COMS W3261
Computer Science Theory
Chapter 5 Notes

Alexander Roth

2014–09–25

Context-Free Grammar and Languages

"Context-Free languages" are languages that have a natural, recursive notation, called "context-free grammars."

Context-Free Grammars

An Informal Example

Let us consider the language of palindromes. String w is a palindrome if and only if $w = w^R$. It is easy to verify that the language L_{pal} of palindromes of 0's and 1's is not a regular language. To do so, we use the pumping lemma. If L_{pal} is a regular language, let n be the associated constant, and consider the palindrome $w = 0^n10^n$. If L_{pal} is regular, then we can break w into $w = xyz$, such that y consists of one or more 0's from the first group. Therefore xz cannot be a palindrome. We have now contradicted the assumption that L_{pal} is a regular language.

There is a natural, recursive definition of when a string of 0's and 1's is in L_{pal}. It starts with a basis saying that a few obvious strings are in L_{pal}, and then exploits the idea that if a string is a palindrome, it must begin and end with the same symbol. Further, when the first and last symbols are removed, the resulting string must also be a palindrome.

A context-free grammar is a formal notation for expressing such recursive definitions of languages. A grammar consists of one or more variables that represent classes of strings, i.e., languages.There are rules that say how the strings in each class are constructed. The construction can use symbols of the alphabet, strings that are already known to be in one of the classes, or both.

Definition of Context-Free Grammars

There are four important components in a grammatical description of a language:

1. There is a finite set of symbols that form the strings of the language being defined. We call this alphabet the *terminals* or *terminal symbols*.

2. There is a finite set of *variables*, also called sometimes *nonterminals* or *syntactic categories*. Each variable represents a language, i.e., a set of strings.

3. One of the variables represents the language being defined; it is called the *start symbol*. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol.

4. There is a finite set of *productions* or *rules* that represent the recursive definition of a language. Each production consists of:

   (a) A variable that is being (partially) defined by the production. This variable is often called the *head* of the production.

   (b) The production symbol $\rightarrow$.

   (c) A string of zero or more terminals and variables. This string, called the *body* of the production, represents one way to form strings in the language of the variable of the head. In so doing, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of that variable.

The four components just described form a *context-free grammar* or just *grammar*, or *CFG*. We shall represent a CFG G by its four components, that is $G = (V, T, P, S)$, where V is the set of variables, T the terminals, P the set of productions, and S the start symbol.

Compact Notation for Productions

It is convenient to think of a production as "belonging" to the variable of its head. We shall often use remakes like "the productions for A" or "A-productions" to refer to the productions whose head is variable A. We may write the productions for a grammar by listing each variable once, and then listing all the bodies of the productions for that variable, separated by vertical bars. That is, the productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ..., $A \rightarrow \alpha_n$ can be replaced by the notation $A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_n$.

Derivations Using a Grammar

We apply the productions of a CFG to infer that certain strings are in teh language of a certain variable. There are two approaches to this inference. The more conventional approach is to use the rules from body to head. That is, we

take strings known to be in the language of each of the variables of the body, concatenate them, in the proper order, with any terminals appearing in the body, and infer that the resulting string is in the language of the variable in the head. This procedure is known as *recursive inference*.

There is another approach to defining the language of a grammar, in which we use the productions from head to body. We expand the start symbol using one of its productions. We further expand the resulting string by replacing one of the variables by the body of one of its productions, and so on, until we derive a string consisting entirely of terminals. The language of the grammar is all strings of terminals that we can obtain this way. This use of grammars is called *derivation*.

The process of deriving strings by applying productions from head to body requires the definition of a new relation symbol $\Rightarrow$. Suppose $G = (V, T, P, S)$ is a CFG. Let $\alpha AB\beta$ be a string of terminals and variables, with A a variable. That is, α and β are strings in $(V \cup T)^*$, and A is in V. Let $A \to \gamma$ be a production of G. Then we say $\alpha A\beta \underset{G}{\Rightarrow} \alpha\gamma\beta$. If G is understood, we say $\alpha A\beta \Rightarrow \alpha\gamma\beta$. Notice that one derivation step replaces any variable anywhere in the string by the body of one of its productions.

We may extend the $\Rightarrow$ relationship to represent zero, one, or many derivation steps, much as the transition function δ of a finite automaton was extended to $\hat{\delta}$. For derivations, we use a * to denote "zero or more steps."

Leftmost and Rightmost Derivations

In order to restrict the number of choices we have in deriving a string, it is often useful to require that at each step we replace the leftmost variable by one of its production bodies. Such a derivation is called a *leftmost derivation*, and we indicate that a derivation is leftmost by using the relations $\underset{lm}{\Rightarrow}$ and $\overset{*}{\underset{lm}{\Rightarrow}}$, for one or many steps, respectively. If the grammar G that is being used is not obvious, we can place the name G below the arrow in either of these symbols.

Similarly, it is possible to require that at each step the rightmost variable is replaced by one of its bodies. If so, we call the derivation *rightmost* and use the symbols $\underset{rm}{\Rightarrow}$ and $\overset{*}{\underset{rm}{\Rightarrow}}$ to indicate one or many rightmost derivation steps, respectively. Again, the name of the grammar may appear below these symbols if it is not clear which grammar is being used.

Any derivation has an equivalent leftmost and an equivalent rightmost derivation. That is, if w is a terminal string, and A a variable, then $A \overset{*}{\Rightarrow} w$ if and only if $A \overset{*}{\underset{lm}{\Rightarrow}} w$, and $A \overset{*}{\Rightarrow} w$ if and only if $A \overset{*}{\underset{rm}{\Rightarrow}}$.

The Language of a Grammar

If $G = (V, T, P, S)$ is a CFG, the *language of* G, denoted $L(G)$, is the set of terminal strings that have derivations from the start symbol. That is,

$$L(G) = \{w \text{ in } T^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

If a language L is the language of some context-free grammar, then L is said to be a *context-free language*, or CFL.

Sentintial Forms

Derivations from the start symbol produce strings that have a special role. We call these "sentential forms." That is, if $G = (V, T, P, S)$ is a CFG, then any string α in $(V \cup T)^*$ such that $S \stackrel{*}{\Rightarrow} \alpha$ is a *sentential form*. If $S \underset{lm}{\stackrel{*}{\Rightarrow}} \alpha$, then α is a *left-sentential form*, and if $S \underset{rm}{\stackrel{*}{\Rightarrow}} \alpha$, then α is a *right-sentential form*. Note that the language $L(G)$ is those sentential forms that are in T^*; i.e., they consist solely of terminals.

The Form of Proofs About Grammars

We first develop an inductive hypothesis that states what properties the strings derived from each variable have.

We prove the "if" part: that if a string w satisfies the informal statement of one of the variables A, then $A \stackrel{*}{\Rightarrow} w$. Typically, we prove the "if" party by induction on the length of w. If there are k variables, then the inductive statement to be proved has k parts, which must be proved as a mutual induction.

We must also prove the "only-if" part, that if $A \stackrel{*}{\Rightarrow} = w$, then w satisfies the informal statement about strings derived from variable A. The proof of this part is typically by induction on the number of steps in the derivation. If the grammar has productions that allow two or more variables to appear in derived strings, then we shall have to break a derivation of n steps into several parts, one derivation from each of the variables. These derivations may have fewer than n steps, so we have to perform an induction assuming the statement for all values n or less.

Parse Trees

There is a tree representation for derivations that has proved extremely useful. This tree shows us clearly how the symbols of a terminal string are grouped into substrings, each of which belongs to the language of one of the variables of the grammar. But perhaps more importantly, the tree known as a "parse tree" when used in a compiler, is the data structure of choice to represent the source program.

We introduce the parse tree and show that the existence of parse trees is tied closely to the existence of derivations and recursive inferences. Certain grammars allow a terminal string to have more than one parse tree. That situation makes the grammar unsuitable for a programming language, since the compiler could not tell the structure of certain source programs, and therefore could not with certainty deduce what the proper executable code for the program was.

Constructure Parse Trees

Let us fix on a grammar $G = (V, T, P, S)$. The *parse tree* for G are trees with the following conditions:

1. Each interior node is labeled by a variable in V.
2. Each leaf is labeled by either a variable, a terminal, or ϵ. However, if the leaf is labeled ϵ, then it must be the only child of its parent.
3. If an interior node is labeled A, and its children are labeled

   $$X_1, X_2, \ldots, X_k$$

   respectively, from the left, then $A \to X_1 X_2 \cdots X_k$ is a production in P. Note that the only time one of the X's can be ϵ is if that is the label of the only child, and $A \to \epsilon$ is a production of G.

The Yield of a Parse Tree

If we look at the leaves of any parse tree and concatenate them from the left, we get a string, called the *yield* of the tree, which is always a string that is derived from the root variable. The fact that the yield is derived from the root will be proved shortly. Of special importance are those parse trees such that:

1. The yield is a terminal string. That is, all leaves are labeled either a terminal or with ϵ.
2. The root is labeled by the start symbol.

These are the parse trees whose yields are strings in the language of the underlying grammar.

Review of Tree Terminology

- Trees are collections of *nodes*, with a *parent-child* relationship. A node has at most one parent, drawn above the node, and zero or more children, drawn below. Lines connect parents to their children.
- There is one node, the *root*, that has no parent; this node appears at the top of the tree. Nodes with no children are called *leaves*. Nodes that are not leaves are *interior nodes*.
- A child of a child of a $\cdots$ node is a *descendant* of that node. A parent of a parent of a $\cdots$ is an *ancestor*. Trivially, nodes are ancestors and descendants of themselves.
- The children of a node are ordered "from the left," and drawn so. If node N is to the left of node M, then all the descendants of N are considered to be to the left of all the descendants of M.

Inference, Derivations, and Parse Trees

Each of the ideas that we have introduced so far for describing how a grammar works gives us essentially the same facts about strings. That is, given a grammar $G = (V, T, P, S)$, we shall show that the following are equivalent:

1. The recursive inference procedure determines that terminal string w is in the language of variable A.
2. $A \stackrel{*}{\Rightarrow} w$.
3. $A \underset{lm}{\stackrel{*}{\Rightarrow}} w$.
4. $A \underset{rm}{\stackrel{*}{\Rightarrow}} w$.
5. There is a parse tree with root A and yield w.

In fact, except for the use of recursive inference, which we only defined for terminal strings, all the other conditions – the existence of derivations, leftmost or rightmost derivations, and parse trees – are also equivalent if w is a string that has some variables.

From Inferences to Trees

Theorem 1. *Let $G = (V, T, P, S)$ be a CFG. If the recursive inference procedure tells us that terminal string w is in the language of variable A, then there is a parse tree with root A and yield w.*

From Tree to Derivations

Theorem 2. *Let $G = (V, T, P, S)$ be a CFG, and suppose there is a parse tree with root labeled by variable A and with yield w, where w is in T^*. Then there is a leftmost derivation $A \underset{lm}{\stackrel{*}{\Rightarrow}} w$ in grammar G.*

Theorem 3. *Let $G = (V, T, P, S)$ be a CFG, and suppose there is a parse tree with root labeled by variable A and with yield w, where w is in T^*. Then there is a leftmost derivation $A \underset{rm}{\stackrel{*}{\Rightarrow}} w$ in grammar G.*

From Derivations to Recursive Inferences

Suppose that we have a derivation $A \Rightarrow X_1X_2 \cdots X_k \stackrel{*}{\Rightarrow} w$. Then we can break w into pieces $w = w_1w_2 \cdots w_k$ such that $X_i \stackrel{*}{\Rightarrow} w_i$. Note that if X_i is a terminal, then $w_i = X_i$, and the derivation is zero steps.

Theorem 4. *Let $G = (V, T, P, S)$ be a CFG, and suppose there is a derivation $A \underset{G}{\stackrel{*}{\Rightarrow}} w$, where w is in T^*. Then the recursive inference procedure applied to G determines that w is in the language of variable A.*

Applications of Context-Free Grammars

1. Grammars are used to describe programming languages. More importantly, there is a mechanical way of turning the language description as a CFG into a parser, the component of the compiler that discovers the structure of the source program and represents that structure by a parse tree.

2. The development of XML (Extensible Markup Language) is widely predicted to facilitate electronic commerce by allowing participants to share conventions regarding the format of orders, product descriptions, and many other kinds of documents. An essential part of XML is the *Document Type Definition* (DTD), which is essentially a context-free grammar that describes the allowable tags and the ways in which these tags may be nested.

Parsers

Many aspects of a programming language have a structure that may be described by regular expressions. However, there are also some very important aspects of typical programming languages that cannot be represented by regular expressions alone.

The YACC Parser-Generator

The generation of a parser has been institutionalized in the YACC command that appears in all UNIX systems. The input to YACC is a CFG, in a notation that differs only in details from the one we have used here. Associated with each production is an *action*, which is a fragment of C code that is performed whenever a node of the parse tree that corresponds to this production is created. Typically, the action is code to construct that node, although in some YACC applications the tree is not actually constructed, and the action does something else, such as emit a piece of object code.

Markup Languages

We shall next consider a family of "languages" called *markup* languages. The "strings" in these languages are documents with certain marks (called *tags*) in them. Tags tell us something about the semantics of various strings within the document.

The markup language with which you are probably most familiar is HTML (HyperText Markup Language). This language has two major functions: creating links between documents and describing the format of a document.

XML and Document-Type Definitions

The fact that HTML is described by a grammar is not in itself remarkable. Essentially all programming languages can be described by their own CFG's, so it would be more surprising if we could *not* so describe HTML. However, when we look at another important markup language, XML, we find that CFG's play a more vital role, as part of the process of using that language.

The purpose of XML is not to describe the formatting of the document, that is the job for HTML,. Rather, XML tries to describe the "semantics" of the text.

To make clear what the different kinds of tags are, and what structures may appear between matching pairs of these tags, people with a common interest are expected to develop standards in the form of a DTD (Document-Type Definition).

A DTD is essentially a context-free grammar, with its own notation for describing the variables and productions.

Ambiguity in Grammars and Languages

As we have seen, applications of CFG's often rely on the grammar to provide the structure of files. The tacit assumption was that a grammar uniquely determines a structure for each string in its language. However, we shall see that not every grammar does provide unique structures.

When a grammar fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. Unfortunately, sometimes we cannot do so. That is, there are some CFL's that are "inherently ambiguous"; every grammar for the language puts more than one structure on some strings in the language.

Ambiguous Grammars

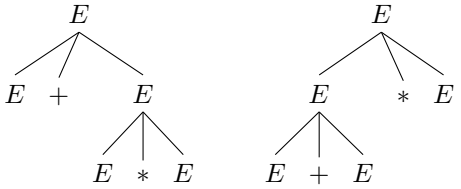
Suppose that we have a grammar that lets us generates expressions with any sequence of $*$ and $+$ operators, and the productions $E \to E + E | E * E$ allow us to generate expression in any order we choose.

For instance, consider the sentential form $E + E * E$. It has two derivations from E:

1. $E \Rightarrow E + E \Rightarrow E + E * E$
2. $E \Rightarrow E * E \Rightarrow E + E * E$

Notice that in derivation (1), the second E is replaced by $E * E$, while in derivation (2), the first E is replaced by $E + E$.

The difference between these two derivations is significant. As far as the structure of the expressions is concerned, derivation (1) says that the second and third expressions are multiplied, and the result is added to the first expression, while derivation (2) adds the first two expressions and multiplies the result by the third. Their parse trees are shown below:



However, the mere existence of different derivations for a string (as opposed to different parse trees) does not imply a defect in the grammar.

It is not a multiplicity of derivations that cause ambiguity, but rather the existence of two or more parse trees. Thus we say a CFG $G = (V, T, P, S)$ is *ambiguous* if there is at least one string w in T^* for which we can find two different parse trees, each with root labeled S and yield w. If each string has at most one parse tree in the grammar, then the grammar is *unambiguous*.

Removing Ambiguity From Grammars

For the sorts of constructs that appear in common programming languages, there are well-known techniques for eliminating ambiguity. The problem with the expression grammar in the previous section is typical, and we shall explore the elimination of its ambiguity as an important illustration.

First, let us note that there are two causes of ambiguity in the grammar:

1. The precedence of operators is not respected.
2. A sequence of identical operators can group either from the left or from the right. For example, if the $*$'s were replaced by $+$'s, we would see two different parse trees from the string $E + E + E$. Since addition and multiplication are associative, it doesn't matter whether we group from the left or the right, but to eliminate ambiguity, we must pick one. The conventional approach is to insist on grouping from the left.

The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a level of "binding strength." Specifically:

1. A *factor* is an expression that cannot be broken apart by any adjacent operator, either a $*$ or a $+$. The only factors in our expression language are:
   (a) Identifiers. It is not possible to separate the letters of an identifier by attaching an operator.
   (b) Any parenthesized expression, no matter what appears inside the parentheses. It is the purpose of parentheses to prevent what is inside from becoming the operand of any operator outside the parentheses.
2. A *term* is an expression that cannot be broke by the $+$ operator.
3. An *expression* will henceforth refer to any possible expression, including those that can be broken by either an adjacent $*$ or an adjacent $+$.

Leftmost Derivations as a Way to Express Ambiguity

While derivations are not necessarily unique, even if the grammar is unambiguous, it turns out that, in an unambiguous grammar, leftmost derivations will be unique, and rightmost derivations will be unique.

Theorem 5. *For each grammar $G = (V, T, P, S)$ and string w in T^*, w has two distinct parse trees if and only if w has two distinct leftmost derivations from S.*

Inherent Ambiguity

A context-free language L is said to be *inherently ambiguous* if all its grammars are ambiguous. If even one grammar for L is unambiguous, then L is an unambiguous language.

Summary of Chapter 5

Context-Free Grammars A CFG is a way of describing languages by recursive rules called productions. A CFG consists of a set of variables, a set of terminal symbols, and a start variable, as well as the productions. Each production consists of a head variable and a body consisting of a string of zero or more variables and/or terminals.

Derivations and Languages Beginning with the start symbol, we derive terminal strings by repeatedly replacing a variable by the body of some production with that variable in the head. The language of the CFG is the set of terminal strings we can so derive; it is called a context-free language.

Leftmost and Rightmost Derivations If we always replace the leftmost (and rightmost) variable in a string, then the resulting derivation is a leftmost (or rightmost) derivation. Every string in the language of a CFG has at least one leftmost and at least one right most derivation.

Sentential Forms Any step in a derivation is a string of variables and/or terminals. We call such a string a sentential form. If the derivation is leftmost (or rightmost), the the string is a left-(right-)sentential form.

Parse Trees A parse tree is a tree that shows the essentials of a derivation. Interior nodes are labeled by variables, and leaves are labeled by terminals or ϵ. For each internal node, there must be a production such that the head of the production is the label of the node, and the labels of its children, read from left to right, form the body of that production.

Equivalence of Parse Trees and Derivations A terminal string is in the language of a grammar if and only if it is the yield of at least one parse tree. Thus, the existence of leftmost derivations, rightmost derivations, and parse trees are equivalent conditions that each define exactly the strings in the language of a CFG.

Ambiguous Grammars For some CFG's, it is possible to find a terminal string with more than one parse tree, or equivalently, more than one leftmost derivation or more than one rightmost derivation. Such a grammar is called ambiguous.

Eliminating Ambiguity For many useful grammars, such as those that describe the structure of programs in a typical programming language, it is possible to find an unambiguous grammar that generates the same language. Unfortunately, the unambiguous grammar is frequently more complex than the simplest ambiguous grammar for the language. There are also some context- free languages, usually quite contrived, that are inherently ambiguous, meaning that every grammar for that language is ambiguous.

Parsers The context-free grammar is an essential concept for the implementation of compilers and other programming-language processors. Tools such as YACC take a CFG as input and produce a parser, the component of a compiler that deduces the structure of the program being compiled.

Document Type Definitions The emerging XML standard for sharing information through Web documents has a notation, called the DTD, for describing the structure of such documents, through the nesting of semantic tags within the document. The DTD is in essence a context-free grammar whose language is a class of related documents.