CS235 Languages and Automata **Handout # 6**
Prof. Lyn Turbak **October 5, 2007**
Wellesley College ***Revised October 10, 2007***

Problem Set 3

Due: 11:59 pm on Monday, October 15

Bug Fixes (October 10): (1) In the Problem 6 notes on writing a DFA state in Forlan, the Forlan symbol should ***not*** *include braces, which are not allowed between angle brackets in the current version of Forlan. So the set* $\{C, E\}$ *should be written as the Forlan symbol* `<C,E>`*, not as* $\{$`<C,E>`$\}$*. (2) The Appendix was missing the definition of* `relationshipRegString`*, which has now been added.*

This is the final version of PS3 containing all 8 problems. Problems 1 – 7 and most of Problem 8 can be done based on material covered through Lecture #15. Problem 8 requires some material covered in Lecture #16.

Required Reading: Stoughton, Sections 3.2, 3.3, 3.5, 3.8 – 3.12

Strongly Suggested Reading: Sipser, Sections 1.1 – 1.3 (46 pages but *really* worth it!)

Optional Reading: Stoughton, Sections 3.4, 3.6, 3.7; Kozen, Sections 3–9, 13–14

Submission:

You should turn in a hardcopy submission packet by slipping it under Lyn's office door by 11:59pm on the due date. This packet should include: (1) your written solutions for each problem. (2) your final version of the `ps3` directory, including the files `ps3.sml`, `L6.dfa`, `L7.dfa`, `hamming.dfa`, and `nonhamming.dfa`. (3) your transcripts of the requested test cases.

You should also submit a softcopy (consisting of your final `ps3` directory) to the drop directory `~cs235/drop/ps3/`*username*, where *username* is your username. To do this, execute the following commands in Linux:

```
cd /students/username/cs235
cp -R ps3 ~cs235/drop/ps3/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/ps3/ps3.sml` via `use` so that you can use the helper functions provided in this file.

Problem 1: Language Proofs Recall the following definitions and facts involving concatenation, powers, and reversals of strings and languages:

Concatenation of languages: $L_1@L_2$ (also written L_1L_2) $= \{x@y \mid x \in L_1 \text{ and } y \in L_2\}$

Raising a language to a power:

$L^0 = \{\%\}$, for all $L \in$ **Lan**;
$L^n = L@(L^{n-1})$, for all $L \in$ **Lan** and $n \in Nat$ s.t. $n \geq 1$;

String Reversal:

$\%^R = \%$;
$(a@x)^R = x^R@a$, for all $a \in$ **Sym** and $x \in$ **Str**;

Language Reversal: L^R stands for $\{x^R \mid x \in L\}$

Reversal of Concatenated Strings: For all $x, y \in$ **Str**, $(x@y)^R = y^R@x^R$.

a. Prove that for all $L_1, L_2 \in \mathbf{Lan}, (L_1@L_2)^R = (L_2^R)@(L_1^R)$. Give an explicit justification for each step in your proof.

b. Give an inductive proof that $L^n = (L^{n-1})@L$ for all $L \in \mathbf{Lan}$ and $n \in Nat$ s.t. $n \geq 1$. Give an explicit justification for each step in your proof.

c. Give an inductive proof that $(L^n)^R = (L^R)^n$. Give an explicit justification for each step in your proof. You may use the result of parts **a** and **b** in your proof.

Problem 2: Regular Expression Equivalence

Consider the following two regular expressions:

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

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

Using the `testSimplify` function described in the appendix to this assignment, 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 from the appendix 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 10-4 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 10-4.

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:

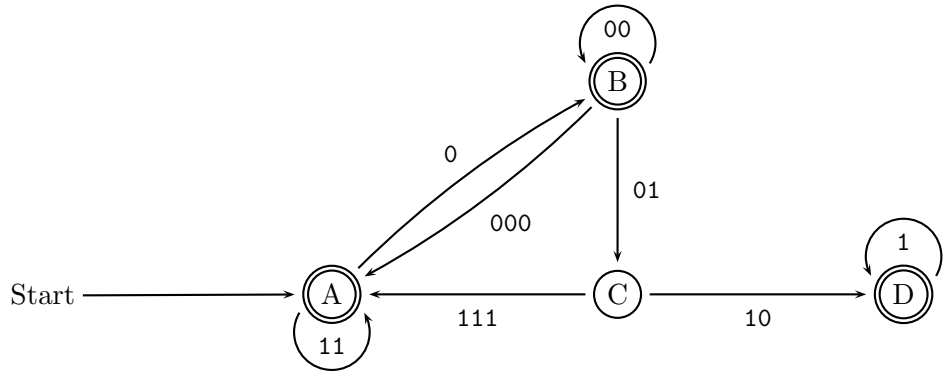
- It's a good idea to make `(0 + 001)*0` one of your intermediate regular expressions.
- You do not need to use rules 15, 22, or 23.

Problem 3: Converting a Regular Expression to a Finite Automaton

a. Using the rules from Lectures 13/14, draw the finite automaton that can be constructed the regular expression `1(01* + ($b)*)*1`. You do *not* have to label the states.

b. Use the three simplification rules for finite automata from Lectures 13/14 to simplify your finite automaton from part **a**. Do *not* make any simplifications that aren't justified by one of the three simplification rules.

Problem 4: Converting a Finite Automaton to a Regular Expression



Consider the finite automaton pictured above. Using Sipser's state-ripping algorithm presented in Lecture 14, derive a regular expression R_4 that denotes the language L_4 accepted by this finite automaton.

Notes:

- At each state-ripping step, you should rip out the state that is the middle state of the fewest transition pairs.
- 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_4 is correct by defining in `ps3.sml` a variable `R4String` to be your answer expressed as an SML string and using the testing function provided for this problem:

```
fun test4() =
  let val L4_fa = FA.input "L4.fa"
      val L4_reg = Reg.fromString R4String
   in DFA.relationship(faToDFA L4_fa, regToDFA L4_reg)
  end
```

  Evaluating `test4()` in Forlan will compare the language denoted by `R4String` to the language accepted by the finite automaton for this problem (which has already been specified in the file `L4.fa`). If the two are equal, the result is:

```
- test4();
languages are equal
val it = () : unit
```

  But if the two are different, `test4()` will give an example of a string in one language that is not in the other. This will help you debug your regular expression.

Problem 5: Classifying Languages

Below are descriptions of various languages over the alphabet {a, b}. Organize these languages into a subset partial order in which L is connected by an edge to L' above it iff $L \subset L'$. If two descriptions describe the same language, they should be placed together in a node.

L_1 = all strings containing an even number of a's and any number of b's.

L_2 = all strings containing an odd number of b's and any number of a's.

$L_3 = L_1 \cup L_2$.

$L_4 = L_1 \cap L_2$.

L_5 = all strings that contain only a's.

L_6 = the language of the regular expression `% + a*a`.

L_7 = all strings over the alphabet {a, b}.

L_8 = the language of the regular expression `$`.

L_9 = the language of the regular expression `$*`.

L_{10} = the language of the regular expression `%`.

L_{11} = the language of the regular expression `%*`.

L_{12} = the language of the regular expression `b*`.

L_{13} = the language of the regular expression `(a + b)*`.

L_{14} = the language of the regular expression `$(aba + bab)`.

L_{15} = the language of the regular expression `b(ab)*`

L_{16} = the language of the regular expression `b*(ab*ab*)*`.

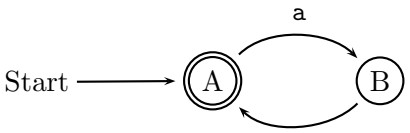
L_{17} = the language of the regular expression `a*b(a*ba*b)*`.

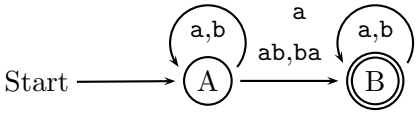
L_{18} = the language of the regular expression `a(a+b)*`.

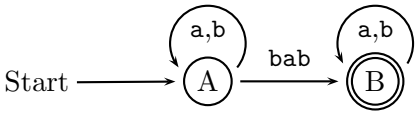
L_{19} = the language of the regular expression `(aa)*b(bb)*`.

L_{20} = all strings that contain the substring `bab`.

L_{22} = the language of the regular expression `(a+b)*ab`.

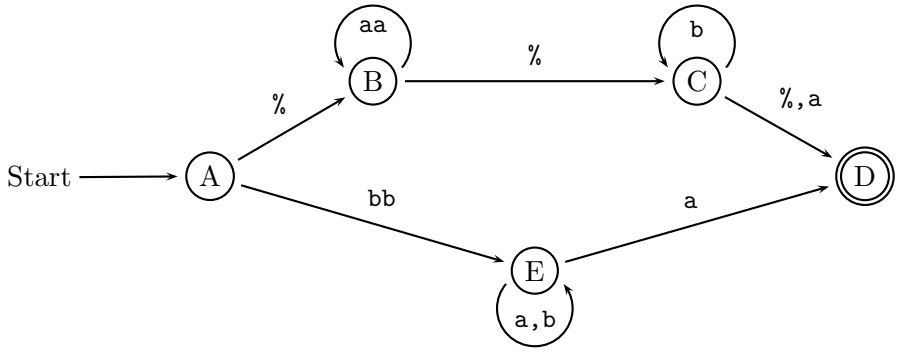
L_{23} = the language of the FA 

L_{24} = the language of the FA 

L_{25} = the language of the FA 

Problem 6: Converting an FA to a DFA

In this problem, you will convert the following finite automaton to a DFA using the steps presented in Lecture #15.



The file `L6.fa` contains a description of this FA in Forlan syntax.

a. : FA to EFA Using the FA to EFA conversion algorithm presented in class, draw an EFA that is equivalent to the above FA.

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

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

d. : Testing Express the solution to part **c** in Forlan FA format in a new file `L6.dfa`. Test that this file is correct by running the following testing function, which is defined in `ps3.sml`:

```
fun test6 () =
  let val L6_fa = FA.input "L6.fa"
      val L6_dfa = DFA.input "L6.dfa"
   in DFA.relationship(faToDFA L6_fa, L6_dfa)
  end
```

If `L6.dfa` is correct, then executing `test6()` in Forlan will have the following behavior:

```
- test6();
languages are equal
val it = () : unit
```

But if `L6.dfa` is incorrect, then executing `test6()` 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 `test6` 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 `L6.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 `L6.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 `L6.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"`.

- The JFA application is helpful for drawing and displaying finite automata and for reading/writing out files in Forlan FA format. It can be invoked on our Linux machines via

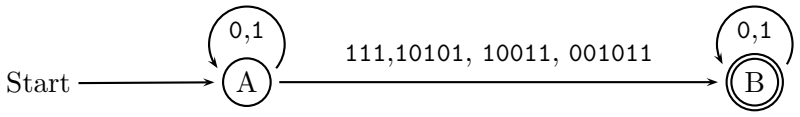
  ```
  jfa &
  ```

  For documentation on JFA, see

  ```
  http://people.cis.ksu.edu/~stough/forlan/jfa/
  ```

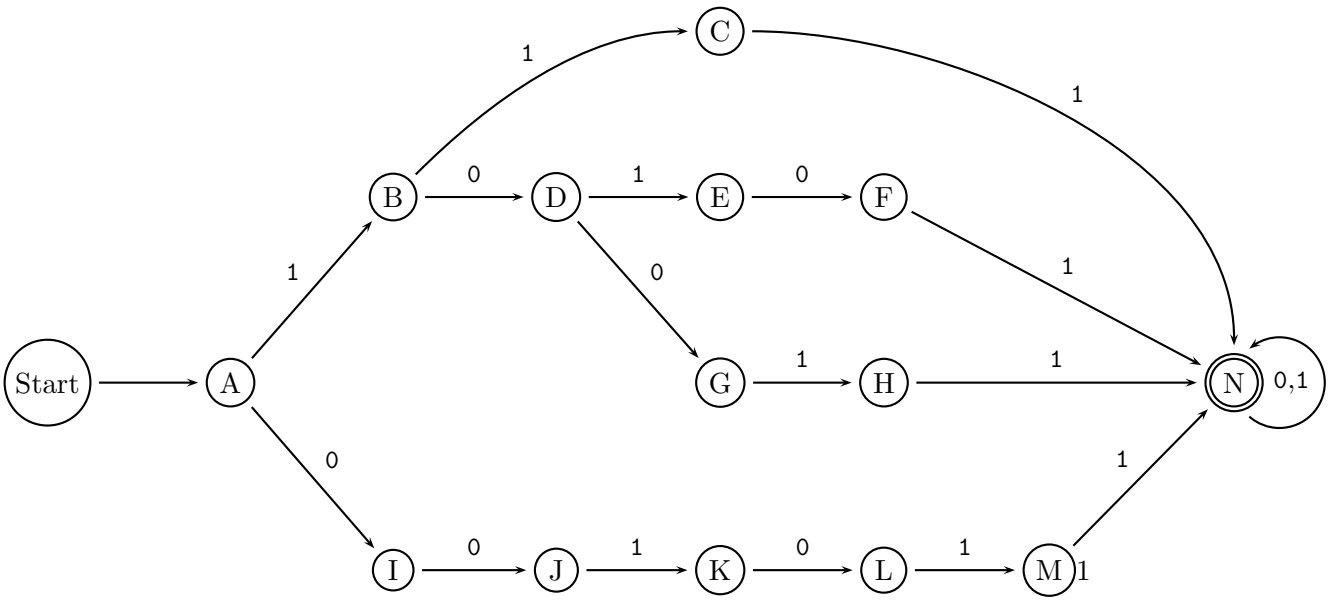
Problem 7: Completing a DFA

Consider the language L_7 consisting of all strings of `0`s and `1`s containing at least one of the following substrings: `111`, `10101`, `10011`, `001011`. It is easy to define an FA that accepts L_7:



This FA is defined in the file `L7.fa` in the `ps3` directory.

It is also easy to *partially* define a DFA that accepts L_7:



(In Stoughton's terminology, the partially defined DFA above is not a valid DFA but it is a valid NFA, EFA, and FA.) This partially defined DFA is specified in the file `L7.dfa` in the `ps3` directory.

The goal of this problem is to complete the DFA in the above diagram and in the file `L7.dfa` by adding the correct missing edges from each state so that the DFA accepts L_7. Recall that in a DFA accepting strings of `0`s and `1`s, each state must have exactly one outgoing edge labeled `0` and one outgoing edge labeled `1`. The challenge here is to figure out the target of each missing edge.

Notes:

- The notes about the format of the file `L6.dfa` in Problem 6 apply to `L7.dfa` as well.

- In `ps3.sml`, the following function has been defined for testing the equivalence of the FA and the DFA:

```
fun test7 () =
  let val L7_fa = FA.input "L7.fa"
      val L7_dfa = DFA.input "L7.dfa"
   in DFA.relationship(faToDFA L7_fa, L7_dfa)
  end
```

  If `L7.dfa` is correct, then executing `test7 ()` in Forlan will have the following behavior:

```
- test7();
languages are equal
val it = () : unit
```

  But if `L7.dfa` is incorrect, then executing `test7 ()` in Forlan will give counterexamples explaining why the languages are not equal. These counterexamples will help you debug your DFA.

Problem 8: Hamming It Up

HammingWorks, Inc., is a company that specializes in software for recognizing Hamming numbers — i.e., natural numbers that are divisible by 2, 3, or 5. Thus far, its main products have been the following two SML functions:

```
fun isHamming n = (n mod 2) = 0 orelse (n mod 3) = 0 orelse (n mod 5) = 0
val isNonHamming n = not o isHamming
```

Their customers have been clamoring for alternative ways to test Hamming numbers. So HammingWorks has promised that the next release of its software (available on Tue. Oct. 16, 2007) will include DFAs for accepting and rejecting Hamming numbers.

Unfortunately, all of the employees of HammingWorks are big Red Sox fans and have been spending all of their time watching the playoff games rather than working on their new product. The company desperately needs outside help.

Because of your familiarity with automata theory and the Forlan toolset, they have hired you as a consultant to finish their project. Here is what you must deliver by midnight on Mon., Oct 15:

- A Forlan DFA file `hamming.dfa` specifying a DFA whose language is the Hamming numbers (represented as strings of the digits `0` through `9`). This DFA should have the minimal possible number of states, and its states should be labeled with capital letters from the beginning of the alphabet (`A`, `B`, `C`, ...), with `A` as the start state.
- A diagram of a DFA described by `hamming.dfa`.
- A Forlan DFA file `nonhamming.dfa` specifying a DFA whose language is the non-Hamming numbers. It should follow the same conventions as in part **a**.
- A diagram of a DFA described by `nonhamming.dfa`.

You quickly realize that developing a minimal Hamming DFA from scratch would be very challenging and take lots of time. Thankfully, with the Forlan tools at your disposal, you have a better plan. You can first design a simple FA that accepts Hamming numbers, which is much simpler than designing a DFA because you can use nondeterminism. Then you can use Forlan to automatically derive a minimal Hamming DFA from your Hamming FA. Once you have a Hamming DFA, it is straightforward to convert it to a nonHamming DFA.

Follow this plan (and the notes below) to complete your assignment. In addition to submitting the four artifacts described above, you should also submit four more artifacts:

- A diagram of your FA for accepting Hamming numbers.
- An explanation in English of why your FA is correct.
- A Forlan FA file `hamming.fa` that corresponds to your FA digram.
- A description of the steps you followed and/or code you used to create the DFA artifacts from your FA artifacts.

Notes/Hints:

- To create a Hamming FA, design three separate FAs (for numbers divisible by 2, by 3, and by 5) and glue these together.
- For determining divisibility by 3, here's a relevant fact: a decimal number is divisible by 3 iff the sum of its digits is divisible by 3. E.g. `1467` is divisible by 3 because `1 + 4 + 6 + 7 = 18`, which is divisible by 3. Based on this fact, there is a 3-state DFA for determining devisibility by 3.
- The JFA application is helpful for drawing and displaying finite automata and for reading/writing out files in Forlan FA format. See the note on using JFA in problem 6.
- See the appendix of this problem set for the specification of some Forlan functions useful in this problem.
- The employees of HammingWorks have done *some* work on this project. They created the following helpful testing functions, which are supplied in `ps3.sml`. You should use these functions to test your FAs and DFAs.

  `testHammingFAFile: string -> int -> bool`
  `testHammingFAFile` ***`filename n`*** reads a Forlan FA from the file named ***`filename`*** and tests it on all strings of digits corresponding to the numbers between 0 and ***`n`***. Returns `true` if the FA acts like a Hamming-number-acceptor on all the numbers in the given range. Otherwise, returns `false` after displaying feedback on inputs in the range that are not correctly handled.

  `testHammingDFAFile: string -> int -> bool`
  This is like `testHammingFAFile`, but additionally verifies that the file describes a DFA.

  `testNonHammingFAFile: string -> int -> bool`
  `testNonHammingFAFile` ***`filename n`*** reads a Forlan FA from the file named ***`filename`*** and tests it on all strings of digits corresponding to the numbers between 0 and ***`n`***. Returns `true` if the FA acts like a non-Hamming-number-acceptor on all the numbers in the given range. Otherwise, returns `false` after displaying feedback on inputs in the range that are not correctly handled.

  `testNonHammingDFAFile: string -> int -> bool`
  This is like `testNonHammingFAFile`, but additionally verifies that the file describes a DFA.

Appendix

This appendix describes the most important Forlan functions you may need for this assignment, as well as some helper functions that have been provided in `ps3.sml`.

Simplification of Regular Expressions

Forlan provides the following three functions for simplification of regular expressions:

`Reg.weakSimplify: reg -> reg`
`Reg.weakSimplify` ***`regexp`*** simplifies ***`regexp`*** using Stoughton's weak simplification algorithm.

`Reg.simplify: (reg * reg -> bool) -> reg -> reg`
`Reg.simplify` ***`subset regexp`*** using Stoughton's strong simplification algorithm to simplify ***`regexp`***, using ***`subset`*** as a conservative approximation to the subset relation. Use `Reg.weakSubset` as the default subset relation.

`Reg.weakSubset: reg * reg -> bool`
`Reg.weakSubset` is the default conservative approximation for use in `Reg.simplify`.

`ps3.sml` provides the following helper functions based on the above:

`testWeakSimplify: string -> string`
`testWeakSimplify` ***`regexpString`*** returns a string representation of the result of using the weak simplification process to simplify the regular expression denoted by the string *regexpString*. `testWeakSimplify` ***`str`*** is equivalent to

```
Reg.toString (Reg.weakSimplify (Reg.fromString str))
```

`testSimplify: string -> string`
`testSimplify` ***`regexpString`*** returns a string representation of the result of using the strong implification process to simplify the regular expression denoted by the string *regexpString*. `testWeakSimplify` ***`str`*** is equivalent to

```
Reg.toString (Reg.simplify Reg.weakSubset (Reg.fromString str))
```

Converting Between and Comparing Regular Language Specifications

Forlan provides the following functions for converting between and comparing different specifications of regular languages:

`regToFA: reg -> fa`
`(regToFA` ***`regexp`***`)` returns a finite automaton that accepts the same language as the regular expression ***`regexp`***.

`faToReg: (reg -> reg) -> fa -> reg`
`(faToReg` ***`subset finaut`***`)` returns a regular expression that accepts the same regular language as the FA ***`finaut`***. The ***`subset`*** parameter is a conservative approximation to the subset relation used in simplifying the resulting regular expression. Use `Reg.weakSubset` as the default subset relation.

`faToEFA: fa -> efa`
`(faToEFA` ***`finaut`***`)` returns an EFA that accepts the same language as the FA ***`finaut`***.

`efaToNFA: efa -> nfa`
`(efaToEFA` ***`efinaut`***`)` returns an NFA that accepts the same language as the EFA ***`efinaut`***.

`nfaToDFA: nfa -> dfa`
`(nfaToDFA` ***`nfinaut`***`)` returns a DFA that accepts the same language as the NFA ***`nfinaut`***.

`EFA.injToFA: efa -> fa`
`(EFA.efaToFA` *`efinaut`*`)` treats an EFA *`efinaut`* as an FA.

`NFA.injToEFA: nfa -> efa`
`(EFA.nfaToEFA` *`nfinaut`*`)` treats an NFA *`nfinaut`* as an EFA.

`DFA.injToNFA: dfa -> nfa`
`(EFA.nfaToEFA` *`dfinaut`*`)` treats a DFA *`dfinaut`* as an NFA.

`DFA.renameStatesCanonically: dfa -> dfa`
`DFA.renameStatesCanonically(`*`dfa`*`)` returns a DFA accepting the same language as *`dfa`* in which the start state is named `A`, and all other states are named `B`, `C`, ...

`DFA.minimize: dfa -> dfa`
`DFA.minimize(`*`dfa`*`)` returns a DFA with the minimal number of states that accepts the same language as *`dfa`*.

`DFA.relationship;: dfa * dfa -> unit`
`DFA.relationship(`*`dfa1, dfa2`*`)` compares the languages denoted by the two DFAs arguments:

- if *`dfa1`* and *`dfa2`* denote the same language, it prints `languages are equal`;
- if the language of *`dfa1`* is a proper subset of the language of *`dfa2`*, it prints `first language is a proper subset of second language` and display a string that is in the second language but is not in the first.
- if the language of *`dfa1`* is a proper superset of the language of *`dfa2`*, it prints `first language is a proper superset of second language` and display a string that is in the first language but is not in thee second.
- if the languages of *`dfa1`* and *`dfa2`* are incomparable, it prints `neither language is a subset of the other language` and display a string that is in the first language but is not in the second as well as a string that is in the second language but is not in the first.

`DFA.relationship` can be used together with the conversion functions to compare the languages of any two specifications for regular languages. The following definitions in `ps3.sml` handle some common cases:

```
val faToDFA = nfaToDFA o efaToNFA o faToEFA

val regToDFA = faToDFA o regToFA

fun relationshipReg(reg1, reg2) = DFA.relationship(regToDFA reg1, regToDFA reg2)

fun relationshipRegString(regString1, regString2) =
  relationshipReg(Reg.fromString regString1, Reg.fromString regString2)
```