BNF Assignment

Learning Abstract

In this problem set I will focus on BNF (Backus Naur Form), a formalism for notating languages originally created for describing Algol. In this assignment I will write both grammars to describe languages as well as parse trees to represent specific examples of the languages. All parse trees for this assignment were drawn in OneNote.

Problem 1 - RB4B

RB4B is a language consisting of the set of all strings composed of a number of preset strings (listed below) in parentheses ending in either `( - )` or `( . + )`.

Preset string options

- `( - )`
- `( - - )`
- `( - . . )`
- `( . - . )`
- `( . . - )`
- `( . + )`

Some examples of sentences in RB4B are

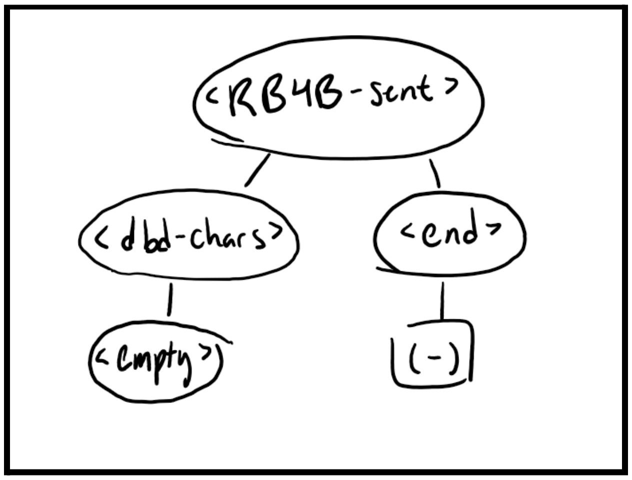
- `( - )`
- `( . + )( - - )( . + )`
- `( - )( - )( - )( . + )`

Task 1 - write a BNF grammar description of RB4B

```
1    <RB4B-sent> ::= <dbd-chars> <end>
2    <dbd-chars> ::= <empty>
3    <dbd-chars> ::= <RB4B-char> <dbd-chars>
4    <RB4B-char> ::= ( - )
5    <RB4B-char> ::= ( - - )
6    <RB4B-char> ::= ( - . . )
7    <RB4B-char> ::= ( . - . )
8    <RB4B-char> ::= ( . . - )
```

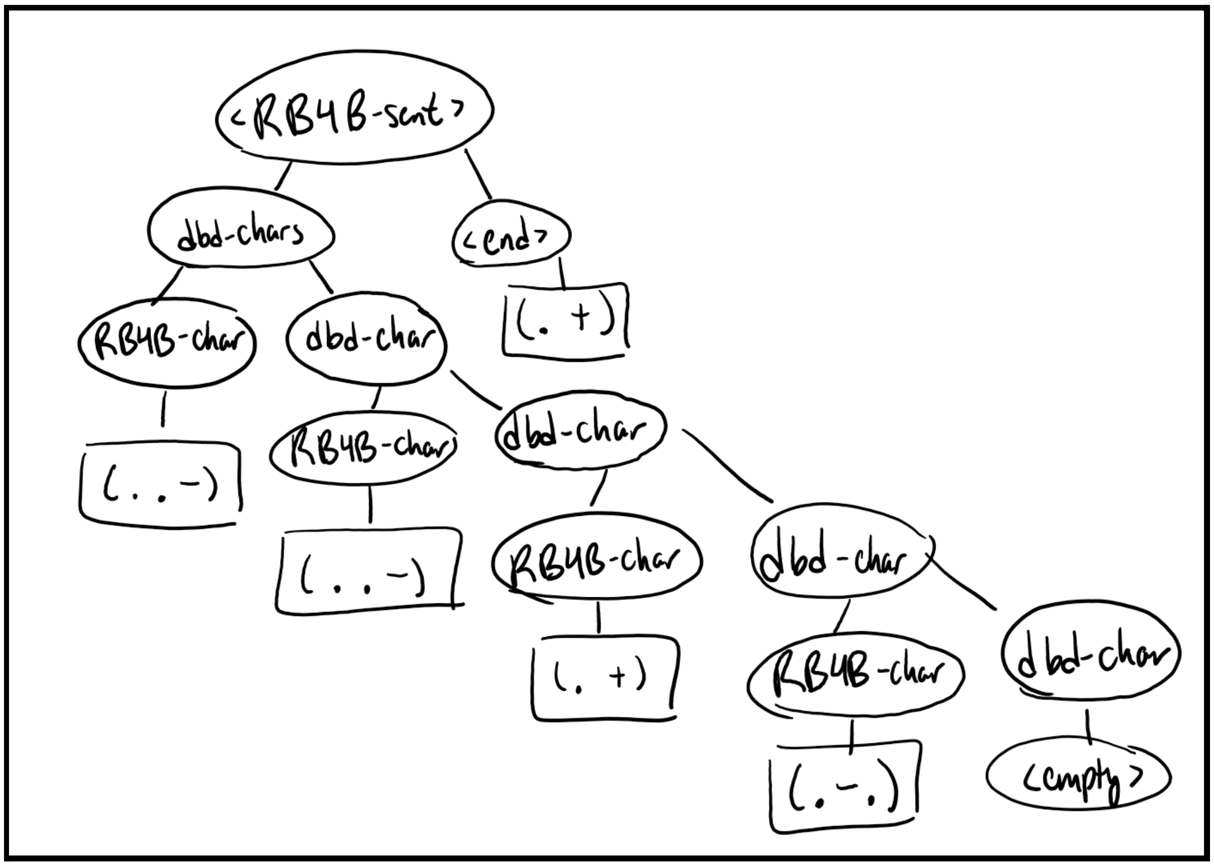
```
9    <RB4B-char> ::= ( . + )
10   <end> ::= ( - )
11   <end> ::= ( . + )
```

Task 2 - Draw a parse tree for `( - )`



Task 3 - draw a parse tree for

`( . . - )( . . - )( . + )( . - . )( . + )`



Problem 2 - SQN (Special Quaternary Numbers)

SQN is a language of all the quaternary numbers which contain no leading zeros (except for the lone digit 0) composed only of the digits `0, 1, 2, 3` . The numbers also cannot have two of the same digits adjacent to another.

SQN examples
- 0
- 102030201
- 102323232301010
- 12321

SQN non-examples
- 0123123123
- 11
- 12301233
- 1234

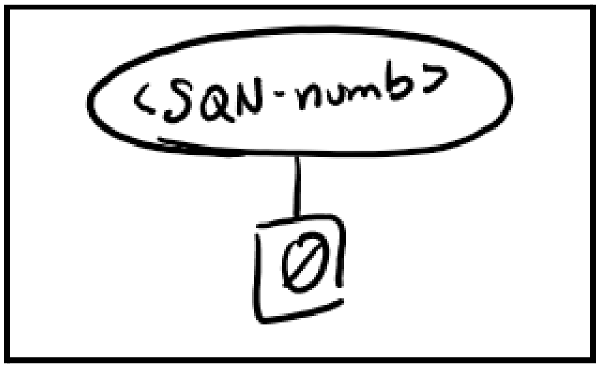
Task 1 - write a BNF grammar description of SQN

```
1    <SQN-numb> ::= 0
2    <SQN-numb> ::= <nzd>

3    <nzd> ::= 1 <nod>
4    <nzd> ::= 2 <nwd>
5    <nzd> ::= 3 <ntd>
6    <nzd> ::= <empty>

7    <nod> ::= 0 <nzd
8    <nod> ::= 2 <nwd
9    <nod> ::= 3 <nwd
10   <nod> ::= <empty>

11   <nwd> ::= 0 <nzd>
12   <nwd> ::= 1 <nod>
13   <nwd> ::= 3 <ntd>
14   <nwd> ::= <empty>
```

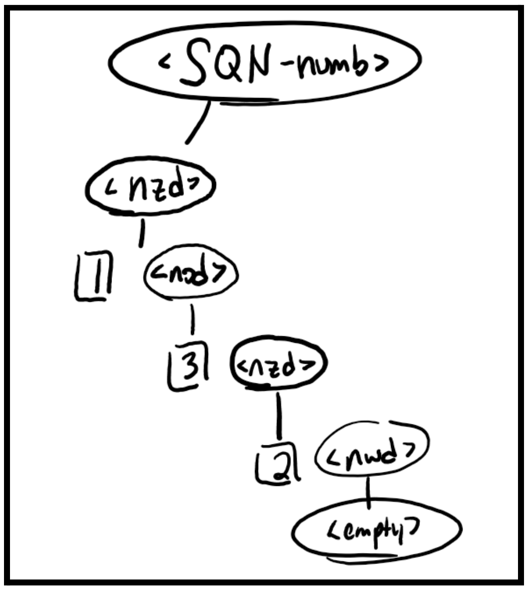
```
15   <ntd> ::= 0 <nzd>
16   <ntd> ::= 1 <nod>
17   <ntd> ::= 2 <nwd>
18   <ntd> ::= <empty>
```

I left the definitions on separate lines for this BNF. This is helpful when referencing rules as you can refer to the rule by line number.

Task 2 - Draw a parse tree for `0`



Task 3 - Draw a parse tree for 132



Task 4 - Explain why a parse tree cannot be drawn for 1223

A parse tree for 1223 would run into an issue following the first two. Because twos are defined as being followed by a <nwd> or non two digit there is no way that the <nwd> could be a two it must be a `0`,1,or 3 followed by its respective non digit non determinate.

Problem 3 - IR123

The language IR123 is composed of all strings formed from the following sequences of bracketed letters. Additionally two adjacent strings cannot be the same occurrence of the predetermined strings.

Predetermined Strings

- `[ C ]`
- `[ D C ]`
- `[ B C ]`
- `[ E D C ]`
- `[ F E C ]`
- `[ G F C ]`

Examples of IR123

- `[ D C ]`
- `[ C ][ D C ][ F E C ]`
- `[ C ][ D C ][ C ] [ D C ][ C ] [ D C ][ C ] [ D C ]`

Examples of IR123

- `[ D C B ]`
- `[ C ][ D B C ][ F E C ]`
- `[ C ][ D C ][ C ][ C ]`

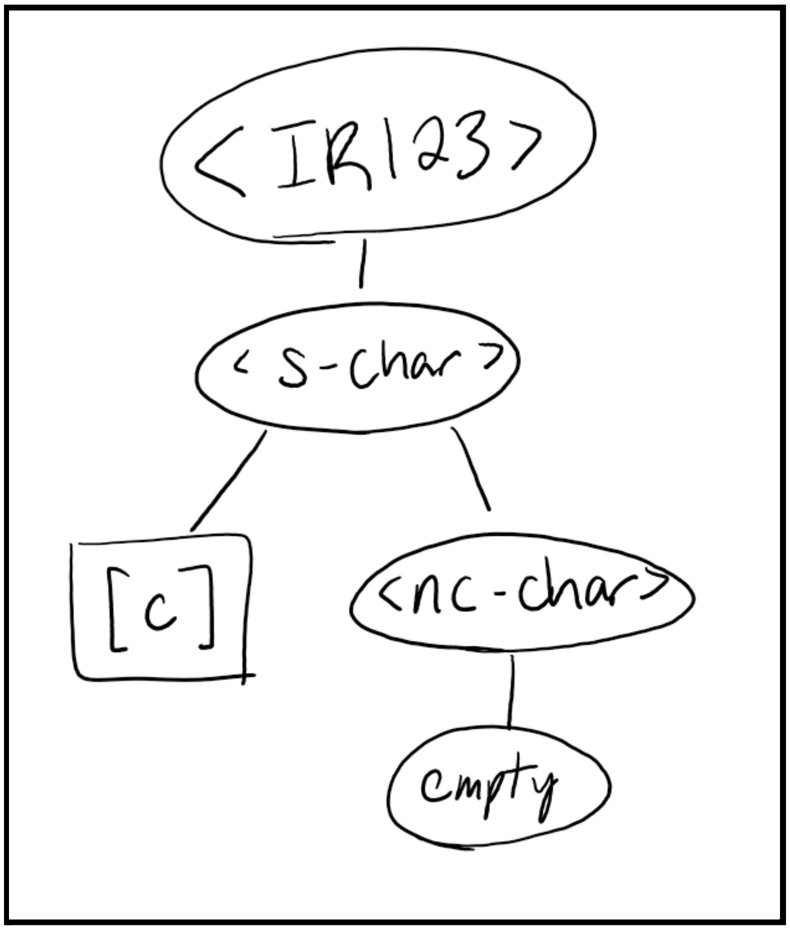
Task 1 - write a BNF grammar description of IR123

```
1    <IR123-sent> ::= <s-char>
2    <s-char> ::= <c-char> | <d-char> | <b-char> |
          <e-char> | <f-char> | <g-char>
3    <c-char> ::= [ C ] <nc-char>
4    <d-char> ::= [ D C ] <nd-char>
5    <b-char> ::= [ B C ] <nb-char>
6    <e-char> ::= [ E D C ] <ne-char>
7    <f-char> ::= [ F E C ] <nf-char>
8    <g-char> ::= [ G F C ] <nb-char>
9    <nc-char> ::= <d-char> | <b-char> | <e-char> |
          <f-char> | <g-char>
```

```
10   <nd-char> ::= <c-char> | <b-char> | <e-char> |
          <f-char> | <g-char> | <empty>
11   <nb-char> ::= <c-char> | <d-char> | <e-char> |
          <f-char> | <g-char> | <empty>
12   <ne-char> ::= <c-char> | <d-char> | <b-char> |
          <f-char> | <g-char> | <empty>
13   <nf-char> ::= <c-char> | <d-char> | <b-char> |
          <e-char> | <g-char> | <empty>
14   <ng-char> ::= <c-char> | <d-char> | <b-char> |
          <e-char> | <f-char> | <empty>
```

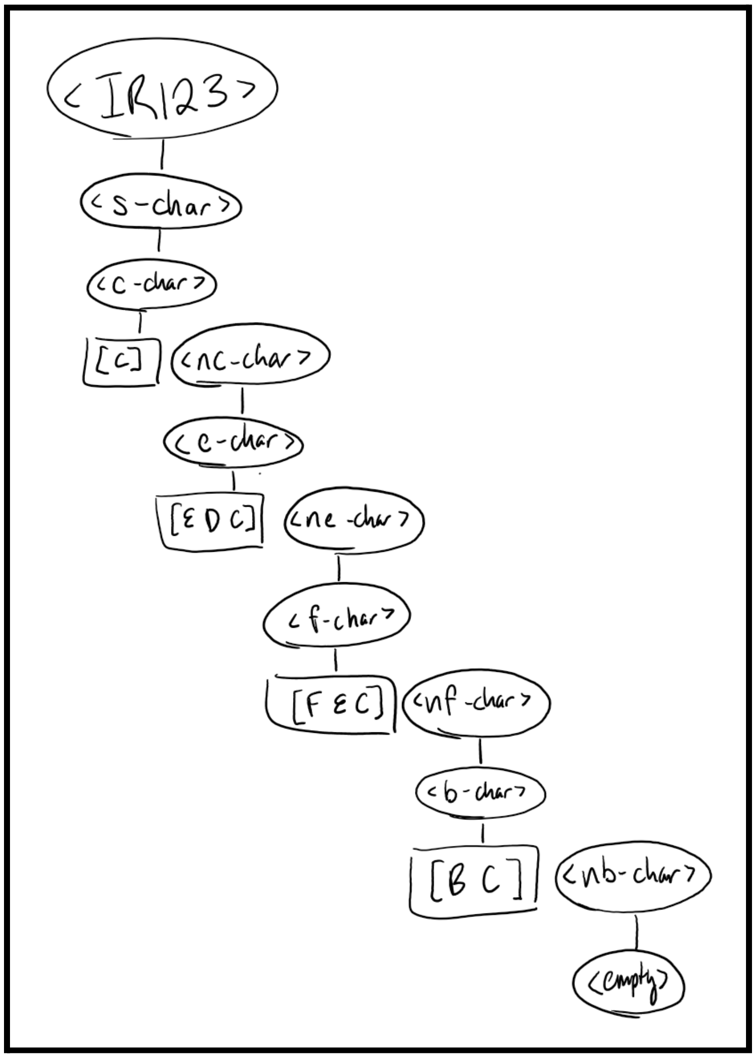
For this BNF grammar I did not leave every rule to its own line. This was done both to save space (as each condense line holds 6 rules) as well as to showcase the different ways a BNF Grammar can be written. Regardless it should be noted that lines 2, and 9-14 each contain 6 rules while the others each hold one.

Task 2 - Draw a parse tree for `[ C ]`



Task 3 - Draw a parse tree for

`[ C ][ E D C ][ F E C ][ B C ]`



Task 4 - Explain why a parse tree cannot be drawn for

`[ D C ][ B C ][ B C ][ C ]`

This parse tree runs into the same issues from Task 4 from the SQN language. After `[ B C ]` a `<nb-char>` must follow. Since the `<nb-char>` cannot parse into `[ B C ]` a parse tree cannot be created.

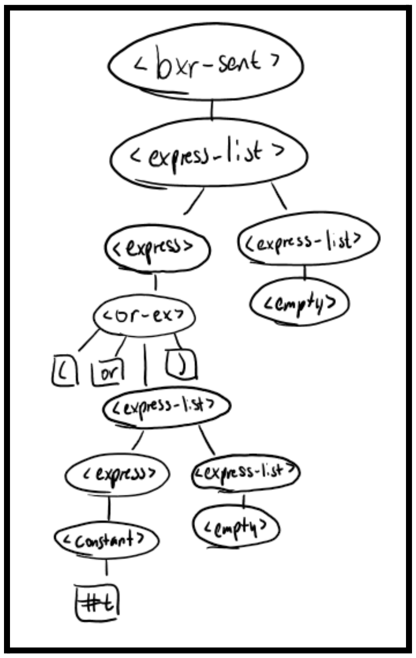
Problem 4 - BXR

BXR is a logical language created using only the constants `#t` and `#f` as well as the operators and, `or`, and `not`.

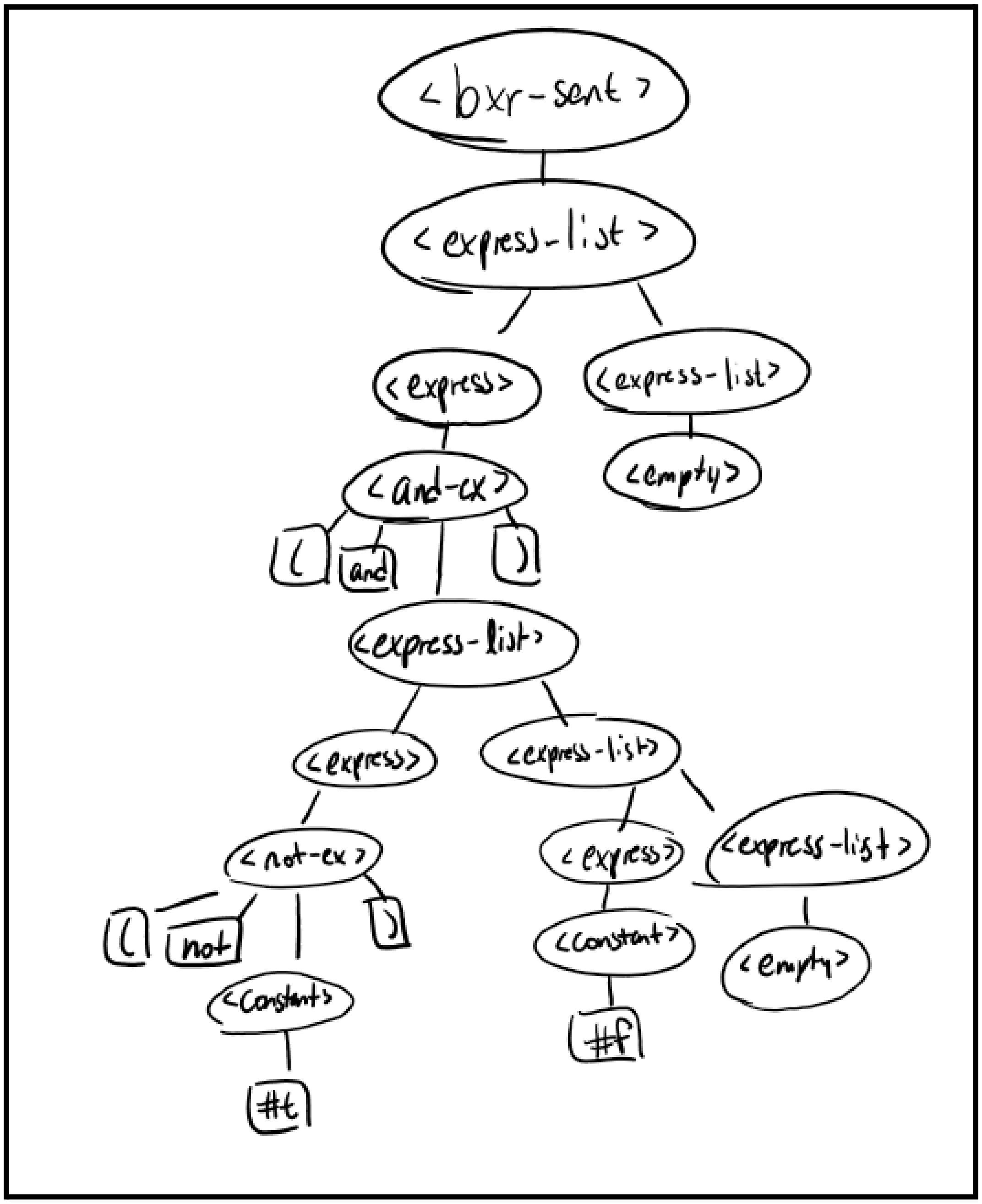
Task 1 - write a BNF grammar description of BXR

```
1     <bxr-sent> ::= <constant>
2     <bxr-sent> ::= <express-list>
3     <constant> ::= #t
4     <constant> ::= #f
5     <express> ::= <and-ex>
6)    <express> ::= <or-ex>
7)    <express> ::= <not-ex>
8)    <express> ::= <constant>
9)    <express-list> ::= <empty>
10)   <express-list> ::= <express> <express-list>
11)   <and-ex> ::= ( and <express-list> )
12)   <or-ex> ::= ( or <express-list>)
13)   <not-ex> ::= ( not <constant> )
```

Task 2 - draw a parse tree for `( or #t )`



Task 3 - draw a parse tree for `( and ( not #t ) #f )`



Problem 5 - Color Fun (CF)

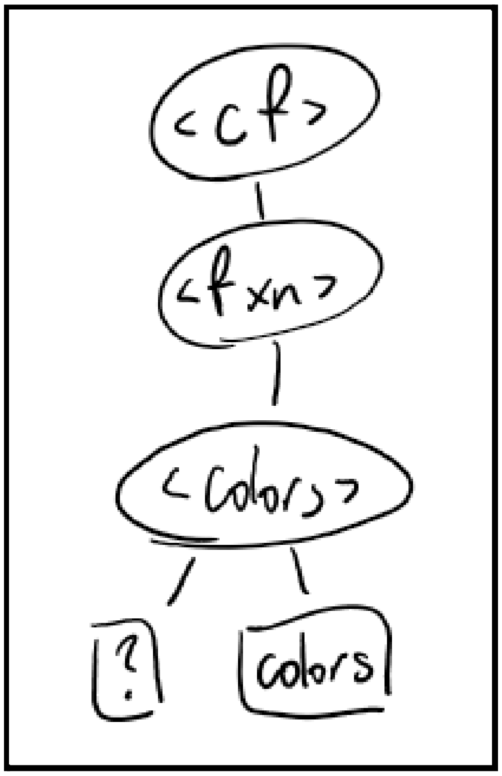
CF is a language in racket used to store and display colors within the interactions pane of Dr. Racket.

Task 1 - write a BNF grammar description of CF

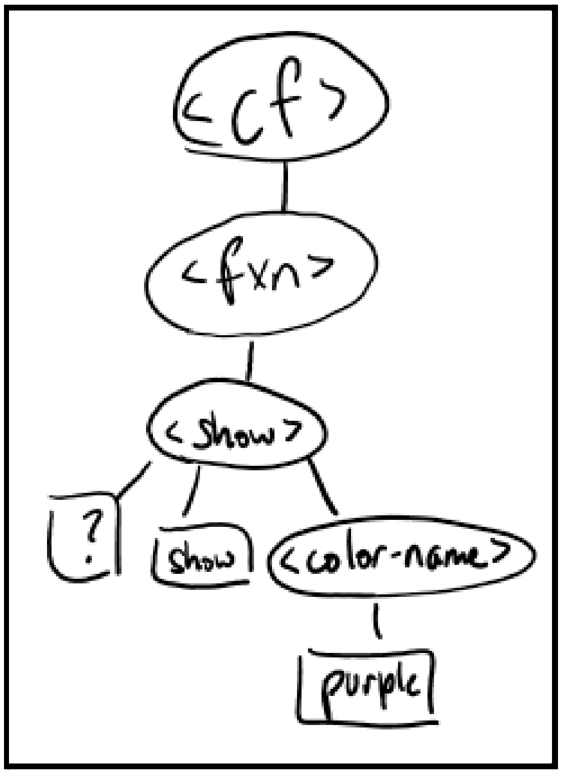
```
1     <cf> ::= <fxn>
2     <fxn> ::= <desc>
3     <fxn> ::= <show>
4     <fxn> ::= <add>
5     <fxn> ::= <colors>
6)    <fxn> ::= <exit>

7)    <desc> ::= ? describe <color-name>
8)    <color-name> ::= red | light red | c1 | c2 | c3 |
           <string>
9)    <string> ::= a string as racket describes strings
           (see racket wiki)

10)   <show> ::= ? show <color-name>

11)   <add> ::= ? add ( <rgb-v> <rgb-v> <rgb-v>
           <opac-v> ) <color-name>
12)   <rgb-v> ::= int range 0-255
13)   <opac-v) ::= <empty>
13)   <opac-v> ::= int range 0-255

14) <colors> ::= ? colors

15) <exit> ::= ? exit
```

This BNF grammar does not state exact tokens and relies that the reader knows what an integer and string are in the context of Racket. Integers are common throughout most programming languages and are left without further explanation while string is left with a link to the racket wiki as programming languages often handle strings slightly differently.

Task 2 - draw a parse tree for- `colors`

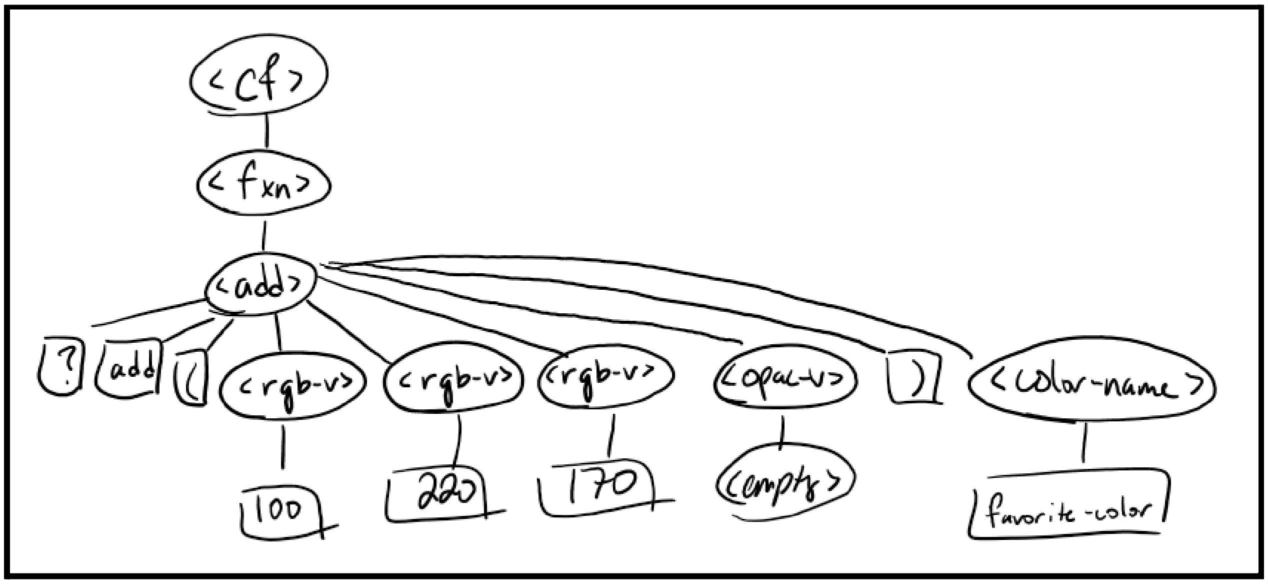


Task 3 - draw a parse tree for `show purple`



Task 4 - draw a parse tree for

```
add ( 100 220 170 ) favorite-color
```



Problem 6 - Color Fun (CF)

Explain BNF in natural language as if you were trying to teach a freshman computer science student.

Backus Naur Form (BNF) is a standardized form for describing programming languages. Creating a standard form helps ensure that any programmer can read the BNF of a language and not only understand the BNF description, but also be able to grasp the language. It takes the form of defining non-terminals (similar to variables in math) and tokens (constants). With the non-terminals eventually being defined in terms of specific tokens. The end goal being that one can draw a line through all of the types of non-terminals until you reach pure tokens for any sentence of the language.