

Examination 1
ENGR3520 Foundations of Computer Science
Fall 2016

You may take this exam at a time and place of your convenience. However, you should treat this exam as a (self-administered) **closed book** in-class examination rather than a take-home; it is intended to be completed in single sitting lasting approximately **1.5 hours**, though you may want to leave yourself extra time in case we misjudged. You may give yourself additional time, within reason, to complete the work, but you should not take the exam over several sessions or use more than about four hours. Complete the exam and **turn it in via GitHub** (scanned if done on paper) before class on Monday 17 October. **Please also bring a hard copy (original or printed) of the exam to class on that day.**

You may complete this exam electronically (e.g. as a word document or pdf) or by hand on paper. If you work on paper, please use a dark pen or some other method that will scan/copy legibly. Pencil has not copied well in the past.

During this exam, **you should not consult with other people, books, notes, internet, or other resources**. You should turn off any electronic devices that may provide a distraction. The only exceptions are: (1) you may use a computer to type your answers, and (2) you may, if you wish, open JFLAP **solely** to use it as a drawing program. Other than the instructors, you should not discuss this exam with anyone until class on 17 October.

If there is something unclear about the exam, you may try to contact the instructors, but in the interim should simply make a reasonable assumption and document that assumption in writing on your exam.

After you have completed the exam, **please copy the following statement onto the final page, filling in the appropriate times and dates, and sign your name:**

I began this exam at <fill in time and date> and completed it at <fill in time and date>. In taking this exam, I have behaved in accordance with the Olin honor code. In particular, I have neither given nor received unauthorized assistance during the completion of this work. I agree not to discuss this exam in any way until 1:30pm on Monday 17 October.

If you cannot write out this phrase and sign your name to it, please explain. [1]

Your exam should be turned in **via GitHub before class on Monday 17 October**. If you complete the exam on paper, you should take advantage of the handy dandy scan-to-email features of the Olin copiers or similar means. In either case, please **bring a hard copy of your exam to class on Monday 17 October**.

DO NOT TURN THIS PAGE UNTIL YOU ARE READY TO BEGIN

[1] This text courtesy of Professor Sarah Spence Adams. It is merely meant to affirm our shared understanding of the context in which you are taking the exam. Thanks for obliging!

(this page intentionally left blank
to facilitate double-sided printing)

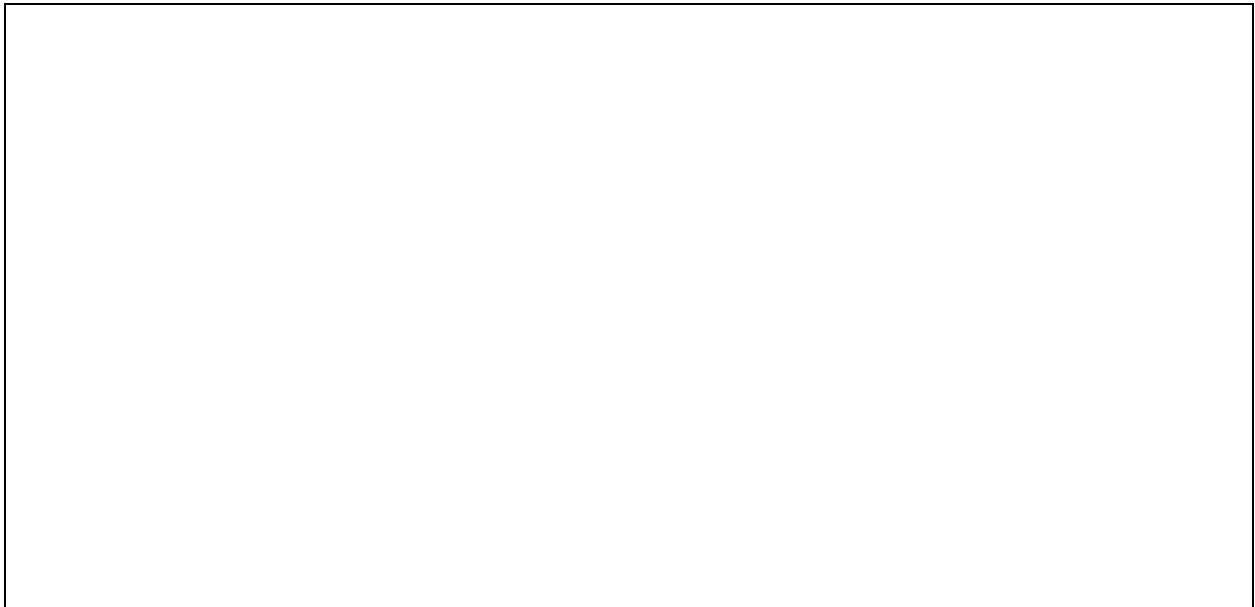
FOCS Fall 2016 Exam #1

Time exam begun: _____ on _____ (date)

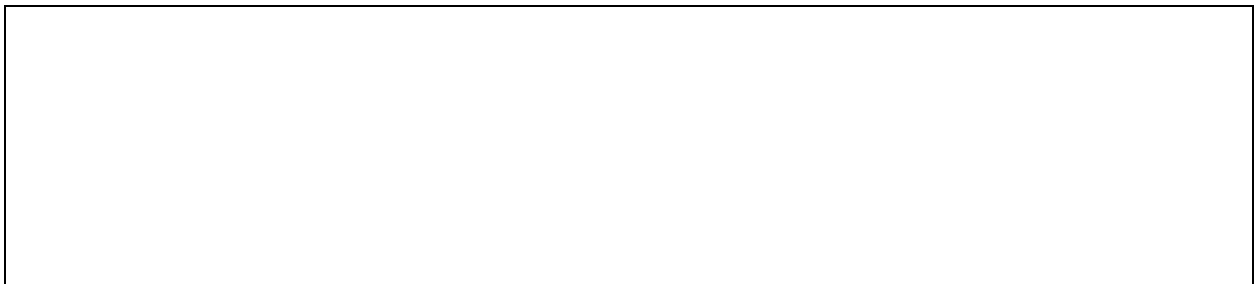
Regular Expressions

Consider the language.... **0(01)*1**

1. [5 points] Produce a finite automaton for this language. Make sure that you label/number your states.

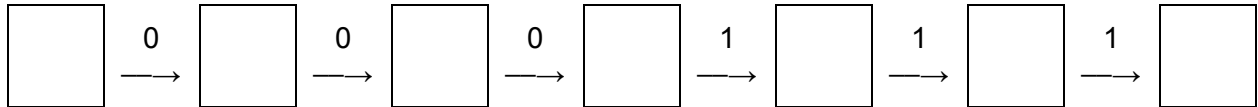


2. [3 points] Is your FSA deterministic or nondeterministic? Why (i.e., explain/justify your answer)?



Run your FSA on the string **000111**

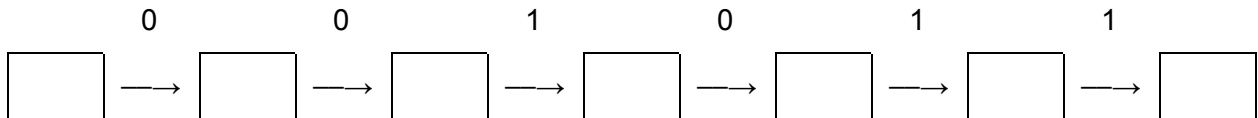
3. [4 points] What is the sequence of states that your automaton passes through when operating on this string? Use the boxes below to indicate how the automaton transitions, with the start configuration in the first box, etc., using the labels/numbers that you selected for your states.



4. [2 points] Does your automaton accept or reject the string?

Now run your FSA on the string **001011**

5. [4 points] What is the sequence of states that your automaton passes through when operating on this string?



6. [2 points] Does your automaton accept or reject the string?

Consider the language L of all strings over $\{0,1\}$ that contain an even number of 0's, and where no two 0's are adjacent.

7. [6 points] The following string is in L :

0 1 0 1 1 0 1 1 1 0

Identify – by circling it in the string above – a non-empty substring which can be removed, or repeated any number of times, each time producing a string that is also in L .

Example: For the language 10^*1 and the string 1001 , the following would be correct answers:

1 0 0 1

1 0 0 1

1 0 0 1

However, this would not be correct:

✖ 1 0 0 1

8. [5 points] Does the existence of this substring prove that the language above is regular? Why or why not?

Propositional Logic

9. [10 points] Using the provided rules of inference and substitution, write a formal proof for the following:

Assume: $(p \text{ AND } q) \text{ OR } (p \text{ AND } r) \text{ OR } ((\text{NOT } p) \text{ AND } (q \text{ OR } r))$

Prove: $q \text{ OR } r$

[The rules are attached to the end of this exam and are also the same as those distributed in class and used for hw11.]

(add lines as needed)

1. $(p \wedge q) \vee (p \wedge r) \vee ((\neg p) \wedge (q \vee r))$

assumption

2.

3.

10. [10 points] Use truth tables -- considering all possible assignments of truth values to the propositions p , q , and r -- to demonstrate that the assumption $[(p \wedge q) \vee (p \wedge r) \vee ((\neg p) \wedge (q \vee r))]$ and conclusion $[q \vee r]$ above are actually equivalent.

p	q	r	$q \vee r$... (add columns as needed)	$(p \wedge q) \vee (p \wedge r) \vee ((\neg p) \wedge (q \vee r))$

...(add rows as needed)

Scheme / Recursion

11. [12 points] For each of the following pieces of scheme code, indicate (in the table below) whether it is:

- Purely functional (does not contain assignment or loops)
- Recursive (calls itself)
- Tail recursive (does not depend on accumulated stack frames to execute)

```

1 (define (accumulate lst fn base)
2   (if (empty? lst)
3       base
4       (fn (first lst) (accumulate (rest lst) fn base))))
5
6 (define (reverse lst)
7   (rev lst '()))
8
9 (define (rev lst new)
10  (if (empty? lst)
11      new
12      (rev (rest lst) (cons (first lst) new))))
13
14 def fact(n):
15   if n == 1:
16     return 1
17   else:
18     return n * fact(n-1)

```

For each function above, indicate which apply (Y) and which do not (N).

Function	Purely functional?	Recursive?	Tail recursive?
accumulate (as defined above)	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No
reverse (as defined above, not including rev)	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No
rev (as defined above)	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No
fact (as defined above)	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No

Scheme Interpreter

Consider the following code for a (pseudo-)scheme interpreter.

[A full version is provided in day4.rkt and attached to the end of this exam; this is just the core]

```

00
01 (define (evaluate expr env)
02   (cond [(null? expr) expr]                ;; Base cases: self-evaluating empty list,
03         [(number? expr) expr]              ;; self-evaluating numbers
04         [(boolean? expr) expr]             ;; and booleans;
05         [(symbol? expr) (lookup-variable expr env)] ;; Symbols (names) need looking up.
06
07         ;; Otherwise the expression begins with (. Check special forms first:
08
09         ;; IPH is a conditional
10         [(eq? (first expr) 'IPH) (if (evaluate (second expr) env) ;; if the TEST is true
11                                     (evaluate (third expr) env)   ;; evaluate the THEN
12                                     (evaluate (fourth expr) env))]  ;; otherwise the ELSE
13
14         ;; DEFINE causes us to evaluate in a different environment (with an added binding)
15         [(eq? (first expr) 'DEFINE) (repl (cons (list (second expr) ;; invoke REPL w/
16                                                    (evaluate (third expr) env)) ;; extended env
17                                                    env))]                ;; (now includes this def)
18
19         ;; QUOTE doesn't evaluate its argument
20         [(eq? (first expr) 'QUOTE) (second expr)]
21
22         ;; LAMBDA creates a closure -- the procedure parameters, body, and current environment
23         [(eq? (first expr) 'LAMBDA) (list 'CLOSURE (second expr) (third expr) env)]
24
25         ;; Otherwise it's a "regular" procedure -- built-in or lambda. Evaluate it and all args,
26         ;; then apply the value of the procedure to the value of the arguments.
27         [(list? expr) (apply-proc (evaluate (first expr) env)
28                                   (map (lambda (expr) (evaluate expr env))
29                                       (rest expr)))]
29
30         ;; Otherwise oops!
31         [else (error "evaluate: not sure what to do with expr" expr)])
32

```

Note that question #13, below, asks you to modify the code above. You should indicate where your modified code goes using the provided line numbers.

Answer the following questions, referring to `evaluate` as necessary:

12. [6 points] We say that something is a “special form” if it does not follow the ordinary rules of evaluation. Why is `LAMBDA` a special form? You may refer to the code in your explanation, but it is not necessary that you do so.

13. [10 points] Add to the above code a clause to handle a new form: `UNLESS`.

`UNLESS` should take two sub-clauses: a condition and an action.

If the condition is false, `UNLESS` should evaluate its action.

[If the condition is true, it doesn't matter what `UNLESS` does. Some simple choices might be: the value of the condition; `#f`; ``()`]

Here are some examples of `UNLESS` in use:

```
(UNLESS (= x 0)
  (/ 1 x)) ;; returns 1/x unless x is 0

(UNLESS (empty? lst)
  (rest lst)) ;; returns (rest lst)
              ;; unless lst is already empty
```

You may write the code for `UNLESS` below. Also indicate the line numbers where you would insert your code

```
;; insert between line _____ and line _____
```

Context-free grammars

Consider the language

$$L1 = \{0(01)^a1(10)^b, \text{ where } a = b\}$$

For example, these strings are in L1:

0 1
0 0 1 1 1 0
0 0 1 0 1 1 1 0 1 0

These are not:

0 1 1 1 0
0 0 1 1 1 0 1 0

14. [6 points] Construct a context-free grammar for L1.

Consider the grammar G1:

- (1) $S \rightarrow 0T$
- (2) $S \rightarrow 00T$
- (3) $T \rightarrow 1S$
- (4) $T \rightarrow \varepsilon$

15. [5 points] Which of these strings are in G1's language?

	In G1's language	NOT in G1's Language
0	<input type="checkbox"/>	<input type="checkbox"/>
01	<input type="checkbox"/>	<input type="checkbox"/>
001	<input type="checkbox"/>	<input type="checkbox"/>
000	<input type="checkbox"/>	<input type="checkbox"/>
0101	<input type="checkbox"/>	<input type="checkbox"/>

16. [10 points] Find a string in G1's language that is not listed in the previous question. Show a left derivation for this string, and draw its parse tree.

After you have completed the exam, ***please copy the honor code statement from the instructions onto this page, filling in the appropriate times and dates, and sign your name.*** If you are filling out this exam electronically, your typed name will be accepted as a proxy for your signature.

Time exam completed: _____ on _____ (date)

(this page intentionally left blank
to facilitate double-sided printing
and attachment removal)

Table of Logical Equivalences

Commutative	$p \wedge q \iff q \wedge p$	$p \vee q \iff q \vee p$
Associative	$(p \wedge q) \wedge r \iff p \wedge (q \wedge r)$	$(p \vee q) \vee r \iff p \vee (q \vee r)$
Distributive	$p \wedge (q \vee r) \iff (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \iff (p \vee q) \wedge (p \vee r)$
Identity	$p \wedge T \iff p$	$p \vee F \iff p$
Negation	$p \vee \sim p \iff T$	$p \wedge \sim p \iff F$
Double Negative	$\sim(\sim p) \iff p$	
Idempotent	$p \wedge p \iff p$	$p \vee p \iff p$
Universal Bound	$p \vee T \iff T$	$p \wedge F \iff F$
De Morgan's	$\sim(p \wedge q) \iff (\sim p) \vee (\sim q)$	$\sim(p \vee q) \iff (\sim p) \wedge (\sim q)$
Absorption	$p \vee (p \wedge q) \iff p$	$p \wedge (p \vee q) \iff p$
Conditional	$(p \implies q) \iff (\sim p \vee q)$	$\sim(p \implies q) \iff (p \wedge \sim q)$

Rules of Inference

Modus Ponens	$p \implies q$ p $\therefore q$	Modus Tollens	$p \implies q$ $\sim q$ $\therefore \sim p$
Elimination	$p \vee q$ $\sim q$ $\therefore p$	Transitivity	$p \implies q$ $q \implies r$ $\therefore p \implies r$
Generalization	$p \implies p \vee q$ $q \implies p \vee q$	Specialization	$p \wedge q \implies p$ $p \wedge q \implies q$
Conjunction	p q $\therefore p \wedge q$	Contradiction Rule	$\sim p \implies F$ $\therefore p$

(this page intentionally left blank
to facilitate double-sided printing
and attachment removal)


```
#lang racket
```

```
(define operator-association-list
  (list (list 'ADD +)
        (list 'SUB -)
        (list 'MUL *)
        (list 'DIV /)
        (list 'GT >)
        (list 'LT <)
        (list 'GE >=)
        (list 'LE <=)
        (list 'EQ =)
        (list 'NEQ (lambda args (not (apply = args))))
        (list 'ANND (lambda (x y) (and x y)))
        (list 'ORR (lambda (x y) (or x y)))
        (list 'NOTT not)))

(define (run-repl)
  (display "Welcome to my repl.")
  (newline)
  (display "Type some scheme-ish at the prompt.")
  (newline)
  (display "Type <return> after each expression:")
  (newline)
  (repl operator-association-list)) ;; start repl with built-in procedure names defined

(define (repl env)
  (display "mini-eval>> ")
  (display (evaluate (read) env))
  (newline)
  (repl env))

(define (lookup-variable var env)
  (let ((binding (assq var env))) ;; look up var in the env association list
    (if binding
        (second binding)
        (error "lookup-variable: unbound variable " var))))

(define (apply-proc proc args)
  (cond [(procedure? proc) (apply proc args)] ;; Handle built-in (primitive, native) procs.
        [(and (list? proc)
               (eq? (first proc) 'CLOSURE)) (evaluate (third proc) ;; so eval proc body
                                                       ;; in the (CLOSURE) environment PLUS param-arg
                                                       (extend-environment (second proc) ;;
                                                                           bindings
                                                                           (params)
                                                                           args
                                                                           (fourth proc)))]])
        ;; (closure env)

(define (extend-environment params args env)
  (cond [(and (null? params) (null? args)) env] ;; Nothing more to add; return the current
environment.
        [(or (null? params) (null? args)) (error "extend-environment: mismatch of parameters
```

```

and arguments" (list 'params params 'args 'args))]
  [else (extend-environment (rest params)
                           (rest args)      ;; Recursively cdr down the param-arg
                           lists,
                           (cons (list (first params) (first args))
                                 env)))]]) ;; using the environment extended with the
                                         ;; first param-arg pairing.

(define (evaluate expr env)
  (cond [(null? expr) expr]                ;; Base cases: self-evaluating empty list,
        [(number? expr) expr]              ;; self-evaluating numbers
        [(boolean? expr) expr]              ;; and booleans;
        [(symbol? expr) (lookup-variable expr env)] ;; Symbols (names) need looking up.

        ;; Otherwise the expression begins with (. Check special forms first:

        ;; IPH is a conditional
        [(eq? (first expr) 'IPH) (if (evaluate (second expr) env)      ;; if the TEST is true
                                     (evaluate (third expr) env)        ;; evaluate the THEN
                                     (evaluate (fourth expr) env))]      ;; otherwise the ELSE

        ;; DEFINE causes us to evaluate in a different environment (with an added binding)
        [(eq? (first expr) 'DEFINE) (repl (cons (list (second expr)    ;; invoke REPL w/
                                                       (evaluate (third expr) env)) ;; extended
                                                       env))
        ;; (now includes this definition)

        ;; QUOTE doesn't evaluate its argument
        [(eq? (first expr) 'QUOTE) (second expr)]

        ;; LAMBDA creates a closure -- the procedure parameters, body, and current environment
        [(eq? (first expr) 'LAMBDA) (list 'CLOSURE (second expr) (third expr) env)]

        ;; Otherwise it's a "regular" procedure -- built-in or lambda. Evaluate it and all
        args,
        ;; then apply the value of the procedure to the value of the arguments.
        [(list? expr) (apply-proc (evaluate (first expr) env)
                                   (map (lambda (expr) (evaluate expr env))
                                        (rest expr)))]

        ;; Otherwise oops!
        [else (error "evaluate: not sure what to do with expr" expr)]))

(define (mini-eval sexpr)
  sexpr)

(run-repl)

```

```

;; In Scheme, parentheses are NOT arbitrary punctuation.
;; You must use them in exactly the right number and placement

1          ;; The number 1; similarly 2.5, -68, etc.
x          ;; A name; if not bound to a value, this gives an error
#t         ;; The true boolean; similarly #f
"A string" ;; As expected
'x         ;; A symbol
+ - * / < > <= >= = and or not eq? zero? null? number? ;; built-in functions

(fun arg1 ... argn)    ;; Use parentheses to call fun on arg1 through argn;
                        ;;   no punctuation between args

;; e.g.
(+ 3 (* 25 6))

(define (proc arg1 ... argn)  ;; Define a function; no punctuation
  body)                       ;;   between args; returns value of body

(if test    ;; Conditional; returns either val of then or of else
  then      ;; Note that there is also a more flexible conditional
  else)     ;;   called COND with somewhat more arcane syntax
;; e.g.
(define (count-down n)      ;; Most interesting functions use recursion
  (if (= n 1)
    (list 1)
    (cons n (count-down (- n 1))))) ;; Speaking of cons...see below

(cons x y)          ;; Returns a pair -- prints as (x . y) -- python 2-tuple
(first (cons x y))  ;; Returns the value of x -- sometimes spelled car
(rest (cons x y))   ;; Returns the value of y -- sometimes spelled cdr
nil                ;; The empty list; prints as ()

(cons x nil)        ;; Returns list of one element (the value of x)

(cons x
  (cons y nil))     ;; Returns list of two elements --
                    ;;   first is the value of x; rest is a list of
                    ;;   one element (the value of y)
(list x y)          ;; Equivalent to (cons x (cons y nil)), w/arbitrary # args

;; Useful list procedures:
append             ;; Non-destructive
reverse
map                ;; e.g. (map double (list 1 2 3)) --> (2 4 6)
filter             ;; e.g. (filter odd? (list 1 2 3)) --> (1 3)
null?              ;; true iff the list is empty, i.e., nil

(lambda (arg1 ... argn)  ;; an anonymous procedure
  body)

```

```
;; useful debugging functions -- often used within begin (see below)
(display "string")
(display x)
(newline)

(begin      ;; scheme's version of blocks
  expr1    ;; sequence:  evaluate the first expression, throw out its value,
  expr2    ;; evaluate the next...
  .
  .
  .      ;; eventually returning the final value
)
```

Racket has built-in documentation

[Python for Lisp Programmers](#) can also be read in reverse.

<http://www.cs.cornell.edu/courses/cs212/1999sp/handouts/scheme-quickref.html> is
a very quick overview of R5RS BUT avoid anything that has a ! in its name