CS 421 – Spring 2007
Lecture Notes Set 21: Ambiguous Grammars

Elsa L. Gunter[1]
Transcribed by: Pooja Mathur

Corresponding to Slides:
Made available: March 12, 2007
Revision 1.0

1 Change Log

1.0 Initial Release.

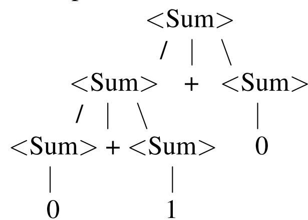
2 Ambiguity

In Languages and Grammars (slides 43 - 44, 49)

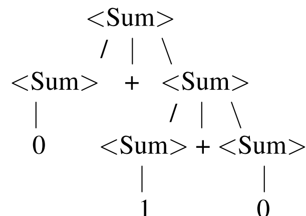
Last time we talked about there being ambiguity in grammars. That with some rules, you have so many options that multiple trees can be made. We also mentioned that there is no algorithm to stop ambiguity from occurring because there are inherently ambiguous grammars that have no disambiguous counterpart. The only way to fix ambiguous grammars in general is to find another grammar that generates the same regular expressions, but just has more non-terminals in there to stop the ambiguity. But when doing this, you need to know if the operations are left associative or right associative. That is, do you start (where starting means the operation is lower in the tree) from the left and apply the operations as you move to the right, or is it vice versa.

Basically, we need to learn the order of operations by the way we parse our expressions. And this leads us to the two main sources for ambuguity, the order of operations and associativity. (Note that these are not the only sources of ambiguity.

Here is an example of this. The expression 0 + 1 + 0 with the rules of <Sum> ::= 0 | 1 | 1 | (<Sum>) | <Sum> + <Sum> can be parsed as:



or...



[1] © 2007, Share and Enjoy

Either tree works for this expression, so the expression is ambiguous. Basically, it's the highest level <Sum> + <Sum>. Either one can break down into a terminal while the other breaks into the non-terminal expression. And since it can go on either side, we say that it is ambiguous.

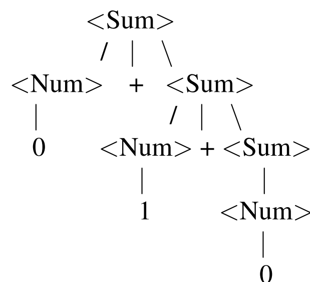
Associativity Matters (slides 50 - 51)

To deal with associativity, you make sure you only have one recusive call per iteration. That is, for the example when we had <Sum> + <Sum>, we have to find a way to make one of those terms simpler. So if you had something that was left associative, then you would start from the left and get to the right with the substitutions, and vice versa for right associativity.

So, what we do, to make the simpler terms is break the non-terminal up. Instead of using our original rules that we had from the last lecture, we are going to add in this <Num> non-terminal that can take us to our numbers and can't lead to getting to choose where the non-terminal expressions go.

<Sum> ::= <Num> | <Num> + <Sum>
<Num> ::= 0 | 1 | (<Sum>)

Now that we don't have the same non-terminals at the '+' levels, we can differentiate sides. So we can just get this now:



So, this is right associativity, which is right recursion. That is, remember, the lower in the tree the sooner the operation occurs. So, here we start at the right and move left wards up the tree.

Order Matters (slides 52 - 55)

Here, we have +'s and *'s that are competing with each other. We need to establish an order of precedence so we know which happens first. So if we wanted to give *'s higher precendence, we nee the +'s to be closer to the top of the trees. We are going to fix this by breaking layers up, like we did for associativity. You need to reflect precedence in your grammars. Higher precedence is said to 'bind more tightly'.

So if you had a grammar that said C * B + D, the C * B would be lower in the tree, and then the product would be sent up the tree to be added to D.

Precendence tables give you the order of operations used by a language. They usually go in order from higher precedence to lower precedence. So, * and / will be located higher in a table than + and -. Operations that have the same precendence are at the same level of the table. During these cases, the associativity rules kick in and that is how you deal with it. That is if they are at the same level the operations at that level better all associate to the left or all to the right. If they are different, then there is something wrong with the language.

Most languages work like the way we have learned since elementary school and the example from last time, 3 + 4 * 5 + 6, will equal 29. However, languages like APL, whose operations are all at the same level of precendence would not give 29.

Here is an example of a grammar that had ambiguity because of precedence...or lack there of:

<exp> ::= <id> | <exp> + <exp> | <exp> * <exp>

In this example, we have multiple problems. There are two problems with associativity. The + uses two <exp>'s and so does the *. Then, both + and * have the same precedence. At any applicable situation, you can use either of the two, creating multiple trees. To fix this, we add a new non-terminal which gets us:

<exp> ::= <mult_exp> | <exp> + <mult_exp>
<mult_exp> ::= <id> | <mult_exp> * <id>

Now, if we use this new grammar, we give * more precendence. That is the +'s will have to appear closer to the top of the tree than the *'s. You can see that by looking at the rules. The <mult_exp> can only lead to id's or other

<mult_exp> steps. They can't lead to any <exp> + <mult_exp> because only a <exp> can lead to that. That means that all the <exp> + <mult_exp> have to appear in the tree before we soley use the <mult_exp>.

Notice that both the + and * associate to the left. That is you can add more +'s under the left side of the <exp> + <mult_exp>. Then under the <mult_exp> rule, that also associates to the left since in <mult_exp> * <id>, the <mult_exp> appears on the left. To make this right associative, we can just flip the non-terminals around the operators. To make the + have more higher precendence, we exchange the <mult_exp> with a <sum_exp>, and switch the operators.

Some examples

Let's look at two operations with the same precedence, + and -, and let's say that + associates to the left and - associates to the right. If we have the expression 0 + 1 - 1 we have a problem. The + and the - are going to be fighting for the 1 in the middle. There's a deadlock in this parse tree derivation. So, we need to establish that everything on a certain level of precedence associate the same way.

So let's say that we had the following rules:

<exp> ::= <id> | <exp> + <exp> | <exp> - <exp>

We want to disambiguate this grammar, we are still going to allow the + and - to be at the same level, but we are going to say that they both associate to the left. To do this, we need to say that there is an <exp> on the left side of the operator, and then a 'something' on the right, so we have the following rule:

<exp> ::= <exp> + <?>

Now we need to look at the -, so we stick with the <exp> in the left again, and then we have another something on the right.

<exp> ::= <exp> - <?>

Now in this case, since it is such a simple language, we can easily fill in the <?> with <id>. If this was a more complicated language there could be other stuff put there but here, we can just say that:

<?> ::= <id> | ...

Note that the rule <exp> ::= <id> is there.