CS59-S16 Make-Up 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.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 or Scheme. You may also use Ruby as "glue", and for Problems 2 and 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: Please follow the formatting specified in `midterm/submission-instructions.txt` in the class directory.

Make sure to name your functions as specified in the problems!

When to submit: by 5pm of Saturday May 14.

Problem 1. *Heads in the Trees*

Assume you are given LISP lists 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. In these noun phrases, one word—usually a noun, but not always—is called the *head word*. This word determines what the phrase is about; the other words are said to *modify* this word. For example, in the noun phrase "a cute fluffy kitten" kitten is the head word.

Note that `NP`s can nest several levels deep, and each may have its own head word!

In most English noun phrases, the head word can be correctly located by using a series of simple rules that match the word and phrase tags. These rules[1] are as follows:

```
(1) If the last word is tagged POS, return thelastword
(2) Else search from right to left for the first child which is an NN, NNP, NNPS,
     NNS, NX, POS, or JJR;
(3) Else search from left to right for the first child which is an NP;
(4) Else search from right to left for the first child which is a $, ADJP or PRN;
(5) Else search from right to left for the first child which is a CD;
(6) Else search from right to left for the first child which is a JJ, JJS, RB,
     or QP;
(7) Else return the last word.
```

Write a (recursive) function **add-heads** that inputs a sexp representing a parse and returns a sexp with the same parse, but with head words additionally tagged as HEAD. For example,

[1] These rules were originally developed by D. Magerman in his 1994 Ph.D. thesis *"Natural Language Parsing as Statistical Pattern Recognition"* and improved by M. Collins in *"HeadDriven Statistical Models For Natural Language Parsing"*, 1999, also a Ph.D thesis. These days, these rules are a classic, but they took serious research.

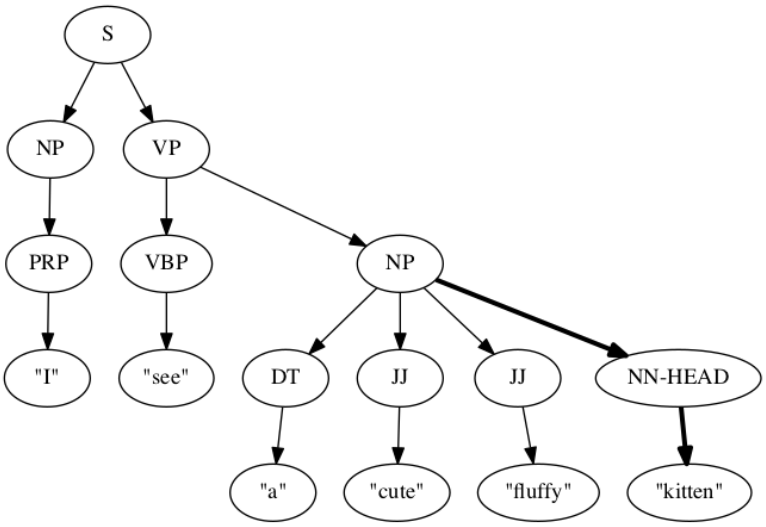
```
(TOP (S (NP (PRP I))
        (VP (VBP see)
            (NP (DT a) (JJ cute) (JJ fluffy) (NN kitten)))))
```

becomes

```
(TOP (S (NP (PRP I))
        (VP (VBP see)
            (NP (DT a) (JJ cute) (JJ fluffy) (NN-HEAD kitten)))))
```

(so annotated by rule (2) above; the rule is correct because the phrase is indeed about a kitten).

Additionally, write a (recursive) function **parse-to-dot** that produces the graphical representation of the parse tree in the *Graphviz* format. For example, the above phrase will be



This function will be useful for debugging your **add-heads**, so you may want to start with it.

Background: 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]

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 overwrite their labels. For example, to modify the Hello World example `http://www.graphviz.org/content/hello`:

```
digraph G{  hello -> world ; hello [label="Hello"];  world [label="Hello"]; }
```

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

---

[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!

Problem 2. *Anything but V!*

Ruby (starting with 2.0) can output the S-expressions (sexps) that its parser produces for programs. See `warmup/expr2sexp.rb` for an example of how to obtain the sexp for a Ruby program.

Assume you are given a sexp parse of a Ruby program. Assume this program contains some variables. Your task is to produce the sexp for an *equivalent* program, in which all variables, whatever their names, are renamed to `v1`, `v2`, etc., so that *no two variables that aren't the same* have the same name.

For example, the sexp representing the program

```
x = 3
x.times do |x|
  puts x + x
end
```

becomes a sexp representing the program

```
v1 = 3
v1.times do |v2|
  puts v2 + v2
end
```

Write the function **rename-vars** that implements this task.

For extra credit, write the function **sexp-to-ruby** that takes the transformed sexp and converts it back into a Ruby program.

Problem 3. *Ruby on a LISP Island*

You find yourself stranded on a remote uninhabited island. You have some parsed sexps of Ruby arithmetic expressions in your luggage, and your survival depends on evaluating them—but your Ruby was lost in the shipwreck. Luckily, after some digging around in the jungle, you manage to find a LISP interpreter!

Write a LISP function **ruby-sexp-to-lisp** that converts any sexp representations of Ruby arithmetic expressions (produced as in Problem 2, e.g., by `warmup/expr2sexp.rb`) to an equivalent LISP program, which, when evaluated, gives the same result. Output the sexp for the LISP program.

Problem 4. *Fibonacci, Meet Factorial*

Factorials are a fixture of web examples for writing a tail-recursive function, and so are Fibonacci numbers. You are probably already yawning as you read this. Let's see if we can make them more interesting :)

Write a (tail-recursive) function that outputs the index of the Fibonacci number that is less than $N!$ for $N \geq 100000$, but the *next* Fibonacci number is larger. For your final answer, your program is allowed only one `defun`, and should be good for any N it gets as an argument (assuming $N!$ fits in your RAM).

Count Fibonacci numbers from 0, so that $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, etc. Thus $3! = 6$ falls between $F_5 = 5$ and $F_6 = 8$, and your function should output 5 for the input $N = 3$.

Use a Common Lisp with tail-call optimization such as the Steel Bank Common Lisp (SBCL) or GCL, or Ruby ≥ 2.0 (with tail-call optimization turned on).

Problem 5. *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 `more-mystery.elc` in the midterm directory contains a bytecode-compiled Emacs Elisp function `more-mystery`. Download the file and load it in a buffer running Emacs' `lisp-interaction-mode`, with `(load-file "more-mystery.elc")`.[4]

Call the function `more-mystery`. It will check your Elisp environment and will print "Not quite" by default, "Warmer" if you are getting closer to the solution, and "You got it!" if you created what it's looking for.

Your objective is to write and run some Elisp code that calls this `more-mystery` function and results in it printing "You got it!"

Note: This compiled bytecode uses some functions specific to Elisp and (somewhat) obfuscates its checks. Expect to read some Elisp manual pages to solve this puzzle.

[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.