Name _____

NetID _____

CS 314 – Principles of Programming Languages

Spring, 2016

Midterm Exam 1 - Answers

- Please write your name and net ID readably above.
- **Do not open this exam** until everyone has an exam and the instructor tells you to begin.
- There are 4 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, 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, your answer is wrong.
- Make clear what is your answer versus intermediate work.

I		/20
II		/20
III		/20
IV		/20
V		/20
VI		/20
Total		/120

I. Suppose we define an "id-line" as a name, followed by '%', followed by a domain, where a "name" is a string of one or more letters and digits and a "domain" is a sequence of one or more names separated by dots.

For instance, the following strings are id-lines:

1 9 c h a p l i n 2 0 % f i l m s . f u n n y . h a h a
a d o g % p e t s

but these are not (*b* indicates a space character):

c a t %	*(no domain)*
c a t % m e o w .	*(no name after the dot)*
d o g % p e t b i t	*(space is not a letter or a digit)*
c & d % f i g h t	*(& is not a letter or a digit)*

Finish the following grammar so that it defines the language id-lines. Be sure to underline terminal symbols. You may use symbols { } and [ ] from EBNF if you want to. You may add extra lines at the bottom if you want to.

ID-LINE → NAME % DOMAIN

NAME → ALPHANUM | NAME ALPHANUM

DOMAIN → NAME | NAME . DOMAIN

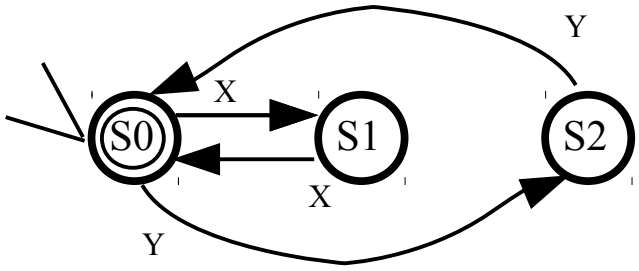
ALPHANUM → a | b | … | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

II. Define a "simple name" as a sequence of one or more '**a**'s and/or '**b**'s and define a "simple domain" as a sequence of one or more simple names separated by '**.**'s. Write a Regular Expression that defines the language of simple domains. Note that these are not id-lines and so will not have any '%' characters.

(a | b)+ (. (a | b)+)* or many equivalent REs

III.

Consider the FA defined by the diagram below:



State S0 is the start state and also the only accepting state.

A. Is this FSA deterministic or non-deterministic? Why?
deterministic: no epsilon moves and for every state/input char pair there is at most one next state

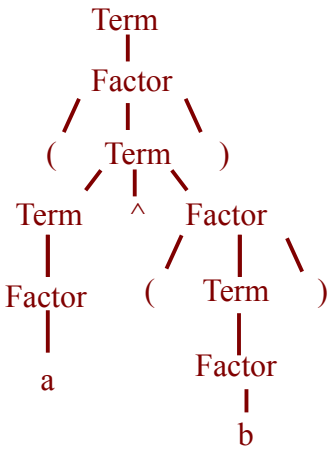
B. For each if the following strings, circle Yes if it is accepted by the FSA above and No if it is not:

i. X X Y Y	Yes	No
ii. Y Y X X	Yes	No
iii. X Y Y X	Yes	No
iv. Y X X Y	Yes	No
v. X X X X	Yes	No

III. Consider the following grammar, G1:
(Terminals are underlined. Term is the start symbol.)

Term => Term v Term | Term ^ Factor | Factor
Factor => a | b | c | (Term)

A. Draw a parse tree to show that
(a ^ (b))
is in the language of G1



B. Show a string that is in the language of G1 and whose parse is ambiguous.
(You do **not** have to show any parse trees, just the string,)

a v b v c (and many others)

For the Scheme questions below, all repetition must be done by recursion (including the recursion implicit in functions like map and assoc). You **may** write and use additional functions if you wish. You may **not** use `do` or any function whose name ends in '!', e.g. you may not use `set!`. You **may** use any other function in R5RS Scheme including the following:

Expression	Value		Expression	Value
(map sqrt '(9 1 4))	(3 1 2)		(reverse '(a (b c) d))	(d (b c) a)
(member 'a '(b c a d a))	(a d a)		(list (+ 2 3) '(a))	(5 (a))
(member 'x '(b c a d))	#f		(cons (+ 2 3) '(a))	(5 a)
(assoc 'x '((a b) (x y)(q r)))	(x y)		(append '(a b) '(c d))	(a b c d)

(null? x) is true if x is the empty list ( ), (eq? x y) is like Java's x = = y,

(- x y) is x – y, similarly for + and *

Many equivalent answers for the following questions

IV. Define the function n-repeats. (n-repeats x n) returns a list of length n, all of whose elements are x. E.g., (n-repeats '(a b) 3) should return ((a b)(a b)(a b)).

```
(define (n-repeats x n)
  (if (= n 0) '( )
      (cons x (n-repeats x (- n 1)))))
```

Define the function fact-tr. (fact-tr n) is n factorial. Fact-tr and any helper functions **must be tail-recursive** if they are recursive at all.

```
(define (fact-tr n)
    (fact-tr-h n 1))
(define (fact-tr-h n so-far)
 (if (= n 0)
    so-far
    (fact-tr-h (- n 1)(* so-far  n))))
```

Define (map-pairs fn lst), where fn is a 2-argument function, e.g. +, and lst is a list of 2-element sub-lists, e.g., ((1 2) (3 4)(5 6)). map-pair calls the function on each sub-list and returns a list of the results. E.g., (map-pairs + '((1 2)(3 4)(5 6))) returns the list (3 7 11). Note that there are correct answers that call map and also those that do not. Either way is ok. NOT calling map is probably easier.

```
(define (map-pairs fn lst)
    (if (null? lst) '( )                                    ...
        (cons (fn (caar lst) (cadar lst))  ◀– OR –▶ (cons (apply fn (car lst))
              (map-pairs1 fn (cdr lst)))))                  ...
OR  (define (map-pairs2 fn lst)
      (map fn
           (map car lst)
           (map cadr lst)))
```