CS 314 – Principles of Programming Languages

Spring, 2017

Practice Midterm Exam 1

- **Do open this exam** until everyone has an exam and the instructor tells you to begin.
- There are 6 pages in this exam, including this one. Make sure you have them all.
- This exam is closed book – closed notes – closed electronics..
- You must put your cellphone, tablet, computer, Ipod, or other electronic devices in a backpack, etc, and leave it out of reach. The only exception is that you can use a watch that only has time-related functions (e.g. not a calculator watch, not a "smart watch").
- Write clearly – if we can't read or can't find your answer your, answer is wrong.
- Make clear what is your answer versus intermediate work.

I. Suppose we have an email message whose subject is the string "Hi". In other words, the subject is a string containing H and i. Let us say a subject is related to this one if it is "Hi" with "Re: " appended to the front **zero** or more times. The R and e can be in any capitalization (R or r, E or e) , the : must immediately follow the E or e, and the : must be followed by **one** or more spaces. E.g. the following are related to Hi, where ␣ indicates a space:

```
Hi
RE:␣Hi
Re:␣␣rE:␣␣␣Re:␣␣Hi
```

but the following are not:

`Re␣:␣Hi` (space before the :)

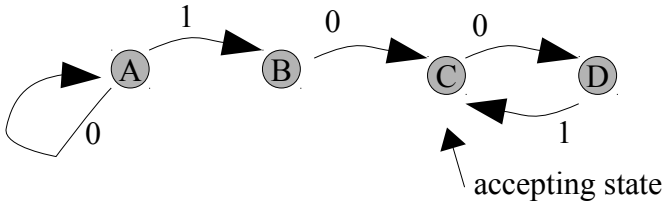
`Re:Hi` (no space after the :)

The alphabet is {`H, i, r, R, e, E, :, ␣`}.

A. Draw a deterministic finite state machine that recognizes strings related to "Hi". Be sure to indicate which state is the start state and which state(s) are accepting.

B. Write a regular expression which represents the set of strings related to "Hi".

II. Consider the Finite State Automaton (FSA) defined by the following diagram:



State A is the start state and state C is the only accepting state.

A. Is this FSA deterministic or non-deterministic? Why?

B. For each if the following strings, circle Yes if it is accepted by the FSA above and No if it is not:

i. 0 0 0 0 1	Yes	No
ii. 1 0 0 1	Yes	No
iii. 0 1 0 0 1	Yes	No
iv. 1 0 0 1 0 1	Yes	No
v. 1 0 0 0 0 1	Yes	No

III. Consider the following grammar, G1: (Terminals are underlined.)

S => S @ Term | Term
Term => Term % Factor | Factor
Factor => p | q | r | (S)

A. Draw a parse tree to show that
p @ (q)
is in the language of G1

B. Add parentheses to the following to show the order of operations implied by G1.

i. p % q % r

ii. p % q @ r

Write scheme functions for each of the following. You may write additional helper functions if you wish. All repetition must be done by recursion (including the recursion implicit in functions like map). You may **not** use `do` or any function whose name ends in '!', e.g. not `set!`.

IV. (sequence first last) returns a list of the integers first, first+1, etc, through last. E.g. (sequence 4 7) returns (4 5 6 7). If first > last, sequence returns the empty list. Write sequence.

(define (sequence first last)

V. Write the function for-n **using map and the function (sequence first last)** specified in the problem above. for-n takes three arguments: start and stop, which are numbers, and fn which is a function of one argument. for-n calls fn several times, first with the argument start, then with start+1 then ... finally with stop. If start>stop, for-n simply returns the empty list without doing any calls to fn. for-n returns a list of the values that the calls to fn return.

(define (for-n start stop fn)

VI. The function (count x lst) takes arguments x, a symbol, and lst, a list, and returns the number of times x appears at the top level in lst. E.g., (count 'a '(a (b a) c a)) returns 2, since the a in (b a) is not at the top level. **count and any helper functions must be tail-recursive.**

(define (count x lst)

VII. Write the function (max-on-ints n fn), which returns the integer k in the range $0 \leq k \leq n$ for which (fn k) is largest. You may not make any assumptions about the values of (fn k), e.g., do **not** assume they are positive.

VIII. Suppose you have a version of Scheme that does not implement the special form 'let' and you want to write your own definition of let. You want to have the same syntax as we discussed in class:

(let ((<variable1> <expression1>)
. . .
(<variablen> <expressionn>))
<expression>)

and the same semantics. Write a Scheme macro to implement let by translating it into an equivalent use of lambda. Do not use let in your translation. E.g,

```
(let ((a (/ 1 2))          should translate to  ((lambda (a b)
      (b (* 3 4)))                                 (+ a b))
  (+ a b))                                      (/ 1 2)
                                                (* 3 4))
```

Finish the implementation below:

```
(define-syntax let
    (syntax-rules ( )
```