
Problem Set #1: BNF

ABSTRACT:

This first problem set is structured to provide a comprehensive understanding of BNF grammar and its role in programming languages. The main objective is to familiarize oneself with structuring BNFs and using them to create parse trees. Below are 5 problems that utilize the creation of BNF grammar and parse trees to construct sentences. The last problem is a brief and clear explanation of BNF, aimed at educating newer computer programmers.

Problem 1 – Laughter

Start Symbol: L

Tokens: {HA, HEE}

Nonterminals: {L, HA, HEE, extra-HEE}

Production Rules:

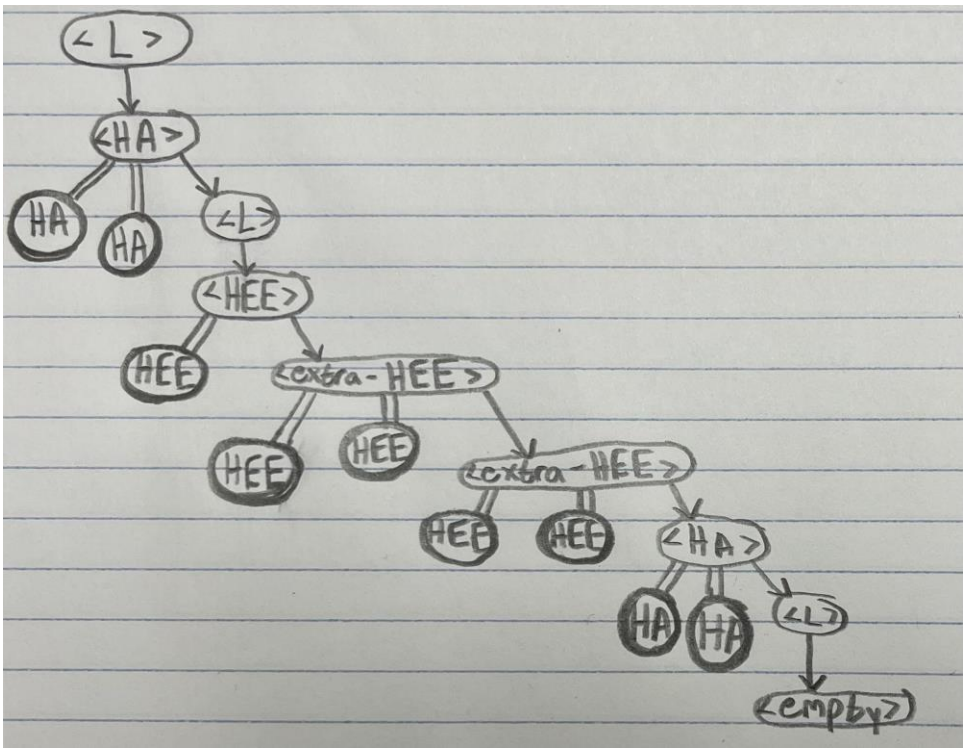
<L> ::= <HA> | <HEE> | <empty>

<HA> ::= HA HA <L> | HA HA <HA>

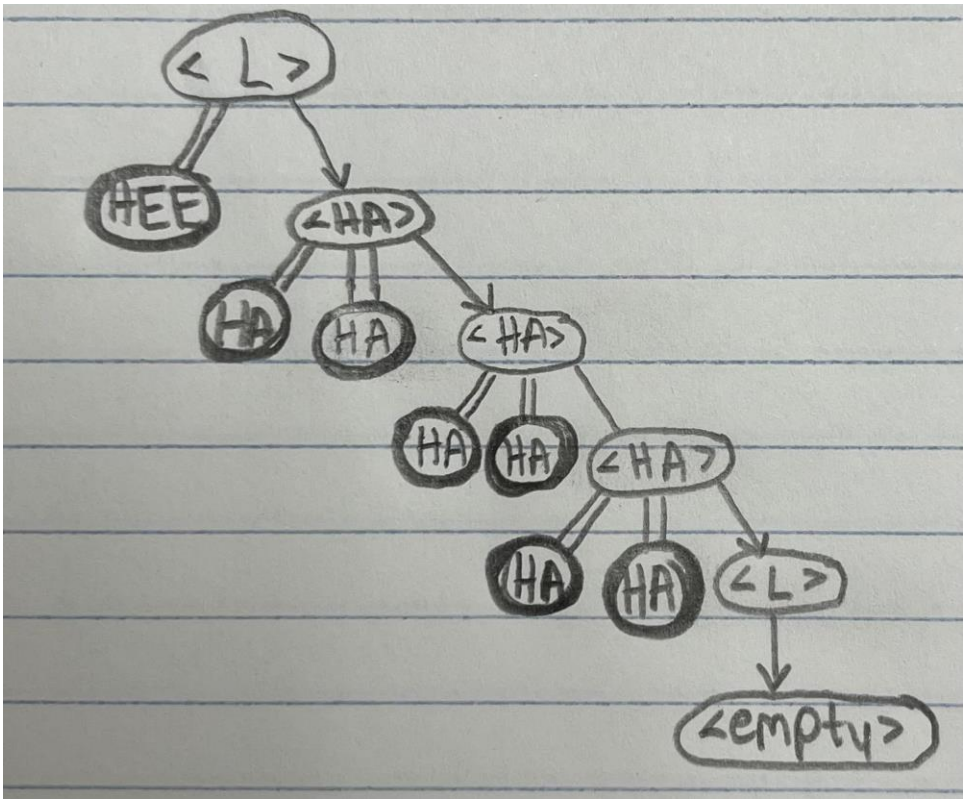
<HEE> ::= HEE | HEE <HA> | HEE <extra-HEE>

<extra-HEE> ::= HEE HEE <extra-HEE> | HEE HEE <HA> | HEE HEE

Parse tree 1: HA HA HEE HEE HEE HEE HEE HA HA



Parse tree 2: HEE HA HA HA HA HA HA



Problem 2 – SQN (Special Quaternary Numbers)

Start Symbol: SQN

Tokens: {0, 1, 2, 3}

Nonterminals: {SQN, ZERO, ONE, TWO, THREE}

Production Rules:

<SQN> ::= 0 | <ONE> | <TWO> | <THREE>

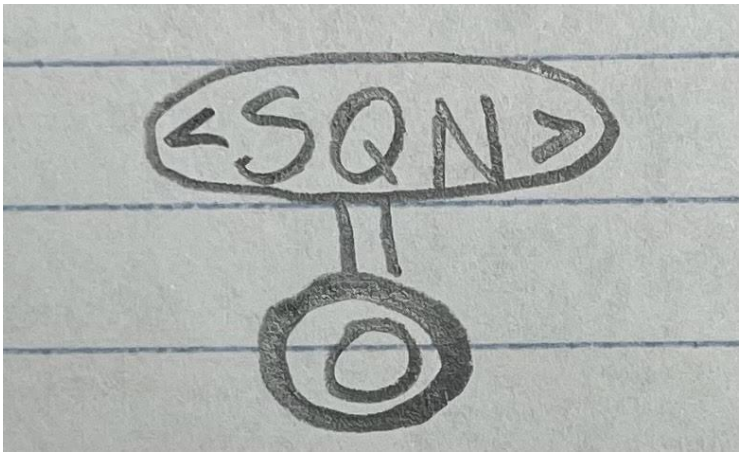
<ZERO> ::= 0 <ONE> | 0 <TWO> | 0 <THREE> | 0

<ONE> ::= 1 <ZERO> | 1 <TWO> | 1 <THREE> | 1

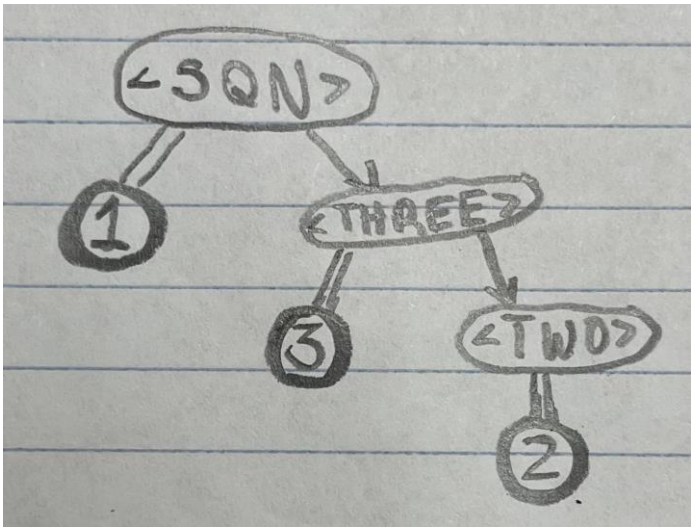
<TWO> ::= 2 <ZERO> | 2 <ONE> | 2 <THREE> | 2

<THREE> ::= 3 <ZERO> | 3 <ONE> | 3 <TWO> | 3

Parse tree 1: 0



Parse tree 2: 132



Explain, in precise terms, why you cannot draw a parse tree, consistent with the BNF grammar that you crafted, for the string: 1223:

Constructing a parse tree, consistent with the BNF grammar for the input string “1223” is not possible due to the grammar’s rule. The grammar prohibits two or more adjacent instances of the same quaternary number. The input string has two consecutive 2’s adjacent to one another. In order for the input string to conform to the BNF grammar, the adjacent 2s must have a 0, 1, or 3 in between them.

Problem 3 – BXR

Start Symbol: S

Tokens: {(, #t, #f, and, or, not,)}

Nonterminals: {S, TF, OP, BXN, NOT}

Production Rules:

<S> ::= (<OP> <BXN>) | <TF> | (<NOT>)

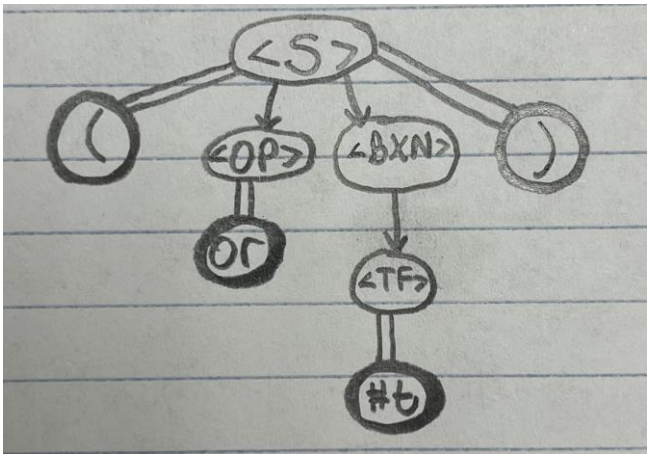
<TF> ::= #t | #f | #t <TF> | #f <TF> | <empty>

<OP> ::= and | or

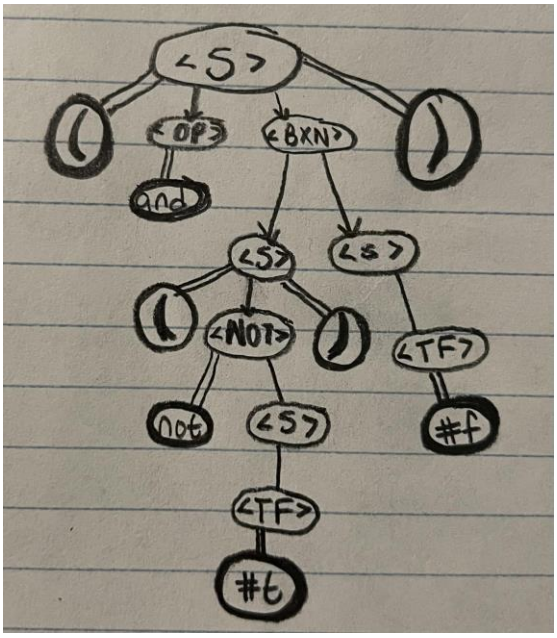
<BXN> ::= <empty> | <S> <S> | <S> |<BXN> <BXN> | <TF>

<NOT> ::= not <S>

Parse tree 1: (or #t)



Parse tree 2: (and (not #t) #f)



Problem 4 – LSS (Line Segment Sequences)

Start Symbol: TRIPLE-LSS

Tokens: {(, distance, angle, color,)}

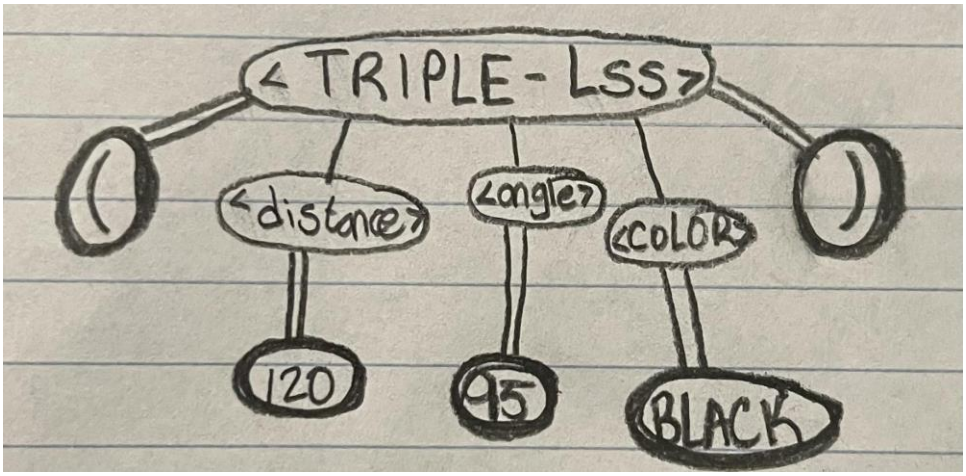
Nonterminals: {TRIPLE-LSS, COLOR}

Production Rules:

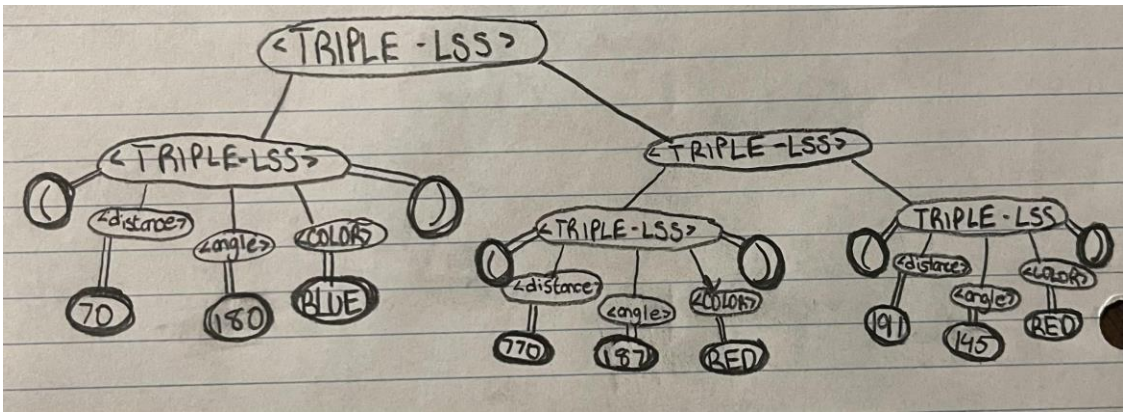
<TRIPLE-LSS> ::= <empty> | (<distance> <angle> <COLOR>) | <TRIPLE-LSS> <TRIPLE-LSS>

<COLOR> ::= RED | BLACK | BLUE

Parse tree 1: (120 95 BLACK)



Parse tree 2: (70 180 BLUE) (770 187 RED) (191 145 RED)



Problem 5 - M-Lines

Start Symbol: M-EVENT

Tokens: {PLAY, REST, RP, LP, S2, S3, X2, X3,}

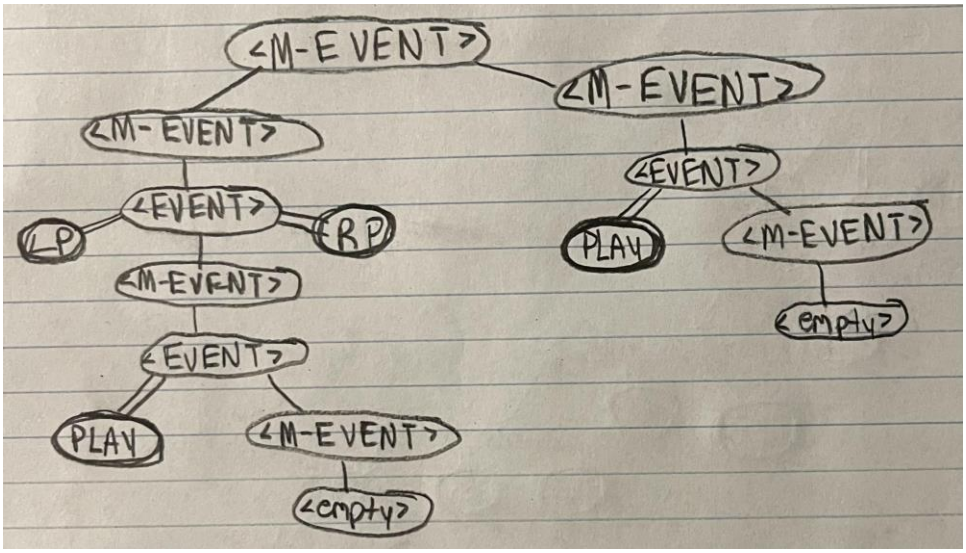
Nonterminals: {M-EVENT, EVENT}

Production Rules:

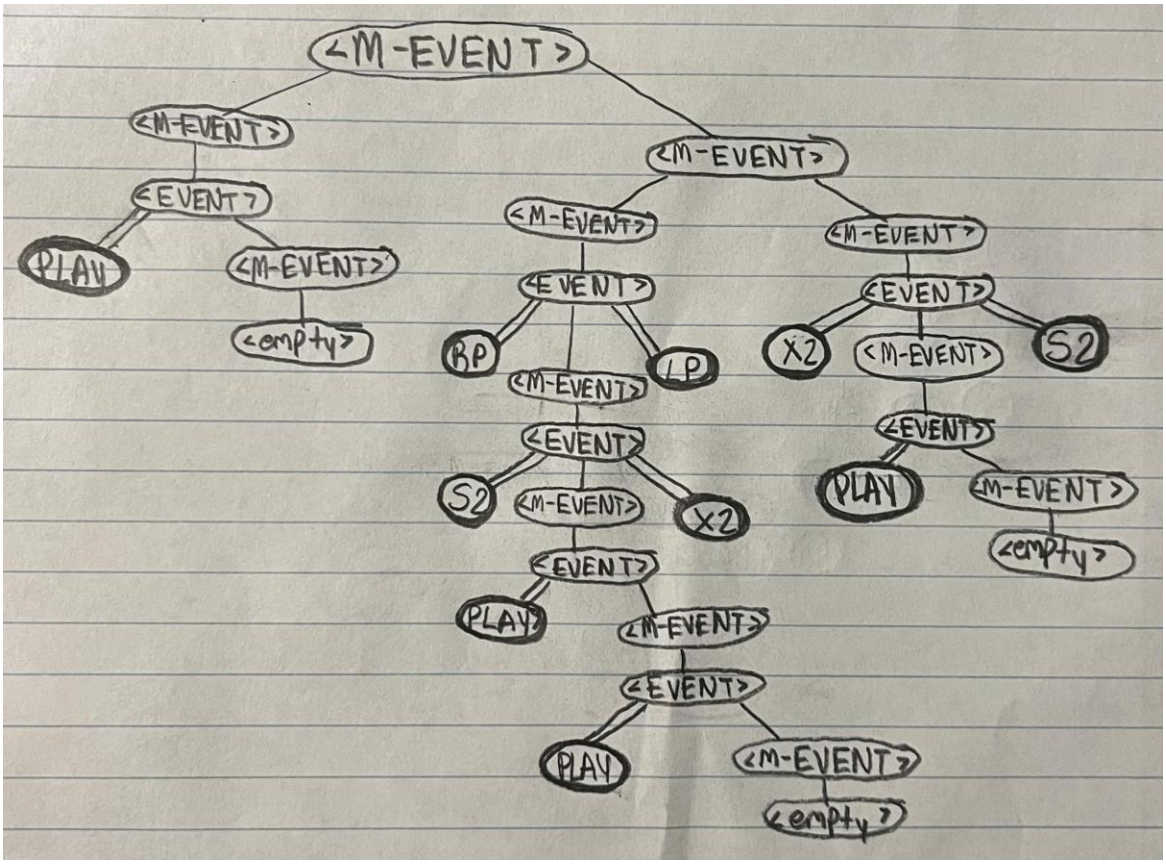
<M-EVENT> ::= <M-EVENT> <M-EVENT> | <EVENT> | <empty>

<EVENT> ::= PLAY <M-EVENT> | REST <M-EVENT> | RP <M-EVENT> LP | LP <M-EVENT> RP | S2 <M-EVENT> X2 | X2 <M-EVENT> S2 | X3 <M-EVENT> S3 | S3 <M-EVENT> X3 | <empty>

Parse tree 1: LP PLAY RP PLAY



Parse tree 2: PLAY RP S2 PLAY PLAY X2 LP X2 PLAY S2



Problem 6 - BNF?

What is BNF?

BNF is short for Backus-Naur Form and is a tool used for constructing grammar to define a language. BNF grammar has four entities which can specify a programming language's grammar, tokens, nonterminal symbols, productions, and a start symbol. Tokens represent the elements of the language. Nonterminal symbols describe the structure of the code. Productions are rules that describe the nonterminal symbols by replacing the symbols with a string of tokens and other nonterminal symbols. The start symbol represents the starting point for the interpretation of the code. Each of these components are used to describe a programming language’s syntax.