CSE 413 Final Exam

December 12, 2016

Name ___________________________________

The exam is closed book, closed notes, no electronic devices, signal flags, tin-can telephones, implants, or other signaling or communications apparatus.

Style and indenting matter, within limits. We're not overly picky about details, but we do need to be able to follow your code and understand it.

Please wait to turn the page until everyone has their exam and you have been told to begin. If you have questions during the exam, raise your hand and someone will come to you. ***Don't*** leave your seat.

Advice: The solutions to many of the problems are short. Don't be alarmed if there is a lot more room on the page than you actually need for your answer.

More gratuitous advice: Be sure to get to all the questions. If you find you are spending a lot of time on a question, move on and try other ones, then come back to the question that was taking the time.

1	/ 18
2	/ 12
3	/ 16
4	/ 12
5	/ 10
6	/ 12
7	/ 8
8	/ 6
9	/ 6
Total	/ 100

Question 1. (18 points) Regular expressions. For each of the following, (i) give a regular expression that generates the set of strings described, and (ii) draw a DFA that accepts that set of strings. There is lots of blank space for your answers – don't worry if you don't need nearly this much room.

Fine print: You may use basic regular expressions (sequences rs, choice r|s, repetition r*, and parentheses for grouping). You may also use + (one or more) and ? (zero or one), and character classes like [ax-z] and [^abc]. You also may use named abbreviations like "vowels ::= [aeiou]" if these help. You may not use additional regular expression operators that might be found in various programming language libraries or software tools.

(a) (6 points) A valid CSE course number like CSE413 consists of the letters 'CSE' followed by exactly 3 digits. The first digit of the number must be a decimal number in the range 1-6; the remaining numbers can be any decimal number in the range 0-9.

(continued on next page)

Question 1. (cont.) As with part (a), (i) give a valid regular expression and (ii) DFA to recognize this set of strings.

(b) (12 points) An ISO-8601 date has the format YYYY-MM-DD, where YYYY are the characters of a 4-digit year (0000 to 9999), MM are the two characters of the month in the range 01 to 12, and DD are the two characters of the day in the range 01 to 31. (For example, the date of this exam is 2016-12-12, and tomorrow will be 2016-12-13.) You do not need to worry about restricting dates to smaller ranges based on the month. For example, if the month is 02, the actual date will be no greater than 29, but you do not need to account for that. But you should be sure that no month is greater than 12 and no day is greater than 31.

Question 2. (12 points) Consider the following grammar:

S ::= (S) S | ε

(Recall that ε is the empty string.)

(a) (8 points) Draw the parse tree for (( )) ( )

(b) (4 points) Describe in English the set of strings generated by this grammar.

Question 3. (16 points) Ruby programming. Write a Ruby program that reads text from standard input and, after reading the entire input, prints all of the words that occur more than once in the input. Each word that is printed should be printed exactly once, but the order in which the words are printed is not specified. Each word should be printed on a separate line. For example, if the input is:

> how now brown cow
> don't have a cow
> the cow jumped over the moon

then the output should consist of the words "cow" and "the", since they are the only words that occur more than once in the input.

You should assume that words are any non-blank sequences of characters in an input line that are separated by one or more blanks. Upper- and lower-case characters ('A' and 'a') are different – you should not convert or transform the input characters. There might, or might not, be leading or trailing blanks at the beginning or end of a line.

For full credit you should use Ruby iterators like `each` to process the contents of any containers like arrays or hashes. Your solution should process the input in linear time – i.e., a solution that reads all of the words into a giant string and then compares every word to every other, which takes $O(n^2)$ time, would not receive full credit. You also should avoid reading and storing the entire input file before processing it – process each line of input as you read it.

A couple of possibly useful facts about strings:

- If `s` is a string, `s.length` is the number of characters in it.
- If `s` is a string, `s.trim` is a copy of `s` with any leading or trailing blanks omitted.
- The string `split` method returns an array of the words in a string. Example: `"one two three".split` returns `["one", "two", "three"]`. If the entire string consists of blanks or has no characters in it, `split` will return an empty array `[]`.

Write your code below or on the next page.

Question 3. (cont.) Additional space for your Ruby code, if needed.

Question 4. (12 points) Ruby inheritance and mixins. Consider the following code that consists of four Ruby classes and an additional "mixin" module. (Recall that if we "include" a mixin module in a class, it incorporates the methods from the module in the current class.)

```
class Apple
 def m1
  puts "A-a"
 end
 def m2
  puts "A-aa"
  self.m1()
 end
end

module Mango
 def m1
  puts "M-m"
 end
 def m3
  self.m1()
 end
end
```

```
class Banana < Apple
 include Mango
 def m4
  puts "B-b"
 end
end

class Citrus < Apple
 def m2
  super
  puts "C-cc"
 end
end
```

```
class Durian < Banana
 include Mango
 def m2
  puts "D-d"
  m3()
 end
 def m4
  super
 end
end
```

For each of the following, write down the output produced by executing that line of code, or, if an error occurs, explain what happens.

a) Apple.new.m1

b) Banana.new.m2

c) Citrus.new.m1

d) Citrus.new.m2

e) Citrus.new.m3

f) Durian.new.m4

The next few questions concern the calculator language from the last two assignments. If you recall, the grammar for the calculator language was as follows:

program ::= *statement* | *program statement*
statement ::= *exp* | *id* = *exp* | `clear` *id* | `list` | `quit` | `exit`
exp ::= *term* | *exp* + *term* | *exp* − *term*
term ::= *power* | *term* * *power* | *term* / *power*
power ::= *factor* | *factor* ** *power*
factor ::= *id* | *number* | (*exp*) | sqrt (*exp*)

We would like to add relational operators to the language. The idea is that an operator like `<` compares the values of two expressions and evaluates to the value 1 if the relation is true or the value 0 if it is false. So, for instance, `3<4` evaluates to 1, while `1<0` evaluates to 0.

Relational operators should have lower precedence than any of the other arithmetic operators. So `2+1<2` means the same as `(2+1)<2`, which evaluates to 0. Relational operators are left associative binary operators just like +, and −, so `3<1<2` is interpreted as `(3<1)<2`, which evaluates to `0<2` or 1.

The rest of the calculator language remains the same, with numeric constants and variables; expressions involving +, −, *, /, **, and parentheses; the `sqrt` function; assignment statements *id* = *exp*`;` and the keywords `clear`, `list`, `quit`, and `exit`. (Most of this information about the existing calculator language is not needed to answer the following questions.)

Question 5. (10 points) Suppose that we have added to the language the full set of six relational operators: `<`, `<=`, `==`, `!=`, `>=` and `>`. After these changes to the language, consider the following input:

```
xvii  =17

clear  a<==>b<!=1+0>>2plus3

sqrt(1<2)>=3===exit42
```

Show how the calculator scanner would divide these input characters into tokens by drawing a box around each sequence of characters that make up a single token. Boxes on the first line are drawn for you. You do not need to show any “end of line” or “end of file” tokens. (Remember that we’re only asking how the scanner would divide the input characters into tokens, not whether the resulting token sequence makes any sense or is a legal calculator program.)

Question 6. (12 points) We need to add the relational operators to the grammar. To keep things simple for this question, we will only deal with the $<$ operator. The others would all be handled similarly, but handling only $<$ is enough here.

In an attempt to add $<$ to the language of expressions, one of our summer interns modified the rule for *exp* as follows:

exp ::= *term* | *exp* + *term* | *exp* − *term* | *exp* $<$ *exp*

(a) (6 points) Show that this grammar is ambiguous.

(b) (6 points) Give a different grammar that will add the $<$ operator to expressions but that is unambiguous, gives $<$ lower precedence than the other arithmetic operators, and ensures that $<$ is left-associative. You only need to rewrite or add the rules (productions) needed to make this change – you do not need to copy down other rules that remain unchanged.

Some short questions on memory management.

Question 7. (8 points) Two of the strategies we looked at for reclaiming memory automatically were reference counting and mark-sweep garbage collection. A claim made in class was that reference counting did not always do as complete a job as garbage collection in reclaiming unreachable (not-in-use) memory.

Give an example of a dynamically allocated data structure that would not be reclaimed by a memory manager using reference counting, but would be successfully reclaimed by a mark-sweep collector. Give a brief and to-the-point explanation of why this is the case.

(You won't need all of the space below, but you will probably find it useful to draw a small diagram or two to illustrate your answer.)

Question 8. (6 points) In traditional languages like C or Java, the *automatic storage* used for function (method) local variables is allocated on a stack. That makes it efficient to allocate storage when a function is called and release it quickly when the function exits. The question is, can this always be done for a functional language like Racket? Why or why not? If it cannot be done, what needs to be done instead? (It should be possible to answer this in a couple of sentences.)

Question 9. (6 points) A *generational garbage collector* performs frequent garbage collection on part of the heap, but does not collect the entire heap nearly as often. Explain why this is done in a couple of sentences: which part of the heap is collected frequently and what does it contain? And why is this an effective strategy?

Have a great winter break and best wishes for the new year!
The CSE 413 staff