Compile Principle - HW of Chapter 3

解雲暄　3190105871

> 3.2 3.3 3.4

3.2

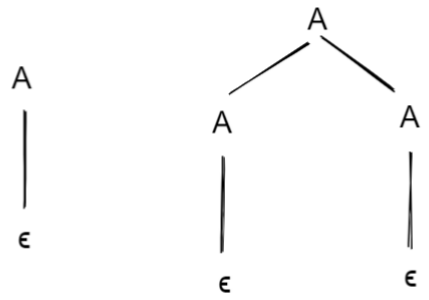
Given the grammar A→AA|(A)|ϵ

a. Describe the language it generates.

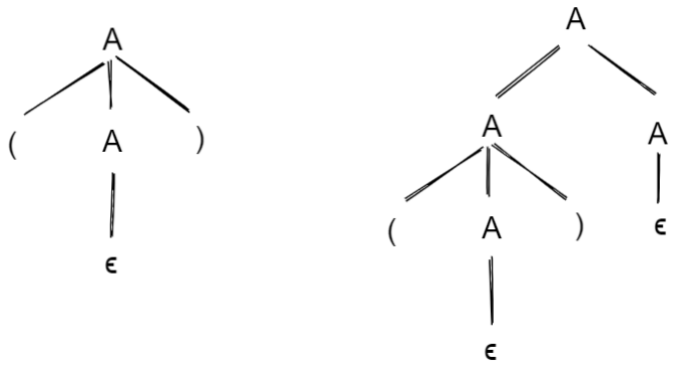
The grammar generates all "balanced parentheses" (like Example 3.5 of the text book in p.105), i.e. all strings with paired parentheses. ϵ (empty string) also included.

b. Show it is ambiguous.

If a grammar generates a string with 2 different parsing trees, we say it is ambiguous. So here is an example, the following 2 parsing trees can both generate an empty string:



Or the following 2 parsing trees can both generate string "()":



3.3

Given the grammar:*

```
exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → *
factor → (exp) | factor | number
```

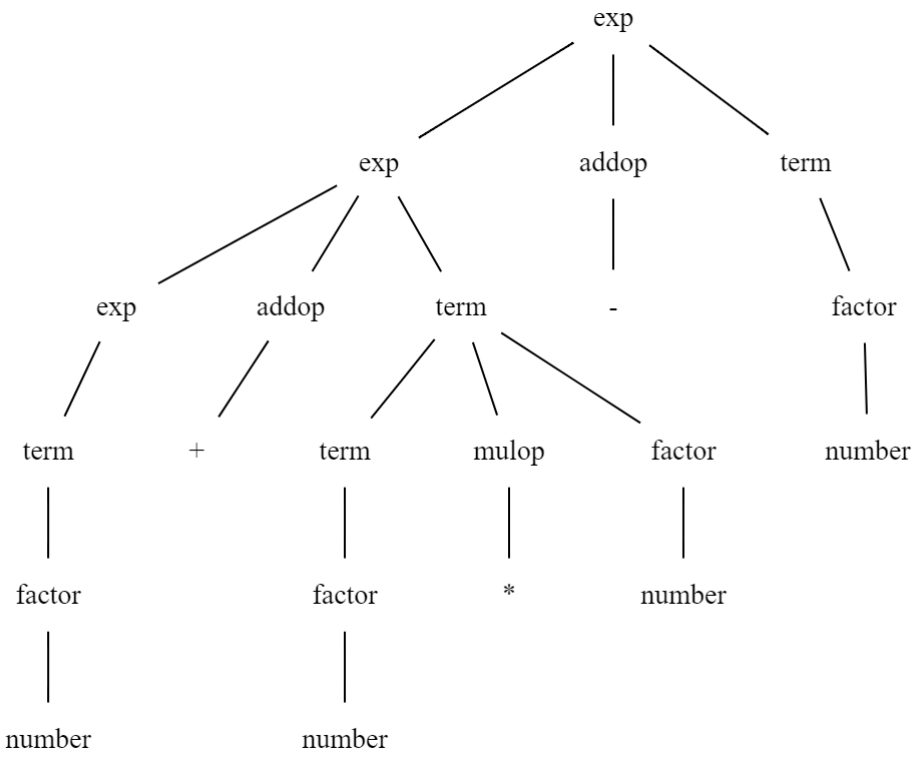
Write down leftmost derivation, parse trees, and absctract syntax trees for the following expressions:

(a) `3+4*5-6`

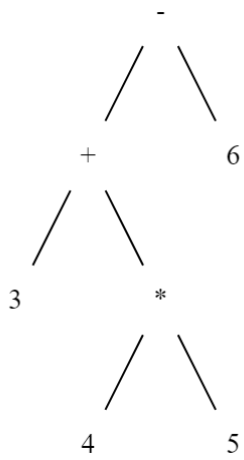
Leftmost derivation:

exp
=> *exp* addop term
=> *exp* addop term addop term
=> *term* addop term addop term
=> *factor* addop term addop term
=> **number** *addop* term addop term
=> **number +** *term* addop term
=> **number +** *term* mulop factor addop term
=> **number +** *factor* mulop factor addop term
=> **number + number** *mulop* factor addop term
=> **number + number *** *factor* addop term
=> **number + number * number** *addop* term
=> **number + number * number -** *term*
=> **number + number * number -** *factor*
=> **number + number * number - number**

Parse tree:



Abstract syntax tree:



3.4

The following grammar generates all regular expressions over the alphabet of letters (we have to use quotes to surround operators, since the vertical bar is an operator as well as a metasymbol):

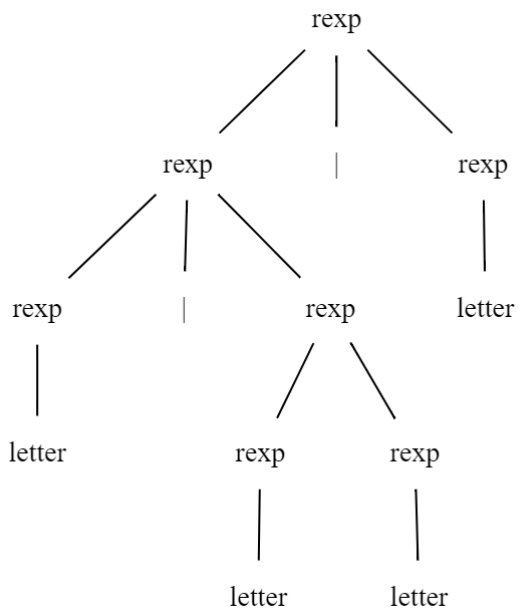
```
rexp → rexp “|” rexp
    | rexp rexp
    | rexp “*”
    | “(” rexp“)”
    | letter
```

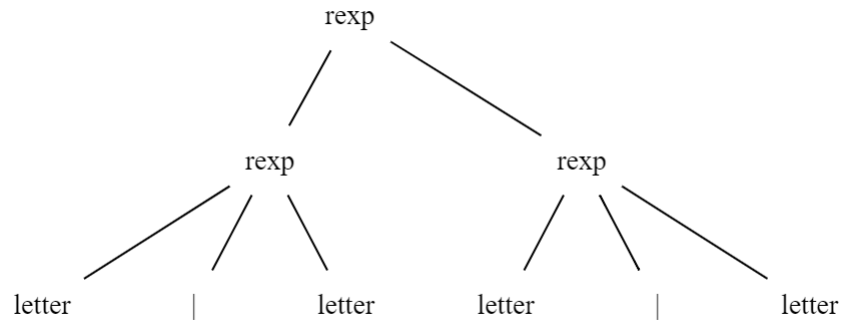
a. Give a derivation for the regular expression `(ab|a)*` using this grammar.

rexp => rexp * => (rexp) * => (rexp | rexp) * => (rexp rexp | rexp) * => (letter rexp | rexp) * => (letter letter | rexp) * => (letter letter | letter) *

b. Show that this grammar is ambiguous.

For example, 2 parse tree below can both generate regular expression `a|bc|d` :





c. Rewrite this grammar to establish the correct precedences for the operations (see chapter 2).

- The parentheses and letters have the highest precedence as they are atomic.
- "*" has higher precedence than concatenation (i.e. `rexp -> rexp rexp`) as we tend to consider `ab*` as `a(b*)` but not `(ab)*`.
- Concatenation has higher precedence than "|" as we tend to consider `a|bc|d` as `a|(bc)|d` but not `(a|b)(c|d)`.

  i.e. **letter** = **()** > ***** > · > **|**

  Therefore, we establish a different rule for each level of precedence and get the grammar below:
  (Here **rexp** stands for **regular expression**, **cat** stands for **concatenation**, **rept** stands for **repetition**, **atmc** stands for **atomic**, i.e. parentheses and letters.)

```
rexp -> rexp "|" cat
      | cat
cat  -> cat rept
      | rept
rept -> rept "*"
      | atmc
atmc -> "(" rexp ")"
      | letter
```

d. What associativity does your answer in part (c) give to the binary operations? Why?

Left associativity. Because we've used left recursion rules in our grammar.

But right associativity is also acceptable as the alternation ("|") and concatenation are both associative and "*" and "()" are unary operators so although right associativity will lead to different syntax trees, but the semantic value is the same. That is, this is an inessential ambiguity.