IIT CS440: Programming Languages and Translators

Homework 2: DFAs, NFAs, Regexes, CFLs

Prof. Stefan Muller
TA: Xincheng Yang

Out: Tuesday, Feb. 16
Due: Tuesday, Mar. 2 11:59pm CST

Updated Mar. 1

This assignment contains 7 written tasks and 6 programming tasks, for a total of 85 points.

0 Logistics and Submission - Important

The same rules as on HW0 and HW1 apply, except as specified below.

0.1 Compilation and Testing

As noted on Piazza, the complicated skeleton code of this assignment will make testing in the `ocaml` toplevel or on TryOCaml very difficult. Instructions for testing your code are provided in the writeups of the programming problems. If you have any issues or questions, let us know or post on Piazza.

0.2 Submission

Your answers to the programming problems will go in the files `dfa.ml`, `nfa.ml` and `regex.ml`. The other ML files are skeleton code that you will not need to modify and don't need to submit. Unlike in previous assignments, **Do not rename the .ml files.** The skeleton code, and the compilation instructions in this writeup, assume the files still have the original names. As before, your written answers should be submitted as a pdf file named something like `Doe_John_440_hw2.pdf`. Put the four files (three .ml files and one .pdf file) into a folder named something like `Doe_John_440_hw2` and zip the file to submit. Submit on Blackboard under the correct assignment.

0.3 Rules for Programming Problems

Unless noted in specific problems, you do not need to program in any specific style (e.g., recursive, tail-recursive, using higher-order functions), though you are encouraged to use higher-order functions when possible as good practice and good coding style.

You may (and are encouraged to) use functions from OCaml's standard library when applicable. Documentation of the standard library functions is available here:
`https://caml.inria.fr/pub/docs/manual-ocaml/libref/index.html`

1 Regular Expressions and Finite Automata

Task 1.1 (Written, 9 points).

For each of the following language descriptions, give a regular expression that matches exactly that language. You don't have to find the shortest possible regular expression. You may use expressions such as $[a-z]$ to represent the letters a through z, and/or regular definitions:

$$\begin{array}{lcl} \textit{letter} & \rightarrow & [a-z] \\ \textit{string} & \rightarrow & \textit{letter}^* \end{array}$$

To make things clearer, use ␣ to mean a single space (e.g. write `"Hello␣world"` instead of `"Hello world"`.)

(a) Strings over the alphabet $\{a, b, c, d\}$ that are in alphabetical order.
Examples: *aabccc*, *abcd*, *bd*, *a*, *ad*
Not examples: *dc*, *ca*, *cba*

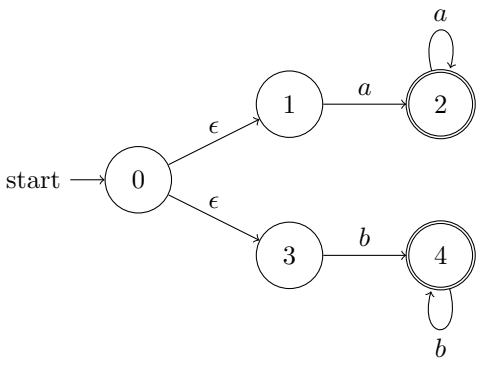
(b) Natural numbers (0 and up), with no leading zero unless the whole thing is a single zero, and going right-to-left, groups of 3 digits are separated by commas.
Examples: 0; 1; 12; 123; 1,234; 12,345; 1,000,000
Not examples: 01; 1000; 123,4; 12,34

(c) Strings consisting of letters a-z and spaces that *do not* begin or end with whitespace and all white space is one space long.
Example: `"so this is ok"`
Not examples: `"this  is bad"`, `" this too"`, `"and this "`

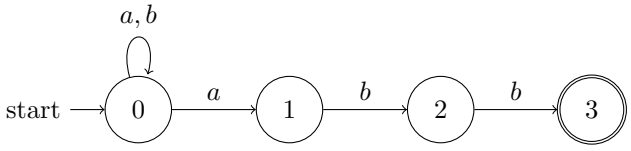
Task 1.2 (Written, 10 points).

For each of the following NFAs, identify the language accepted by the NFA (describe it or give a regular expression).

(a)



(b)

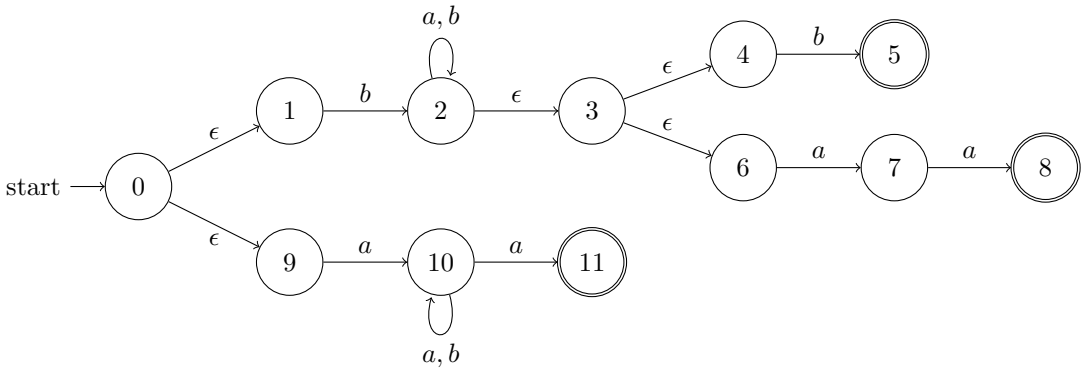


1.1 NFA to DFA

Consider the following NFA, which accepts strings that:

- begin with b and end with b or aa (with 0 or more as and bs in between) or
- begin and end with a (with 0 or more as and bs in between).

(convince yourself this is the lanugage recognized by the NFA).



Task 1.3 (Written, 6 points).

Recall that $\epsilon-closure(s)$ for a state s is the set of states reachable from s using only ϵ-transitions, and that this idea of "reachability" is transitive, that is, if there is an ϵ-transition from s to s' and one from s' to s'', then s'' is in $\epsilon-closure(s)$.

Compute

(a) $\epsilon-closure(0)$

(b) $\epsilon-closure(3)$

(c) $\epsilon-closure(2)$

Task 1.4 (Written, 10 points).

Use the subset construction to convert the NFA above into a DFA. You may present the resulting DFA using either a transition diagram or a transition table. Recall that, in the subset construction, states of the DFA correspond to sets of states of the NFA. Label your DFA states with the set of NFA states, e.g. $\{1, 2, 3\}$. The start state should be $\epsilon-closure(0)$.

Task 1.5 (Written, 5 points).

Why does the DFA you constructed above not have an "error" state? (Think about when, during scanning an input, we would realize that a string cannot be in this language.)

2 DFA Simulator

In the next section, we're going to build a regular expression matcher, which requires simulating an NFA. As a warmup, in this section, you're going to simulate a DFA, which is a little easier. Your code for this section will be in `dfa.ml`. At the top, we give a few type definitions for DFAs: states and symbols are just integers and characters, respectively, but we still define them using "type synonyms":

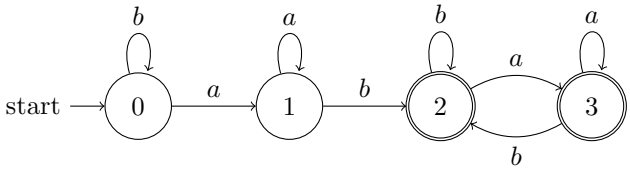
```
type state = int
```

This defines a type `state` that's just the same as `int`, but we can refer to `state` to make it clearer when we mean a state in the code. (This is also good software engineering practice because if we decide we don't want to just represent states using ints later, we can change it easily.)

DFAs are implemented as records:

```
type dfa = { states : int; (* Number of states *)
             (* States are numbered 0 through states
              * State 0 is the start state *)
             accept : state list; (* List of accept states *)
             alpha  : symbol list; (* Alphabet of the DFA *)
             trans  : (state * symbol * state) list
             (* List of transitions in the form
              * (from state, transition symbol, to state) *)
           }
```

For example, we could represent the DFA



with the record

```
{ states = 4; (* 4 states, 0-3 *)
  accept = [2; 3]; (* states 2 and 3 are accepting *)
  alpha = ['a'; 'b']; (* alphabet is a, b *)
  trans = [(0, 'b', 0); (0, 'a', 1);
           (1, 'a', 1); (1, 'b', 2);
           (2, 'b', 2); (2, 'a', 3);
           (3, 'a', 3); (3, 'b', 2)]
}
```

Writing large DFAs like that is hard though, so we've provided functions that parse DFAs and inputs from input files. The format of the input file is as follows:

```
<number of states> <number of accepting states> <size of alphabet>
<accepting states, one per line>
<alphabet, a character string with no spaces>
<input, a character string with no spaces>
<transitions, one per line, each of the format:>
<old state> <symbol> <new state>
```

An example is in `examples/dfa1`. You may want to write your own examples for testing.

The function `Parser.parse_file : string -> dfa * symbol list` takes a filename and returns a pair of the DFA and the input. The parsing code is in a different file, so you won't be able to use it by just copying and pasting code into the toplevel or TryOCaml. You can still test using asserts, though. For example:

```
assert (transition (fst (Parser.parse_file "examples/dfa1")).trans 'a' 0 = 1)
```

Updated (Code updated 3/1). You can then check if your assert passes by compiling the file and running it:

```
ocamlc -o dfa_test parseDFA.ml dfa.ml
./dfa_test
```

If you don't see any errors, then your assertion passed. You can also test that your whole DFA simulator works using the instructions at the end of this section.

Task 2.1 (Programming, 5 points).

Implement the function
`transition : (state * symbol * state) list) -> symbol -> state -> state`.
`transition d.trans sy st` should return the new state that we'll be in if we see symbol `sy` while in state `st`, where `d.trans` is the list of transitions of the DFA (in the form above).

For example, if `d` is the DFA above,

- `transition d.trans 'a' 0 = 1`
- `transition d.trans 'b' 1 = 2`

You should raise the exception `IllformedInput` if the state or symbol isn't in the transition list (e.g. if `st` is 5 or `sy` is 'q'). The exception `IllformedInput` takes a string, so you can output a helpful message that will be printed, e.g.

```
raise (IllformedInput
                (Printf.sprintf
                   "State, symbol pair (%d, %c) not in transitions"
                    state symb))
```

Now you'll write the function that actually simulates the DFA. Applying `dfa_sim dfa st input` simulates execution of a DFA starting in state `st` on the input `input`. It should return true if we end up in an accepting state, and false otherwise. We should get an `IllformedInput` exception if something is wrong (e.g., a transition is missing from `dfa.trans` or we see a symbol not in `dfa.alpha`). The initial state of the DFA is always state 0, but you'll probably want `dfa_sim` to call itself recursively when you're in a new state, so `dfa_sim` takes the current state as an argument.

Task 2.2 (Programming, 8 points).

Implement the function `dfa_sim: dfa -> state -> symbol list -> bool` as described above. Remember that a DFA only accepts or rejects once it has read all of the input: we can pass through an accepting state, then read more input, and end up in a rejecting state.

2.1 Testing

Run `make dfa` to compile the whole DFA simulator. If you don't have make installed, you can run `ocamlc -o dfa parseDFA.ml dfa.ml dfaMain.ml` instead.

Then run `./dfa <filename>` to parse the DFA and input from the given file and run your DFA simulator on it. The program will print either "ACCEPTED" or "REJECTED" depending on whether your simulator returns true or false.

3 Regular Expression Matcher

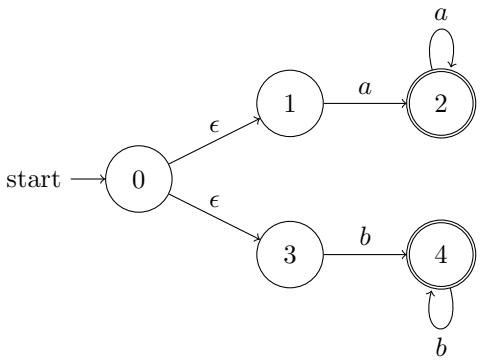
Lots of software has to solve the problem of deciding whether or not a piece of text matches a given regular expression. In fact, the find/replace feature of the editor you're using to write your code probably implements it. In this section, you'll build part of a regular expression matcher. To do this, you'll write code that simulates an NFA (recall that we can "compile" any regular expression into an NFA that determines whether input matches it).

3.1 NFA Simulator

You'll implement the NFA simulator in `nfa.ml`. This file already includes type definitions for NFAs. They're similar to the types for DFAs above, with the difference that the type of a transition is now

`state * symbol option * state`. If the symbol is `None`, this is an ϵ-transition.

We can represent the NFA



with the record

```
{ states = 5; (* 5 states, 0-4 *)
  accept = [2; 4]; (* states 2 and 4 are accepting *)
  alpha = ['a'; 'b']; (* alphabet is a, b *)
  trans = [(0, None, 1); (0, None, 3); (* epsilon transitions *)
           (1, Some 'a', 2); (3, Some 'b', 4);
           (2, Some 'a', 2); (4, Some 'b', 4)]
}
```

Recall that NFAs don't always list all of the transitions: an unlisted transition is assumed to automatically go to a rejecting state and stay there regardless of the rest of the input. So, for example, on the above NFA, if we're in state 1 and see a b, we can immediately reject the input. States like this that lead to immediate rejecting states aren't listed in the transition list (so, in particular, unlike with the DFA simulator, seeing a state and symbol not listed in the transition list should no longer raise an exception).

Again, we've given you the function `Parser.parse_file: string -> nfa * symbol list`. The file format is the same as above, except that the lines for transitions can now have the form `<state> <state>` indicating an ϵ-transition. You can test asserts by compiling as follows:

```
ocamlc -o nfa_test parseNFA.ml nfa.ml
./nfa_test
```

You can, of course, simulate an NFA by converting it to a DFA using the algorithm we learned in class and using your DFA simulator. However, this is both less efficient (because the DFA might be exponentially larger than the NFA) and more complicated than just simulating the NFA directly. An efficient algorithm for this is listed in Chapter 3.7.2 of Aho et al.'s "Purple Dragon Book." The basic idea is to keep track of a set S of states that can be reached from the initial state with the input we've seen so far. We start off with $S = \{0\}$, and then repeat the following until we've seen all the input.

- Take the ϵ-closure of S.
- If we've reached the end of the input, then accept if and only if at least one state in S is an accepting state.
- Otherwise, if the next symbol is a, for each state $s \in S$, remove s from S and add to S any states that s transitions to on a (since this is an NFA, this may be zero, one or more states).

In the above, we talk about sets, but we haven't seen a set data structure in OCaml (there is one in the standard library, but it's a little tricky to work with). Instead, we'll still use lists with the observation that we can treat a list as a set if we sort it and remove duplicates. We've given you a function `norm : state list -> state list` that does just this.

Task 3.1 (Programming, 7 points).

Implement the function
`transitions : (state * symbol option * state) list -> symbol option -> state -> state list`
`transitions n.trans (Some sy) st` should return the list of states we can transition to on seeing symbol `sy` in state `st`. `transitions n.trans None st` should return the list of states with ϵ-transitions from `st`.

For example, if `n` is the NFA above,

- `transitions n.trans None 0 = [1; 3]`
- `transitions n.trans (Some 'a') 1 = [2]`
- `transitions n.trans (Some 'b') 1 = []`
- `transitions n.trans None 1 = []`

Your output does not need to be normalized, i.e. it may contain duplicates and need not be sorted (though you can do this if you want).

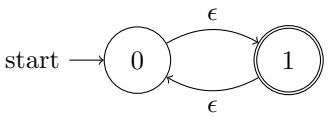
Next, you'll implement a function `eps_clos` that takes the ϵ-closure of a set of states, as defined above. The procedure for this is:

- For each $s \in S$, add to S any states with ϵ-transitions from s.
- Repeat until you've reached a fixed point, that is, running the above step doesn't add any more states to S.

For example, on the NFA above,

- `eps_clos [0] = [0; 1; 3]`
- `eps_clos [1] = [1]`
- `eps_clos [0; 1] = [0; 1; 3]`
- `eps_clos [0; 2] = [0; 1; 2; 3]`

Note that we don't ever remove states from S when taking the ϵ-closure. The second step is important to make sure your function doesn't run into an infinite loop. For example, consider the NFA



When taking the ϵ-closure of `[0]`, we can stop as soon as we've gotten to `[0; 1]`, but if we're not careful, we might try to add `0` to the set again, and then add `1` again, and so on... you should recognize when you're not adding any new states and stop. Note that taking the ϵ-closure of a set is always guaranteed to terminate because on each iteration, if we don't add any new states, we stop and you can only continue adding states until you've added all the states in the NFA, at which point you have to stop. Note that the Purple Dragon Book gives an algorithm for computing the ϵ-closure, but their algorithm is very imperative: we're trying to think more functionally.

Task 3.2 (Programming, 12 points).

Implement the function `eps_clos : nfa -> state list -> state list` described above.

Hint: You may (or may not) find the standard library function `List.concat` helpful.

Task 3.3 (Programming, 10 points).

Implement the function `nfa_sim: nfa -> state list -> symbol list -> bool`. `nfa_sim nfa states input` should:

- Let `states'` be the ϵ-closure of `states` (we've done this part for you)
- If we're out of input, return true if any state in `states'` is an accepting state, otherwise return false.
- Otherwise, look at the next symbol, update the set of states accordingly, and call `nfa_sim` with the new set of states.

Hint: You may (or may not) find the standard library function `List.exists` helpful.

3.2 Testing the NFA Simulator

Run `make nfa` to compile the whole NFA simulator. If you don't have make installed, you can run `ocamlc -o nfa parseNFA.ml nfa.ml nfaMain.ml` instead.

Then run `./nfa <filename>` to parse the NFA and input from the given file and run your NFA simulator on it. The program will print either "ACCEPTED" or "REJECTED" depending on whether your simulator returns true or false.

3.3 Converting Regular Expressions to NFAs

The file `regex.ml` has a type definition for regular expressions, a bunch of code for converting regular expressions to NFAs, a bunch more code for parsing regular expressions, and then a bit of code at the end that runs a regular expression matcher on an input string and regular expression. Most of this code is already written. You'll just write one small function that builds an NFA that recognizes a single symbol.

Task 3.4 (Programming, 3 points).

Implement the function `symb_nfa : symbol option -> nfa`. `symb_nfa (Some s)` should return an NFA over the alphabet `[s]` that accepts only the input `s`. `symb_nfa None` should return an NFA over the empty alphabet that accepts only the empty string.

Hint: In both cases, the NFA should have two states and one transition.

3.4 Testing the Regular Expression Matcher

Run `make regex` to compile the regular expression matcher. This depends on your NFA code, so you need to have done that part already. If you don't have make installed, you can run `ocamlc -o regex parseNFA.ml nfa.ml regex.ml regexMain.ml` instead.

Then run `./regex <regular expression> <string>` to check if the given string matches the regular expression. The program will print either "ACCEPTED" or "REJECTED" depending on whether your matcher returns true or false.

For example, `./regex "ab*" "abbbbbbb"` should print "ACCEPTED" but `./regex "ab*" "b"` should print "REJECTED." *Updated (Typo fixed 3/1)*

4 Standard Written Questions

Task 4.1 (Written, 0 points).

How long (approximately, in hours/minutes of actual working time) did you spend on this homework, total? Your honest feedback will help us with future homeworks.

Task 4.2 (Written, 0 points).

Who, if anyone, did you collaborate with (and in what way), and what outside sources, if any, did you consult in working on this homework?