CS 421 – Spring 2007
Lecture Notes Set 20:
Parse Trees

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

Corresponding to Slides: `17 - bnf-grammars (slides 29-52)`
Made available: March 7, 2007
Revision 1.0

1 Change Log

1.0 Initial Release.

2 From last time...

Last time we talked about derivations from BNF Grammars. Recall the example the stemmed from the non-terminal <Sum>. At any point in the derivation, if there were more than one non-terminals, you have a choice on which non-terminal to work on at each step. And in addition to that, you have a choice on which rule to choose for that non-terminal. Having this choice is a necessary feature for BNF Grammars if you were restricted to one non-terminal, like we were in the example.

Also, how you work on non-terminal <A> is completely independent from how you work on non-terminal <B>. They are separate cases that lead to a final derivation. What you fill in for one doesn't force you to fill in something different (or even the same thing) in the other non-terminal. What we want to do is take away the superficial overhead of making the choices. The choices will still be there but they won't look as obvious as they did in the last lecture. In the last lecture, you coul look at the derivation and tell exactly what steps were taken when with which rules to derive the expression. We want to hide that, and that is the purpose of parse trees.

3 Parse Trees

3.1 Overview (slide 29)

The idea we are going to introduce is a gaphical representation of BNF derivations and technically, equivalence classes for derivations. That is the only way two derivations are different is if the order by which they substituted rules in. If a different order was choosen, but the rules stayed the same, their parse trees would also be the same. The nodes of the tree represent terminals and non-terminals. The ones that are terminals are leaves. The non-terminals are lead to a branch for each non-terminal and/or terminal in the rule used on it.

3.2 Example (slides 30 - 37)

Let's say we had the grammar with terminals of +, *, 0 and 1, non-terminals of <exp>, <factor> and <bin> with the following rules:

<exp> = <factor>
<exp> = <factor> + <factor>
<factor> = <bin>

[1] © 2007, Share and Enjoy

```
<factor> = <bin> * <exp>
<bin> = 0
<bin> = 1
```

and the start symbol of <exp> and we want to build a parse tree for the following expression: 1 * 1 + 0.

So, we start with the start symbol:

```
<exp> : 1 * 1 + 0
```

From here, we choose an arbitrary rules that stems from <exp>. So, we will choose <factor>.

```
<exp> : 1 * 1 + 0
   |
<factor>
```

Now we choose an arbitrary rule for an arbitrary leaf. There is only one leaf, so we choose that and the rule we choose is <bin> * <exp>.

```
    <exp> : 1 * 1 + 0
       |
    <factor>
     /  |  \
  <bin> * <exp>
```

Now, we choose two rules, one rule for <bin> and one for <exp>. So, we will choose the rule that <bin> = 1 and <exp> = <factor> + <factor>.

```
      <exp> : 1 * 1 + 0
         |
      <factor>
       /  |  \
  <bin>   *   <exp>
    |         /  |  \
    1  <factor> + <factor>
```

Notice that at this point you can’t look at the tree and tell that we choose to find a rule for <bin> before looking at <exp>, which is exactly what we want. So, we have two non-terminals now. We choose rules for them. We’ll choose the <bin> rule for both.

```
      <exp> : 1 * 1 + 0
         |
      <factor>
       /  |  \
  <bin>   *   <exp>
    |         /  |  \
    1  <factor> + <factor>
           |          |
         <bin>      <bin>
```

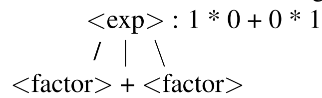
And we end with chooseing rules for both of the <bin>’s.

```
      <exp> : 1 * 1 + 0
         |
      <factor>
       /  |  \
  <bin>   *   <exp>
    |         /  |  \
    1  <factor> + <factor>
           |          |
         <bin>      <bin>
           |          |
           1          0
```

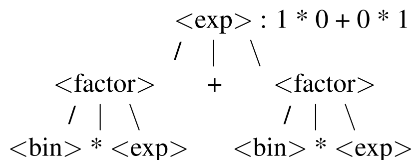
At this point, there are no more non-terminal leaves left, and notice that the leaves of the parse tree make up our expression we wanted to derive.

3.3 Another Example

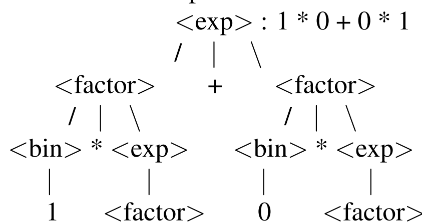
Let's build a parse tree for 1 * 0 + 0 * 1 starting with <exp>. <exp> : 1 * 1 + 0



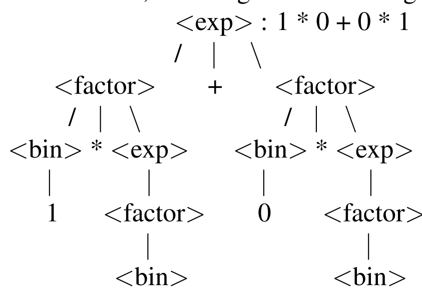
The first thing we did was, we choose an arbitrary rules that stems from <exp>. So, we will chose <factor> + <factor>.



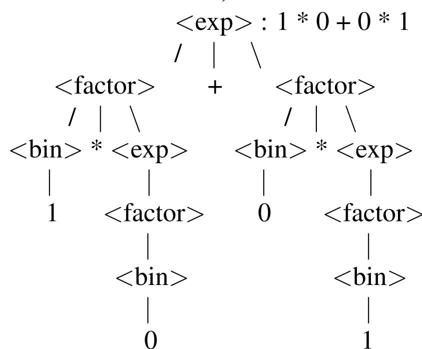
Here, we chose <bin> * <exp> for both non-terminal leaves.



We chose a 1 for the first <bin>, 0 for the secong <bin>, and then for each <exp> we chose <factor> because we know that from <factor>, we can get <bin>'s to get the the terminals we want.



So we placed the <bin> rules in, all that is left is to do is get the terminals.



And here, again, if you follow the leaves of the tree, you find that we have derived the expression.

3.4 Data Stucture (slides 39 - 42)

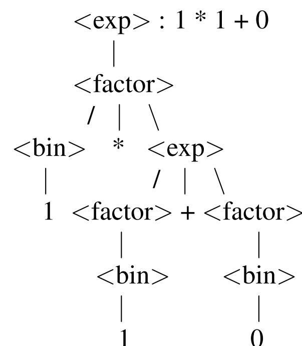
This graph is a mathematical expression. If we find that the making the parse tree leads us to multiple trees, we say that the expression is ambiguous. The tree also tells us about the intended grouping of the terminals in our expression. Like in the example we just finished, we are saying that the subexpression to the left of the + is grouped as well as the subexpression to the right of the +. This is a way of giving semantics to the string. The stuff at the top of the tree happens last. So here, we are saying that the * happens first and then we will get up to the + later.

In OCaml, the datatypes used are a very good match for creating these parse trees. Constructors can be considered the rules, datatypes themselves are the non-terminals.

So to write our grammar in OCaml, we could do something like:

```
datatype exp = Factor2Exp of factor
    | Plus of factor * factor
 and factor = Bin2Factor of bin
    | Mult of bin * exp
 and bin = Zero | One
```

So here we created that constructors for our rules. And if we had the first parse tree we made:



We could write it in OCaml as:

```
Factor2Exp
(Mult(One,
        Plus(Bin2Factor One,
                Bin2Factor Zero)))
```

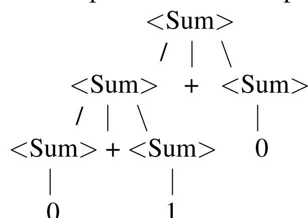
That is our exression goes from a factor to an expression of one star expression of a factor that goes to a bin that goes to a one and pluses that with a factor that goes to a bin that goes to zero.

4 Ambiguity

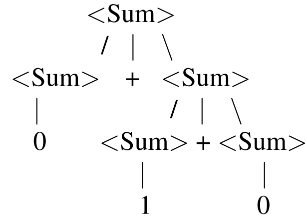
In Languages and Grammars (slides 43 - 44, 49)

So, we mentioned before that a grammar is ambiguous if there are more than one parse trees for it. If all the BNF's in a language are ambiguous then we say that the language is inherently ambiguous. Basically, we need to know if, say in the 1 * 1 + 0 case, are we doing the plus first or the times. We are going to learn this order by the way we parse our expressions. 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 can be parsed as:



or...



Either tree works for this expression, so the expression is ambiguous.

Order Matters (slides 45 - 46, 52)

Even though everyone learned this in elementary school, we are going to take a moment to emphasize that order matters. If you had the expression 3 + 4 * 5 + 6, you could get (3 + 4) * (5 + 6) = 7 * 11 = 77. Or ((3 + 4) * 5) + 6 = (7 * 5) + 6 = 35 + 6 = 41. Or 3 + (4 * (5 + 6)) = 3 + (4 * 11) = 3 + 44 = 47. Or, 3 + (4 * 5) + 6 = 3 + 20 + 6 = 29. That is four different answers to the same number and the same operations, but the operations were just applied in a different order.

To fix this, we need some operators to bind more tightly than other operators. We are going to do this by breaking layers up, like we will do for associativity. But if you wanted addition, for example, to happen before multiplication, you would want the plus to appear lower in the parse tree, because the closer it is to the bottom level of the tree, the sooner it will happen.

Associativity Matters (slides 47 - 48, 50 - 51)

Associativity matters too, or rather, the lack there of. With a addition and multiplication, it didn't matter the which exact order the numbers were in, as long as they had the correct operations applied on them. But here is an example where assiciativity matters. 7 - 5 - 2. (7 - 5) - 2 = 0 but 7 - (5 - 2) = 4.

To deal with this, 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:

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

We are going to add in this <Num> non-terminal that can take us to our numbers

<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.