CSE 413 Final Exam

December 11, 2008

Name ________Sample Solution__________

The exam is closed book, except that you may have a single page of hand-written notes for reference plus the page of notes you had for the midterm (although you are unlikely to need that one).

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	/ 12
2	/ 7
3	/ 5
4	/ 7
5	/ 12
6	/ 12
7	/ 15
8	/ 10
9	/ 10
10	/ 10
Total	/ 100

Question 1. (12 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 5 b's" instead of just transcribing the expressions from regular expression notation into English.

(a) (a|b)* c (a|b)* c (a|b)*

All strings of a's, b's, and c's with exactly two c's.

(b) (x*y*)* xx (x|y)*

All strings of x's and y's that have at least one pair of adjacent x's.

Question 2. (7 points) Regular expressions II. Write a regular expression or set of regular expressions that generates all decimal numbers (digits 0-9) with an integer and fraction part, but no exponent, formed according to these rules:

- There must be at least one digit before and after the decimal point.
- There may not be any leading zeros in the integer part or trailing zeros in the fraction part unless that part of the number consists of a single zero.

Examples of legal numbers according to these rules: 3.1415926535, 0.015, 170.0, 0.0, 500.001, 0.5

Example of illegal numbers: 17 (no decimal point), 17. (no digits after the decimal point), .01 (no digits before the decimal point), 012.5 (leading 0), 1.0000 (trailing zeros), 0.250 (trailing zero in the fraction part – the 0 in the integer part is ok).

Fine print: You may use basic regular expressions (sequences rs, choice r|s, and repetition r* and, of course, 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.

(0 | [1-9][0-9]*) . (0 | [0-9]*[1-9])

Question 3. (5 points) A scanner for Java, C, or C++ has to deal with identifiers and various operators including `+`, `-`, `++`, `--`, `=`, `+=`, `-=`, `==`, `!=`, and so forth. Recalling that a scanner uses the principle of longest match when splitting input characters into tokens, show how the following string is split into tokens. Draw a box around each character or sequence of characters that form a token in the following input. You should assume that these characters are adjacent in the input with no spaces in between. The first two boxes are drawn for you.

(Remember that the tokens don't need to make up a meaningful or legal program. The scanner divides the input characters into tokens without worrying about context.)

$$\boxed{a}\boxed{+}\boxed{\texttt{cow}}\boxed{+=}\boxed{=}\boxed{d}\boxed{--}\boxed{9}$$

Question 4. (7 points) Give a context free grammar that generates all strings containing zero or more a's, b's, and c's that have twice as many a's as b's (if there are any a's and b's in the string).

Obviously there are many possible solutions that will work. Here is one.

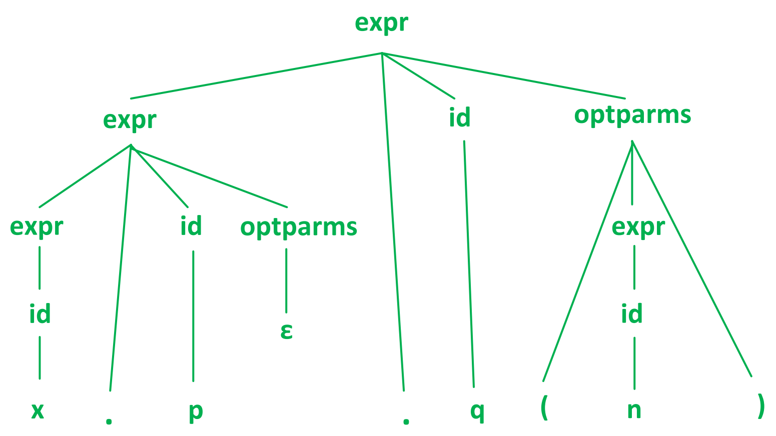
S ::= S a S a S b S | S a S b S a S | S b S a S a S | c | S S | ε

Question 5. (12 points) Here is a fragment of a grammar for method calls in a language somewhat like Java or Ruby. A method call has an optional parameter list with a single parameter.

> *expr* ::= *id* | *expr* **.** *id optparms*
> *optparms* ::= ε | (*expr*)

The non-terminal *id* can derive any single lower-case letter `a, b, c, …, z`. The symbol ε is the Greek letter epsilon, denoting the empty string.

(a) Draw the full parse tree for the expression `x.p.q(n)`.



(b) Give the leftmost derivation of the expression `x.p.q(n)` as diagramed in part (a).

expr → expr . id optparms
→ expr . id optparms . id optparms
→ id . id optparms . id optparms
→ x . id optparms . id optparms
→ x . p optparms . id optparms
→ x . p . id optparms
→ x . p . q optparms
→ x . p . q (expr)
→ x . p . q (id)
→ x . p . q (n)

Question 6. (12 points) Your customers aren't too happy with a programming language that only supports method calls with a single parameter. One of the interns on your staff suggests changing the grammar from the previous problem as follows, to optionally allow multiple expressions in a parameter list separated by commas.

> *expr* ::= *id* | *expr* **.** *id optparms*
> *optparms* ::= ε | (*exprs*)
> *exprs* ::= *expr* | *exprs* **,** *exprs*

(a) Show that this proposed extension results in an ambiguous grammar.

It's sufficient to show that just part of the grammar is ambiguous, in this case the last rule. Here are two distinct leftmost derivations of x, x, x.

expr → exprs , exprs → exprs , exprs , exprs → expr , exprs , exprs → id , exprs , exprs → x , exprs , exprs → x , expr , exprs → x , id , exprs → x , x , exprs → x , x , expr → x , x , id → x , x , x

expr → exprs , exprs → expr , exprs → id , exprs → x , exprs → x , exprs , exprs → x , expr , exprs → x , id , exprs → x , x , exprs → x , x , expr → x , x , id → x , x , x

(Actually, most solutions just showed two different parse trees for something like x, x, x. That is probably an easier way to visualize the ambiguity, but your instructor is too lazy to do battle with the computer drawing tools again to do that here. Left as an exercise, as the cliché goes.)

(b) Describe how to fix this proposed grammar to get rid of the ambiguity and still allow more than one expression in a parameter list, with commas separating multiple expressions if there are more than one.

Rewrite the last rule in the grammar to use either left- or right-recursion:

> **exprs ::= expr | exprs , expr**

or

> **exprs ::= expr | expr , exprs**

Enough theory for the moment! Time for some Ruby hacking.

Question 7. (15 points) Write a Ruby program that reads text from standard input and reports the word that occurs most frequently in the input and how often it occurs. For example, if the input is

```
to be or not to be
to do is to be
do be do be do
```

the output should be

```
be  5
```

since the word “be” occurs 5 times and no other word occurs that frequently. If several words have the same number of occurrences and they are the most frequent, your program should arbitrarily pick one of them and print it and the number of times it appears. For instance, if the input is

```
one two one two three
```

the program could print either

```
one  2
```

or

```
two  2
```

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.

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

```
freq = {}   # word=>frequency hash table. Could also use Hash.new

# read input and count words
while line = gets
  words = line.split
  words.each do | w |         # Could also use { ... } for blocks
    if freq[w]
      freq[w] += 1
    else
      freq[w] = 1
    end
  end
end

# find most frequent word and print it
max_freq = 0
max_word = ""    # not strictly needed, but
                 # ensures not null if no input
freq.each do | word, n |
  if n > max_freq
    max_freq = n
    max_word = word
  end
end

print max_word, "  ", max_freq    # puts also ok
```

Question 8. (10 points) We would like to extend the calculator language from the interpreter project by adding a conditional expression as follows. The additional expression is

condexp ::= `if` *exp*$_1$ `then` *exp*$_2$ `else` *exp*$_3$ `end`

The meaning of the conditional expression is that all three expressions, *exp*$_1$, *exp*$_2$, and *exp*$_3$ are evaluated, then if *exp*$_1$ has a value that is not 0, the result of the conditional expression is the value of *exp*$_2$, otherwise its value is the value of *exp*$_3$.

Your job is to write a method `condexp` to parse and evaluate this conditional expression (and the solution is shorter than the question!). You should make the following assumptions in your solution:

- The interpreter already contains a method `exp` to parse and return the value of expressions, and it has been modified to call `condexp` when it needs to process a conditional expression.
- The scanner has been modified to recognize the keywords `if`, `then`, `else`, and `end` and return these as separate tokens. 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 the strings "if", "then", "else", and "end" for these tokens.
- There is a global variable `current_token` that contains the "if" `Token` that begins a conditional expression at the moment `condexp` is called. Method `condexp` is responsible for advancing through the input and leaving the token following the conditional expression in `current_token` when `condexp` returns.
- You should assume that there are no syntax errors, missing or extra tokens, or other problems in a conditional expression.

Write your `condexp` method on the next page. You can remove this page for reference if it makes things easier while you are working.

Question 8. (cont.) Complete the definition of `condexp` below.

```
# parse and evaluate condexp ::= if exp1 then exp2 else exp3 end

def condexp
    current_token = next_token      # skip if
    exp1 = exp
    current_token = next_token      # skip then
    exp2 = exp
    current_token = next_token      # skip else
    exp3 = exp
    current_token = next_token      # skip end
    if exp1 != 0
      exp2
    else
      exp3
    end
end
```

Notice that we don't actually have to use an explicit `return`, since the `if` expression is the last one in the method, and the value of an `if` expression is the value of the last expression evaluated in whichever branch is executed. But it's fine if you did use `return`.

A couple of notes about common errors:

- **It is necessary to actually call exp recursively to parse (and evaluate) both branches of the conditional expression. It is not sufficient just to scan forward looking for an "else" or "end" token, or just skip ahead an extra token or two, since the subexpressions may be arbitrarily complicated, and might contain nested conditional expressions. (This is also why the problem specified that both exp_2 and exp_3 should be evaluated. The way the interpreter was designed, there is no mechanism to parse an expression without evaluating it.)**

- **The invariant is that all parser methods should assume that `current_token` is the first token of whatever it is that they are to parse, and they should consume whatever they parse and leave the first token of whatever follows in `current_token` when they are done. So it is not enough just to call `next_token` to skip the keywords without setting `current_token`.**

The next couple of pages are short-answer questions about type systems and languages. Please keep your answers brief and to the point (and legible!!). Your readers thank you.

Some representative answers are given below. Other answers were possible for full credit, but they had to be specific and bring up important points, not incidental ones.

Question 9. (10 points) Ruby is a dynamically typed language ("duck typing"). Languages like Java and C++ are statically typed languages.

(a) Describe one distinct advantage that static type systems have over dynamic typing.

A couple of possibilities

- **Able to detect a variety of errors by examining programs without executing them. In particular, static type systems for object-oriented languages can ensure that all objects have methods to respond to messages sent to them, and that methods are called with the correct number and types of parameters.**
- **The additional information available from the type system can allow the compiler to better optimize or special-case the generated code to run more efficiently than can be done when the type information is unknown until the program is executed.**

(b) Describe one distinct advantage that dynamic typing has over static type systems.

- **The biggest one is that it allows code to be reused and adapted to situations that it was not originally designed for.**
- **Another advantage is that it is easier to prototype and incrementally develop code because more details can be omitted or postponed until later in the project.**

Question 10. (10 points) Various languages have different approaches to allowing new classes to be created from other classes, modules, or interfaces. For instance, C++ allows multiple inheritance, where a new class can be defined as a subclass of more than one other class. Java only allows a class to have a single superclass, but allows it to implement multiple interfaces.

(a) Describe one advantage of the C++ approach allowing multiple superclasses, compared to the Java approach of a single superclass plus multiple interfaces.

The biggest advantage is that it allows a new class to inherit implementations (not just specifications) from more than one existing class.

It also may allow the code to more naturally model the problem domain and the interactions between different kinds of objects and abstractions in it.

(b) Describe one advantage of the Java approach restricting classes to a single superclass plus optionally implementing multiple interfaces, compared to the C++ approach that allows multiple superclasses.

Simplicity at several levels. Programs that use multiple inheritance extensively are prone to having complex, brittle relationships between classes. The Java approach avoids many of these problems. It also allows a simpler implementation, since we don't need to deal with issues like object layout and method dispatch for objects that inherit implementations from more than one immediate ancestor class.