ... A's accepted
state 2: 2, 8, ... A's accepted
...
state 6: 6, 12, ... A's accepted
```

then the corresponding pointer positions are as follows:

```
start:      threeAs X
     |      twoAs Y;
threeAs: /* empty */
       | A <1,4> A <2,5> A <3,6> threeAs;
twoAs: /* empty */
     |  A <1,3,5> A <2,4,6> twoAs;
```

That is, after the first `A` in `threeAs`, the parser could have accepted 6i+1 or 6i+4 `A`s, where i is 0, 1, etc. Similarly, after the first `A` in `twoAs`, the parser could have accepted 6i+1, 6i+3, or 6i+5 `A`s.

Contents of name.output

Now that we have defined states, we can look at the conflicts described in *`name`*`.output`. The format of the file has varied among versions of bison, but it always includes a listing of all the rules in the grammar and all the parser states. It usually has a summary of conflicts and other errors at the beginning, including rules that are never used, typically because of conflicts. For each state, it lists the rules and positions that correspond to the state, the shifts and reductions the parser will do when it reads various tokens in that state, and what state it will switch to after a reduction produces a nonterminal in that state. We'll show some ambiguous grammars and the *`name`*`.output` reports that identify the ambiguities. The files that bison produces show the cursor as a dot, but we'll show it as an up arrow (↑) to make it easier to read and to be consistent with the examples so far.

Reduce/Reduce Conflicts

Consider the following ambiguous grammar:

```
start:      a Y
     |      b Y ;
a:    X ;
b:    X ;
```

When we run it through bison, a typical state description is as follows:

```
state 3

    1 start: a ↑ Y

    Y  shift, and go to state 6
```

In this state, the parser has already reduced an `a`. If it sees a `Y`, it shifts the `Y` and moves to state 6. Anything else would be an error. The ambiguity produces a reduce/reduce conflict in state 1:

```
state 1

    3 a: X ↑
    4 b: X ↑

    Y          reduce using rule 3 (a)
    Y          [reduce using rule 4 (b)]
    $default   reduce using rule 3 (a)
```

The fourth and fifth lines show a conflict between rule 3 and rule 4 when token `Y` is read. In this state, it's reading an `X`, which may be an `a` or a `b`. They show the two rules that might be reduced. The dot shows where in the rule you are before receiving the next token. This corresponds to the pointer in the bison grammar. For reduce conflicts, the pointer is always at the end of the rule. In a conflict, the rule not used is shown in brackets; in this case bison chose to reduce rule 3, since it resolves reduce/reduce conflicts by reducing the rule that appears earlier in the grammar.

The rules may have tokens or nonterminals in them. The following ambiguous grammar:

```
start:      a Z
     |      b Z;
a:   X y;
b:   X y;
y:   Y;
```

produces a parser with this state:

```
state 6

    3 a: X y .
    4 b: X y .

    Z          reduce using rule 3 (a)
    Z          [reduce using rule 4 (b)]
    $default  reduce using rule 3 (a)
```

In this state, the parser has already reduced a `Y` to a `y`, but the `y` could complete either an `a` or a `b`. Transitions on nonterminals can lead to reduce/reduce conflicts just as tokens can. It’s easy to tell the difference if you use uppercase token names, as we have.

The rules that conflict do not have to be identical. This grammar:

```
start:     A B x Z
     |     y Z;
x:    C;
y:    A B C;
```

when processed by bison, produces a grammar containing this state:

```
state 7

    3 x: C .
    4 y: A B C .

    Z          reduce using rule 3 (x)
    Z          [reduce using rule 4 (y)]
    $default  reduce using rule 3 (x)
```

In state 7, the parser has already accepted `A B C`. Rule `x` has only `C` in it, because in the `start` rule from which `x` is called, `A B` is accepted before reaching `x`. The `C` could complete either an `x` or a `y`. Bison again resolves the conflict by reducing the earlier rule in the grammar, in this case rule 3.

Shift/Reduce Conflicts

Identifying a shift/reduce conflict is a little harder. To identify the conflict, we will do the following:

- Find the shift/reduce error in *`name`*`.output`.
- Identify the reduce rule.
- Identify the relevant shift rule(s).
- See what state the reduce rule reduces to.
- Deduce the token stream that will produce the conflict.

This grammar contains a shift/reduce conflict:

```
    start:     x
         |     y R;
    x:    A R;
    y:    A;
```

Bison produces this complaint:

```
state 1

    3 x: A . R
    4 y: A .

    R  shift, and go to state 5

    R  [reduce using rule 4 (y)]
```

State 1 has a shift/reduce conflict between shifting token `R`, which moves to state 5, and reducing rule 4 when it reads an `R`. Rule 4 is rule `y`, as shown in this line:

```
    4 y: A .
```

You can find the reduce rule in a shift/reduce conflict the same way you find both rules in a reduce/reduce conflict. The reduction number is listed in the `reduce using` line. In the previous case, the rule with the shift conflict is the only rule left in the state:

```
3 x: A . R
```

The parser is in rule `x`, having read `A` and about to accept `R`. The shift conflict rule was easy to find in this case, because it is the only rule left, and it shows that the next token is `R`. Bison resolves shift/reduce conflicts in favor of the shift, so in this case if it receives an `R`, it shifts to state 5.

The next thing showing may be a rule instead of a token:

```
start:      x1
     |      x2
     |      y R;
x1:   A R;
x2:   A z;
y:    A;
z:    R;
```

Bison reports several conflicts, including this one:

```
state 1

    4 x1: A ↑ R
    5 x2: A ↑ z
    6 y: A ↑

    R  shift, and go to state 6

    R  [reduce using rule 6 (y)]

    z  go to state 7
```

In the previous example, the reduction rule is as follows:

```
6 y: A ↑
```

so that leaves two candidates for the shift conflict:

```
    4 x1: A ↑ R
    5 x2: A ↑ z
```

Rule `x1` uses the next token, `R`, so you know it is part of the shift conflict, but rule `x2` shows the next symbol (not token). You have to look at the rule for `z` to find out whether it starts with an `R`. In this case it does, so there is a conflict for an `A` followed by an `R`: it could be an `x1`, an `x2` that includes a `z`, or a `y` followed by an `R`.

There could be more rules in a conflicting state, and they may not all accept an `R`. Consider this extended version of the grammar:

```
    start:      x1
         |      x2
         |      x3
         |      y R;
   x1:    A R;
   x2:    A z1;
   x3:    A z2
   y:     A;
   z1:    R;
   z2:    S;
```

Bison produces a listing with this state:

```
state 1

    5 x1: A ↑ R
    6 x2: A ↑ z1
    7 x3: A ↑ z2
    8 y: A ↑

    R  shift, and go to state 7
    S  shift, and go to state 8

    R  [reduce using rule 8 (y)]

    z1  go to state 9
    z2  go to state 10
```

The conflict is between shifting to state 7 and reducing rule 8. The reduce problem, rule 8, is the rule for `y`. The rule for `x1` has a shift problem, because the next token after the dot is `R`. It is not immediately obvious whether `x2` or `x3` caused conflicts, because they show rules `z1` and `z2` following the dots. When you look at rules `z1` and `z2`, you find that `z1` contains an `R` next and `z2` contains an `S` next, so `x2` that uses `z1` is part of the shift conflict and `x3` is not.

In each of our last two shift/reduce conflict examples, can you also see a reduce/reduce conflict? Run bison, and look in *`name`*`.output` to check your answer.

Review of Conflicts in name.output

We'll review the relationship among our pointer model, conflicts, and *`name`*`.output`. First, here is a reduce/reduce conflict:

```
start:      A B x Z
     |      y Z;
x:    C;
y:    A B C;
```

The bison listing contains the following:

```
state 7

    3 x: C ↑
    4 y: A B C ↑

    Z          reduce using rule 3 (x)
    Z          [reduce using rule 4 (y)]
    $default   reduce using rule 3 (x)
```

There is a conflict because if the next token is `Z`, bison wants to reduce rules 3 and 4, which are the rules for both `x` and `y`. Or using our pointer model, there are two pointers, and both are reducing:

```
start:       A B x z
     |       y Z;
```

```
      x:    c ↑ ;
      y:    A B C ↑ ;
```

Here is a shift/reduce conflict example:

```
      start:     x
           |     y R;
      x:         A R;
      y:         A;
```

Bison reports this conflict:

```
   state 1

      3 x: A ↑ R
      4 y: A ↑

      R  shift, and go to state 5

      R  [reduce using rule 4 (y)]
```

If the next token is `R`, bison wants both to reduce the rule for `y` and to shift an `R` in the rule for `x`, causing a conflict. Or there are two pointers, and one is reducing:

```
      start:    x
           |    y R;
      x:    A ↑ R;
      y:    A ↑ ;
```

Common Examples of Conflicts

The three most common situations that produce shift/reduce conflicts are expression grammars, if/then/else, and nested lists of items. After we see how to identify these three situations, we'll look at ways to get rid of the conflicts.

Expression Grammars

Our first example is adapted from the original 1975 Unix yacc manual.

```
    expr: TERMINAL
        | expr '-' expr ;
```

The state with a conflict is as follows:

```
state 5

    2 expr: expr ↑ '-' expr
    2     | expr '-' expr ↑

    '-'  shift, and go to state 4

    '-'       [reduce using rule 2 (expr)]
    $default  reduce using rule 2 (expr)
```

Bison tells us that there is a shift/reduce conflict when it reads the minus token. We can add our pointers to get the following:

```
    expr: expr ↑ - expr ;
    expr: expr - expr ↑ ;
```

These are in the same rule, not even different alternatives with the same LHS. You can have a state where your pointers can be in two different places in the same rule, because the grammar is recursive. (In fact, all of the examples in this section are recursive. Most tricky bison problems turn out to involve recursive rules.)

After accepting two `expr` s and `-` , the pointer is at the end of rule `expr` , as shown in the second line of the earlier pointer example. But `expr - expr` is also an `expr` , so the pointer can also be just after the first `expr` , as shown in the first line of the earlier example. If the next token is not `-` , then the pointer in the first line disappears because it wants `-` next, so you are back to one pointer. But if the next token is `-` , then the second line wants to reduce, and the first line wants to shift, causing a conflict.

To solve this conflict, look at *`name`*`.output` , shown earlier, to find the source of the conflict. Get rid of irrelevant rules in the state (there are not any in this tiny example), and you get the two pointers we just discussed. It becomes clear that the problem is as follows:

```
        expr - expr - expr
```

The middle `expr` might be the second `expr` of an `expr - expr`, in which case the input is interpreted as follows:

```
        (expr - expr) - expr
```

which is left associative, or might be the first `expr`, in which case the input is interpreted as follows:

```
        expr - (expr - expr)
```

which is right associative. After reading `expr - expr`, the parser could reduce if using left associativity or shift using right associativity. If not instructed to prefer one or the other, this ambiguity causes a shift/reduce conflict, which bison resolves by choosing the shift. Figure 7-1 shows the two possible parses.

Later in this chapter, we cover the ways to handle this kind of conflict.

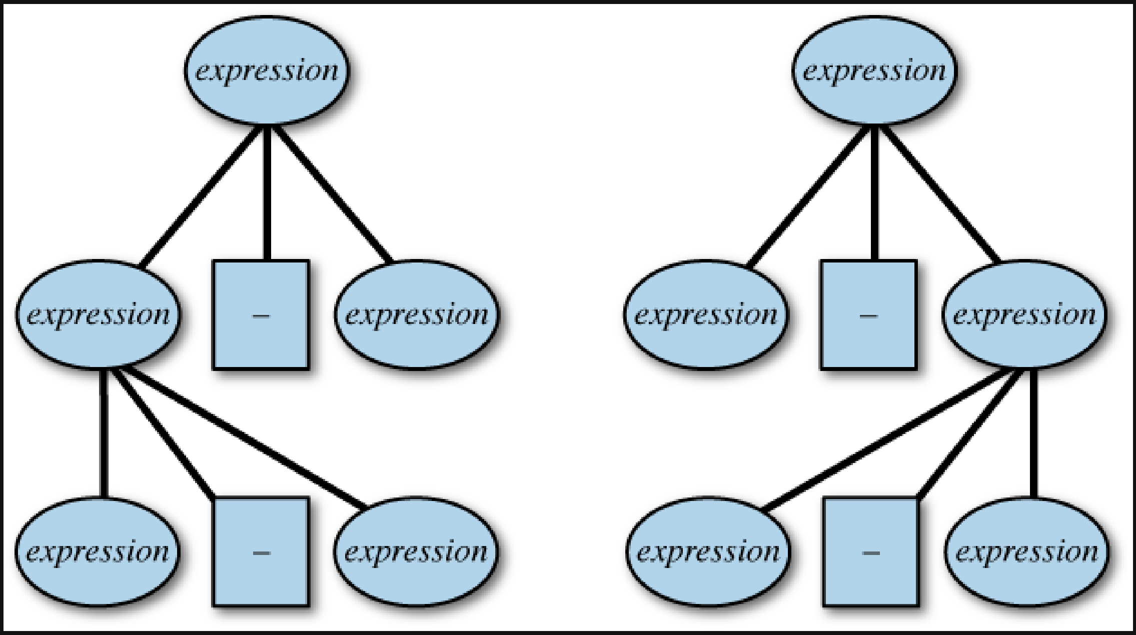

Figure 7-1. Two parses of expr - expr - expr

IF/THEN/ELSE

Our next example is also from the Unix yacc manual. Again, we have added a terminal symbol for completeness:

```
stmt:  IF '(' cond ')' stmt
    |  IF '(' cond ')' stmt ELSE stmt
    |  TERMINAL;
cond:  TERMINAL;
```

Bison complains:

```
State 9 conflicts: 1 shift/reduce
  ...

state 9

    1 stmt: IF '(' cond ')' stmt ↑
    2     | IF '(' cond ')' stmt ↑ ELSE stmt

    ELSE  shift, and go to state 10

    ELSE      [reduce using rule 1 (stmt)]
    $default  reduce using rule 1 (stmt)
```

In terms of pointers this is as follows:

```
stmt: IF ( cond ) stmt ↑ ;
stmt: IF ( cond ) stmt ↑ ELSE stmt ;
```

The first line is the reduce part of the conflict, and the second is the shift part. This time they are different rules with the same LHS. To figure out what is going wrong, we check to see where the first line reduces to. It has to be a call to `stmt`, followed by an `ELSE`. There is only one place where that happens:

`stmt: IF ( cond ) stmt` *<return to here>* `ELSE stmt ;`

After the reduction, the pointer returns to the same spot where it is for the shift part of the conflict. This problem is very similar to the one with `expr - expr - expr` in the previous example. And using similar logic, in order to reduce `IF ( cond ) stmt` into `stmt` and end up here:

`stmt: IF ( cond ) stmt` *<here>* `ELSE stmt ;`

you have to have this token stream:

```
IF ( cond ) IF ( cond ) stmt ELSE
```

Again, do you want to group it like this:

```
IF ( cond ) { IF ( cond ) stmt } ELSE stmt
```

or like this?

```
IF ( cond ) { IF ( cond ) stmt ELSE stmt }
```

The next section explains what to do about this kind of conflict.

Nested List Grammar

Our final example is a simple version of a problem that novice bison programmers often encounter:

```
start:            outerList Z ;
outerList:  /* empty */
      |     outerList outerListItem ;

outerListItem:    innerList ;

innerList:  /* empty */
      |     innerList innerListItem ;

innerListItem:    I ;
```

Bison reports this conflict:

```
state 2

    1 start: outerList ↑ Z
    3 outerList: outerList ↑ outerListItem

    Z  shift, and go to state 4

    Z         [reduce using rule 5 (innerList)]
    $default  reduce using rule 5 (innerList)

    outerListItem  go to state 5
    innerList      go to state 6
```

Once again we can analyze the problem step by step. The reduce rule is the empty alternative of `innerList`. That leaves two candidates for the shift problem. Rule `start` is one, because it explicitly takes `Z` as the next token. The nonempty alternative of `outerList` might be a candidate, if it takes `Z` next. We see that `outerList` includes an `outerListItem`, which is an `innerList`. In this situation, `innerList` can't include an `innerListItem`, because that includes an `I`, and this conflict occurs only when the next token is a `Z`. But an `innerList` can be empty, so the `outerListItem` involves no tokens, so we might actually be at the end of the `outerList` as well, since as the first line in the conflict report told us, an `outerList` can be followed by a `Z`.

This all boils down to this state: We have just finished an `innerList`, possibly empty, or an `outerList`, possibly empty. How can it not know which list it has just finished? Look at the two list expressions. They can both be empty, and the inner one sits in the outer one without any token to say it is starting or finishing the inner loop. Assume the input stream consists solely of a `Z`. Is it an empty `outerList`, or is it an `outerList` with one item, an empty `innerList`? That's ambiguous.

The problem with this grammar is that it is redundant. It has to have a loop within a loop, with nothing to separate them. Since this grammar actually accepts a possibly empty list of `I` s followed by a `Z`, it can easily be rewritten using only one recursive rule:

```
    start:              outerList Z ;
    outerList:    /* empty */
          |       outerList outerListItem ;
    outerListItem:      I ;
```

But rewriting it this way is rarely the right thing to do. More likely there really are supposed to be two nested lists, but you forgot to include punctuation in `outerListItem` to delimit the inner from the outer loop. We offer some suggestions later in this chapter.

How Do You Fix the Conflict?

The rest of this chapter describes what to do with a conflict once you've figured out what it is. We'll discuss how to fix classes of conflicts that have commonly caused trouble for bison users.

When trying to resolve conflicts, start with the rules that are involved in the most conflicts, and work your way down the list. More often than not, when you resolve the major conflicts, many of the minor ones will go away, too.

If you're writing a parser for a language you're inventing, conflicts in the parser often indicate ambiguities or inconsistencies in the language's definition. First figure out what's causing the conflict, and then decide whether the language is OK and you just need to adjust the grammar to describe the language correctly or whether the language is ambiguous, in which case you may want to change both the language and the grammar to be unambiguous. Languages that bison has trouble parsing are often hard for people to parse in their heads; you'll often end up with a better language design if you change your language to remove the conflicts.

IF/THEN/ELSE (Shift/Reduce)

We saw this conflict earlier in this chapter. Here we describe what to do with the shift/reduce conflict once you've tracked it down. It turns out that the default way that bison resolves this particular conflict is usually what you want it to do anyway. How do you know it's doing what you want it to do? Your choices are to (1) be good enough at reading bison descriptions, (2) be masochistic enough to decode the `*name*.output` listing, or (3) test the generated code to death. Once you've verified that you're getting what you want, you ought to make bison quit complaining. Conflict warnings may confuse or annoy anyone trying to maintain your code, and if there are other conflicts in the grammar that indicate genuine errors, it's hard to tell the real problems from the false alarms.

The standard way to resolve this conflict is to have separate rules for matched and unmatched `IF/THEN` statements and to rewrite the grammar this way:

```
stmt:       matched
      |     unmatched
      ;
matched:    other_stmt
      |     IF expr THEN matched ELSE matched
      ;
unmatched:  IF expr THEN stmt
      |     IF expr THEN matched ELSE unmatched
      ;
other_stmt: /* rules for other kinds of statement */  ...
```

The nonterminal `other_stmt` represents all of the other possible statements in the language.

It's also possible to use explicit precedence to tell bison which way to resolve the conflict and to keep it from issuing a warning. If your language uses a `THEN` keyword (as Pascal does), you can do this:

```
%nonassoc THEN
%nonassoc ELSE

%%

stmt:     IF expr THEN stmt
      |   IF expr stmt ELSE stmt
      ;
```

In languages that don't have a `THEN` keyword, you can use a fake token and `%prec` to accomplish the same result:

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%%

stmt:       IF expr stmt          %prec LOWER_THAN_ELSE ;
       |    IF expr stmt ELSE stmt;
```

The shift/reduce conflict here is a conflict between shifting an `ELSE` token and reducing a `stmt` rule. You need to assign a precedence to the token (`%nonassoc ELSE`) and to the rule, with `%nonassoc THEN` or

`%nonassoc LOWER_THAN_ELSE` and `%prec LOWER_THAN_ELSE`. The precedence of the token to shift must be higher than the precedence of the rule to reduce, so `%nonassoc ELSE` must come after `%nonassoc THEN` or `%nonassoc LOWER_THAN_ELSE`. It makes no difference for this application if you use `%nonassoc`, `%left`, or `%right`, since there's no situation with a conflict that involves shifting a token and reducing a rule containing the same token.

The goal here is to hide a conflict you know about and understand, *not* to hide any others. When you're trying to mute bison's warnings about other shift/reduce conflicts, the further you get from the previous example, the more careful you should be. Use `%nonassoc`, so if you accidentally do add other rules that create such a conflict, bison will still report it. Other shift/reduce conflicts may be amenable to a simple change in the bison description. And, as we mentioned, *any* conflict can be fixed by changing the language. For example, the `IF/THEN/ELSE` conflict can be eliminated by insisting on `BEGIN-END` or braces around the `stmt`.

What would happen if you swapped the precedence of the token to shift and the rule to reduce? The normal `IF-ELSE` handling makes the following two equivalent:

```
    if expr if expr stmt else stmt
    if expr { if expr stmt else stmt }
```

It seems only fair that swapping the precedence would make the following two equivalent, right?

```
    if expr if expr stmt else stmt
    if expr { if expr stmt } else stmt
```

Nope. That's not what it does. Having higher precedence on the shift (normal `IF-ELSE`) makes it always shift the `ELSE`. Swapping the precedence makes it *never* shift the `ELSE`, so your `IF-ELSE` can no longer have an else.

Normal `IF-ELSE` processing associates the `ELSE` with the most recent `IF`. Suppose you want it some other way. One possibility is that you allow only one `ELSE` with a sequence of `IF`s, and the `ELSE` is associated with the first `IF`. This would require a two-level statement definition, as follows:

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%%

stmt:       IF expr stmt2 %prec LOWER_THAN_ELSE
      |     IF expr stmt2 ELSE stmt;

stmt2:  IF expr stmt2;
```

But don't do that, since a language is extremely counterintuitive.

Loop Within a Loop (Shift/Reduce)

This conflict occurs when the grammar has two nested list-creating loops, with no punctuation to say where the boundaries between entries in the outer list are.

```
start:           outerList Z ;
outerList: /* empty */
      |    outerList outerListItem ;

outerListItem:   innerList ;
innerList: /* empty */
      |    innerList innerListItem ;

innerListItem:   I ;
```

Assuming that's really what you want, the resolution of this conflict depends on whether you want repetitions to be treated as one outer loop and many inner loops or as many outer loops of one inner loop each. The difference is whether the code associated with `outerListItem` gets executed once for each repetition or once for each set of repetitions. If it makes no difference, choose one or the other arbitrarily. If you want many outer loops, remove the inner loop:

```
start:              outerList Z ;
```

```
outerList:    /* empty */
      |       outerList innerListItem ;

innerListItem:      I ;
```

If you want many inner loops, remove the outer loop:

```
start:              innerList Z ;

innerList:    /* empty */
      |       innerList innerListItem ;

innerListItem:  I ;
```

In practice, it's pretty rare to have a pair of nested lists with no punctuation. It's confusing to bison, and it's confusing to us humans, too. If the outer list is something like a list of statements in a programming language, if you change the language and put a semicolon after each `outerListItem`, the conflicts go away:

```
start:           outerList Z ;
outerList: /* empty */
      |    outerList outerListItem ';' ;

outerListItem:   innerList ;
innerList: /* empty */
      |    innerList innerListItem ;

innerListItem:   I ;
```

Expression Precedence (Shift/Reduce)

```
expr:        expr  '+'  expr
      |      expr  '-'  expr
      |      expr  '*'  expr
      |      ...
        ;
```

If you describe an expression grammar but forget to define the precedence with `%left` and `%right`, you get a truckload of shift/reduce conflicts. Assigning precedence to all of the operators should resolve the conflicts. Keep in mind that if you use any of the operators in other ways, for example, using a `-` to indicate a range of values, the precedence can also mask conflicts in the other contexts.

Limited Lookahead (Shift/Reduce or Reduce/Reduce)

Most shift/reduce conflicts are because of bison's limited lookahead. That is, a parser that could look further ahead would not have a conflict. For example:

```
statement: command optional_keyword '('  identifier_list ')'
      ;

optional_keyword: /* blank */
      |      '(' keyword ')'
      ;
```

The example describes a command line that starts with a required command, ends with a required identifier list in parentheses, and has in the middle an optional keyword in parentheses. Bison gets a shift/reduce conflict with this when it gets to the first parenthesis in the input stream, because it can't tell whether it is part of the optional keyword or the identifier list. In the first case, the parser would shift the parenthesis within the `optional_keyword` rule, and in the second, it would reduce an empty `optional_keyword` and move on to the identifier list. If a bison parser could look further ahead, it could tell the difference between the two. But a parser using the regular bison parsing algorithm can't.

The default is for bison to choose the shift, which means it always assumes the optional keyword is there. (You can't really call it optional in that case.) If you apply precedence, you could get the conflict to resolve in favor of the reduction, which would mean you could never have the optional keyword.

We can *flatten* the description, expanding the `optional_keyword` rule where it occurs in `statement`:

```
statement:     command '(' keyword ')' '(' identifier_list  ')'
       |  command '('  identifier_list ')'
```

```
                ;
```

By flattening the list, we allow the parser to scan ahead with multiple possible pointers until it sees a keyword or identifier, at which point it can tell which rule to use.

Flattening is a practical solution in this example, but when more rules are involved, it rapidly becomes impractical because of the exponential expansion of the bison description. You may run into a shift/reduce conflict from limited lookahead for which your only practical solution is to change the language or use a GLR parser.

It's also possible to get a reduce/reduce conflict because of limited lookahead. One way is to have an overlap of alternatives:

```
statement:  command_type_1 ':' '[' ...
        |      command_type_2 ':' '(' ...

command_type_1:  CMD_1  | CMD_2 | CMD_COMMON  ;

command_type_2:  CMD_A  | CMD_B | CMD_COMMON  ;
```

If the input includes `CMD_COMMON`, the parser can't tell whether it's parsing a `command_type_1` or `command_type_2` until it sees the bracket or parenthesis, but that's two tokens ahead. The solution for this is flattening, as we did earlier, or making the alternatives disjoint, as described in the following section.

You can also get a reduce/reduce conflict from limited lookahead because actions in the middle of a rule are really anonymous rules that must be reduced:

```
statement: command_list { <action for '['form> }':' '[' ...
        |    command_list { <action for '('form> }':' '(' ...
```

This is already flattened, so there's nothing you can do to get it to work without using a GLR parser. It simply needs a two-token lookahead, and LALR(1) parsers don't have that. Unless you're doing some sort of exotic communication between the parser and lexer, you can just move the action over:

```
statement:  command_list ':' '[' { <action for '[' form> } ...
       |    command_list ':' '(' { <action for '(' form> } ...
```

Overlap of Alternatives (Reduce/Reduce)

In this case, you have two alternative rules with the same LHS, and the inputs accepted by them overlap partially. The easiest way to make this work in a regular bison parser is to make the two input sets disjoint. For example:

```
person:     girls
       |    boys
       ;

girls:      ALICE
       |    BETTY
       |    CHRIS
       |    DARRYL
       ;

boys:       ALLEN
       |    BOB
       |    CHRIS
       |    DARRYL
       ;
```

You will get a reduce/reduce conflict on `CHRIS` and `DARRYL` because bison can't tell whether they're intended to be `girls` or `boys`. There are several ways to resolve the conflict. One is as follows:

```
person:    girls  | boys | either;

girls:     ALICE
      |    BETTY
      ;

boys:      ALLEN
      |    BOB
      ;

either:    CHRIS
```

```
      |     DARRYL
      ;
```

But what if these lists were really long or were complex rules rather than just lists of keywords? What would you do if you wanted to minimize duplication and `girls` and `boys` were referenced many other places in the bison description? Here's one possibility:

```
person:     just_girls
      |     just_boys
      |     either
      ;

girls:      just_girls
      |     either
      ;

boys:       just_boys
      |     either
      ;

just_girls: ALICE
      |     BETTY
      ;

just_boys:  ALLEN
      |     BOB
      ;

either:     CHRIS
      |     DARRYL
      ;
```

All references to `boys | girls` have to be fixed. GLR doesn't help much here since the original grammar is ambiguous, so you'd still have to deal with the ambiguity.

But what if it's impractical to make the alternatives disjoint? If you just can't figure out a clean way to break up the overlap, then you'll have to leave the reduce/reduce conflict, use a GLR parser, and deal explicitly with the ambiguity using the techniques discussed in Chapter 9.

If you don’t use a GLR parser, bison will use its default disambiguating rule for reduce/reduce, which is to choose the first definition in the bison description. So in the first `girls | boys` example earlier, `CHRIS` and `DARRYL` would always be `girls`. Swap the positions of the `boys` and `girls` lists, and `CHRIS` and `DARRYL` are always `boys`. You’ll still get the reduce/reduce warning, and bison will make the alternatives disjoint for you, exactly what you were trying to avoid.

Summary

Ambiguities and conflicts within the bison grammar are just one type of coding error, one that is problematic to find and correct. This chapter has presented some techniques for correcting these errors. In the chapter that follows, we will look at other sources of errors.

Our goal in this chapter has been for you understand the problem at a high enough level that you can fix it. To review how to get to that point:

- Find the shift/reduce error in *`name`*`.output`.
- Identify the reduce rule.
- Identify the relevant shift rule(s).
- See where the reduce rule will reduce back to.
- With this much information, you should be able to identify the token stream leading up to the conflict.

Seeing where the reduce rule reduces to is typically straightforward, as we have shown. Sometimes a grammar is so complicated that it is not practical to use our “hunt-around” method, and you will need to learn the detailed operation of the state machine to find the states to which you reduce.

Exercises

1. All reduce/reduce conflicts and many shift/reduce conflicts are caused by ambiguous grammars. Beyond the fact that bison doesn’t like them, why are ambiguous grammars usually a bad idea?
2. Find a grammar for a substantial programming language like C, C++, or Fortran, and run it through bison. Does the grammar have conflicts? (Nearly all of them do.) Go through the *`name`*`.output`

listing, and determine what causes the conflicts. How hard would they be to fix?

3. After doing the previous exercise, opine about why languages are usually defined and implemented with ambiguous grammars.