

YOUR NAME PLEASE:

Computer Science 201b
Practice Final Exam
Spring 2015
2.5 hour exam + .5 hour of writing up

Closed book and closed notes. Show ALL work you want graded on the test itself.

For problems that do not ask you to justify the answer, an answer alone is sufficient. However, if the answer is wrong and no derivation or supporting reasoning is given, there will be no partial credit.

GOOD LUCK!

problem	points	actual
1	12	
2	12	
3	12	
4	12	
5	12	
6	12	
7	12	
8	12	
9	12	
10	12	
total	120	

1.(a) (6 points)

Write a Racket procedure (`insert x lst`) that takes as input a number `x` and a list `lst` of numbers in non-decreasing order, and returns a list of numbers in non-decreasing order that includes `x` and all the elements of `lst`. There may be duplicates in the lists.

Examples:

```
(insert 7 '(1 4 5 10 12)) => '(1 4 5 7 10 12)
```

```
(insert 3 '()) => '(3)
```

```
(insert 2 '(1 2 4)) => '(1 2 2 4)
```

```
(insert 2 '(1 2 2 4)) => '(1 2 2 2 4)
```

1.(b) (4 points)

Assuming that the insert procedure works as in part (a), use it to write a procedure (isort lst) that takes a list lst of numbers and returns the list of numbers sorted into non-decreasing order. There may be duplicates.

Examples:

```
(isort '(6 3 10 1 5)) => '(1 3 5 6 10)
```

```
(isort '()) => '()
```

```
(isort '(4 2 2 1 2)) => '(1 2 2 2 4)
```

1.(c) (2 points)

Give an example of a number x and a list lst of n elements in non-decreasing order that make your insert procedure from part (a) run for

(i) the shortest time (best case):

(ii) the longest time (worst case):

2. An and/or expression is recursively defined. The base case is a Boolean value or a symbol. The recursive case is a list containing three elements:
1. the left operand, which is an and/or expression,
 2. the operation symbol, '+' or '*',
 3. the right operand, which is an and/or expression.

Examples: 'x, #t, '(#t + #f), '(((x + y) * (u + v)) + (a + b))

2.(a) (3 points)

If `exp` is an and/or expression that is not a symbol or a Boolean value, what Racket expressions will give the following parts of `exp` (the first one is done):

(i) the left operand: `(first exp)`

(ii) the operation symbol:

(iii) the right operand:

2.(b) (6 points)

Write a Racket procedure (`reformat exp`) that takes an and/or expression `exp` and reformats it so that the operation symbol comes first in the list, followed by the left operand and then the right operand.

Examples:

`(reformat #t) => #t`

`(reformat 'hi) => 'hi`

`(reformat '(x + y)) => '(+ x y)`

`(reformat '(((x + y) * (u + v)) + (a + b))) => '(+ (* (+ x y) (+ u v)) (+ a b))`

2.(b) (more space)

2.(c) (3 points)

Draw the tree of recursive calls (and no return values) for the procedure call:
(reformat '((a + b) * (c + (d + e))))).

3. Recall that the TC-201 assembly-language instructions are:
halt, load address, store address, add address, sub address,
input, output, jump address, skipzero, skippos, skiperr,
loadi address, storei address

and the directive: data number, which reserves one memory location
and stores the number in it.

3.(a) (10 points)

Write a TC-201 program in assembly language that reads in a number, n
and then prints out the first n odd positive integers, in increasing order,
and then halts. You may assume that n is between 1 and 5000, inclusive.
You may use symbolic addresses. An example of the behavior of the program:

```
input = 4  
output = 1  
output = 3  
output = 5  
output = 7
```

3.(b) (1 point)

What is the largest value of n for which we could expect to have a TC-201 program output the first n odd positive integers in increasing order. Exponential notation is fine. Please justify your answer.

3.(c) (1 point)

Could there exist a Racket program that takes as input a configuration config of the TC-201 computer and correctly decides whether starting in config the machine would eventually reach a configuration with the run-flag equal to 0? Assume that the user types 0 in response to every input instruction. Please justify your answer.

4.

You are to design a combinational circuit with four inputs: a_1 , a_0 , b_1 , b_0 , and two outputs: c_1 , c_0 . a_1 and a_0 are interpreted as a 2-bit binary number A , b_1 and b_0 are interpreted as a 2-bit binary number B , and c_1 and c_0 are interpreted as a 2-bit binary number C . The value of C should be the minimum of A and B .

For example, if $a_1 = 1$ and $a_0 = 0$ then $A = 2$ because 10 in binary is 2.

4.(a) (6 points)

Complete the rows of the truth table giving c_1 and c_0 as a function of a_1 , a_0 , b_1 , b_0 :

a_1	a_0	b_1	b_0		c_1	c_0
0	0	0	0			
0	0	0	1			
0	0	1	0			
0	0	1	1			
0	1	0	0			
0	1	0	1			
0	1	1	0		0	1
0	1	1	1			
1	0	0	0			
1	0	0	1			
1	0	1	0			
1	0	1	1			
1	1	0	0			
1	1	0	1			
1	1	1	0			
1	1	1	1			

Note that for the row shown, a_1 and a_0 are 0 and 1, so $A = 1$, and b_1 and b_0 are 1 and 0, so $B = 2$. Thus, $C = \min(A,B) = 1$, so $c_1 = 0$ and $c_0 = 1$.

4.

4.(b) (2 points)

Give Boolean expressions for c_1 and c_0 .

$c_1 =$

$c_0 =$

4.(c) (4 points)

Draw a combinational circuit for computing outputs c_1 and c_0 from inputs a_1, a_0, b_1, b_0 . You may use AND and OR gates of any number of inputs, NOT gates of one input, and XOR gates of two inputs. You may draw gates as simple boxes; be sure to label your input and output wires and your gates (if you don't use the standard symbols.)

5. The following is a context-free grammar in BNF for a small subset of syntactically correct Racket expressions.

```
<exp> ::= <id> | <number> | <boolean> | <lambda exp>
        | <procedure call> | <if exp>
```

```
<id> ::= "x" | "y" | "z" | "+" | "*" | "="
<number> ::= "0" | "1" | "3" | "7" | "43"
<boolean> ::= "#t" | "#f"
<lambda exp> ::= "(" "lambda" <formals> <exp> ")"
<formals> ::= "(" <exp>* ")"
<procedure call> ::= "(" <exp> <exp>* ")"
<if exp> ::= "(" "if" <exp> <exp> <exp> ")"
```

5.(a) (3 points each)

For each expression below, draw a parse tree showing how it can be derived from <exp> using the rules above.

(i) 43

(ii) (if (= y z) 1 0)

5.(a) continued.

(iii) $((\lambda x (+ x 7)) 3)$

5.(b) (3 points)

If L is a context-free language and $\text{reverse}(L)$ is the set of all reverses of strings in L , must $\text{reverse}(L)$ also be a context-free language? Please briefly justify your answer.

6. Consider a Racket procedure (`make-set name`) that takes a symbol name as input and returns a Racket procedure that implements a set object with local storage which can process the following commands:

```
name          -- returns the name of the set
contains? value -- returns #t if value is a member of the
                set, or #f if value is not a member of the set
include value  -- changes the set so that value is a member of
                the set, and returns the symbol 'ok
```

Initially the set has no members. Examples of using (`make-set name`):

```
> (define s1 (make-set 'first-set))
> (s1 'name)
'first-set
> (s1 'contains? 2)
#f
> (s1 'include 2)
'ok
> (s1 'include 3)
'ok
> (s1 'contains? 2)
#t
> (define s2 (make-set 'second-set))
> (s2 'contains? 2)
#f
> (s2 'contains? 'second-set)
#f
> (s1 'contains? 3)
#t
> (s2 'name)
'second-set
>
```

6.(a) (9 points)

Write a Racket procedure to implement `(make-set name)`.

No error checking is necessary.

6.(b) (3 points)

Describe or draw the accessible environments immediately after the completion of the `(s1 'include 3)` command above.

Include and label the search pointer of each environment and the birth pointers of the procedures `make-set` and `s1`.

7.(a) (2 points)

Draw the box and pointer representation of the list `x` constructed as follows.

```
> (define x (cons 13 (cons 17 (cons 14 '()))))  
> x  
'(13 17 14)  
>
```

`x:`

7.(b) (2 points)

Suppose `x` is as in 7(a) and we define `y` as follows

```
> (define y (cons 99 x))  
> y  
'(99 13 17 14)  
>
```

Draw a new box and pointer diagram showing the resulting structure of `x` and `y`.

`x:`

`y:`

7.(c) (2 points)

Explain why we can consider the Racket procedures `rest` (or `cdr`) and `cons` to be constant time, that is, $O(1)$.

7.(d) (3 points)

Consider the following Racket procedure.

```
(define (rev lst rlst)
  (if (null? lst)
      rlst
      (rev (rest lst) (cons (first lst) rlst))))
```

Draw the tree of recursive calls (and no returns) for the procedure call `(rev '(a b c) '())`.

7.(e) (3 points)

For the procedure in 7(d) suppose we call `(rev lst '())` where the list `lst` contains n numbers. Give a "Big Theta" bound in terms of n for the running time of this procedure call, and explain why it is correct.

8. For each of the following pairs consisting of a string and a regular expression, determine whether the string is in the language of the regular expression or not, answering YES or NO. Recall that $|$ stands for "or" and $*$ for "Kleene star", that is, zero or more repetitions. In each case the alphabet is $\{a,b,c\}$.

(2 points each)

8.(a) aacco $(a)^*(b)^*(c)^*$

8.(b) abaacbco $(a|b)^*(b|c)^*$

8.(c) abcbo $((ab | ba)^* | (bbc | cbc)^*)$

8.(d) bccbacba $(bc | bcc)(bac | cba)(cba | aa)$

8.(e) accbcc ((a | b)c(c)*)*

8.(f) cabacbbbb (ca | ba | aa | cb | bb | ab)*

9. Answer each question briefly. (2 points each)

9.(a) What is the main thing a compiler does?

9.(b) Give an example of a set of strings that is not regular.

9.(c) What is memoization, and when is it useful?

9.(d) Give a small example of a sequential (not combinational) circuit.

9.(e) We proved that there can be no program to solve the Halting Problem. State the Halting Problem for Racket programs.

9.(f) Is the following procedure tail recursive? Why or why not?

```
(define (proc lst n)
  (if (null? lst)
      n
      (proc (cdr lst) (+ n 1))))
```

10. A *rewriting system* R consists of a finite alphabet of symbols and a finite set of rules. Each rule is of the form $s_1 \rightarrow s_2$, where s_1 and s_2 are strings of symbols. The lefthand side of the rule is s_1 and the righthand side of the rule is s_2 .

A *derivation step* is a pair of strings (t, u) such that for some rule $s_1 \rightarrow s_2$, u is obtained from t by replacing one occurrence of s_1 as a substring of t by s_2 . A *completed derivation* is a finite sequence of strings (t_1, t_2, \dots, t_n) such that
(1) each consecutive pair forms a derivation step, and
(2) no rule applies to the last string, t_n .
In this case, we say t_1 *may yield* t_n .

As an example, consider a rewriting system with alphabet $\{x, y\}$ and the single rule $yx \rightarrow xy$. For this system, we have a completed derivation:

$yyxyx, yxyyx, yxyxy, yxxyy, xyxyy, xxyyy$.

Note that no rule applies to the last string. Thus, $yyxyx$ may yield $xxyyy$.

10.(a) (3 points)

In the example one-rule system above, if a string s may yield a string t , describe how are s and t related.

10.(b) (6 points)

For the alphabet $\{0,1\}$ construct rules for a rewriting system such that for every non-empty string s , s may yield exactly one of 0 or 1, and s may yield 1 if and only if the number of 1's in s is odd. (You'll need more than one rule.)

Examples of the behavior of this system:

101110 may yield 0, but not 1.

1101101 may yield 1, but not 0.

0000 may yield 0, but not 1.

10.(c) (3 points)

We define a decision problem as follows. The input is a rewriting system R and a string s . (We assume the alphabet of R contains 1.) The output is "yes" if s may yield 1 in the system R , and "no" otherwise.

Is this a computable problem or not? Please briefly justify your answer.