Exercises for Section 2.2

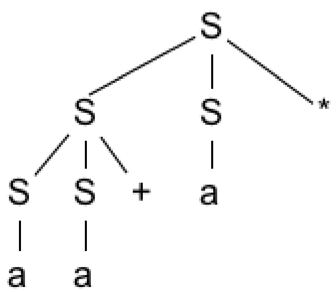
2.2.1

Consider the context-free grammar:

S -> S S + | S S * | a

1. Show how the string `aa+a*` can be generated by this grammar.
2. Construct a parse tree for this string.
3. What language does this grammar generate? Justify your answer.

Answer

1. `S` -> `S` S * -> `S` S + S * -> a `S` + S * -> a a + `S` * -> a a + a *
2. 

3. L = {Postfix expression consisting of digits, plus and multiple signs}

2.2.2

What language is generated by the following grammars? In each case justify your answer.

1. S -> 0 S 1 | 0 1
2. S -> + S S | - S S | a
3. S -> S (S) S | ε
4. S -> a S b S | b S a S | ε
5. S -> a | S + S | S S | S * | (S)

Answer

1. L = $\{0^n1^n \mid n>=1\}$
2. L = {Prefix expression consisting of plus and minus signs}
3. L = {Matched brackets of arbitrary arrangement and nesting, includes ε}
4. L = {String has the same amount of a and b, includes ε}
5. L = {Regular expressions used to describe regular languages} refer to wiki

2.2.3

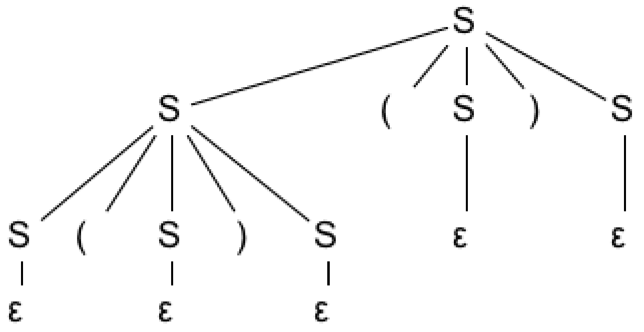
Which of the grammars in Exercise 2.2.2 are ambiguous?

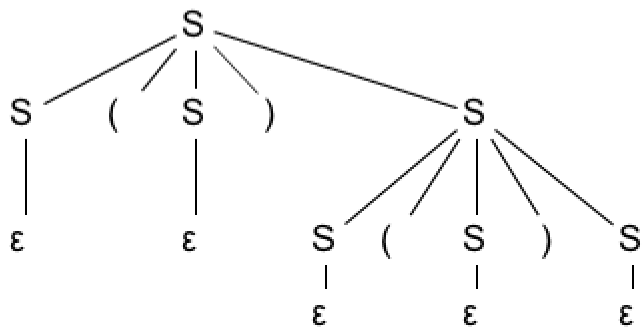
Answer

1. No

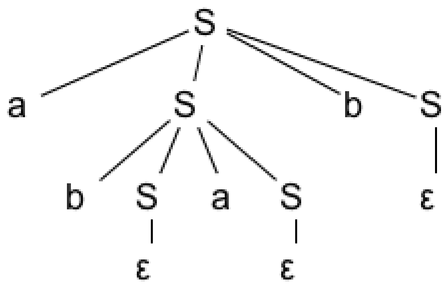
2. No

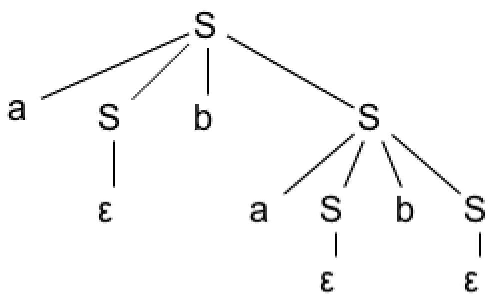
3. Yes



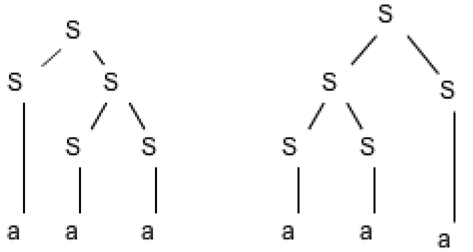


4. Yes





5. Yes



2.2.4

Construct unambiguous context-free grammars for each of the following languages. In each case show that your grammar is correct.

1. Arithmetic expressions in postfix notation.
2. Left-associative lists of identifiers separated by commas.
3. Right-associative lists of identifiers separated by commas.
4. Arithmetic expressions of integers and identifiers with the four binary operators +, -, *, /.
5. Add unary plus and minus to the arithmetic operators of 4.

Answer

```
1. E -> E E op | num

2. list -> list , id | id

3. list -> id , list | id

4. expr -> expr + term | expr - term | term
   term -> term * factor | term / factor | factor
   factor -> id | num | (expr)

5. expr -> expr + term | expr - term | term
   term -> term * unary | term / unary | unary
   unary -> + factor | - factor | factor
   factor - > id | num | (expr)
```

2.2.5

1. Show that all binary strings generated by the following grammar have values divisible by 3. Hint. Use induction on the number of nodes in a parse tree.

   num -> 11 | 1001 | num 0 | num num

2. Does the grammar generate all binary strings with values divisible by 3?

Answer

1. Proof

   Any string derived from the grammar can be considered to be a sequence consisting of 11 and 1001, where each sequence element is possibly suffixed with a 0.

   Let `n` be the set of positions where `11` is placed. `11` is said to be at position `i` if the first `1` in `11` is at position `i`, where `i` starts at 0 and grows from least significant to most significant bit.

Let `m` be the equivalent set for `1001`.

The sum of any string produced by the grammar is:

sum

$= \Sigma_n (2^1 + 2^0) * 2^n + \Sigma_m (2^3 + 2^0) * 2^m$

$= \Sigma_n 3 * 2^n + \Sigma_m 9 * 2^m$

This is clearly divisible by 3.

2. No. Consider the string "10101", which is divisible by 3, but cannot be derived from the grammar.

   Readers seeking a more formal proof can read about it below:

   Proof:

   Every number divisible by 3 can be written in the form `3k`. We will consider `k > 0` (though it would be valid to consider `k` to be an arbitrary integer).

   Note that every part of num(11, 1001 and 0) is divisible by 3, if the grammar could generate all the numbers divisible by 3, we can get a production for binary k from num's production:

```
3k = num   -> 11 | 1001 | num 0 | num num
 k = num/3 -> 01 | 0011 | k 0   | k k
 k         -> 01 | 0011 | k 0   | k k
```

   It is obvious that any value of `k` that has more than 2 consecutive bits set to 1 can never be produced. This can be confirmed by the example given in the beginning:

   10101 is 3*7, hence, k = 7 = 111 in binary. Because 111 has more than 2 consecutive 1's in binary, the grammar will never produce 21.

2.2.6

Construct a context-free grammar for roman numerals.

Note: we just consider a subset of roman numerals which is less than 4k.

Answer

[wikipedia: Roman_numerals](https://en.wikipedia.org/wiki/Roman_numerals)

- via wikipedia, we can categorize the single roman numerals into 4 groups:

```
I, II, III | I V | V, V I, V II, V III | I X
```

then get the production:

```
digit -> smallDigit | I V | V smallDigit | I X
smallDigit -> I | II | III | ε
```

- and we can find a simple way to map roman to arabic numerals. For example:
  - XII => X, II => 10 + 2 => 12
  - CXCIX => C, XC, IX => 100 + 90 + 9 => 199
  - MDCCCLXXX => M, DCCC, LXXX => 1000 + 800 + 80 => 1880
- via the upper two rules, we can derive the production:

  romanNum -> thousand hundred ten digit

  thousand -> M | MM | MMM | ε

  hundred -> smallHundred | C D | D smallHundred | C M

  smallHundred -> C | CC | CCC | ε

  ten -> smallTen | X L | L smallTen | X C

  smallTen -> X | XX | XXX | ε

  digit -> smallDigit | I V | V smallDigit | I X

  smallDigit -> I | II | III | ε