COMP 105 Assignment: Scheme

Due Thursday, February 17 at 11:59PM.

Preliminaries: Setup & Interpreters

The executable micro-Scheme interpreter is in `/comp/105/bin/uscheme`; if you are set up with `use comp105`, you should be able to run `uscheme` as a command. The interpreter accepts a `-q` (``quiet'') option, which turns off prompting. Your homework will be graded using `uscheme`. You can get source code by using `git`; since you probably already have a clone from the first assignment, you can make sure it's up to date by going to the source-code directory and running

```
git pull -v
```

When using the interpreter interactively, you may find it helpful to use `ledit`, as in the command

```
ledit uscheme
```

Dire Warnings

Since we're talking about functional programming, **none of the Scheme programs you submit should use any imperative features**. Banish `set`, `while`, `print`, and `begin` from your vocabulary! If you break this rule for any problem, you get a zero for that problem. (You may find it useful to use `begin` and `print` while debugging, but they must not appear in any code you submit.)

As a substitute for assignment, use `let` or `let*`. *Also use `let` or `letrec` for ``helper'' functions. Except as noted below, do not define helper functions at top level*. Avoid passing unnecessary parameters.

Your solutions should be valid micro-Scheme; in particular, they must pass the following test:

```
/comp/105/bin/uscheme -q < myfilename
```

without any error messages. If your file produces error messages, we won't test your solution and you will earn No Credit for functional correctness (you can still earn credit for readability).

Overview, organization, and what to submit

For this assignment, you will do exercises **1**, **5 (b-g,i-j)**, **6**, **12**, **13**, **20**, **26**, and **35** from pages 134-145 of Ramsey and Kamin, plus the three problems **A**, **B**, and **T** below. There are also extra-credit problems of significant interest (and difficulty).

Place your solutions to problems **A**, **1**, **5**, **6**, **12**, **13**, **B**, **20**, and other extra credit that you choose to submit in a file called `solution.scm`. (The solution to problem **T** goes in its own file, `solver-tests.scm`.) Be sure to put the solutions in order and to preceed each solution by a comment that looks like something like this:

```
;;
;; Problem 12
;;
```

Place your solution to question 26 in a file called `semantics.pdf`. You can create this file using LaTeX or Lyx another mathematical word processor, or you can write your solution by hand and scan it.

One problem you can do in pairs

35. *The uScheme interpreter (19%)*. Do exercise 35, part (b) on page 144-145 of Ramsey and Kamin: create a trace facility that will print the function, arguments, and result for any function application. This means you can control tracing from an interpreted program, e.g., you can `(set &trace 7)` to trace the next 7 applications. Use any negative number to trace

indefinitely. (We are stealing this trick from the Icon programming language.)

We recommend that you complete this problem first, as you may find the trace facility useful in debugging the code you write for the other problems. (Alternatively, you can use our version, which has support for `&trace` compiled in.)

Output should look something like this:

```
-> (set &trace -1)
-> (((curry =) 3) 4)
(curry <procedure>) => ...
(curry <procedure>) => <(lambda (x) (lambda (y) (f x y))), {f -> <procedure>}>
((curry =) 3) => ...
((curry =) 3) => <(lambda (y) (f x y)), {x -> 3, f -> <procedure>}>
(((curry =) 3) 4) => ...
  (f 3 4) => ...
  (f 3 4) => #f
(((curry =) 3) 4) => #f
#f
-> (set &trace 7)
-> (gcd 2222 100)
(gcd 2222 100) => ...
  (= 100 0) => ...
  (= 100 0) => #f
  (mod 2222 100) => ...
    (/ 2222 100) => ...
    (/ 2222 100) => 22
    (* 100 22) => ...
    (* 100 22) => 2200
    (- 2222 2200) => ...
    (- 2222 2200) => 22
  (mod 2222 100) => 22
  (gcd 100 22) => ...
  ... &trace goes to 0 ...
  (gcd 100 22) => 2
(gcd 2222 100) => 2
2
```

Don't forget to include the sample traces called for in the problem.

For extra credit (ALPHAVARS), write free variables of closures in alphabetical order. For more extra credit (ELLIPSIS), if code in a closure takes more than 40 characters, end it with an ellipsis and balanced parentheses.

Hints:

- Most of what you need is in Appendix B, except that the default printing functions don't expand closures. We suggest you define two new printing escapes `%w` and `%W`, to print values with a depth of 1 instead of the default depth of 0.
- If you are really clever, you will think of a way to use printing escapes to deal with indentation.
- The example prints only free variables that are not bound in the global environment. The code in Appendix B already supports this operation, but you'll need to set `globalenv` in `main`.

My solution to this problem required me to add or change under 60 lines of C code. You can try it out using the binary code on the linux or lab machines.

Details of all the problems you must do individually

A. *Good functional style (8%)*. The function

```
(define f-imperative (y) (x) ; x is a local variable
  (begin
    (set x e)
```

```
(while (p x y)
   (set x (g x y)))
(h x y)))
```

is in a typical imperative style, with assignment and looping. Write an equivalent function `f-functional` that doesn't use the imperative features `begin` (sequencing), `while` (goto), and `set` (assignment). You may use as many ``helper functions'' as you like, as long as they are defined using `let` or `letrec` and not at top level.

Hint #1: If you have trouble getting started, rewrite `while` to use `if` and `goto`. Now, what is like a `goto`?

Hint #2: `(set x e)` creates a binding of `e` to the name `x`. What other ways do you know of creating a binding of `e` to the name `x`?

Don't be confused about the purpose of this exercise. The exercise is a ``thought experiment.'' We don't want you to write and run code for some *particular* choice of `g`, `h`, `p`, `e`, `x`, and `y`. Instead, we want you write a function that works the same as `f-imperative` given *any* choice of `g`, `h`, `p`, `e`, `x`, and `y`. So for example, if `f-imperative` would loop forever on some inputs, your `f-functional` should also loop forever on exactly the same inputs.

Once you get your mind twisted in the right way, this exercise should be easy. The point of the exercise is not only to show that you can program without imperative features, but to help you develop a technique for eliminating such features. You'll use this technique again later on.

1. *Recursive functions on lists (6%).* Do exercise 1 on page 134 of Ramsey and Kamin. Use higher-order functions when you can, but expect to need recursion for some parts of the problem.

5, 6. *Higher-order functions (14%).* Do exercise 5 on pages 135-136 of Ramsey and Kamin, parts (b) to (g), part (i), and part (j). Do exercise 6 on page 136. You must *not* use recursion---solutions using recursion will receive zero credit. (This restriction applies only to code you write. For example, `gcd`, which is in the initial basis, or `insert`, which is given, may use recursion.) *For problem 5 only*, you may define helper functions at top level.

For problem 6, you get full credit if your implementations return correct results. You get **EXTRA CREDIT** if you can duplicate `exists?` and `all?` *exactly*. To earn the extra credit, it must be impossible to write a uScheme program that produces different output with your version than with a standard version.

13. *Functions as values (12%).* Do exercise 13 on pages 137–138 of Ramsey and Kamin.

B. *Higher-order, polymorphic sorting (13%).* Using `filter` and `curry`, define a function `qsort` that, when passed a binary comparison function (like <), returns a Quicksort function. So, for example,

```
-> ((qsort <) '(6 9 1 7 4 14 8 10 3 5 11 15 2 13 12))
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

-> ((qsort >) '(6 9 1 7 4 14 8 10 3 5 11 15 2 13 12))
(15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
```

You will also find it helpful to use the function-composition function `o`.

If you are not familiar with Quicksort, we have prepared a short Quicksort handout online.

Your Quicksort should not use the `append` function in any of its disguises. In other words, you shouldn't copy cons cells unnecessarily. (If you can't figure this part out, go ahead and use `append`; you will get partial credit.) *Hint #1: Use method of accumulating parameters covered in class when we discussed `revapp`. That is, think about writing a helper function that takes at least two arguments: a list `l` to be sorted and another list `tail` to be appended to the sorted list `l`.*

Hint #2: What part of Quicksort could `filter` and `o` help with?

Your code should use as few helper functions as possible. In particular, if you count up the number of occurrences of `define` and `lambda`, they should total at most three. (And if you give up and use `append`, that should save you a `lambda`.) If you need more lambda abstractions, you are doing something wrong. As usual, any helper functions should be defined internally using `let` or `letrec`, not at top level.

Remember to **give a brief explanation of why your recursive sort routine terminates**. *If you write more than a dozen lines of code for this problem, you're probably in trouble.*

> (For the bloody-minded among you, the C standard library specifies a higher-order Quicksort routine. How short an implementation can you write in C? How many more bugs did you find in your C version than in your Scheme version? How much longer did it take you? Do you find the answers surprising when you compare your experience with C to your experience with Scheme? *No credit is being offered for the answers to any of these C-related questions. I include them only so you can torture your friends who haven't had this course...* In case you wanted to know, P. J. Plauger has written a pretty good Quicksort in about 65 lines of ANSI standard C. He is quite careful about efficiency issues, like bounding use of the call stack.)

Here are some exacting test cases:

```
((qsort <) '(1 1 1))
((qsort <=) '(1 1 1))
((qsort <) '())
```

You might also try using qsort to sort a list of lists by putting the shortest lists first.

20. *Continuation-passing style (13%)*. Do exercise 20 on page 140 of Ramsey and Kamin. Don't overlook the possibility of deeply nested formulas with one kind of operator under another! My solution to this problem is under 50 lines of micro-Scheme.

T. *Testing your solver (3%)*. In file `solver-tests.scm`, submit three test cases that together exercise *all* the capabilities of your solver. These test cases should be in their own file, and they should contain two `val` bindings for each test case: `f1` should be the formula input the the solver, and `s1` should be either a satifying assignment, or if no satisfying assignment exists, then it should be the symbol `no-solution`. If, for example, I wanted to code the test case that appears on page 94 of the book, I might write

```
(val f1 '(and (or x y z) (or (not x) (not y) (not z)) (or x y (not z))))
(val s1 '((x #t) (y #f)))
```

As another test case, I might write

```
(val f2 '(and x (not x)))
(val s2 'no-solution)
```

Be sure to consider combinations of the various Boolean operators. Explain *why* these particular test cases are important—your test cases must not be too complicated to be explained.

We hope to run every submitted solver on every test case. Your goal should be to design test cases that cause other solvers to fail.

12. *Let-binding (4%)*. Do exercise 12 on page 137 of Ramsey and Kamin. You should be able to answer the questions in at most a few sentences.

26. *Operational semantics and language design (8%)*. Do all parts of exercise 26 on page 142 of Ramsey and Kamin. Be sure your answer to part (b) compiles and runs under `uscheme`.

Extra Credit

Extra credit (FIVES). *Programs as data*. To deepen your understanding of LISP and Scheme, here is a toy example of the kind of symbolic problem for which LISP is famous.

Consider the class of well-formed arithmetic computations using the numeral 5. These are expressions formed by taking the integer literal 5, the four arithmetic operators +, -, *, and /, and properly placed parentheses. Such expressions correspond to binary trees in which the internal nodes are operators and every leaf is a 5. Write a Scheme program to answer one or more of the following questions:

- What is the smallest positive integer than cannot be computed by an expression involving exactly five 5's?
- What is the largest prime number that can computed by an expression involving exactly five 5's?
- Exhibit an expression that evaluates to that prime number.

And, without implementing anything,

- Explain how you would change your implementation to use *exact* division instead of integer division.

Hints:

- You can build S-expressions that represent the arithmetic expressions in the problem, and you can just call `eval` to find out what they evaluate to.
- This problem involves an exhaustive search (for all numbers that can be computed with 5's), so good techniques are important. This is an excellent problem for dynamic programming (handout online).
- It will help you debug if you write a 'set of integer' implementation that keeps elements in order.
- You may want to speed up the search by writing a specialized version of eval.
- You may have to do something special to avoid division by zero. This is something of a pain in LISP, but you can cheat by specializing eval.
- Rational arithmetic is a good way to implement exact division.

Extra credit (FUNENV): In section, you will have talked about representing environments as functions, not as assocation lists. If you used this new representation, how would you change the metacircular evaluator in Ramsey and Kamin, Section 3.15? (You don't have to write the code, just explain how you would do it.) *Hint: you'll have to find a suitable value for the function to return in case the symbol isn't in the environment. Nil is probably not a good choice. In fact, nothing is a very good choice. This kind of dilemma motivates the use of exceptions in languages like CLU, ML, Modula-3, Ada, and C++.*

Extra credit (LAMBDA). `lambda` is more powerful than you might think. For extra credit, do any or all parts of Exercise 23 in Ramsey and Kamin, page 141. Test your work using the following scenario:

```
-> (define nth (n l)
      (if (= n 1) (car l)
                  (nth (- n 1) (cdr l))))
-> (val l (cons 'first (cons 'second (cons 'third nil))))
-> (nth 2 l)
second
-> (nth 3 l)
third
```

Hints:

- Perhaps you should use closures to represent cons cells and the empty list. Remember how we used `lambda` to store data when we did the the random-number generator in class.
- Given that `cons` should probably return a function (closure), try to make that function as simple as possible.

How to get code and what to submit

Get your copy of the code from the book by running

```
git clone linux.cs.tufts.edu:/comp/105/book-code
```

If you already have a copy, update your copy by running

```
git pull -v
```

The `bare` version contains just the uScheme code from the book, with simple comments marking the page number of each chunk of code. The `commented` version includes part of the book text as commentary. You can use whichever version you like, although we expect most of you will find it easier to work with the `bare` version.

What to submit for your joint work with your partner

For your joint work with your partner, run `submit105-uscheme-pair` from a directory that contains the Makefile and the modified C code from the `uscheme` interpreter. In addition to your code, please provide

- A short `README` file which describes, at a high level, the design and implementation of your solution for problem 35 (details of your solution are best left to comments in the source code)
- A file `traces.txt` containing the sample traces asked for in the problem

What to submit for your individual work

Provide another `README` file containing the following information:

- whom you have collaborated with
- how many hours you have spent on the assignment
- what problems are submitted, including extra credit

If you want, include any insights about problems other than problem 35, but detailed remarks about your solutions are probably best left to comments in the source code.

If you wish, you may also turn in a file named `transcript.txt` that contains test cases for your solutions. You don't have to give us test cases; the test cases shown above are there to help you, not to make more work for you.

When you are ready, run `submit105-uscheme-solo` to submit your work, which should include the following files:

- `README`: This documentation file is mandatory.
- `solution.scm`: This source file is mandatory.
- `solver-tests.scm`: This source file is mandatory.
- `semantics.pdf`: This source file is mandatory; you may prepare it by computer, or you can scan a handwritten solution.
- `transcript.txt`: This optional file can contain your test cases.