Homework #1: CMPT-379

Distributed on Tue, Sep 13; due on Tue, Sep 20

Anoop Sarkar – anoop@cs.sfu.ca

(1) This assignment is the first step towards a full lexical analyzer. You have to implement the *recognition* of an input string into component tokens as defined by a deterministic finite-state automaton (DFA) (also known as *simulating* the behaviour of the DFA on an input string). The following is pseudo-code for the simple recognition algorithm (this assumes only one token in the input).

```
algorithm dfa_recognize (string i)
// See Algorithm 3.1 in the Dragon book
{
   s = 0; // set s to the start state 0
   c = nextchar(i);
   while c != eof {
     s = move(s, c); // for this hw, this step has to be O(1) time
     c = nextchar(i);
   }
   if (s is a final state) { // for this hw, this step has to be O(1) time
     print "dfa: state=s token=i\n"; // this step assumes one token in input
   }
}
```

What is required: You have to submit a program written in C++ (or in ANSI C) that can be run as follows (notice the use of quotes to keep the semicolon safe from the shell):

```
dfa int-input.txt "int;"
```

and after reading the DFA stored in `int-input.txt` (corresponds to the DFA in Figure 2) it produces the following output tokenization:

```
dfa: state=3 token=int
dfa: state=4 token=;
```

In cases where the input is not recognized by the DFA, the program should report an error: `illegal token`.

```
dfa int-input.txt "sin"
illegal token
```

In addition, your program has to conform to the more detailed requirements as listed below.

a. Your program should read in a textual representation of a DFA from an input file and store it in a suitable data structure.

   The data structure used should be chosen so that the constraints mentioned in the pseudo-code above can be implemented. The constraints are: $O(1)$ (worst-case constant-time) access for the function `move(s, c)` and for checking whether a state is a final state.

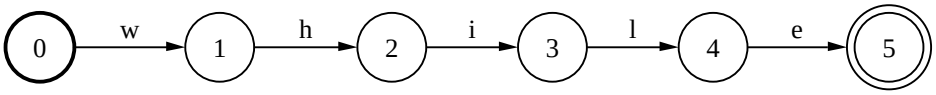

Figure 1: DFA for recognizing a single keyword `while`.

For example, for the DFA in Figure 1 the text representation (stored into the file `while-input.txt`) is:

```
0 1 w
1 2 h
2 3 i
3 4 l
4 5 e
5
```

`0` is always assumed to be the start state, and state `5` is the only final state defined. The format for each line is either: `fromstate`⟨*int*⟩ `tostate`⟨*int*⟩ `inputchar`⟨*char*⟩; or `finalstate`⟨*int*⟩. See the file `int-input.txt` on the web page for the representation for Figure 2.

b. Implement the DFA recognition algorithm described above (see Section 3.6 in the Dragon book). Your code should be able, in principle, to load multiple (i.e. more than one and different) DFAs from different input files. In other words, do not implement the DFA data structures as global variables. You can assume a maximum number of states if needed and you can assume that the input characters are always ASCII (see `man ascii` for details).

c. Extend the DFA recognition algorithm to the pattern matching algorithm for DFAs described in Section 3.8 from the Dragon book. You have to implement the following aspects of pattern matching:

- Find the longest match when matching tokens. e.g. the input string *int* should match the final state 3 with one token `int` assuming the DFA in Figure 2. You should **not** get three tokens: `i` with final state 1, and `n`, `t` each with final state 4.
- Once one token is found, reset to the start state and continue scanning until the input string has no more characters left to scan. More precisely, when no next state is available, the last final state found is reported, and the DFA is reset to the start state and scanning of the input string begins after the last character that was scanned before reaching the last final state found. e.g. the input string *int;* should match token `int` with final state 3, and token `;` with final state 4.

For now, ignore the idea from the Dragon book of picking a pattern with higher priority (since we don't have access to this information yet).

d. In order to make testing your code possible, provide a `makefile` that can be use to run tests by invoking `make test`. Refer to the course web page for more details on how to write makefiles and the conventions to be used in this course. This program should be invoked and produce output exactly as in the examples in the **What is required** section.

Hints on testing your code: Invent some tricky DFAs, write them down in the input file format (as shown above) and test your code. We will be testing your code on the two DFAs shown here, plus some additional DFAs not shown here that are designed to check for errors in your code.

Your code should be readable and simple. Include a few sensible comments especially when making assumptions in the code. This is a good place to set up code that you can use for the rest of the semester. This includes classes or functions for memory management (especially if you are using C instead of C++) and for error reporting.

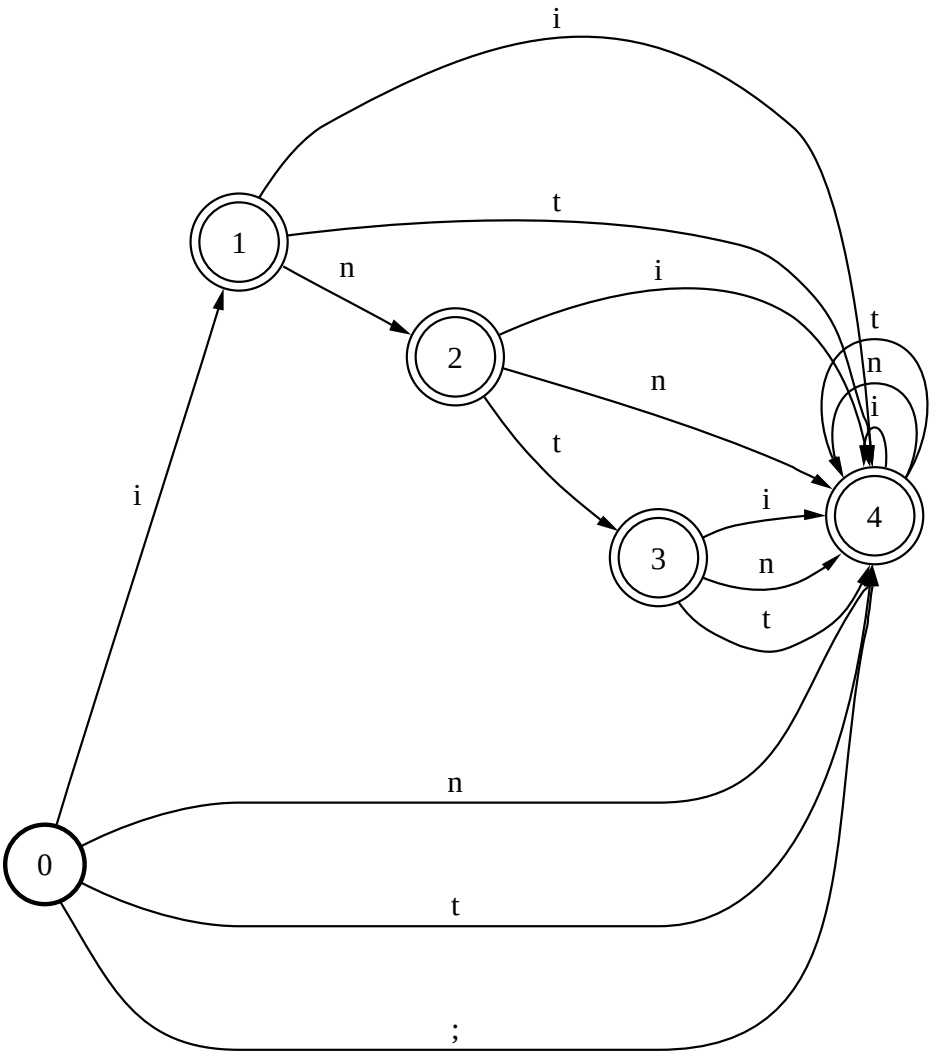

Figure 2: DFA for recognizing a keyword `int` (corresponds to state 3) or identifiers like `tnt` (corresponds to states 1, 2, 4).