CS250 Winter 2003

Week 12, Lecture 3:
Context free languages (CFL’s)

Reference: Micheal Sipser,
*Introduction to the theory of computation*,
QA267 S56 1997 [Two Hour Loan]
Schulich Science & Engineering Reserves.
A **formal language** is a set of finite-length words (or "strings") over a finite alphabet. While the alphabet is a finite set and every string has finite length, a language may very well have infinitely many member strings.

- the set of all words over \{a, b\} or the set \{a^n | n is prime\}
- the set of syntactically correct programs in Java
- the set of inputs upon which a certain algorithm halts

A formal language can be stated as the strings: produced by a regular expression, or accepted by a DFA, or produced by a **formal grammar**.

We generate strings from a start symbol 'S', then apply rules that indicate how some symbol combinations may be replaced by other ones. E.g.: Let the alphabet be \{a, b\} and the rules:

\[
S \to aSa \quad \text{(rule 1)} \\
S \to ba \quad \text{(rule 2)}
\]

then we can rewrite "S" to "aSa" by replacing 'S' with "aSa" (rule 1), and we can then rewrite "aSa" to "aaSaa" by again applying the same rule. This is repeated until the result contains only symbols from the alphabet. In our example we can rewrite S as follows: S → aSa → aaSaa → aabaaa. The language of the grammar consists of all the strings that can be generated that way: ba, abaa, aabaaa, aabaaaaa, etc.
**Definition:** A formal grammar $G$ consists of: a finite set $N$ of non-terminal symbols; a finite set $\Sigma$ of terminal symbols that is disjoint from $N$; a finite set $P$ of production rules where a rule is of the form:

$$\text{string in } (\Sigma \cup N)^* \rightarrow \text{string in } (\Sigma \cup N)^*$$

with a restriction: a rule’s LHS contains $\geq$ one non-terminal symbol. A symbol $S \in N$ is the start symbol.

**Definition:** The language of $G = (N, \Sigma, P, S)$, denoted as $L(G)$, consists of all strings over $\Sigma$ that can be generated by starting with $S$ and then applying the production rules in $P$ until no non-terminal symbols remain.

**Definition:** A context-free grammar is a formal grammar in which every production rule is of the form

$$V \rightarrow w$$

where $V$ is a non-terminal symbol and $w$ is a string consisting of terminals or non-terminals. The term “context-free” comes from that the variable $V$ can be replaced by $w$, regardless of the context it occurs in.

**Definition:** A formal language is context-free if there is a context-free grammar which generates it.

**Example 1:** A simple context-free grammar is

$$S \rightarrow aSb \mid \varepsilon$$

where $\mid$ is used to separate different options for the same non-terminal. This grammar generates the language \{a^n b^n : n \geq 0\} which is not regular.
Example 2: Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables $x$, $y$ and $z$:

$$
S \rightarrow T + S \mid T - S \mid T \\
T \rightarrow T \ast T \mid T / T \mid (S) \mid x \mid y \mid z
$$

This grammar can for example generate the string: 

"$(x + y) \ast x - z \ast y/(x + x)$".

Example 3: The grammar $G = (\{S\}, \{a, b\}, S, P)$ with productions

$$
S \rightarrow aSa \\
S \rightarrow bSb \\
S \rightarrow \varepsilon
$$

is context-free and $L(G) = \{ww^R : w \in \{a, b\}^*\}$.

Example 4: Grammar $G = (\{S\}, \{a, b\}, S, P)$ with $P$:  

$$
S \rightarrow aSb \mid \varepsilon
$$

is context-free and $L(G) = \{a^mb^n : m = n\}$.

Example 5: The language $L = \{a^mb^n : m \neq n\}$ is a CFL since we can show $G = (\{S\}, \{a, b\}, S, P)$ with $P$:  

$$
S \rightarrow AS_1 \mid S_1B \\
S_1 \rightarrow aS_1b \mid \varepsilon \\
A \rightarrow aA \mid a \\
B \rightarrow bB \mid b
$$
Derivations and syntax trees

The simplest way to describe how in a certain context-free grammar a string can be derived from the start symbol is to list the consecutive strings of symbols, beginning with the start symbol and ending with the string, and the rules that have been applied.

If we use the strategy: “always replace the left-most non-terminal first”, then the list of applied grammar rules is sufficient. This is called the **leftmost derivation of a string**. E.g. for the following grammar:

1. \( S \rightarrow S + S \)
2. \( S \rightarrow 1 \)

and the string “1 + 1 + 1”, the left derivation is the list [ 1, 1, 2, 2, 2 ]. Analogously the **rightmost derivation** is defined as the list that we get if we always replace the rightmost non-terminal first. In this case, it is the list [ 1, 2, 1, 2, 2 ].

This is fundamentally important when a compiler **parses** a program. A piece of executable code is produced for every grammar rule that is applied. Therefore it is essential to know whether the parser determines a leftmost or a rightmost derivation because this determines the order in which the the pieces of code will be executed.
A derivation also imposes in some sense a **hierarchical structure** on the string that is derived. For example the structure of the string “1 + 1 + 1” would, according to the leftmost derivation, be:

\[
\{\{\{1\}_S + \{1\}_S\}_S + \{1\}_S\}_S
\]

where \{\ldots\}_S indicates a substring recognized as belonging to \(S\). This hierarchy can also be seen as a tree:

```
S
+-- S
|   +-- S
|   |   +-- 1
|   |   +-- 1
|   +-- 1
```

This tree is called a **concrete syntax tree of the string**. In this case the presented leftmost and the rightmost derivation define the same syntax tree.
However another leftmost derivation of the same string is possible:

\[
S \rightarrow (1) \ S + S \rightarrow (2) \ 1 + S \rightarrow (1) \ 1 + S + S \\
\rightarrow (2) \ 1 + 1 + S \rightarrow (2) \ 1 + 1 + 1
\]

and it defines the following syntax tree:

```
  S
 / \  /
/ \ / \\
S + S
\  \  /
1 S + S
   \  /
    1 1
```

If for certain strings in the language of the grammar there are more than one parsing tree then the grammar is said to be an ambiguous grammar. In programming languages, there should be only one interpretation of each statement, so ambiguity must be removed when possible. Some grammars are inherently ambiguous.

Towards programming a compiler...

Context-free grammars (CFL’s) are are powerful enough to describe the syntax of programming languages. On the other hand, CFL’s are simple enough to allow the construction of efficient parsing algorithms which, for a given string, determine whether and how it can be generated from the grammar. [More in 13-1 and 13-2.]