CSE 413 Final Exam Spring 2011 Sample Solution

Question 1. (10 points) Regular expressions I. Describe the set of strings generated by each of the following regular expressions. For full credit, give a description of the sets like "all sets of strings made up of a's, b's, and c's with 4 a's and at least as many b's as a's". Don't just transcribe the expressions from regular expression notation into English.

(a) 0(10)* | 1(01)*

Strings of alternating 0's and 1's that begin and end with the same character, either 0 or 1.

(b) ((0|1)* (2|3)+)+

Strings consisting of 0's, 1's, 2's, and 3's, where there is at least one 2 or 3, and that end with 2 or 3.

Question 2. (12 points) Regular expressions II. Write a regular expression or set of regular expressions that generate the following sets of strings.

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

(a) All strings of a's, b's, and c's that have at least one character and have the property that all of the c's (if any) appear to the right of all of the a's (if any). Examples of such strings: a, b, abba, bcb, acc, ababbcb. Examples of strings that are not in the set: ca, abbacbccbbabc.

There are many possible solutions. One is (a|b)* (a|b|c) (b|c)*

(b) Ruby-like integers written as either decimal or binary integer literals. A decimal integer is any string of one or more digit characters (0-9). A binary integer begins with the characters 0b or 0B and is followed by one or more 0's and 1's. Examples of binary integer literals: 0b0, 0B1, 0b001011000.

[0-9]+ | 0(b|B)(0|1)+

Question 3. (8 points) Suppose we are writing a scanner for Java (or, for the purposes of this question, it could be C or C++). The scanner needs to recognize identifiers, reserved words, operators, punctuation, and so forth.

(a) Suppose we have the following characters in the input, with spacing as shown:

```
if thing == x = ret urn + 1
```

List below the tokens that would be recognized by a Java scanner as it reads this input. (Remember that the scanner does not care whether the sequence of tokens it returns makes up a sensible program.)

IF ID(thing) EQUALS ID(x) BECOMES ID(ret) ID(urn) PLUS INT(1)

Note: Any reasonable token names were okay, but answers did need to show the difference between reserved words, each of which is a separate token class, and identifiers, all of which have the same token class but which include the actual identifier as a parameter.

(b) Now suppose we scan exactly the same input, but with all spaces omitted. What tokens would be returned by the scanner as it reads this altered input?

```
ifthing==x=return+1
```

ID(ifthing) EQUALS ID(x) BECOMES RETURN PLUS INT(1)

Question 4. (8 points) Give an unambiguous context free grammar that generates Scheme lists containing identifiers and balanced sets of parentheses and square brackets. Identifiers may only occur inside of parentheses or brackets. Examples: `(a)`, `[b c]`, `([(a)])`, `(a ((b c) d))`, `(a b c)`, `(let ([x y] [p q]) (x p))`. Non examples and reasons why not: `a` (no parentheses or brackets), `(a b (c)` (mismatched parentheses), `(a [b c))` (improper nesting).

You should assume that `id` is a non-terminal that stands for an arbitrary identifier. You do not need to worry about including specific identifiers like `a` or `b`. You also do not need to worry about whitespace – assume that if you generate the sequence `id id id` there would be appropriate whitespace to avoid any ambiguities.

There are, of course, many possible grammars that work. Here is one:

***list* ::= (*items*) | [*items*]**
items* ::= *items item* | *item
item* ::= *id* | *list

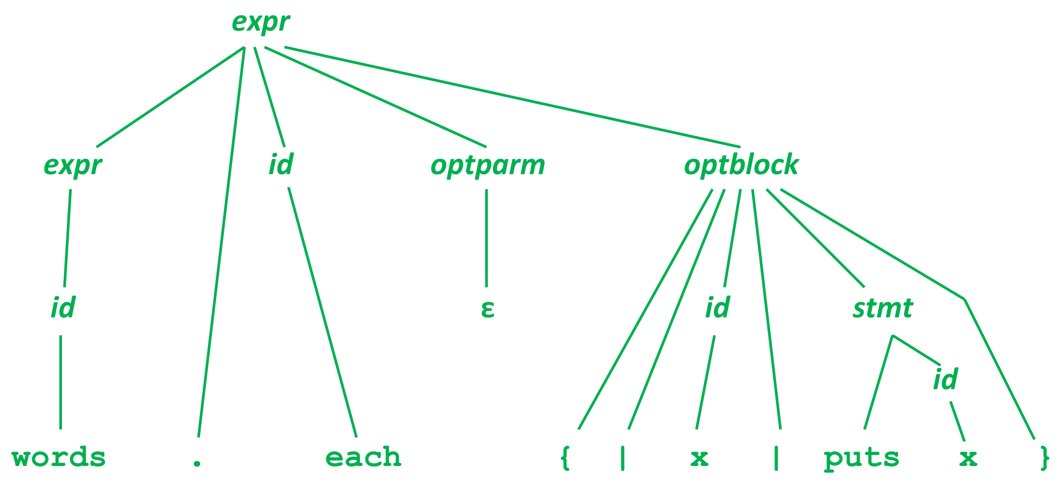
The question was not clear whether the grammar should generate empty lists ([ ] and ( )) or not. We allowed answers that included empty lists even though that was not the original intent.

Question 5. (8 points) Here is a grammar for simplified Ruby method calls. A method call has a receiver object (an expression), a method name, an optional single parameter, and is optionally followed by a block. A block contains a statement and an optional parameter.

> *expr* ::= *id* | *expr* **.** *id* *optparm* *optblock*
> *optparm* ::= ε | (*expr*)
> *optblock* ::= ε | { *stmt* } | { | *id* | *stmt* }
> *stmt* ::= puts *id* | gets

The non-terminal *id* can derive any identifier (sequence of letters). The symbol ε is the Greek letter epsilon, denoting the empty string. Note that the vertical bars (| |) surrounding *id* in a block are literal symbols, not the "choice" or alternative symbols found elsewhere in the grammar rules.

Draw the full parse tree showing the derivation of this expression: `words.each {|x| puts x}`



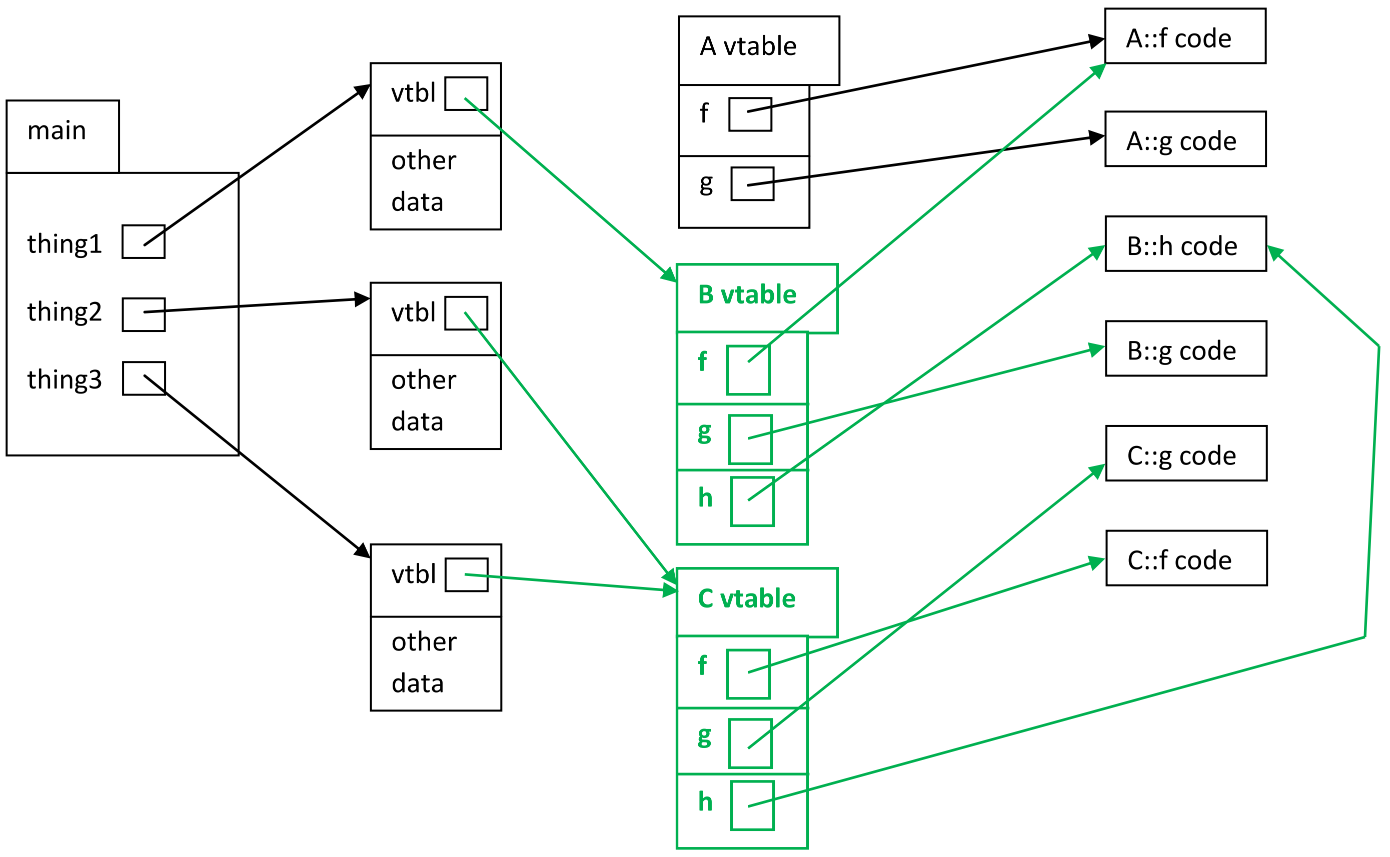
Question 6. (12 points) Dynamic dispatch (otherwise known as virtual function calls)

Consider the following Java program:

```
class A {
    void f() { g(); System.out.println("A::f"); }
    void g() {      System.out.println("A::g"); }
}

class B extends A {
    void h() { g(); System.out.println("B::h"); }
    void g() {      System.out.println("B::g"); }
}

class C extends B {
    void g() {      System.out.println("C::g"); }
    void f() { h(); System.out.println("C::f"); }
}

class Dispatch {
   public static void main(String[] args) {
     A thing1 = new B();
     B thing2 = new C();
     C thing3 = new C();

     /* Part (a) (see next page) Complete the diagram showing objects,
        vtables, and methods at this point during execution. */

     thing1.f();
     System.out.println("---");
     thing2.f();
     System.out.println("---");
     thing3.h();
   }
}
```

Answer questions about this program on the next page. You can remove this page for convenience while you work on the problem.

Question 6. (cont) (a) In lecture we talked about the virtual function table mechanism used to connect objects, classes, and methods in statically typed object-oriented languages like Java and C++. The variables declared in the main method, the objects they reference, the method code from the various classes, and the virtual function table (vtable) for class A are shown. You should complete this diagram to show the additional tables and connections that exist when execution of the program reaches the point shown by the comment in the middle of the main method on the previous page.



Note: it is essential that the order of the method pointers in the vtables match between base classes and their subclasses so that method dispatch will work regardless of the actual class of the object.

(b) What output does this program produce when it is executed? (Hint: your answer to part (a) might be useful in following the execution to get the right answer.)

```
B::g
A::f
---
C::g
B::h
C::f
---
C::g
B::h
```

Question 7. (12 points) Write a Ruby program that reads text from standard input and searches for words in the input. The first line of the input is the list of words to look for. The remaining lines should be examined to see if they contain any of the words in the given list. If any of the words are found, the line containing them should be printed out preceded by its line number. For example, if the input is

```
how now brown cow
old macdonald had a farm
e-i-e-i-o
that farm was on a brown dirt road
and on that farm he had a cow
cowabunga said old macdonald
```

the output should be

```
3: that farm was on a brown dirt road
4: and on that farm he had a cow
```

since line 3 contains the word "brown" and line 4 contains the word "cow". Line 5 is not included in the output because none of the words in the search list match a word in that line, even though "cow" is a substring of "cowabunga". The initial line containing the words to search for is numbered as line 0; successive lines are numbered from 1. Each input line should be printed at most once in the output, even if it contains several copies of words in the search list.

To simplify things, you should read the input one line at a time with `gets` and use the `String split` method to break each line into words. The basic behavior of `split` is to return an array of the words, where words are defined as strings of characters separated by whitespace. Example:

```
"one two three".split   =>  ["one", "two", "three"]
```

To keep the problem simple, assume that there is no extra leading or trailing whitespace in the input lines, and assume that all words in the input contain only lower case letters, so you don't have to deal with issues of punctuation or capitalization.

For full credit you should use Ruby iterators like `each` to process the contents of containers like arrays and hashes. Recall that if `h` is a hash, you can iterate through its key/value pairs with

```
h.each {|key, value| ... }
```

Write your code on the next page. If you find it helpful, you can remove this page from the exam for reference while you are working.

Next page please….

Question 7. (cont.) Code for the word search program:

There are many possible solutions, of course, and as long as your solution used Ruby appropriately and produced the right answer it received credit.

Many people stored the list of search words in an array and compared each word in the input against every word in the list. A more efficient solution is to store the search words in a hash and look up each input word directly.

```
words = gets.split
dict = {}
words.each{|w| dict[w] = true}
lnum = 1
while line = gets
   line.split.each do |w|
      if dict[w]
         puts lnum + ": " + line
         break
      end
   end
   lnum++
end
```

Question 8. (20 points) One of our clients would like us to extend the calculator language and interpreter to add a function to compute the maximum value of one or more expressions. For instance, `max(1+2)` would evaluate to 3, and `max(17, 3-42, 5)` would evaluate to 17. Of course variables could also appear in expressions, as in the original calculator language. For reference, here is the original calculator grammar (although you may not need to use it):

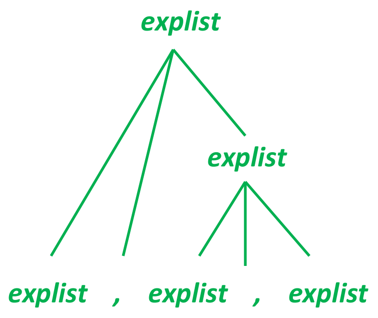
program ::= *statement* | *program statement*
statement ::= *exp* | *id* = *exp* | `unset` *id* | `list` | `quit` | `exit`
exp ::= *term* | *exp* + *term* | *exp* – *term*
term ::= *power* | *term* * *power* | *term* / *power*
power ::= *factor* | *factor* ** *power*
factor ::= *id* | *number* | (*exp*)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

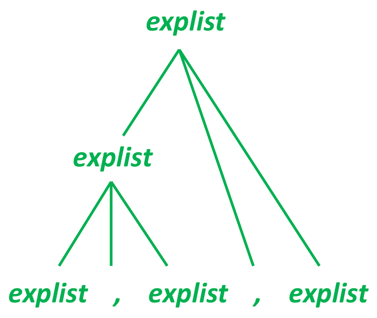
One of those pesky summer interns proposes adding this new `max` function to the calculator language by adding an alternative to the rule for *factor* and adding an additional rule to handle the `max` function parameter lists:

factor ::= ... | `max` (*explist*)
explist ::= *explist* , *explist* | *exp*

(a) Show that the original grammar with these additions is now ambiguous. (Hint: you only need to show ambiguity for a part of a grammar that has a problem, not necessarily for all of the grammar rules.)

All that is needed is to show two different parse trees for something that can be derived from the grammar, or, alternatively, to give two distinct leftmost or rightmost derivations for something. Here are two parse trees for *explist* , *explist* , *explist* .





(continued next page)

Question 8. (cont.) (b) Give a different grammar rule or rules that add the `max` function to the rule for *factor* and handle its parameter list, but that are not ambiguous.

> ***factor* ::= … | `max` (*explist*)**
> ***explist* ::= *exp* | *explist* , *exp***

A right-recursive rule would work equally well to remove the ambiguity, and would actually be preferable for a recursive-descent parser.

(c) We would like to add this new `max` function to our calculator interpreter. What changes, if any, need to be made to the original calculator scanner? Describe any necessary changes to the scanner, the set of tokens, or anything else that must be modified. You do not need to give the implementation, just describe the needed changes.

The new grammar rules add two new token classes to the lexical grammar: `max` and comma. The scanner needs to be modified to recognize them and return an appropriate token object whose kind is "max" or "," when it scans them.

(d) On the next page, write the additional Ruby code needed in method `factor` and any other necessary code needed to parse and evaluate this new `max` function and its parameter list. Your answer should be guided by your solutions to parts (b) and (c), but, as usual, adjusted as needed for use in a recursive-descent parser. You should make the following assumptions:

- The scanner has been modified as describe above in your solution to part (c). You can call the scanner's `next_token` method whenever you need to get the next token from the input. If for some reason you need it, the `kind` method of a `Token` object returns strings containing the token text like "list", "sqrt", "+".
- There is a global variable `current_token` that contains the next unprocessed `Token` in the input at all times. Your code needs to update this variable appropriately as it parses the input.
- Functions like `exp` exist to parse each grammar nonterminal and return its value, if any.
- You should assume that there are no syntax errors, missing or extra tokens, or other problems in the calculator input.

Write your solution on the next page.

Question 8. (cont.) (d) Below, write your modifications to method *factor* and any other code needed to parse and evaluate the new `max` function expression.

This solution assumes the parser contains instance variables named `@scan` and `@current_token` referring to the scanner itself and the current token. We made allowances if your code was clear but not exactly the same.

The following code should be inserted in `factor`:

```
if @current_token.kind == "max"
   @scan.next_token                      # skip "max"
   @current_token = @scan.next_token     # skip "("
   current_max = exp                     # max = first expression
   while @current_token.kind == ","
      @current_token = @scan.next_token  # skip ","
      next_exp = exp                     # parse next exp
      if next_exp > current_max          # update max if needed
         current_max = next_exp
      end
   end
   @current_token = @scan.next_token     # skip ")"
   current_max                           # return value of factor
end
```

A couple of short-answer questions to wrap up. Please keep your answers brief and to the point (and legible!!). Your readers thank you.

Question 9. (5 points) When we looked at automatic storage management, one of the algorithms we discussed was Generational Garbage Collection. In the basic version of this scheme, the heap is divided into two parts, a "nursery", which is the space where new objects are allocated, and the rest of the heap where long-lived objects are moved after a number of garbage collections. What is the rationale for splitting the heap this way? What advantage does it provide, if any?

A generational garbage collector takes advantage of the observation that most objects have short lifetimes and objects that survive several garbage collections are likely to remain reachable for a long time. By scanning the nursery frequently, the collector recovers a significantly higher percentage of objects for the effort expended than it would if it scanned the entire heap each time. The full heap still needs to be collected occasionally to reclaim long-lived objects that do become unreachable.

Question 10. (5 points) Java includes *interfaces* that can be used to specify a collection of methods and constants. Ruby doesn't contain anything similar. Would it make any sense to add interfaces to Ruby as a new language feature? Why or why not?

No. Interfaces are all about static type checking, while Ruby is a dynamically typed language. The purpose of interfaces in Java is to specify types for classes and objects so the implementation can perform static type checking and guarantee there will be no "message not understood" errors during execution. Ruby implementations do not (and cannot) perform static type checking.