The Dangling Else

Consider the following grammar:

```
%token IF EXP ELSE XX
%%
stmts     : stmts stmt ';'
          | stmt ';'
          ;

stmt      : IF '(' EXP ')' stmt ELSE stmt
          | IF '(' EXP ')' stmt
          | XX
          ;
%%
```

In the above grammars in this file tokens that are all uppercase are terminals that match the same string in lower case. For instance in the above grammar: **if (exp) xx else xx ;** is a legal statement. Unfortunately the grammar has a problem when compiled in bison:

```
+ bison -v -t -d check.y
check.y contains 1 shift/reduce conflict.
```

Checking the `.output` file shows:

```
State 11 contains 1 shift/reduce conflict.

Grammar
rule 1    stmts -> stmts stmt ';'
rule 2    stmts -> stmt ';'
rule 3    stmt -> IF '(' EXP ')' stmt ELSE stmt
rule 4    stmt -> IF '(' EXP ')' stmt
rule 5    stmt -> XX

state 11

    stmt  ->  IF '(' EXP ')' stmt . ELSE stmt   (rule 3)
    stmt  ->  IF '(' EXP ')' stmt .   (rule 4)

    ELSE        shift, and go to state 12

    ELSE        [reduce using rule 4 (stmt)]
    $default    reduce using rule 4 (stmt)
```

It is clear that a shift would move the **ELSE** using rule 3 but where did the problem with the reduce come from? The problem is that rule 3 also shows that a **stmt** stmt can be followed by **ELSE** because the then-part is a stmt and can be followed by **ELSE**. But rule 4 is a **stmt** therefore rule 4 can be followed by an **ELSE**! So if an **ELSE** token is seen then it should reduce AND it should shift it. Hence an shift/reduce error.

The fix is this we need to make sure there is no ambiguity in our grammar that would suggest that both parses that shift and reduce are viable. The classic ambiguous case is:

```
if ( exp ) if ( exp ) xx else xx ;
```

does that mean (case 1) as indicated by the brackets:

```
if ( exp ) [ if ( exp ) xx else xx ];
```

or does that mean (case 2):

```
if ( exp ) [ if ( exp ) xx ] else xx ;
```

? The fix is to be sure that the else is either always with the nearest if (case 1) or farthest if (case 2). Classically the fix is to do case 1.

We enforce this by using a grammar like this:

```
%token IF EXP ELSE XX
%%
stmts     : stmts stmt ';'
          | stmt ';'
          ;

stmt      : matched
          | unmatched
          ;

matched   : IF '(' EXP ')' matched ELSE matched
          | XX
          ;

unmatched : IF '(' EXP ')' matched
          | IF '(' EXP ')' unmatched
          | IF '(' EXP ')' matched ELSE unmatched
          ;
%%
```

Statements that match **matched** are like:

```
if ( exp )
  xx
else
  xx

if ( exp )
  xx
else
  if ( exp ) xx else xx

if ( exp )
   if ( exp )
     xx
   else
     if ( exp ) xx else xx
else
   if ( exp ) xx else xx
```

Statements that match **unmatched** are like:

```
if ( exp )
  xx

if ( exp )
  xx
else
  if ( exp ) xx

if ( exp )
  if ( exp ) xx else xx
else
  if ( exp ) xx
```

Most importantly is what is left out of the grammar above! There are 4 **posssible** combinations of if-else:

```
IF '(' EXP ')' matched ELSE matched
IF '(' EXP ')' matched ELSE unmatched
IF '(' EXP ')' unmatched ELSE matched
IF '(' EXP ')' unmatched ELSE unmatched
```

However, **the last two do not occur** in the grammar and are not allowed. It is this that prohibits the occurrence of an else that is NOT matched to the nearest if! That is the grammar simply does not allow an unmatched if to come between an **else** and its **if**. This removes the ambiguity of two parses and leaves only the one we want.

Now that we understand what is going on, how can we add a **while** statement? **While** statements are of the form:

```
WHILE '(' EXP ')' stmt
```

but stmt can be either **matched** or **unmatched**. Consider the expression:

```
if ( exp ) while ( exp ) if ( exp ) xx else xx
```

As before we have two cases (parsings) indicated by the brackets:

```
if ( exp ) while ( exp ) [ if ( exp ) xx else xx ]
if ( exp ) [ while ( exp ) if ( exp ) xx ] else xx
```

We need to write our grammar to preclude the second of these two parses. That is a **while** with an unmatched **if** must not occur between an **if** and its **else**. This removes the ambiguity and allows the grammar to parse only the first meaning (the one we want).

This suggests that there are two kinds of **while**. A **while** that has an **unmatched** after it and a **while** that has a **matched** after it. The **while** with the **unmatched** must not be allowed to come before an else in the same way that the **if** without the **else** must not.

I have all but told you the answer but most importantly the reasoning behind it. I must leave something for you to figure out so I will stop here except to point out that the grammar I gave above for solving the dangling if could be slightly shorter by replacing

```
unmatched : IF '(' EXP ')' matched
          | IF '(' EXP ')' unmatched
```

with

```
unmatched : IF '(' EXP ')' stmt
```

Do not hesitate to send me email questions with the timestamp of you submit or to drop by and ask questions. Appointments (send email) work best since I have lots of meetings.

Good luck,

Robert Heckendorn Up One Level Last updated: Mar 1, 2005 22:29