Basic Formal Language Theory Solutions

James Rogers
jrogers@cs.ucf.edu

PART I

Basic Concepts—Solutions

Part I
Basic Concepts

1 Computation and Languages

1. Prove Claim 2. Show how, using an algorithm to solve the problem as a subroutine, one can construct an algorithm for checking the problem. This claim depends only on the assumption that there is a unique solution for every instance (1 & 2), not on the assumption that instances and solutions can be enumerated (3).

   (Solution)

   Suppose we have an algorithm for solving the problem. Given an instance and a possible solution we can check the solution by running the algorithm for solving the problem on that instance. If the result is the same as the proposed solution we answer 'Yes' otherwise we answer 'No'.

   To verify correctness of this algorithm we have to show both that we only answer 'Yes' when the proposed solution is correct and that whenever the proposed solution is correct we will answer yes. The first is a consequence of the correctness of the algorithm for solving the problem. We only answer 'Yes' if the proposed solution is the same as the solution returned by the algorithm for solving the problem, which, by assumption, is a correct solution. The second part depends on the fact that there is a unique solution for each instance. Thus if the proposed solution is correct it will be the same as the solution provided by the solving algorithm and we will answer 'Yes'.

2 Formal Languages

2. Is L^* ever empty? What about L^+? Under what circumstances does L^+ contain ε? (Consider the cases: $L = \emptyset$ and $L = \{\varepsilon\}$.)

   (Solution)

   Since $L^0 \subseteq L^*$ and $L^0 \overset{\text{def}}{=} \{\varepsilon\}$, L^* is never empty. L^+, on the other hand, will be empty if $L = \emptyset$, since $L^i \overset{\text{def}}{=} L^{i-1} \cdot L$ for all $i \geq 1$ and $L' \cdot \emptyset = \emptyset$ for any L'.
   $\varepsilon \in L^+$ iff $\varepsilon \in L$.

3 An Informal Preview

3. First model: Computer has a fixed number of bits of storage. You will model this by limiting your program to a single fixed-precision unsigned integer variable, e.g., a single one-byte variable (which, of course, can store only values in the range $[0, \ldots, 255]$), etc. Limit yourself, further, to a single call to `input()` which occurs in the argument of a `case` (or `switch`) statement. The reason for this will become clear in the last part of this question.

   (a) Sketch an algorithm to recognize the language: $\{(ab)^i \mid i \geq 0\}$ (that is, the set of strings in $\{a, b\}^*$ consisting of zero or more repetitions of ab: $\{ab, abab, ababab, \ldots\}$).

   (Solution)

```
last :='b';   ---Tracks previous input symbol
loop
{  case (input())
   {  'a':  if (last == 'b') then
            {  next();
               last :='a'
               }
            else
            {  exit(False) }
            endif;
      'b':  if (last == 'a') then
            {  next();
               last :='b'
               }
            else
            {  exit(False) }
            endif;
      EOF:   if (last == 'b') then
             {  exit(True) }
             else
             {  exit(False) }
             endif;
      }
   endcase
   }
endloop
```

(b) How many bits do you need for this (how much precision do you need)? Can you do it with a single bit integer?

(Solution)

All that is necessary is to keep track of whether the last symbol was an '*a*' or an '*b*'. Thus a single bit suffices.

(c) Sketch an algorithm to recognize the language: $\{(abbba)^i \mid i \geq 0\}$.

(Solution)

Here we can let the variable `last` range over the set $\{\varepsilon, a, ab, abb, abbb, \text{Fail}\}$, i.e., it is an enumeration type taking values that are proper prefixes of '*abbba*' (plus Fail).

```
last :=$\varepsilon$;   ---Tracks previous input
loop
{  case (input())
   {  '$a$':  case (last)
              {  $\varepsilon$:  {  last:='$a$'; next() };
                 '$a$','$ab$','$abb$',Fail:  {  last:=Fail; next() };
                 '$abbb$':  {  last:=$\varepsilon$; next() }
                 };
      '$b$':  case (last)
              {  $\varepsilon$,'$abbb$',Fail:  {  last:=Fail; next() };
                 '$a$':  {  last:='$ab$'; next() }
                 '$ab$':  {  last:='$abb$'; next() }
                 '$abb$':  {  last:='$abbb$'; next() }
                 };
      EOF:   if (last == $\varepsilon$) then
             {  exit(True) }
             else
             {  exit(False) }
             endif;
      }
   endcase
   }
```

```
endloop
```

(d) How many bits do you need for this?

(Solution)

This solution needs to distinguish between the six values of `last` so, evidently, three bits are needed.

(e) Suppose we relax the last limitation and allow any (finite) number of calls to `input` occurring anywhere in the program. Sketch an algorithm for recognizing the language of part (a) using (apparently) *no* data storage. Argue that any algorithm for recognizing this language must store at least one bit of information. Where does your program store it?

(Solution)

```
while (input()=/= EOF)
{  if (input() == 'a') then   ---Have seen wa for $w \in (ab)^*$
   {  next();
      if (input() == 'b') then   ---Have seen wab for $w \in (ab)^*$
      {  next() }
      else
      {  exit(False) }
      endif
      }
   else
   {  exit(False) }
   endif
   }
endwhile;
exit(True)
```

While this code uses no variables it must, in any case, distinguish the situation in which the previous input was an '*a*' from that in which it was a '*b*'—in the one case a '*b*' is expected, in the other a '*b*' should result in failure. Here the value of the previous input symbol is tracked by the position in the code. In effect, the variable `last` is being stored in the program counter.

4. Second model: Computer has a single unbounded precision counter which you can only increment, decrement and test for zero. (You may

assume that it is initially zero or you may include an explicit instruction to clear.) Limit your program to a single unsigned integer variable, and limit your methods of accessing it to something like `inc(i)`, `dec(i)` and a predicate `zero?(i)` which returns true iff $i = 0$. This integer has unbounded precision—it can range over the entire set of natural numbers—so you never have to worry about your counter overflowing. It is, however, restricted to only the natural numbers—it cannot go negative, so you cannot decrement past zero.

(a) Sketch an algorithm to recognize the language: $\{a^i b^i \mid i \geq 0\}$. This is the set of strings consisting of zero or more 'a's followed by *exactly the same number* of 'b's.

(Solution)

Here we need to check the string for two properties: all 'a's precede all 'b's and the number of 'a's and 'b's are equal. The idea is to increment the counter while scanning the 'a's and decrement it while scanning the 'b's. At all points in the computation the value of the counter will be the number of 'a's scanned so far minus the number of 'b's. If it is zero at the end of the string then the string is in the language.

The variable `cnt` will be the counter; assume it is initialized to zero.

```
while (input() == 'a')
{  inc(cnt); next() }
endwhile;
while (input() == 'b' and not(zero?(cnt)))
  ---Exits prior to EOF if input not in $a^*b^*$ or if $a^ib^j$, $j > i$.
{  dec(cnt); next() }
endwhile;
exit(input() == EOF and zero?(cnt))
```

(b) Can you do this within the first model of computation? Either sketch an algorithm to do it, or make an informal argument that it can't be done.

(Solution)

The reason this cannot be done in the first model is that we can put no finite bound on the number of 'a's we may have to count. In particular the program will have to be able to distinguish strings that start with a sequence of i 'a's from those that start with j 'a's for every i and $j \neq i$. Why? Because if the rest of the string is a sequence of i 'b's we will want to accept it in the first case but reject it in all the others. But if we we restrict ourselves to a k-bit variable, one that can hold values between 0 and $2^k - 1$, we will only be able to distinguish 2^k strings altogether. Thus, if we can correctly recognize a^ib^i for $0 \leq i < 2^k$, we will have to confuse strings starting with 2^k or more 'a's with strings starting with fewer than 2^k in which case we will be bound to accept some string with fewer 'b's than 'a's.

(c) Give an informal argument that one can't recognize the language: $\{a^ib^ic^i \mid i \geq 0\}$ within this second model of computation (i.e, with a single counter).

(Solution)

Here the intuition is that in order to check that the number of 'b's is equal to the number of 'a's we have to lose the count of 'a's. The counter can be used just once; it can test equality of two strings but not three.

5. Third model: Computer has a single LIFO stack containing fixed precision unsigned integers (so each integer is subject to overflow problems) but which has unbounded depth (so the stack itself never overflows). In your program you should limit yourself to accessing this with methods like `push(i)`, `top()`, `pop()`, and a predicate like `empty?()`. These will push a value into the stack, return the value stored in the top of the stack (the most recent value pushed), discard the top of stack, and test the stack for empty, respectively. Don't forget that you have no storage outside of the stack, so you need to work directly with the values in the stack—you can't pull a value out of the stack and assign it to some other variable.

   (a) Sketch an algorithm to recognize the language: $\{wcw^{\mathrm{R}} \mid w \in \{a,b\}^*\}$. This is the set of strings made up of any sequence of 'a's and 'b's followed by a 'c' and then exactly the same sequence of 'a's and 'b's in reverse order, so these are all *palindromes* over the alphabet $\{a,b,c\}$ in which 'c' occurs only as the middle symbol. It includes strings like:

$abbacabba \quad aca \quad abaabcbaaba \quad c$ (which, of course, equals $\varepsilon c \varepsilon$).

(Solution)

The idea here is to push symbols into the stack until encountering the 'c' and then pop them and match them to the remainder of the string. Since they come out of the stack in the opposite order of the way they went in, the second portion of the string will match iff it is the reverse of the first.

```
while (input() == '$a$' or input() == '$b$')
{  push(input()); next() }
endwhile;
if (input() == '$c$') then
{  next ();
   while (not empty?() and top() == input())
   {  pop(); next() }
   endwhile
   }
endif;
exit(empty?() and input == EOF)
```

(b) What is your intuition about recognizing this language within the second model (i.e., using just a single counter)?

(Solution)

While it is possible to uniquely represent each possible prefix w with a counter, by, for instance, letting the binary representation of the value of the counter represent the string (there are a number of complicating details, but they are not insurmountable), to make this work we would evidently need to be able to double the value of the counter in order to get from one prefix to the next. Note that we could do this using two counters, since we can use the second counter to double the value of the first. (How?)

(c) What is your intuition about the possibility of recognizing the language $\{wcw \mid w \in \{a, b\}^*\}$? This is the set of strings made up of any sequence of 'a's and 'b's followed by a 'c' and then exactly the same sequence of 'a's and 'b's in exactly the same order; it's referred to as the *(deterministic) copy language.*

(Solution)

Here we would need to access the bottom of the stack somehow—to access it in FIFO rather than LIFO order. Again, if we were to use two stacks we could do this, but we cannot do it with just a single stack.

6. Fourth and final model: Computer has a single FIFO queue of fixed precision unsigned integers with the length of the queue unbounded. You can use access methods similar to those in the third model. In this model you will have something like `front()` that will return the value in the front of the queue (the eldest item) rather than `top()`.

   (a) Sketch an algorithm to recognize the copy language ($\{wcw \mid w \in \{a, b\}^*\}$).

   (Solution)

   Following the intuition of the last part of the previous problem, we can just modify the previous algorithm to call `front()` instead of `top()`.

   (b) What is your intuition about the possibility of recognizing the palindrome language of the previous question ($\{wcw^{\mathrm{R}} \mid w \in \{a, b\}^*\}$)?

   (Solution)

   Here intuition fails. It turns out that one *can* recognize wcw^{R} using just a queue. Rather than explain how, we'll let you ponder this one. (Hint: it is possible to access a queue LIFO.)

4 Inductive Proof

7. Prove for all $w, v \in \Sigma^*$ for any alphabet Σ, that $(wv)^{\mathrm{R}} = v^{\mathrm{R}} \cdot w^{\mathrm{R}}$.

   (Solution)

   Proof: By induction on $|v|$.

   (Basis:)

   $(w\varepsilon)^{\mathrm{R}} = w^{\mathrm{R}} = \varepsilon w^{\mathrm{R}}$.

   (IH:)

   Let $v = u\sigma$, $\sigma \in \Sigma$, and assume, for induction, that the lemma is true for wu.

   (Induction:)

   Then

$$\begin{aligned}
(wu\sigma)^{\mathrm{R}} &= \sigma(wu)^{\mathrm{R}} \text{ by definition} \\
&= \sigma(u^{\mathrm{R}}w^{\mathrm{R}}) \text{ by IH} \\
&= (\sigma u^{\mathrm{R}})w^{\mathrm{R}} \text{ by associativity of concatenation} \\
&= (u\sigma)^{\mathrm{R}}w \text{ by definition} \\
&= v^{\mathrm{R}}w^{\mathrm{R}}.
\end{aligned}$$

⊣

8. Let T_2 be the set of all binary-branching trees, where

   - The trivial tree (consisting of a single node) is in T_2.
   - If t_1 and t_2 are trees in T_2, then the tree formed by taking t_1 as the left subtree, t_2 as the right subtree, and a new node as a root is also a tree in T_2.
   - Nothing else.

   For every tree $t \in T_2$ let $d(t)$ be the *depth* of t defined:

$$d(t) = \begin{cases} 0 & \text{if } t \text{ is trivial,} \\ 1 + \max(d(t_1), d(t_2)) & \text{if } t \text{ is constructed from } t_1 \text{ and } t_2. \end{cases}$$

The *leaves* in a tree $t \in T_2$ are its trivial subtrees—the nodes which have no descendents. Let $l(t)$ denote the number of leaves in the tree t. Prove for all $t \in T_2$ that $d(t) + 1 \leq l(t) \leq 2^{d(t)}$.

(Solution)

(Basis:)

Suppose t is trivial. Then it contains exactly one trivial subtree (t itself) and, by definition, $d(t) = 0$. Thus:

$$d(t) + 1 = 1 = l(t) \text{ and } 2^{d(t)} = 2^0 = 1 = l(t).$$

(IH:)

Suppose, for induction, that t is constructed from t_1 and t_2 and that the result is true for both t_1 and t_2.

(Induction:)

WLOG[1] assume $d(t_1) \geq d(t_2)$.
Then $d(t) = 1 + d(t_1)$, $l(t) = l(t_1) + l(t_2)$ and $l(t_1) + l(t_2) \geq l(t_1) + 1$ (since $l(t_2) \geq 1$). Thus

$$d(t) + 1 = 1 + d(t_1) + 1 \leq l(t_1) + 1 \text{ (by IH) } \leq l(t_1) + l(t_2) = l(t),$$

and

$$l(t) = l(t_1) + l(t_2) \leq 2^{d(t_1)} + 2^{d(t_2)} \leq 2^{d(t_1)} + 2^{d(t_1)} = 2^{1+d(t_1)} = 2^{d(t)}.$$

[1]Without Loss Of Generality