Ian Shin
Theory of Computation Extra Credit Problem
16 December 2024

2.4: Give context-free grammars that generate the following languages. In all parts, the alphabet Σ is {0,1}.

a. {w| w contains at least three 1s}

S → R1R1R1R
R → 0R | 1R | ε

- **S → R1R1R1R**: This rule ensures that the string w generated by the grammar contains at least three '1's. The non-terminal R can generate any string (including the empty string, ε) consisting of '0's and '1's. By embedding three '1's within non-terminals R, we guarantee that no matter what R produces, there will always be at least three '1's in the string.
- **R → 0R | 1R | ε**: This rule allows R to generate any sequence of '0's and '1's, including the possibility of generating an empty string. This flexibility ensures that R can precede or follow the '1's in the main rule with any combination of '0's and '1's or nothing at all.

This CFG is correct for generating strings over {0,1} that contain at least three 1s. The non-terminal R can generate sequences of '0's and '1's in any order and quantity, including the empty string, thus allowing the three '1's in the production from S to appear anywhere in the string, interspersed with any number of '0's.

b. {w| w starts and ends with the same symbol}

S → 0P0 | 1P1 | 0 | 1
P → 0P | 1P | ε

- **S → 0P0 | 1P1 | 0 | 1**: This production ensures that every string www generated by this grammar starts and ends with the same symbol. The rules 0P0 and 1P1 cover cases where the string is at least three characters long. The rules 0 and 1 cover the cases where the string consists of a single character, which trivially starts and ends with the same symbol.

- **P → 0P | 1P | ε**: This rule allows P to generate any sequence of '0's and '1's, including the possibility of generating an empty string (ε). When P generates an empty string, the productions 0P0 and 1P1 in S reduce to 00 and 11, respectively, which are strings of length two that start and end with the same symbol.

This CFG is correct for generating strings over {0,1} that start and end with the same symbol. The grammar effectively handles:

- Strings that are exactly one character long (0 and 1).
- Strings of two characters that start and end with the same symbol (00 and 11) by setting P to ε.
- Strings longer than two characters where the first and last characters are the same, with any combination of '0's and '1's in between, produced by P.

By ensuring that both the first and last characters are explicitly the same in the rules 0P0 and 1P1, and by allowing P to generate any sequence of '0's and '1's or nothing at all, the grammar correctly generates all strings where the first and last symbols are identical. This covers all cases required by the language description.

c. {w| the length of w is odd}

S → 0 | 1 | 00S | 01S | 10S | 11S

S → 0 | 1: These productions allow the generation of strings of length 1, which is odd.

S → 00S | 01S | 10S | 11S: Each of these productions adds two characters to the length of the string generated by SSS recursively. Since two is even, adding it to an already odd length (the recursive SSS) will result in another odd-length string. This method ensures that every string produced has an odd length because starting from 1 (odd), adding 2 (even) perpetually keeps the length odd.

d. {w| the length of w is odd and its middle symbol is a 0}

S → 0 | 0S0 | 0S1 | 1S0 | 1S1

S → 0: This production is the base case and covers the simplest form of the language. It generates a single-character string '0', which naturally has an odd length (1) and its middle symbol is '0'.

0S0 | 0S1 | 1S0 | 1S1: Each of these productions builds upon a smaller string that already meets the conditions of having an odd length and a middle '0'. By wrapping an existing valid string (S) with '0' or '1' on both sides, the resulting string grows by two characters, ensuring that the total length remains odd:

- **0S0 and 0S1**: These rules specifically maintain the condition that the middle symbol is '0' because, as the string from S is extended by two characters, the original middle '0' of S remains in the middle. For example, if S is "010" (middle '0'), extending it to "00100" or "00101" keeps '0' in the middle position.
- **1S0 and 1S1**: Although these rules extend S by surrounding it with '1's, the middle symbol from the original S (which is '0') remains in the middle of the extended string. For instance, extending "010" to "10100" or "10101" preserves the middle '0'.

e. {w| w = wR, that is, w is a palindrome}

S → 0 | 1 | 0S0 | 1S1 | ε

Explanation:

- **S → ε**: This rule allows the generation of the empty string, which is trivially a palindrome as it reads the same forward and backward (nothing in both directions).
- **S → 0 | 1**: These rules generate single-character strings '0' and '1'. Both of these strings are palindromes because a single character reads the same forward and backward.
- **S → 0S0 | 1S1**: These recursive productions are key to building longer palindromes. They allow the grammar to generate new palindromes by sandwiching an existing palindrome (generated by S) between matching characters. How each rule works:
    - **0S0**: This production takes any string that S generates (which, by definition, is a palindrome), and surrounds it with '0' on both ends. The result is still a palindrome because reversing it produces the same sequence: the reverse of the inner palindrome (which is the same as the original inner palindrome) flanked by '0's.
    - **1S1**: Similarly to the previous rule, this production surrounds any palindrome generated by S with '1's. The palindrome property is maintained because the sequence reads the same backward, with the inner palindrome being symmetrical and '1's matching on both ends.

f. The empty set

S → S

S → **S**: This rule is an example of a recursive production that does not lead to the creation of any terminal string. In simpler terms, the non-terminal SSS recursively replaces itself indefinitely and never resolves to a terminal symbol, which are the actual characters (like '0' or '1') used to construct strings in a language.