Lesson #2: BNF

What's It All About?

This lesson presents a brief history of BNF, a few words on grammars and languages, a simple example language, a description of BNF, and a BNF description of the simple example language. Three additional examples are then presented. A BNF sketch of arithmetic expressions in Racket is presented. A BNF sketch of Racket definitions is presented. A BNF sketch of the classic Lisp conditional form, as manifested in Racket, is presented.

Time permitting, a subsequent lesson will discuss variants of BNF.

History of BNF

This brief history takes the form of an enumeration of a few salient events pertinent to the development of BNF, plus a couple of "extras" that single out contributions of John Backus.

1. Fortran was the first high level programming language, designed, developed, and deployed in the 1950s. John Backus was the designer, and led the development team that produced the language at IBM.

2. ALGOL was developed in Europe by a joint committee of European and American computer scientists. As part of the effort, John Backus proposed a mechanism for defining the grammar of Algol 58, which came to be known Backus normal form. Peter Naur revised and expanded Backus's notation for the definition of Algol 60. Donald Knuth suggested that this system for defining syntax be called Backus-Naur form, because, he said, BNF really was not actually a "normal form".

3. The following words, taken from "Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists", summarize Backus's motivation and accomplishment with respect to BNF:

    > Backus liked the ideas embodied by Algol, but felt frustrated by the difficulty of expressing them clearly.
    >
    > > They would just describe stuff in English. Here's this statement – and here's an example. You were hassled in these Algol committees [with unproductive debates over terminology] enough to realize that something needed to be done. You needed to learn how to be precise. ((Backus))
    >
    > To address this problem, Backus applied a formalism called *context-free languages* that had just been invented by linguist Noam Chomsky. Chomsky's work in turn had its roots in Emil Post's theoretical work on rewriting grammars. ((Interestingly, Backus claims to have known nothing about Chomsky's work at the time.))

4. This, again taken from "Out of Their Minds", is an interesting account of the establishment of BNF:

Backus's invention eventually became famous as Backus-Naur form due to a chain of fortuitous events that started with Backus trying to explain his ideas about precise grammars in a paper for the UNESCO meeting on Algol in Paris in June 1959.

> Of course, I had it done too late to be included in the proceedings. So I hand-carried this pile to the meeting. So it didn't get very good distribution. But Peter Naur read it and that made all the difference. ((Backus))

Naur was a Danish mathematician who improved Backus's notation and used it to describe all of Algol. When the programming language community started to experiment with Algol, Naur's manual proved to be the best reference available for describing the language syntax.

5. Backus argued forcefully for functional programming in his 1977 Turing Award lecture entitled "Can Programming Be Liberated from the von Neumann Style?" Influenced by John McCarthy's Lisp and Kenneth Iverson's APL, Backus proposed a language called FP in the lecture, as a vehicle for making his points about functional programming.

Grammars and Languages

A **grammar** is a finite specification of a language. A **language** is a set of strings of symbols defined over some vocabulary of symbols.

How does a grammar specify a language? There are lots of possibilities! English is one possibility, and it works well for some languages. Exhaustive enumeration is another possibility, and it works well for some languages. BNF is yet another possibility, and it works well for some languages.

Think of "the set of all bit strings" and you will be imagining an infinite language defined over just two symbols. The quoted words in the previous sentence serve as an English grammar for this language.

Think of the set of all legal Scrabble words. This is an example of a finite language. The official Scrabble dictionary serves as an exhaustive enumeration grammar for this language.

Think of the set of all Java programs. This is a large, fairly complex language. BNF would be a suitable mechanism by which to craft a grammar for this language.

LEL - Little English Language

It will be useful to have a little language lying around that is just the right size for writing a first BNF description. For now, a very informal description of the language will be presented.

In this language, a *sentence* is a noun phrase, followed by a verb, followed by another noun phrase. A *noun phrase* takes one of two forms, either an article followed by a noun, or an article followed by a qualifier followed by a noun. A *verb* can be the word `kissed` of `hugged` or `chased` or `protected` or `hornswoggled`. A *noun* can be the word `robot` or `baby` or `toddler` or `hat` or `dog`. A *qualifier* can be the word `howling` or `talking` or `dancing` or `barking` or `happy` or `laughing`. An *article* can be the word `a` or `the`.

The previous paragraph can be viewed as an English language grammar for LEL. Can you find your way through all of the words to imagine some sentences in this language? Give it a try. Write down a few sentences in the LEL.

Description of BNF

Descriptions often take the form of a definition, followed by an example, followed by an elaboration of the definition, and that is the form that this description of BNF will take.

Definition of BNF

A BNF grammar consists of four entities:

1. A set of **tokens**, which are considered to be part of the language being defined.
2. A set of **nonterminal symbols**, which are not part of the language being defined, but which play an essential supporting role in defining the language.
3. A set of **productions**, or **rewriting rules**, each of which may be used to map a nonterminal symbol into a string of tokens and nonterminals.
4. A **start symbol**, which is a nonterminal symbol that in some conceptual sense represents the language being defined.

Example BNF Description - LEL

By way of example, these four entities for LEL (the little subset of English featured in the preceding BNF example) would be:

1. tokens = {KISSED, HUGGED, CHASED, PROTECTED, HORNSWOGGLED, ROBOT, BABY, TODLER, HAT, DOG, HOWLING, TALKING, DANCING, BARKING, HAPPY, LAUGHING, A, THE}
2. nonterminals = {S,NP,V,N,Q,A}
3. productions = the following set of rules

```
<S> ::= <NP> <V> <NP>
<NP> ::= <A> <N>
<NP> ::= <A> <Q> <N>
<V> ::= KISSED
<V> ::= HUGGED
<V> ::= CHASED
<V> ::= PROTECTED
<V> ::= HORNSWOGGLED
<N> ::= ROBOT
<N> ::= BABY
<N> ::= TODLER
```

```
<N> ::= HAT
<N> ::= DOG
<Q> ::= HOWLING
<Q> ::= TALKING
<Q> ::= DANCING
<Q> ::= BARKING
<Q> ::= HAPPY
<Q> ::= LAUGHING
<A> ::= A
<A> ::= THE
```

4. start symbol = `S`

Notational Details

In BNF work (for example, when writing productions, drawing parse trees), nonterminals are enclosed in angular brackets. Tokens are written "as is", meaning they are not enclosed in angular brackets. Rules take the form of a nonterminal, followed by the "rewrite symbol", `::=`, followed by a disembodied list of tokens and/or nonterminals.

Additionally, the special nonterminal symbol <empty> is used to stand for the empty string of symbols, and the special symbol | is used to express multiple rules with the same left-hand side in an abbreviated form. For example, using the | symbol, the little grammar for English could be written as:

1. `<S> ::= <NP> <V> <NP>`
2. `<NP> ::= <A> <N>`
3. `<NP> ::= <A> <Q> <N>`
4. `<V> ::= KISSED | HUGGED | CHASED | PROTECTED | HORNSWOGGLED`
5. `<N> ::= ROBOT | BABY | TODDLER | HAT | DOG`
6. `<Q> ::= HOWLING | TALKING | DANCING | BARKING | HAPPY | LAUGHING`
7. `<A> ::= A | THE`

Question: How many BNF rules are encoded in these six lines of BNF text?

Interesting Idea

BNF is generally described in natural language, as it was here. But the BNF grammar can actually be used to describe itself! Just how this might be done is considered on the Wiki page:

`https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form`

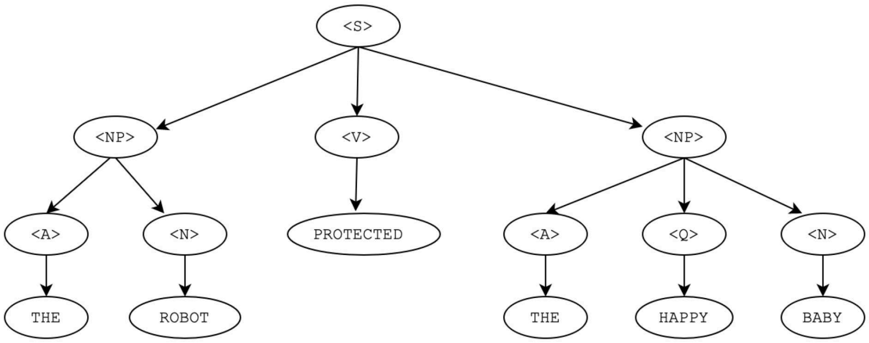
BNF and Language Definition

BNF is a mechanism for writing grammars. And grammars exist to define languages. How does a BNF description define a language?

In BNF, the mechanism of definition, which can be used for the generation or recognition of sentences in the language, is grounded in the notion of "parse trees". The **language defined by a BNF grammar** is the set of all strings of tokens obtained by reading the leaves of some parse tree associated with the grammar from left-to-right.

To create a parse tree for a sentence in a language defined by a BNF grammar, start by writing down the start symbol. Then repeatedly grow the tree by focussing on some nonterminal symbol in the tree, selecting a production for that nonterminal, and adding children to the nonterminal symbol in the tree corresponding to the symbols on the right hand side of the selected rule.

The following example parse tree in the context of the LEL grammar confirms that "THE ROBOT PROTECTED THE HAPPY BABY" is in LEL:



This is such an inflexible language that all of the parse trees for sentences in the language are structured in just three ways. The only differences beyond these few structural variations are found in the rendering of tokens.

Important Note on Programming Language Descriptions

Descriptions of the syntax of programming languages are generally, by convention, decomposed into two levels of description. The **phrase structure** elements of the language are generally described in terms of BNF, or perhaps a variant of CFG (context free grammar). The **lexical items** of the language are generally described using some other mechanism, perhaps regular expressions.

Consequently, when describing a phrase structure using BNF, the BNF description tends to leave nonterminal symbols which correspond to lexical level entities **dangling**. If you are drawing a parse tree corresponding to some entity and you find yourself at one of those **dangling** nonterminals, just draw the desired lexical entity (i.e., token) as the sole child of the **dangling** nonterminal, indicating in some manner that you are breaking out of the BNF formalism. (I tend to do this by employing an alternative representation for the token node, or sometimes by employing a different looking arrow from the **dangling** nonterminal to the token.)

Moreover we sometimes leave nonterminals **dangling** for reasons other than the conventional one (that they stand for something at the lexical level of the language). In such cases, simply take the liberty of extending the parse tree in a non-rigorous manner, drawing token leaves below that **dangling** node, somehow indicating that you are breaking from the BNF formalism at that point.

A BNF Grammar for Arithmetic Expressions in Racket

Simplified BNF grammar for Arithmetic Expressions in Racket.

1. `<ae> ::= ( <op> <aes> ) | <num> | <var>`
2. `<op> ::= + | - | * | /`
3. `<aes> ::= <empty> | <ae> <aes>`

Why is this a “simplification”? Because there are more than four arithmetic operators in Racket.

Parse Tree Drawing Exercise: Arithmetic Expressions

Draw a parse tree for each of the following arithmetic expressions, assuming the previously given BNF description of arithmetic expressions.

1. `100`
2. `( + 33 55 88 )`
3. `( / ( + a b ) 2 )`

A BNF Grammar for Color Fun

```
<color-fun> ::= <add-cmd> | <colors-cmd> | <show-cmd> | <describe-cmd> | <exit-cmd>
<add-cmd> ::= add <color> <id> | add color <id>
<colors-cmd> ::= colors
<show-cmd> ::= show <id>
<describe-cmd> ::= describe <id>
<exit-cmd> ::= exit
<color> ::= ( <rgb> <rgb> <rgb> ) | ( <rgb> <rgb> <rgb> <rgb> )
```

Parse Tree Drawing Exercise: Color Fun

Draw a parse tree for each of the following Color Fun commands, assuming the previously given BNF description of the Color Fun grammar.

1. `colors`
2. `show purple`
3. `add ( 100 220 170 ) favorite-color`

BNF Grammar Writing Exercise #1: QN (Quaternary Numbers)

Consider language QN to be the set of all unsigned quaternary numbers (numbers composed of the symbols 0, 1, 2 and 3) with no leading zeros (that is, no zeros at the start of the number, except for the number 0). Examples:

1. `0`
2. `1001`
3. `123000`
4. `1020302010`

Your tasks ...

1. Write a BNF grammar for this language. Constraint: Refine all nonterminal symbols.
2. Draw a parse tree for the following sentence: `0`
3. Draw a parse tree for the following sentence: `30201`

BNF Grammar Writing Exercise #2: AE (Arithmetic Expressions)

Consider language AE consisting of arithmetic expressions involving fully parenthesized operations featuring the four basic operators and operands which take the form of numbers or identifiers (variables). Examples:

1. `64`
2. `( 44 - 151 )`
3. `( pi * ( radius * radius ) )`
4. `( ( -1005.03 * x ) + ( 212.4 * y ) )`

Note that the language define here is a simple subset of what might typically be imagined as a language of arithmetic expressions.

Your tasks ...

1. Write a BNF grammar for this language. Constraint: Refine all nonterminal symbols **except** for numbers and identifiers.
2. Draw a parse tree for the following sentence: `23`
3. Draw a parse tree for the following sentence: `( 33 - 55 )`
4. Draw a parse tree for the following sentence: `( pi * ( 55 * 55 ) )`

BNF Grammar Writing Exercise #3: Dots

Consider the language Dots consisting of arbitrarily long sequences of zero or more repetitions of a color token followed by a string of one or more DOT tokens, where the color tokens are drawn from the set {RED, YELLOW, BLUE, ORANGE, GREEN, PURPLE, WHITE, BLACK}. Examples:

1. `BLUE DOT DOT WHITE DOT PURPLE DOT DOT DOT`
2. `GREEN DOT`
3. `BLACK DOT RED DOT DOT BLUE DOT DOT DOT`

Your tasks ...

1. Write a BNF grammar for this language. Constraint: Refine all nonterminal symbols.
2. Draw a parse tree for the following sentence: `GREEN DOT`
3. Draw a parse tree for the following sentence: `BLUE DOT DOT DOT WHITE DOT BLUE DOT DOT DOT`

A BNF Sketch of Racket Definitions

Previously, the syntax for the mechanism of definition in Racket was introduce informally, for both variable definition and function definition. Here is a sketch of it in BNF:

1. `<definition> ::= <var-def> | <fun-def>`
2. `<var-def> ::= ( define <id> <form> )`
3. `<fun-def> ::= ( define <header> <body> )`
4. `<header> ::= ( <fun-name> <parameters> )`
5. `<fun-name> ::= <id>`
6. `<parameters> ::= <ids>`
7. `<ids> ::= <empty> | <id> <ids>`
8. `<body> ::= <form> <forms>`
9. `<forms> ::= <empty> | <form> <forms>`

Why is this a “sketch”? Because neither `<id>` nor `<form>` were refined, for one reason or another.

Parse Tree Drawing Exercise: Racket Definitions

Draw a parse tree for each of the following Racket definition, assuming the previously given BNF description of Racket definitions.

1. `( define perfect 496 )`
2. `( define ( square x ) ( * x x ) )`

A BNF Sketch of COND in Racket

In Lisp, the `cond` form is the most essential selection construct. Sometimes the `if` form is nice to use too, but `cond` is classic Lisp. Here is a BNF sketch of a simplification of the `cond` form:

1. `<cond> ::= ( cond <case> <cases> )`
2. `<case> ::= ( <situation> <action> )`
3. `<cases> ::= <empty>` | <case> <cases>
4. `<situation> ::= <form>`
5. `<action> ::= <form> <forms>`
6. `<forms> ::= <empty> | <form> <forms>`

Why is this a "sketch"? Because `<form>` is not refined. Why is it a "simplification"? Because not all of the options for various nonterminals are provided.

Informal Parsing Exercise: The COND in Racket

Preliminary note: This exercise does not involve drawing explict parse trees! Rather, it calls upon you to informally (mentally) parse some forms according to the given BNF for the conditional construct in Racket.

For each of the following examples of the **cond** from in Racket, how many instances of the **case** construction do you find embedded in the example form?

1. Example 1

```
( cond
  ( ( not ( equal? input '( end ) ) )
    ( interpret input )
    ( color-fun )
  )
)
```

2. Example 2

```
( define ( interpret input )
 ( cond
  ( ( = ( length input ) 1 ) ( interpret-1 input ) )
  ( ( = ( length input ) 2 ) ( interpret-2 input ) )
  ( ( = ( length input ) 3 ) ( interpret-3 input ) )
  ( ( = ( length input ) 5 ) ( interpret-5 input ) )
  ( ( = ( length input ) 6 ) ( interpret-6 input ) )
 )
)
```

3. Example 3

```
( cond
 ( ( equal? input ’( colors ) )
  ( display-colors the-colors )
  ( nl )
 )
 ( ( equal? input ’( help ) )
  ( help )
 )
 ( else
  ( display "CF: Unrecognizable command: " )
  ( write input )
  ( nl )
 )
)
```

Syntax and Semantics

Just thinking, after all of this, that it might be nice to actually define “syntax” and “semantics” with respect to programming languages.

- The **syntax** of a programming language is the set of rules governing how tokens of the language are arranged.
- The **semantics** of a programming language refers to the manner in which arrangements of tokens behave.