BNF Assignment

Written by David Hennigan

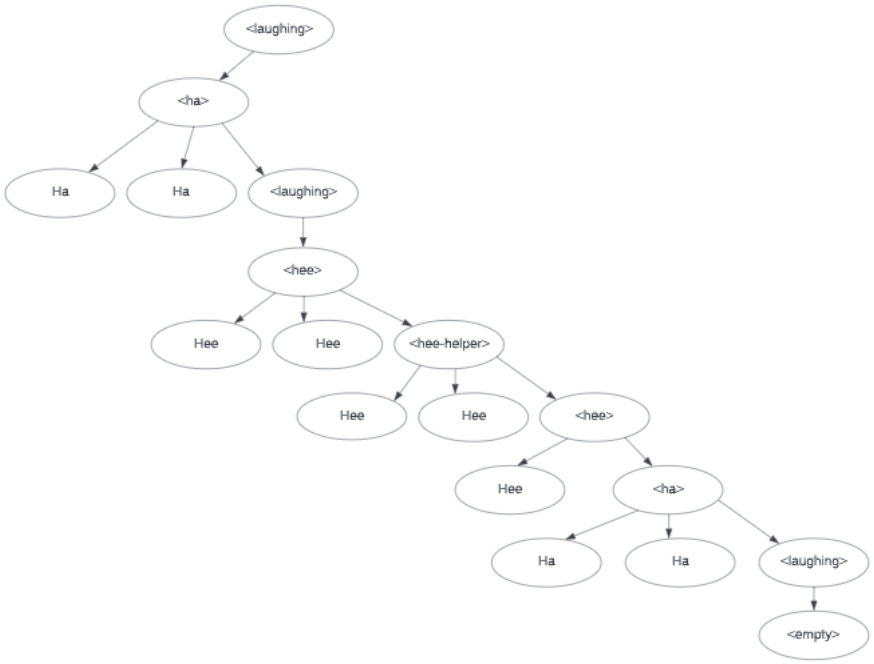
Abstract

This assignment is composed of six problems, five of which require the creation of BNF grammars; The remaining problem includes a succinct explanation of BNF designed to articulate the main ideas to relatively new programmers. The problems involving the creation of BNF grammars will include the BNF grammar, parse trees for specific sentences, and some may include explanations for why a specified output isn't possible with the BNF grammar. The overarching goal of this assignment is to become familiar with how BNF grammars are structured and how parse trees are generated from the associated grammar.

Problem 1: Laughter

BNF Grammar Description:
<laughing> ::= <ha> | <hee> | <empty>
<ha> ::= Ha Ha <laughing>
<hee> ::= Hee <ha> | Hee Hee <hee-helper> | Hee
<hee-helper> ::= Hee <ha> | Hee Hee <hee> | Hee

Parse Tree for: Ha Ha Hee Hee Hee Hee Hee Ha Ha

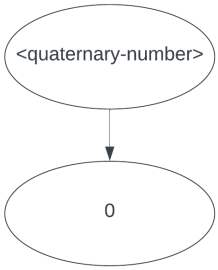


Parse Tree for: Hee Ha Ha Ha Ha Ha Ha

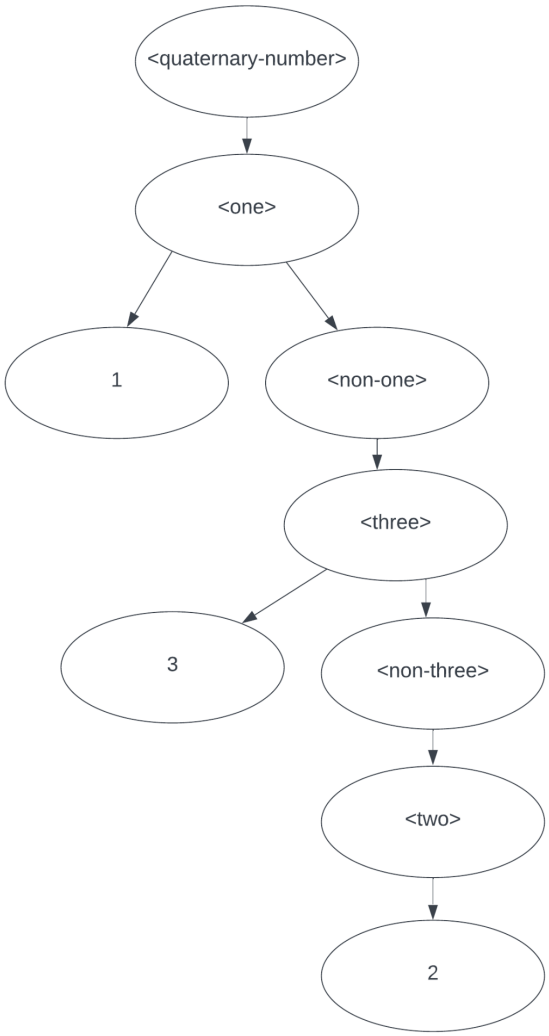
Problem 2: SQN (Special Quaternary Numbers)

BNF Grammar Description:
<quaternary-number> ::= 0 | <one> | <two> | <three>
<zero> ::= 0 | 0 <non-zero>
<non-zero> ::= <one> | <two> | <three>
<one> ::= 1 | 1 <non-one>
<non-one> ::= <zero> | <two> | <three>
<two> ::= 2 | 2 <non-two>
<non-two> ::= <zero> | <one> | <three>
<three> ::= 3 | 3 <non-three>
<non-three> ::= <zero> | <one> | <two>

Parse Tree for: 0



Parse Tree for: 132

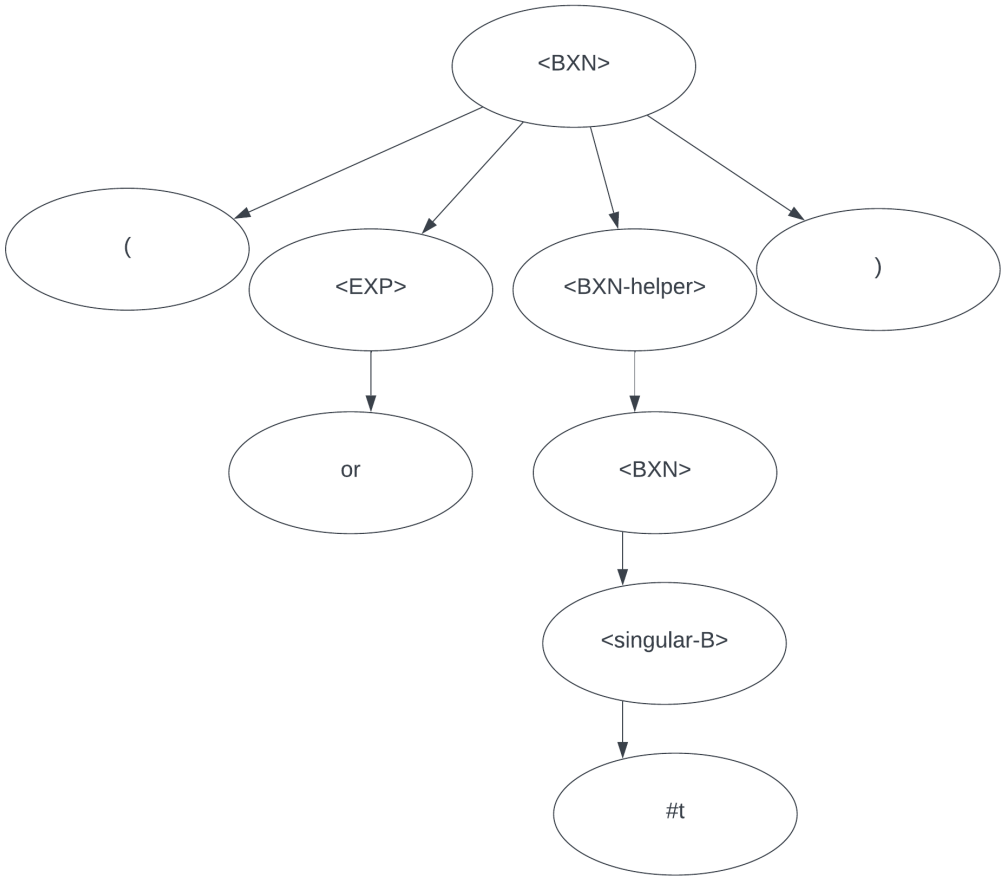


Explanation of why 1223 is not in the language:
The string 1223 is not possible with the BNF grammar I designed because whenever a 2 is called it can be followed by anything but a 2 specified as the non-terminal <non-two>. This non-terminal makes it impossible for 2s to repeat since the only non-terminal that can create a 2 is <two> and the only options are for a 2 that ends the string or a 2 that continues the string followed by a <non-two> which can only lead to 0, 1, or 3 for terminals.

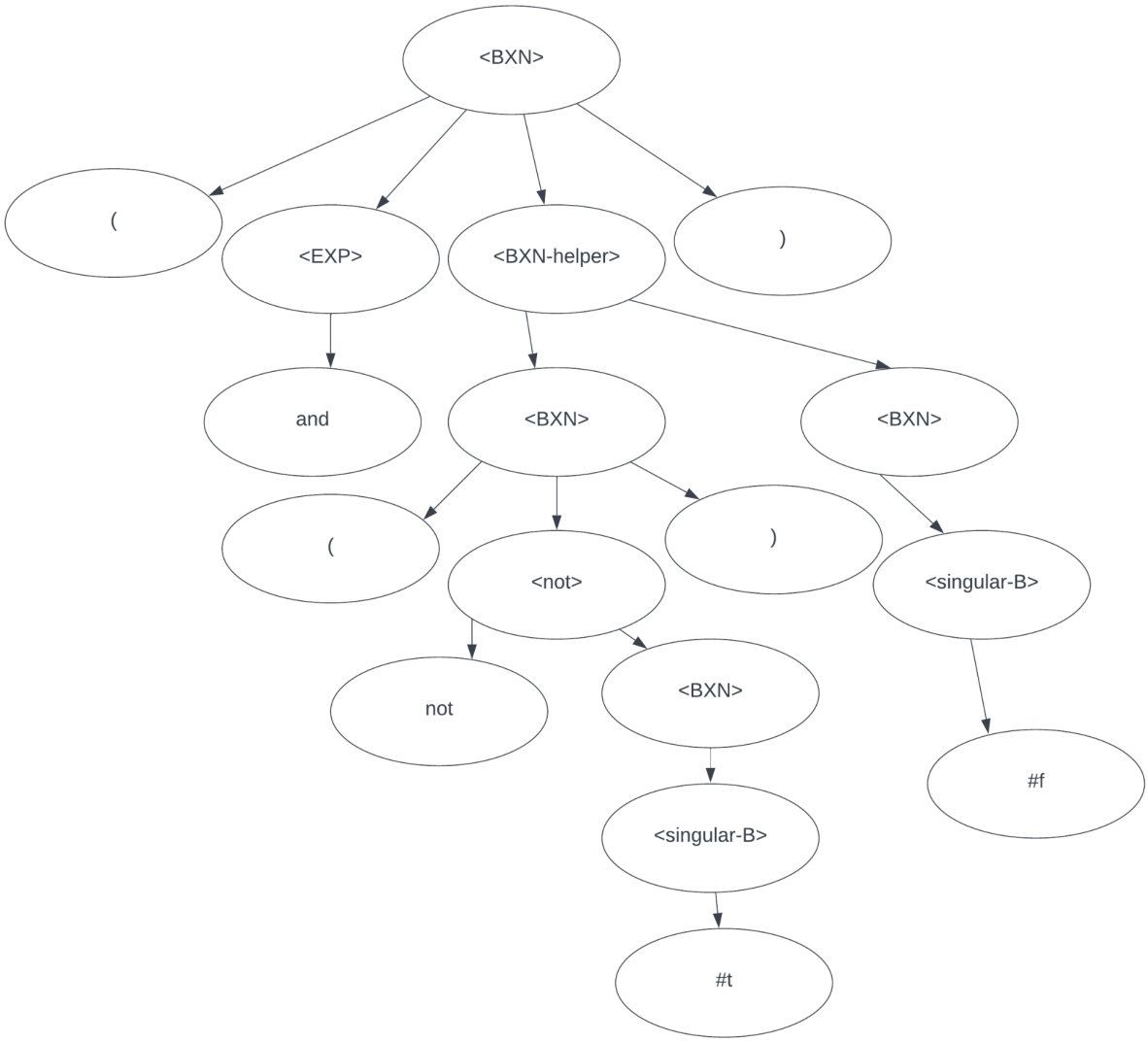
Problem 3: BXR (Boolean Expressions in Racket)

BNF Grammar Description:
<BXN> ::= (<EXP> <BXN-helper>) | (<not>) | <singular-B>
<EXP> ::= and | or
<BXN-helper> ::= <BXN> <BXN> | <B> | <BXN> | <BXN-helper> <BXN-helper> | <empty>
<not> ::= not <BXN>
<singular-B> ::= #f | #t
<B> ::= #f <B> | #t <B> | <empty>

Parse Tree for: (or #t)



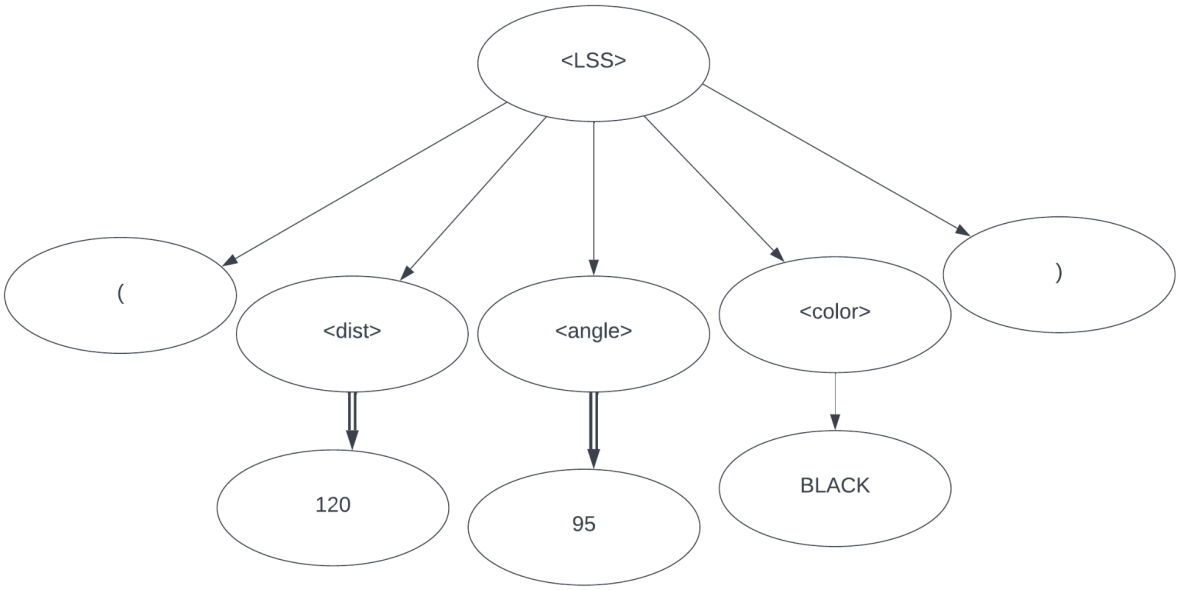
Parse Tree for: (and (not #t) #f)



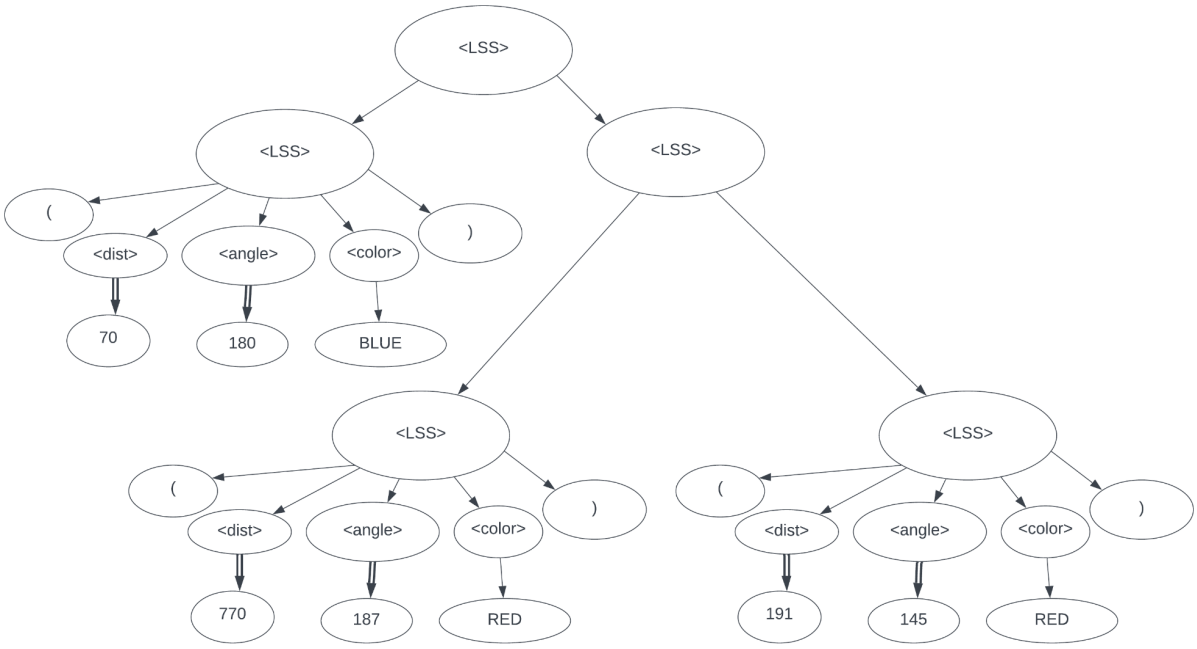
Problem 4: LSS (Line Segment Sequence)

BNF Grammar Description:
<LSS> ::= <empty> | <LSS> <LSS> | (<dist> <angle> <color>)
<color> ::= RED | BLACK | BLUE

Parse Tree for: (120 95 BLACK)



Parse Tree for: (70 180 BLUE) (770 187 RED) (191 145 RED)



Problem 5: M-Lines

BNF Grammar Definition:
<Event-Sequence> ::= <Event-Sequence> <Event-Sequence> | <Event> | <Sandwich> |
<empty>
<Event> ::= PLAY <Event-Sequence> | REST <Event-Sequence>
<Sandwich> ::= RP <Event-Sequence> LP | LP <Event-Sequence> RP |
S2 <Event-Sequence> X2 | X2 <Event-Sequence S2 |
S3 <Event-Sequence> X3 | X3 <Event-Sequence> S3

Parse Tree for: LP PLAY RP PLAY

Parse Tree for: PLAY RP S2 PLAY PLAY X2 LP X2 PLAY S2

Problem 6: BNF?

BNF is a tool for producing a grammar to define a language. Since many languages are infinitely large, the purpose of producing a grammar is to provide a way to specify characteristics of a language in some finite manner. The BNF grammar is composed of defining symbols as either tokens or non-terminals, delegation of a start symbol, and production rules which allow for the mapping of non-terminals to non-terminals or tokens. Using the BNF tools briefly described it is possible to generate grammars.