CS59-F16 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-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 languages for these exercises are LISP and Ruby. LISP fits better with processing S-expressions (with Ruby, you'd need to devise some way of reading these in first, which will likely add to your debugging workload), Ruby may work best for you when text requires transformation. You can use Ruby as the glue language throughout. If writing your solution in Ruby, use _ instead of - in your function names.

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 November 5, 2016.

Problem 1. *Visualizing Ruby parses*

Ruby (starting with 2.0) can output the S-expressions (sexps) that its parser produces for code. These expressions are Ruby's lists containing other lists, symbols (starting, as per Ruby convention, with :), strings, and other elements, such as pairs of `(row, column)` locators of the sexp part in the original Ruby source.

These sexps can be fairly easily converted to a form readable for LISP (see examples below).

```
#  This is a quick-and-very-dirty hack! If strings in the sexp
#   contain []s, commas, newlines, etc., this will break.
#   Proper escaping will be needed, which this simple hack
#   will break.
# Add \n to the first string argument of tr to suppress newlines
#   in the printed sexp.
require 'ripper'
require 'pp'
code = <<CODE
x = 5
y = x - 4
x*x + 2*x*y + y**2
CODE
puts Ripper.sexp(code).pretty_inspect.tr("[],", "()  ")
```

or

```
require 'ripper'
require 'pp'
code = "lambda {|x| x+5}.call 6"
puts Ripper.sexp(code).pretty_inspect.tr("[],", "()  ")
```

or

```
require 'ripper'
require 'pp'
code = "10.times do |x| puts x ; end"
puts Ripper.sexp(code).pretty_inspect.tr("[],", "()  ")
```

Assume you are given a sexp resulting from a Ruby program that does not contain literal strings with any special symbols such as `'['`, `']'`, `','` and others that would require special escaping when being converted to LISP with a simple hack as above.

a) Write a function that takes a sexp of a Ruby program 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-blocks` that gets a program sexp as above and returns a list of all its subexpressions that represent Ruby blocks and lambdas. Write a function `block-variables` that takes such an expression and returns the list of its formal parameters.[1]

c) Write a function `find-free-variables` that takes a Ruby program sexp and returns a list of variables that are free in that program (and would cause "undefined variable" runtime errors if not additionally defined).

For example, `y` is free in `10.times do |x| puts x+y ; end`, and causes an error; x is not free.

```
irb(main):001:0> 10.times do |x| puts x+y ; end
NameError: undefined local variable or method `y' for main:Object
from (irb):1:in `block in irb_binding'
from (irb):1:in `times'
from (irb):1
from /usr/bin/irb:12:in `<main>'
```

Given the sexp for the above problem, `find-free-variables` should return `["y"]`.

(Extra credit) Write a pattern-matcher `match-pattern` for Ruby sexps similar to that of the problem *3c)* of the regular midterm that implements the `*` and `**` patterns. Use it to re-implement (b) with such patterns. Add pattern matching primitives to make you block extracting pattern most general and most succinct.

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

[1] For examples of Ruby syntax involving blocks, google around; there are plenty of good examples online, e.g., `http://blog.honeybadger.io/using-lambdas-in-ruby/`

[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 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. *Bytecode breakdown*

Ruby provides a facility to expose and disassemble the bytecode of compiled code (see example below).

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

```
code = "x = 5; y = x - 4; x**3 + 3*x*x*y + 3*x*y*y + y**3"
puts RubyVM::InstructionSequence.compile(code).disasm
```

a) Write a function `run-bytecode` that, given a string produced by the *disasm* method on some code, will compute the value produced by that code when evaluated. Assume that the original code was free of errors and evaluates without runtime errors, to an integer.

b) (Extra credit) Extend your `run-bytecode` to handle lambda functions and their calls. For example:

```
code = <<CODE
lambda {|x, y| x + y*y}.call 5, 2
CODE
puts RubyVM::InstructionSequence.compile(code).disasm
```

evaluates to 9, whereas

```
code = <<CODE
y = 3
lambda {|x| x + y*y}.call 5
CODE
puts RubyVM::InstructionSequence.compile(code).disasm
```

produces 14.

[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 3. *Encoding and decoding lambdas with de Bruijn index*

You are given LISP sexps that would evaluate to lambda function definitions. There are several restrictions placed on these lambda expressions: the only allowed operations are defining new lambda functions of exactly one argument, applying them to an argument, and calling an external function via `funcall`. There's nothing but lambda definitions, lambda applications, variables used as formal parameters, and applications of external functions via `funcall`—this isn't pure lambda calculus, but close. Note that external function symbols must occur free in the lambda expression: `((lambda (+) (funcall + 2 3)) '*)` is 6 (even if this looks a little weird), because `+` is not free in the expression.

a) Write a function `lambda-to-debruijn` that produces a de Bruijn index-encoded expression for a sexp as above (see `https://en.wikipedia.org/wiki/De_Bruijn_index`). The return value has no variables and no formal parameter lists (they are not needed to decode the expression), just numbers. For example, `(lambda (x) x)` becomes `(ldb 1)`, `(lambda (x) (lambda (y) (funcall x y)))` becomes `(ldb (ldb (funcall 2 1))`, and so on. In the resulting expression, only symbols that were free in the argument sexp remain, every other symbol (except `lambda` and `funcall` themselves) is replaced by a number.

For example for the pure lambda calculus expression *λx. λy. λz. x z (y z)*

```
(lambda (x) (lambda (y) (lambda (z) (funcall (funcall x z) (funcall y z)))))
```

the expression becomes

```
(ldb (ldb (ldb (funcall (funcall 3 1) (funcall 2 1)))))
```

(in lambda calculus/De Bruijn notation, it's *λλλ3 1 (2 1)*).

b) Write a function `debruijn-to-lambda` that takes a De Bruijn index-encoded expression as above, and returns an equivalent LISP lambda sexp, which can be evaluated by LISP's `eval`. Check your (a) and (b) against each other.

Note that the lambda calculus convention (and Haskell's) w.r.t. applications is that they associate to the left, i.e., the lambda expression *λx. λy. λz. x y z* means *λx. λy. λz. (x y) z* and, in LISP, means

```
(lambda (x) (lambda (y) (lambda (z) (funcall (funcall x y) z))))
```

This convention may seem strange, but it turns out to be convenient when you get used to it.

Problem 4. *More 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 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:

[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.

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

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