

# CSE105 (Winter 2014): Homework 3

Instructor: Daniele Micciancio

Due on Wednesday, February 5, 2014.

Solutions to this homework assignment should consist of both a writeup with the proofs for problems 2, 4 and 5, and a collection of JFLAP files and Haskell programs to be submitted using the class accounts. As a clarification, the definition of FST used in all problems is the one given in Homework 1, problem 2d, with transition function  $\delta : Q \times \Sigma \rightarrow Q \times \Gamma^*$ , rather than the simplified definition from the textbook. Refer to homework 1 solutions and the lecture notes on the class webpage for details.

**What to submit:** The writeup for problems 2 and 4 should be submitted in class on the due date. (This may be either hand written, or typed. If you type your solution, still print it and bring it to class. No electronic submission of pdf files is required.) Submit also your files using turnin. Files should be named exactly `HW3p1.rex`, `HW3p1.out`, `HW3p2a.jff`, `HW3p2b.jff`, `HW3p2c.jff`, `channel.jff`, `encode.jff`, `decode.jff`, `HW3p4.hs`, `HW3p5.hs`, `HW3p6a.jff`, `HW3p6b.jff`. Zip the files together and submit them using turnin as in previous assignments.

## 1 Extracting Valid Email Addresses using Regular Expressions

In this problem, you are going to be working with real data and the UNIX command line tool `egrep`. You can refer to this link for a quick tutorial [http://www.cs.columbia.edu/~tal/3261/fall107/handout/egrep\\_mini-tutorial.htm](http://www.cs.columbia.edu/~tal/3261/fall107/handout/egrep_mini-tutorial.htm) or read the manual page for `egrep` (`man egrep` from the command line) to see what format is used for regular expressions, and `egrep` command line options. `Egrep` works a line at a time and it matches at any point in the line, not necessarily at the beginning. That means, it acts as though the regular expression starts and ends with  $\Sigma^*$  to match any number of any character.

Your task is to design a regular expression that would identify valid email addresses from a document using `egrep`. You can refer the Wikipedia page [http://en.wikipedia.org/wiki/Email\\_address](http://en.wikipedia.org/wiki/Email_address) for the rules for valid email IDs. While creating the regex, keep the following in mind:

- The format of email addresses is `local-part@domain`
- The local-part can have upper case and lower-case letters (from the English alphabet), numbers and (some) special characters.
- Domain name can have letters, numbers, dots and hyphens.
- Domain name or local-part cannot start/end with a special symbol.
- Hierarchical domain names are allowed (i.e. `emailID@xyz.abc.sample.com` is allowed)
- Neither local-part nor domain name can have two consecutive dots.
- The special characters to be considered for this assignment are `{., -, %, +, -}`
- Comments and international characters are not allowed.

|                  | Valid Email Addresses                | Invalid Email Addresses           |
|------------------|--------------------------------------|-----------------------------------|
| <b>Examples:</b> | <code>smp@example.com</code>         | <code>.smp@exmaple.com</code>     |
|                  | <code>drt@example-1.com</code>       | <code>smp..drt@example.com</code> |
|                  | <code>drt.smp@example.exp.com</code> | <code>smp@drt@example.com</code>  |

You can test your solution by finding out the number of valid email addresses in the file `sample-email.txt`, available on the course webpage. If your expression is correct you should get 9 valid ids. Finally, save your

regular expression in a file named `HW3p1.rex`, and use it to extract all valid email addresses (according to the simplified rules described above) from the wikipedia page `http://en.wikipedia.org/wiki/Email_address`. (You can download the page from the command line with the command `wget http://en.wikipedia.org/wiki/Email_address -O wikipedia-email.html`. Alternatively, just download the page with a web browser and save the page source to a file `wikipedia-email.html`.) Your `HW3p1.rex` file should contain just the regular expression, in `egrep` format, so that you can search for the strings matching your pattern using the command `egrep -o -f HW3p1.rex wikipedia-email.html`. (Here the “-o” option is used to print out only the emails, rather than the whole lines containing them.) Save the result of the search in a file `HW3p1.out`, and submit it together with `HW3p1.rex` as part of your solutions.

## 2 Proofs by Reduction

For each of the following languages **(a)**, **(b)**, **(c)**, state whether it is a regular or nonregular language, and prove your claim by giving an FST reduction between the language and one of the following two languages which were already proved (or claimed) to be regular/nonregular in class:

- $L_1 = \{0^n 1^n | n \geq 1\}$  which is nonregular.
- $L_2 = \{w \in \{0, 1\}^* | w \text{ is a multiple of 3 in base 2}\}$  which is regular.

Refer to the notes posted on the class website for more information on FST reductions: <http://cseweb.ucsd.edu/classes/wi14/cse105-a/LecFST2.pdf>. For this problem, submit the write up of your proof (in class), and also submit your FSTs implementing the reduction drawn in JFLAP as saved as files `HW3p2a.jff`, `HW3p2b.jff` and `HW3p2c.jff`. The written portion of your solutions should contain three separate sections, clearly labeled as corresponding to the three languages below. Each section should contain a statement of what is being proved (is the language regular or nonregular?), which languages are used in the reduction, the direction of the reduction (written using the  $A \leq_{FST} B$  notation), a mathematical description of the function computed by the reduction, the diagram of an FST computing the function, and a brief proof explaining why the reduction is correct. (Remember: a function  $f$  is a valid reduction from  $A$  to  $B$  if, for all input  $x$ , we have that  $x \in A \implies f(x) \in B$  and  $x \notin A \implies f(x) \notin B$ .)

- (a)**  $L = \{w | w \text{ has equal number of 1s and 0s}\}.$
- (b)**  $L = \{ww | w \in \Sigma^*\}.$
- (c)**  $L = \{w \in \{0, 1, 2, 3\}^* | w \text{ is multiple of 3 in base 4}\}.$

## 3 Error Correcting Codes

Read the lecture notes on Non-deterministic Finite State Transducers (NFST) posted on the course webpage. In this problem we explore the use of NFST to model (a toy version of) a realistic application: the design of an error correcting code (and corresponding decoding scheme) to reliably transmit messages over a noisy channel.

Consider the following setting: two parties, a sender  $S$  and a receiver  $R$ , want to exchange messages over a communication channel that allows to transmit sequences of bits. The communication channel is not perfectly reliable: for each transmitted bit  $b \in \{0, 1\}$ , the channel may deliver either the correct bit  $b$ , or corrupt it and deliver  $1 - b$  instead. However, transmission errors are not too frequent: it may be assumed that whenever an error occurs (i.e., a bit gets flipped by the channel), the next two bits will be transmitted without errors. The sender and the receiver can use this communication channel to achieve errorless communication as follows:

- the sender transmits a redundant encoding of each input bit  $b \in \{0, 1\}$ , e.g., obtained by replicating each input bit 3 times, and sending  $bbb$  instead.

- at the other end of the communication channel, the receiver performs error correction and decoding by taking the majority of each group of three received bits  $b_1b_2b_3 \in \{0,1\}^3$ . E.g., when receiving 010, the receiver determines that 000 was sent, and therefore the original input bit was 0.

The communication channel can be modeled by an NFST that flips some bits nondeterministically subject to the constraints above. The encoding and decoding algorithms are actual programs that you may want to run. So, they should be modeled as *deterministic* FSTs that for every input string, perform exactly one computation, and produce a single output string. You can think of FSTs as a special case of NFST with the property that for any state  $q$  and input symbol  $a$  the transition function  $\delta(q, a) = \{(p, w)\}$  outputs a set containing precisely one pair (consisting of a new state  $p$  and output string  $w$ .)

You can use jflap to draw your FSTs and NFSTs. In jflap, NFST are called “Mealy machines”. Jflap allows to run only NFST that are deterministic. So, for debugging purposes, you will be able to run only the machines corresponding to the encoding and decoding algorithms. You can still use JFLAP to draw, save and print the NFST representing the communication channel, but trying to run it within jflap will result in an error message.

(a) Use an NFST to model the communication channel. More specifically, give an NFST  $C$  that on input a string  $w \in \{0,1\}^n$  outputs the set of all possible strings produced by the channel when  $w$  is transmitted. As a reminder, this is the set of all strings  $u \in \{0,1\}^n$  of length  $n$  such that, for all  $i = 1, \dots, n-2$ , if  $w[i] \neq u[i]$  (i.e., if the  $i$ th bit gets flipped) then  $w[i+1] = u[i+1]$  and  $w[i+2] = u[i+2]$  (i.e., the following two bits are transmitted without errors). [Hint: you can solve this part using an NFST  $C$  with just 3 states.] Save your NFST as file `channel.jff`.

(b) Give a FST modeling the encoding algorithm. More specifically, give an FST  $E$  that on input  $w \in \{0,1\}^n$  outputs a string of length  $3n$  obtained by repeating each input bit 3 times. Notice: this time your FST should be deterministic, according to the definition in Homework 1 problem 2(d). Save the FST in a file `encode.jff`. [Hint: you can solve this part using an FST  $E$  with just 1 state!]

(c) Give a FST corresponding to the decoding algorithm  $D$  that corrects the errors introduced by the channel and recovers the original message  $w$ , and save it in a file `decode.jff`. More specifically,  $D$  should be such that if you compose the functions computed by the three (N)FST together, you obtain function such that  $D \circ C \circ E(w) = \{w\}$  for every  $w \in \{0,1\}^n$ . Hint: For the decoder I think 5 states should be enough.

(d) Use the closure of NSFT under composition (see lecture notes) to combine the NSFT from parts (a), (b) and (c) into an NFST for  $D \circ C \circ E$  and verify that that it indeed computes the identity function, i.e., on input  $w$ , it outputs  $\{w\}$ . Notice, you can verify that an NFST computes the identity function simply by checking that all of its transitions (or at least those reachable from the start state) are labeled either with 0/0 or with 1/1.

You can compose the NFST in either order  $(D \circ C) \circ E$  or  $D \circ (C \circ E)$ , but one of the two ordering may give a simpler solution. You should perform the transformation by hand following the definition of composition of NFST in the lecture notes. As usual, when composing the automata, you may start from the start state, and draw only the states that are reachable from it.

As an aid to debug your encoding and decoding FST we wrote a haskell program that reads the three NFST from the files `encode.jff`, `channel.jff`, and `decode.jff`, composes them, and outputs the resulting NFST in a file `nfst.jff` which you can open in JFLAP for inspection. The NFST will likely contain many useless states, which cannot be reached from the start state of the NFST, and you may delete without affecting the behavior of the NFST. Simplify the NFST by removing these useless states, and check that the resulting NFST computes the identity function.

**What to submit:** The solution to this problem should consists of 3 JFLAP files `encode.jff`, `channel.jff`, and `decode.jff`. Part (d) is intended primarily as a way for you to check your solution, and you don’t need to submit anything for it.

## 4 Composing FST

In the lecture notes we proved that NFST are closed under function composition, and we also provided a haskell program implementing the composition operation. In this problem you are asked to prove that the same is true for deterministic FST.

Prove that for any two FST  $T_1, T_2$ , computing functions  $f_{T_1}$  and  $f_{T_2}$ , the function composition  $(f_{T_2} \circ f_{T_1})(x) = f_{T_2}(f_{T_1}(x))$  is also computable by an FST. Prove the statement by giving a transformation that combines any two FST  $T_1$  and  $T_2$  (with compatible alphabets) into a new FST  $T$  that computes  $f_T = f_{T_2} \circ f_{T_1}$ . (Notice the order of composition:  $T$  should correspond to first running  $T_1$  and then passing the result to  $T_2$ .) Implement the transformation as a haskell program following the guidelines on the course webpage, and save it in a file `HW3p4.hs`. *[Hint: you may use the proof/construction for NFST in `FST.hs` as a model, but the solution for FST is somehow simpler.]*

As part of your solutions include both a writeup of your proof of this closure property using mathematical notion, as well as the file `HW3p4.hs` containing your haskell program. (See guidelines to get a template for this file.)

## 5 Composing an FST with a DFA

In this last problem we revisit Problem 1 from the midterm. Prove that if  $A \leq_{FST} B$  and  $B$  is regular, then  $A$  is also regular. You should prove this statement by giving a construction that on input an FST computing the reduction, and a DFA for  $B$ , outputs a DFA for  $A$ . Implement the transformation as a haskell program following the guidelines on the course webpage, and save your solution as file `HW3p5.hs`. (See guidelines to get a template for this file.)

As part of this problem you should submit both a written mathematical proof, and the file containing your implementation of the construction. The written portion should start with a clear statement of what is being proved, using the definition of reduction, etc., and be easily readable independently of the haskell implementation.

## 6 From automata to regular expressions

Do exercise 1.21 from the textbook. You may use Jflap as a study tool, but then perform the conversion from automata to regular expressions (using GNFA) by hand. Use JFLAP just to input the two resulting regular expressions, and save them as files `HW3p6a.jff` and `HW3p6b.jff`. Submit these two files as your solution.

(After converting the automata to regular expressions by hand, and writing them in jflap, for debugging purposes you can use jflap to convert your regular expressions back to automata, and run them on test inputs of your choice.)