Home Work 1 Solutions

**4.4.4: c)**

S -> S ( S ) S | ε

( ( ) ( ) )

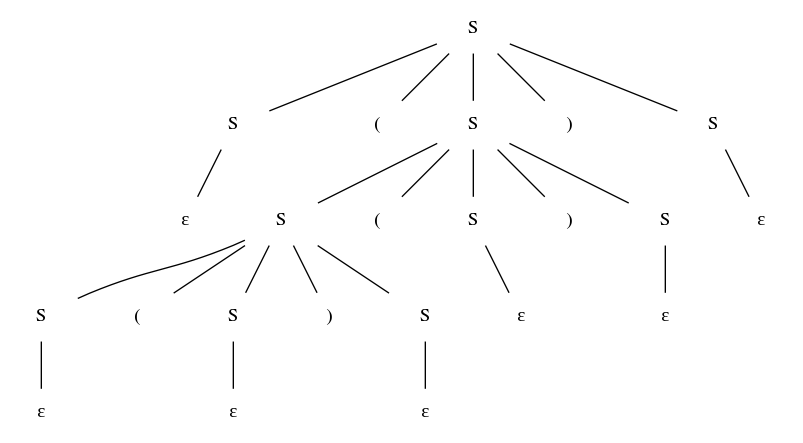
1. Leftmost derivation:

S => S ( S ) S => ( S ) S => ( S ( S ) S ) S => ( S ( S ) S ( S ) S ) S =>( ( S ) S ( S ) S ) S =>( ( ) S ( S ) S ) S => ( ( ) ( S ) S ) S => ( ( ) ( ) S ) S => ( ( ) ( ) ) S => ( ( ) ( ) )

1. Rightmost derivation:

S => S ( S ) S =>S ( S ) => S ( S ( S ) S ) =>S ( S ( S ) S ( S ) S )=>S ( S ( S ) S ( S ) )=>S ( S ( S ) S ( ) ) =>S ( S ( S ) ( ) ) => S ( S ( ) ( ) ) => S ( ( ) ( ) ) => ( ( ) ( ) )

1. Parse tree:



1. The grammar is ambiguous. The above string has 2 parse trees.
2. All valid parenthesis.

**d)**

S -> S + S | S S | ( S ) | S \* | a

(a + a) \* a

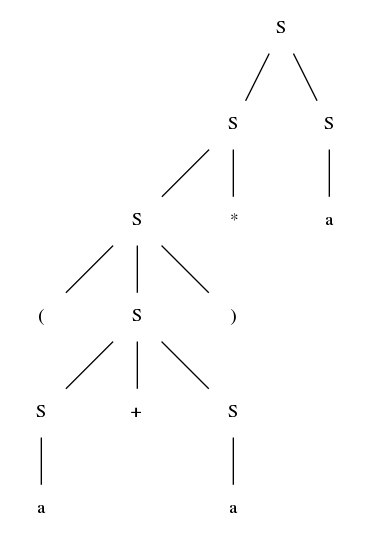
1. Leftmost derivation:

S => S S => S \* S => ( S ) \* S => ( S + S ) \* S => ( a + S ) \* S => ( a + a ) \* S => ( a + a ) \* a

1. Rightmost derivation:

S => S S => S a => S \* a => ( S ) \* a => ( S + S ) \* a => ( S + a ) \* a => ( a + a ) \* a

1. Parse tree:



1. The grammar is ambiguous. The given string has a unique parse tree but the string a + a \* a has 2 parse trees.
2. Regular expressions on ‘a’ with +, \* and parenthesis.

**4.2.4:**

Given any production rule with the extended grammar notation, we show how to represent the same language by using the original grammar notation.

1. A -> X [ Y ] Z can be replaced by A -> X Z | X Y Z
2. A -> X { Y Z } can be replaced by A -> X | X A’, A’ -> Y Z A’ | Y Z

**4.4.1: c)**

The left recursion and ambiguity of the grammar can be removed by replacing the production rule

S -> S ( S ) S by S -> ( S ) S.

Using the production rules:

S -> ( S ) S | ε,

|  |  |  |  |
| --- | --- | --- | --- |
| Symbol | ( | ) | $ |
| S | S -> ( S ) S | S -> ε | S -> ε |

**d)** The left recursion of the grammar can be removed by replacing the production rules

S -> S + S | S S | ( S ) | S \* | a

Using the production rules:

S -> ( S ) S’ | a S’

S’ -> + S S’ | S S’ | \* S’ | ε

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Symbol** | **\*** | **+** | **(** | **)** | **a** | **$** |
| S |  |  | S -> ( S ) S’ |  | S -> a S’ |  |
| S’ | S’ -> \* S’ | S’ -> + S S’ | S’ -> S S’ | S’ -> ε | S’ -> S S’ | S’ -> ε |

Grammar is ambiguous, hence, in S’, for symbol (, a, there are two valid production rules,

S’ -> S S’ and S’ -> ε. We fix the ambiguity by removing the production rule S’ -> ε (Why not the other one?).

**4.4.4: c)**

For the original grammar:

First (S) = { ( }

Follow (S) = { (, ), $ }

For the modified grammar of previous question:

First (S) = { ( }

Follow (S) = { ), $ }

**d)**

For the original grammar:

First (S) = { a, ( }

Follow (S) = { +, \*, a, (, ), $ }

For the modified grammar of previous question:

First (S) = { a, ( }

Follow (S) = { +, \*, a, (, ), $ }

First (S’) = { +, \*, (, a, ε }

Follow (S’) = { +, \*, a, (, ), $ }

**4.4.6: a)**

Algorithm: Remove epsilon production rules of a grammar

Input: A grammar G with epsilon production rules

Output: A grammar without epsilon production rules [note that there will be one epsilon production rule if ε is in the language of the grammar]

Method:

Step 1: compute set of nullable non-terminals:

NS = { N | N is a nonterminal and N -> ε is a production rule in G }

While NS does not change:

For each production rule L -> R1 R2 … Rk:

if R1, R2, … Rk are in NS, add L to NS

Step 2: remove epsilon production rules:

remove all epsilon production rules in G

for each N in NS:

for each production rule A -> B N C in G:

add the production rule A -> B C

Step 3: If ε is in the language of G, add a new non terminal S’ as the start symbol,

add the production rule

S’ -> S | ε

where S was the original start symbol.

**b)** S -> a S b S | b S a S | ε

NS = { S }

S -> a b | a S b | a b S | a S b S | b a | b S a | b a S | b S a S

S’ -> S | ε