Mentor – A Tool for Formal Languages and Automata

We developed a tool called **Mentor** that lets you describe automata through a simple syntax and evaluate them. This makes doing homework easier by letting Mentor evaluate and debug your automaton rather than tracing it out by hand. Mentor has a command-line interface you can use when writing up your homework solutions (it is available on CSIL). We also integrated Mentor into Gradescope so that homework problems can be automatically graded.

Obtaining and invoking Mentor

You can find mentor at `~emre/bin/mentor` at CSIL. You can run the tool using the command line, like so (here `$` is your command-line prompt):

```
$ ~emre/bin/mentor input-file command arguments
```

From here on, when giving command-line examples we will abbreviate `~emre/bin/mentor` to `mentor`. To see a list of commands and their descriptions, you can just run `mentor` (without any arguments). The commands supported are as follows:

- `<description-file> accept <input-string>`: Run the automaton given in `<description-file>` on the input string. See the next section for how to describe automata and for some examples.
- `<description-file> graph <output-file>`: Parse the automaton described in `<description-file>` and produce a PDF containing a visual representation of the automaton at `<output-file>`.
- `<description-file> generate <number>`: For a context-free grammar or pushdown automaton in `<description-file>` enumerate the first `<number>` strings in shortlex order.

The automaton description language

The description file format is like below. Note that the description file is *case sensitive*:

```
alphabet: {<alphabet>}
start: <starting state>
accepting: {<accepting states>}

<transitions>
```

- The type of the automaton is determined by the name of the description file:
  - If the file name ends with `.dfa`, then it is interpreted as a deterministic finite automata.

  - Similarly, files with names ending with `.nfa` are interpreted as non-deterministic finite automata.
  - Mentor supports `.dfa`, `.nfa`, `.re` (regular expression), `.cfg` (context-free grammar), and `.pda` (pushdown automaton).

The alphabet and the input strings

The alphabet is a set of symbols where each symbol consists of digits or lower case letters. For example, the following sets are valid alphabets: `{01, 10, 00, 11}`, `{0, 1}`, `{foo, bar}`, `{a, b}`.

A valid input string is a string of symbols from the alphabet. The symbols are separated by space if the alphabet contains symbols that consist of more than one character, otherwise the spaces are optional. For example,

- `foo bar foo` is a valid string for the alphabet `{foo, bar}`.
- `01101010` is a valid string for the alphabet `{0, 1}`.

State names and state sets

- State names consist of alphanumeric characters and underscore.
- Accepting states and other sets of states are denoted like sets in math. For example, `{Q1, Q3, Q5}` is a set containing `Q1`, `Q3`, `Q5`.

Transitions

A transition is of the form: `sourceState (character -> targetState)`.

As a full example, the DFA below accepts strings of `a`s whose length is a multiple of 3.

```
alphabet: {a}
start: Q0
accepting: {Q0}

Q0 (a -> Q1)
Q1 (a -> Q2)
Q2 (a -> Q0)
```

If you create a file named `3x.dfa` with the contents above, you can try the following commands to check if the DFA recognizes a string (here `$` is your command prompt):

```
$ mentor 3x.dfa accept a
Input "a" is not recognized by the automaton

$ mentor 3x.dfa accept aaa
Input "aaa" is recognized by the automaton
```

```
$ mentor 3x.dfa accept aaaaaa
Input "aaaaaa" is recognized by the automaton

$ mentor 3x.dfa accept aaaaa
Input "aaaaa" is not recognized by the automaton

$ mentor 3x.dfa accept ''
Input "" is recognized by the automaton

$ mentor 3x.dfa accept aba
Character 'b' at index 1 of the input string is not in the alphabet
```

The following command will produce a PDF file containing the diagram in Figure 1.

```
$ mentor 3x.dfa graph 3x.dfa.pdf
```

There are some features of the DSL for your convenience. You can cluster multiple transitions from the same state using the following notation.

```
Q1 (a -> Q2) (b -> Q3)
```

This is the same as writing the following:

```
Q1 (a -> Q2)
Q1 (b -> Q3)
```

Empty transitions Nondeterministic finite automata allow transitions that do not consume any character. To represent such transitions, you write `_` as the label of a transition.

For example, the following code describes the automaton in Figure 2.

```
alphabet: {a, b, c, d}
start: Q0
accepting: {Q1, Q2}

Q0 (_ -> Q1) (_ -> Q2)
Q1 (b -> Q1) (c -> Q1) (d -> Q1)
Q2 (a -> Q2) (c -> Q2) (d -> Q2)
```

Sanity checks

The tool automatically performs some sanity checks to see if the automaton is valid. The checks done for the DFA are:

- whether each state has a transition for each character in the alphabet.
- whether each state has only one transition for each character in the alphabet.

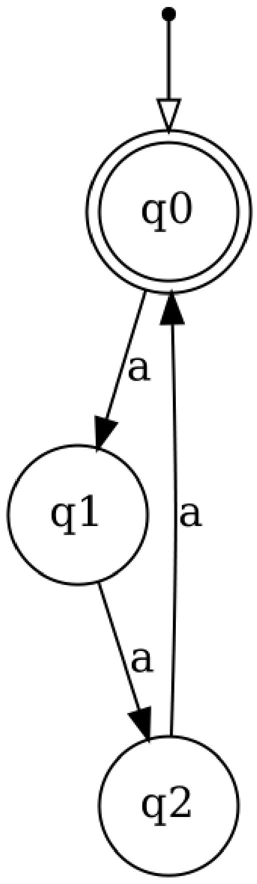

Figure 1: Diagram for `3x.dfa`

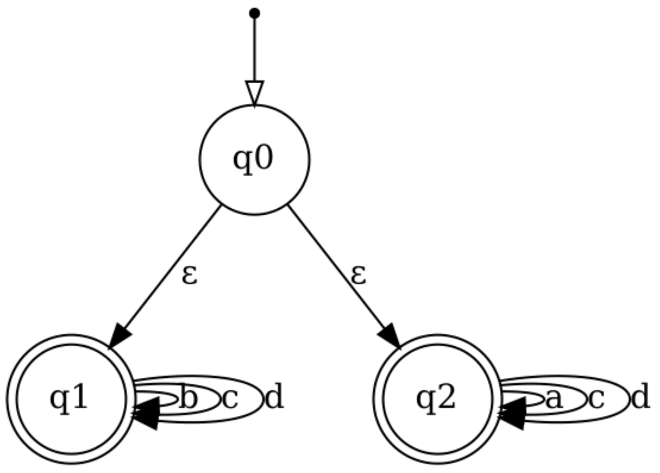

Figure 2: NFA with empty string transitions.

Full syntax description

Now that we've reviewed what Mentor is and went through a few examples, the following describes the full syntax of our formal languages and automata description language.

Deterministic finite automata (DFA)

Expected format for DFA. A comma-separated list of characters, a single start state, and a comma-separated list of accepting states. A listed state should have one or more state transitions (`<char> -> <state>`).

```
alphabet: '{'<char> (',' <char>)*'}'
start: <state>
accepting: '{'<state> (',' <state>)*'}'
<state> ('('<char> -> <state>')')+
```

Example:

```
alphabet: {a, b}
start: q0
accepting: {q0, q2}

q0 (a -> q1) (b -> q2)
q1 (a -> q1) (b -> q1)
q2 (a -> q0) (b -> q2)
```

Nondeterministic finite automata (NFA)

Expected format for NFA. A comma-separated list of characters, a single start state, and a comma-separated list of accepting states. A listed state should have one or more state transitions (`<char> -> <state>`).

```
alphabet: '{'<char> (',' <char>)*'}'
start: <state>
accepting: '{'<state> (',' <state>)*'}'
<state> ('('(<char> | '_') -> <state>')')+
```

where `_` represents the empty string.

Example:

```
alphabet: {a, b}
start: q0
accepting: {q3}

q0 (_ -> q1) (_ -> q2) (_ -> q3)
q1 (a -> q1) (a -> q2) (_ -> q2)
q2 (_ -> q3)
q3 (b -> q3)
```

Regular expressions (RE)

Regular expressions support the following operations:

```
Union: |
Kleene star: *
Intersection: &
Complement: -
```

Where `()` represents the empty string and `.` is a shorthand for all of the alphabet. Terminals defined in the `alphabet` set can be any character except the following: `|&-*(),{}`.

Example:

```
alphabet: {a, b}

(a(a | b) & ((a | b)(a | b))*) | ((aa)* & -(a*a*))
```

Context-free grammars

The expected syntax is one or more rules of the following form, one per line:

```
terminals: '{'<char> (',' <char>)*'}'
<char> -> (<char>+ | '_') ('|' (<char>+ | '_'))*
```

where _ represents the empty string, the first rule is assumed to be for the start symbol, and the nonterminals are assumed to be any character that appears on the left-hand side of a rule. On the right-hand side, terminals and non-terminals must be separated by spaces. Terminals and nonterminals should not include the following characters: `_|->`

Example:

```
terminals: {a, b}

S -> U | V
U -> T a U | T a T | U a T
V -> T b V | T b T | V b T
T -> a T b T | b T a T | _
```

Pushdown automata

Expected format for PDA.

```
start: <state>
accepting: '{'<state> (',' <state>)*'}'
<state> ('('(<char> | '_')','(<char> | '_') -> (<char>+ | '_')','<state>')')* ...
```

where _ represents the empty string and the parenthesized expression represents `(<input symbol>,<stack symbol> -> <symbols to push>,<nextstate>)`.

Example:

```
start: q0
accepting: {q0, q3}

q0 (_,_ -> $,q1)
q1 (0,_ -> 0,q1) (1,0 -> _,q2)
q2 (1,0 -> _,q2) (_,$ -> _,q3)
```