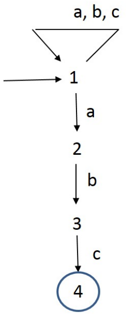
CS 301: Formal Languages and Functional Programming, Fall, 2016
Programming Assignment 3, due: December 2, 2016, 23:59

For this assignment, you will build a Scheme interpreter that will process strings as directed by a nondeterministic finite automaton. The work is divided into two parts.

1. The NFA will be represented as a list of lists. The first interior list will contain the accepting states, named with integers. For each state, there may be a subsequent list giving the available transitions from that state. Each of these transitions will itself be a list containing the source state as the first element and lists of destinations associated with a particular input string symbol. The special symbol `’eps` denotes a ϵ-transition. Here are two examples.

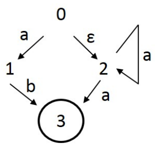
   The NFA below has the representation `’((4) (1 (a 1 2) (b 1) (c 1)) (2 (b 3)) (3 (c 4)))` and accepts strings ending in `abc`. That is, it accepts strings $(a+b+c)^*abc$.

   The transition diagram appears to the left. The accepting states are (4). The transitions from state 1 are (1 (a 1 2) (b 1) (c 1)). In particular, the transitions from state 1 available on an ’a’ are 1 and 2; on a ’b’ or a ’c’, only destination 2 is available. The transitions for state 2 are (2 (b 3)), and so forth. Note that state 4 has no transitions and is simply missing from the list. I found it convenient to construct the machine in stages:

   ```
   (define final ’(4))
   (define trans1a ’(a 1 2))
   (define trans1b ’(b 1))
   (define trans1c ’(c 1))
   (define trans2b ’(b 3))
   (define trans3c ’(c 4))
   (define trans1 (list 1 trans1a trans1b trans1c))
   (define trans2 (list 2 trans2b))
   (define trans3 (list 3 trans3c))
   (define term-abc (list final trans1 trans2 trans3))
   ```

   Machine term-abc is then `’((4) (1 (a 1 2) (b 1) (c 1)) (2 (b 3)) (3 (c 4)))`.

   A second example involves ϵ-transitions, indicated by `eps` in the definition. Its diagram appears to the left, with the single accepting state 3, from which it is clear that the machine accepts $ab + a^*a$. An incremental definition is

   ```
   (define tt0a ’(a 1))
   (define tt0eps ’(eps 2))
   (define tt1b ’(b 3))
   (define tt2a ’(a 2 3))
   (define f2 ’(3))
   (define mach2 (list f2 (list 0 tt0a tt0eps) (list 1 tt1b) (list 2 tt2a)))
   ```

   giving the complete definition `’((3) (0 (a 1) (eps 2)) (1 (b 3)) (2 (a 2 3)))`.

   The first part of this assignment is to write a Scheme function `transitions` that takes three arguments: a state, a symbol (or `’eps`), and a machine. It returns a list of possible destination states reachable from the argument state via the argument symbol. For example, the example machines described above deliver the following destinations.

   (a) `(transitions 1 ’a term-abc)` returns `’(1 2)`

   (b) `(transitions 2 ’b term-abc)` returns `’(3)`

   (c) `(transitions 2 ’a term-abc)` returns `’()`

   (d) `(transitions 3 ’eps term-abc)` returns `’()`

   (e) `(transitions 0 ’eps mach2)` returns `’(2)`

2. Notice that the machine structure in the preceding part does *not* contain a start state. For purposes of recursion, it is convenient to start the machine from various states, even though an accepting path from the true start state is necessary to accept a string.

   For the second part of the assignment, you are to construct two mutually recursive functions: `nfa-execute` and `backtracker`. The first takes three arguments: a string (a simple list of atomic literals), a start state, and a machine. The machine is a structure as defined in the preceding question, and the start state must be one of its states. `nfa-execute` attempts to find an accepting path from its given start state to an accepting state. If it can find such a path, it returns a list of nodes along that path. If no such path exists, it returns the null list. For example,

   `(nfa-execute '(b b a a c b a c b a b c), 1, term-abc)` returns `'(1 1 1 1 1 1 1 1 1 1 2 3 4)`.

   `(nfa-execute '(b b a a c b a c b a b b), 1, term-abc)` returns `'()`.

   In these examples, we call `nfa-execute` with the true start state, and therefore we can interpret the result as a yes/no verdict on the membership of the given string in the language represented by the machine. However, `nfa-execute` can seek an accepting path from any node, and this flexibility allows us to recursively search for an accepting path.

   `nfa-execute` achieves its goal by using the second function, `backtracker`, which in turn can call `nfa-execute`. Five arguments are necessary for a `backtracker` call: a string, a start state, a list of available destinations that can be reached by consuming the first string symbol, a list of available destinations that can be reached by a ϵ-transition, and finally a machine. To advance, `nfa-execute` extracts the destination lists with the `transition` function of the preceding problem and then call the backtracker.

   Suppose the call is `(backtracker string start trans epstrans machine)`. There are some initial checks. If `string` is null and `start` is an accepting state, then the backtracker return a singleton list containing `start`. If the string is not null and destinations are available in `trans`, the backtracker attempts to continue the path via `nfa-execute` on `(cdr string)`, `(car trans)`. If that fails, it tries a recursive call to itself using `string, start, (cdr trans), epstrans, machine`. If `trans` is null, it proceeds in a similar fashion with `epstrans`, using the full string, of course, since `epstrans` contains destination states reachable via a ϵ-transition. Backtracker exhausts all available possibilities from `trans` before trying those from `epstrans`. If none of these alternatives succeed, it returns the null list.

   Here are a few more examples using the machines from the preceding question.

```
> (nfa-execute '(b b a a c b a c b a b c) 1 term-abc)
'(1 1 1 1 1 1 1 1 1 1 2 3 4)
> (nfa-execute '(b b a a c b a c b a b b) 1 term-abc)
'()
> (nfa-execute '(a a a a a a a a a) 0 mach2)
'(0 2 2 2 2 2 2 2 2 2 3)
> (nfa-execute '(a b) 0 mach2)
'(0 1 3)
> (nfa-execute '(a b a a a b) 0 mach2)
'()
> (nfa-execute '(a) 0 mach2)
'(0 2 3)
> (nfa-execute '(b a) 0 mach2)
'()
```

   You may assume that there are no cycles in the machine in which all links are epsilon-transitions.