YOUR NAME PLEASE:

SOLUTIONS

[Revised 5/2/17]

Computer Science 201
Final Exam
Friday, May 1, 2-5 pm
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!

<table>
<thead>
<tr>
<th>problem</th>
<th>points</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>108</td>
<td></td>
</tr>
</tbody>
</table>
1.(a) (6 points)
Write a single recursive Racket procedure

(member? item lst)

that takes a Racket value item and a list lst, and returns #t if some top-level element of lst is equal? to item, and #f otherwise. Do not use Racket procedures other than: null?, empty?, equal?, car, first, cdr, rest, cons, if, cond.

Examples:

(member? 13 '()) => #f
(member? 'a '(b a c)) => #t
(member? '(2 3) '((1 2) (3 2) (4 2))) => #f
(member? "not" '("is" "this" "not")) => #t

(define (member? item lst)
  (cond
    [(null? lst) #f]
    [(equal? item (first lst)) #t]
    [else
      (member? item (rest lst))])))
1.(b) (4 points)
For the procedure (member? item lst) you wrote in part 1.(a), draw the tree of recursive calls (no returns) for the procedure application

\[
\text{(member? 'd '(a b c d e))}
\]

\[
\begin{align*}
\text{(member? 'd '(a b c d e))} \\
& \quad | \\
\text{(member? 'd '(b c d e))} \\
& \quad | \\
\text{(member? 'd '(c d e))} \\
& \quad | \\
\text{(member? 'd '(d e))}
\end{align*}
\]

1.(c) (2 points)
Suppose the procedure (member? item lst) you wrote in part 1.(a) is called with a number and a list of n numbers.
Give examples of best and worst case inputs, and big Theta bounds on their running times (as a function of n). Assume comparing two numbers with equal? takes time Theta(1).

**Best:** (item is first on list)
\[
\text{item} = 1 \quad \text{lst} = '(1 2 \ldots n) \quad \Theta(1)
\]

**Worst:** (item is not on list)
\[
\text{item} = n+1 \quad \text{lst} = '(1 2 \ldots n) \quad \Theta(n)\]
2. Suppose we define the following structs to represent arithmetic expressions:

(struct plus (arg1 arg2) #:transparent)
(struct times (arg1 arg2) #:transparent)

Recall that these automatically define constructors (plus, times), type predicates (plus?, times?) and selectors (plus-arg1, plus-arg2, times-arg1, times-arg2.)

We inductively define an Arithmetic Expression as follows:
1. A number is an Arithmetic Expression,
2. If exp1 and exp2 are Arithmetic Expressions, then so are (plus exp1 exp2) and (times exp1 exp2).

2.(a) (6 points)
Write a single recursive procedure (val exp) that takes an Arithmetic Expression according to the definition given, and returns its value, interpreting plus as addition, and times as multiplication.

Examples:
(val -3) => -3
(val (plus 4 5)) => 9
(val (times (plus 4 -6) (plus 3 7))) => -20

(define (val exp)
  (cond
    [(number? exp) exp]
    [(plus? exp)
      (+ (val (plus-arg1 exp))
         (val (plus-arg2 exp)))]
    [else
      (* (val (times-arg1 exp))
         (val (times-arg2 exp)))]))
2. (b) (3 points)
Draw the tree of recursive calls *and return values* for the
procedure call: `(val (times (plus 2 4) (plus 3 5)))`
using your procedure value from 2.(a).

```
<table>
<thead>
<tr>
<th></th>
<th>(val (times (plus 2 4) (plus 3 5))</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(val (plus 2 4))</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>(val 2)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(val (plus 3 5))</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>(val 3)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(val 5)</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

2. (c) (3 points)
Given an Arithmetic Expression, define its "deep reverse" as
the Arithmetic Expression obtained by reversing the arguments to
every plus or times constructor in the expression. Write one
recursive Racket procedure `(drev exp)` that takes an Arithmetic
Expression `exp` as input and returns the deep reverse of `exp`.

Examples:
(drev -23) => -23
(drev (plus 3 4)) => (plus 4 3)
(drev (plus (times 2 3) (plus 6 1))) => (plus (plus 1 6) (times 3 2))

```
(define (drev exp)
  (cond
    [(number? exp) exp]
    [(plus? exp)
      (plus (drev (plus-arg2 exp))
            (drev (plus-arg1 exp)))]
    [else
      (times (drev (times-arg2 exp))
             (drev (times-arg1 exp)))])
)
```
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) (9 points)
Write a TC-201 program in assembly language that reads in a number n
and then prints out the square of n (that is, n x n). If n is zero,
it halts, otherwise it reads in and squares another number.
You may assume that n is between 0 and 100, inclusive. Please use
symbolic opcodes and addresses! Example run of the program:

input = 4
output = 16
input = 6
output = 36
input = 0
output = 0

read-loop:  load zero
            store sum
            input           
read:       store n
            store count
square:     load count
            skippos
            jump done
            sub one
            store count
            load sum
            add n
            store sum
            jump square
done:       load sum
            output
            skipzero
            jump read-loop
halt
zero:       data 0
sum:        data 0
n:           data 0
count:      data 0
one:        data 1
3. (b) (2 points)
What modifications could you make to your program of part 3.(a) to allow inputs of numbers between -100 and 100, inclusive?

The instructions from read to square could be replaced by

read:       input
            store n
skip sos
jump negate
negate:     load zero
            sub n
store n
store count
square:     (etc.)

3. (c) (1 point)
In part 3.(a), could we extend the range of n to be 0 to 200? Why or why not?

No, because \(200 \times 200 = 40,000\) but the largest positive integer representable in the TC-200 is \(2^{15} - 1 = 32,767\).
4.
You are to design a combinational circuit with three inputs: a, b, c and two outputs: d, e, given by the following truth table.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

4.(a) (4 points)
Give Boolean expressions (using and, or, not) for d and e as functions of a, b, c:

\[
d = a' + b \cdot c
\]

\[
e = a' \cdot c + a \cdot c'
\]
4. (b) (6 points)
Give a circuit to compute outputs d and e from inputs a, b, and c, as specified in the truth table above. Be sure to label your input and output wires. Use only NOT gates, 2-input AND gates, and 2-input OR gates. Rectangles labeled with gate symbols are fine.

4. (c) (2 points)
What is the minimum number of gate delays to guarantee that both outputs d and e are correct for your circuit in 4.(b), and why?

3 gate delays because the longest path, from a or c to e, has gates NOT, AND, OR in sequence.
5.(a)
The following is a context-free grammar in BNF for a small subset of syntactically correct Racket expressions.

\[
\begin{align*}
\text{<exp>} & : = \text{<var>} \mid \text{<op>} \mid \text{<number>} \mid \text{<boolean>} \mid \text{<let exp>} \\
& \quad \mid \text{<procedure call>} \mid \text{<if exp>}
\end{align*}
\]

\[
\begin{align*}
\text{<var>} & : = "x" \mid "y" \mid "z"
\text{<op>} & : = "+" \mid ";" \mid ":=
\text{<number>} & : = "0" \mid "1" \mid "2" \mid "3" \mid "7" \mid "22"
\text{<boolean>} & : = "#t" \mid "#f"
\text{<let exp>} & : = "(" "let" <env> <exp> ")"
\text{<env>} & : = "(" <bind> ")"
\text{<bind>} & : = "(" <var> <exp> ")"
\text{<procedure call>} & : = "(" <exp> <exp> ")"
\text{<if exp>} & : = "(" "if" <exp> <exp> <exp> ")"
\end{align*}
\]

For each expression below, draw a parse tree showing how it can be derived from <exp> using the rules above. (3 points each)

5.(a)(i) \( z \)

\[
\begin{align*}
\text{<exp>}
\quad & \text{<var>}
\quad & z
\end{align*}
\]

5.(a)(ii) \( (\leq y 22) \)

\[
\begin{align*}
\text{<exp>}
\quad & \text{<procedure call>}
\quad & \text{<exp>}
\quad & \text{<exp>}
\quad & \text{<exp>}
\quad & \text{<op>}
\quad & \text{<var>}
\quad & \text{<number>}
\quad & \leq
\quad & y
\quad & 22
\end{align*}
\]
5. (a) (iii) \((\text{let } ((x \ 7) \ (y \ 3)) \ (+ \ x \ y))\)

5. (b) (3 points)
Suppose \(L\) is the context-free language of a context-free grammar \(G\) with terminal alphabet \(T\), nonterminal alphabet \(N\), start symbol \(S\), and rules \(R\). Explain how to modify \(G\) to get a grammar \(H\) for the language \((L)^*\), that is, all strings consisting of a concatenation of 0 or more strings from \(L\).

Let \(S'\) be a new start symbol for \(H\), and let the terminals of \(H\) be the terminals of \(G\) and the nonterminals of \(H\) are the nonterminals of \(G\) and \(S'\). The rules of \(H\) are the rules of \(G\) and

\[
S' \rightarrow \varepsilon \quad \text{(to produce the empty string)}
\]

\[
S' \rightarrow SS' \quad \text{(to produce a concatenation of elements of } L)\]
6. Consider a Racket procedure (make-timer) to make a timer object with local storage that implement the following commands:

'\texttt{start}' starts (or re-starts) the timer, and returns '\texttt{ok}'

'\texttt{value}' returns the time (in seconds) since it was last started or re-started -- this command should return 'none if the timer has not yet been started

When you call the Racket procedure (current-inexact-milliseconds), the value returned is the current value of the number of milliseconds elapsed since midnight UTC, Jan. 1, 1970. A millisecond is $1/1000$ of a second.

Examples of using (make-timer):
> (define s1 (make-timer))
> (s1 'value)
'none
> (s1 'start)
'ok
> (s1 'value)
6.58752197265625
> (s1 'value)
12.2689970703125
> (define s2 (make-timer))
> (s2 'start)
'ok
> (s2 'value)
7.517176025390625
> (s1 'value)
36.67952807617188
> (s1 'start)
'ok
> (s1 'value)
7.12656689453125
>
6.(a) (8 points)
Write one Racket procedure to implement (make-timer).
No error checking is necessary.

\[
\text{(define } \text{(make-timer)} \text{)}
\]
\[
\text{(let ((time 'none)}
\[
\text{(lambda (cmd)}
\[
\text{(case cmd}
\[
\text{[(start)}
\[
\text{\hspace{1cm}(set! time \text{(current\-inexact\-milliseconds)))}}
\[
\text{'ok}]
\[
\text{[(value)}
\[
\text{\hspace{1cm}(if (equal? time 'none)}
\[
\text{\hspace{2cm}'none)}
\[
\text{\hspace{2cm}(- \text{(current\-inexact\-milliseconds)}
\[
\hspace{2cm}\text{time)}
\[
\hspace{2cm}1000)))]])}
\]
\[
\text{]}]
\]
\]
\]

6.(b) (4 points)
Describe the concepts of Racket environment, search pointer, and birth pointer, and how they interact with procedure creation and procedure application.

An environment is a \text{table (or list)} of identifiers and their values.

The search pointer of an environment points to the environment (if any) to search next for the value of an identifier.

The birth pointer of a procedure points to the environment where the procedure was created.

When a procedure is applied, a local environment is created with its formal and actual arguments and whose search pointer points to the birth environment of the procedure.
7. (a) (8 points)
Write a procedure \( \text{subsets lst} \) that takes a list \( \text{lst} \) of values and returns a list of all the lists that can be obtained from \( \text{lst} \) by omitting zero or more elements. You may write auxiliary procedures -- please state what they do. You may assume that all the elements of \( \text{lst} \) are distinct. The lists returned may be in any order.

Examples:
(\( \text{subsets } () \) => \( () \))
(\( \text{subsets } (a \ b \ c) \) => \( () \ (c) \ (b \ c) \ (a \ c) \ (a \ b) \ (a \ b \ c) \))

\[
\begin{align*}
\text{define } & (\text{subsets} \ \text{lst}) \\
& (\text{if} \ (\text{null?} \ \text{lst}) \\
& \quad (()) \\
& \quad (\text{let} \ ((\text{prev} \ (\text{subsets} \ \text{rest lst})))) \\
& \quad (\text{append} \\
& \quad \quad (\text{prev} \\
& \quad \quad \quad (\text{map} \ (\lambda x) \\
& \quad \quad \quad \quad (\text{cons} \ ((\text{first lst}) \ x)) \\
& \quad \quad \quad \text{prev}))))))
\end{align*}
\]
7.(b) (2 points)
Draw a box-and-pointer diagram for the list x constructed as follows.

\[
(\text{define } x (\text{cons (cons 1 '()) (cons 2 (cons 3 '()))}))
\]

7.(c) (2 points)
Consider the procedure \( f n \) defined as follows:

\[
(\text{define } f n)
(\text{if } (\leq n 1)
 (\text{cons 'a (cons 'b '())})
(\text{let ((prev (f (- n 1)))})
 (\text{cons prev (cons prev '())})))
\]

Draw a box-and-pointer diagram of the value returned by \( f 3 \).
8. For each of the following regular expressions, describe or draw a deterministic finite state machine that recognizes the same language. Your machine may be "incomplete", but be sure to indicate the start state, the accepting state(s) and all the relevant transitions. Recall that we use "\|" for "or", "*" for repetition 0 or more times, and "(" and ")" for grouping.

(3 points each)

8. (a) \((a*(b)*c)*\)

8. (b) \((a \| b) c(c*)*\)
8.(c) \((ab \mid ba)^*\)

8.(d) \((a|b|c)^* (a|b|cc)\)
9. Answer each question briefly. (2 points each)

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

Translate symbolic opcodes and addresses (or instructions and data) into equivalent machine instructions and data (or bit patterns.)

9. (b) Show that the set of strings of balanced parentheses is context-free. Examples of strings in the language are:

((), ()(), ()(()), ()()(()), ()()()()

(terminals: { (, )} nonterminals: {S} start: S)

S → ()
S → (S)
S → SS

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

Storing argument/value pairs during a recursive computation and looking them up rather than recomputing them. It is useful when there is much repeated computation.
9. (d) Recall that \( \text{NOR}(s,t) \) is defined by the truth table:

\[
\begin{array}{c|cc}
 s & t & \text{NOR}(s,t) \\
\hline
 0 & 0 & 1 \\
 0 & 1 & 0 \\
 1 & 0 & 0 \\
 1 & 1 & 0
\end{array}
\]

Suppose we have a circuit with wires \( x, y, u, v \) defined by the equations:

\[ u = \text{NOR}(x, v) \]
\[ v = \text{NOR}(y, u) \]

What are all the stable configurations of this circuit?

9. (e) Is it possible to construct a Turing machine that takes as input a string representing a positive integer \( n \) (in decimal) and halts with output \( 1 \) if \( n \) is the square of some integer \( m \), and halts with \( 0 \) otherwise? Why or why not?

Yes, we could start with \( m = 1 \) and square it and check equality with \( n \). If \( m^2 \) is greater than \( n \) we output \( 0 \) and halt. If \( m^2 \) is equal to \( n \) we output \( 1 \) and halt. Otherwise, we increment \( m \) and repeat. This must eventually halt.

(or: this is a computable function, and a Turing machine can compute any computable function.)

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

\[
\text{(define (proc lst)}
\quad \text{(if (null? lst)}
\quad \quad 0
\quad \quad (+ 1 (proc (rest lst)))))
\]

No, because the value of the recursive call \( \text{(proc (rest lst))} \) is modified before it is returned.