Written Set 1 Solutions

Problem One: Subset Construction

i.

\[
\begin{align*}
\text{start} & \quad 0 & & 0, 2 \\
& & b & & a \\
& & b & & a \\
& & a & & b \\
& & b & & a \\
& & a, b & & \\
& & a, b & & \\
\end{align*}
\]

ii.

\[
\begin{align*}
\text{start} & \quad q_0 & & q_1 \\
& & a & & b \\
& & b & & a, b \\
& & a & & b \\
& & b & & a \\
& & a & & b \\
& & b & & a, b \\
\end{align*}
\]

iii.

\[
\begin{align*}
\text{start} & \quad 0 & & 1,4,5 & & 2 \\
& & a & & a & & \\
& & b & & b & & \\
& & a, b & & a, b & & \\
& & a, b & & a, b & & \\
\end{align*}
\]
Problem Two: Maximal Munch

```
%%
a*b          printf("1");
(a|b)*b       printf("2");
c*           printf("3");
```

i. aaabccabbb
The result is **132**, with the tokenization aaab cc abbb.

ii. cbbbbac
The result is **32a3**. The tokenization is c bbbb a c, where the single character 'a' does not
match any regular expression and is thus echoed back to the console.

iii. cbabc
The result is **323**, with tokenization c bab c.

Problem Three: The Limits of Conflict Resolution

Consider this flex script:

```
%%
"aa"          { return 1; }
"a"           { return 2; }
"ab"          { return 3; }
```

The string “aab” could be tokenized as “a,” “ab.” However, maximal-munch will first
match “aa,” and then will fail to match “b.”
Problem Four: Converting Extended Regular Expressions

1. \( R? \), which matches zero or one copies of \( R \):

Intuitively, we can either skip over the machine for \( R \), or work through the machine.

2. \( R^+ \), which matches one or more copies of \( R \):

Intuitively, we have to make it through the machine for \( R \) at least once, and can then cycle around through it as many more times we'd like.

3. \( R^n \), which matches exactly \( n \) copies of \( R \).
   This construction is defined inductively. We match \( R^0 \) with

Then, we will match \( R^{n+1} \) with

This works because we can inductively define \( R^0 = \varepsilon \), and \( R^{n+1} = RR^n \).
Problem Five: Right-to-Left Scanning

i. Modify the existing algorithm for converting regular expressions to NFAs so that the generated NFA accepts the reverse of strings that match the regular expression. Briefly justify why your construction is correct.

There are many approaches to solving this problem. Here are three:

1. You can construct the NFA as before, then reverse all of the transition arrows. Then, make the old accept state a new start state, and the old start state the new accept state.

2. You can transform the regular expression as follows, and then apply the existing algorithm: given a regular expression $R$, define the function REV as follows:

   1. REV(a) = a for any single character a,
   2. REV(ε) = ε for any single character ε,
   3. REV($R_1 \ | \ R_2$) = REV($R_1$) \ | \ REV($R_2$),
   4. REV($R_1 \ R_2$) = REV($R_2$) \ REV($R_1$),
   5. REV($R^*$) = REV($R$)*, and
   6. REV((R)) = (REV(R))

   For example, REV(a(b | c)*d) = d(b | c)*a.

3. You can modify the construction for the $R_1 R_2$ portion of the construction so that instead of chaining $R_1$ into $R_2$, instead you chain $R_2$ into $R_1$, so that the contents of $R_2$ are matched before $R_1$.

ii. Give an example of a set of regular expressions and a string so that the left-to-right scan of the string produces a different set of tokens than the right-to-left scan. Assume that you’re using the maximal-munch algorithm for conflict resolution.

Here is one possible set of regular expression:

```plaintext
%%
aa       { return 1; }
ab       { return 1; }
a | b     { return 2; }
```

If you scan the string aab from left-to-right, you get the tokenization aa b. If you scan this string from right to left, you get a ab.
Problem Six: Slowing Down flex Scanners

Consider the following flex script:

```flex
%%
a*b { return 1; }
a { return 2; }
```

Then let \( f(n) = a^n \) (that is, \( n \) copies of the character \( a \)). When the above scanner runs on this string, it will have to scan all \( n \) characters on the first iteration to check to see that the regular expression \( a*b \) does not match. Since it does not, it will use the second regular expression to match just the first character. The next iteration will scan all remaining \( n-1 \) characters before matching just one \( a \), the iteration after that will scan \( n-2 \) characters, etc. This means that the number of characters scanned is

\[
n + (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n+1)}{2}
\]

which is \( \Theta(n^2) \).