CS59-S16 Midterm

Terms and Conditions. This midterm is open-book, open-shell, open-Internet (in your submission, make a note 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. 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. Additionally, for *Problem 4*, submit your result file as a ZIP archive named `midterm-DDDD-p4.txt.gz`, also as an attachment.[1]

Make sure to name your functions as specified in the problems. Check the midterm directory for further submission instructions before submitting.

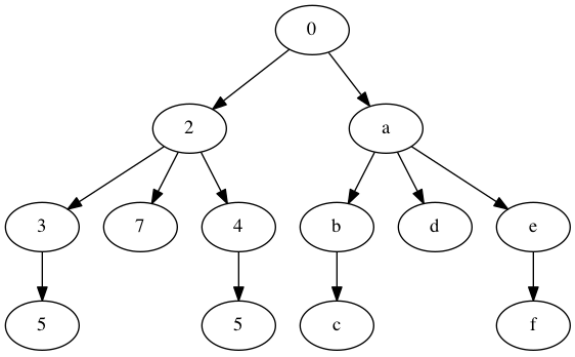
When to submit: by class time on Tuesday May 3.

Problem 1. *Drawing Trees*

Assume you are given LISP lists that represent trees as in the warm-up exercises. For example,

```
(0 (2 (3 (5)) (7) (4 (5))) (a (b (c)) (d) (e (f))))
```

should produce



Write a (recursive) function **list-to-dot** that takes such a list and produces the representation of a tree in the *Graphviz* format.

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

[1] You can create this file by running `gzip midterm-DDDD-p4.txt` from a Unix shell.

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

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 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.

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. *Simplify 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 parse of an algebraic formula. For example:

```
require 'ripper'
require 'pp'
code = "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. Thus the formula above is $x^3 + 3x^2y + 3xy^2 + y^3$.

Write a function **simplify** that, when given such a sexp derived from a formula with one or more variables $(x, y, ...)$, and an integer, substitutes that integer for x and simplifies the resulting algebraic expression—and returns the simplified sexp.

For example, given the sexp for the above formula and $x = 2$, `simplify` should produce a Ruby parsed sexp equivalent to $y^3 + 6y^2 + 12y + 8$ (the terms in the returned sexp can be in any order).

Extra credit: Make it so that when `simplify` is given a list of dotted pairs of variable names and their values for its second argument (instead of an integer), it will simplify the expression as much as possible. For example, the second argument of `((x . 2) (y . 3))` should produce a sexp representing the number 125 (i.e., `(:program, ((:@int, "125", (1, 0))))`)

Throughout, you can ignore the line and column numbers, so the above return value could be `(:program, ((:@int, "125")))`)

For further credit, make your function process multiline programs with assignments. For example, a program with

```
code <<CODE
x = 2
```

[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!

```
y = x*x - 1
a = x**3 + 3*x*x*y + 3*x*y*y + y**3
CODE
```

should produce $x = 2, y = 3, a = 125$. Your program should reject sexp programs that are not a series of assignments of algebraic formulas to variables.

Problem 3. *Parse like Master Yoda*

Master Yoda is a major character in the Star Wars movies. He is an alien, and a master of the Force, which makes him, among other things, a powerful fighter.

Yoda speaks fluent Galactic Basic (represented in the movies by English), but something about his native grammar makes him invert the word order in each sentence and clause, putting the verb at the end of each. Thus "Your rule is at an end, and it was too long already" becomes "At and end your rule is, and too long already, it was".[4]

Yoda inspired a number of Internet sites that transform English sentences, such as `http://www.yodaspeak.co.uk/`

Assume you are given parsed English sentences as sexps (see the `yoda/` subdirectory for samples). Write a function **yodify** that prints Yoda's version of these sentences. You may ignore capitalization and punctuation.

Problem 4. *Recursive Telephones*

Write a (tail-recursive) function **telephone** that computes the 100000th number of the sequence given by the following recursive relation:

$$T(n) = T(n-1) + (n-1) \cdot T(n-2)$$

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

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

In addition to your code, turn in a **compressed** file containing this number (and nothing but this number and whitespace). Name the file as described in *Terms and Conditions* above.

Note: These numbers are known as the *telephone numbers* in mathematics. Among other things, they represent the number of patterns that n telephone subscribers can be connected with each other at the same time—assuming that connections are only non-intersecting pair-wise (no "conference calls" of three or more participants, each subscriber is on one call or none).

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 `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")`.[5]

This function contains dynamically bound variables. Call it, and you should see it print "No secrets."

[4] This phenomenon is also observed in humans, and is apparently called *Interlanguage fossilization.*

[5] 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.

Your objective is to write and run some Elisp code that calls this `make-mystery` function and results in it printing "Secret found!" This can be done by setting the value of one of the dynamically bound variables used by the function to 31337.