3b) i. **In your write-up, give a refactored version of the *re* grammar from Figure 1 that eliminates ambiguity in BNF (not EBNF).**
Apostrophes mark terminals.

$$
\begin{aligned}
re &::= union \\
union &::= union\ '|'\ intersect \mid intersect \\
intersect &::= intersect\ '\&'\ concat \mid concat \\
concat &::= concat\ not \mid not \\
not &::= \ '\sim'\ not \mid star \\
star &::= star\ '*'\ \mid star\ '?' \mid star\ '+'\ \mid atom \\
atom &::= \ 'c' \mid '\#' \mid '!' \mid '.'
\end{aligned}
$$

ii. **Explain briefly why a recursive descent parser following your grammar with left recursion would go into an infinite loop.**
The way a recursive descent parser works would mean the parser would always match on the left-side non-terminal first. It would never reach any terminals, repeatedly calling a function to match on the non-terminals in an infinite loop.

iii. **In your write-up, give a refactored version of the *re* grammar that replaces left-associative binary operators with *n*-ary versions using EBNF.**

$$
\begin{aligned}
re &::= union \\
union &::= intersect\ \{'|'\ intersect\} \\
intersect &::= concat\ \{'\&'\ concat\} \\
concat &::= not\ \{not\} \\
not &::= \{'\sim'\}\ star \\
star &::= atom\ \{'*' \mid '?' \mid '+'\} \\
atom &::= \ 'c' \mid '\#' \mid '!' \mid '.'
\end{aligned}
$$

iv. **In your write-up, give the full refactored grammar in BNF without left recursion and new non-terminals like *unions* for lists of symbols. You will need to introduce new terminals for *intersects* and so forth.**

$$
\begin{aligned}
re &::= union \\
union &::= intersect\ unions \\
unions &::= \epsilon \mid '|'\ intersect\ unions \\
intersect &::= concat\ intersects \\
intersects &::= \epsilon \mid '\&'\ concat\ intersects \\
concat &::= not\ concats \\
concats &::= \epsilon \mid notconcats \\
not &::= star \mid '\sim'\ not \\
star &::= star\ '*' \mid star \mid atom \\
stars &::= \epsilon \mid '*'\ stars \mid '?'\ stars \mid '+'\ stars \\
atom &::= 'c' \mid '\#' \mid '!' \mid '.'
\end{aligned}
$$

3c) i. **In your write-up, give typing and small-step operational semantic rules for regular expression literals and regular expression tests based on the informal specification given above. Clearly and concisely explain how your rules enforce the constraints given above and any additional decisions you made.**

$$\text{TypeRegex:}\frac{}{\Gamma \vdash /\hat{}re\$/ : RegEx}$$

Similar to the typing rules for other types, TypeRegex simply connects the expression to the proper type.

$$\text{TypeTest:}\frac{\Gamma \vdash e_1 : Regex \qquad \Gamma \vdash e_2 : String}{e_1.test(e_2) : Bool}$$

As per the specification on how tests work, e_1 is required to be a regular expression and e_2 a string. The test is written in the syntax provided and returns a Bool as expected.

$$\text{SearchTest1:}\frac{e_1 \to e_1'}{e_1'.test(e_2) \to e_1'.test(e_2)}$$

Step on e_1 until it reaches a value first.

$$\text{SearchTest2:}\frac{e_2 \to e_2'}{/\hat{}re\$/.test(e_2) \to /\hat{}re\$/.test(e_2')}$$

Step on e_2 until it reaches a value before moving to DoTest rules.

$$\text{DoTestTrue:}\frac{s \in /\hat{}re\$/}{/\hat{}re\$/.test(s) \to true}$$

The function *test* returns true if the string s is in the language described by regular expression *re*. I'm not entirely sure if that is the correct way to write the premise, though I don't know how to write it differently.

$$\text{DoTestFalse:}\frac{s \notin /\hat{}re\$/}{/\hat{}re\$/.test(s) \to false}$$

The function *test* returns false if the string s is not in the language described by regular expression *re*.