CONTEXT FREE LANGUAGES

**1 INTRODUCTION**

**Context free languages, CFLs, include all the regular languages and many additional languages. In particular, most of the known programming languages are either CFLs themselves or can be defined approximately by a CFL.**

**The class of machines that recognize a CFL are called *push-down automata*, PDAs.**

**The following is an example of a CFG, which we’ll call G1:**

##### A → 0A1

**A → #**

**It is easy to verify that grammar G1 generates for example the string 000#111. The *derivation* or sequence of substitutions used to obtain the string is:**

**A → 0A1** **→ 00A11 → 000A111 → 000#111 → 03#13.**

**As we have seen earlier, we can also represent the same information in a more pictorial way using a parse tree.**

**All strings generated in this way constitute the *language* L(G1) of this CFG.**

**After some experimentation with grammar G1 allow us to show that the language it generates is**

**L(G1) = {0n#1n | n > 0}**

**EXAMPLE 2.2:**

**Consider the grammar G3 = ({S}, {a, b}, S, P). The set of rules P is**

**S → aSb | SS | **

**This grammar generates strings such as *abab, aaabbb, aababb*. You can see more easily what this language is if you think of *a* as a left parenthesis “(“ and *b* as a right parenthesis “)”.**

**Viewed this way, L(G3) is the language of all strings of properly nested parenthesis.**

**MORE EXAMPLES OF CONTEXT FREE LANGUAGES**

**The syntax of programming languages is described mostly by grammars, in particular, context-free grammars, CFG. (Some languages cannot be generated by any grammar).**

**EXAMPLES:**

**1. L1 = {wcwR / w in (a | b)\*} is CF (wR is w reversed)**

**2. L2 = {wcw / w in (a | b)\*} is not CF**

**3. L3 = {an bn cm dm / m, n ≥ 1} is CF**

**4. L4 = {an bm cn dm / m, n ≥ 1} is not CF**

**5. L5 = {an bn / n ≥ 0} is CF**

**6. L6 = {an bn cn / n ≥ 0} is not CF**

**7. L7 = { an bm cm dn / m, n ≥ 1} is CF**

**1) L1 = {wcwR / w in (a | b)\*} is CF (wR is w reversed). The grammar that generates L1 is:**

**S → aSa | bSb | c**

**2) L3 = {an bn cm dm / m, n ≥ 1} is CF. The grammar that generates L3 is:**

**S → AB;**

**A → aAb | ab;**

**B → cBd | cd**

**3) L5 = {an bn / n ≥1} is CF. The grammar that generates L5 is:**

**S → aSb | ab.**

**4) L7 = {an bm cm dn / m, n ≥ 1} is CF. The grammar that generates L7 is:**

**S → aSd | aAd   
A → bAc | bc**

###### DESIGNING CONTEXT-FREE GRAMMARS

**As with the design of finite automata, discussed earlier, the design of context-free grammars requires creativity.**

**Indeed, context-free gram­mars are even trickier to construct than finite automata because we are more accustomed to programming a machine for specific tasks than we are to describ­ing languages with grammars.**

**The following techniques are helpful, singly or in combination, when you're faced with the problem of constructing a CFG.**

**UNION OF CFLs**

**First, many CFGs are the union of simpler CFGs.**

**If you must construct a CFG for a CFL that you can break into simpler pieces, do so and then construct individ­ual grammars for each piece.**

**These individual grammars can be easily combined into a grammar for the original language by putting all their rules together and then adding the new rule**

**S → S1 | S2 | … | Sk**

**where the variables *S*iare the start variables for the individual grammars.**

**Solving several simpler problems is often easier than solving one complicated problem.**

**Example**

**For example, to get a grammar for the language**

**{0n1n | n ≥ 0} ∪ {1n0n | n ≥ 0}.**

**First construct the grammar**

**S1 →0S11 | **

**for the language {0n1n | n ≥ 0} and then construct the grammar**

**S2 →1S20 | **

**for the language {1n0n | n ≥ 0}.**

**Lastly, add the rule S → S1 | S2 to give the grammar**

**S → S1 | S2**

**S1 →0S11 | **

**S2 →1S20 | **

**Constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language.**

**You can convert any DFA into an equivalent CFG as follows.**

**1. Make a variable *Ri* for each state *qi* of the DFA.**

**2. Add the rule *Ri* → *aRj* to the CFG if *(qi, a) = qj* is a transition in the DFA.**

**3. Add the rule *Ri* →  if *qi* is an accept state of the DFA.**

**4. Make *R0* the start variable of the grammar, where *q0* is the start state of the machine.**

**Verify on your own that the resulting CFG generates the same language that the DFA recognizes.**

**Moreover, certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to re­member an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring.**

**This situation occurs in the language {0n1n| *n ≥* 0} because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s.**

**You can construct a CFG to handle this situation by using a rule of the form *R* → *uRv,* which gen­erates strings wherein the portion containing the *u's* corresponds to the portion containing the *v’s.***

**Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. That situation occurs in the grammar that generates arithmetic expressions in the example below.**

**EXAMPLE 2.3:**

**Consider the grammar G4 = (V, , R, <EXPR>), where V is {<EXPR>, <TERM>, <FACTOR>},  is {a, +, \*, (, ) }. The rules are**

**<EXPR> → <EXPR> + <TERM> | < TERM>**

**<TERM> → <TERM> \* <FACTOR> | <FACTOR>**

**<FACTOR> → (<EXPR>) | a.**

**This grammar generates for example strings such as *a+a\*a,*  and (*(a+a)\*a*), and many others.**

**Any time the symbol *a* appears, an entire parenthesized expression might appear re­cursively instead.**

**To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.**

**Example 2.4**

**Given L = context free, show that L2 is also context free.**

**Answer: If S derives L then S1 🡪 SS derives L2.**

**Example 2.5**

**What language is generated by S 🡪 aSbb | a? Is it CF?**

**Answer: {an+1b2n, n >= 0}, Yes, it is.**

**MORE CFL & CFG EXAMPLES (For practice, work all these examples!)**

1. Use S 🡺 aSa | bSb |  is a CFG to derive aabbaa. Verify the general wwR case.

2. Use S 🡺 abB, A 🡺 aaBb, B 🡺 bbAa, A 🡺  to show the general case: ab(bbaa)nbba(ba)n, n ≥ 0.

3.The language L = {anbm, n ≠ m} has two cases: When n > m, use the grammar S 🡺 AB, B 🡺 aBb | , A 🡺 aA | a. For the case n < m the grammar is: S 🡺 AC | CB, C 🡺 aCb |  **,** A **🡺** aA | a, B **🡺** bB | b. The case m = n is well known. (Previously we have given some versions of its grammar).

4. The grammar S 🡺 aSb | SS | , generates the strings abaabb, aababb and ababab. This grammar is able to generate pair of parentheses of the type (()), ()()() and the set of all properly nested parentheses by just replacing a and b by the left and right parentheses respectively. The language in a and b is

L = {w in {a, b}\* : na(w) = nb(w) and na(v) ≠ nb(v) where v is any prefix of w}.

5. Use S 🡺 AB, A 🡺 aaA | , B 🡺 Bb |  to generate L(G) = {a2nbm, n >= 0, m >=0}. Just derive S 🡺 AB 🡺aaAB 🡺aaB 🡺 aaBb 🡺 aab (productions #1, 2, 3, 4 and 5) and S🡺 AB 🡺 ABb 🡺 aaABb 🡺 aaAb 🡺aab (productions #s 1, 4, 2, 5 and 3).

6. The grammar S 🡺 aAB, A 🡺 bBb, B🡺 A |  gives a rightmost derivation: S 🡺 aAB 🡺 aA 🡺 abBb 🡺 abAb 🡺 abbBbb 🡺 abbbb, or a leftmost derivation S 🡺aAB 🡺 abAbB 🡺 abbBbbB 🡺 abbbbB 🡺 abbbb.

**CHOMSKY NORMAL FORM (CNF)**

**When working with context-free grammars, it is often convenient to have them in simplified form. One of the simplest and most useful forms is called the Chom­sky normal form (CNF). We will find CNF useful when we are giving algorithms for working with context-free grammars.**

#### DEFINITION 2.5

**A context-free grammar is in *Chomsky normal form (CNF)* if every rule is of the form**

**A → BC**

**A → a**

**where *a* is any terminal and A, B, and C are any variables, except that B and C may not be the start variable. In addition, we permit the rule *S →* , where *S* is the start variable.**

**Example 6.7**

**The grammar S → AS | a; A → SA | b is in CNF. However, the grammar S → AS | AAS; A→ SA | aa is not in CNF because both productions S → AAS and A → aa violate the conditions of the definition of CNF.**

**THEOREM 2.6 (Theorem 6.6 in Linz’s book)**

**Any context-free language is generated by a context-free grammar in *Chomsky normal form*, CNF or equivalently, any CFG with ∉ L(G) has an equivalent grammar in CNF.**

**PROOF IDEA (and the algorithm itself!).**

**We can convert *any* context free grammar G into Chomsky normal form. The conversion has several stages wherein rules that violate the conditions are replaced by equivalent ones that are satisfactory.**

**First we add a new start symbol. Then we eliminate all *-rules* of the form A→ . We also eliminate all unit rules of the form A→ B.**

**In both cases the grammar is then patched up to be sure that it still generates the same language. Finally, we convert the remaining rules into proper form.**

#### PROOF (and PROCEDURE):

**First, we add a new start symbol *S0* and the rule *S0 → S*, where *S* was the original start symbol.**

**This change guarantees that the start symbol doesn't occur on the right-hand side of a rule.**

**Second, we take care of all  rules.**

**1. We remove a -rule *A →* , where *A* *is not the start variable*.**

**2. Then for each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted.**

**In other words, if *R → uAv* is a rule in which *u* and *v* are strings of variables and terminals, we add rule *R → uv*.**

**We do so *for each occurrence* of an A, so the rule *R → uAvAw* causes us to add *R → uvAw*, *R → uAvw*, and *R → uvw*.**

**If we have the rule R → A, we add R →  unless we had previously removed the rule R → .**

**We repeat these steps until we eliminate all  rules *not involving the start variable*.**

**Third, we handle all unit rules.**

**We remove a unit rule A → B. Then, when­ever a rule B → u appears, we add the rule A → u unless this was a unit rule previously removed.**

**As before, *u* is a string of variables and terminals.**

**We re­peat these steps until we eliminate all unit rules.**

**Finally, we convert all remaining rules into the proper form.**

**We replace each rule *A → u1u2....uk* where *k ≥ 3* and each *ui* is a variable or terminal symbol, with the rules *A → u1A1*, *A1→ u2A2*, *A2→ u3A3 , .... Ak-2→ uk-1uk*. The Ai's are new variables.**

**If k = 2, we replace any terminal *ui* in the preceding rule(s) with the new variable *Ui* and add the rule *Ui → ui.***

**EXAMPLE 2.7**

**Let G6 be the following CFG:**

**S → ASA | aB;**

**A → B | S;**

**B → b | ,**

**and convert it to Chomsky normal form by using the conversion procedure just given.**

**The following series of grammars illustrates the steps in the conversion. Rules shown underlined have been just added.**

**Rules shown in italic have just been removed.**

**1. The original CFG G6is shown on the left. The result of applying the first step to make a new start symbol appears on the right.**

**S0 → S**

**S → ASA | aB S → ASA | aB**

**A → B | S A → B | S**

**B → b |  B → b | **

**2. Remove rule *B →* , shown on the left, and also the rule *A →* , shown on the right.**

**S0 → S S0 → S**

**S → ASA | aB | a S → ASA | aB | a | SA | AS | S**

**A → B | S |  A → B | S | **

**B → b |  B → b**

**3a. Remove unit rules *S → S*, shown on the left, and *S0 → S*, shown on the right.**

**S0 → S S0 → *S* | ASA | aB | a | SA | AS**

**S → ASA | aB | a | SA | AS | *S* S → ASA | aB | a | SA | AS**

**A → B | S A → B | S**

**B → b B → b**

**3b. Remove unit rules *A → B* and *A → S*.**

**S0 → ASA | aB | a | SA | AS S0 → ASA | aB | a | SA | AS**

**S → ASA | aB | a | SA | AS S → ASA | aB | a | SA | AS**

**A → *B* | S | b A → *S* | b | ASA | aB | a | SA | AS**

**B → b B → b**

**4. Convert the remaining rules into the proper form by adding additional vari­ables and rules.**

**The final grammar in CNF is equivalent to *G* and appears as follows. (Actually the procedure given in the previous theorem produces several variables *Ui* along with several rules *Ui* → a.**

**We simplified the resulting grammar by using a single variable *U* and rule *U* → a.)**

**S0 → AA1 | UB | a | SA | AS**

**S → AA1 | UB | a | SA | AS**

**A → b | AA1 | UB | a | SA | AS**

**A1 → SA**

**U → a**

**B → b**

**Example 6.8 in Linz’s book gives as an example to convert to CNF the grammar**

**S → Aba**

**A → aab**

**B → Ac**

**And the final solution is obtain in two steps (no  – productions) plus addition of variables Ba Bb Bc:**

**S → ABBa**

**A → BaBbBc**

**B → ABc**

**Ba → a**

**Bb → b**

**Bc → c.**

**And then introduce D1 and D2 to get the first two productions S → ABBa; A → BaBbBc in CNF as follows: S → AD1; D1 → BBa; A → BaD2; D2 → BaBb. The rest of the productions are already in CNF and remain untouched.**

**Next, we solve an important problem related to CNF:**

**PROBLEM 2.19**

**Show that, if G is a CFG in Chomsky normal form, then for any string w ∈ L(G) of length n ≥ 1, exactly 2n-1 steps are required for any derivation of w.**

**SOLUTION**

**Every rule of a CNF grammar is of the form: A → BC**

**A → a**

**(except possibly for the rule S → , where S is the start variable. This rule isn’t used in the derivation of w because n ≥ 1).**

**Consider the derivation of w.**

**Each application of the rule of the form A → BC increases the length of the string by 1. So we have *n-1* steps here.**

**Besides that, we need exactly *n* applications of terminal rules (A → a) to convert the variables into terminals.**

**Therefore, exactly *2n – 1* steps are required.**