Course Project 3
Nondeterministic Finite Transducers

CSE 30151 Spring 2017

Version of January 19, 2017

To make the regular expression matcher more useful, let's extend it to regular expressions that can output strings. When a line matches, instead of just printing it out verbatim, it can perform some transformation on the string. The new tool will be called `more` (match and output using regular expression).[1] These kinds of regular expressions have been used extensively in computational linguistics [Kaplan and Kay, 1994]. They have some overlap with `sed`'s `s/·/·/` command, but there are important differences.

The new `:` operator (which we give lower precedence than `|`) is used to define transformations: α`:`β means "read a string matching α and write a string matching β." For example:

`(0:1|1:0)*`	bitwise negate a binary number
`(0|1)*(0:1)(1:0)*`	increment a binary number

(Try doing these with `sed`.) We'll also add another operator: α`;`β composes two transformations, feeding the output of α into the input of β.

Note that because of nondeterminism, more than one output string may be possible. For example, if the regular expression is `(a|b)*(a:b)(a|b)*` and the string is `aaa`, then `baa`, `aba`, and `aab` are all possible outputs. Your implementation should just print one of them (it doesn't matter which).

Getting started

The repository includes the following files:

```
bin/
  compare_nft
  singleton_nft
  parse_rre
```

[1]There's already a Unix tool called `more`, but who uses that anymore?

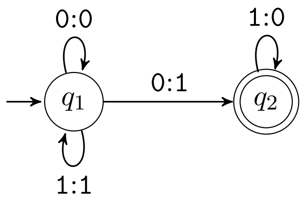
```
  cross_nft
  compose_nft
  output_nft
  more
examples/
  sipser-t1.nft
  sipser-t2.nft
tests/
  test-cp3.sh
doc/
  cp3.pdf
cp3/
```

Please place the programs that you write into the `cp3/` subdirectory.

1 Finite transducers

Finite transducers are mentioned in the book in Exercises 1.24–27. The basic idea is that whereas a finite automaton has transitions with just an input symbol, a finite transducer has transitions with both an input symbol and an output symbol.

For example, here's a (nondeterministic) finite transducer that increments a binary number:



Note three differences between the book's finite transducers and ours:

- The book's transducers are deterministic; ours are nondeterministic.
- The book writes transitions as $q \xrightarrow{a/b} r$; we'll stick with more common usage and write $q \xrightarrow{a:b} r$.
- The book's transducers do not distinguish accept and reject states; in effect, every state is an accept state. We do distinguish between accept and reject states.

Formally, a *nondeterministic finite transducer* (NFT) is a tuple $(Q, \Sigma, \Gamma, s, F, \delta)$, where

- Q is a finite set of states

- Σ is a finite input alphabet
- Γ is a finite output alphabet
- $s \in Q$ is the start state
- $F \subseteq Q$ are the accept states
- $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is a transition function

When the NFT is in state q and reads input symbol a, if $(r, b) \in \delta(q, a)$, it can transition to state r and write output symbol b. We write $R(M)$ for the string relation recognized by M; that is,

$$R(M) = \{(u, v) \mid M \text{ accepts } u \text{ while writing } v.\}$$

Extend your NFA data structure to a data structure that represents a nondeterministic finite transducer (NFT). It should support at least the following operations:

- Add a new transition $q \xrightarrow{a:b} r$.
- Iterate over all states.
- Iterate over all *input* symbols.
- Iterate over transitions out of state q.
- Iterate over transitions on *input* symbol a.

2 Retrofitting NFA operations for NFTs

Modify all of the NFA operations (except intersect, which we won't need) to work on NFTs.

- Read/write NFTs, in the same format as NFAs, but with symbols of the form $a : b$. See `sipser-t1.nft` and `sipser-t2.nft` for examples.
- emptyset() should build a NFT recognizing $\emptyset$.
- The singleton operation, given a string w, should build a NFT recognizing $\{(w, w)\}$.
- symbol(a) should build a NFT recognizing $\{(a, a)\}$.
- epsilon() should build a NFT recognizing $\{(\varepsilon, \varepsilon)\}$.
- The regular operations union, concat, and star.

Update your `singleton_nfa` to a program `singleton_nft` that uses your new NFT data structure and operations. Test it by running `test-cp3.sh`. It's possible (though not likely) that the automatic tester can't tell whether your NFT is correct; in that case, the grader will take a closer look to be sure. There aren't tests for the other operations; they shouldn't be too hard.

3 Extending the parser

Extend the parser to the following grammar (with $S = \mathsf{Compose}$):

$$
\begin{aligned}
\mathsf{Compose} &\to \mathsf{Compose}\boxed{\texttt{;}}\mathsf{Cross} \\
\mathsf{Compose} &\to \mathsf{Cross} \\
\mathsf{Cross} &\to \mathsf{Union}\boxed{\texttt{:}}\mathsf{Union} \\
\mathsf{Cross} &\to \mathsf{Union} \\
\mathsf{Union} &\to \mathsf{Union}\boxed{\texttt{|}}\mathsf{Conc} \\
\mathsf{Union} &\to \mathsf{Concat} \\
\mathsf{Concat} &\to \mathsf{Concat}\ \mathsf{Unary} && \text{if not followed by } \boxed{\texttt{;}}\boxed{\texttt{:}}\boxed{\texttt{|}}\boxed{\texttt{)}} \text{ or end of string} \\
\mathsf{Concat} &\to \varepsilon && \text{otherwise} \\
\mathsf{Unary} &\to \mathsf{Primary}\boxed{\texttt{*}} \\
\mathsf{Unary} &\to \mathsf{Primary} \\
\mathsf{Primary} &\to a && \forall a \in \Sigma \\
\mathsf{Primary} &\to \boxed{\texttt{@}} \\
\mathsf{Primary} &\to \boxed{\texttt{(}}\mathsf{Compose}\boxed{\texttt{)}}
\end{aligned}
$$

The pseudocode for parsing the new nonterminals would be:

```
function parseCompose()
    $M \leftarrow$ parseCross()
    while next token is $\boxed{;}$ do
        read $\boxed{;}$
        $M \leftarrow$ compose($M$, parseCross())
    return $M$

function parseCross()
    $M \leftarrow$ parseUnion()
    if next token is $\boxed{:}$ then
        read $\boxed{:}$
        $M \leftarrow$ cross($M$, parseUnion())
    return $M$
```

Notice the small changes needed in parsePrimary and parseConcat.

It's an error to write $\alpha{:}\beta{:}\gamma$. However, the grammar does allow $(\alpha{:}\beta){:}\gamma$ and $\alpha{:}(\beta{:}\gamma)$. The semantics of $:$ given below also allows for these cases. They aren't very useful, though.

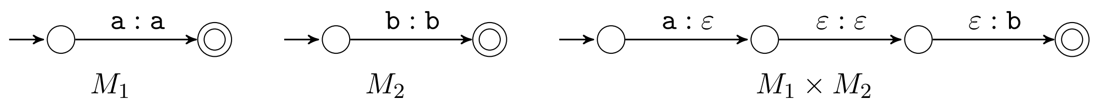
If you want to, you can modify the parser to rule them out. See Appendix A for some suggestions for how to do this.

As in CP2, write a program called `parse_rre` that takes a regular expression defined by the above grammar and outputs an AST. Test your program by running `test-cp3.sh`.

4 Cross product

Write a function `cross` that takes two NFTs, M_1 and M_2, and returns the NFT recognizing $\{(u, y) \mid (u, v) \in R(M_1), (x, y) \in R(M_2)\}$. This is the cross (Cartesian) product of the input language of M_1 and the output language of M_2.

This construction is very similar to the concatenation construction. We build a new automaton that simulates M_1 but discards all the output symbols. Then it simulates M_2 but discards all the input symbols. For example, for the regular expression `a:b`, we would have



More formally, given

$$M_1 = (Q_1, \Sigma_1, \Gamma_1, s_1, F_1, \delta_1)$$
$$M_2 = (Q_2, \Sigma_2, \Gamma_2, s_2, F_2, \delta_2),$$

assuming that $Q_1 \cap Q_2 = \emptyset$, the cross product is

$$M_1 \times M_2 = (Q_1 \cup Q_2, \Sigma_1, \Gamma_2, s_1, F_2, \delta),$$

where δ is defined as follows:

- For all $q \in Q_1$ and $a \in \Sigma_1 \cup \{\varepsilon\}$, if $(r, b) \in \delta_1(q, a)$, then $(r, \varepsilon) \in \delta(q, a)$.
- For all $q \in F_1$, $(s_2, \varepsilon) \in \delta(q, \varepsilon)$.
- For all $q \in Q_2$ and $a \in \Sigma_2 \cup \{\varepsilon\}$, if $(r, b) \in \delta_2(q, a)$, then $(r, b) \in \delta(q, \varepsilon)$.
- Nothing else is in δ.

Write a program called `cross_nft` to test your function:

`cross_nft` *nftfile nftfile*

should write the cross-product of the two NFTs to stdout. It's possible (though not likely) that the automatic tester can't tell whether your NFT is correct; in that case, the grader will take a closer look to be sure.

5 Composition

Extend your intersection algorithm to a composition algorithm that takes two NFTs, T_1 and T_2, and returns the NFT $M_1 \triangleright M_2$ recognizing $\{(u, w) \mid (u, v) \in L(M_1), (v, w) \in L(M_2)\}$.[2] For example, if we take the NFT that increments a binary number and compose it with itself, we get a NFT that increments a binary number by two.

Here's a formal definition of composition. If

$$\begin{aligned} M_1 &= (Q_1, \Sigma, \Gamma, s_1, F_1, \delta_1) \\ M_2 &= (Q_2, \Gamma, \Delta, s_2, F_2, \delta_2), \end{aligned}$$

let

$$M = (Q_1 \times Q_2, \Sigma, \Delta, (s_1, s_2), F_1 \times F_2, \delta)$$

where δ is defined as follows:

- For all $q_1, q_2 \in Q$, and $a \in \Sigma \cup \{\varepsilon\}, b \in \Gamma, c \in \Delta \cup \{\varepsilon\}$, if $(r_1, b) \in \delta_1(q_1, a)$ and $(r_2, c) \in \delta_2(q_2, b)$, then $((r_1, r_2), c) \in \delta((q_1, q_2), a)$.
- For all $q_1, q_2 \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, if $(r_1, \varepsilon) \in \delta_1(q_1, a)$, then $((r_1, q_2), \varepsilon) \in \delta((q_1, q_2), a)$.
- For all $q_1, q_2 \in Q$, $c \in \Delta \cup \{\varepsilon\}$, if $(r_2, c) \in \delta_2(q_2, \varepsilon)$, then $((q_1, r_2), c) \in \delta((q_1, q_2), \varepsilon)$.
- Nothing else is in δ.

Write a program called `compose_nft` to test your function:

`compose_nft` *nftfile nftfile*

should write the composition of the two NFTs to stdout. Test your program by running `test-cp3.sh`. It's possible (though not likely) that the automatic tester can't tell whether your NFT is correct; in that case, the grader will take a closer look to be sure.

6 Selecting the output string

Because of nondeterminism, it's possible that the NFT generates multiple possible output strings for a single input string. In that case, we're just going to print out one arbitrary one. Write a function:

- Argument: NFT M
- Return: an arbitrary string v such that $(u, v) \in R(M)$

[2]The $\triangleright$ symbol is made-up notation. Most authors use $\circ$ for composition, but Sipser uses $\circ$ for concatenation, so we needed something different.

You can do this using breadth-first search, similar to the emptiness check except that you have to remember a path to every state that you visit.

Write a program called `output_nft` to test your function:

`output_nft` *nftfile*

If the NFT has an accepting path, this program should write one possible output to stdout and exit with code 0. Otherwise, it should write nothing and exit with code 1. Test your program by running `test-cp3.sh`.

7 Putting it together

Write a function puts all the above functions together:

- Arguments: regular expression α, string w
- Return: one arbitrary output string if α matches w; raises an exception or returns some sentinel value otherwise

It should compile α into a NFT M and convert w into a singleton NFT M_w. Then, it composes them to form $M_w \triangleright M$. If $M_w \triangleright M$ has an accepting path, return the output symbols (not including ε).

Put your function into a command-line tool:

`more` *regexp*

where *regexp* is a regular expression. The program should read lines from stdin and, for each line that matches the regular expression, it should write to stdout one possible output string. Test your program by running `test-cp3.sh`.

Submission instructions

Your code should build and run on `student`*nn*`.cse.nd.edu`. The automatic tester will clone your repository, run `make` in subdirectory `cp3`, and then run `tests/test-cp3.sh`. You're advised to try all of the above steps and ensure that all tests pass.

To submit your work, please push your repository to Github and then create a new release with tag `cp3`. If you need to re-submit, create another release with a new tag starting with `cp3`, like `cp3.1`.

Rubric

Data structure	3
Operations	6
Parser	6
Cross product	4
Composition	4
Select output	4
Main program	3
Total	30

A Forbidding nested cross-products

One way to forbid expressions of the form $\alpha : (\beta : \gamma)$ or $(\alpha : \beta) : \gamma$ is to modify the syntax. In place of the nonterminals Cross, Union, Concat, Unary, and Primary, use Cross_1, Cross_2, Union_1, Union_2, etc. A subscript of 1 is used for expressions that do not contain a cross-product (**:**), while a subscript of 2 is used for expressions that do contain a cross-product. Then the CFG rule

$$\mathsf{Cross}_2 \rightarrow \mathsf{Union}_1 \boxed{:} \mathsf{Union}_1$$

would forbid nested cross-products.

Another way is to modify the semantics: add a flag to your NFT class that indicates whether it contains a cross-product. Then cross can check whether either of its arguments has this flag set; if so, it prints an error message.

References

Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994. URL `http://www.aclweb.org/anthology/J94-3001.pdf`.