# Csc344 BNF Assignment

By: Miguel Cruz

## Abstract

This assignment is all about BNF. BNF Grammars for given language was composed. and BNF parse trees were drawn.

## Problem 1 - RB4B

### The Language

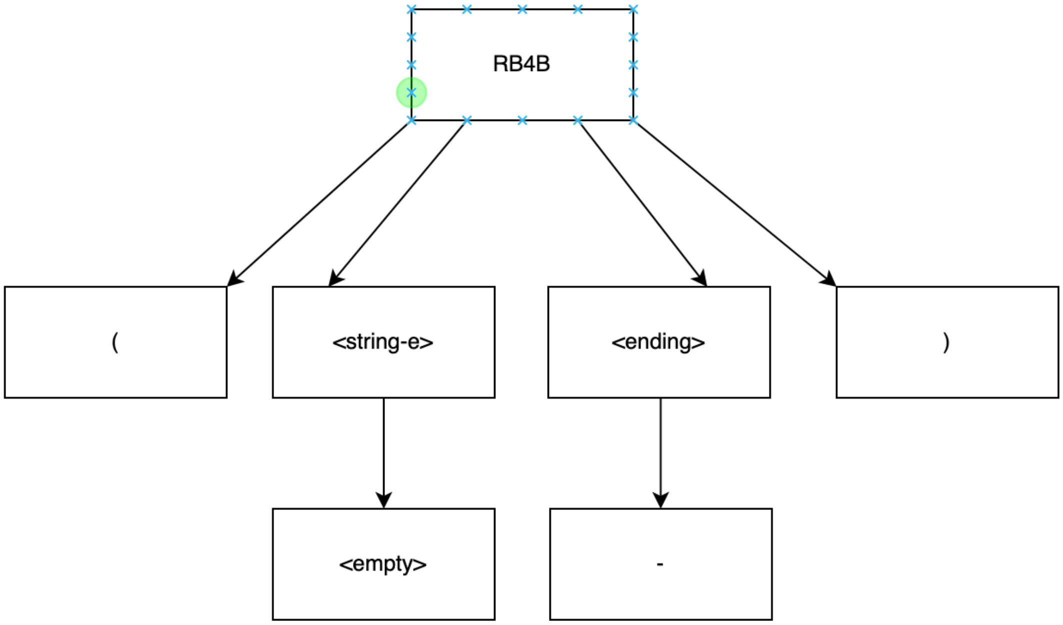
Consider the language RB4B which consists of the set of all strings of one or more occurrences of the following parenthesized sequences of dots and dashes and plusses, which ends with either ( - ) or ( . + ):

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

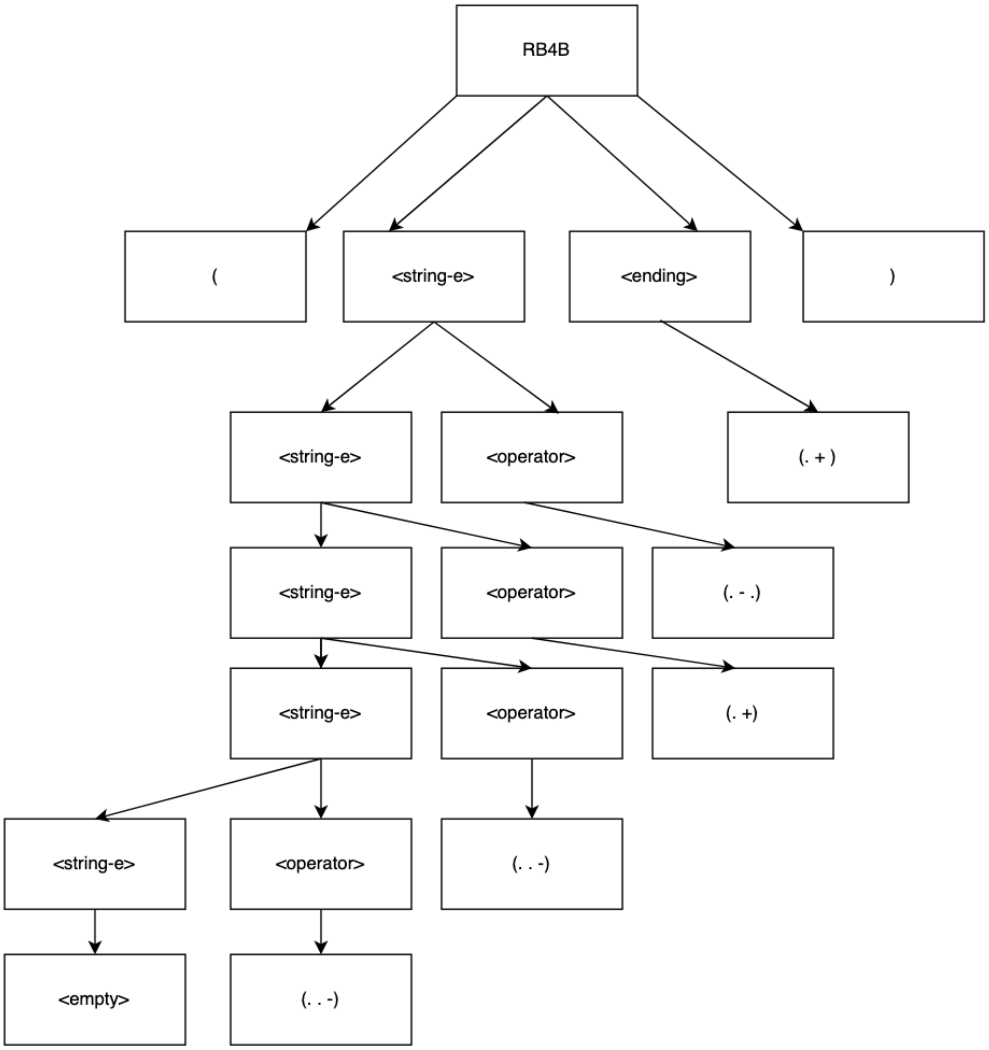
### Grammar Description

```
1. <RB4B> ::= <string-e> <ending>
2. <string-e> ::= <operator> <string-e> | <empty>
3. <operator> ::= ( - ) | ( - - ) | ( - . . ) | ( . - . ) | (. . - ) | ( . + )
4. <ending> ::= ( - ) | ( . + )
```

## BNF Parse Tree 1



## BNF Parse Tree 2



# Problem 2 - SQN (Special Quaternary Numbers)

## The Language

Consider the language SQN which consists of the set of all quaternary numbers with no leading zeros, and with no two adjacent occurrences of the same quaternary digit:

For example, the following are sentences in this language:

1. `0`
2. `123012301230`
3. `10121320212303132`

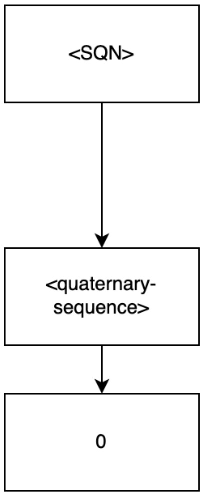
The following strings, for example, are **not** sentences in this language:

1. `1233`
2. `010`
3. `12322221`

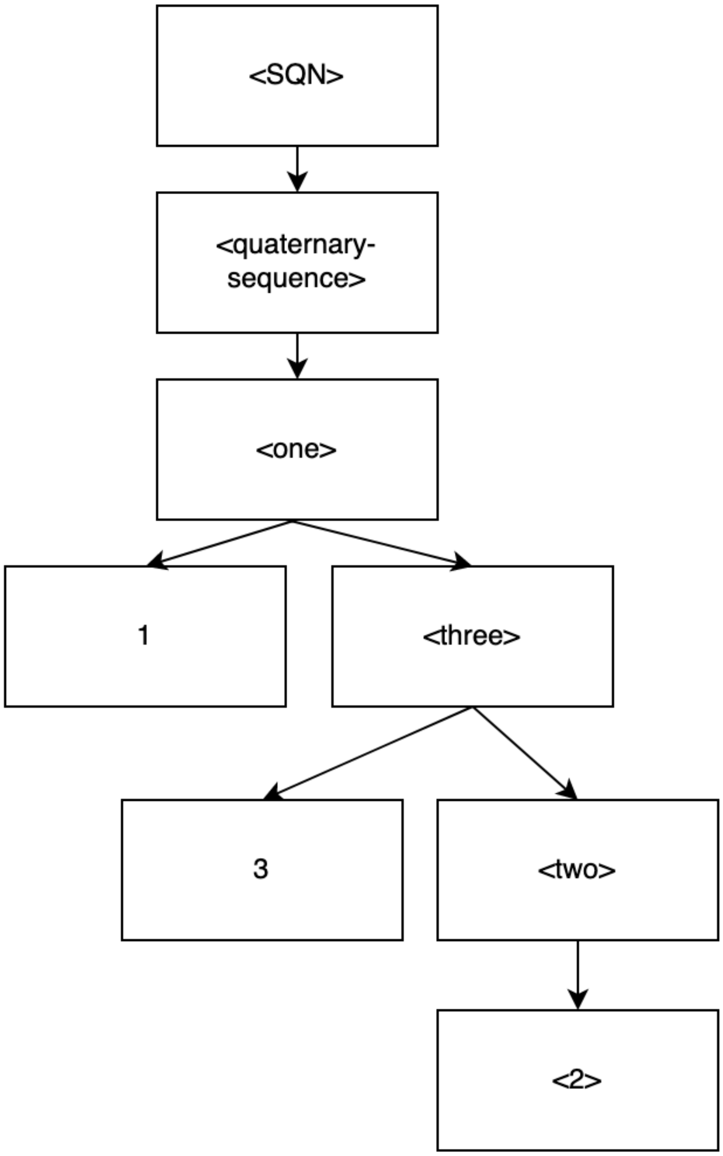
## Grammar Description

```
1. <SQN> ::= <quaternary-sequence>
2. <quaternary-sequence> ::= 0 <empty> | 1 <cantBe1> | 2 <cantBe2> | 3 <cantBe3>
3. <cantBe0> ::= 1 <cantBe1> | 2 <cantBe2> | 3 <cantBe3> | <empty>
4. <cantBe1> ::= 0 <cantBe0> | 2 <cantBe2> | 3 <cantBe3> | <empty>
5. <cantBe2> ::= 0 <cantBe0> | 1 <cantBe1> | 3 <cantBe3> | <empty>
6. <cantBe3> ::= 0 <cantBe0> | 1 <cantBe1> | 2 <cantBe2> | <empty>
```

## BNF Parse Tree 1



## BNF Parse Tree 2



## Question

The string, 1223, would not be able to work with the BNF grammar.

# Problem 3 - IR123

## The Language

Consider the language IR123 which consists of the set of all strings of one or more occurences of the following bracketed sequences of letters, subject to the constraint that there are no two adjacent occurrences of the same bracketed sequence:

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

For example, the following are sentences in this language:

1. `[ C ]`
2. `[ C ][ E D C ][ F E C ][ B C ]`
3. `[ D C ][ B C ][ D C ][ B C ][ D C ][ G F C ][ F E C ][ E D C ][ C ]`

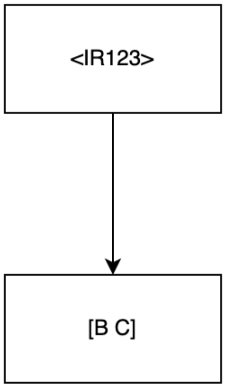
And, for example, the following are not sentences in this language:

1. `[ C ][ C ]`
2. `[ E D C ][ F E C ][ F E C ][ C ]`

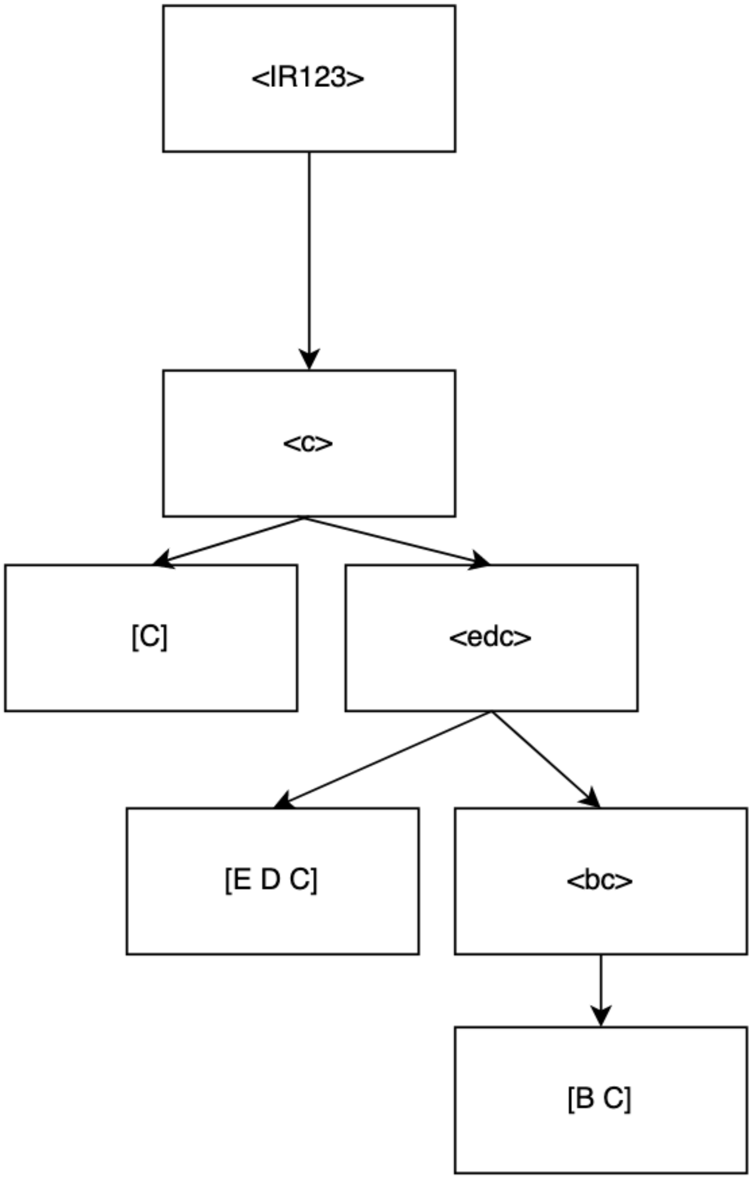
## Grammar Description

```
1. <IR123> ::= <letter-sequence>
2. <letter-sequence> ::= [ C ] <notC> | [ D C ] <notDC> | [ B C ] <notBC> | [ E D C ] <notEDC> | [ F E C ] <notFEC> | [ G F C] <notGFC>
3. <notC> ::= [ D C ] <notDC> | [ B C ] <notBC> | [ E D C ] <notEDC> | [ F E C ] <notFEC> | [ G F C] <notGFC> | <empty>
4. <notDC> ::= [ C ] <notC> | [ B C ] <notBC> | [ E D C ] <notEDC> | [ F E C ] <notFEC> | [G F C] <notGFC> | <empty>
5. <notBC> ::= [ C ] <notC> | [ D C ] <notDC> | [ E D C ] <notEDC> | [ F E C ] <notFEC> | [G F C] <notGFC> | <empty>
6. <notEDC> ::= [ C ] <notC> | [ D C ] <notDC> | [ B C ] <notBC> | [ F E C ] <notFEC> | [ G F C] <notGFC> | <empty>
```

## BNF Parse Tree 1



## BNF Parse Tree 2



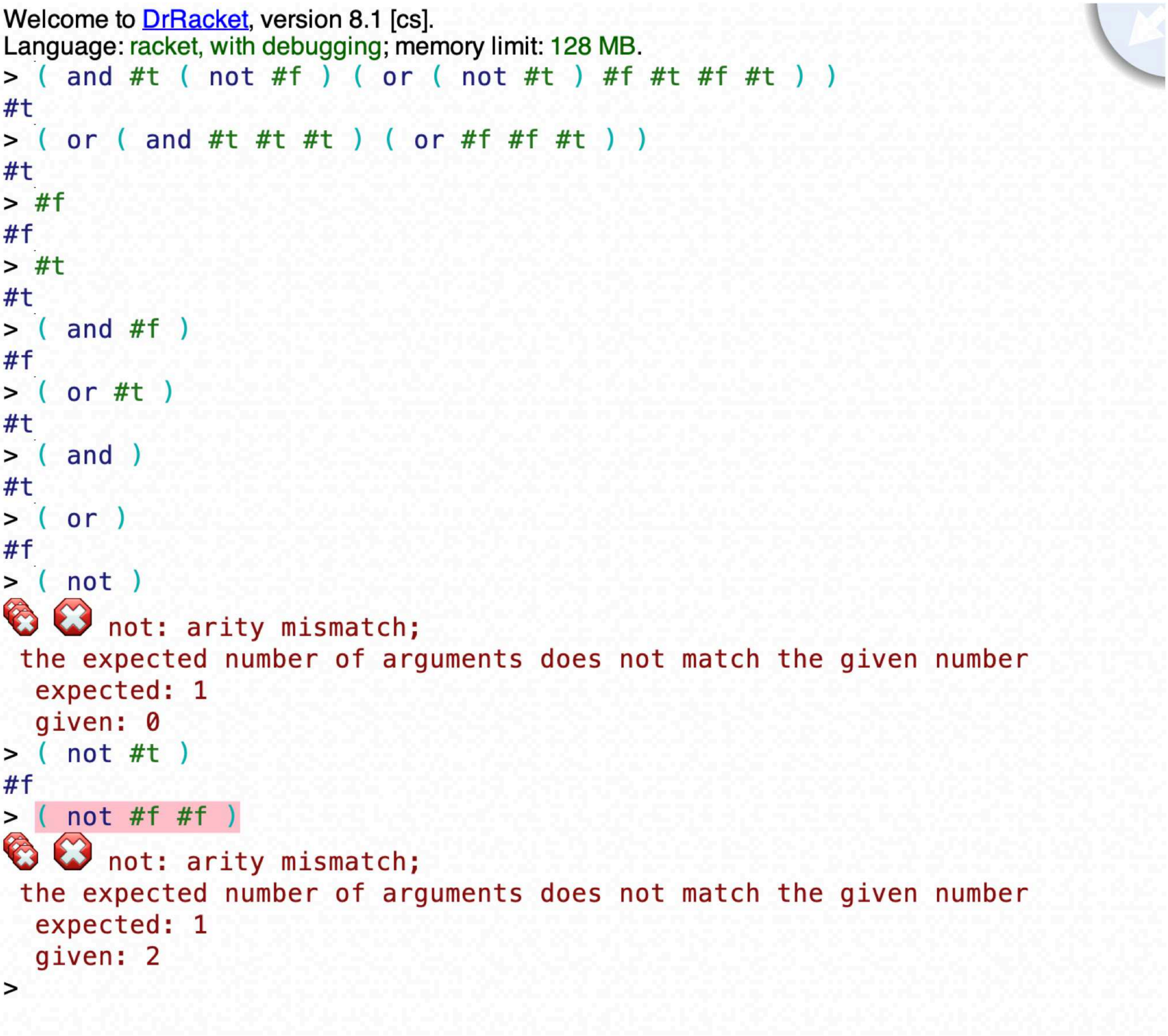
## Question

The statement does not exist in the grammar since <two> := 2 <two>

## Problem 4 - BXR

## The Language

Consider language BXR to be the set of Boolean valued expressions in **Racket** which are composed of the constants `#t` and `#f` and just the operators `and`, `or` and `not`. Here is a short Racket session that provides examples of BXN sentences:

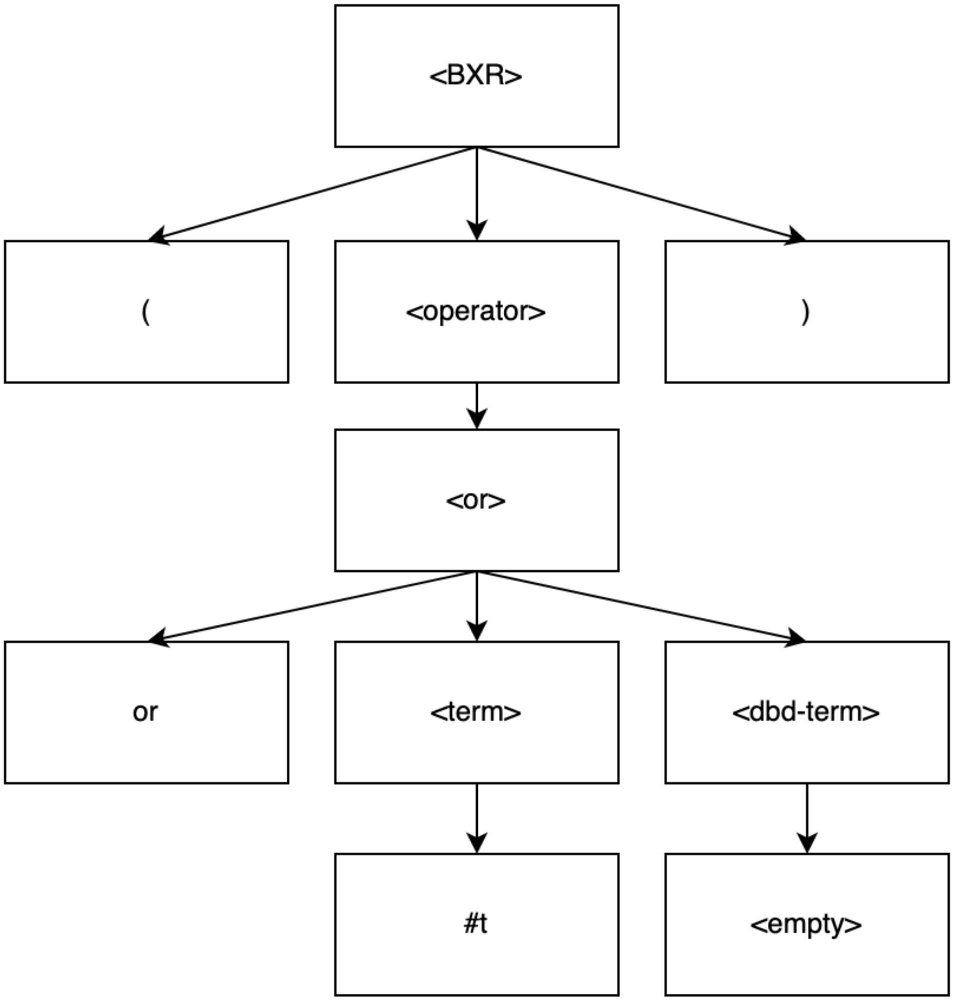


```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( and #t ( not #f ) ( or ( not #t ) #f #t #f #t ) )
#t
> ( or ( and #t #t #t ) ( or #f #f #t ) )
#t
> #f
#f
> #t
#t
> ( and #f )
#f
> ( or #t )
#t
> ( and )
#t
> ( or )
#f
> ( not )
not: arity mismatch;
 the expected number of arguments does not match the given number
  expected: 1
  given: 0
> ( not #t )
#f
> ( not #f #f )
not: arity mismatch;
 the expected number of arguments does not match the given number
  expected: 1
  given: 2
>
```

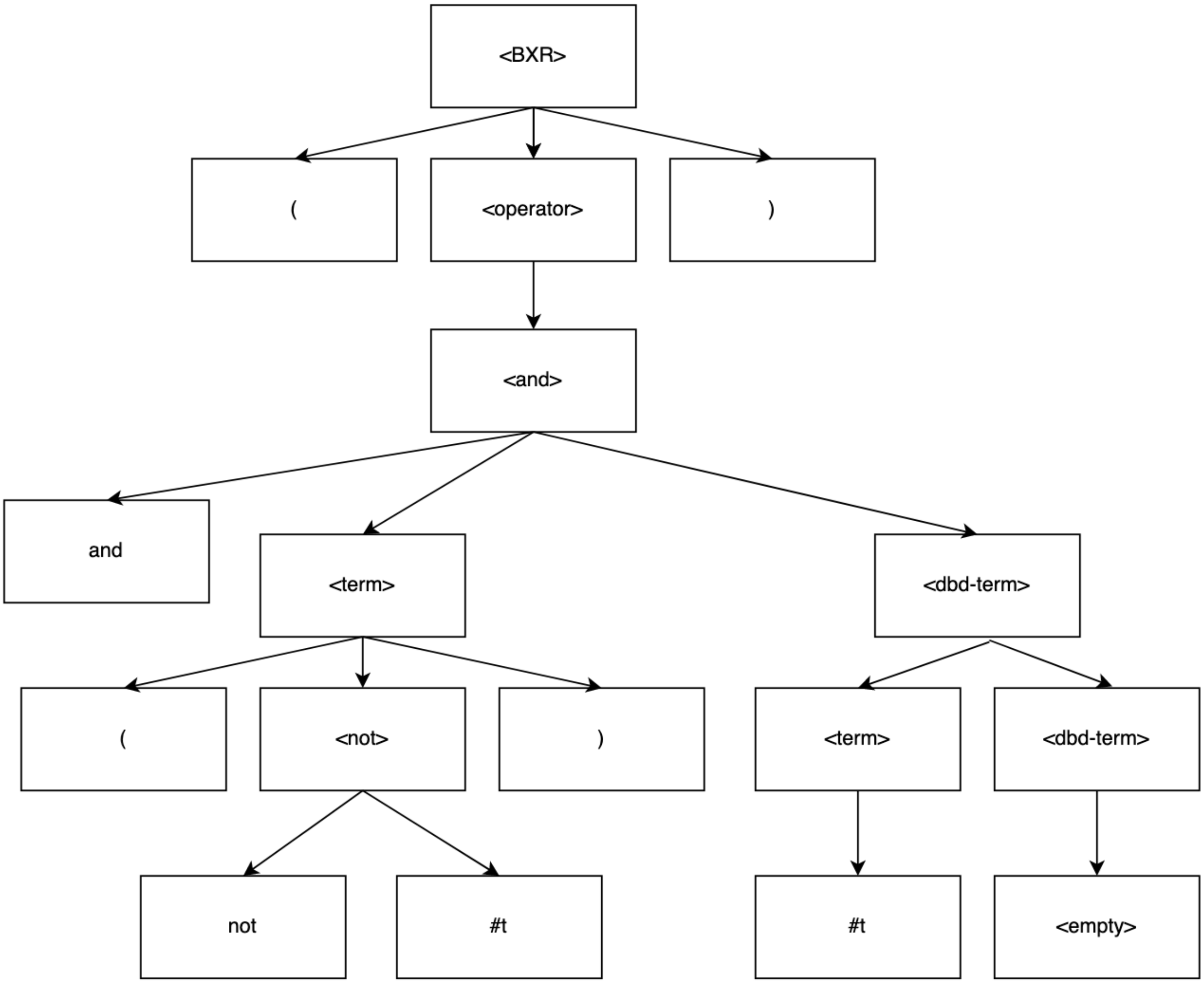
## Grammar Description

```
1. <BXR> ::= #t | #f | ( <operator> )
2.<operator> ::= <and> | <or> | <not>
3.<and> ::= and <term> <dbd-term>
4.<or> ::= or <term> <dbd-term>
5.<not> ::= not <term> <dbd-term>
6.<term> ::= #t | #f | ( <and> ) | ( <or> ) | ( <not> )
7.<dbd-term> ::= <term> <dbd-term> | <empty>
```

## BNF Parse Tree 1



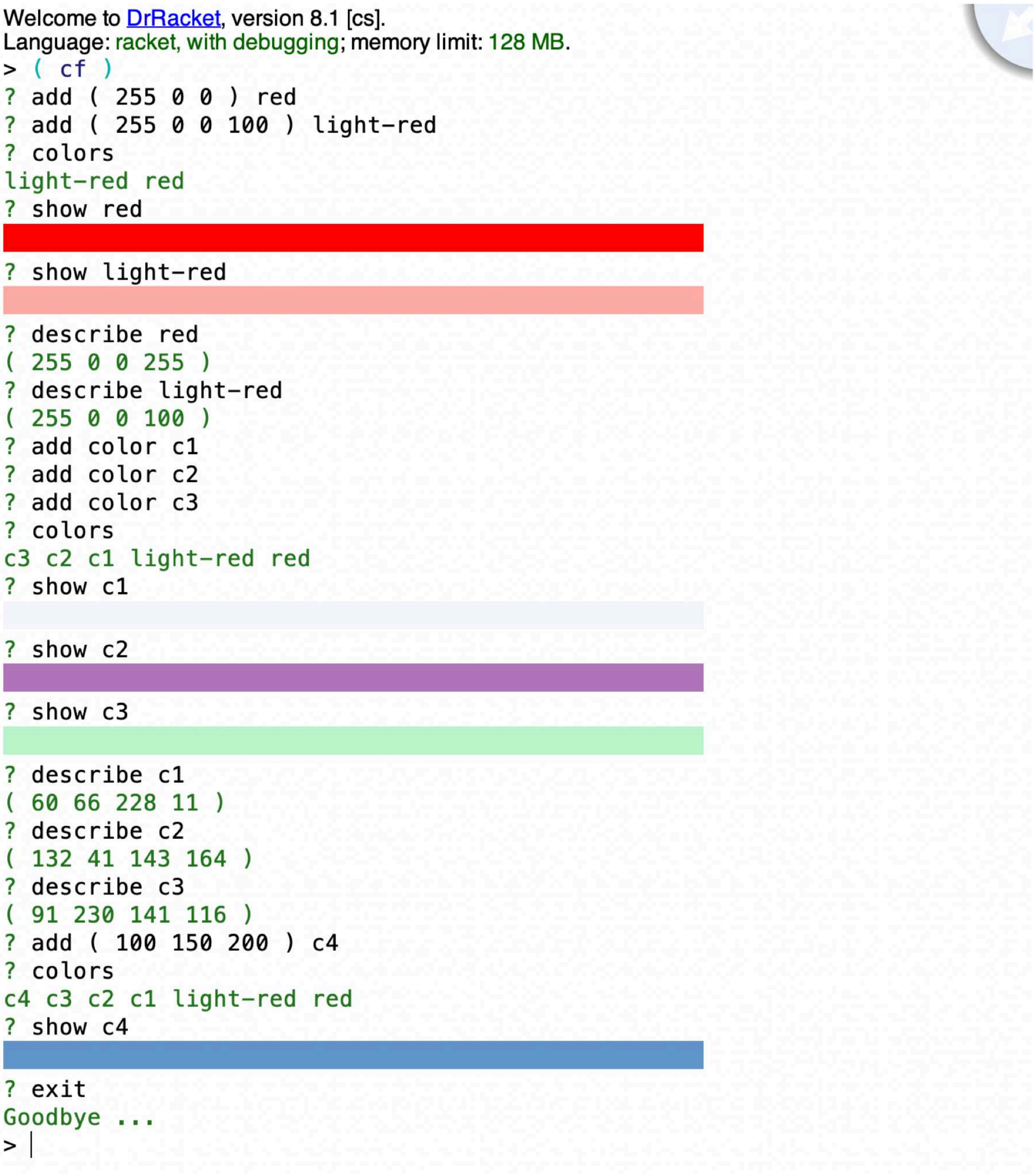
# BNF Parse Tree 2



## Problem 5 - CF (Color Fun)

## The Language

Please consider CF to be a pseudonym for a little language called “Color Fun”. The full language can readily be inferred from the following demo by just looking at the lines beginning with the question mark prompt, and imagining reasonable generalizations in terms of RGB values. All of the sentences of CF are represented, at least suggestively, in the demo.

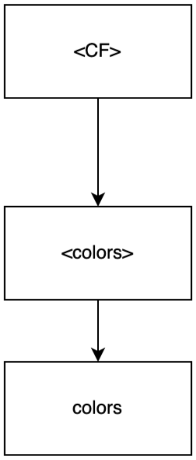


```
Welcome to DrRacket, version 8.1 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( cf )
? add ( 255 0 0 ) red
? add ( 255 0 0 100 ) light-red
? colors
light-red red
? show red

? show light-red

? describe red
( 255 0 0 255 )
? describe light-red
( 255 0 0 100 )
? add color c1
? add color c2
? add color c3
? colors
c3 c2 c1 light-red red
? show c1

? show c2

? show c3

? describe c1
( 60 66 228 11 )
? describe c2
( 132 41 143 164 )
? describe c3
( 91 230 141 116 )
? add ( 100 150 200 ) c4
? colors
c4 c3 c2 c1 light-red red
? show c4

? exit
Goodbye ...
> |
```

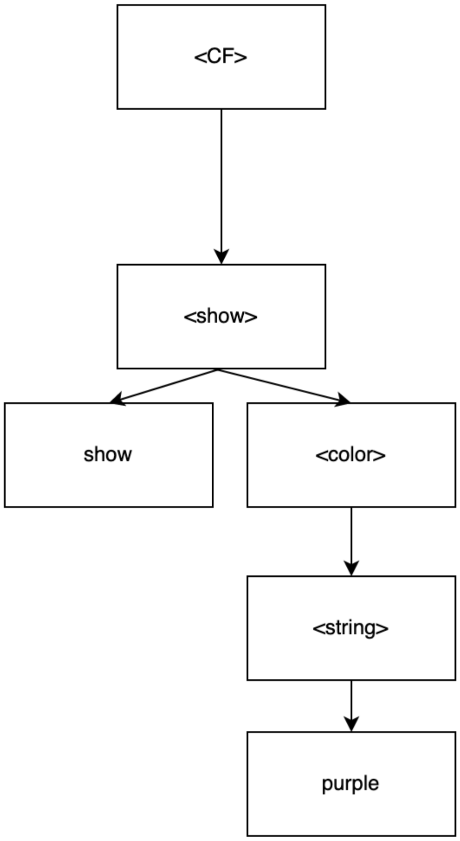
## Grammar Description

```
1. <CF> ::= <colors> | <add-color> | <show-color> | <describe-color> | <exit>
2. <colors> ::= colors | <color-num> | <color-name>
3. <add-color> ::= add <color-num> <color-name>
4. <color-name> ::= <string>
5. <color-num> ::= ( <int> <int> <int> ) | ( <int> <int> <int> <int> )
6. <show-color> ::= show <color-name>
7. <describe-color> ::= describe <color-name>
8. <exit> ::= exit
```

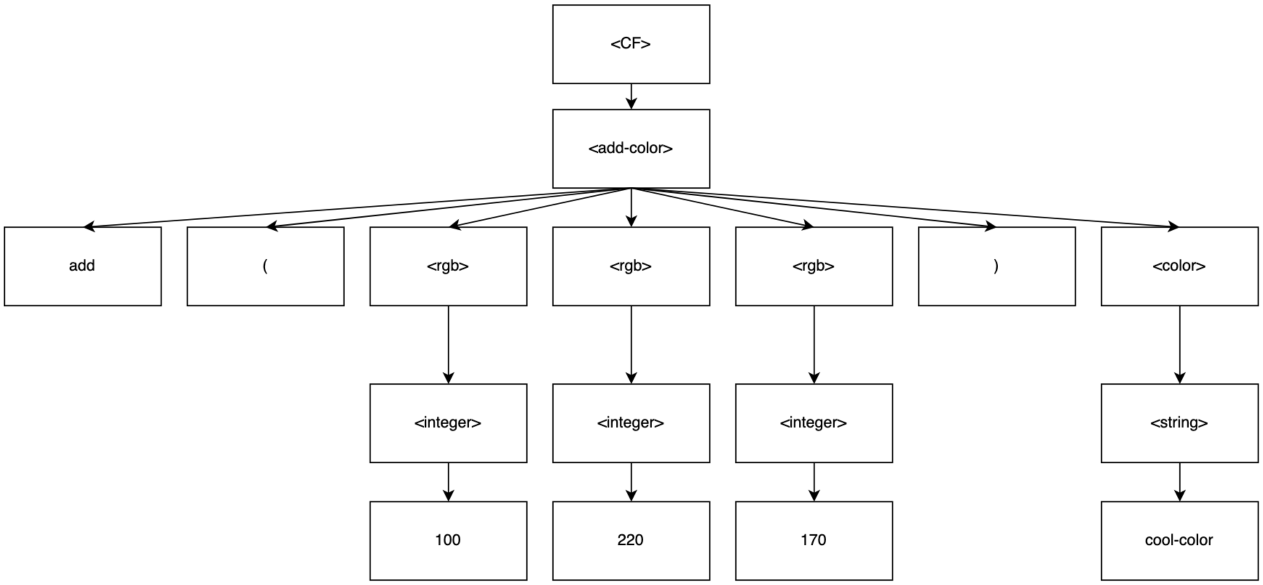
## BNF Parse Tree 1



## BNF Parse Tree 2



## BNF Parse Tree 3



## Problem 6 - BNF?

Imagine that a freshman computer science major asks you the question: "What is BNF?" Please write an answer, **in natural language (English, please), without examples**, in a manner that you believe will serve to meaningfully inform the student about the **nature** and **significance** of BNF. Please do so in no more than 100 words.

Backus–Naur form (BNF) is a notation technique for context-free grammars, often used to describe the syntax of languages used in computing, such as programming languages, document formats, instruction sets and communication protocols. BNF was created by John Backus and Peter Naur in 1960 for use in the ALGOL 58 programming language.