Course Project 3
String Transformations

CSE 30151 Spring 2017

Version of April 5, 2017

In CP2, you built a regular expression matcher, which reads lines of input and prints each line that matches the regular expression. But just printing the line out is boring; can we do something more useful with it?

For example, in computational linguistics, it might be useful to write a regular expression that matches plural nouns (like `computers`, `theories`), but it would be more useful to transform plural nouns into their parts (`computers` to `computer+s`, `theories` to `theory+s`). In some languages, these transformations can become quite complex.

The Unix tools `tr` and `sed` can transform strings, but there are limits to both: `tr` cannot do the simple plural transformation described above; `sed` can, but it, too, has trouble with contextual transformations like “capitalize all words inside parentheses.”

In this project, we’ll use the machinery of regular expressions, extended with one new operator, to perform such transformations, upgrading `mere` to a tool called `more` (match and output using regular expression).[1] Although this document is long, the implementation is all based closely on CP1–2 and actually requires writing relatively little code.

Getting started

We’ve made some updates, so please have one team member run the commands

```
git pull https://github.com/ND-CSE-30151-SP17/theory-project-skeleton
git push
```

and then other team members should run `git pull`.

You will need a correct solution for CP1 and CP2 for this project. You may use the official solution or another team’s solution, as long as you properly cite your source.

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

Overview

We will enable regular expressions to perform transformations by adding a new operator, transduction (`:`), which we give lower precedence than union (`|`). An expression $\alpha : \beta$ means "read a string matching α and write a string matching β." Read, or try out, the following examples (user input in blue, comments in gray):

```
$ bin/more "0:1"                match a single 0 and invert it
0
1
1                               rejected because not 0
000                             rejected because not 0
101                             rejected because not 0

$ bin/more "(0:1)|(1:0)"        match a single bit and invert it
0
1
1
0
000                             rejected because more than one bit
101                             rejected because more than one bit

$ bin/more "((0:1)|(1:0))*"     invert all bits
0
1
1
0
000
111
101
010

$ bin/more "(0|1)*(0:1)(1:0)*"  invert all bits from the last 0 rightward
0
1
1                               rejected because no 0
000
001
101
010
```

Another way of describing the last expression is that it adds one to a binary number. This is an example of something that would be very difficult to do with `sed`.

Note that because an expression can match in more than one way, more than one output string may be possible. For example, consider the expression `(0|1)*(0:1)(0|1)*`. If the

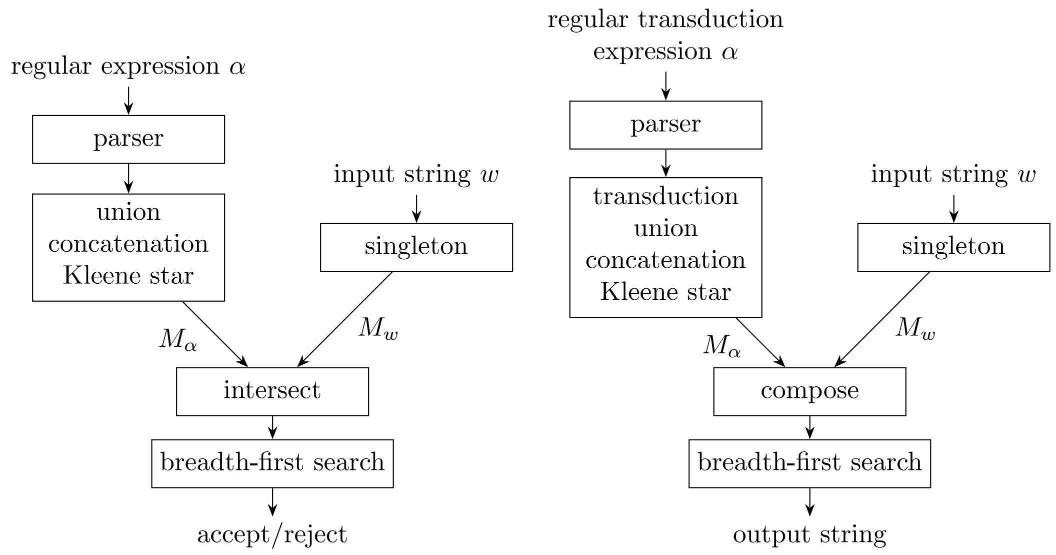


Figure 1: Overview of CP1–2 (left) and this project (right).

input string is `000`, then there are three output strings: `001`, `010`, and `100`. In that case, just one of them should be printed out (it doesn't matter which).

Remember that the regular expression matcher from CP1 and CP2 worked as shown in Figure 1 (left). This project consists of modifications to each component. In §1, you'll extend the parser to process regular *transduction* expressions. In §2, you'll extend NFAs to nondeterministic finite *transducers* (NFTs), and in §3, you'll extend the parser semantics to generate a NFT. In §4, you'll extend the NFA simulator (singleton, intersection, and breadth-first search) to simulate a NFT.

1 Regular Transduction Expressions

In this part, you'll extend the parser for regular expressions to a parser for our extended regular expressions, which we call *regular transduction expressions.*

1.1 Grammar (3 points)

The grammar from CP2 is written in the file `cp3/grammar.txt`. Modify this file so that the grammar handles the transduction (`:`) operator. The order of operations should be (from highest to lowest): star (`*`), concatenation, union (`|`), transduction (`:`). Getting the grammar right is important, so feel free to check your answer with an instructor or TA. When you're satisfied with your grammar, don't forget to commit it.

1.2 Parser (5 points)

Extend your parser from CP2 to use your new grammar. The semantics of $\alpha : \beta$ should be a call to transduce(α, β).

One tricky point is that in CP2, parseConcat stopped if the next character was `|` or `)`, or if there was no next character. These are the only possible characters that can follow a Concat. Under the new grammar, what additional character(s) could follow a Concat? Modify parseConcat accordingly.

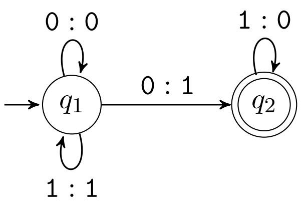
For testing purposes, write a program called `cp3/parse_rte` that parses a regular transduction expression and outputs a string repesentation of the parse tree. For example, `cp3/parse_rte "ab:c|d"` should give

```
transduce(concat(symbol(a),symbol(b)),union(symbol(c),symbol(d)))
```

Test your program by running `tests/test-cp3.sh`.

2 Finite Transducers

We will implement regular transduction expressions by converting them to *nondeterministic finite transducers* (NFTs). For a formal definition, please see Appendix A. Here's an example of a NFT that increments a binary number:



It looks like a NFA except that each transition is labeled $a : b$, where a and b are either symbols or ε. This means that the NFT, when following the transition, reads input a and writes output b. Suppose that the input string is `001`. The accepting run for this string is:

output	state	input
ε	q_1	`001`
`0`	q_1	`01`
`01`	q_2	`1`
`010`	q_2	ε

If there is *any* run that reaches the end of the input string in an accept state, then the transducer accepts the input string. Note that, for a given input string, there may be more than one accepting run, so there may be more than one possible output string. The *transduction* recognized by a NFT M, which we write $L(M)$, is the set of all string pairs (u, v) such that M accepts u and outputs v.

2.1 Data Structure (3 points)

Modify your NFA data structure into a data structure that represents a NFT. The main change is that transitions should have both an input and an output. It should support the same access operations as a NFA. It may be useful to have a function that takes a symbol a and returns (an iterator over) all transitions of the form $q \xrightarrow{a:b} r$.

Modify your NFA reader/writer to read/write NFTs, in the same format as before, except that a line for a transition $q \xrightarrow{a:b} r$ has four fields: q, a, b, and r. For examples, see `examples/sipser-t1.nft` and `examples/sipser-t2.nft`.

3 Parser Semantics

In this part, you'll implement the semantic actions for the parser of Part 1.2. These are all easy (or even trivial) modifications to functions you wrote in CP2.

3.1 Regular operations (3 points)

Modify the following NFA operations to work on NFTs. The NFA and NFT versions of these operations are identical, except that wherever the NFA version creates a transition $q \xrightarrow{a} r$ or $q \xrightarrow{\varepsilon} r$, the NFT version creates $q \xrightarrow{a:a} r$ or $q \xrightarrow{\varepsilon:\varepsilon} r$, respectively.

- emptyset() should build a NFT recognizing $\emptyset$ (rejects everything).
- epsilon() should build a NFT recognizing $\{(\varepsilon, \varepsilon)\}$.
- symbol(a) should build a NFT recognizing $\{(a, a)\}$.
- union(M_1, M_2) should build a NFT that applies either M_1 or M_2 to the input string.
- concat(M_1, M_2) should build a NFT that cuts the input string into two parts and applies M_1 to the first part and M_2 to the second part.
- star(M_1) should build a NFT that cuts the input string into zero or more parts and applies M_1 to each of them.

You are **not** required to pass tests for this subpart. But if you write programs `union_nft`, `concat_nft`, or `star_nft`, then `tests/test-cp3.sh` will check them for you. If it says "UNKNOWN," this is a possible failure, but the grader will take a closer look to be sure.

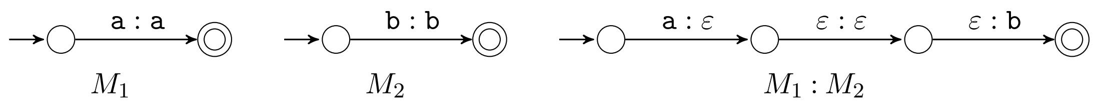
3.2 Transduction (3 points)

The semantics of the transduction (`:`) operator should be a function that takes two NFTs, M_1 and M_2, and returns the NFT that recognizes the transduction

$$\{(u, y) \mid (u, v) \in L(M_1) \text{ and } (x, y) \in L(M_2)\}.$$

Typically, this operator is used on NFTs that don't perform any transformations, and this is where it makes most sense. For example, if M_1 only accepts and outputs `a`, and M_2 only accepts and outputs `b`, then $(M_1 : M_2)$ only accepts `a` and outputs `b`.

This construction is very similar to the concatenation construction. Set all the outputs of M_1 to ε (accept every input that M_1 accepts, but discard the output), and set all the inputs of M_2 to ε (output every string that M_2 outputs, regardless of input). Then add $\varepsilon : \varepsilon$ transitions from every accept state of M_1 to the start state of M_2 to link them together. For example:



For a formal definition, please see Appendix A.

Write a function called `transduce` that implements the above, and write a program called `cp3/transduce_nft` to test it:

`cp3/transduce_nft` *nftfile nftfile*

should read NFTs M_1 and M_2 from the two files and write $M_1 : M_2$ to stdout. Test your program using `tests/test-cp3.sh`. If the automatic tester says "UNKNOWN," this is a possible failure, but the grader will take a closer look to be sure.

4 Simulating NFTs

In CP1, we simulated a NFA M on a string w by creating a singleton NFA that recognized $\{w\}$, intersecting that with M, and then looking for an accepting path in the resulting NFA. For NFTs, we can do something very similar:

1. Create a singleton NFT M_w that recognizes $\{(w, w)\}$; that is, it only accepts w, and it only outputs w.
2. *Compose* M_w and M, making a NFT $M_w \triangleright M$ that feeds the output of M_w to the input of M. So $M_w \triangleright M$ only accepts w and outputs what M outputs on w, but for any other input string, it rejects.
3. So we can tell whether M accepts w by searching for any path through $M_w \triangleright M$. If it has one, concatenate its output symbols to form an output string.

You'll update your NFA simulator to do these three steps in the following subparts.

4.1 Singleton (1 point)

Update your NFA `singleton` operation to create a NFT instead, and update the test program `cp2/singleton_nfa`:

`cp3/singleton_nft` w

should write to stdout the NFT that recognizes $\{(w, w)\}$. Test your program by running `tests/test-cp3.sh`. If the automatic tester says "UNKNOWN," this is a possible failure, but the grader will take a closer look to be sure.

4.2 Composition (3 points)

The composition of two NFTs T_1 and T_2, which we write as $M_1 \triangleright M_2$, is the NFT recognizing

$$\{(u, w) \mid (u, v) \in L(M_1), (v, w) \in L(M_2)\}.$$

Composition is very similar to intersection. We build a new NFT that simulates M_1 and M_2 simultaneously. But whereas the intersection construction feeds the same input string to both M_1 and M_2, the composition construction feeds the input string to M_1 and feeds the output of M_1 into the input of M_2. Figure 2 shows three examples that illustrate the three cases of the construction. For a formal definition, please see Appendix A.

Write a function `compose` that implements this construction, and write a program `cp3/compose_nft` to test it:

`cp3/compose_nft` *nftfile nftfile*

should write the composition of the two NFTs to stdout. Test your program by running `tests/test-cp3.sh`. If the automatic tester says "UNKNOWN," this is a possible failure, but the grader will take a closer look to be sure.

4.3 Selecting the output string (3 points)

The final step is to extract an output string, if there is one. Write a function that takes a NFT M, and returns an arbitrary string v such that $(u, v) \in L(M)$; it returns a sentinel value or raises an exception if there is none.

You can do this using breadth-first search, similar to the emptiness check from CP1 except that you have to remember the path from the start state to each state. (Actually, the official solution for CP1 includes a function `any_path` that more or less does this already.) When you find a path, concatenate the output symbols along the path (omitting ε) to get the output string.

Write a program called `cp3/output_nft` to test your function:

`cp3/output_nft` *nftfile*

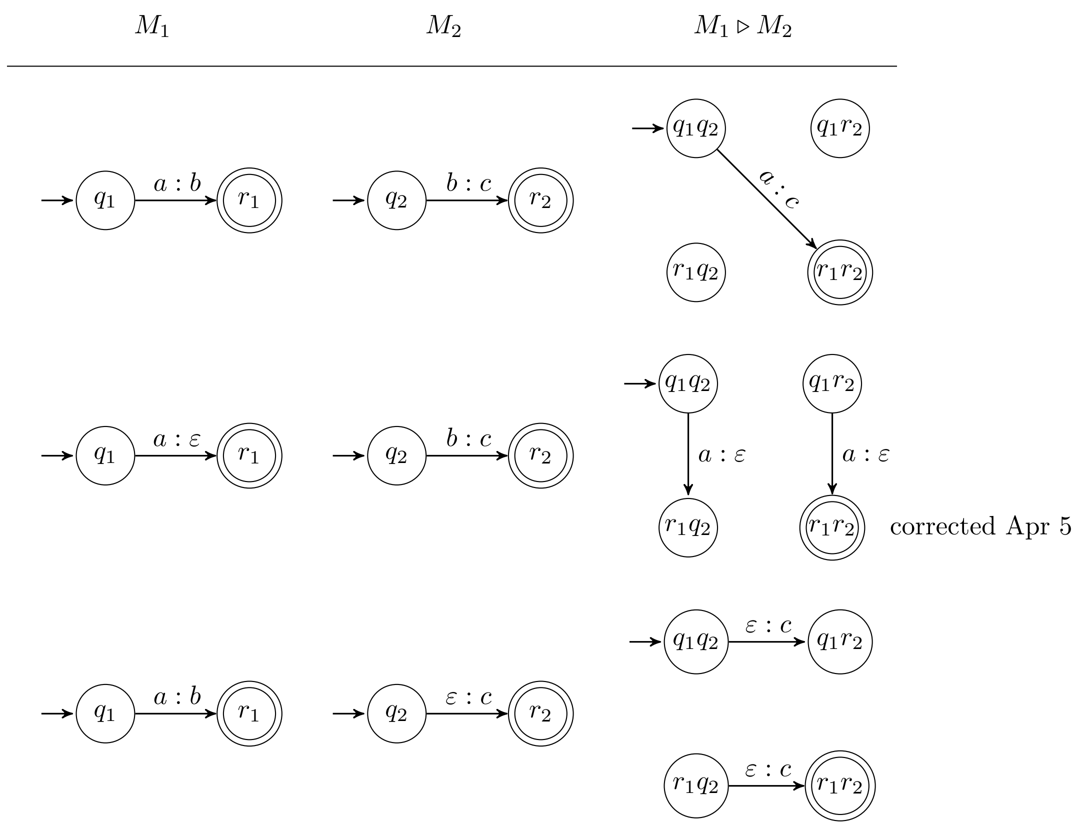


Figure 2: Three examples illustrating the three cases of the composition construction.

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 `tests/test-cp3.sh`.

5 Putting it together (6 points)

Write a program called `cp3/more` that puts all the above together. It should be called like

`cp3/more` *rtexp*

where *rtexp* is a regular transduction expression. See Section 1 for example runs. It should work like this:

- Compile *rtexp* to a NFT M, using the parser of §1.2 with the semantic actions of §3.
- For each line w that is read from stdin:

1. Convert w into a singleton NFT M_w (§4.1).
2. Compose them to form $M_w \triangleright M$ (§4.2).
3. If $M_w \triangleright M$ has an accepting path, print the output string to stdout (§4.3).

Test your program by running `tests/test-cp3.sh`.

Submission instructions

Your code should build and run on `student`*nn*`.cse.nd.edu`. The automatic tester will clone your repository, `cd` into its root directory, run `make -C cp3`, and 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: If you are working in a branch, please merge to `master`. Push your repository to Github and then create a new release with tag version `cp3` (note that the tag version is not the same thing as the release title). If you are making a partial submission, then use a tag version of the form `cp3-123`, indicating which parts you're submitting.

A Formal Definitions

Nondeterministic finite transducer Let Σ be a finite alphabet, and let $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. A *nondeterministic finite transducer* (NFT) is a tuple $(Q, \Sigma, \delta, s, F)$, where Q, s, and F are as in a NFA, and δ is a transition function

$$\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q \times \Sigma_\varepsilon).$$

A run of a NFT starts in state s. If the NFT is in state q and the next input symbol is a:

- If $\delta(q, a)$ contains (r, b), it can read a, write b, and transition to state r, or
- If $\delta(q, \varepsilon)$ contains (r, b), it can write b and transition to state r.

The NFT accepts u while writing v iff there is a run that reads u and writes v and ends in an accept state. We write $L(M)$ for the transduction recognized by M; that is,

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

Transduction Given

$$M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$$
$$M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2),$$

assuming that $Q_1 \cap Q_2 = \emptyset$, let

$$M_1 : M_2 = (Q_1 \cup Q_2, \Sigma, \delta, s_1, F_2),$$

where δ is defined as follows:

- For all $q \in Q_1$ and $a \in \Sigma_\varepsilon$, 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 $a \in \Sigma_\varepsilon$, if $(r, b) \in \delta_2(q, a)$, then $(r, b) \in \delta(q, \varepsilon)$.
- Nothing else is in δ.

Composition Given

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

let

$$M_1 \triangleright M_2 = (Q_1 \times Q_2, \Sigma, \delta, (s_1, s_2), F_1 \times F_2)$$

where δ is defined as follows:

- For all $q_1, q_2 \in Q$, and $a \in \Sigma_\varepsilon, b \in \Sigma, c \in \Sigma_\varepsilon$, 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 $q_1, q_2 \in Q$, $a \in \Sigma_\varepsilon$, 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 $q_1, q_2 \in Q$, $c \in \Sigma_\varepsilon$, 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.