Using EBNF to Define the Syntax of Languages

Max Hailperin

MC28, November 22, 1995

Every language has certain sequences of symbols that are *well formed* or *syntactically correct.* For example, in Scheme, `(lambda (x) (if x x 0))` is well formed, while `(if lambda 0 (x) x` is not. Sometimes a well-formed sequence of symbols will be meaningless; for example, in Scheme there is no meaning assigned to the expression `(#T #F)`, but it has the correct syntactic form, since the boolean literals `#T` and `#F` are expressions and any collection of one or more expressions surrounded by parentheses, such as `(#T #F)`, is also an expression. We say such an expression is *semantically incorrect.*

The first step in defining a formal language, such as a programming language, is generally to precisely define its *syntax*, i.e., which sequences of symbols are well formed. In this handout we'll look at a concise notation, EBNF, that is commonly used for defining the syntax of formal languages. (The name EBNF stands for Extended Backus-Naur Form, since this notation is an extension to a form of syntax definition that John Backus developed and then Peter Naur popularized by using it in the published definition of Algol, which he edited.)

EBNF is one example of a notation for language *grammars*, which specify how *syntactic categories* are recursively structured. The basic idea is to be able to say things like "any collection of one or more expressions surrounded by parentheses is also an expression," which is an inherently recursive statement. The only difference is that rather than saying it in English, we have a notation for saying it that is both more concise and more precise. Regarding precision, notice that the English version could be misread as saying that each of the individual expressions is surrounded by parentheses, rather than the whole collection. Regarding concision, here is the EBNF version:

$\langle\text{expression}\rangle \longrightarrow \texttt{(}\langle\text{expression}\rangle^{+}\texttt{)}$

This collection of symbols with an arrow in it is called a *production* of the grammar. The word $\langle$expression$\rangle$ with the angle brackets around it is a *syntactic category name* or *nonterminal.* The purpose of the grammar is generally to define one specific *syntactic category*; for Scheme it would be $\langle$expression$\rangle$. However, along the way it may be useful to define other syntactic categories; for example, in specifying the syntax of `cond` expressions, we may want to use a syntactic category called $\langle$clause$\rangle$. The meaning of a production is that the right-hand side specifies one form that is permissable for the syntactic category listed on the left-hand side. For example, the above production gives one form that an $\langle$expression$\rangle$ can have.

The parentheses in the example production's right-hand side are necessary symbols that must appear in any $\langle$expression$\rangle$ of that form; these are called *terminal* symbols. For another example, the production

$\langle$expression$\rangle$ $\longrightarrow$ `(if` $\langle$expression$\rangle$ $\langle$expression$\rangle$ $\langle$expression$\rangle$`)`

contains the keyword `if` as a terminal symbol as well as the parentheses. (Alternatively, we could consider each of the individual letters `i` and `f` to be a separate terminal symbol; we'll duck the question of which approach is preferable and why.)

At this point we have two productions for $\langle$expression$\rangle$, because we have given two different forms that $\langle$expression$\rangle$s can have. This is normal; many syntactic categories will be specified by a collection of productions specifying alternative forms the category can have. It makes the grammar easier to read if all the productions for a particular category are grouped together; there is a notational shorthand that is generally used for this. In the case of our two productions for $\langle$expression$\rangle$, this shorthand notation would be as follows:

$\langle$expression$\rangle$ $\longrightarrow$ `(`$\langle$expression$\rangle^{+}$`)`
    | `(if` $\langle$expression$\rangle$ $\langle$expression$\rangle$ $\langle$expression$\rangle$`)`

The vertical bar is used to indicate that another production for the same left-hand side follows. Any number of productions can be grouped together in this way. If the right-hand sides are short, then it is also permissable to list them on the same line, as follows:

$\langle$digit$\rangle$ $\longrightarrow$ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Note, incidentally, that none of these productions for $\langle$digit$\rangle$ contains any nonterminal symbols on the right-hand side. Every grammar must have some productions like that, to provide the base case for the recursion inherent in grammars.

The first production given for $\langle$expression$\rangle$ had a superscript plus sign in its right hand side; this is a special notation that means "one or more." In particular, $\langle\text{expression}\rangle^+$ is the EBNF way to say "one or more $\langle$expression$\rangle$s." This was used in order to say that one form an $\langle$expression$\rangle$ can have is a pair of parentheses surrounding one or more $\langle$expressions$\rangle$s.

There is another very similar notation that can be used to say "zero or more." For example, suppose we want to specify the syntax of `lambda` expressions. The body should consist of one or more $\langle$expression$\rangle$s, but the parameter list should be allowed to have zero or more $\langle$variable$\rangle$s in it, so that we can have parameterless procedures. This would be expressed as follows:

$\langle\text{expression}\rangle \longrightarrow$ `(lambda (`$\langle\text{variable}\rangle^*$`)` $\langle\text{expression}\rangle^+$`)`

The general rule is that a syntactic category name with a superscript asterisk indicates zero or more instances of the category, while a syntactic category name with a superscript plus sign indicates one or more instances of the category.

At this point, we have presented the entire EBNF notation, and can show a larger example. Following the example we'll consider some topics related to the use of EBNF. Here is the syntax of the "mini-scheme" accepted by the meta-circular evaluator:

$\langle\text{expression}\rangle \longrightarrow \langle\text{self evaluating}\rangle \mid \langle\text{quotation}\rangle \mid \langle\text{variable}\rangle \mid \langle\text{definition}\rangle$
$\quad\mid \langle\text{assignment}\rangle \mid \langle\text{lambda}\rangle \mid \langle\text{conditional}\rangle \mid \langle\text{application}\rangle$

$\langle\text{self evaluating}\rangle \longrightarrow \langle\text{number}\rangle$

$\langle\text{quotation}\rangle \longrightarrow$ `(quote` $\langle\text{datum}\rangle$`)`

$\langle\text{definition}\rangle \longrightarrow$ `(define` $\langle\text{variable}\rangle$ $\langle\text{expression}\rangle$`)`
$\quad\mid$ `(define (`$\langle\text{variable}\rangle$ $\langle\text{variable}\rangle^*$`)` $\langle\text{expression}\rangle^+$`)`

$\langle\text{assignment}\rangle \longrightarrow$ `(set!` $\langle\text{variable}\rangle$ $\langle\text{expression}\rangle$`)`

$\langle$lambda$\rangle \longrightarrow$ `(lambda (`$\langle$variable$\rangle$`*) `$\langle$expression$\rangle^{+}$`)`

$\langle$conditional$\rangle \longrightarrow$ `(cond `$\langle$clause$\rangle$`*)`

$\langle$clause$\rangle \longrightarrow$ `(`$\langle$expression$\rangle$ $\langle$expression$\rangle^{+}$`)`
  | `(else `$\langle$expression$\rangle^{+}$`)`

$\langle$application$\rangle \longrightarrow$ `(`$\langle$expression$\rangle$ $\langle$expression$\rangle$`*)`

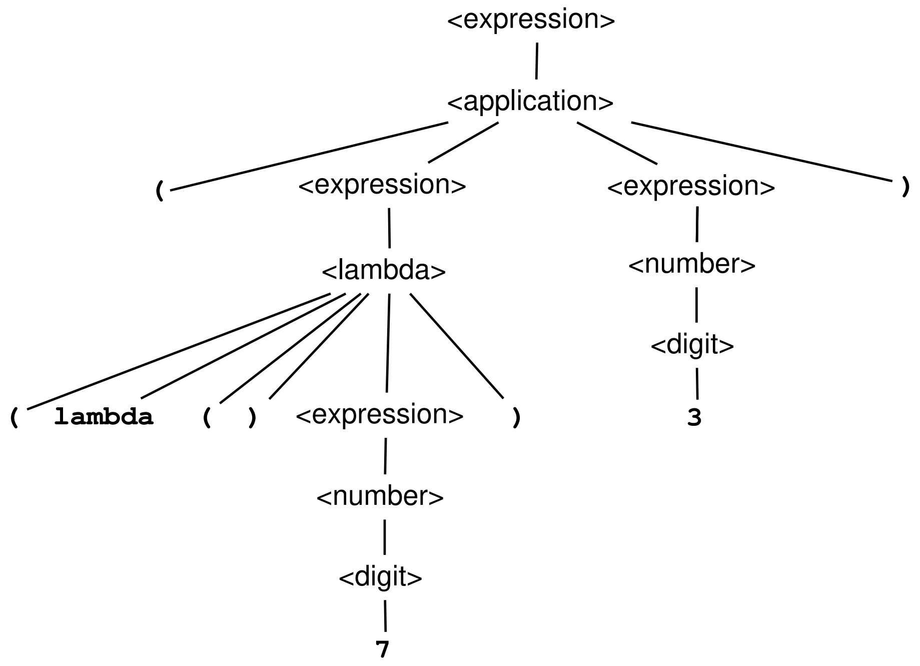
$\langle$variable$\rangle \longrightarrow \langle$any symbol allowed by the underlying Scheme$\rangle$
$\langle$number$\rangle \longrightarrow \langle$any number allowed by the underlying Scheme$\rangle$
$\langle$datum$\rangle \longrightarrow \langle$any datum allowed by the underlying Scheme$\rangle$

You will notice that at the end of the grammar there are three syntactic categories left defined only in terms of the underlying Scheme. The only thing stopping us from expressing these in EBNF is that the textbook chooses not to precisely specify them. To give an example of how they might be specified, here is one possible syntax for $\langle$number$\rangle$, building on the earlier definition of $\langle$digit$\rangle$:

$\langle$number$\rangle \longrightarrow \langle$digit$\rangle^{+}$
  | $\langle$digit$\rangle^{+}$`.`$\langle$digit$\rangle^{*}$ | $\langle$digit$\rangle^{*}$`.`$\langle$digit$\rangle^{+}$
  | $\langle$digit$\rangle^{+}$`/`$\langle$digit$\rangle^{+}$

Because grammars are recursive, it is natural to use mathematical induction to prove facts about the languages they define. The base case comes from productions with only terminal symbols on the right hand side, while the inductive step comes from the other productions. We won't go into the details of this here, but let's at least sketch an example. Suppose we wanted to prove that any $\langle$expression$\rangle$ has an equal number of left and right parentheses. In the base case we'd look at productions like the ones for $\langle$digit$\rangle$ and trivially see that there are an equal number of the two kinds of parentheses (since there are none of either kind). In the inductive step, we'd look at a right hand side like `(`$\langle$expression$\rangle$ $\langle$expression$\rangle^{+}$`)` and inductively assume that each of the smaller subexpressions had an equal number of left and right parentheses. Adding all these up and adding one more left parenthesis and one more right parenthesis again yields and equal number of left and right parentheses for the overall expression.

Using an EBNF grammar, we can draw a *parse tree* showing the organizational structure of an expression; for example, the application expression `((lambda () 7) 3)` has the following parse tree:



If a grammar has the property that no expression can be diagramed using two different parse trees, the grammar is called *unambiguous*. If, on the other hand, some expressions have more than one parse tree, the grammar is *ambiguous*. Since we generally define the meaning (or *semantics*) of an expression based on the structure of its parse tree, we generally try to use only unambiguous grammars. As a simple example of an ambiguous grammar, consider the following:

$\langle\text{snore}\rangle \longrightarrow$ `z` $\mid$ $\langle\text{snore}\rangle\langle\text{snore}\rangle$

The grammar for $\langle$snore$\rangle$s given above specifies any string of one or more `z`s, but does so without using the superscripted plus sign—it uses recursion instead. This is actually a specific example of a more general phenomenon: if we removed the asterisk and plus sign from EBNF, we would not lose any *expressive power*: we'd still be able to specify all the same languages. For example, here is how the production for $\langle$lambda$\rangle$ could be rewritten:

$\langle$lambda$\rangle$ $\longrightarrow$ `(lambda (`$\langle$optional variables$\rangle$`)` $\langle$expressions$\rangle$`)`

$\langle$optional variables$\rangle$ $\longrightarrow$ $\langle$empty$\rangle$ | $\langle$variable$\rangle$ $\langle$optional variables$\rangle$

$\langle$expressions$\rangle$ $\longrightarrow$ $\langle$expression$\rangle$ | $\langle$expression$\rangle$ $\langle$expressions$\rangle$

$\langle$empty$\rangle$ $\longrightarrow$

Notice that to allow zero variables, we used a syntactic category $\langle$empty$\rangle$ that has a totally empty right-hand side on its one and only production. If you compare this version with the previous one, you will probably agree that although the asterisk and plus sign don't contribute any additional expressive power, they do contribute *notational convenience*.

We've seen that we can live without the asterisk and plus sign, using only recursion. How about the other way around? What if we allowed asterisk and plus sign but forbade recursion? Some syntactic categories can certainly be defined this way. For example, our definition of $\langle$number$\rangle$ in terms of $\langle$digit$\rangle$ made no use of recursion. However, our definition of $\langle$expression$\rangle$ made considerable use of recursion. In our full grammar for "mini-scheme" the recursion isn't *direct recursion*, in that $\langle$expression$\rangle$ doesn't appear on the right-hand side of any of the productions for $\langle$expression$\rangle$. Instead, we have a case of *indirect recursion*, via such other syntactic categories as $\langle$application$\rangle$. It turns out that if you outlaw all recursion (indirect as well as direct), it is no longer possible to define syntactic categories like $\langle$expression$\rangle$. In fact, if all you have is repetition using asterisk and plus sign, you can't even produce a grammar for such a simple language as the one consisting of `()`, `(())`, `((()))`, etc., i.e., any number of left parentheses followed by the same number of right parentheses. In a successor course on formal languages and automata theory you'll prove this, as well as addressing other similar questions of the relative expressive power of various forms of language definition.

Don't be fooled into thinking that EBNF (complete with recursion) is the ultimate in power, however. Even with recursion you can't use EBNF to define such a simple language as the one consisting of `(x)`, `((xx))`, `(((xxx)))`, etc., i.e., any number of left parentheses, the same number of `x`s, and the same number of right parentheses. For most programming languages, however, EBNF proves to be a good balance between power and convenience.