Maggie Johnson
CS109

Context-Free Grammars

Key Topics

* Introduction
* Context Free Grammars
* Parse Trees

Introduction

As you know, the earliest programs were written in either machine or assembly language since these were the only languages that computers directly understood. Programming in these languages was (and is) quite cumbersome and prone to error, so early researchers developed English-like high-level languages which were easier for human programmers to work with. Unfortunately, computers still understood only the lower-level languages. So, programs written in high-level languages had to be translated back into a low-level language before they could be executed. The designers of the first high-level languages realized that the problem of translation is analogous to the problems humans face every day when they decipher the sentences that they hear and read in English.

Elementary schools used to be called Grammar schools because one of the most important subjects taught was English grammar. A **grammar** is a set of rules by which valid sentences in a language are constructed. Our ability to understand the meaning of a sentence depends on our ability to understand how it was formed from the rules of grammar. Determining how a sentence can be formed from the rules is called **parsing**.

There are two sets of rules in English grammar. One set is the **semantic** rules which are concerned only with the meaning of the sentence. The other set is the **syntax** rules which are concerned only with the form of the sentence. The following sentences are syntactically valid in English but are not semantically valid:

Trees fly. Fred melts. Cars swim.

Some of the rules of English grammar are:

1. A sentence can be a subject followed by a predicate.
2. A subject can be a noun-phrase.
3. A noun-phrase can be an adjective followed by a noun-phrase
4. A noun-phrase can be an article followed by a noun-phrase.
5. A noun-phrase can be a noun.
6. A predicate can be a verb followed by a noun-phrase.
7. A noun can be: apple | bear | cat | dog
8. A verb can be: eats | follows | gets | hugs
9. An adjective can be: itchy | jumpy
10. An article can be: a | an | the

Is there anything wrong with this grammar? _____
How might you fix it? _____

Now, consider the following sentence: *The itchy bear hugs the jumpy dog*. The method by which this sentence can be generated is illustrated below:

Rule		
1	1. sentence ->	subject predicate
2	2.	noun-phrase predicate
6	3.	noun-phrase verb noun-phrase
4	4.	article noun-phrase verb noun-phrase
3	5.	article adjective noun-phrase verb noun-phrase
5	6.	article adjective noun verb noun-phrase
4	7.	article adjective noun verb article noun-phrase
3	8.	article adjective noun verb article adjective noun-phrase
5	9.	article adjective noun verb article adjective noun
10	10.	*the* adjective noun verb article adjective noun
9	11.	*the itchy* noun verb article adjective noun
8	12.	*the itchy bear* verb article adjective noun
7	13.	*the itchy bear hugs* article adjective noun
10	14.	*the itchy bear hugs the* adjective noun
9	15.	*the itchy bear hugs the jumpy* noun
7	16.	*the itchy bear hugs the jumpy dog*

Notice that all we did to generate the sentence was start at the highest level and do a series of substitutions. We began by substituting for the underlined symbols, and then once we got the appropriate series of symbols, we began substituting actual words of the underlined symbol type. The underlined symbols are called **nonterminals** meaning that these symbols can be replaced by other symbols or actual words. The actual words are called **terminals** meaning that they cannot be replaced by anything else. Once we have substituted "bear", we are stuck with it; there is no rule that allows us to substitute for "bear".

Midway through this production procedure, we developed the sentence into as many nonterminals as we could. This represents the **grammatical parse** of the sentence. After that point, all we did was substitute terminals.

Notice that the subset of grammar rules given above are recursive in nature. For example, we know that a noun-phrase can be an adjective followed by a noun-phrase. This leads to the following:

noun-phrase	->	adjective noun-phrase
		adjective adjective noun-phrase
		adjective adjective adjective noun-phrase
		adjective adjective adjective noun

We could get 20 adjectives out of this if we wanted.

So far, we have been dealing only with syntactically correct sentences; semantics has not entered into the picture. This is exactly what we mean by the term **formal language**: the sentence is grammatically correct according to the rules; we don't care if it does not make sense.

Context-Free Grammars

The sequence of applications of the rules that produces a finished string of terminals is called a **derivation**. The grammar rules are often called **productions**; they indicate the

possible substitutions. Note that a derivation may or may not be unique; this means that we might start with an initial symbol (or **start symbol**) and apply productions in more than one different way to generate the same finished product. Also, note that often we do the derivation in a particular way: from left to right, .i.e, a **leftmost derivation** is one where the leftmost nonterminal is always the one to be substituted.

This whole structure we have been discussing is called a context-free grammar. The idea was invented by Noam Chomsky in 1956; he created several mathematical models for languages. CFG's are of primary importance in computer science because procedural programming languages can be represented with CFG's. Thus, they play an important role in the design of compilers.

A **context-free grammar** (**CFG**) has:

1) An alphabet $\sum$ of symbols called **terminals**, from which we make strings that will be the words of the language.
2) A set of symbols called **nonterminals**, one of which is S, the **start symbol**.
3) A finite set of **productions** of the form:

one nonterminal -> finite string of terminals &/or nonterminals &/or the empty string

One of these productions must have S on its left side.

Just to be consistent, we will always indicate terminals in lowercase (and special symbols) and nonterminals in uppercase.

Example 1

The only terminal in the grammar is "a", and the productions are:

1) S -> aS
2) S -> ∂

If we apply production 1 four times and then apply production 2:

S -> aS
aaS
aaaS
aaaaS
aaaa∂ = aaaa

This is a derivation of a^4 in this grammar. In fact, this grammar generates the language represented by the regular expression **a***.

During the course of applying productions, we generate what are called **working strings** which are the intermediate steps in a derivation. Notice also that ∂ plays a special role in a CFG. It is not a nonterminal because we could never have the following production:

∂ -> something

It is also not a terminal because it just vanishes from the final string. We will consider ∂ a special symbol. If we have a production

N -> ∂

this production allows us to delete N from a string at any point in a derivation.

Example 2

terminals: a, b; nonterminals: S, X, Y; productions:

S -> X
S -> Y
X -> ∂
Y -> aY
Y -> bY
Y -> a
Y -> b

All the words in this language are either of type X or type Y. The only possible continuation for type X is ∂ so ∂ is the only word of type X. Words of type Y can be any string of a's and b's except the null string. So, the two types together give us: (**a** + **b**)*.

Notice that we could have used the "|" sign which means "or" to write the productions in this grammar:

S -> X | Y
X -> ∂
Y -> aY | bY | a | b

This whole format for representing a grammar (arrows, vertical bars, terminals and nonterminals) is called **BNF** or **Backus Normal Form** or **Backus-Naur Form.** It was invented by John Backus and Peter Naur.

What language is represented by the following grammar? _____

S -> aB | bA
A -> a | aS | bAA
B -> b | bS | aBB

Example 3

An example of a CFG to represent fully parenthesized arithmetic expressions in a programming language:

S -> AE
AE -> (AE+AE) | (AE-AE) | (AE*AE) | (AE/AE) | (AE) | -(AE) | NUMBER
NUMBER -> DIGIT
NUMBER -> NUMBER DIGIT
DIGIT - > 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Parse Trees

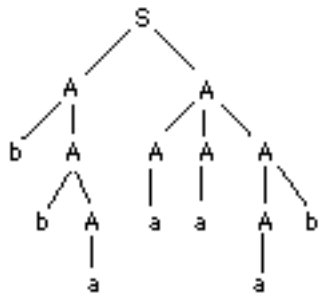
Parse trees are used to illustrate the steps in parsing a sentence. In the context of CFG's, parse trees are sometimes called **syntax trees** or **production trees** or **derivation trees**.

Example 4

Consider the CFG:

S -> AA
A -> AAA | bA | Ab | a

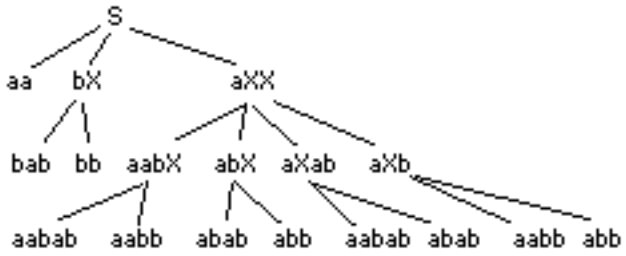
A parse tree of the string bbaaaab:



The only rule for formation of parse trees is every nonterminal sprouts branches leading to every symbol in the right side of the production that replaces it.

A **total language tree** is a parse tree that shows all the words in the language. It differs from a parse tree in that the children of a node are the right side of the production, not each individual symbol:

S -> aa | bX | aXX
X -> ab | b

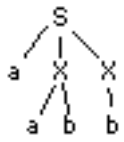


What characteristics of the productions of a grammar indicate the corresponding language is finite? _____

There are only 7 words in this language. A CFG is called **ambiguous** if for at least one word in the language that it generates, there are two possible derivations of the word that correspond to *different* parse trees.

Is the CFG above ambiguous? _____

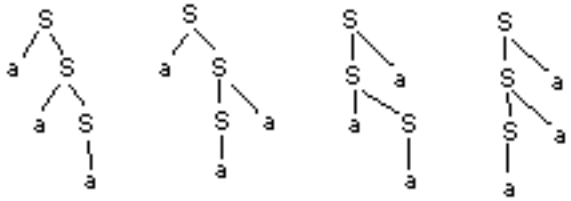
Consider the word "aabb". The parse tree for this word:



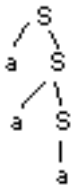
This is the only possible parse tree for "aabb" but you can get to the word in two different ways in the same tree: aXb or aabX. Thus, the CFG is not ambiguous.

An example of an ambiguous CFG is:S -> aS | Sa | a

The word "aaa" can be generated by four different trees:



We could redefine the grammar as: S -> aS | a. This removes the ambiguity so "aaa" has only the following parse tree:



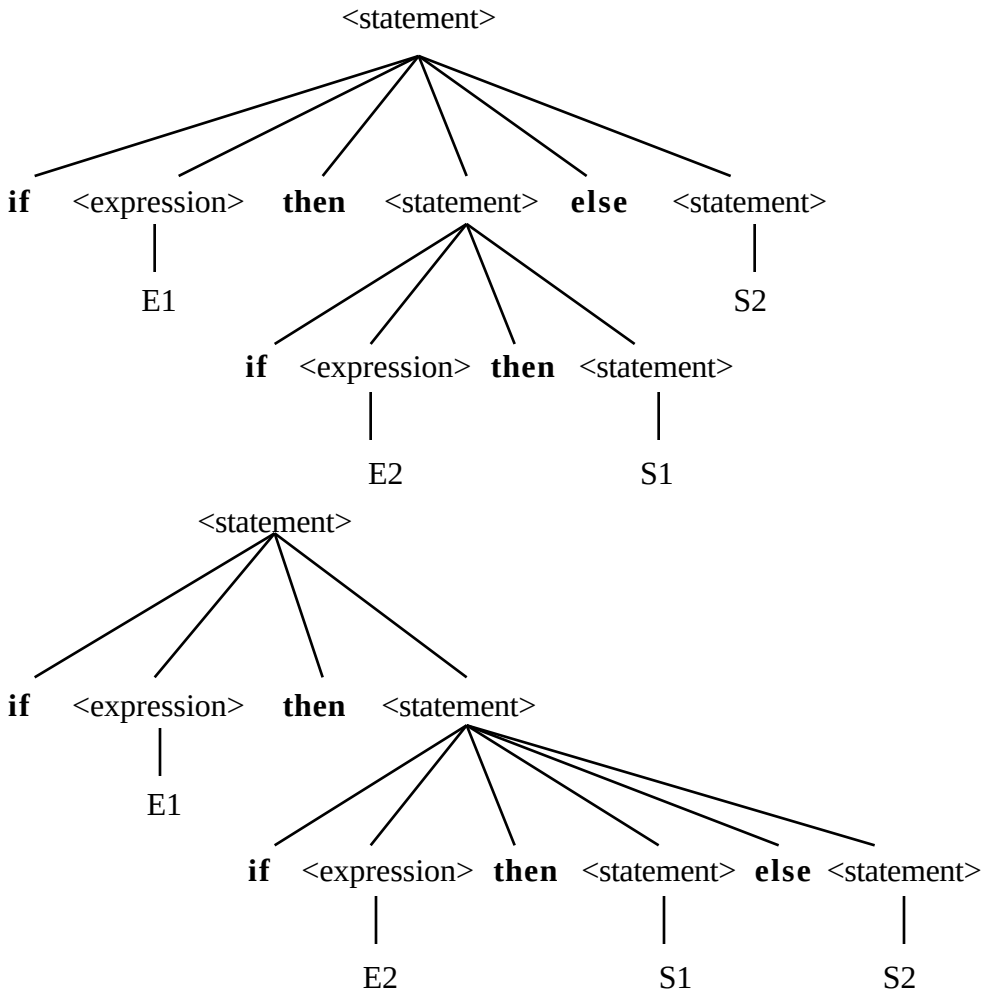
A very important example of an ambiguous grammar and your first look at a CFG for a programming language (Pascal): consider this fragment of the Pascal syntax:

<*statement*> -> **if** <*expression*> **then** <*statement*> |
if <*expression*> **then** <*statement*> **else** <*statement*> |
.....

Can we find a derivation of the following bit of code using this grammar, assuming reasonable definitions for <*expression*> and <*statement*>?

```
if p <> Nil then
   if p^.next = nil then
      writeln('Next is nil')   (* S1 *)
   else
      p := p^.next;            (* S2 *)
```

In fact, we can find not only one derivation, but two:



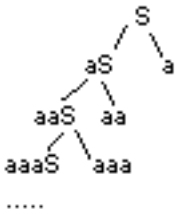
Not only did we find two derivations, but we found two different derivation trees - ambiguity. This should be somewhat alarming when you consider that a compiler has to decide which to use; and the two different derivations result in different interpretations. If the compiler uses the second derivation then nothing will happen if p is nil. If it uses the first derivation, then if p is nil the program will blow up. (Real compilers avoid this by assuming rules about the binding of **if** statements or by using the fix shown below. Smart programmers avoid these problems altogether by using **begin** and **end** to avoid ambiguity).

Ambiguity also exists in English because English grammar is ambiguous. Consider the sentence "I went to a movie with Robin Williams." Does this mean I saw a movie in which Mr. Williams performed, or did he accompany me to the theater? The problem is that the prepositional phrase "with Robin Williams" could modify either the verb "went" or the object "movie". This is similar to the problem we were having in the Pascal example: trying to determine what the "else" modified.

It is sometimes (but not always) possible to rewrite a grammar to remove ambiguity. We can modify the grammar for if statements to bind an **else** to the closest unmatched **then**.using this grammar:

<*statement*> -> <*matched*> | <*unmatched*>
<*matched*> -> **if** <*expression*> **then** <*matched*> **else** <*matched*> |
.....
<*unmatched*> -> **if** <*expression*> **then** <*statement*> |
if <*expression*> **then** <*matched*> **else** <*unmatched*>

Returning to the topic of total language trees:
Finally, note that certain grammars cannot be represented as total language trees because they represent infinite languages. The grammar above is an infinite language representing **a**+: any number of a's not including the null string. An attempt at a total language tree looks like this:



.....

We have shown that the class of regular languages is very restrictive; we cannot express all the languages that we need with regular expressions. Above, it was shown that many regular languages can be represented by CFG's. In fact, CFG's can handle more than just regular languages; CFG's are powerful enough for programming languages. For example, recall the language L = $\{a^n b^n$ for n = 0, 1, 2, 3...$\}$ which we proved was nonregular:

L = {∂, ab, aabb, aaabbb, aaaabbbb, ...} or L = $\{a^n b^n$ for n = 0, 1, 2, 3...$\}$

What is the CFG for this language? _____

Bibliography & Historical Notes

* The term "compiler" was coined in the early 1950's by Grace Hopper. Translation was then viewed as the "compilation" of a sequence of subprograms selected from a library. Compilation as we now know it was then called "automatic programming" and there was almost universal skepticism that it would ever be successful. The first real compilers in the modern sense were the FORTRAN compilers of the late 1950s. The very first one took 18 person-years to write.

J. Backus, "The FORTRAN Automatic Coding System," *Proceedings AFIPS Western Joint Conference*, Baltimore: Spartan Books, 1957.

* Context Free Grammars were first studied by Noam Chomsky as a formalism for describing natural languages. Similar formalisms were used to define the syntax of two early programming languages (Fortran by Backus, and Algol-60 by Naur). As a result, CFG's are often referred to as Backus-Naur Form (BNF); The relationship between CFG's and BNF's was perceived in Ginsburg and Rice. The study of CFG's through their mathematical properties begins with Bar-Hillel, Perles and Shamir; an advanced book on this area is Ginsburg. For a thorough study of CFG's and their applications, refer to Hopcroft and Ullman, or Aho, Sethi and Ullman.

A. Aho, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Reading,

MA: Addison-Wesley, 1986.

J. Backus, "The FORTRAN Automatic Coding System," *Proceedings AFIPS Western Joint Conference*, Baltimore: Spartan Books, 1957.

V. Bar-Hillel, M. Perles, E. Shamir, "On Formal Properties of Simple Phrase Structure Grammars," *Zeitschrift für Phonetik, Sprachwissenschaft, und Kommunikationsforschung*, 14 (1961), 143-172.

N. Chomsky, "Three Models for the Description of Language," *IRE Transactions on Information Theory*, IT-2:3 (1956) 113-124.

S. Ginsburg, *The Mathematical Theory of Context-Free Language*s, New York: McGraw-Hill, 1966.

S. Ginsburg, H. Rice, "Two Families of Languages Related to Algol," *Journal of ACM*, 9:3 (1962), 350-371.

J. Hopcroft, J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading, MA: Addison-Wesley, 1979.

P. Naur (ed.), "Revised Report on the Algorithmic Language Algol 60," *Communications of the ACM*, 6:1 (1963), 1-17.

* Ambiguity in CFG's was first studied in:

R. Floyd, "On Ambiguity in Phrase Structure Languages," *Communications of the ACM* 5 (1962), 526-534.

D. Cantor, "On the Ambiguity Problem of Backus Systems,"*Journal of the ACM*, 9:4 (1962), 477-479.