**Definitions of Important Concepts Related to Grammars**

**Definition 1:** A grammar G = (V,T,S,R) is called a *regular grammar* (RG) if all rules/productions of it (members of set R) are of either of the form (in case of left-regular grammar):

A → ε | a | aB

or of the form (in case of right regular grammar):

A → ε | a | Ba

Here A, B are members of V: set of variables/non-terminals in G,

a is a member of T: set of terminal symbols (symbols in alphabet) in G,

S is a variable that designates the start symbol of G

R = set of rules/productions

Note: Each regular grammar is equivalent to a regular expression and as such is equivalent to a DFA/NFA/ε-NFA

**E.g.1:** L1 = {0}\* = { ε , 0, 00, 000, ….}

A regular grammar to represent L1 is:

Set of terminals, T = {0}

Set of variables/non-terminals, V = {A}

Start symbol, S = A

Set of rules/productions, R = {A → ε | 0A }

Here the production/rule **A → 0** indicates that 0 is a member of L1.

And the rule **A → 0A** indicates that if A is a member of L1 then so is 0A.

**E.g.2:** L2 = {0,1}\* [RE = (0/1)\*]

A regular grammar to represent L2 is:

Set of terminals = {0,1}

Set of variables/non-terminals = {B}

Start symbol = B

Set of rules/productions = {B → ε | B0 | B1 } [ALT: {B → ε | 0B | 1B }]

L2 = { ε, 0, B=1, 00, 01, B0=10, B1=11,…..}

When you are asked to design a grammar for a given language you must ensure that (1) All the strings in that language can be generated using your grammar (no false-negative) AND (2) All strings that can be generated by your grammar are members of the given language (no false-positive). In other words, the language of your grammar must be exactly equal to the given language; it can’t be a subset/superset of the given language.

Same holds for automata design. If you are asked to design an automaton for language L; the language of your automaton must be equal to (not subset/superset) L.

**E.g.3:** L3 = set of binary strings that end with 101. Derive 110101 using your grammar.

**Solution:**

A = (0/1)\*101 = B 101

A regular grammar to represent L3 is:

Set of terminals = {0,1}

Set of variables/non-terminals = {A, B}

Start symbol = A

Set of rules/productions = {A → B101, B → ε / B0 / B1 }

Derivation of 110101 using this grammar:

A => B101

=> B0101 [ applying B → B0 ]

=> B10101 [ applying B → B1 ]

=> B110101 [ applying B → B1 ]

=> 110101 [ applying B → ε]

**ALT. Solution:**

Set of terminals = {0,1}

Set of variables/non-terminals = {B}

Start symbol = B

Set of rules/productions = { B → 101 / 0B / 1B }

Derivation of 110101 using this grammar:

B => 1B [ applying B → 1B ]

=> 11**B** [ applying B → 1B ]

=> 11**0B** [ applying B → 0B ]

=> 110101 [ applying B → 0B ]

**E.g.4:** L4 = set of binary strings that start with 101

A = 101 (0/1)\* = 101B

A regular grammar to represent L4 is:

Set of terminals = {0,1}

Set of variables/non-terminals = {A, B}

Start symbol = A

Set of rules/productions = {A → 101B, B → ε | B0 | B1 }

ALT. Solution:

Set of terminals = {0,1}

Set of variables/non-terminals = {B}

Start symbol = B

Set of rules/productions = {B → 101 | B0 | B1}

**E.g.5:** L5 = set of binary strings that contain 101 as a substring

A = (0/1)\*101 (0/1)\*

B 101 B

A regular grammar to represent L5 is:

Set of terminals = {0,1}

Set of variables/non-terminals = {A, B}

Start symbol = A

Set of rules/productions = {A→B101B, B → ε | B0 | B1 }

**Definition 2:** A grammar G = (V,T,S,R) is called a Context Free Grammar (CFG) if all rules/productions of it (members of set R) are of the form:

A → γ

Here γ is a member of (T U V)\* i.e. γ is a string (possibly empty) formed by terminal and/or non-terminal (i.e. variables) symbols.

A CFG (which can’t be represented using RG) for binary palindromes:

Set of terminals = {0,1}

Set of variables = {S}

Start symbol = S

Set of rules/production =

{

S → ε / 0 / 1, …. (1) [base case]

S → 1S1 / 0S0 ..... (2) [general case]

}

Language of binary palindromes = { ε, S=0, 1, 00, 11, 0S0=000, 1S1=101, ….}

Another CFG for L = {0n1n: n≥1} = {01, S=0011, 0S1=0**0011**1, ….}:

Set of terminals = {0,1}

Set of variables = {S}

Start symbol = S

Set of rules/production =

{

S → 01 …. (1) [base case]

S → 0S1 ..... (2) [general case]

}

Derivation of **000111** using this grammar:

S => 0S1 [using rule (2)]

=> **00**S**11** [using rule (2)]

=> **000111** [using rule (1)]

Another CFG for L = {0n1n: n≥2} = {0011, 000111, ….}:

Set of terminals = {0,1}

Set of variables = {S}

Start symbol = S

Set of rules/production =

{

S → 0011 …. (1) [base case]

S → 0S1..... (2) [general case]

}

Derivation of 00001111 using this grammar:

S => 0S1 [using rule (2)]

=> 00**S**11 [using rule (2)]

=> 00001111 [using rule (1)]

**Practice Problems:** Write CFG for the following languages

* L0 = {(01)n: n≥0} = {ε, S=01, 01S=0101, 010101, …}
* L1 = {0n1n: n≥0} = {ε, 01, S=0011, 0S1=000111, …}
* L2 = {0n+11n: n≥0} = {0}{0n1n: n≥0} = {0} L1

A = 0 S

L2 = {0, S=001, 0S1=00011, 0000111, ….}

* L3 = {0n1n+2: n≥0} = {0n1n: n≥0}{11} = L1{11}
* L4 = {0n12n: n≥0}
* L5 = {02n1n: n≥0}. Derive 000011 using your CFG.
* L6 = {02n1n+1: n≥0}
* L7 = {02n1n+1: n≥1}
* L8 = {0n+112n: n≥1}
* L9 = Set of balanced parentheses = {(), (()), ()(), ()()(), (())(()()),… }
* L10 = set of odd length binary palindromes
* L11 = {0m1n: n>m≥0}={0m1m1n-m: n>m≥0}

= {0m1m:m ≥ 0}{1n-m: n-m≥1} = L1{1}+

B ->SA

* L12 = {0n+11n-m: n>m≥0}
* L13 = set of binary strings containing exactly two 1s
* L14 = set of binary strings containing at least two 1s
* L15 = set of binary strings containing at most two 1s
* L16 = set of binary strings containing odd number of 1s

**Solutions:**

**L0:** Set of terminals = {0,1}, Set of variables/non-terminals = {S}, Start symbol = S

Set of rules = {S → ε / 01S } ALT: {S → ε / S01 }

**L1:** Set of terminals = {0,1}, Set of variables/non-terminals = {S}, Start symbol = S

Set of rules = {S → ε / 0S1 }

**L2:** Set of terminals = {0,1}, Set of variables/non-terminals = {S,A}, Start symbol = A, Set of rules = {A → 0S, S → ε / 0S1 }

**ALT SOLUTION:** Set of terminals = {0,1}, Set of variables/non-terminals = {S}, Start symbol = S, Set of rules = {S → 0 / 0S1 }

**L3:** Set of terminals = {0,1}, Set of variables/non-terminals = {A,S}, Start symbol = A, Set of rules = {A → S11, S → ε / 0S1 }

**ALT. SOLUTION:** Set of terminals = {0,1}, Set of variables/non-terminals = {S}, Start symbol = S, Set of rules = {S → 11 / 0S1 }

**L4:** Set of terminals = {0,1}, Set of variables/non-terminals = {S}, Start symbol = S

Set of rules = {S → ε / 0S11 }

**L5:** Set of terminals = {0,1}, Set of variables/non-terminals = {S}, Start symbol = S

Set of rules = {S → ε / 00S1 }

Derivation of 000011:

S => 00S1 [applying S → 00S1]

=> 0000S11 [applying S → 00S1]

=> 000011 [applying S → ε]

S =>\* 000011

**L6:** Set of terminals = {0,1}, Set of variables/non-terminals = {S}, Start symbol = S

Set of rules = {S → 1 / 00S1 }

**L7:** Set of terminals = {0,1}, Set of variables/non-terminals = {S}, Start symbol = S

Set of rules = {S → 0011 / 00S1 }

Derivation of 0000001111 using this CFG:

S => 00S1 [applying S → 00S1]

=> 0000S11 [applying S → 00S1]

=> 0000001111 [applying S → 0011]

**L11**: Set of terminals = {0,1}, Set of variables/non-terminals = {S,A,B},

Start symbol = B, Set of rules = {B → SA, S → ε / 0S1, A → 1 / 1A}

Leftmost Derivation of 0011111:

B => SA [B → SA]

=> 0S1A [S → 0S1]

=> 00S11A [S → 0S1]

=> 0011**A** [S → ε]

=> 0011**1A** [A → 1A]

=> 001111A [A → 1A]

=> 0011111 [A → 1]

Rightmost Derivation of 0011111:

B => SA [B → SA]

=> S1**A** [A → 1A]

=> S1**1A** [A → 1A]

=> S111 [A → 1]

=> 0S1111 [S → 0S1]

=> 00S11111 [S → 0S1]

=> 0011111 [S → ε]

**Definition 3:** A CFG is called ambiguous if it is possible to derive the same string using that CFG via multiple leftmost derivations.

ALT. Definition: A CFG is called ambiguous if it is possible to derive the same string using that CFG via multiple rightmost derivations.

It’s hard to implement a compiler using ambiguous grammar.

E.g. of ambiguous grammar:

set of terminals, T = {+,-, id, const}

set of variables, V = {E}

set of rules, R = {E → E + E | E – E | id | const}

start symbol, S = E

**Practice on Ambiguous grammar:**

E.g. grammar:

set of terminals, T = {+,-, id, const}

set of variables, V = {E}

set of rules, R = {E → E + E | E – E | id | const}

start symbol = E

Show a leftmost derivation of the string “id + id - const”:

E=>**E**-E

=>**E+E**-E

=>id+E-E

=>id+id-**E**

=>id+id-**const**

Show another leftmost derivation of the string “id + id - const”:

E=>**E**+E

=>**id**+E

=>id+**E**-E

=>id+**id**-E

=>id+id-const

Since there are multiple leftmost derivations for the same string using this grammar, this grammar is ambiguous.

Note: it’s also possible to show that “id + id - const” can be derived using this grammar via multiple rightmost derivations and therefore this grammar is ambiguous.

**E.g. of another ambiguous grammar:**

set of terminals, T = {(, )}

set of variables, V = {B}

set of rules, R = {B → (B) | () | BB}

start symbol, S = B

Show a leftmost derivation of “()()()”:

S = B

=> **B**B

=> **BB**B

=> ()BB

=> ()()B

=> ()()()

Show another leftmost derivation of “()()()”:

S = B

=> **B**B

=> **()**B

=> ()BB

=> ()()B

=> ()()()

Since this grammar can be used to derive the same string “()()()” via multiple leftmost derivations, this grammar is ambiguous.

Language of CFG, G = (V,T,S,R) is defined as:

L(G) = {w ε T\*: S =>\* w}