
CS 421 – Spring 2007

Lecture Notes Set 19:

BNF Grammars

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 16 - ocamllex (slides 27-end) 17 - bnf-grammars (slides 1-27)
Made available: March 5, 2007
Revision 1.0

1 Change Log

1.0 Initial Release.

2 Comments

2.1 In general (slides 27 - 28)

To deal with comments, we are going to add another regular expression to our parser. That is the open comment and close comment expression. Now, it may seem very simple to use these regular expressions. You see the open comment, and you wait until you see a close comment. First, note that when you see the first open comment, you want to start parsing differently. Other regular expressions that aren't the open/close regular expressions, don't mean anything to this parser, which means we have to handle them differently.

To deal with this, we make two lexers. In our main lexer, when we see an open comment, we call our other parser. This other parser, basically sits and waits until the close comment regular expression is seen. Anything that is not that expression, simply gets eaten up by the parser. When the close comment regular expression is finally seen, then the action associated with that would be to go back to the first parser with whatever is left of lexbuf.

OCaml doesn't do this, but if we wanted to add into a language, the single line comment, the way to do that would be, to go to your comment parser, but instead of simply eating characters until the close comment expression was seen, the parser would be eating characters until the newline character expression was seen. When that is seen, the action would be to go back to the main parser and continue parsing like normal.

Even so, we still do have OCaml's comment structure quite exactly yet. This is the C++ way of handling things. There really isn't much in there for nested comments. OCaml is different though, OCaml wants to handle nested comments, but it can't do it like the way we have been. We need to keep track of how many open/close comment expressions we've seen.

2.2 Nested comments (slides 29 - 31)

Now, we need to tell our comment parser how many open comments expressions we have seen without close comment expressions. If we do that, we can begin to handle nested comments. So, we have the argument depth that keeps track of how many comment expressions are waiting to be closed. The simple easy case is that we have only seen one comment expression, and so when we call the comment parser, we are giving it the additional argument of 1. Then, inside the comment parser, when another open comment expression is seen, we call comment again, with 1 added to the depth argument. If we see a close comment expression, we call comment with 1 subtracted from its depth argument, however, if the depth is 1 and you see a close comment expression, then that is when it is time to back to

¹© 2007, Share and Enjoy

the main parser. And like normal, for a comment parser, anything that is not an open/close comment expression gets thrown aside.

The idea is that the depth argument is like a counter. Every time you see an open comment expression, that means you have at least 1 close comment to see. Then, if inside the first open comment expression, you see another open comment expression, you add another to your counter. You continue this trend until your counter is reduced to 1, where the next close comment expression will take lexbuf back to the main parser.

This is a very typical way of using arguments with parsers.

3 Grammars (slide set 2: slides 2 - 3)

We talked about this before. Grammars are formalized descriptions of which character sets of accepted by a certain language. They are basically the regular expressions we have been discussing for a few languages now. We discussed the way of using ASCII characters to describe a language. We have talked about using finite state automata as well.

3.1 Sample Grammar (slide 4)

Here, our language is describing all parenthesized sums of 0's and 1's. So our language specifically includes the left parenthesis, right parenthesis, the character, 0, the character 1, and finally the addition character. So we have, '(', ')', '1', '0', and '+'. These five characters make up our language. We also have the special symbol, <Sum>. Even though it isn't exactly a single character, look at it pretending it is one. So, when we say that <Sum> ::= 0, we are really saying that <Sum> can have the value of 0. <Sum> ::= 1 means that <Sum> can have the value 1. With the addition operator, we can say that <Sum> can have a value that is the addition of two <Sum>'s. And finally, we can say that you can put parentheses around it and it will still make sense.

3.2 Syntax for BNF Grammars (slides 5 - 7)

We will start with the characters, which on the corresponding slides are denoted as the early lower case letters, in this case, *a*, *b*, *c*,... These are your terminals, that is, the characters we are actually going to use to build our strings. Separately, you give a completely disjoint set of characters for nonterminals. These, will will denote as the later upper case letter, *X*, *Y*, *Z*, ... Non-terminals, are the variables we will use to build up the strings in our language. From the non-terminal set of characters, we need to pull one special symbol, that we will call our start symbol. It will tell us how to start building strings for our language.

Now that we have to pieces of our language, we will start to build rules, which can also be called productions. We use the ::= or an arrow to denote that a production is being made. That is, we have some non-terminal is the production of some terminal. The left side of the production symbol is always the non-terminal. So if we had $X ::= a$, we are saying that *X* is the production of *a*. Note that terminals and non-terminals can appear on the right side.

So with our example from before, our terminals were, 0, 1, +, (, and). There, technically no meaning to the terminals. They are simply symbols and should be treated as such. Our non-terminal was the <Sum>. For our start symbol, since we only have one non-terminal, it is trivial to choose the <Sum>. (In BNF Grammars, it is traditional to put angle brackets around non-terminals, so that is what we have done.) So, the basis of our language may look like:

$$\begin{aligned} \langle \text{Sum} \rangle &::= 0 \\ \langle \text{Sum} \rangle &::= 1 \\ \langle \text{Sum} \rangle &::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \\ \langle \text{Sum} \rangle &::= (\langle \text{Sum} \rangle) \end{aligned}$$

But this can also be written as the following, applying the use of the verticle bar as the or symbol:

$$\langle \text{Sum} \rangle ::= 0 \mid 1 \mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \mid (\langle \text{Sum} \rangle)$$

3.3 BNF Derivations (slides 8 - 25)

Now that we know the basic syntax, we can start to derive strings. With our rules, we are going to build up strings. We are going to do this by substitution. We are going to take our non-terminals and replace them with strings.

Let's say that we have $X ::= yZw$ and that $Z ::= v$. We can replace the Z in the right side of the first expression with the terminal v . So, $X ::= yZw ::= yvw$. We call the derivation *right-most* if we always end up replacing the right-most non-terminal. There can be numerous non-terminals in the right side of the rule, but if you always replace the one farthest to the right, you are performing a right-most derivation.

Now, let's look at an example. We will start with the start symbol we had earlier, the $\langle \text{Sum} \rangle$.

$$\langle \text{Sum} \rangle ::=$$

Now we pick a non-terminal, the only one we have is the start symbol, so we pick that one. Next, we pick a rule (let's say the $\langle \text{Sum} \rangle + \langle \text{Sum} \rangle$ rule) that the $\langle \text{Sum} \rangle$ can go to, and substitute.

$$\langle \text{Sum} \rangle ::= \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$$

From here, we pick another non-terminal. There are two to choose from on the right side of the rule. We arbitrarily pick the first one. And again, we pick a rule (the $(\langle \text{Sum} \rangle)$ rule) and then substitute.

$$::= (\langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$

Then we apply the same procedure, pick a non-terminal, pick a rule, substitute:

$$::= (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + \langle \text{Sum} \rangle$$

$$::= (\langle \text{Sum} \rangle + 1) + \langle \text{Sum} \rangle$$

$$::= (\langle \text{Sum} \rangle + 1) + 0$$

$$::= (0 + 1) + 0$$

And in the end, we get that $(0 + 1) + 0$ is generated by the grammar. Note, that at this point, the 0, 1, +, (, and) have no specific meaning. Do not add the numbers together and say that the grammar generates 1, because here we derived that it generated more than just 1, we got $(0 + 1) + 0$.

Let's look at another example. This time, see if you can follow the example along yourself, figuring out which non-terminal and rule is chosen at each step.

$$\langle \text{Sum} \rangle ::= (\langle \text{Sum} \rangle)$$

$$::= (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle)$$

$$::= (0 + \langle \text{Sum} \rangle)$$

$$::= (0 + (\langle \text{Sum} \rangle))$$

$$::= (0 + (0))$$

So our generated string is $(0 + (0))$.

Overall, the meaning of a BNF grammar consists of the set of all strings that have only terminals on the right side and can be derived with simply the start symbol, which is exactly what we have been doing the previous two examples.

Now have seen how to generate strings. What we have been wanting to know all along though, is whether a string is in a language or not. But, we can't just generate all possible strings, which can technically be an infinite amount of strings, and then at the very end, ask, was our particular string one of these infinitely many strings? No, this is why we like parsing.

3.4 Extended Syntax (slide 26)

What we have been talking about so far are the basics of BNF Grammar syntax. Let's, now, look at the extended syntax.

You can use the vertical bar as the or symbol. That is, instead of having to write $X ::= y$, $X ::= z$, you can abbreviate it as $X ::= y | z$. This is helpful for making the rules more compact and readable.

If you wanted to use terminal, optionally, you can write $X ::= y[v]z$ to mean $X ::= yz$, $X ::= yvz$. This is saying that we can have the terminal v in there, but we don't have to have it.

If you wanted to use the possible use of repeated terminals, that is, the Kleene Star, $X ::= y\{v\}^*z$, can be expressed by the following $X ::= yz$, $X ::= yVz$, $V ::= v$, and $V ::= vV$. Note, that we aren't abbreviating anything here. We are actually adding in a new non-terminal V .

3.5 Regular grammars (slide 27)

Regular grammars are the subclass of BNF Grammars used to describe regular expressions. So, you can have $\langle \text{non-terminal} \rangle ::= \langle \text{terminal} \rangle$, $\langle \text{non-terminal} \rangle ::= \langle \text{non-terminal} \rangle$, $\langle \text{non-terminal} \rangle ::= \langle \text{non-terminal} \rangle \langle \text{terminal} \rangle$,

or $\langle \text{non-terminal} \rangle ::= \langle \text{terminal} \rangle \langle \text{non-terminal} \rangle$. This defines the same class of languages as described by regular expressions. It can be used as an alternate input for lexers, instead of the lexbuf we have been using.

So, as an example, if we wanted to describe the strings where every substring has of even length has the same number of 1's and 0's with regular grammars, instead of regular expressions, we would get:

$$\begin{aligned} \langle \text{Balanced} \rangle &::= \epsilon \\ \langle \text{Balanced} \rangle &::= 0 \langle \text{OneAndMore} \rangle \\ \langle \text{Balanced} \rangle &::= 1 \langle \text{ZeroAndMore} \rangle \\ \langle \text{OneAndMore} \rangle &::= 1 \langle \text{Balanced} \rangle \\ \langle \text{ZeroAndMore} \rangle &::= 0 \langle \text{Balanced} \rangle \end{aligned}$$

Here our non-terminals are $\langle \text{Balanced} \rangle$, $\langle \text{ZeroAndMore} \rangle$, and $\langle \text{OneAndMore} \rangle$, where are start symbol is $\langle \text{Balanced} \rangle$. Our terminals are ϵ , 1, and 0.