CS59-F16 Midterm

Terms and Conditions. This midterm is open-book, open-shell, open-Internet (in your submission, make a note of the resources you used). You are allowed to discuss tools and techniques with your classmates on the class list, on Slack at `https://cs59-fall16.slack.com`, or in person (in the latter case, please make a note of your discussion). The one rule you must abide by is, **Do not disclose actual solutions or their parts.**

Languages: The intended language for these exercises is LISP. You may also use Ruby as "glue", and for Problems 2–4—but LISP-y recursive style is expected throughout, and LISP is the most natural way of working with input S-expressions. With Ruby, you'd need to devise some way of reading these in first, which will likely add to your debugging workload.

What to submit: Submit your work as an ASCII *plain text* file named `midterm-DDDD.txt` where `DDDD` are the last four symbols of your ID. Submit the file by emailing it as an attachment to me.

Make sure that your functions are *named* exactly as specified in the problems, and take exactly the kinds of arguments the problem describes. Check the midterm directory for further submission instructions before submitting.

When to submit: by noon of Saturday October 29.

Problem 1. *Parsing the classics*[1]

You are given LISP lists (sexps) that represent grammatical parses of English sentences. You will find examples of such inputs in `midterm/opennlp`, and I will generate them on request for any phrases you send me (as plain ASCII text, one sentence per line). In these parse trees, every word is tagged with its part-of-speech tag, and each group of related words (a phrase) with its phrase tag.

An important part of these parse trees are noun phrases, marked with the NP tag. NPs can nest several levels deep, and contain phrases or clauses that additionally describe (or, as linguists say, *modify*) the noun.

a) Write a function that takes a sexp of a sentence parse and outputs its representation in the Graphviz language (see Addendum A for the summary of the Graphviz format and how to run the tool). This function will help you debug tasks (b) and (c).

b) Write a function `extract-nps` that takes a sexp of a sentence parse and returns a (flat) list of strings corresponding to the noun phrases occurring in the sentence. Your output should be English text with no tags.

c) Write a function `match-pattern` that takes a sexp of a sentence parse and another sexp that represents a fragment of a parse, and is allowed to have `*` instead of one or more nodes. We'll call this second sexp a *pattern*. The function should match the pattern against every node of the sentence, and return a list of all nodes that match. All words and tags in the pattern must match exactly, cons cell for cons cell, with one exception: `*` matches any node and any subtree beneath it (but not the node's siblings). In case of no match, the function should return `nil`.

For example:

```
(match-pattern '(TOP (S (CC And) (S (NP (PRP we)) (VP (VBD saw) (NP (PRP him)))) (. !)))
               '((NN *)))
```

[1] Well, children classics, not the Greek and Latin kind.

```
(match-pattern '(TOP (NP (NP (DT The) (NN cat)) (PP (IN in) (NP (DT the) (NN hat))) (. !)))
               '((NN *)))
```

produce `nil` and `((NN cat) (NN hat))` respectively. `(NP *)` would match `(NP (PRP we))`, `(NP (PRP him))` in the first example sentence, and `(NP (NP (DT The) (NN cat)))` in the second. To match `(NP (DT the) (NN hat))` you'd need to use `(NP * *)`.

(Extra credit) Introduce a `**` pattern that would match any node and all of its sibling nodes after the first match. E.g., `(NP **)` will match all of

```
(NP (PRP we))
(NP (PRP him))
(NP (NP (DT The) (NN cat)) (PP (IN in) (NP (DT the) (NN hat))))
(NP (DT The) (NN cat))
(NP (DT the) (NN hat))
```

Addendum A *Creating diagrams with Graphviz*

Graphviz `http://www.graphviz.org/` is a free tool released by AT&T Research.[2] Graphviz provides a simple way to render nice-looking pictures of graphs by specifying them in a very simple language. At its simplest, the language of Graphviz (called DOT) is almost trivial—and yet powerful.

As an example, see the Unix Family Tree in the Graphviz Gallery, `http://www.graphviz.org/content/unix`. First look at the graph (it's "almost" a tree, except for some extra arrows connecting a few nodes with their non-immediate descendants), then click on the picture to see the code.[3]

You won't need more Graphviz language than in these examples. Just be aware that node names for different nodes must be different, and if you want two different nodes with the same label, you'll need to give the nodes different names and explicitly specify their labels. For example, to modify the Hello World example `http://www.graphviz.org/content/hello`:

```
digraph G{  node1 -> node2 ; node1 [label="Hello"];  node2 [label="Hello"]; }
```

Graphviz can draw much more complex graphs than just trees; see `http://www.graphviz.org/Gallery.php` for some examples.

Note: As you can imagine, some LISP packages for this tool already exist. However, they are quite complex, as they seek to take full advantage of Graphviz capabilities. Your code can (and should) be much simpler.

Problem 2. *Evaluate Ruby formulas*

Ruby (starting with 2.0) can output the S-expressions (sexps) that its parser produces for programs. See warm-up for an example of how to obtain the sexp for an arithmetic expression.

Assume you are given a sexp resulting from a Ruby program that contains a mix of algebraic formulas and variable assignments. For example:

[2] You can install Graphviz with `port install graphviz` on MacPorts, `apt-get install graphviz` on Ubuntu or Debian Linux.

[3] The command to make this graph as a PNG graphics file is `dot -Tpng unix.gv.txt > unix.png`. You can also use the `neato` tool instead of the `dot` tool; it doesn't work so well on this example, but excels at `http://www.graphviz.org/content/traffic_lights`. Remember to redirect the output of `dot` to a file; otherwise the binary contents of the generated image will be written to your terminal!

```
require 'ripper'
require 'pp'
code = "x = 5; y = x - 4; x**3 + 3*x*x*y + 3*x*y*y + y**3"
pp Ripper.sexp(code)
```

Recall that in Ruby `x**n` means raising to a power, x^n. Accordingly, `eval(code)` would return 216.

a) Write a function **evaluate-ruby-sexp** that, when given such a sexp derived from such a program as `code` above, will evaluate it and return the value. Your program should return `:error` when a variable is used before it is assigned. You program should similarly reject sexp programs that are not a series of assignments or algebraic formulas.

b) (Extra credit)

For further credit, write a function `evaluate-or-simplify` that allows programs where some variables are not assigned to. In such cases, the program treats these variables as unknown algebraic quantities, and outputs a formula in them (as a string of Ruby code). Try to maximally simplify the output formula. For example,

```
code <<CODE
x = 2 + y
x*x + 2*x*y - 4*y + y*y
CODE
```

should produce `4 + 4*y + 4*y*y` (or `4*y*y + 4*y + 4`, or an equivalent formula that cannot be further simplified.)

Your function will be a simple computer algebra system. For more information on computer algebra, see chapter 2.5.3 of *Structure and Interpretation of Computer Programs* (2nd ed.) by Harold Abelson and Gerald Jay Sussman with Julie Sussman, `https://mitpress.mit.edu/sicp/`.

Problem 3. *Spiral, spiral on the wall*

Write a (tail-recursive) function **count-digits** that calculates how many times each digit occurs in the 100000th Fibonacci number. Fibonacci numbers are defined as the recursive sequence

$$F(n) = F(n-1) + F(n-2)$$

where $F(1) = 1, F(2) = 1$.

Use a Common Lisp with tail-call optimization such as the Steel Bank Common Lisp (SBCL), or Ruby (with tail-call optimization turned on).

Your function should return a list of 10 counts, corresponding to the numbers of occurrences of 0, 1, 2, ..., 9.

You are only allowed one `defun` or `def` in this problem. Use it wisely.

Problem 4. *Mysterious bindings*

For this problem, you'll need Emacs version 24 or later. If you cannot install such a version, let me know, and I will attempt to provide a variant of this problem for you.

The file `mystery.elc` in the midterm directory contains a bytecode-compiled Emacs Elisp function `make-mystery`. Download the file and load it in a buffer running Emacs' `lisp-interaction-mode`, with `(load-file "mystery.elc")`.[4]

[4]It's best to make a new directory, download the file there, and start Emacs in the same directory; otherwise, you'll need to modify the above command to add the file path.

Call it, and you should see it print “No secrets.”

Your objective is to write and run some Elisp code that calls this `make-mystery` function and results in it printing “Secret found!”

From my transcript:

```
(load-file "mystery.elc")
(make-mystery)
"No secrets."

;
;  Do something special here
;
(make-mystery)
"Secret found!"
```