Chapter 2

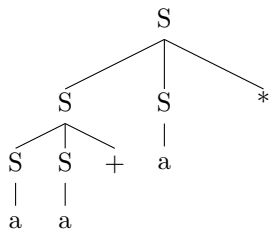
2.2.7 Excercises for Section 2.2

Excercise 2.2.1

a)

$$S \rightarrow S1\ S2\ *$$
$$S1 \rightarrow S3\ S4\ +$$
$$S2 \rightarrow a$$
$$S3 \rightarrow a$$
$$S4 \rightarrow a$$

b)



c) The language generated is the post-fix notation of numbers with multiplication and addition operands.

Excercise 2.2.2

a) The language created is 0^n1^n, where $n \in N^*$.

b) This language is the prefix notation of the addition and difference of the digit a.

c) The language is $[(^n)^n]^m$, where $m, n \in N$ and for every diffferent m the n is different, so closed parenthesis of any depth and length.

d) The language is $(a^n b^n)^m$, where $m, n \in N$ and for different m, the n is also different. So different sequences of a and b where both letter have the same number of appearances.

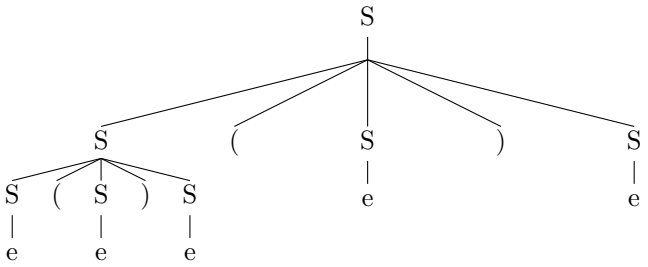
e) This is a grammar to create regular languages (Wikipedia link).

2.2.3

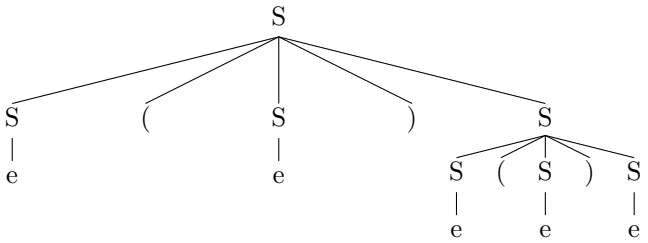
The grammars that are ambiguous are:

- Grammar c: Creating the string "()()" can be done in two ways

  Way A:

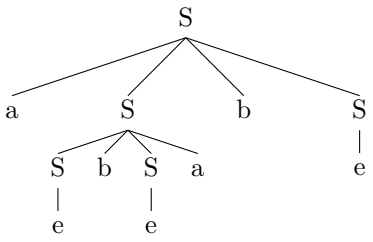


  Way B:

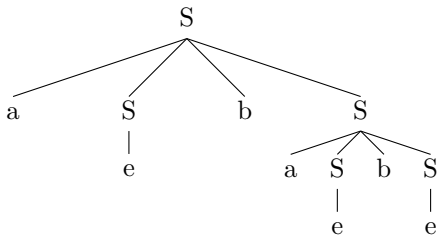


- Grammar d: Creating the string "abab" can be done in two ways:

  Way A:

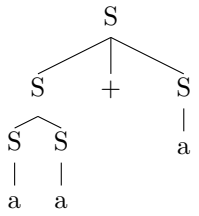


  Way B:

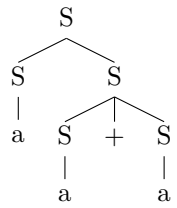


- Grammar e: Creating the string "a a+a" can be done in two ways:

  Way A:



  Way B:



Excercise 2.2.4

a) This is called reverse polish notation (Wikipedia Link)

$$expr \rightarrow expr\ expr\ \text{op} \mid \text{digit}$$

b)

$$list \rightarrow list\ , \text{id} \mid \text{id}$$

c)

$$list \rightarrow \text{id}\ , list \mid \text{id}$$

d)

$$expr \rightarrow expr\ + factor \mid expr\ -\ factor \mid factor$$
$$factor \rightarrow factor * \text{digit} \mid factor / digit \mid digit$$
$$digit \rightarrow \text{id} \mid \text{integer} \mid (expr)$$

e)

$$expr \rightarrow expr\ +factor \mid expr\ -\ factor \mid factor$$
$$factor \rightarrow factor * \text{digit} \mid factor/term \mid term$$
$$term \rightarrow +digit \mid -digit \mid digit$$
$$digit \rightarrow \text{id} \mid \text{integer} \mid (expr)$$

Excercise 2.2.5

a) We can easily see that the terminal strings 11 and 1001 are both divisible by 3, since they are the binary 3 and 9. These will always be the leaves of the parse tree.

By going one level up, there are two options:

1) The first one is the non-terminal $num\ 0$:

This operation is the equevalent of multiplying the number we have by two in the decimal system.

It is easy also to see that this non-terminal, is a multiplicate of two, since $3 * 2$ or $9 * 2$ is divisible by 3.

We can also see that going any numbers of levels up on the parse tree and having a number divisible by 3, and having only this non-terminal, the production will be divisible by 3, since $2^n * 3 * (number \div 3)$ is divisible by 3.

2) The second one is the non-terminal $num\ num$.

We assume that both nums are divisible by 3, throught the other 3 productions that we saw.

But then the operation will be the number $\text{num1} * 2^n + \text{num2}$, where n is the number of digits of num2.

Since num1 and num2 are divisible by 3, we can rewrite the previous equation as:

$$3 * (\text{num1} \div 3) * 2^n + 3 * (\text{num2} \div 3) =$$
$$3 * ((\text{num1} \div 3) * 2^n + (\text{num2} \div 3))$$

So that product will be divisible by 3.

Now if in a higher node we have two nums that are created by this production and they are parts of another production, then again thanks to all the previous mentioned productions, they will be divisible by 3.

So all productions of this grammar are divisible by 3.

b) No, the grammar cannot generate the number 10101 (21_{dec}), so it doesn not create all teh strings divisible by 3.

Excercise 2.2.6

Information about roman numerals can be found here.

mill → CM *tenths* | M *mill* | *fhundr*
fhundr → CD *tenths* | D *tenths* | DC *tenths* | DCC *tenths* | DCCC *tenths* | *hundr*
hundr →