Algorithms for AI and NLP (Fall 2008, Exercise 2)

Goals

1. Understand better the use of (nesting) lexical bindings and the functions `push()` and `pop()`;
2. design a datastructure and associated functions to read and manipulate FSAs;
3. experiment with alternate representations for the transition matrix, investigating efficiency.

1 Bring up the Editor and the Allegro Common Lisp Environment

- As always, start our integrated Lisp development environment, by typing (at the shell prompt):

  ```
  acl &
  ```

- For this session, we provide three files—'`sheep.lsp`', '`brown.lsp`', and '`krugman.txt`'—which contain textual representations of two FSAs (see below), as well as a random sample of text taken from the New York Times web page. Our course setup should have copied the file '`sheep.lsp`' into your IFI home directory already, and the other two will also occur there before the end of this week. We recommend that you create a new file, say '`exercise2.lsp`', and start developing your code there. That is, rather than typing directly into the Lisp prompt (through the emacs '`*common-lisp*`' buffer), put all your code into '`exercise2.lsp`' from the beginning. You can then use emacs commands like 'M-C-x' (evaluates the top-level s-expression currently containing the cursor) or 'C-c C-b' (evaluates the entire buffer) to interactively load your code into the Lisp system; remember that, after each change you make in the file '`exercise2.lsp`', you need to re-evaluate the code before your changes will take effect. For testing purposes, we still recommend you use the '`*common-lisp*`' Lisp prompt.

2 Local Variables and Lexical Binding

(a) Consider the following definitions for two global variables and one function.

```
? (defparameter foo 4) → foo
? (defparameter bar 2) → bar

? (defun mystery (foo)
    (format t "~a~%" foo)
    (format t "~a~%" bar)
    (let ((foo 2)
          (bar (+ foo 10)))
      (format t "~a~%" foo)
      (format t "~a~%" bar)
      (let* ((foo 3)
             (bar (+ foo 10)))
        (format t "~a~%" foo)
        (format t "~a~%" bar))))
→ mystery
```

Explain the print-out that results from each of the `format()` calls in the body of `mystery()` when evaluating '`(mystery 42)`'. Maintain a table to keep track of the values for *foo* and *bar* at each step.

(b) The built-in Common-Lisp *special forms* `push()` and `pop()` manipulate lists with a stack semantics (last-in, first-out), e.g.

```
? (defparameter *stack* '(1 2 3)) → *stack*
? *stack* → (1 2 3)
? (push 0 *stack*) → (0 1 2 3)
? *stack* → (0 1 2 3)
? (pop *stack*) → 0
? *stack* → (1 2 3)
```

Experiment with `push()` and `pop()` and consider writing functions that have the exact same behaviour. Why is it *impossible* to (re-)implement `push()` and `pop()` as functions?

3 Implementing and Reading FSAs

(a) To represent FSAs in our code, we will implement an abstract data type. Define a structure *fsa*, with components *vocabulary*, *transitions*, *final*, and *size*. We will use the *vocabulary* component to record—as a set of symbols (implemented as a list)—the legitimate input symbols for our machine. The *transitions* component in *fsa* will contain a representation of the transition matrix, and we will experiment with different ways of encoding that information in Lisp. The *final* component will hold a set of integers (again implemented as a list), indicating which of the states are final (i.e. accepting) states. Finally, the value of *size* will be an integer, indicating the total number of states in the machine. To encode the states themselves, we will use integers, consecutively numbered, starting from zero. By convention, the initial state of our machines will always be numbered zero.

(b) Take a look at the file '`sheep.lsp`' (provided through our course setup) and make sure you understand how the information in the file corresponds to the 'sheep talk' example of Jurafsky & Martin (2008). To get started, we will use a very simple-minded representation for the transition matrix, viz. as a simple set of transitions (once more implemented as a list), each in the exact same form as we use in '`sheep.lsp`'.

Write a function `read-fsa()` that takes one argument, a file name corresponding to a textual FSA representation; the function should read s-expressions from the file until it reaches the end. It needs to distinguish two types of s-expressions, lists and integers (corresponding to transitions or final states, respectively); `read-fsa()` constructs an *fsa* instance and fills in the components of the structure with information read from the file. For example, constructing a machine from '`sheep.lsp`' should look somewhat like this:

```
? (setf fsa (read-fsa "~/sheep.lsp"))
→ #S(FSA :VOCABULARY (! A B)
         :TRANSITIONS ((2 A 3) (3 ! 4) (3 A 3) (1 A 2) (0 B 1))
         :FINAL (4) :SIZE 5)
```

Note that `read-fsa()` needs to determine the data type of each s-expression that it reads from the file, which can be either a *list* or an *integer*. We have seen the predicate to test whether an object is a list already; see whether you can guess (or find) the name of the predicate that returns true for integers. Further note that, when using lists to represent sets, there is a convenient variant of `push()` called `pushnew()`, which will avoid creating duplicate elements. Thus, the following equivalence holds:

$$\texttt{(pushnew } X \texttt{ } Y\texttt{)} \equiv \texttt{(unless (member } X \texttt{ } Y\texttt{) (push } X \texttt{ } Y\texttt{))}$$

Finally, note that the Lisp reader—when asked to read an s-expression from a stream—will ignore comments, i.e. all lines in our FSA files starting with a semikolon ('`;`'). Hence, you need not do anything special to read over those comments.

(c) To practice (among other things) the use of the *fsa* structure, the mighty `loop()` facility, and beautifully formatted output, write a function that prints out the transition matrix, one line per state, where states occur in numeric order. That is:

```
? (print-fsa fsa)
→ 0: 'B' -> 1
  1: 'A' -> 2
  2: 'A' -> 3
  3: 'A' -> 3, '!' -> 4
  4: [final]
```

Note that our current internal representation of the transition matrix is not organized in any particular order. But `print-fsa()` needs to output all outgoing transitions from a given state on a single line, and then move on to the numerically following state. One option would be to pre-process the *transitions* elements inside of `print-fsa()`, i.e. go through the list once—before producing any printed output—and prepare an auxiliary 'index' (local to `print-fsa()`): for each state in the machine (represented as an

integer), the index should contain all transitions originating at that state. What would be a suitable data structure to represent the index? Another option would be to impose an ordering on the *transitions* component in our machine, i.e. make sure that its elements are numerically sorted, according to the state from which they originate (i.e. the `first()` of each transition).

(d) Next, write a function `next-states()` to return the set of possible states to transition into, given a machine, the current state, and one input symbol; e.g.

```
? (next-states fsa 3 'a)
→ (3)
? (next-states fsa 3 'b)
→ nil
```

(e) Finally, translate the function `nd-recognize()` from our initial lecture on FSAs (see the slide copies) into Common Lisp. Here are some example calls:

```
? (nd-recognize fsa '(b a a a a a a a a a !))
→ t
? (nd-recognize fsa '(b a a a a a a a a a a))
→ nil
? (nd-recognize fsa '(b a a a a a a a a a a h !))
→ nil
```

Note that, to retrieve the next input symbol (at position *index*, using the naming conventions from our slides) we recommend the built-in Lisp function `elt()`. Please look up the definition of `elt()` and observe how it is very similar to `nth()`. So, what is the difference between the two functions, and why do you think we recommend `elt()` rather than `nth()`?

(f) To test our implementation of `nd-recognize()` on a machine that actually is non-deterministic, write down an FSA that recognizes the language {aaa, aaaa, aaaaaa, aaaaaaaa, aaaaaaaaa, ...}. This language has a singleton vocabulary (containing only the symbol 'a'), and it contains sequences of n repetitions of 'a', where n is either a multiple of three or of four. Consider sketching the machine as a state-and-transition diagram, using 'pencil and paper' first. Then read off the transition matrix from your sketch, and create a file '`as.lsp`' containing the definition of the automaton—using the same notation we used for '`sheep.lsp`' above. Read the machine from your file and test it on several inputs; make sure everything works as it should. Please include the testing calls you used in what you submit.

4 Implementing and Reading FSAs

(a) Our current implementation of the transition matrix is somewhat inefficient. Assuming you implemented `next-states()` as a `loop()`, how many iterations are there for each call to `next-states()`? And in case you wrote `next-states()` as a recursive function, how often does it call itself recursively for each top-level entry into the function? Thinking back to what we had to do in printing our machines, say in a sentence or two how the efficiency of our current FSA implementation could likely be improved.

(b) Write a function `symbol-code()` that, given a machine and one input symbol, maps each symbol from the vocabulary of the machine to an integer between 0 and $n-1$, where n is the size of the vocabulary of the machine. For invalid input symbols, `symbol-code()` returns `nil`. For example:

```
? (symbol-code fsa 'b)
→ 2
? (symbol-code fsa 'h)
→ nil
```

(c) Now that we have a way of 'projecting' our input symbols into consecutive integers, write a function to 'compile' a machine, converting our original representation of the transition matrix into an actual matrix, i.e. a Lisp array. How many dimensions do we need in our array, and how many cells along each dimension? The function `compile-fsa()` takes as its sole input a machine, converts its transition matrix into an array, and then destructively changes the *transitions* component of its argument. This is the kind of function that is mostly interesting for its side-effect, so it does not matter what it returns as its value; but it may be convenient to let it return the newly created array, i.e. the compiled transition matrix. For example:

```
? (compile-fsa fsa)
→ #2A((NIL NIL (1)) (NIL (2) NIL) (NIL (3) NIL) ((4) (3) NIL) (NIL NIL NIL))
```

(d) What remains to be done is adapt `next-states()` to operate on the new internal representation of the transition matrix. Make the necessary changes to `next-states()` and then decide whether we also need to make changes to `nd-recognize()`; if so, apply those changes.

Go back to our earlier examples (sheep talk and the silly '`a`' language) and make sure your revised implementation still obtains all the correct results. If not, debug your new code and repeat the testing until all is 'hunky and dory' (that is, correct). Celebrate appropriately.

To actually observe differences in the speed of execution, we will need a slightly larger automaton. The file '`brown.lsp`' contains a machine that will accept all unique word types from (our abbreviated version of) the Brown Corpus. Make sure that you are using compiled code, i.e. use one of the commands from the Allegro – emacs interface to compile and load your complete '`exercise2.lsp`'. Or simply use the Allegro top-level command '`:cl`' to accomplish the same; e.g.

```
? :cl exercise2.lsp
⇝ ;;; Compiling file exercise2.lsp
⇝ ;;; Writing fasl file exercise2.fasl
⇝ ;;; Fasl write complete
⇝ ; Fast loading exercise2.fasl
```

As in earlier examples, observe that we use the ⇝ symbol to indicate printed output to the terminal, *not* indicating the result of evaluation. In fact, the Allegro top-level commands (all starting with a colon, i.e. all named by keyword symbols) are meta-level instructions to the Lisp system rather than s-expressions to be evaluated. Hence, they can deviate from standard Lisp syntax ('`:cl exercise2.lsp`' is not a well-formed s-expression), and they are not evaluated in the standard sense.

Read the FSA from '`brown.lsp`' and make sure you can apply it sucessfully. Now take a peek at the additional file '`krugman.txt`', which contains one recent article from the New York Times (NYT), conveniently presented to us in plain text form, with one word per line. Similar to our corpus experiments in the first problem set, read the article into a list of strings, i.e. a list whose elements are the individual words from the file '`krugman.txt`'.

Finally, use the FSA recognizing the most frequent words in the Brown Corpus to identify the *new* words in the NYT article. To apply our FSA implementation to strings (rather than to lists of symbols, as we have done so far), which changes—if any—need to be made to the code? How many new words are there? Can you see a difference in using the FSA with the original, naïve internal representation of the transition matrix, as opposed to our compiled variant? Note that Lisp provides an operator `time()`, which takes as its argument an s-expression: `time()` evaluates its argument and afterwards reports on cpu time and memory usage. For example:

```
? (let ((aaa (loop repeat 1000 collect 'a)))
    (time (nd-recognize fsa (append '(b) aaa '(!)))))
⇝ ; cpu time (non-gc) 220 msec user, 10 msec system
⇝ ; cpu time (gc)     0 msec user, 0 msec system
⇝ ; cpu time (total)  220 msec user, 10 msec system
⇝ ; real time  235 msec
⇝ ; space allocation:
⇝ ;  677,751 cons cells, 7,410,864 other bytes, 0 static bytes
→ t
```

5 Submitting Your Results

Please submit your results in email to Stephan ('`oe@ifi.uio.no`') and Lars ('`larsbun@ifi.uio.no`') before 12:00 noon on Tuesday, October 7. Please provide all files that you created as part of this exercise (e.g. minimally '`exercise2.lsp`' and '`as.lsp`', if you folllowed our suggestion above), including all code and answers to the questions above. Note that it is good practice to generously document your code with comments (using the '`;`' character, making the Lisp system ignore everything that follows the semicolon).