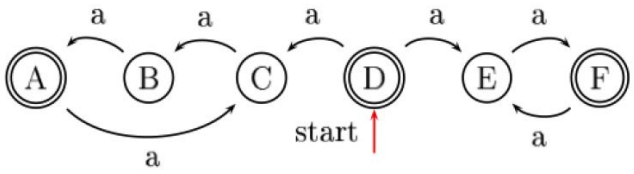
Theoretical Section

(a) Write a **context-free, right-linear grammar** that generates the language described by the following NFA:



Solution:

We let D be the start symbol:

$$D \to \mathrm{a}C \mid \mathrm{a}E \mid \varepsilon$$
$$E \to \mathrm{a}F$$
$$F \to \mathrm{a}E \mid \varepsilon$$
$$C \to \mathrm{a}B$$
$$B \to \mathrm{a}A$$
$$A \to \mathrm{a}C \mid \varepsilon$$

(b) Describe a general method by which **any** NFA can be converted into a grammar of this form.

Solution:

Let $(Q, \Sigma \cup \{\varepsilon\}, \delta, q_s, F)$ be an NFA, where:

Q - A set of states.

Σ - A set of input symbols.

δ - The transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathrm{P}(Q)$.

q_s - The start state ($q_s \in Q$).

F - The set of accepting states ($F \subseteq Q$).

Here's an algorithm to convert this NFA to a context-free, right-linear grammar:

1. Set the q_s to be the name of the start symbol.
2. For every state $q \in Q$, do the following:
   a. If $q \in F$, add the rule $q \to \varepsilon$.
   b. For every $q' \in \delta(q, \varepsilon)$, add the rule $q \to q'$.
   c. For every $\mathrm{a} \in \Sigma$ and $q' \in \delta(q, \mathrm{a})$, add the rule $q \to \mathrm{a}q'$.

The grammar produced by this algorithm is **context-free** since the left-hand side of every rule contains only a non-terminal. It is **right-linear** since the right-hand side of every rule contains at most one non-terminal.

(c) Recall that the naïve expression grammar:

$$E \to E+E \mid E-E \mid E*E \mid E/E \mid \# \mid (E)$$

is ambiguous and therefore not suitable for parsers.

Someone suggested that we use layering to write an equivalent unambiguous grammar:

$$E \to T+E \mid T-E \mid T$$
$$T \to F*T \mid F/T \mid F$$
$$F \to \# \mid (E)$$

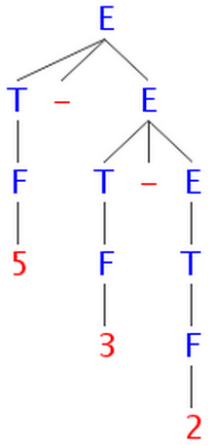
What is the problem with this formulation?

Illustrate your claim by an example for an expression and its respective parse tree.

Solution:

The problem with this formulation is that it makes subtraction right-associative, instead of left-associative (It actually makes all of the arithmetic operators right-associative, but it is especially problematic for subtraction).

For example, the expression: $5-3-2$ will return the following parse tree:



The tree corresponds to the expression $5-(3-2)$, which equals 4.

Yet by the laws of arithmetic, we expect it to mean $(5-3)-2$, which equals 0.

(d) You are presented with the following CFG:

$$S \to AC$$
$$A \to \mathrm{a}A \mid b$$
$$C \to CD \mid \mathrm{c}$$
$$D \to \mathrm{d}$$

where a, b, c, d are terminals.

Is this grammar LL(1)? If not, fix it so that it would be.

Construct the FIRST and FOLLOW sets, fill the LL(1) parse table, and show how an LL(1) parser would process the input “abcdd”.

Solution:

It’s not an LL(1) grammar because C is left recursive. In recursive-descent parsing, we can only expand the leftmost non-terminal at each step, so we will have infinite loop.

Fixed grammar:

$$S \to AC$$
$$A \to \mathrm{a}A \mid \mathrm{b}$$
$$C \to \mathrm{c}C'$$
$$C' \to DC' \mid \varepsilon$$
$$D \to \mathrm{d}$$

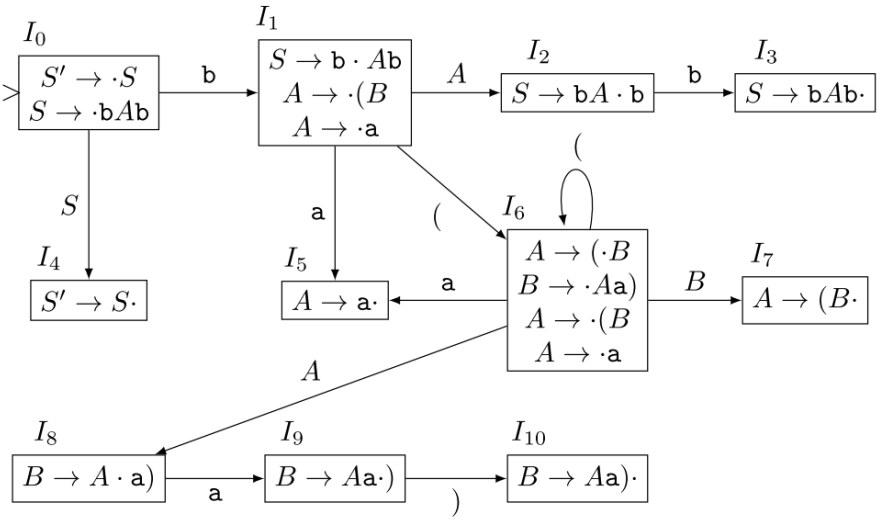
	$	d	c	b	a
S				$S \to AC$	$S \to AC$
A				$A \to \mathrm{b}$	$A \to \mathrm{a}A$
C			$C \to \mathrm{c}C'$		
C'	$C' \to \varepsilon$	$C' \to DC'$			
D		$D \to \mathrm{d}$			

$FIRST(S) \to \{\mathrm{a},\mathrm{b}\}$ $FOLLOW(S) \to \{\$\}$
$FIRST(A) \to \{\mathrm{a},\mathrm{b}\}$ $FOLLOW(A) \to \{\mathrm{c}\}$
$FIRST(C) \to \{\mathrm{c}\}$; $FOLLOW(C) \to \{\$\}$
$FIRST(C') \to \{\mathrm{d},\varepsilon\}$ $FOLLOW(C') \to \{\$\}$
$FIRST(D) \to \mathrm{d}$ $FOLLOW(D) \to \{\mathrm{d},\$\}$

Input	Step #	Stack
abcdd$	1	S
abcdd$	2	AC
abcdd$	3	$\mathrm{a}AC$
bcdd$	4	AC
bcdd$	5	$\mathrm{b}C$
cdd$	6	C
cdd$	7	$\mathrm{c}C'$
dd$	8	C'
dd$	9	DC'
dd$	10	$\mathrm{d}C'$
d$	11	C'
d$	12	DC'
d$	13	$\mathrm{d}C'$
$	14	C'

$	15	Done

(e) We found this Characteristic Finite State Machine in the classroom.



All the states are accepting states.

(i) What is the CFG for which this CFSM was constructed?

Solution:

$S \to \mathrm{b}A\mathrm{b}$

$A \to (B \mid \mathrm{a}$

$B \to A\mathrm{a})$

(ii) The sequence "b a b" is a valid word in the language generated by this grammar. Why, therefore, is there no accepting run of the CFSM on this word?

Solution:

There is no accepting run of the CFSM because reduce actions are not represented by transitions in the CFSM, so when we run the CFSM on "b a b" we reach state I_5 after reading the "a" and get "stuck".

The LR(0) parser, on the other hand will perform reduce actions and "rewind" the CFSM to a previous state, from whence the execution can continue.

Here's the sequence of shift/reduce actions (without showing the stack pushes):

$$I_0 \overset{b}{\to} I_1 \overset{a}{\to} I_5 \to \text{reduce to } I_1 \overset{A}{\to} I_2 \overset{b}{\to} I_3 \to \text{reduce to } I_0 \overset{S}{\to} I_4 \to \text{reduce to } I_0$$

(iii) Write down the sequence of shift/reduce actions an LR(0) parser will make on the input:

b ((a a) a) b

and the contents of the parser stack after each action takes place.

Solution:

In the following table, the stack grows from left to right.

Stack	Input	Action
I_0	b ((a a) a) b $	Shift, move to I_1
I_0 b I_1	((a a) a) b $	Shift, move to I_6
I_0 b I_1 (I_6	(a a) a) b $	Shift, move to I_6
I_0 b I_1 (I_6 (I_6	a a) a) b $	Shift, move to I_5
I_0 b I_1 (I_6 (I_6 a I_5	a) a) b $	Reduce: $A \rightarrow$ a
I_0 b I_1 (I_6 (I_6 A I_8	a) a) b $	Shift, move to I_9
I_0 b I_1 (I_6 (I_6 A I_8 a I_9	) a) b $	Shift, move to I_{10}
I_0 b I_1 (I_6 (I_6 A I_8 a I_9) I_{10}	a) b $	Reduce: $B \rightarrow A$a)
I_0 b I_1 (I_6 (I_6 B I_7	a) b $	Reduce: $A \rightarrow (B$
I_0 b I_1 (I_6 A I_8	a) b $	Shift, move to I_9
I_0 b I_1 (I_6 A I_8 a I_9	) b $	Shift, move to I_{10}
I_0 b I_1 (I_6 A I_8 a I_9) I_{10}	b $	Reduce: $B \rightarrow A$a)
I_0 b I_1 (I_6 B I_7	b $	Reduce: $A \rightarrow (B$
I_0 b I_1 A I_2	b $	Shift, move to I_3
I_0 b I_1 A I_2 b I_3	$	Reduce: $S \rightarrow$ bAb
I_0 S I_4	$	Reduce: $S' \rightarrow S$
I_0	$	Accept