Dr. Olcay Kursun
AUM Computer Science
Formal Languages and Automata Theory
06/08/2022

Practice Questions Solutions:

1. Given $\Sigma = \{0, 1\}$, what is Σ^4? Given $L = \{00, 1, 10\}$, what is L^4?
2. How can we define the reverse of a string? For example, $abc^{rev} = cba$. See the definition of "Dot" above on page 6 (i.e. concatenation of two strings) for inspiration.
3. How can we define the reverse of a language (set of strings)? For example, $\{ab, abc\}^{rev} = \{ba, cba\}$. See the definition of L·M on page 6 for inspiration.
4. What is $\{a, aa, aaa\} \cdot \{a, aa, b\}$ equal to?
5. Describe the language L(G) of the grammar G below.
   $G = (N, \Sigma, P, S)$ where $N = \{S, X, Y\}$, $\Sigma = \{a, b, c\}$, and
   P = { 1: $S \rightarrow aS$,
   2: $S \rightarrow bX$,
   3: $S \rightarrow cY$,
   4: $X \rightarrow bX$,
   5: $X \rightarrow b$
   6: $Y \rightarrow cY$
   7: $Y \rightarrow cc$ }

6. Give 3 example strings (with their sequences of production rules) that belong to L(G) of Question 3.
7. Give 3 example strings that do not belong to L(G) of Question 3.
8. Give a grammar that produces $L = \{ x \in \{0, 1\}^* \mid |x| = 3 \}$
9. Give a grammar that produces $L = \{ x \in \{0, 1\}^* \mid |x|_0 = 3 \}$
10. Give a grammar that produces $L = \{ x \in \{0, 1\}^* \mid |x|_0 = |x|_1 \}$

1. Given $\Sigma = \{0, 1\}$, what is Σ^4? Given $L = \{00, 1, 10\}$, what is L^4?

   Solution:

$\Sigma^0 = \{\lambda\}$
$\Sigma^1 = \{0,1\}$
$\Sigma^2 = \{00,01,10,11\}$
$\Sigma^3 = \{000,001,010,011,100,101,110,111\}$
$\Sigma^4 = \Sigma \cdot \Sigma^3 = \{0000, 0001, 0010, ...\}$, a total of 16 strings

$L^0 = \{\lambda\}$
$L^1 = \{00, 1, 10\}$

L^2 = {0000, 001, 0010, 100, 11, 110, 1000, 101, 1010}, no more than 9 strings, make sure to check we don't have the same string listed twice.

Better write a program for it:

```
def concat(A, B):
  return {x+y for x in A for y in B }

L = {'00', '1', '10'}

L0 = {''}  #by definition (basis)
L1 = concat(L0, L)
L2 = concat(L1, L)
L3 = concat(L2, L)
L4 = concat(L3, L)

print(L4)
```

```
{'1111', '10101010', '100101', '0010000', '1010100', '110100', '101100',
'1010110', '10011', '1010101', '001001', '1010010', '001101', '000011',
'10001010', '11001', '11011', '0010001', '100110', '110000', '101001',
'0000110', '1000010', '00111', '1000100', '00100010', '10000010',
'0010010', '001100', '11100', '1100000', '00101010', '0000101', '11101',
'10100000', '110001', '001011', '1000110', '0010100', '110010',
'10101000', '111000', '1011000', '1011010', '00000000', '101011', '11110',
'0010110', '0000001', '00000010', '10000000', '0010101', '1000000',
'111010', '1000001', '1010000', '00101000', '00100000', '10001000',
'110101', '100011', '1101010', '0000100', '0011010', '1000101', '110110',
'10111', '100100', '0011000', '101110', '1001000', '1001010', '101101',
'00001010', '1101000', '00001000', '100001', '1100010', '10100010',
'1010001', '001110'}
```

```
print(len(L4))
```

```
81
```

2. How can we define the reverse of a string? For example, abc^{rev} = cba. See the definition of "Dot" above on page 6 (i.e. concatenation of two strings) for inspiration.

   Solution:
   How do we implement a string in TFL? As you know the strings are created as members of sigma-star. It is a starting point. Without sigma-star, we don't have strings. So, the definition of sigma-star should give us clues on how to implement a string. We should not be forced to use the "string" primitive data type, we should not have to use "list" or arrays.

   (Streamlined version of the Kleene-star) DEFINITION.
   $\boldsymbol{\Sigma^*}$, the set of all strings over the alphabet, Σ, is given inductively as follows.
   Basis: $\lambda \in \boldsymbol{\Sigma^*}$
   Induction rule: If $x \in \boldsymbol{\Sigma^*}$, and $a \in \Sigma$, then dot(a,x) $\in \boldsymbol{\Sigma^*}$ and dot(x,a) $\in \boldsymbol{\Sigma^*}$.

   In the definition you can use the infix operator (instead of "dot") for example as $a \cdot x \in \boldsymbol{\Sigma^*}$ and $x \cdot a \in \boldsymbol{\Sigma^*}$. But I am using "dot" to help you understand that we are making the definitions ourselves and we are not using some existing definition/formulations of concatenation. After all, programming languages (Python/lists/strings) did not exist before the area of Formal Languages.

   This definition tells us that λ is a string and "a" is a string for every member "a" of the alphabet. The other strings are just obtained by pairing/doting two existing strings.

   To help you understand this better, I will briefly touch upon recursive functions. Recursion builds complex functions from simpler ones (or alternatively, complex ones can be broken into simpler ones). We have some primitive functions that cannot be taken apart any further, such as zero and successor functions. Everything else can be defined by combining these (and few other) primitive functions.

   The successor function that takes x to x + 1 can be used repetitively to obtain other numbers. So, it wouldn't be wrong to say there is nothing called 1; there is S(0). There is nothing called 3; there is S(S(S(0))).

   By the same token, there is nothing like a string of "abc"; there are the following equivalent forms:

   dot(dot(dot(λ, a), b), c)
   dot(a, dot(b, dot(c, λ)))
   dot(a, dot(dot(b, λ), c))

   Note that there is no such thing as dot(dot(a,b), dot(c,d)) because by definition "dot" is a function that requires one of its arguments to be an alphabet symbol. To be more precise, either both of the arguments are equal to lambda or one of the arguments is a character (symbol from sigma). In mathematical terms, this is expressed as:
   dot: $\{(\lambda,\lambda)\} \cup \Sigma^* \times \Sigma \cup \Sigma \times \Sigma^* \rightarrow \Sigma^*$.

The bottom-line is if a string (a member of sigma-star) is not lambda, then it can be decomposed to a "dot" of a string and a character.

So, how do we reverse x = dot(dot(dot(λ, a), b), c)
Well x must be dot of a string (suppose we call that y) and an alphabet symbol (suppose we call that s).

x = dot(y, s) where y = dot(dot(λ, a), b) and s = 'c'

In that case, Reverse is a function that works as follows
Reverse(x) = Reverse(dot(y, s)) = dot(s, Reverse(y))

Of course Reverse(y) is computed recursively. Sometimes x = dot(y, s) will be used and sometimes we will have x = dot(s, y) in hand. One of them must apply, of course until we get to lambdas when we should terminate the recursion at x=λ when Reverse(x) is simply computed as Reverse(λ) = λ.

Let's show it on an example:

```
Reverse( dot( dot( dot(λ, a), b), c) )   = dot(c, Reverse( dot( dot(λ, a), b) ) )
                                         = dot(c, dot(b, Reverse( dot(λ, a) ) ) )
                                         = dot(c, dot(b, dot(a, Reverse(λ) ) ) )
                                         = dot(c, dot(b, dot(a, λ ) ) )
```

Of course dot(c, dot(b, dot(a, λ))) is not what we want to see on the screen, so we can convert that object to string if we write all this as a Python program by defining __str__ method for the string object.

Let's show it on another example:

```
Reverse( dot(a, dot(dot(b, λ), c)) )     = dot(Reverse(dot(dot(b, λ), c)), a)
                                         = dot(dot(c, Reverse(dot(b, λ))), a)
                                         = dot(dot(c, dot(Reverse(λ), b)), a)
                                         = dot(dot(c, dot(λ, b)), a)
```

Again, of course, this answer is not what we want to see on the screen but when we print it we can simply print the characters one after another as "cba". Printing it as "cba" does not change the fact that we had a "linked list" of alphabet symbols in memory. This sort of representation appears to be counter-productive but greatly improves mathematical formulations, recursive computations, and proofs by recursion/induction.

3. How can we define the reverse of a language (set of strings)? For example, $\{ab, abc\}^{rev} = \{ba, cba\}$. See the definition of L·M on page 6 for inspiration.

   Solution:

   Reverse(L) = { Reverse(x) | for all x in L}

   Note that Reverse is overloaded: Reverse(L) is defined for sets of strings and Reverse(x) is defined in the previous question for string inputs.

4. What is {a, aa, aaa} · {a, aa, b} equal to?

   Solution:
   {aa, aaa, aaaa, aaaaa, aaab, aab, ab}

```
{x+y for x in {'a', 'aa', 'aaa'} for y in {'a', 'aa', 'b'} }
```

5. Describe the language L(G) of the grammar G below.
   G = (N, Σ, P, S) where N = {S, X, Y}, Σ = {a, b, c}, and

   P = { 1: $S \rightarrow aS$,
   2: $S \rightarrow bX$,
   3: $S \rightarrow cY$,
   4: $X \rightarrow bX$,
   5: $X \rightarrow b$
   6: $Y \rightarrow cY$
   7: $Y \rightarrow cc$ }

   Solution:

   L(G) = {a}* ({bb}{b}* ∪ {ccc}{c}*) or alternatively
   $L(G) = \{a^n b^m \mid n \geq 0, m \geq 2\} \cup \{a^n c^m \mid n \geq 0, m \geq 3\}$

6. Give 3 example strings (with their sequences of production rules) that belong to L(G) of Question 3.

   Solution:
   The strings abb, ccc, aaaabbbb are exemplary members of L(G) with the following production rule sequences: 125, 37, and 11112445.

7. Give 3 example strings that do not belong to L(G) of Question 3.

   Solution:
   The strings bba, bcbcbc, aaa are exemplary non-members of L(G)

8. Give a grammar that produces L= { x ∈ {0, 1}* | |x| = 3 }

   Solution:
   Let us generate all the strings over {0,1} of length 3 as follows.
   G = (N, Σ, P, S), where N={S,A,B} and Σ = {0,1} and P has the following rules:
   $S \rightarrow 0A$
   $S \rightarrow 1A$
   $A \rightarrow 0B$
   $A \rightarrow 1B$
   $B \rightarrow 0$
   $B \rightarrow 1$

Using the rules above is more efficient. But, alternatively, we can even write the rules as follows (which works for finite languages but you cannot have infinitely many rules if the given language was infinite).

$S \rightarrow 000$
$S \rightarrow 001$
$S \rightarrow 010$
$S \rightarrow 011$
$S \rightarrow 100$
$S \rightarrow 101$
$S \rightarrow 110$
$S \rightarrow 111$

9. Give a grammar that produces L= { x ∈ {0, 1}* | $|x|_0 = 3$ }

   Solution:
   If the number of 0 symbols in the member strings is expected to be 3 (no restrictions on the number of 1s), then we can construct the grammar as follows:
   G = (N, Σ, P, S), where N={S,A,B} and Σ = {0,1} and P has the following rules:
   $S \rightarrow 0A$
   $S \rightarrow 1S$
   $A \rightarrow 0B$
   $A \rightarrow 1A$
   $B \rightarrow 0C$
   $B \rightarrow 1B$
   $C \rightarrow \lambda$
   $C \rightarrow 1C$

The grammar above and the grammar in Question 8 are right-linear-grammars (RLG), which means that the left-hand-side of the rules is always a single variable and the right-hand-side of the rules has at most one variable and when that variable is there it is the rightmost symbol of the rule.

For example, let us generate “001011” ?
$S \Rightarrow 0A \Rightarrow 00B \Rightarrow 001B \Rightarrow 0010C \Rightarrow 00101C \Rightarrow 001011C \Rightarrow 001011.$

10. Give a grammar that produces $L = \{ x \in \{0, 1\}^* \mid |x|_0 = |x|_1 \}$

    Solution:
    Checking equality of the number of 0s and 1s in a string requires higher complexity. The rules will be more complex than RLGs.

1: $S \rightarrow SS$
2: $S \rightarrow 0S1$
3: $S \rightarrow 1S0$
4: $S \rightarrow \lambda$

For example, let us find which sequence of the rules generates “001011” ?
$S \Rightarrow^2 0S1 \Rightarrow^1 0SS1 \Rightarrow^2 00S1S1 \Rightarrow^4 001S1 \Rightarrow^2 0010S11 \Rightarrow^4 001011.$

As you see we have sentential forms that contain more than one variable such as 0SS1 and 00S1S1. In an RLG the variable (only one) would always be the rightmost (because the rules always end with one variable as described above).