Matilda Knapp
CSC 344

Problem set #1: BNF Assignment

Learning Abstract

This assignment focuses on illustrating a few variations of Backus Naur Form. Overall five languages are illustrated, RB4B, SQN, IR123, BXR, and CF. This assignment also ends with a short explanation of Backus Naur Form that may be helpful for a beginner in computer science. Each language is paired with at least one example parse tree, which I believe are especially helpful to look at for thoses fairly new to this concept. In the accompanying specification, there is some more information on how these languages were derived. I hope everything here is easy to understand, I tried my best to make straightforward names.

Problem 1: Language-RB4B

Step 1: Forming BNF Grammar RB4B

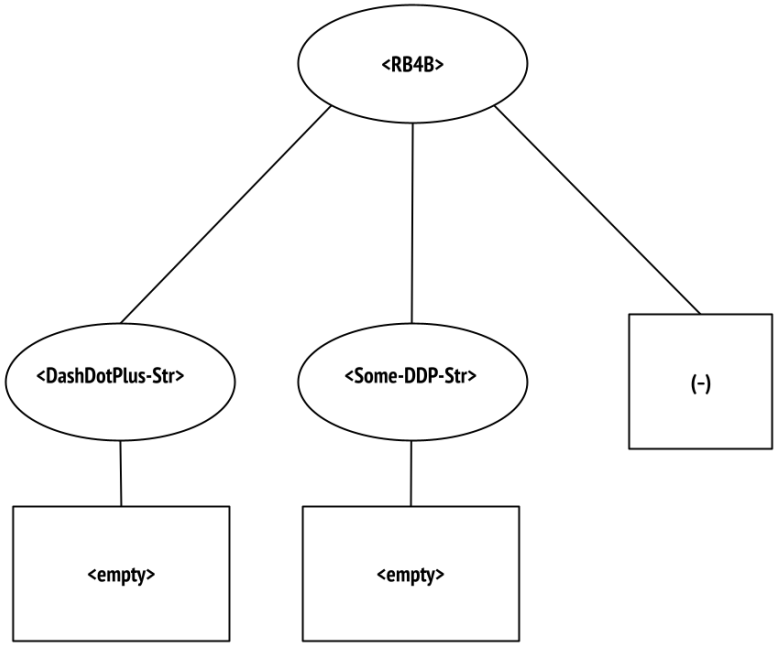
<RB4B> :: = <DashDotPlus-Str> <Some-DDP-Str> (-)

<RB4B> :: = <DashDotPlus-Str> <Some-DDP-Str> (.+)

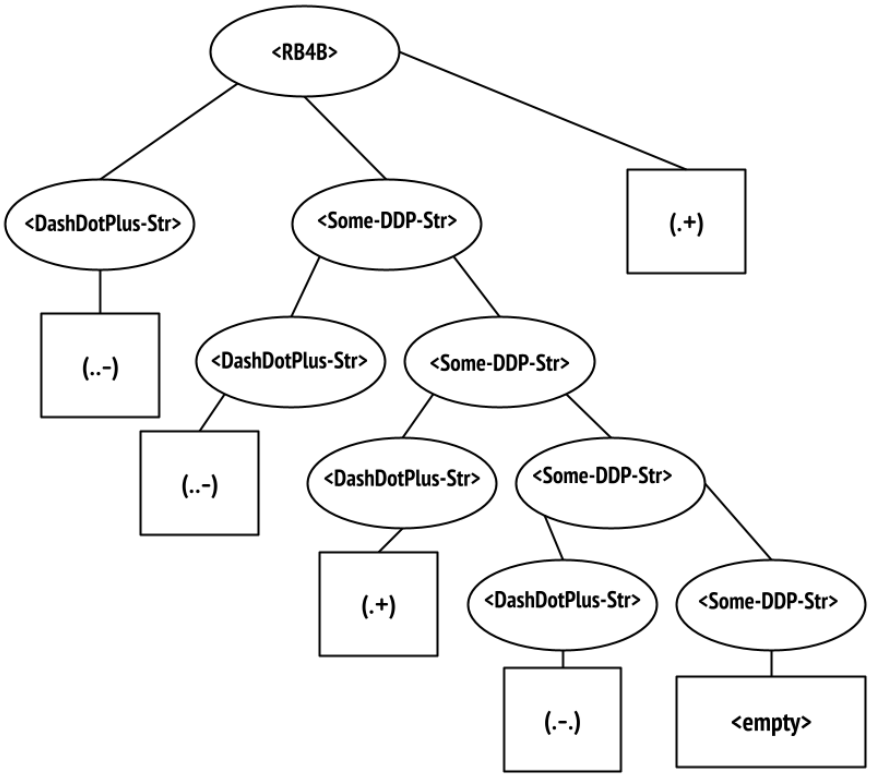
<DashDotPlus-Str> :: = (-) | (--) | (-..) | (.-.) | (..-) | (.+) | <empty>

Some-DDP-Str :: = <empty> | <DashDotPlus-Str> Some-DDP-Str

Step 2: parse tree for (–)



Step 2: parse tree for: (..–) (..–)(.+)(.–.)(.+)



Problem 2: Language-SQN (Special Quaternary Numbers)

Step 1: Forming BNF Grammar SQN

<SQN> :: = 0 | 1 <NonOneStart>

<SQN> :: = 2 <NonTwoStart>

<SQN> :: = 3 <NonThreeStart>

<NonOneStart> :: = <NO-OneEnd> <NonOneStart> | 0 | 2 | 3 | <empty>

<NonOneStart> :: = <NO-TwoEnd> <NonTwoStart> | 0 | 2 | 3 | <empty>

<NonOneStart> :: = <NO-ThEnd> <NonThreeStart> | 0 | 2 | 3 | <empty>

<NonTwoStart> :: = <NTw-OneEnd> <NonOneStart> | 0 | 1 | 3 | <empty>

<NonTwoStart> :: = <NTw-TwoEnd> <NonTwoStart> | 0 | 1 | 3 | <empty>

<NonTwoStart> :: = <NTw-ThEnd> <NonThreeStart> | 0 | 1 | 3 | <empty>

<NonThreeStart> ::= <NTh-OneEnd> <NonOneStart> | 0 | 1 | 2 | <empty>

<NonThreeStart> ::= <NTh-TwoEnd> <NonTwoStart> | 0 | 1 | 2 | <empty>

<NonThreeStart> ::= <NTh-ThEnd> <NonThreeStart> | 0 | 1 | 2 | <empty>

<NO-OneEnd> :: = 01 | 21 | 31

<NO-TwoEnd> :: = 02 | 32

<NO-ThEnd> :: = 03 | 23

<NTw-OneEnd> :: = 01 | 31

<NTw-TwoEnd> :: = 02 | 12 | 32

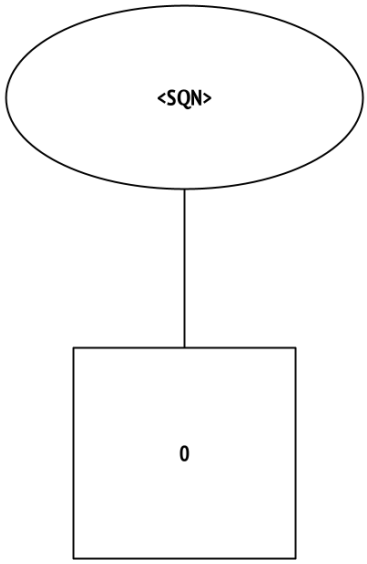
<NTw-ThEnd> :: = 03 | 13

<NTh-OneEnd> :: = 01 | 21

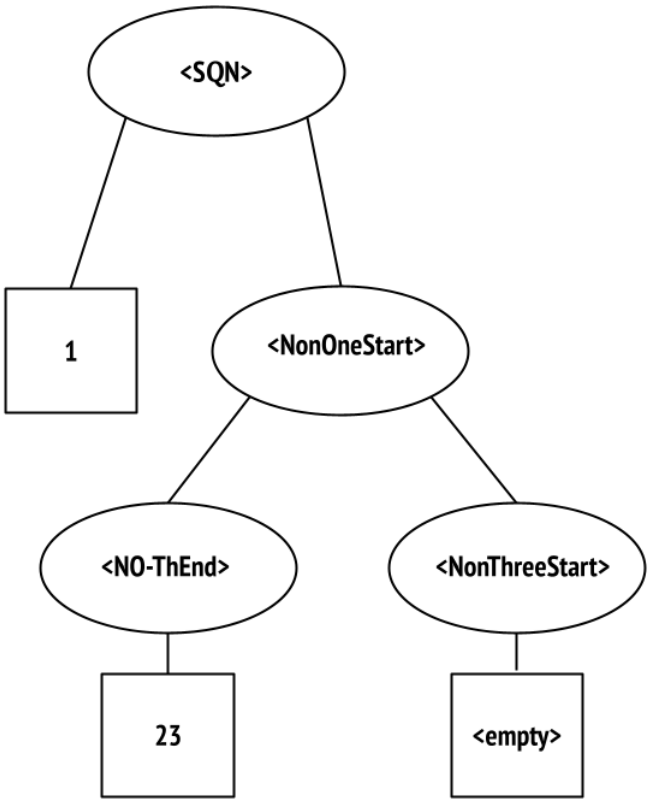
<NTh-TwoEnd> :: = 02 | 12

<NTh-ThEnd> :: = 03 | 13 | 23

Step 2: parse tree for: 0



Step 3: parse tree for: 123



Step 4: Why can't you draw a parse tree for 1223 in this language?

I can't draw a parse tree for 1223 in this language because I will be stopped when I try to add the double twos. I will be able to add the 1 as the first number, but then I'm given the option to end with a single 2, or one of the three options NO-OneEnd, NO-TwoEnd or NO-ThreeEnd. I might try to pick NO-TwoEnd, since these are the strings of two numbers ending in two, but this option doesn't contain a string for 22, since that would be two adjacent numbers, and this language is made to avoid such a situation.

Problem 3: Language-IR123

Step 1: Forming BNF Grammar IRI123 (added lines to help spaceout)

<IR123> :: = [C] <Non-C-Start> | [D C] <Non-DC-Start>

<IR123> :: = [B C] <Non-BC-Start> | [E D C] <Non-EDC-Start>

<IR123> :: = [F E C] <Non-FEC-Start> | [G F C] <Non-GFC-Start>

<Non-C-Start> :: = [D C] <Non-DC-Start> | [B C] <Non-BC-Start>

<Non-C-Start> :: = [E D C] <Non-EDC-Start> | [F E C] <Non-FEC-Start>

<Non-C-Start> :: = [G F C] <Non-GFC-Start> | <empty>

<Non-DC-Start> :: = [C] <Non-C-Start> | [B C] <Non-BC-Start>

<Non-DC-Start> :: = [E D C] <Non-EDC-Start> | [F E C] <Non-FEC-Start>

<Non-DC-Start> :: = [G F C] <Non-GFC-Start> | <empty>

<Non-BC-Start> :: = [C] <Non-C-Start> | [D C] <Non-DC-Start>

<Non-BC-Start> :: = [E D C] <Non-EDC-Start> | [F E C] <Non-FEC-Start>

<Non-BC-Start> :: = [G F C] <Non-GFC-Start> | <empty>

<Non-EDC-Start> :: = [C] <Non-C-Start> | [D C] <Non-DC-Start>

<Non-EDC-Start> :: = [B C] <Non-BC-Start> | [F E C] <Non-FEC-Start>

<Non-EDC-Start> :: = [G F C] <Non-GFC-Start> | <empty>

<Non-FEC-Start> :: = [C] <Non-C-Start> | [D C] <Non-DC-Start>

<Non-FEC-Start> :: = [B C] <Non-BC-Start> | [E D C] <Non-EDC-Start>

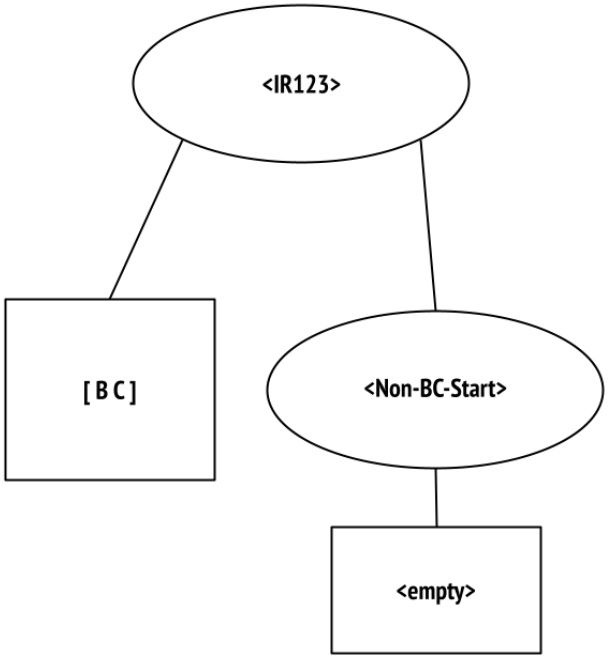
<Non-FEC-Start> :: = [G F C] <Non-GFC-Start> | <empty>

<Non-GFC-Start> :: = [C] <Non-C-Start> | [D C] <Non-DC-Start>

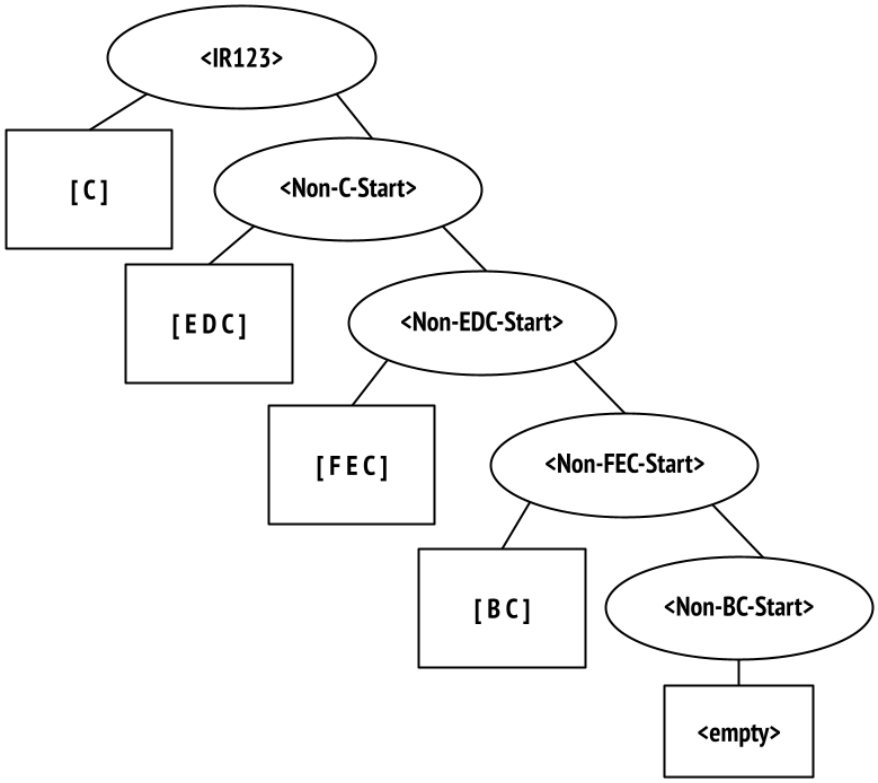
<Non-GFC-Start> :: = [B C] <Non-BC-Start> | [E D C] <Non-EDC-Start>

<Non-GFC-Start> :: = [F E C] <Non-FEC-Start> | <empty>

Step 2: parse tree for: [B C] “Next Page”



Step 3: parse tree for: [C] [E D C] [F E C] [B C]



Step 4: Why can't you draw a parse tree for [D C] [B C] [B C] [C] in this language?

You can't draw a parse tree for [D C] [B C] [B C] [C] in this language because, when you choose [B C] you have the <Non-BC-Start> parameter next. The <Non-BC-Start> parameter contains all strings but [B C], as the name implies. Therefore you cannot have repeating adjacent strings in this language.

Problem 4 Language-BXR

Step 1: Forming BNF Grammar BXR

<BXR> :: = (<Str-BXR> <More-Str-BXR>)

<Str-BXR> :: = and <statement> | or <statement>

<statement> :: = <TF-Mix> | (<Not-One>) | (and <TF-Mix>)

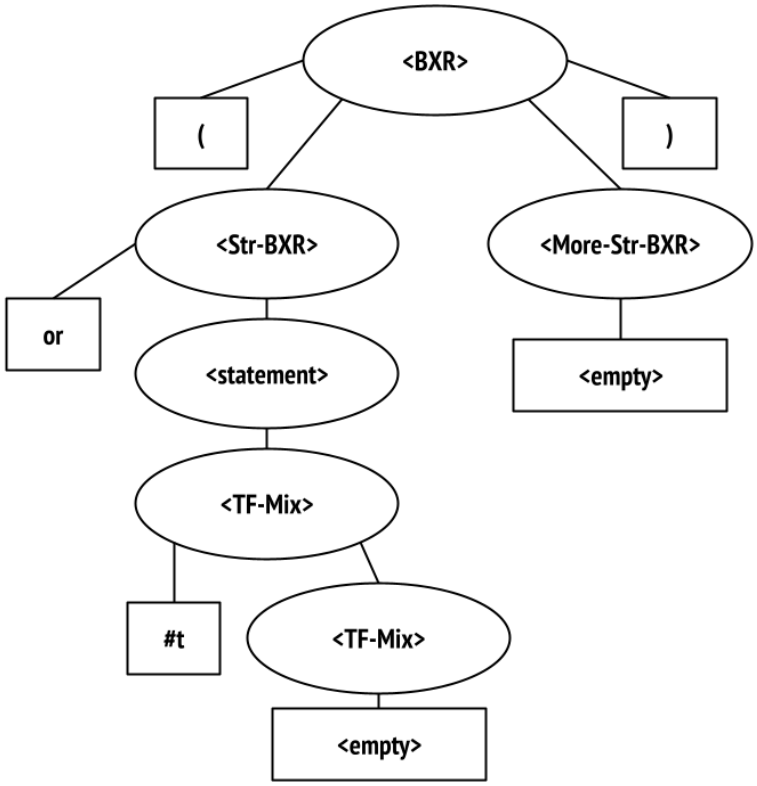
<statement> :: = (or <TF-Mix>) | <empty>

<More-Str-BXr> :: = <statement> <More-Str-BXr> | <empty>

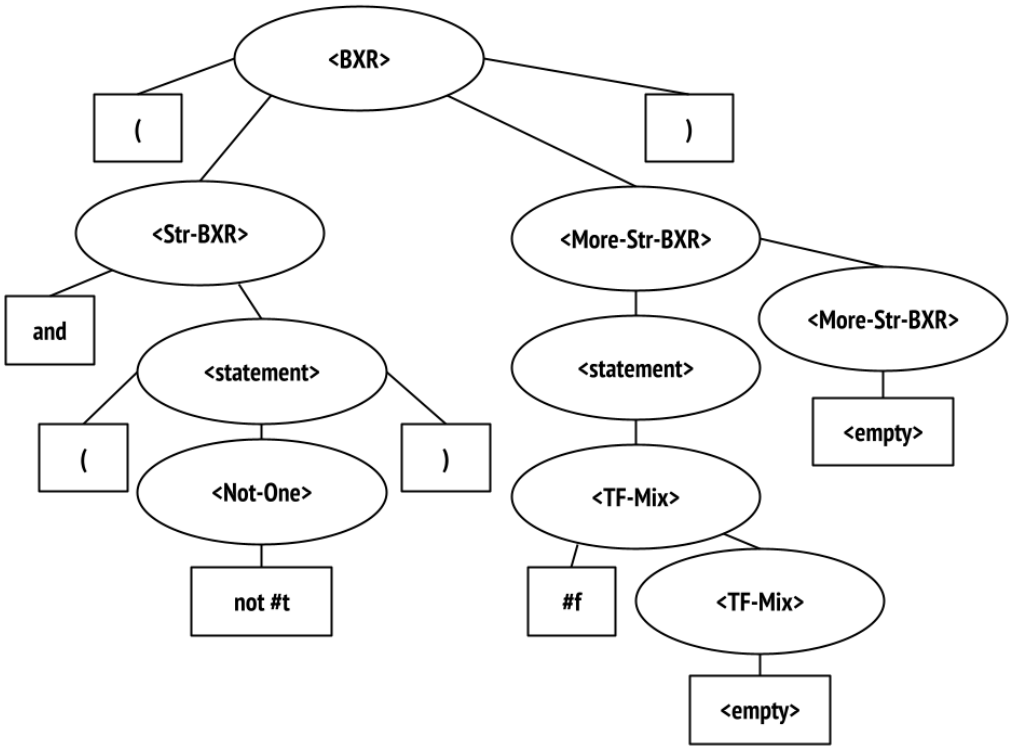
<TF-Mix> :: = #t <TF-Mix> | #f <TF-Mix> | <empty>

<Not-One> :: = not #t | not #f

Step 2: parse tree for: (or #t) "Next Page"



Step 3: parse tree for: (and (not #t) #f)



Problem 5: Language-CF (Color Fun)

Step 1: Forming BNF Grammar CF

<CF> :: = ? <function>

<function> :: = <addfunc> | colors | <showfunc> | <describefunc> | exit

<addfunc> :: = add color <name> | add (<three-four-num>) <name>

<three-four-num> :: = <n> <n> <n> | <n> <n> <n> <n>

<showfunc :: = show <name>

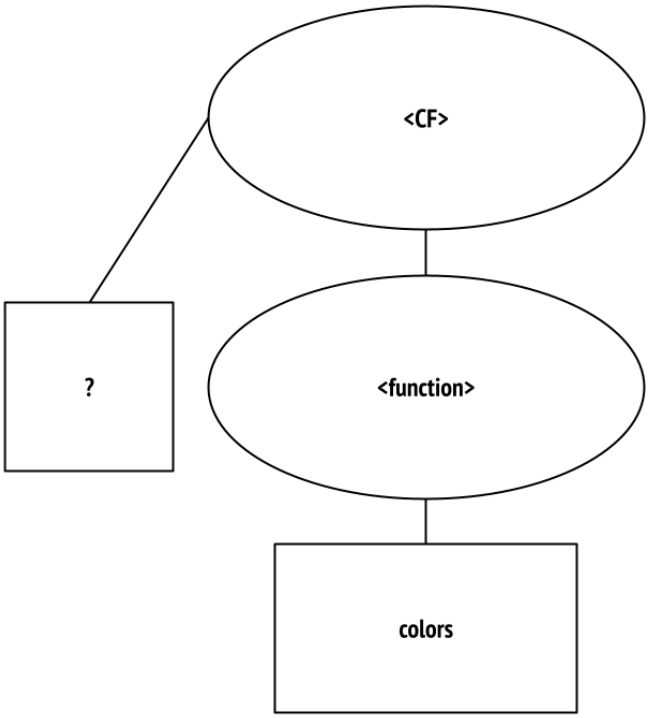
<describefunc> :: = describe <name>

<name> :: = red | light-red | c3 | c2 | c1 | purple | <etc>

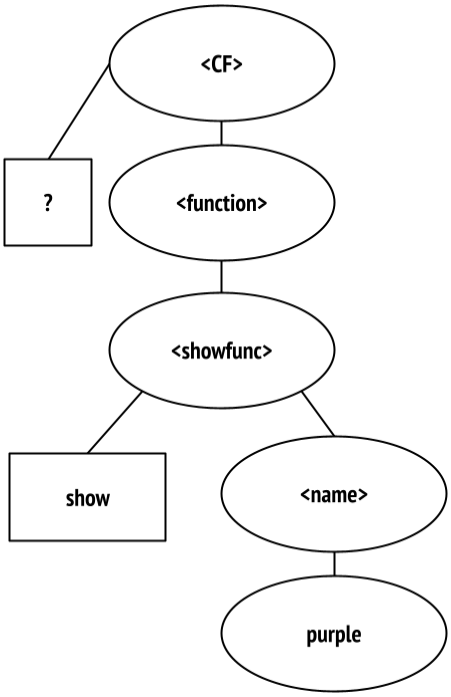
<n> :: = 1 | 2 | 3 | 4 | <etc>... | 250 | 251 | 252 | 253 | 254 | 255

<etc :: = Not all values can be listed

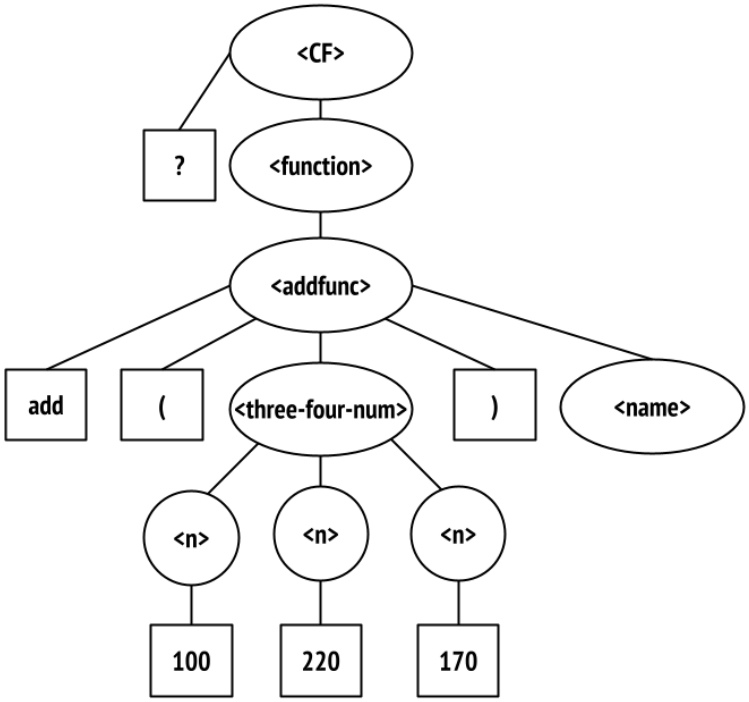
Step 2: Parse tree for sentence: colors



Step 3: Parse tree for sentence: show purple



Step 4: Parse tree for sentence: add (100 220 170) favorite-color



Problem 6: What is BNF? (No more than 100 words)

BNF stands for Backus Naur Form. We use BNF to indicate symbols in a language, and in what order they are to be presented. This is helpful because in code we constantly use these languages to do things. From the perspective of a beginner syntax may seem arbitrary, but BNF can be used to show it's complexity. The creators of BNF are Peter Naur, and John Backus, hence the name.