CS235 Languages and Automata
Prof. Lyn Turbak
Wellesley College

Handout # 12
October 12, 2012

Problem Set 6
Due: Thursday, October 18

Reading: Sipser, Sections 1.3; Stoughton, Sections 3.1 – 3.4, 3.11;
(optional) Kozen Ch. 3–9, 11, 13–14

Submission:

You should turn in a hardcopy submission packet by slipping it under Lyn's office door. This packet should include: (1) your written solutions for each problem and (2) your final version of the `ps6` directory, including the files `ps6.sml`, `L1R.fa`, and `L2.dfa`. (3) your transcripts of the requested test cases.

You should also submit a softcopy (consisting of your final `ps6` directory) to the drop directory `~cs235/drop/ps6/`*username*, where *username* is your username. To do this, execute the following commands in Linux:

```
cd /students/username/cs235
cp -R ps6 ~cs235/drop/ps6/username/
```

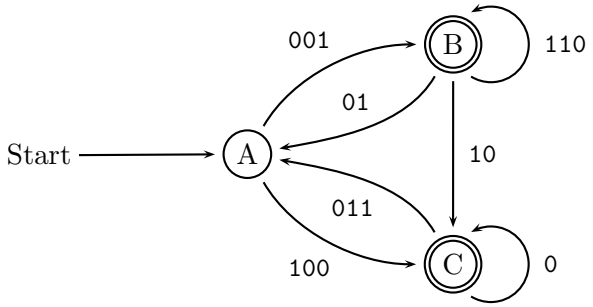
Note: All programming on this assignment uses the Forlan modules and so must be done in a version of SML with the Forlan toolset loaded. Additionally, it is assumed that you have loaded the file `~/cs235/ps6/ps6.sml` via `use` so that you can use the helper functions provided in this file.

Note: There are a total of 130 points on this assignment. There are also two optional extra credit problems that are worth a total of 45 points.

Problem 1 [25]: FA Reversal

a [10] Suppose that language L is accepted by the the finite automaton $FA_L = (Q_L, \Sigma_L, T_L, s_L, F_L)$. Define a finite automaton $FA_R = (Q_R, \Sigma_R, T_R, s_R, F_R)$ that accepts the reversed language L^R. Your definition should be in a form similar to that of the EFA to NFA definition in slide 15-10 and the NFA to DFA definition in slide 15-16. (These are slides from Wed. Oct. 10 Lecture #15 *Transforming Finite Automata: Nondeterministic and Deterministic Finite Automata Are Equivalent.*)

b [5] Using your approach from part **b**, draw an FA that accepts the reversal of the language defined by the following FA:



c [10] Use Forlan to test that your FA from part **b** accepts the reversal of the language of the pictured FA on all binary strings of up to length 12. The `ps6` folder contains a file `L1.fa` corresponding to the FA pictured above. For this problem, you should do the following:

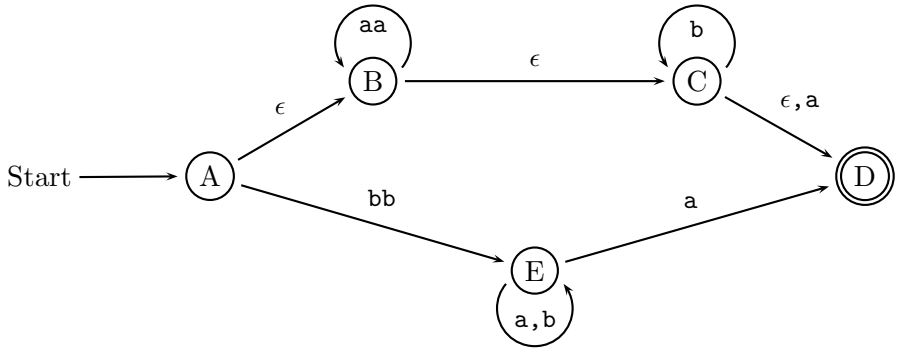
1. Define a file `L1R.fa` for the reversed FA.
2. In the file `ps6.sml`, define a testing function `testL1R` of type `int -> bool` such that `testL1R` *n* compares the FAs in `L1.fa` and `L1R.fa` on all binary strings of up to length *n*. This function should return `true` iff `L1R.fa` accepts only the reversal of all strings accepted by `L1.fa`, and otherwise should return `false`. In the `false` case, `testL1R` should also print a list of each counterexample strings s, indicating whether s is accepted by `L1.fa` and the reversal of s is accepted by `L1R.fa`.

   `ps6.sml` provides the following helper function that you may find useful in this problem:

   ```
   fun revString s = String.implode (List.rev (String.explode s))
   ```

3. Show that `testL1R 12` returns `true` (submit a transcript for this).

Problem 2 [20]: FA to DFA Conversion In this problem, you will convert the following finite automaton to a DFA using the steps presented in the Lecture #15 slides (handed out Wed. Oct 5).



The file `L2.fa` contains a description of this FA in Forlan syntax.

a [3]: FA to EFA Using the FA to EFA conversion algorithm presented in ths slides, draw an EFA that is equivalent to the above FA.

b [5]: EFA to NFA Using the EFA to NFA conversion algorithm presented in the slides, draw an NFA that is equivalent to the EFA from part **a**.

c [10]: NFA to DFA Using the NFA to DFA conversion algorithm presented in the slides (i.e., the subset construction), draw a DFA that is equivalent to the NFA from part **b**.

d [2]: Testing Using JFA, created a file named `L2.dfa` that expresses your solution to part **c** in Forlan DFA format. Test that this file is correct by running the following testing function, which is defined in `ps6.sml`:

```
fun test2 () =
  let val L2_fa = FA.input "L2.fa"
      val L2_dfa = DFA.input "L2.dfa"
   in DFA.relationship(faToDFA L2_fa, L2_dfa)
  end
```

If `L2.dfa` is correct, then executing `test2()` in Forlan will have the following behavior:

```
- test2();
languages are equal
val it = () : unit
```

But if `L2.dfa` is incorrect, then executing `test2()` in Forlan will give counterexamples explaining why the languages are not equal. These counterexamples will help you debug your DFA.

Notes:

- Although the testing function `test2` uses `faToDFA`, *you* should not use any of `faToEFA`, `efaToNFA`, `nfaToDFA`, or `faToDFA` to construct your automata. Instead, you should construct them by hand.
- In the file `L2.dfa`, you should write each state as an angle-bracket-delimited comma-separated sequence of state symbols, which Forlan treats as a single symbol. For example, the state corresponding to the set {`C`,`E`} would be written as the Forlan symbol `<C,E>`. The current version of Forlan does *not* allow braces between the angle brackets.
- In the file `L2.dfa`, there *must* be a semicolon between every two transitions but there *must not* be a semicolon after the last transition. If you put a semicolon after the last transition, reading the file will give the error `end-of-file unexpected`. If you neglect a semicolon after any transition but the last, reading the file will give the error *`Q`* `unexpected`, where *`Q`* is the first symbol in the next transition.
- The reader for `L2.dfa` verifies that it is a valid DFA. If you neglect to include a missing edge from state *`state`* with label *`label`*, the DFA reader will report the error `no transitions for state/input symbol pair : "`*`state, label`*`"`.

Problem 3 [40]: Regular Expressions

Consider the following languages over over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^*$.

$L_{ab\_or\_bc}$: All strings that contain `ab` or `bc`.

$L_{ab\_and\_bc}$: All strings that contain `ab` and `bc`.

$L_{cs\_at\_odd\_positions}$: All strings in which every odd position (0-indexed) is a `c`.
(This language contains the empty string.)

$L_{even\_as\_or\_odd\_bs}$: All strings that contain an even number of `a`s or an odd number of `b`s.

$L_{geq\_2\_bs\_and\_leq\_1\_c}$: All strings containing at least 2 `b`s and at most 1 `c`.

For each language L_{name} specified above, do the following:

1. In the file `ps6.sml`, define an SML string named *`name`*`_reg` that is a regular expression string that specifies the language L_{name}. E.g., the string `"a* + (bc)*"` specifies a language containing all strings that are sequences of `a`s or sequences of `bc`s.
2. In `ps6.sml`, define an SML predicate named *`name`*`_pred` that returns `true` for every string in L_{name} and `false` for every string not in L_{name}. Strive to make your predicates as simple and concise as possible. (Review similar predicates from the solutions to previous assignments!)
3. In `ps6.sml`, there is a testing function named `test_`*`name`* that, given a natural number n, will generate all strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ with length $\leq n$ and for each such string will compare its membership in the languages defined by the associated regular expression and the predicate. You should use `test_`*`name`* to debug your regular expression until it works correctly on all strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^*$ of up to length 8. In your submission you should indicate whether your regular passes this test; if not, you should include a transcript showing all counterexamples on which it fails.

Notes:

- `ps6.sml` contains a testing function named `test3_all` that, given a natural number n, will run all five testing functions on strings of up to length n.
- Although you will use Forlan to test your regular expressions in this problem, you should *not* use Forlan to generate the regular expressions from FAs for the languages. Instead, you should develop the regular expressions for each language "by hand".

Problem 4 [20]: Converting a Regular Expression to a Finite Automaton

a [10] Using the rules from Lecture #17 (whose slides will be handed out on Mon. Oct. 15), draw the finite automaton that can be constructed from the regular expression `1(01* + ($1)*)*1`. Follow the rules carefully; in the past, students have tended to be rather sloppy, leaving out many states. A correct solution will have *lots* of states, some of which are unreachable from the start state.

b [3] Using JFA, created a file named `L4a.fa` that expresses your solution to part **a** in Forlan FA format. Test that this file is correct by running the following testing function, which is defined in `ps6.sml`:

```
fun test4a () =
  let val L4a_fa = FA.input "L4a.fa"
      val L4a_reg = Reg.fromString "1(01* + ($1)*)*1"
      val faToDFA = nfaToDFA o efaToNFA o faToEFA
      val regToDFA = faToDFA o regToFA
  in DFA.relationship(faToDFA L4a_fa, regToDFA L4a_reg)
  end
```

If `L4a.fa` is correct, then executing `test4a()` in Forlan will have the following behavior:

```
- test4a();
languages are equal
val it = () : unit
```

But if `L4a.fa` is incorrect, then executing `test4a()` in Forlan will give counterexamples explaining why the languages are not equal. These counterexamples will help you debug your FA.

c [5] Use the three simplification rules for finite automata from Lecture #17 (Mon. Oct. 15) to simplify your finite automaton from part **a**. Explicitly justify each of your simplification steps by one of the rules. Do *not* make any simplifications that aren't justified by one of the three simplification rules.

d [2] Using JFA, created a file named `L4c.fa` that expresses your solution to part **c** in Forlan FA format. Similar to part **b**, test that this file is correct by running the following testing function, which is defined in `ps6.sml`:

```
fun test4c () =
  let val L4c_fa = FA.input "L4c.fa"
      val L4c_reg = Reg.fromString "1(01* + ($1)*)*1"
      val faToDFA = nfaToDFA o efaToNFA o faToEFA
      val regToDFA = faToDFA o regToFA
  in DFA.relationship(faToDFA L4c_fa, regToDFA L4c_reg)
  end
```

Problem 5 [25]: Converting a Finite Automaton to a Regular Expression

Consider the finite automaton FA_5 in Fig. **??**. Using Sipser's state-ripping algorithm presented in the slides for Lecture #17 (Mon. Oct. 15), derive a regular expression R_5 that denotes the language L_5 accepted by this finite automaton.

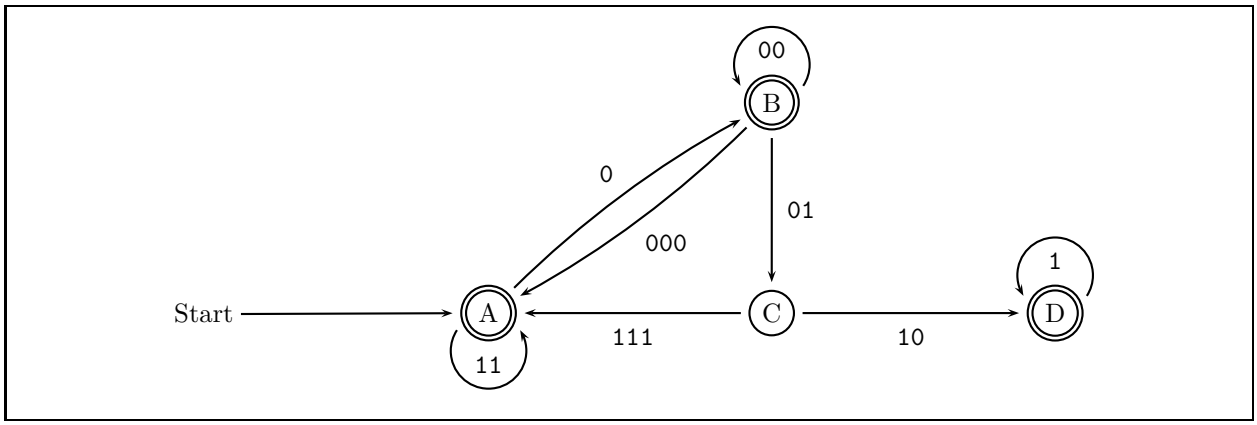


Figure 1: The finite automaton FA_5.

Notes:

- At each state-ripping step, you should rip out the state that is the middle state of the fewest transition pairs. Use this rationale to explicitly justify how you choose the next state to rip.
- To receive full credit, you must show the GNFA that results after each state-ripping step. It's also a good idea to explicitly show the work that you do to replace each transition pair by a single transition.
- You must verify that your answer R_5 is correct by defining in `ps6.sml` a variable `R5_string` to be your answer expressed as an SML string and using the testing function provided for this problem:

```
fun test5() =
  let val L5_fa = FA.input "L5.fa"
      val L5_reg = Reg.fromString R5_string
      val faToDFA = nfaToDFA o efaToNFA o faToEFA
      val regToDFA = faToDFA o regToFA
   in DFA.relationship(faToDFA L5_fa, regToDFA L5_reg)
  end
```

  Evaluating `test5()` in Forlan will compare the language denoted by `R5String` to the language accepted by the finite automaton for this problem (which has already been specified in the file `L5.fa`). If the two are equal, the result is:

```
- test5();
languages are equal
val it = () : unit
```

  But if the two are different, `test5()` will give an example of a string in one language that is not in the other. This will help you debug your regular expression.

Extra Credit Problems

Extra Credit 1 [25]: More FA-to-Regular-Expression Conversion

This problem is optional. You should only attempt it after completing the rest of the problems.

Give regular expressions for the following languages:

$L_{ab\_minus\_bc}$: All strings that contain `ab` but not `bc`.

$L_{neither\_ab\_nor\_bc}$: All strings that contain neither `ab` nor `bc`.

You may solve this problem by defining FAs for these languages (see PS5 Problem 1) and then *manually* applying the state-ripping algorithm (do not use Forlan for this step). For full credit, you must (1) show the details of the state-ripping algorithm and (2) use Forlan to verify that your solution is correct (as in Problem 3 of this problem set).

Extra Credit 2 [20]: Regular Expression Equivalence

Consider the following two regular expressions, as well as the SML functions in Fig. **??**.

R_a = `0 + 0(0(0 + 100)* + (0 + 010)*010)`

R_b = `(0*)*(%* + 0)*(0010*)*0`

```
(* Return a string representation of the result of using the strong simplification
   process to simplify the regular expression denoted by the given string. *)
val testSimplify str =
  Reg.toString (Reg.simplify Reg.weakSubset (Reg.fromString str))

(* Convert an FA to a DFA *)
val faToDFA = nfaToDFA o efaToNFA o faToEFA

(* Convert a regular expression to a DFA *)
val regToDFA = faToDFA o regToFA

(* Determine the relationship of the languages defined by two regular expressions *)
fun relationshipReg(reg1, reg2) = DFA.relationship(regToDFA reg1, regToDFA reg2)

(* Determine the relationship of the languages defined by two regular expressions,
   expressed as strings *)
fun relationshipRegString(regString1, regString2) =
  relationshipReg(Reg.fromString regString1, Reg.fromString regString2)
```

Figure 2: SML functions involving the simplification and comparison of regular expressions.

Using the `testSimplify` function, we can see that Stoughton's strong simplifier simplifies R_a to itself and simplifies R_b to an expression that does not look anything like R_a:

```
- testSimplify("0 + 0(0(0 + 100)* + (0 + 010)*010)");
val it = "0 + 0(0(0 + 100)* + (0 + 010)*010)" : string

- testSimplify("(0*)*(%* + 0)*(0010*)*0");
val it = "(0 + 001)*0" : string
```

Nevertheless, we can use the `relationshipRegString` function to show that R_a and R_b denote the same regular language:

```
- relationshipRegString("0 + 0(0(0 + 100)* + (0 + 010)*010)", "(0*)*(%* + 0)*(0010*)*0");
languages are equal
val it = () : unit
```

An alternative way to show that R_a and R_b denote the same regular language is to use the equivalence rules from Slide 16-14 to algebraically convert R_a to R_b — i.e., to show that there is a sequence of equivalences

$$R_1 \approx R_2 \approx \cdots \approx R_{n-1} \approx R_n$$

where $R_1 = R_a$ and $R_n = R_b$ and each $\approx$ step is justified by a rule from Slide 17-14.

In this problem, follow this alternative approach to show that R_a and R_b are equivalent. Justify each step of your equivalence by citing the rule used at each step.

Hints:

- It's a good idea to make `(0 + 001)*0` one of your intermediate regular expressions.
- You do not need to use rules 15, 22, or 23.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS235 Problem Set 6
Due Thursday, October 18, 2012

Name:

Date & Time Submitted:

Collaborators (*anyone you worked with on the problem set*):

By signing below, I attest that I have followed the collaboration policy specified in the slides on the first day of class.
Signature:

In the **Time** *column, please estimate the time you spend on the parts of this problem set. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the* **Score** *column when grading your problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [25]		
Problem 2 [20]		
Problem 3 [40]		
Problem 4 [20]		
Problem 5 [25]		
Extra Credit 1 [25]		
Extra Credit 2 [20]		
Total		