LECTURE 3

Context-Free Grammars

1 Where are Context-Free Grammars (CFGs) Used?

CFGs are a more powerful formalism than regular expressions. They are more powerful in the sense that whatever can be expressed using regular expressions can be expressed using grammars (short for context-free grammars here), but they can also express languages that do not have regular expressions. An example of such a language is the set of well-matched parenthesis. Grammars are used to express syntactic rules. These rules are used by the compiler to take a steam of tokens (the output from a scanner/lexical analyzer) and parse it for syntactic correctness, e.g. checking that each construct is well formed, all parentheses are matched, or all keywords are spelled correctly. This process is known as parsing.

2 Definitions

A *context-free grammar* G is a 4-tuple <N,T,P,S> where N is a set of *nonterminals*, T is a set of *terminals*, P is a set of *production* rules of the form $A\rightarrow\alpha$, A is an element of set N, i.e. $A\in N$, and $\alpha\in(N\bigcup T)^*$, and S is a specific non-terminal called the *start symbol*. Sometimes, the set of terminals is also referred to as the alphabet.

Recall that for a set of strings I, the notation I*, Kleene closure, refers to the set of all strings obtained by concatenation of zero or more elements taken from the set I in any order. For example, if I={a,b,A,B}, then the set I* is

{ε, a, b, A, B, aa, bb, AA, BB, ab, ba, aA, Aa,aB, Ba, bA, Ab, bB, Bb,...}

where ε is the empty string.

Here are other definitions related to context-free grammars and languages:

A *derivation* using the rule $A\rightarrow\alpha$ is the process of obtaining a new string from a string w by replacing an occurrence of A in w with α.

A *sentence* is a string consisting of only terminal symbols. A *valid sentence with respect to grammar G* is a sentence that can be derived using the production rules of G starting from S and ending with a sentence. A *leftmost* (*rightmost*) derivation is one in which at every step the leftmost (resp. rightmost) nonterminal is replaced. For a grammar G, the language *generated* by G (denoted L(G)) is the set of all valid sentences with respect to G. A grammar is said to be *ambiguous* if L(G) contains a string w such that w has two or more leftmost or rightmost derivations. Sometimes it is possible to rewrite a grammar so that the new grammar produces the same language as the original, but it is not ambiguous. When all grammars for a language are ambiguous, then that language is said to be *inherently ambiguous*. There are languages that are inherently ambiguous. This is a mathematically established fact the proof of which is beyond the scope of this course.

A tree is a *parse* tree for a sentence w of L(G) if every node of the tree has a label from $N \cup T$ and it further satisfies the following conditions.

- The root of the tree is labeled S.
- All the leaves are labeled with terminal symbols which when concatenated from left to right will give w.
- All interior nodes are labeled with a nonterminal symbol.
- If an interior node is labeled A and its children from left to right are labeled X_1, X_2... X_k where each X_i is either a terminal or a nonterminal symbol, then G must contain production rule of the form $A \rightarrow X_1X_2...X_k$.

3 Examples

3.1 Equal Number of a's and b's

Implicit in the question is the fact that T={a,b}. Here is a grammar for it:

N = {S}
S = S
P = {$S \rightarrow aSb$, $S \rightarrow bSa$, $S \rightarrow SS$, $S \rightarrow \varepsilon$ }

For brevity, we write the above grammar as $S \rightarrow aSb \mid bSa \mid SS \mid \varepsilon$, and leave out the details. Our convention is to assume that the left-side nonterminal of the first production rule is the start symbol. Uppercase letters stand for nonterminals. Numbers and lowercase letters constitute the terminals.

3.2 Set of Balanced Parenthesis

Here T = {(,)}. The grammar is $S \rightarrow (S) \mid SS \mid \varepsilon$.

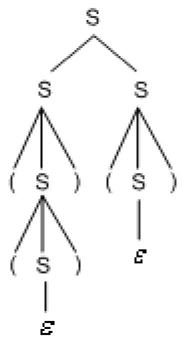

Figure 3-1 A parse tree for the string (())()

Here is an example of a leftmost derivation:

$$S \rightarrow SS \rightarrow (S)S \rightarrow ((S))S \rightarrow (())S \rightarrow (())(S) \rightarrow (())()$$

3.3 Ambiguity

Consider the grammar for arithmetic expressions involving addition and multiplication operators:

$$E \rightarrow E+E$$
$$E \rightarrow E*E$$
$$E \rightarrow ID$$

It is easy to see that this grammar produces all arithmetic expressions consisting of + and *. Consider the sentence ID+ID*ID. This can be parsed in two different ways:

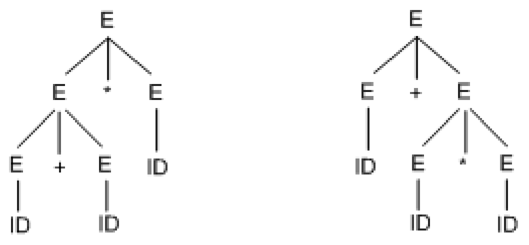

Figure 3-2 Ambiguous way to parse ID+ID*ID

In the figure above, the parse tree to the left gives the addition operator precedence over multiplication. In other words, an expression such as 3+5*9 is evaluated as (3+5)*9 with a result of 72. Whereas, the tree to the right, does what is considered the standard practice in programming languages, i.e. giving * precedence over +. The previous example would be evaluated as 3+(5*9) resulting in 48.

The above grammar can be rewritten so that the new grammar produces the same language as the original at the same time every string in the language would have a unique parse tree. For example, the following grammar produces parse trees that give * precedence over +:

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T{*}ID \mid ID$$

In general, this is not always possible. As mentioned before, some languages don't have unambiguous grammars.

The ambiguity in the grammar for arithmetic expressions exists even if we had only one operator. Consider

$$E \rightarrow E+E \mid ID$$

This grammar is ambiguous as well. The sentence ID+ID+ID has two parse trees as shown below:

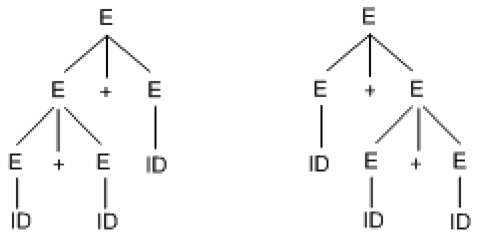

Figure 3-3 The Associativity Problem

The parse tree to the left interprets ID+ID+ID as (ID+ID)+ID while the one to the right treats it as ID+(ID+ID). In mathematics, since addition is associative, both interpretations yield identical results. However, in programming languages, owing to the limited precision of floating point numbers, the addition operator is not strictly associative. To avoid confusion, many programming languages specify the operator evaluation order when two operators of equal precedence are encountered. Typically, + and * are left associative. That is, the interpretation used by the first tree is used. An example of an operator that is right associative is the exponentiation operator. The grammar above can also made unambiguous with left-to-right associativity as

$$E \rightarrow E+ID \mid ID$$

As an exercise, try to come up with an unambiguous grammar for arithmetic expressions involving +, * and ^ (exponentiation) operators so that ^ has precedence over * which has precedence over +. Make the ^ operator right associative and the other two operators left associative.

3.4 A Language that is not Context-Free

Consider the language $\{a^n b^n c^m \mid m, n \geq 1\}$. A grammar for this language is

$$S \rightarrow PQ$$
$$P \rightarrow aPb \mid ab$$
$$Q \rightarrow cQ \mid c$$

However, a slight variation of this language $\{a^n b^n c^n \mid n \geq 1\}$ can shown to be not context free!

3.5 Chomsky Hierarchy

The language $\{a^n b^n c^n \mid n \geq 1\}$ can be captured by context-sensitive grammar (CSG). These grammars are a more powerful formalism than CFGs. There are languages that cannot be expressed using CSGs. For these, we use unrestricted grammars.

The hierarchical relationship between various formalisms in language theory is shown as a Venn diagram. This is called the Chomsky hierarchy.

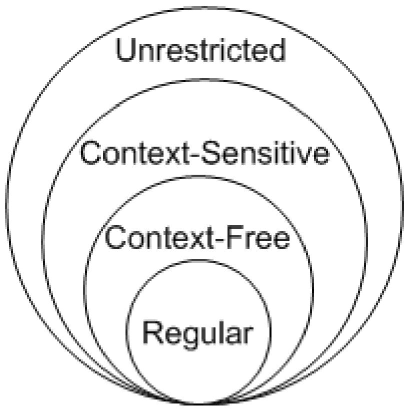

Figure 3-4 Chomsky Hierarchy.

As is clear from the figure above that if a language is regular, then it is also context-free. The converse is not true. For example, $\{a^n b^n \mid n \geq 1\}$ is context-free, but not regular. Similarly, if a language is context-free, then it is context-sensitive. Again, the inclusion is strict. As mentioned before $\{a^n b^n c^n \mid n \geq 1\}$ is not context-free, but context-sensitive. This behavior continues for another level to unrestricted grammars. The discussion involving the last two sets is beyond the scope of this course.

It should be noted however that the fact that some languages do not have CFGs has an impact on programming-language translation. That is, certain syntactic constraints cannot be enforced during the parsing phase using context-free grammars. Some examples are

- Checking that a variable is declared before it is used,
- Ensuring that an array declared to have two dimensions is referenced with exactly two subscripts,
- Making sure that no identifier is declared twice within the same block,
- Verifying that the number and the order of parameters in a function call match those in the signature of the function, and

- Type checking.

These are typically enforced as a separate phase after parsing. This check is known as the *static-semantic* check; despite the name, it should be noted that the checks enforced in this phase are strictly syntactic and have nothing to do with semantics.

4 Equivalent Syntactic Formalisms

There are two formalisms that are commonly used to express the syntax of programming languages: Extended BNF (EBNF) notation and syntax graphs. It should be noted that these formalisms only provide convenience and are not any more powerful than context-free grammars.

In EBNF, the nonterminals are enclosed between angular brackets. The production rules use the symbol ::= instead of an arrow. Recursive production rules can be simplified by using Kleene closure. Square brackets are used indicate an optional element that can be used at most once. Zero or more uses of an element is expressed by enclosing that element within curly braces. Vertical bar, as shown in the examples before, gives the alternatives for a production.

Here is an example grammar in EBNF for assignment statements:

⟨assignment⟩ ::= ⟨variable⟩=⟨expression⟩
⟨expression⟩ ::= ⟨term⟩{ [+ | -]⟨term⟩}*
⟨term⟩ ::= ⟨primary⟩{[× | /]⟨primary⟩}*
⟨primary⟩ ::= ⟨variable⟩ | ⟨number⟩ | (⟨expression⟩)
⟨variable⟩ ::= ⟨identifier⟩ | ⟨identifier⟩**[**⟨subscript list⟩**]**
⟨subscript list⟩ ::= ⟨expression⟩{,⟨expression⟩}*

In the grammar above, choices on the right side are enclosed between regular square brackets whereas the terminal symbol for square bracket that is used to enclose array indices is shown in bold. This grammar can be shown graphically using syntax charts as in the figure below:

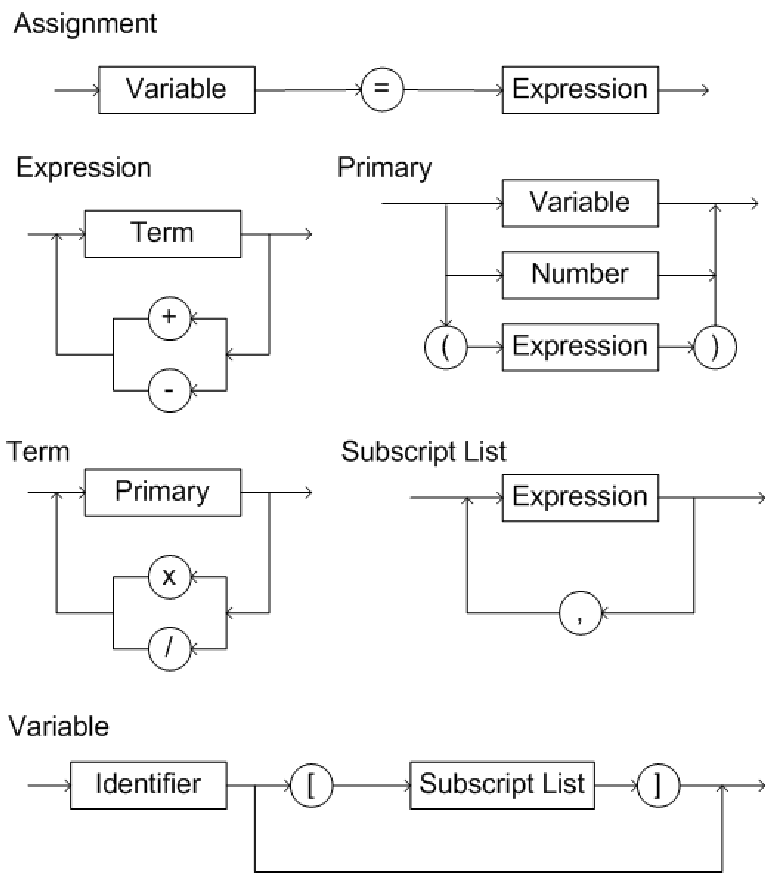

Figure 3--5 Syntax graph corresponding to the example EBNF grammar

Exercises

1. Give context-free grammars generating the following sets:
   - (a) The set of palindromes (strings that read the same forward as backward) over {a,b}.
   - (b) The complement of $\{a^n b^n \mid n \geq 0\}$

2. Show that CFGs are closed under concatenation, union, and Kleene closure. (A set *S* is *closed* under an operation, if the result of applying that operator also belongs to the set *S*. For example, the set of integers are closed under addition, but not under division.)

Problems

1. Give context-free grammars generating the following sets:
   - (a) $\{a^i b^j c^k \mid i \neq j \text{ or } j \neq k\}$.
   - (b) The set of all valid regular expressions over {0,1,|,(,),*}
   - (c) {w | w contains more 1s than 0s}
2. Consider the following unambiguous grammar for arithmetic expressions:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid ID$

Change it to an unambiguous grammar that produces all arithmetic expressions with no redundant parentheses. Note that a set of parentheses is redundant if its removal does not change the expression, e.g., the parentheses are redundant in ID+(ID*ID) but not in (ID+ID)*ID.

3. Is the following grammar ambiguous?

   $S \rightarrow aB \mid bA$
   $A \rightarrow a \mid aS \mid bAA$
   $B \rightarrow b \mid bS \mid aBB$

4. Show using mathematical induction that the strings in the language produced by the following grammar do not contain *ba*.

   $S \rightarrow aS \mid Sb \mid a \mid b$

5. Give a simple English description of the language generated by the following grammar:

   $S \rightarrow aSb \mid bY \mid Ya$
   $Y \rightarrow bY \mid aY \mid \varepsilon$

Answers to Exercises

1. (a) $S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$

   (b) The desired set consists of two subsets: One consisting of all strings that have ba as a substring, and the other $\{a^m b^n \mid m \neq n\}$. Producing them separately and taking a union, we get

   $S \rightarrow S_1 \mid S_2$
   $S_1 \rightarrow X\ ba\ X$
   $X \rightarrow aX \mid bX \mid \varepsilon$

   $S_2 \rightarrow a\ S_2\ b \mid A \mid B$
   $A \rightarrow aA \mid a$
   $B \rightarrow bB \mid b$

2. We will show CFGs are closed under concatenation. Let G_1 and G_2 be two context-free grammars. Let S_1 and S_2 be the start of the production rules in G_1 and G_2 respectively. Now, we can create new grammar G_3 from G_1 and G_2 that produces $L(G_1)L(G_2)$. Introduce a new start symbol S_3 and the production rule $S_3 \rightarrow S_1S_2$ to G_3. Also, add all production rules of G_1 and G_2 to G_3. It is easy to see that G_3 is a context-free grammar that produces $L(G_1)L(G_2)$. Proofs for other operations are similar.