BNF and EBNF: What are they and how do they work?

By: Lars Marius Garshol

Contents

- Introduction
  - What is this?
  - What is BNF?
- How it works
  - The principles
  - A real example
  - EBNF: What is it, and why do we need it?
  - An EBNF sample grammar
- Uses of BNF and EBNF
  - Common uses
  - How to use a formal grammar
- Parsing
  - The easiest way
    - Top-down parsing (LL)
    - An LL analysis example
    - An LL transformation example
  - The slightly harder way
    - Bottom-up parsing (LR)
  - LL or LR?
  - More information
- Appendices
  - Acknowledgements

Introduction

What is this?

This is a short article that attempts to explain what BNF is, based on message <wkwwagbizn.fsf@ifi.uio.no> posted to comp.text.sgml on 16.Jun.98. Because of this it is a little rough, so if it leaves you with any unanswered questions, email me and I'll try to explain as best I can.

It has been filled out substantially since then and has grown quite large. However, you needn't fear. The article gets more and more detailed as you read on, so if you don't want to dig really deep into this, just stop reading when the questions you are interested in have been answered and things start getting boring.

What is BNF?

Backus-Naur notation (more commonly known as BNF or Backus-Naur Form) is a formal mathematical way to describe a language, which was developed by John Backus (and possibly Peter Naur as well) to describe the syntax of the Algol 60 programming language.

(Legend has it that it was primarily developed by John Backus (based on earlier work by the mathematician Emil Post), but adopted and slightly improved by Peter Naur for Algol 60, which made it well-known. Because of this Naur calls BNF Backus Normal Form, while everyone else calls it Backus-Naur Form.)

It is used to formally define the grammar of a language, so that there is no disagreement or ambiguity as to what is allowed and what is not. In fact, BNF is so unambiguous that there is a lot of mathematical theory around these kinds of grammars, and one can actually mechanically construct a parser for a language given a BNF grammar for it. (There are some kinds of grammars for which this isn't possible, but they can usually be transformed manually into ones that can be used.)

Programs that do this are commonly called "compiler compilers". The most famous of these is YACC, but there are many more.

How it works

The principles

BNF is sort of like a mathematical game: you start with a symbol (called the start symbol and by convention usually named S in examples) and are then given rules for what you can replace this symbol with. The language defined by the BNF grammar is just the set of all strings you can produce by following these rules.

The rules are called production rules, and look like this:

```
  symbol := alternative1 | alternative2 ...
```

A production rule simply states that the symbol on the left-hand side of the := must be replaced by one of the alternatives on the right hand side. The alternatives are separated by |s. (One variation on this is to use ::= instead of :=, but the meaning is the same.) Alternatives usually consist of both symbols and something called terminals. Terminals are simply pieces of the final string that are not symbols. They are called terminals because there are no production rules for them: they terminate the production process. (Symbols are often called non-terminals.)

Another variation on BNF grammars is to enclose terminals in quotes to distinguish them from symbols. Some BNF grammars explicitly show where whitespace is allowed by having a symbol for it, while other grammars leave this for the reader to infer.

There is one special symbol in BNF: @, which simply means that the symbol can be removed. If you replace a symbol by @, you do it by just removing the symbol. This is useful because in some cases it is difficult to end the replacement process without using this trick.

So, the language described by a grammar is the set of all strings you can produce with the production rules. If a string cannot in any way be produced by using the rules the string is not allowed in the language.

A real example

Below is a sample BNF grammar:

```
  S  := '-' FN |
        FN
  FN := DL |
        DL '.' DL
  DL := D |
        D DL
  D  := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

The different symbols here are all abbreviations: S is the start symbol, FN produces a fractional number, DL is a digit list, while D is a digit.

Valid sentences in the language described by this grammar are all numbers, possibly fractional, and possibly negative. To produce a number, start with the start symbol S:

```
  S
```

Then replace the S symbol with one of its productions. In this case we choose not to put a '-' in front of the number, so we use the plain FN production and replace S by FN:

```
  FN
```

The next step is then to replace the FN symbol with one of its productions. We want a fractional number, so we choose the production that creates two decimal lists with a '.' between them, and after that we keep choosing replacing a symbol with one of its productions once per line in the example below:

```
  DL . DL

  D . DL

  3 . DL

  3 . D DL

  3 . D D

  3 . 1 D

  3 . 1 4
```

Here we've produced the fractional number 3.14. How to produce the number -5 is left as an exercise for the reader. To make sure you understand this you should also study the grammar until you understand why the string 3..14 cannot be produced with these production rules.

EBNF: What is it, and why do we need it?

In DL I had to use recursion (ie: DL can produce new DLs) to express the fact that there can be any number of Ds. This is a bit awkward and makes the BNF harder to read. Extended BNF (EBNF, of course) solves this problem by adding three operators:

- ? : which means that the symbol (or group of symbols in parenthesis) to the left of the operator is optional (it can appear zero or one times)
- * : which means that something can be repeated any number of times (and possibly be skipped altogether)
- + : which means that something can appear one or more times

An EBNF sample grammar

So in extended BNF the above grammar can be written as:

```
  S := '-'? D+ ('.' D+)?

  D := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

which is rather nicer. :)

Just for the record: EBNF is not more powerful than BNF in terms of what languages it can define, just more convenient. Any EBNF production can be translated into an equivalent set of BNF productions.

Uses of BNF and EBNF

Common uses

Most programming language standards use some variant of EBNF to define the grammar of the language. This has two advantages: there can be no disagreement on what the syntax of the language is, and it makes it much easier to make compilers, because the parser for the compiler can be generated automatically with a compiler-compiler like YACC.

EBNF is also used in many other standards, such as definitions of protocol formats, data formats and markup languages such as XML and SGML. (HTML is not defined with a grammar, instead it is defined with an SGML DTD, which is sort of a higher-level grammar.)

You can see a collection of BNF grammars at the BNF web club .

How to use a formal grammar

OK. Now you know what BNF and EBNF are, what they are used for, but perhaps not why they are useful or how you can take advantage of them.

The most obvious way of using a formal grammar has already been mentioned in passing: once you've given a formal grammar for your language you have completely defined it. There can be no further disagreement on what is allowed in the language and what is not. This is extremely useful because a syntax description in ordinary prose is much more verbose and open to different interpretations.

Another benefit is this: formal grammars are mathematical creatures and can be "understood" by computers. There are actually lots of programs that can be given (E)BNF grammars as input and automatically produce code for parsers for the given grammar. In fact, this is the most common way to produce a compiler: by using a so-called compiler-compiler that takes a grammar as input and produces parser code in some programming language.

Of course, compilers do much more checking that just grammar checking (such as type checking) and they also produce code. None of these things are described in an (E)BNF grammar, so compiler-compilers usually have a special syntax for associating code snippets (called actions) with the different productions in the grammar.

The best-known compiler-compiler is YACC (Yet Another Compiler Compiler), which produces C code, but others exist for C++, Java, Python as well as many other languages.

Parsing

The easiest way

Top-down parsing (LL)

The easiest way of parsing something according to a grammar in use today is called LL parsing (or top-down parsing). It works like this: for each production find out which non-terminals the production can start with. (This is called the start set.)

Then, when parsing, you just start with the start symbol and compare the start sets of the different productions against the first piece of input to see which of the productions have been used. Of course, this can only be done if no two start sets for one symbol both contain the same terminal. If they do there is no way to determine which production to choose by looking at the first terminal on the input.

LL grammars are often classified by numbers, such as LL(1), LL(0) and so on. The number in the parenthesis tells you the maximum number of terminals you may have to look at at a time to choose the right production at any point in the grammar. So for LL(0) you don't have to look at any terminals at all, you can always choose the right production. This is only possible if all symbols have only one production, and if they only have one production the language can only have one string. In other words: LL(0) grammars are not interesting.

The most common (and useful) kind of LL grammar is LL(1) where you can always choose the right production by looking at only the first terminal on the input at any given time. With LL(2) you have to look at two symbols, and so on. There exist grammars that are not LL(k) grammars for any fixed value of k at all, and they are sadly quite common.

An LL analysis example

As a demonstration, let's do a start set analysis of the sample grammar above. For the symbol D this is easy: all productions have a single digit as their start set (the one they produce) and the D symbol has the set of all ten digits as its start set. This means that we have at best an LL(1) grammar, since in this case we need to look at one terminal to choose the right production.

With DL we run into trouble. Both productions start with D and thus both have the same start set. This means that one cannot see which production to choose by looking at just the first terminal of the input. However, we can easily get round this problem by cheating: if the second terminal on input is *not* a digit we must have used the first production, but if they both are digits we must have used the second one. In other words, this means that this is at best an LL(2) grammar.

I actually simplified things a little here. The productions for DL alone don't tells us which terminals are allowed after the first terminal in the D @ production, because we need to know which terminals are allowed after a DL symbol. This set of terminals is called the follow set of the symbol, and in this case it is '.' and the end of input.

The FN symbol turns out to be even worse, since both productions have all digits as their start set. Looking at the second terminal doesn't help since we need to look at the first terminal after the last digit in the digit list (DL) and we don't know how many digits there are until we've read them all. And since there is no limit on the number of digits there can be, this isn't an LL(k) grammar for any value of k at all (there can always be more digits than k, no matter which value of k value you choose).

Somewhat surprisingly perhaps, the S symbol is easy. The first production has '-' as its start set, the second one has all digits. In other words, when you start parsing you'll start with the S symbol and look at the input to decide which production was used. If the first terminal is '-' you know that the first production was used. If not, the second one was used. It's only the FN and DL productions that cause problems.

An LL transformation example

However, there is no need to despair. Most grammars that are not LL(k) can fairly easily be converted to LL(1) grammars. In this case we'll need to change two symbols: FN and DL.

The problem with FN is that both productions begin with DL, but the second one continues with a '.' and another DL after the initial DL. This is easily solved: we change FN to have just one production that starts with DL followed by FP (fractional part), where FP can be nothing or '.' followed by a DL, like this:

```
FN := DL FP
FP := @ | '.' DL
```

Now there are no problems with FN anymore, since there's just one production, and FP is unproblematic because the two productions have different start sets. End of input and '.', respectively.

The DL is a tougher nut to crack, since the problem is the recursion and it's compounded by the fact that we need at least one D to result from the DL. The solution is to give DL a single production, a D followed by DR (digits rest). DR then has two productions: D DR (more digits) or @ (no more digits). The first production has a start set of all digits, while the second has '.' and end of input as its start set, so this solves the problem.

This is the complete LL(1) grammar as we've now transformed it:

```
S  := '-' FN | FN
FN := DL FP
FP := @ | '.' DL
DL := D DR
DR := D DR | @
D  := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

The slightly harder way

Bottom-up parsing (LR)

A harder way to parse is the one known as shift-reduce or bottom-up parsing. This technique collects input until it finds that it can reduce an input sequence with a symbol. This may sound difficult, so I'll give an example to clarify. We'll parse the string '3.14' and see how it was produced from the grammar. We start by reading 3 from the input:

```
3
```

and then we look to see if we can reduce it to the symbol it was produced from. And indeed we can, it was produced from the D symbol, which we replace the 3 with. Then we note that we can produce the D from DL and replace the D with DL. (The grammar is ambiguous, which means that we can reduce further to FN, which would be wrong. For simplicity we just skip the wrong steps here, but an unambiguous grammar would not allow these wrong choices.) After that we read the . from the input and try to reduce it, but fail:

```
D

DL

DL .
```

This can't be reduced to anything, so we read the next character from the input: 1. We then reduce that to a D and read the next character, which is 4. 4 can be reduced to D, then to DL, and then the "D DL" sequence can be further reduced to a DL.

```
DL .

DL . 1
DL . D

DL . D 4

DL . D D

DL . D DL

DL . DL
```

Looking at the grammar we quickly note that FN can produce just this "DL . DL" sequence and do a reduction. We then note that FN can be produced from S and reduce the FN to S and then stop, as we've completed the parse.

```
DL . DL

FN

S
```

As you may have noted we could often choose whether to do a reduction now or wait until we had more symbols and then do a different reduction. There are more complex variations on this shift-reduce parsing algorithm, in increasing complexity and power: LR(0), SLR, LALR and LR(1). LR(1) usually needs unpractically large parse tables, so LALR is the most commonly used algorithm, since SLR and LR(0) are not powerful enough for most programming languages.

LALR and LR(1) are too complex for me to cover here, but you get the basic idea.

LL or LR?

This question has already been answered much better by someone else, so I'm just quoting his news message in full here:

```
I hope this doesn't start a war...

First - - Frank, if you see this, don't shoot me.  (My boss is Frank
DeRemer, the creator of LALR parsing...)

(I borrowed this summary from Fischer&LeBlanc's "Crafting a Compiler")

  Simplicity    - - LL
  Generality    - - LALR
  Actions       - - LL
  Error repair  - - LL
  Table sizes   - - LL
  Parsing speed - - comparable (me: and tool-dependent)

Simplicity - - LL wins
==========
The workings of an LL parser are much simpler.  And, if you have to
debug a parser, looking at a recursive-descent parser (a common way to
program an LL parser) is much simpler than the tables of a LALR parser.

Generality - - LALR wins
==========
For ease of specification, LALR wins hands down.  The big
difference here between LL and (LA)LR is that in an LL grammar you must
left-factor rules and remove left recursion.

Left factoring is necessary because LL parsing requires selecting an
alternative based on a fixed number of input tokens.
```

```
Left recursion is problematic because a lookahead token of a rule is
always in the lookahead token on that same rule.  (Everything in set A
is in set A...)  This causes the rule to recurse forever and ever and
ever and ever...

To see ways to convert LALR grammars to LL grammars, take a look at my
page on it:
  http://www.jguru.com/thetick/articles/lalrtoll.html

Many languages already have LALR grammars available, so you'd have to
translate.  If the language _doesn't_ have a grammar available, then I'd
say it's not really any harder to write a LL grammar from scratch.  (You
just have to be in the right "LL" mindset, which usually involves
watching 8 hours of Dr. Who before writing the grammar...  I actually
prefer LL if you didn't know...)


Actions - - LL wins
=======
In an LL parser you can place actions anywhere you want without
introducing a conflict

Error repair - - LL wins
============
LL parsers have much better context information (they are top-down
parsers) and therefore can help much more in repairing an error, not to
mention reporting errors.

Table sizes - - LL
===========
Assuming you write a table-driven LL parser, its tables are nearly half
the size.  (To be fair, there are ways to optimize LALR tables to make
them smaller, so I think this one washes...)

Parsing speed - comparable (me: and tool-dependent)
```

--Scott Stanchfield in article <33C1BDB9.FC6D86D3@scruz.net> on comp.lang.java.softwaretools Mon, 07 Jul 1997.

More information

John Aycock has developed an unusually nice and simple to use parsing framework in Python called SPARK, which is described in his very readable paper.

The definitive work on parsing and compilers is 'The Dragon Book', or Compilers : Principles, Techniques, and Tools, by Aho, Sethi and Ullman. Beware, though, that this is a rather advanced and mathematical book.

A free online alternative, which looks rather good, is this book, but I can't comment on the quality, since I haven't read it yet.

Frank Boumphrey has another EBNF tutorial.

Henry Baker has written an article about parsing in Common Lisp, which presents a simple, high-performant and very convenient framework for parsing. The approach is similar to that of compiler-compilers, but instead relies on the very powerful macro system of Common Lisp.

One syntax for specifying BNF grammars can be found in RFC 2234. Another can be found in the ISO 14977 standard.

Appendices

Acknowledgements

Thanks to:

- Jelks Cabaniss, for encouraging me to turn the news article into a web article, and for providing very useful criticism of the article once it appeared in web form.
- C. M. Sperberg-McQueen for extra historical information about the name of BNF.
- Scott Stanchfield
  for writing the great comparison of LALR and LL. I have asked for permission to quote this, but have received no reply, unfortunately.
- James Huddleston for correcting me on John Backus' name.

Last update 2003-07-21, by Lars M. Garshol.