Continuations

COMP 105 Assignment

Due Monday, October 16, 2017 at 11:59PM

Contents

This assignment is all individual work. There is **no pair programming**.

Overview

The purpose of this assignment is to give you additional experience with higher-order, polymorphic functions and to give you practice using continuations for a backtracking search problem. The assignment builds on the previous two assignments, and it adds new ideas and techniques that are described in section 2.10 of Ramsey's book.

Setup

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

```
ledit uscheme
```

Also, if you didn't see the Piazza announcement about `&trace`, it is described in the revised Scheme homework[1].

Dire Warnings

The μScheme programs you submit must not use any imperative features. **Banish `set`, `while`, `print`, and `begin` from your vocabulary! If you break this rule for any exercise, you get No Credit for that exercise.** 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*`.

Except as noted below, do not define helper functions at top level. Instead, use `let` or `letrec` to define helper functions. When you do use `let` to define inner helper functions, avoid passing as parameters values that are already available in the environment.

Your solutions must be valid μScheme; in particular, they must pass the following test:

```
/comp/105/bin/uscheme -q < myfilename > /dev/null
```

without any error messages or unit-test failures. 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 structure and organization). If your file includes failing unit tests, you might possibly get some credit for functional correctness, but we cannot guarantee it.

We will evaluate functional correctness by testing your code extensively. **Because this testing is automatic, each function must be named be exactly as described in each question. Misnamed functions earn No Credit.**

Reading Comprehension (10 percent)

These questions are meant to guide you through the readings that will help you complete the assignment. Keep your answers brief and simple.

As usual, you can download the questions[2].

1. Look at `mk-insertion-sort` in Section 2.9.2, which starts on page 138.

   (a) Calling `(mk-insertion-sort >)` returns a function. What does the returned function do?

   (b) Given that the internal function `sort` (defined with `letrec` and `lambda`) takes only the list `xs` as argument, how does `sort` know what order to sort in?

   You are ready to start problem Q.

2. Read Section 2.12.3, which starts on page 155. What is the difference between DefineOldGlobal and DefineNewGlobal?

   You are ready to start problem 44.

[1] scheme.html#diagnostic-tracing
[2] ./cqs.continuations.txt

3. Set aside an hour to study the conjunctive-normal-form solver in Section 2.10.1, which starts on page 141. This will help you a lot in solving Exercise 21.

   (a) Look at code chunk 145b on page 145. In English, describe how `(one-solution f)` produces the answer `((x #t) (y #f))`. Walk through each function call, what the input to the function is, how the input is processed, and what the output of the function call is.

   (b) Look at code chunk 145d. As you did with 145b, describe how `(one-solution '((x) ((not x))))` produces the answer `no-solution`.

   You are ready to start Exercise 21.

4. Look at the first paragraph of Exercise 21 on page 213. Each bullet gives one possible rule for creating a formula. For each bullet, write one example formula (using the μScheme notation) that is created according to the rule for that bullet—four examples in total.

   - symbol:
   - not:
   - and:
   - or:

   You are ready to start problems F and T.

Programming and Language Design (90 percent)

Overview

For this assignment, you will explore an alternative semantics for `val` (**44**), you will build an efficient, polymorphic Quicksort (**Q**), and you will build a recognizer (**F**), and a solver (**21**) for Boolean formulas, with test cases (**T**).

Language-design problem

44. *Operational semantics and language design*. Do all parts of Exercise 44 on page 222 of Ramsey. Be sure your answer to part (b) compiles and runs under `uscheme`.

Related reading: Rules for evaluating definitions in section 2.12.3, especially the two rules for VAL.

Programming problems

Q. *Higher-order, polymorphic Quicksort*. 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)
```

As always, `qsort` must be accompanied by a contract and by unit tests written with `check-expect` or `check-error`. Each internal function written with `lambda` should be accompanied by a contract, but internal functions cannot be unit-tested.

If you are not familiar with Quicksort, we have prepared a short Quicksort handout[3] online.

Your Quicksort **should not use the `append` function** in any of its disguises. By not using `append`, you avoid copying cons cells unnecessarily. (If you can't figure this part out, go ahead and use `append`; you will get partial credit.)

Any helper functions must be defined internally using `let` or `letrec`, not at top level. Use as few helper functions as possible. Your code should contain at most three occurrences of `define` and `lambda`. (And if you give up and use `append`, you should have at most two.) If you are using more, you are doing something wrong.

The comments for your code should include a brief **explanation of why your recursive sort routine terminates.**

Hint #1: Use the 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 `xs` to be sorted and another list `tail` to be appended to the sorted list `xs`.

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

If you write more than a dozen lines of code for this exercise, you're probably in trouble.

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

My solution is 11 lines of µScheme.

Related reading: Section 2.9 on polymorphism in Scheme, especially section 2.9.2, which shows an example of a polymorphic, higher-order insertion sort. The definition of `revapp` and ideas about accumulating parameters in section 2.3.2, which starts on page 99. The examples of currying and function composition in section 2.7.2. Examples of `filter` in section 2.8.

F. *Recognizing formulas*. Exercise 21 on page 213 describes a little language of Boolean formulas represented as S-expressions. Define a function `formula?`, which when given an *arbitrary* S-expression, returns `#t` if the S-expression represents a Boolean formula and `#f` otherwise. Follow the definition in the book exactly.

Related reading: The definition of `equal?` in section 2.3. The definition of *LIST(A)* in section 2.6. The opening paragraph of exercise 21.

T. *Testing SAT solvers*. Create three test cases to test solutions to Exercise 21.
Your test cases will be represented by six `val` bindings, to variables `f1`, `s1`, `f2`, `s2`, `f3`, and `s3`.

- Value `f1` should be a Boolean formula as described in Exercise 21.
- Value `s1` should be an association list that represents a satisfying assignment for formula `f1`. If no satisfying assignment exists, value `s1` should be the symbol `no-solution`.
- Values `f2`, `s2`, `f3`, and `s3` are similar: two more formulas and their respective solutions.

[3] ../handouts/quicksort.pdf

For example, if I wanted to code the test case that appears on page 145 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 a second test case, I might write

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

Put your test cases into the template[4] at http://www.cs.tufts.edu/comp/105/homework/sat_solver_template.scm.

In comments in your test file, explain *why* these particular test cases are important—your test cases must not be too complicated to be explained. Consider different combinations of the various Boolean operators.

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

Related reading: The opening paragraph of exercise 21. The example formulas and satisfying assignments on page 145 (at the very end of section 2.10.1.

21. *SAT solving using continuation-passing style*. Do Exercise 21 on page 213 of Ramsey. You must define a function `find-formula-true-asst` which takes three parameters: a formula, a failure continuation, and a success continuation. The failure continuation should not accept any arguments, and the success continuation should accept two arguments: the first is the current (and perhaps partial) solution, and the second is a resume continuation. The solution to this exercise is under 50 lines of μScheme. Don't overlook the possibility of deeply nested formulas with one kind of operator under another.

The following unit tests will help make sure your function has the correct interface:

```
(check-expect (procedure? find-formula-true-asst) #t) ; correct name
(check-error (find-formula-true-asst))                ; not 0 arguments
(check-error (find-formula-true-asst 'x))             ; not 1 argument
(check-error (find-formula-true-asst 'x (lambda () 'fail)))   ; not 2 args
(check-error
   (find-formula-true-asst 'x (lambda () 'fail) (lambda (c r) 'succeed) z)) ; not 4 args
```

These additional checks also probe the interface, but they require at least a little bit of a solver—enough so that you call the success or failure continuation with the right number of arguments:

```
(check-error (find-formula-true-asst 'x (lambda () 'fail) (lambda () 'succeed)))
    ; success continuation expects 2 arguments, not 0
(check-error (find-formula-true-asst 'x (lambda () 'fail) (lambda (_) 'succeed)))
    ; success continuation expects 2 arguments, not 1
(check-error (find-formula-true-asst '(and x (not x)) (lambda (_) 'fail) (lambda (_) 'succeed)))
    ; failure continuation expects 0 arguments, not 1
```

And here are some more tests that probe if you can solve a few simple formulas, and if so, if you can call the proper continuation with the proper arguments.

```
(check-expect   ; x can be solved
```

[4] ./sat_solver_template.scm

```
   (find-formula-true-asst 'x
                           (lambda () 'fail)
                           (lambda (cur resume) 'succeed))
   'succeed)

(check-expect   ; x is solved by '((x #t))
   (find-formula-true-asst 'x
                           (lambda () 'fail)
                           (lambda (cur resume) (find 'x cur)))
   '#t)

(check-expect   ; (not x) can be solved
   (find-formula-true-asst '(not x)
                           (lambda () 'fail)
                           (lambda (cur resume) 'succeed))
   'succeed)

(check-expect   ; (not x) is solved by '((x #f))
   (find-formula-true-asst '(not x)
                           (lambda () 'fail)
                           (lambda (cur resume) (find 'x cur)))
   '#f)

(check-expect   ; (and x (not x)) cannot be solved
   (find-formula-true-asst '(and x (not x))
                           (lambda () 'fail)
                           (lambda (cur resume) 'succeed))
   'fail)
```

You can download all the tests[5]. You can run them at any time with

```
-> (use solver-interface-tests.scm)
```

This problem is (forgive me) the most satisfying problem on the assignment.

Related reading: Section 2.10 on continuation passing, especially the CNF solver in section 2.10.1.

What and how to submit

You must submit five files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - The numbers of the problems that you solved
  - The number of hours you worked on the assignment
- A `cqs.continuations.txt` containing the reading-comprehension questions[6] with your answers

[5] ./solver-interface-tests.scm
[6] ./cqs.continuations.txt

edited in

- A PDF file `semantics.pdf` containing the solution to Exercise **44**. If you already know LaTeX[7], by all means use it. Otherwise, write your solution by hand and scan it. Do check with someone else who can confirm that your work is legible—if we cannot read your work, we cannot grade it.

  N.B. Part of your solution to Exercise 44 includes μScheme code, which we ask you to compile to make sure that it works. We nevertheless want you to include the code in your PDF along with your semantics and your explanation—*not* in one of the other files.

- A file `solution.scm` containing the solutions to Exercises **Q**, **F**, and **21**. You must precede each solution by a comment that looks like something like this:

  ```
  ;;
  ;; Problem Q
  ;;
  ```

- A file `solver-tests.scm` containing the definitions of formulas `f1`, `f2`, and `f3` and the definitions of solutions `s1`, `s2`, and `s3`, which constitutes your answer to Exercise **T**.

As soon as you have the files listed above, run `submit105-continuations` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

Avoid common mistakes

The most common mistakes on this assignment have to do with the Boolean-formula solver in Exercise 21. They are

- It's easy to handle fewer cases than are actually present in the exercise. You can avoid this mistake by considering all ways the operators `and`, `or`, and `not` can be combined pairwise to make formulas.
- It's easy to write near-duplicate code that handles essentially similar cases multiple times. This mistake is harder to avoid; I recommend that you look at your cases carefully, and if you see two pieces of code that look similar, try abstracting the similar parts into a function.
- It's easy to write code with the wrong interface—but if you use the unit tests above, they should help.

Another common mistake is to **forget to explain why `qsort` terminates**.

[7] http://www.latex-project.org/