Notes on shift-reduce parsing and resolving conflicts in ocamlyacc - Supplementary notes for lecture 10, 2/12/10

CS 421, Prof. Kamin

In this note, we go through four grammars and show how to resolve conflicts reported by ocamlyacc. For each grammar, we do the following:

1. Run ocamlyacc. To simplify the process, we give {0} as the semantic action for each rule. Each of the grammars has a conflict. (In every case, it is a shift/reduce conflict; we have no reduce/reduce conflicts.)

2. We look at the ocamlyacc verbose ("ocamlyacc -v") output, to see where the conflict is.

3. Based on that output, we attempt to construct an example that shows the conflict. A conflict means that an input string leads to a stack/lookahead configuration in which the parser cannot make a clear choice between shifting and reducing. If the grammar is ambiguous, we will just show one input that leads to such a configuration, and which therefore has multiple parses. If the grammar is unambiguous, we will show two inputs that lead to the same configuration, but differ after the lookahead symbol.

4. Using our understanding of the source of the conflict, we will resolve it in one of the three ways described in class: modify the grammar; use precedence and associativity declarations; or do nothing. We will show first that this resolves the conflict illustrated by the example we obtained in step 3; running ocamlyacc proves that we did in fact eliminate all conflicts.

Grammar 1

Stmt $\rightarrow$ MethodCall | ArrayAsgn
MethodCall $\rightarrow$ Target ( ) ;
Target $\rightarrow$ id | id . id
ArrayAsgn $\rightarrow$ id (int) = int ;

This grammar is unambiguous. ocamlyacc reports a shift/reduce conflict.

(This is the exact input we presented to ocamlyacc:

```
%token oparen cparen id semic dot int equal
%start STMT
%type <int> STMT
%%
STMT : METHODCALL {0} | ARRAYASGN {0}
```

```
METHODCALL : TARGET oparen cparen semic  {0}
TARGET : id  {0}  | id dot id {0}
ARRAYASGN : id oparen int cparen equal int semic {0}
```

We will not show this for the other grammars we study. Note that we used semantic action {0} for every production, because we were only interested in seeing the parsing conflicts.)

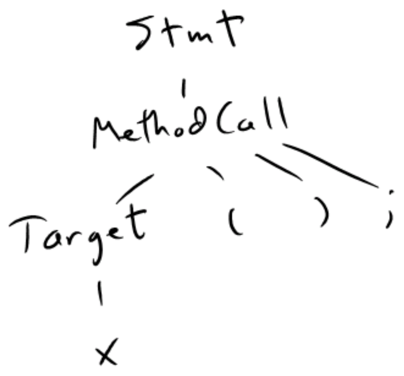
Specifically, the verbose output from ocamlyacc includes the following. (The entire output file, example1.output, is about 160 lines long; the interesting part starts where it gives the conflict, which happens to be at line 43, and ends when the next "state" is shown):

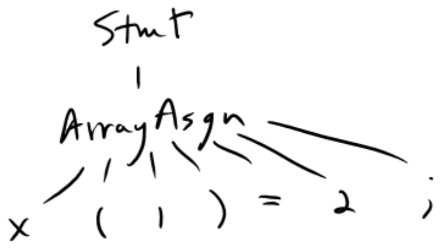
```
3: shift/reduce conflict (shift 8, reduce 4) on oparen
state 3
     TARGET : id .  (4)
     TARGET : id . dot id  (5)
     ARRAYASGN : id . oparen int cparen equal int semic  (6)

     oparen  shift 8
     dot  shift
```

This says: If there is an id on the stack and the lookahead symbol is '(', the parser cannot choose between shifting and reducing. We know the lookahead symbol is '(' because it says that on the first line. We know the top symbol on the stack is id because the three productions have a period after an id. Most importantly, in the line "TARGET : id .", the period is *at the end* of the production, while in the other two lines it is in the middle. This is where the shift/reduce conflict comes in: there can be an input where, with id on the stack and '(' in the input, the correct action is to reduce Target $\rightarrow$ id, and another input that leads to the same situation but where shift is the correct action.

Before proceeding, see if you can look at the grammar and find two such inputs.

Looking back at the grammar, it is easy to see where an id on the stack should be reduced using Target$\rightarrow$ id. Target is used in MethodCall, and is followed by a '('. So we can consider a simple sentence like "x();". It is also easy to see where this same configuration requires a shift action: in the rule for ArrayAsgn, an id is followed by '('; the '(' should be shifted so that, eventually, this production can be used. This suggests a sentence like "x(1) = 2;". Let's look at the parse trees for these two inputs:



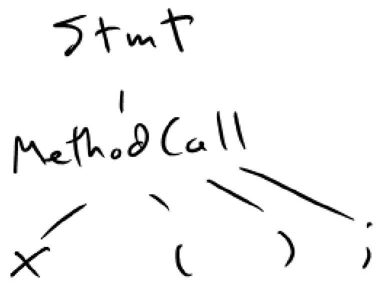


We can easily see where the conflict comes in: on the left, the x needs to be reduced to put Target on the stack, on the right, the '(' should be shifted. (Keep in mind that reductions always consume the *top* of the stack. On the left, we can't shift '(' and then *later* do the Target → id reduction. It has to be done as soon as the '(' is the lookahead symbol.)

We can resolve the conflict here by observing that the Target → id production is what's causing the problem. Suppose we eliminate it and just put the id directly in the MethodCall production:

Stmt → MethodCall | ArrayAsgn
MethodCall → Target ( ) ; | id () ;
Target → id . id
ArrayAsgn → id (int) = int ;

Now, the parse tree for "x();" is



and there is no conflict: with x on the stack and '(' as the lookahead symbol, the correct action in both cases is to shift. (Note that the other kind of target, id . id, does not present a problem; a period in the input should always be shifted; while, if the stack contains "id . id" and the lookahead is '(', the correct action is reduce Target → id . id.) Running ocamlyacc on the modified grammar shows that we have eliminated the conflict.

To summarize, our strategy for eliminating the conflict in this case was to remove the “extraneous” non-terminal in the MethodCall production, by moving one of the right-hand sides of Target to the MethodCall production.

Grammar 2

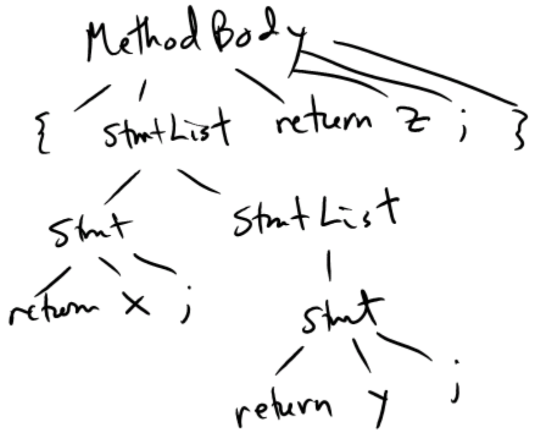
MethodBody $\rightarrow$ { StmtList return id ; }
StmtList $\rightarrow$ Stmt StmtList | Stmt
Stmt $\rightarrow$ return id ; | id = id + id ;

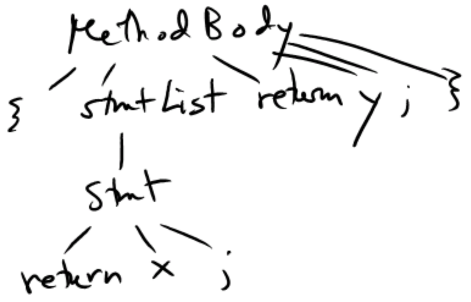
Again, this is not ambiguous. ocamlyacc reports a conflict, to wit:

```
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
```

This says: When Stmt is on the stack (more precisely, a parse tree whose root is labeled Stmt), and the ‘return’ keyword is in the input, the parser can either shift or reduce by StmtList $\rightarrow$ Stmt. Again, we know this is the reduction because in the fourth line, the period appears at the end of the production.

Again, let us look for sentences that illustrate the conflict. We want to look for an example of using each of the two productions for StmtList. This is not hard: The StmtList $\rightarrow$ Stmt Stmtlist production is used whenever the method body has more than one statement (not counting the ending “return” statement); the StmtList $\rightarrow$ Stmt is always used, even if the method body has just one statement. So let’s try those two possibilities: “{return x; return y; return z; }” and “{return x; return y;}”. Here are their parse trees:



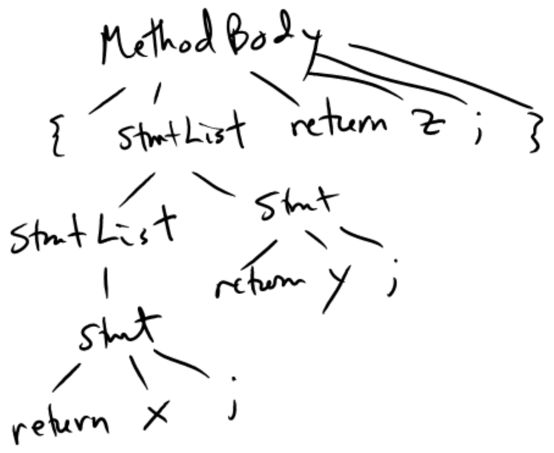


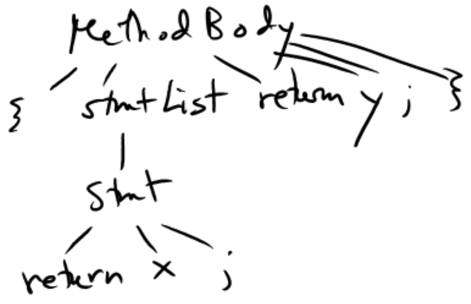
Here is the problem: In both cases, after shifting "return x ;", we reduce to Stmt. Now the stack contains Stmt and the input is "return". However, in the parse tree on the left, we need to continue shifting, so that we can eventually reduce "return y ;" to Stmt, then to StmtList and finally use StmtList → Stmt Stmtlist; in the parse tree on the right, we should instead immediately reduce using StmtList → Stmt.

In this case, the conflict can be resolved simply by using left-recursion in the StmtList productions instead of right-recursion:

StmtList → StmtList Stmt | Stmt

Now the two sentences have these parse trees:





The tree on the right is unchanged. But now consider the tree on the left and note what happens after we've reduced "return x ;" to Stmt and we see "return" as the lookahead symbol: we reduce to StmtList. So there is no longer a choice: with Stmt on the stack and "return" in the input, reduce by StmtList → Stmt. Running the revised grammar through ocamlyacc again shows that we have eliminated the conflict.

So, the strategy in this case was simply: use left-recursion instead of right-recursion. (Recall that left-recursion can never be used in top-down parsing; in bottom-up parsing, both left- and right-recursion are possible, but left-recursion often works better.)

Grammar 3

Expr $\rightarrow$ Expr + Expr | Expr * Expr | id | (Expr)

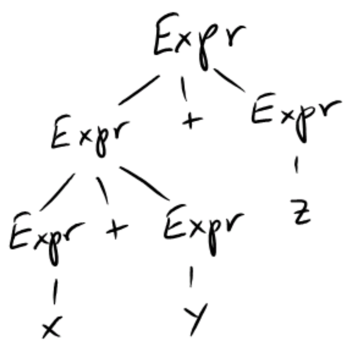
This is our ambiguous expression grammar. It is probably safe to say that most grammar conflicts reported by ocamlyacc are a result of ambiguity in the grammar. No ambiguous grammar can be conflict-free. ocamlyacc actually reports four shift/reduce conflicts for this grammar. Here is part of the ocamlyacc –v output for this grammar:

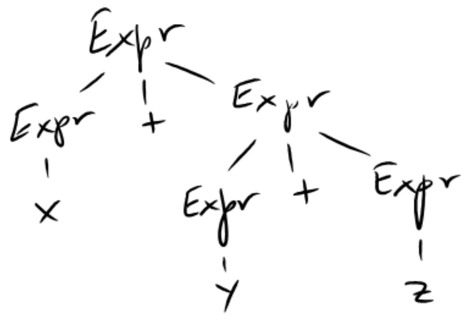
```
10: shift/reduce conflict (shift 7, reduce 1) on plus
10: shift/reduce conflict (shift 8, reduce 1) on star
state 10
     Expr : Expr . plus Expr  (1)
     Expr : Expr plus Expr .  (1)
     Expr : Expr . star Expr  (2)

     plus  shift 7
     star  shift 8
     $end  reduce 1
     cparen  reduce 1
```

As you can see, there are two conflicts in this one rule. (In this grammar, plus and star are treated identically, so any conflicts will involve both of them or neither. The other two conflicts are similarly associated with a single state, but I won't bother to discuss that because it's quite similar.)

As usual, we see one production here that has the period all the way on the right. This says that the parser can get to a state in which the stack contains "Expr plus Expr" and if the lookahead symbol is either plus or star, it cannot know whether to shift or reduce by that production.

The problem here is directly related to the ambiguity of the grammar. Consider the input "x+y+z". It has two parse trees:





It may not be obvious, but in each case, we get to a point where we have "Expr + Expr" on the stack. In both cases, the parse begins by shifting and reducing x, shifting +, and shifting and reducing y, leaving "Expr + Expr" on the stack and + as the lookahead symbol. But at this point, we make a different choice: if we now reduce using Expr → Expr + Expr, we will get the tree on the left; if we shift the +, we will reduce using Expr → Expr + Expr later, and end up with the tree on the right.

Both actions are "correct" in the sense that they will lead to a parse tree for the input, but we know that they are not both "correct" in the sense of the language semantics; plus should associate to the left, so the tree on the left is the one we want. As discussed in class, the same issue arises if we mix plus and times in an expression: the parse tree should reflect the precedence of times over plus.

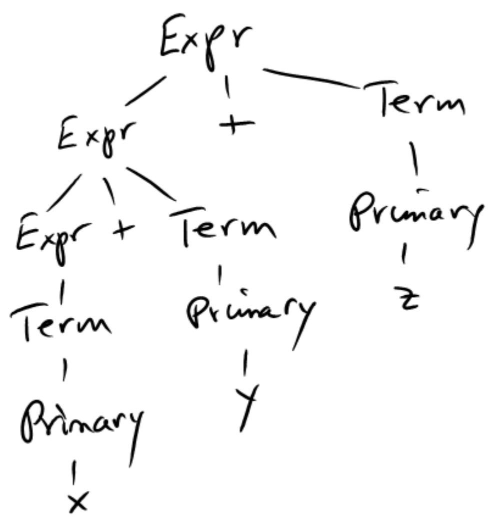
For this grammar, I'll give two solutions.

Solution 1

We can modify the grammar, turning it into a "stratified" grammar:

```
Expr    →  Expr  + Term  |  Term
Term    →  Term * Primary  | Primary
Primary →  id  |  ( Expr )
```

(Grammars like this have been studied for a long time, and the names "Term" and "Primary" are traditionally used.) You should convince yourselves that this grammar is unambiguous. The only parse tree for "x+y+z" is



which correctly associates addition to the left.

Solution 2

ocamlyacc provides a completely different way of solving this problem, using associativity and precedence declarations. To understand these, consider these four situations that lead to conflicts in this grammar (the two we mentioned above and the two that were in the expr.output file that we didn't show):

1. Stack contains: Expr + Expr, lookahead symbol is +

2. Stack contains: Expr + Expr, lookahead symbol is *

3. Stack contains: Expr * Expr, lookahead symbol is +

4. Stack contains: Expr * Expr, lookahead symbol is *

Consider first what the correct parse trees are in each of these cases. Since both + and * are left-associative, in cases 1 and 4 we want a tree that "leans to the left." Since * has precedence over +, in case 3, we also want a tree that "leans to the left." However, for the same reason, in case 2, we want the tree to "lean to the right."

Now think of these cases in terms of parse actions. In each of the four cases, reducing by the appropriate production (either Expr $\rightarrow$ Expr + Expr or Expr $\rightarrow$ Expr * Expr) will produce the left-leaning tree, while shifting will produce the right-leaning tree. If we only had a way to tell the parser: "When this conflict arises, reduce in every case except case 2." More precisely: "If the top-most symbol on the stack is +, then if the lookahead symbol is *, shift, and if the lookahead symbol is +, reduce; if the top-most symbol on the stack is *, then if the lookahead symbol is either + or *, reduce."

Or, more generally: "If the top-most symbol on the stack is the same as the lookahead symbol, then reduce; otherwise, if the lookahead symbol has lower precedence, reduce, and if it has higher precedence, shift."

ocamlyacc provides a way to say exactly that to the parser generator. Just after the %token line, add these two lines:

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

These indicate that both are left-associative, and, because the plus comes first, it has lower precedence. We can look at the verbose output again. Because the grammar itself has not changed, the number of "states," and in fact the overall structure of the output, will be the same. So we can look directly at the corresponding place in the output:

```
state 10
     Expr : Expr . plus Expr  (1)
     Expr : Expr plus Expr .  (1)
     Expr : Expr . star Expr  (2)

     star  shift 8
     $end  reduce 1
     cparen  reduce 1
     plus  reduce 1
```

As you can see, this looks almost identical (except there are no conflicts reported). The important difference is in the last line. Previously, there was a line, "`plus shift 7`," indicating that when the stack had Expr + Expr on top and the lookahead symbol was +, the parser should shift. As we have seen, this would lead to a right-associating parse. Now, we have "`plus  reduce 1`,", indicating to reduce in that case, giving the correct left-leaning parse tree. When star is the lookahead symbol, it still shifts, giving the correct *right*-leaning tree. (The "1" is the number of the production Expr $\rightarrow$ Expr + Expr; the first part of the verbose output, which we didn't show, lists all the productions and gives them each a number.)

As another sanity check, let's look at the analogous case for *: when the stack has Expr * Expr on top and the lookahead symbol is + or *. The parser should reduce in both cases – in the first case because * has higher precedence than +, and in the second because * is left-associative. Here is what the corresponding part of the verbose output says:

```
state 11
     Expr : Expr . plus Expr  (1)
     Expr : Expr . star Expr  (2)
     Expr : Expr star Expr .  (2)

     .  reduce 2
```

And that is exactly what it does, as shown in the very last line.

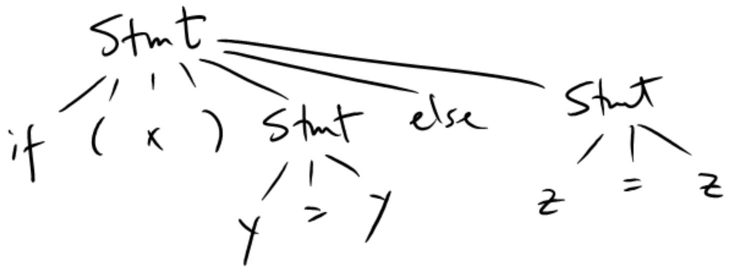
Grammar 4

Stmt → id = id | if (id) Stmt | if (id) Stmt else Stmt

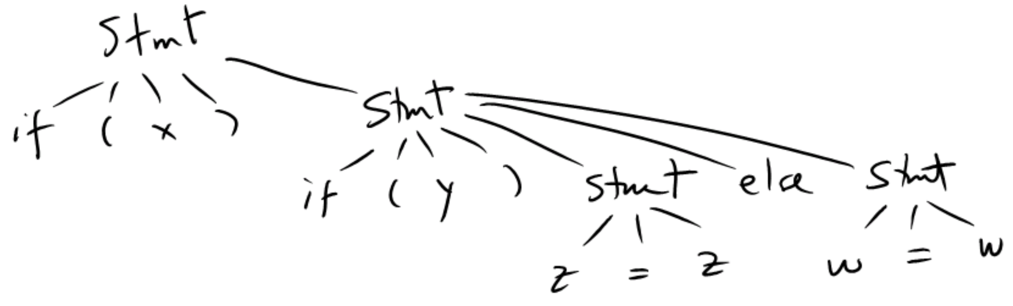
This grammar represents, in simplified form, the syntax of if statements in C or Java. We've simplified the condition and included just one other statement in order to focus on the interesting part of this, which is the optional "else" clause. This grammar is ambiguous. Here is the relevant part of the verbose output from ocamlyacc:

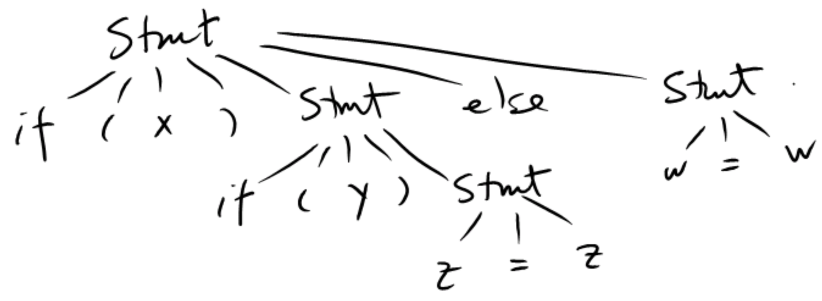
```
11: shift/reduce conflict (shift 12, reduce 2) on else
state 11
     Stmt : if oparen id cparen Stmt .  (2)
     Stmt : if oparen id cparen Stmt . else Stmt  (3)

     else  shift 12
     $end  reduce 2
```

Here's what it says: If we have "if (x) Stmt" on the stack, and the lookahead symbol is "else", then we don't know whether to shift or reduce. Again, we should see if we can find a statement that gets us into the problem. Doing so will reveal the source of the ambiguity. A first try might be the statement "if (x) y=y else z=z". This does lead to a state in which the stack contains "if (x) Stmt" and the input symbol is "else". However, there is no ambiguity here; the only correct action is shift, which produces this (correct) parse tree:



Looking closer, we can find another sentence that leads to this stack configuration, and which does cause a problem: "if (x) if (y) z=z else w=w". This is a nested if, and the problem is that the else clause may belong to either the first or second if. That is, if we had braces, we might disambiguate this either as "if (x) { if (y) z=z else w=w }" or "if (x) { if (y) z=z } else w=w". We do not have braces, but that's really not the point: the point is that the parser will produce a parse tree that reflects one of these two ways of interpreting the statement. It will be one of these:





These trees represent the two ways of grouping the statements, that is, they say which "if" the "else" belongs to. They represent different meanings for the statement – in the first, "w=w" is executed only if x and y are both false, while in the second, it is executed if x is false – and therefore only one of them can be correct.

Again, both parse trees lead to a stack configuration of the form "if (id) Stmt" with else in the input. More specifically, in both cases the entire stack will contain "if (x) if (y) Stmt". The first tree will be obtained if we shift the else; this will eventually lead to the configuration "if (x) if (y) Stmt else Stmt", and we will then reduce by Stmt → if (id) Stmt else Stmt and then by Stmt → if (id) Stmt. The second will be obtained if we reduce by Stmt → if (id) Stmt; we will then end up with configuration "if (x) Stmt else Stmt" and reduce by Stmt → if (id) Stmt else Stmt. (The difference is which production we reduce by first.)

So, the question is whether to shift or reduce, which comes down to the question: which parse tree is correct, in the sense that it gives the meaning the language specification requires. In fact, in every programming language you've ever used, the specification says that the first parse given above is the correct one – it associates the else clause with the closest if. To put it differently, it corresponds to what you intend if you write the statement with this indentation:

```
if (x)
    if (y)
       z = z
    else
       w = w
```

(It is a very common error in programming to enter this as

```
if (x)
    if (y)
       z = z
else
    w = w
```

and assume this reflects the actual meaning of the statement.)

So, how do we resolve the conflict? We again have two choices: either modify the grammar, or use an ocamlyacc trick. In this case, modifying the grammar is actually quite tricky. (A discussion of this can be found at http://marvin.cs.uidaho.edu/~heckendo/CS445F07/danglingElse.html.)

Here's the ocamlyacc trick: don't do anything. We can see from our discussion above that the correct action when this conflict arises – when "if (id) Stmt" is on the stack and else is the lookahead symbol – is to shift. Now look again at the ocamlyacc verbose output above. It says, "`else shift 12`," indicating that it will shift when it sees an else. In other words, ocamlyacc does the right thing – at least, it's right in this case – without our doing anything. (If you were looking very carefully, you might have noticed in our earlier examples, whenever there was a shift/reduce conflict, the ocamlyacc output indicated that it would shift.)

The only remaining issue is that it is annoying to keep getting the error message, even if we know it's harmless. You can avoid that with yet another ocamlyacc trick. We have in fact employed this in the grammar skeleton we supplied with MP6. We have the production:

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

The grammar has the "dangling else" ambiguity we've been discussing. Adding the "`%prec ELSE`" annotation tells ocamlyacc that if ELSE is the lookahead symbol, it should shift (again, this is like saying that ELSE has high precedence). It would have done that anyway, but this way it will not produce an error message. As indicated in the assignment, you should just leave that in the grammar and not worry about it.