**Theoretical Part**

1. The grammar for the NFA:
2. For any NFA, we set the grammar according to the states:

Given an NFA with the 5-tuple: { we add the following rules:

1. If state q belongs to F (Finite states):
2. If state
3. With these grammar rules, associativity of arithmetic operations doesn't follow the convention. Adding or multiplying don't pose a problem, subtracting and diving do. For example:

12/6/2. The Parsing tree:

The result will be 12/(6/2)=12/3=4

But expected result is (12/6)/2=2/2=1

T

T

F

/

T

/

F

F

12

6

2

1. Since C is left recursive, meaning it has a non-terminal left of a production (CD while C is non-terminal), the grammar isn't LL(1) – We may loop infinitely with this grammar.

To solve it, we take the following steps:

We make new, non-left-recursive rules with the following steps:

Now the new set of rules is:

Next step – Make the table of Terminals Vs States:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d |  |
| S | S->AC | S->AC |  |  |  |
| A | A->aA | A->b |  |  |  |
| C |  |  | C->cC' |  |  |
| C' |  |  |  | C'->DC' | C'-> |
| D |  |  |  | D->d |  |

The First of all states:

First(S) = First(A)

First(A) = {a, b}

First(C) = {c}

First(C') = First(D) {ε}

First(D) = {d}

The Follow of all states:

Follow(S) = {$}

Follow(A) = {c}

Follow(C) = {$}

Follow(C') = {$}}

Follow(D) = {d, $}

For the input abcdd:

|  |  |  |
| --- | --- | --- |
| Input | Step# | Stack (top is on left) |
| abcdd$ | 1 | S |
| abcdd$ | 2 | AC |
| abcdd$ | 3 | aAC |
| bcdd$ | 4 | AC |
| bcdd$ | 5 | bC |
| cdd$ | 6 | C |
| cdd$ | 7 | cC' |
| dd$ | 8 | C' |
| dd$ | 9 | DC' |
| dd$ | 10 | dC' |
| d$ | 11 | C' |
| d$ | 12 | DC' |
| d$ | 13 | dC' |
| $ | 14 | C' |
| $ | 15 |  |

* 1. The CFG for which this CFSM was constructed:
  2. The transitions in the CFSM don't implement the reduce actions, so when reading the input, b gets us to state I1 and a to I5. This state doesn't make a transition with b, the next letter in the input and so the CFSM doesn't accept.
  3. The LR(0) will work on the input as follows:

|  |  |  |
| --- | --- | --- |
| Stack | Input | Action |
| I\_0 | b ( ( a a ) a ) b$ | Shift, ->I1 |
| I\_0bI1 | (( a a ) a ) b$ | Shift, ->I6 |
| I\_0bI1(I6 | ( a a ) a ) b$ | Shift, ->I6 |
| I\_0bI1(I6(I6 | a a ) a ) b$ | Shift, ->I5 |
| I\_0bI1(I6(I6aI5 | a ) a ) b$ | Reduce A->a |
| I\_0bI1(I6(I6AI8 | a ) a ) b$ | Shift, ->I9 |
| I\_0bI1(I6(I6AI8aI9 | ) a ) b $ | Shift, I10 |
| I\_0bI1(I6(I6AI8aI9I)10 | a ) b$ | Reduce B->Aa) |
| I\_0bI1(I6(I6BI7 | a ) b$ | Reduce A->(B |
| I\_0bI1(I6AI8 | a ) b$ | Shift, ->I9 |
| I\_0bI1(I6AI8aI9 | ) b$ | Shift, ->I10 |
| I\_0bI1(I6AI8aI9)I10 | b$ | Reduce B-> Aa) |
| I\_0bI1(I6BI7 | b$ | Reduce A->(B |
| I\_0bI1AI2 | b$ | Shift, ->I3 |
| I\_0bI1AI2b | $ | Reduce S->bAb |
| I\_0SI4 | $ | Reduce S'->S |
| I\_0 | $ | done |