Context-Free Grammars

Notes on Automata and Theory of Computation

Chia-Ping Chen

Department of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan ROC

Introduction

- Consider the language,

$$L = \{a^n b^n, n \geq 0\}.$$

- L describes a nested structure, such as nested parenthesis.
- L has been shown not to be regular.
- We will introduce the **context-free grammar** (cfg) which can characterize L.
- A language is context-free if there exists a cfg for it. The set of context-free languages includes the regular set as a subset.

Parsing

- The **membership problem** of cfg is this:

  Given a cfg G and a string w, is $w \in L(G)$?

- If $w \in L(G)$, then there is a sequence of production rules that leads to w starting from S.
- An important concept in learning cfg is **parsing.** A parsing algorithm determines how a string w can be derived with a grammar G.
- Parsing describes sentence structure. It is important for understanding natural languages as well as programming languages.

Context-Free Grammar

- A grammar $G = (V, T, S, P)$ is context-free if all production rule in P has the form

$$A \to x,$$

where $A \in V$ and $x \in (V \cup T)^*$.

- It is called context-free because the left side has a single variable. No context of the variable is relevant. The application of a rule does not depend on other parts of the sentential form.
- We can see that a regular grammar is a cfg.

Context-Free Language

- Recall that the language of a grammar G is defined by

$$L(G) = \{w \in T^* : S \stackrel{*}{\Rightarrow} w\}.$$

- A language L is said to be **context-free** if $L = L(G)$ for some cfg G.
- For example, a regular language is context-free since a regular grammar is context-free.

Example of cfg

- The following language

$$L = \{ww^R : w \in \{a, b\}^*\}.$$

is context-free since it can be generated by

$$S \rightarrow aSa \mid bSb \mid \lambda.$$

- Note if $x \in L$, then $x^R = x$. Such a language is also called **palindrome**.

Another Example

- We design a cfg for the language

$$L = \{a^n b^m : n \neq m\}.$$

- We consider the rules for $n > m$ and $n < m$.
  For extra a's, we decompose S by a string of a's (A), followed by an equal number of b's (S_1).

$$S \to AS_1;\ S_1 \to aS_1b \mid \lambda;\ A \to aA \mid a.$$

  Similarly for extra b's. So the rules for L is

$$S \to aS_1 \mid S_1B;\ S_1 \to aS_1b \mid \lambda;\ A \to aA \mid a;\ B \to Bb \mid b.$$

Yet Another Example

- A grammar can be context-free but not linear, e.g.

$$S \to aSb \mid SS \mid \lambda.$$

- Looking simple, this cfg is a useful one as it accepts

$$L = \{w \in \{a,b\}^* : \; n_a(w) = n_b(w), \\ n_a(v) \geq n_b(v) \text{ for prefix } v \text{ of } w\},$$

  which is a homomorphism to the set of properly nested parentheses.

Derivation

- A **derivation** of a string $w \in L(G)$ is a sequence of sentential forms from S to w.
- When a cfg is not linear, a production rule may have more than one variables on the right side, so there may be more than one variable in a sentential form.
- In such cases, we have a choice for the next variable to be replaced by a corresponding right side.

Leftmost/Rightmost Derivation

- A derivation is said to be **leftmost** if in each step the leftmost variable in the sentential form is replaced.
- It is **rightmost** if the rightmost variable is replaced in each step.
- Leftmost and rightmost derivations always exist for a string $w \in L(G)$.

Derivation Tree

- A **derivation tree** of a cfg $G = (V, T, S, P)$ is a tree.
  - The root is S.
  - An interior node is labeled by $A \in V$.
  - A leaf is labeled by $a \in T$ or λ.
  - The label of an interior node and the labels of its children constitute a rule in P.
  - A leaf labeled λ has no siblings.
- A derivation tree shows which rules are used in the derivation of w. The order of the rules used is not shown in the tree.

Partial Derivation Tree

- A **partial derivation** tree is similar to a derivation tree, except that
  - The root may not be S.
  - A leaf is labeled by $A \in V \cup T \cup \{\lambda\}$.
- The string of symbols from left to right of a tree, omitting $\lambda's$, is called the **yield**. Here "left to right" means the tree is traversed in a depth-first manner, always taking the leftmost unexplored branch.
- The yield of a derivation tree for w is w.

Theorem

- We first establish the connection between derivation and derivation tree.
- Let G be a cfg.
  - If $w \in L(G)$, i.e. there exists a derivation $S \stackrel{*}{\Rightarrow} w$, then there exists a derivation tree whose yield is w.
  - Conversely, if w is the yield of a derivation tree, then $w \in L(G)$.
- In addition, if t_G is any partial derivation tree rooted by S, then the yield of t_G is a sentential form of G.

Proof

- We first show that for every sentential form, say u, there is a corresponding partial derivation tree. If u can be derived from S in one step, there there must be a rule $S \to u$. Suppose the claim is true for all sentential forms derivable in n steps. For a u that is derived from S in $(n+1)$ steps, the first n steps correspond to a partial tree by the inductive assumption, and a new partial derivation tree can be built based on the last step of the production.
- Similarly, we can prove that every partial derivation tree rooted by S corresponds to a sentential form.
- The theorem is proved since a terminal string in $L(G)$ is a sentential form, and a derivation tree is a partial derivation tree.

Existence of Leftmost Derivation

- The derivation tree is a representation of derivation. In this representation, the order of production rules in the derivation is irrelevant.
- From a derivation tree, we can always get a sequence of partial derivation trees rooted by S in which the leftmost node of variable is expanded.
- In terms of sentential form, the leftmost variable is expanded, which corresponds to a leftmost derivation.
- We conclude that for each $w \in L(G)$, there is a leftmost derivation.

Parsing

- Given G, we may want to know $L(G)$, i.e. the set of strings that can be derived using G.
- Given G and a string w, we may be interested in whether $w \in L(G)$. This is the membership problem.
- Suppose $w \in L(G)$, then there exists a sequence of productions that w is derived from S. Parsing is the process of finding such a sequence.

Brute Force Parsing

- The brute-force (exhaustive) method to decide whether $w \in L(G)$ would be to construct all derivations and see if any of them matches w.
- We can do this recursively.
  - First we construct all x derived from S in one step. If none matches w, we expand the leftmost variable for every x, which gives all sentential forms derived from S in two steps, and so on.
  - If $w \in L(G)$, there is a leftmost derivation for w in a finite number of steps. So eventually w will be matched.
- Let's look at an example.

Flaw and Remedy

- The brute-force parsing has a serious flaw: it may never terminate. In fact, if $w \notin L(G)$, clearly w will never be matched.
- In the case $w \notin L(G)$, we want to be able to terminate the search when we are sure of it.
- We can put some restriction on the form of production rules to be able to terminate the search when $w \notin L(G)$. These restriction should have virtually no effect on the descriptive power of cfg's.

Theorem

- If all of the production rules are *not* of the forms

$$A \to \lambda, \text{ or } A \to B.$$

  then the exhaustive search can terminate in no more than $2|w|$ rounds.

- (proof) With the above condition, each step in derivation either increases the number of terminals or the length in the sentential form. Since none of these numbers can be more than $|w|$ to derive w, we need no more than $2|w|$ steps to decide if $w \in L(G)$.

Efficiency Issue

- While the previous theorem guarantees a termination, the number of sentential forms may grow excessively large.
- If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after the first round, $|P|^2$ sentential forms after the second round, and so on. So the maximum number of sentential forms generated during exhaustive search is

$$n \leq |P| + |P|^2 + \cdots + |P|^{2|w|} = O(|P|^{2|w|+1}).$$

- Exhaustive search is thus generally very inefficient.

Simple Grammar

- A more efficient algorithm than the exhaustive search to decide whether $w \in L(G)$ can do the job in a number of steps proportional to $|w|^3$.
- Even $O(|w|^3)$ can be excessive. Is there a linear-time parsing algorithm?
- A cfg $G = (V, T, S, P)$ is said to be a **simple grammar**, or **s-grammar**, if all of its production rules are of the form

  $$A \to ax,$$

  where $a \in T, x \in V^*$ and any pair (A, a) occurs at most once in P.

Linear Time

- For a simple grammar G, any string $w \in L(G)$ can be parsed in $|w|$ steps.
- Suppose $w = a_1 a_2 \ldots a_n \in L(G)$. Since there can be only at most one rule with S on the left and a_1 on the right, the derivation has to begin with

$$S \Rightarrow a_1 A_1 \ldots A_m.$$

Similarly, there can be only at most one rule with A_1 on the left and a_2 on the right, so the next sentential form has to be

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 \ldots A_2 \ldots A_m.$$

Each step produces one more terminal, so the entire derivation cannot have more than $|w|$ steps.

Ambiguity of Grammar

- A cfg G is said to be **ambiguous** if there exists some $w \in L(G)$ with two or more distinct derivation trees (parses).
- Ambiguity implies the existence of two or more leftmost derivations for some string.
- See example 5.11.
- While it may be possible to associate precedence with operators, it is better to rewrite the grammars.
- Ambiguity is not desired in programming languages. In some cases, one can rewrite an ambiguous grammar in an equivalent and unambiguous one.

Ambiguity of Language

- Suppose L is a context-free language.
  - It is *not* ambiguous if there exists an unambiguous cfg for L.
  - Otherwise, i.e. if all cfg's for L are ambiguous, then L is said to be **(inherently) ambiguous**.
- While the grammar in example 5.11 is ambiguous, the language is not, as there is a non-ambiguous cfg that generates the same language.
- It is a difficult matter to show that a language is inherently ambiguous. See example 5.13.

Example

- Consider the language

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}, \quad n, m \geq 0.$$

- L is a context-free language. Specifically, $L = L_1 \cup L_2$, where $P_1 = S_1 \to S_1 c | A, \; A \to aAb | \lambda$, and similarly for L_2. A grammar for L is

$$P = P_1 \cup P_2 \cup \{S \to S_1 | S_2\}.$$

- A string $a^i b^i c^i$ has two distinct derivations, one begins with $S \to S_1$ and the other begins with $S \to S_2$, so the grammar is ambiguous.
- It does not follow that the language is ambiguous. A rigorous proof is quite technical and is omitted here.

Programming Languages

- One important application of formal languages is in the definition of programming languages and in the construction of compilers and interpreters.
- We want to define a programming language in a precise manner so we can use this definition to write translation programs.
- Both regular and context-free languages are important in designing programming languages. One is used to recognize certain patterns and the other is used to model more complicated structures.

Backus-Naur Form

- A programming language can be defined by a grammar. This is traditionally specified by the **Backus-Naur form** (BNF), which is essentially same as cfg but with a different system of notation.
- It is easy to look at an example of BNF to see how it corresponds to a cfg.

Syntax and Semantics

- Those aspects of a programming language that can be modeled by a cfg are called **syntax**.
- Even if a program is syntactically correct, it may not be acceptable. For example, type clashes may not be permitted in a programming language.
- The **semantics** of a programming language models aspects other than those modeled by the syntax. It is related to the interpretation or meaning of objects.
- It is an ongoing research to find effective methods to model programming language semantics.

Transforming Grammars

- In our definition of cfg's, there is no restriction on the form of the right side of a rule.
- Such flexibility is in fact *not* necessary. That is, given a cfg, we can transform it to an equivalent cfg whose rules conform to certain restrictions.
- Specifically, a **normal form** is a restricted class of cfg but which is broad enough to cover *all* context-free languages (except perhaps $\{\lambda\}$).
- We will introduce the Greibach and the Chomsky normal forms.

A Technical Note

- The empty string λ often requires special attention, so we will assume that the languages are λ-free in the following discussion.
- This is based on the following facts.
  - If L is a λ-free context-free language, then $L \cup \{\lambda\}$ is context-free as well.
  - In addition, suppose L is context-free, then there exists a cfg for $L - \{\lambda\}$.

Substitution Rule

- Suppose variables $A \neq B$ and there is a rule

  $$A \to x_1 B x_2.$$

  Then one can substitute this rule by

  $$A \to x_1 y_1 x_2 \mid x_1 y_2 x_2 \mid \ \ldots \ \mid x_1 y_n x_2.$$

  where $B \to y_1 \mid y_2 \mid \ \ldots \ \mid y_n$ is the set of rules with B as the left side.
- In other words, B can be replaced by all strings it derives in one step.

Proof

- Suppose $w \in L(G)$ so

$$S \stackrel{*}{\Rightarrow}_G w.$$

If the sequence of derivations does not include that rule, then the same sequence exists for $\widehat{G}$, so $w \in L(\widehat{G})$. If it does include that rule, then B eventually has to be replaced. It can be assumed that B is replaced immediately, and then obviously there is a rule in $\widehat{G}$ leading to the next sentential form. Therefore $w \in L(\widehat{G})$.

Useless Production

- A variable A is said to be **useful** iff there exists w such that

$$S \stackrel{*}{\Rightarrow} xAy \stackrel{*}{\Rightarrow} w,$$

  where $x, y \in (V \cap T)^*$. Otherwise it is **useless**.
- A variable may be useless because
  - it cannot be reached from S
  - it cannot derive a terminal string
- A production rule is useless if it involves any useless variables. They can be removed from P without changing $L(G)$.

Dependency Graph

- To decide if a variable can be reached from S, we can use a **dependency graph** as follows.
- In this graph, each vertex corresponds to a variable. There is an edge from C to D iff there exists a rule of the form
$$C \to xDy.$$
- As a result, a variable A is useless if there is no path from S to A in this dependency graph.

Theorem

- Let G be a cfg. Then there exists an equivalent cfg $\widehat{G}$ which has no useless variables or productions.
- We first construct G_1 that involves only variables that can derive terminal strings.
  1. Set $V_1 = \emptyset$. Repeat until no variables are added to V_1. Add A to V_1 if there exists a rule $A \to \alpha$ where all symbols of α are in $V_1 \cup T$.
  2. Take P_1 as those rules in P that involves only symbols in $V_1 \cup T$.
- We then remove variables in V_1 not reachable from S by constructing the aforementioned dependency graph.

λ-Production

- A **λ-production** is

$$A \rightarrow \lambda.$$

- A variable A is said to be **nullable** if it is possible that

$$A \stackrel{*}{\Rightarrow} \lambda.$$

- A λ-production can be removed. Example 6.4 gives an example.

Theorem

- Let G be a cfg and $\lambda \notin L(G)$. Then there exists an equivalent cfg $\widehat{G}$ without λ-production.
- We first find the set of nullable variables V_N.
  1. For all A with $A \rightarrow \lambda$, add A to V_N.
  2. Repeat until no variables are added to V_N. For any $B \in V$, if there exists a rule $B \rightarrow \alpha$ where all symbols of α are in V_N, then add B to V_N.
- For a production rule $A \rightarrow x_1 \ldots x_m$ in P, put this rule, as well as those with nullable variables replaced by λ's in all possible combinations, in $\widehat{P}$.

Unit-Production

- A **unit-production** is

$$A \to B, \quad A, B \in V.$$

- Let G be a cfg without λ-productions. Then there exists an equivalent cfg $\widehat{G}$ without unit-production.
  - We first add all non-unit production rules of P to $\widehat{P}$.
  - Then we find all $A \neq B$ such that $A \stackrel{*}{\Rightarrow} B$, and add to $\widehat{P}$

    $$A \to y_1 | \ldots | y_n,$$

    where $B \to y_1 | \ldots | y_n$ is the set of all rules in $\widehat{P}$ with B on the left side.

Theorem

- Let L ($\lambda \notin L$) be a context-free language. Then there exists a cfg G for L, where G does not have
  - useless production rules
  - λ-productions
  - unit-productions.

Chomsky Normal Form

- A cfg is said to be in **Chomsky normal form** if all production rules are of the form

$$A \rightarrow BC, \text{ or } A \rightarrow a.$$

  where $a \in T$ and $B, C \in V$.
- The right side is either a single terminal symbol or a string of two variables.
- (Theorem 6.6) Let L ($\lambda \notin L$) be a context-free language. Then there exists a cfg in Chomsky normal form for L.

Greibach Normal Form

- A cfg is said to be in **Greibach normal form** if all production rules are of the form

$$A \to ax,$$

  where $a \in T$ and $x \in V^*$.

- A right side has to be a terminal symbol followed by a variable string of an arbitrary length.
- (Theorem 6.7) Let L ($\lambda \notin L$) be a context-free language. Then there exists a cfg in Greibach normal form for L.

Membership Algorithm

- The membership problem for cfg is

  Given G and w, decide if $w \in L(G)$.

- An algorithm to answer correctly for all instances of G and w is called a membership algorithm for cfg.
- Does there exist a membership algorithm for cfg? We claimed that there is one with complexity $O(|w|^3)$. This is the CYK algorithm, after Cocke, Younger and Kasami.

CYK Algorithm

- The idea of CYK is to solve one big problem by solving a sequence of smaller ones.
- Assume we have a grammar in Chomsky normal form and a string $w = a_1 \ldots a_n$.
  - Define the set of variables

$$V_{ij} = \{A \in V : A \stackrel{*}{\Rightarrow} w_{ij} = a_i \ldots a_j\}.$$

  - Note $w \in L(G) \Leftrightarrow S \in V_{1n}$.

Details

- To decide V_{ij}, observe that $A \in V_{ii}$ iff $A \rightarrow a_i$. So V_{ii} for all i can be decided trivially.
- For $j > i$, $A \stackrel{*}{\Rightarrow} w_{ij}$ iff $A \rightarrow BC,\ B \stackrel{*}{\Rightarrow} w_{ik}$, and $C \stackrel{*}{\Rightarrow} w_{k+1j}$. That is

$$V_{ij} = \bigcup_{k \in \{i,\ldots,j-1\}} \{A : A \rightarrow BC, B \in V_{ik}, C \in V_{k+1j}\}.$$

- The order of computation is thus
  - Compute $V_{11}, V_{22}, \ldots, V_{nn}$.
  - Compute $V_{12}, V_{23}, \ldots, V_{n-1n}$.
  - Compute $V_{13}, V_{24}, \ldots, V_{n-2n}$, and so on.