Continuations

COMP 105 Assignment

Due Tuesday, February 26, 2019 at 11:59PM

Contents

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

Overview

Continuations are a key technology in event-driven user interfaces, such as are found in games and in many Web frameworks. For example, continuations are used in JavaScript as "callbacks." Continuations are also used to implement sophisticated control flow, including backtracking. This assignment introduces you to continuations through backtracking search. It also gives you additional experience with higher-order, polymorphic functions. The assignment builds on the previous two assignments, and it adds new ideas and techniques that are described in section 2.10 of *Build, Prove, and Compare*.

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
```

In this assignment, you may find the `&trace` feature especially useful. It is described in the Scheme homework[1].

Dire Warnings

The μScheme programs you submit must not use any imperative features. **Banish `set`, `while`, `println`, `print`, `printu`, 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 `println` 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. Revisit the definition of values in figure 2.1 on page 95. Next, study the definition of the predefined function `all?` in section 2.8.3, which starts on page 133. Like many higher-order functions, `all?` has a subtle contract: it is permissible to call `(all? p? xs)` if and only if there exists a set A such that following are true:

    - Parameter `p?` is a function that accepts one argument, which must be a member of A. Then `p?` returns a Boolean.

    - Parameter `xs` is in *LIST(A)*. That is, `xs` is a list of values, and every element of `xs` is a member of A.

    In other words, `xs` must be a list of values, each of which may be passed to `p?`.

    With this contract in mind, answer these questions:

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

(a) Write a value that is never permissible as the second, `xs` argument to `all?`, no matter what `p?` is:

(b) Write a value *made with `cons`* that is never permissible as the second, `xs` argument to `all?`, no matter what `p?` is:

(c) Write a value that may or may not be permissible as the second, `xs` argument to `all?`, depending on what `p?` is. Your value should be *permissible* if `p?` is `number?`, but *impermissible* if `p?` is `prime?`:

~~(You are welcome to use the interpreter to verify that each impermissible call results in a checked run-time error.)~~ (An impermissible call might or might not result in a checked run-time error.)

2. Study the description of function `list-of?` in problem **L** below, and also the hints. Now, for each value of `xs` that you listed in the previous question, answer whether it is permissible to pass that value to `list-of?` along with the predicate `prime?`, and if so, what result `list-of?` should return.

   (a) Permissible? If so, what does `list-of?` return?

   (b) Permissible? If so, what does `list-of?` return?

   (c) Permissible? If so, what does `list-of?` return?

   *You are ready for problem **L**.*

3. Section 2.12.3, which starts on page 157, describes the semantics of the true-definition forms. Use the semantics to answer two questions about the following sequence of definitions:

```
(val f (lambda () y))
(val y 10)
(f)
```

   Given evaluating `lambda` in an environment ρ creates a closure using that same ρ, if the definitions above are evaluated in an environment in which $y \in \text{dom } \rho$, then what is the result of the call to `f`? Pick A, B, or C.

   (A) It returns `10`.
   (B) An error is raised: `Run-time error: name y not found`.
   (C) It returns whatever value `y` had before the definitions were evaluated.

   If the definitions above are evaluated in an environment in which $y \notin \text{dom } \rho$, what is the result of the call to `f`? Pick either A or B.

   (A) It returns `10`.
   (B) An error is raised: `Run-time error: name y not found`.

   You are ready to start problem 45.

4. Read the description of Boolean formulas in the section “Representing Boolean formulas” below. Then revisit the description of µScheme’s records in the recitation on higher-order functions. (Or if you prefer, read section 2.16.6, which starts on page 194, which shows how records are implemented.) Now assume you are given a formula f_0, and answer these questions:

   (a) How, in constant time, do you tell if f_0 has the form `(make-not` f`)`?

   (b) How, in constant time, do you tell if f_0 has the form `(make-and` fs`)`?

(c) How, in constant time, do you tell if f_0 has the form `(make-or` *fs*`)`?

You are ready to start problems L and F.

5. Read the definition of evaluation in problem E below, including the definition of the environment used by `eval-formula`.

   Each of the following Boolean formulas is evaluated in an environment where `x` is `#t`, `y` is `#f`, and `z` is `#f`. What is the result of evaluating each formula? (For each formula, answer `#t`, "true", `#f`, or "false.")

   (a) x, which in μScheme is constructed by `'x`

   (b) $\neg x$, which in μScheme is constructed by `(make-not 'x)`

   (c) $\neg y \wedge x$, which in μScheme is constructed by `(make-and (list2 (make-not 'y) 'x))`

   (d) $\neg x \vee y \vee z$, which in μScheme is constructed by
   `(make-or (list3 (make-not 'x) 'y 'z))`

   (e) Formula `(make-not (make-or (list1 'z)))`, which has a tricky `make-or` applied to a list of length 1, and so can't be written using infix notation

   You are ready to start problem E.

6. Read about the Boolean-satisfaction problem for CNF formulas, in section 2.10.1, which starts on page 143. The rules for satisfaction are the same for all Boolean formulas, even those that are not in CNF.

   For each of the following Boolean formulas, if there is an assignment to `x`, `y`, and `z` that satisfies the formula, write the words "is solved by" and a satisfying assignment. Incomplete assignments are OK. If there is no satisfying assignment, write the words "has no solution."

   Examples:

   $x \vee y \vee z$, which in μScheme is constructed by `(make-or '(x y z))`, is solved by `'((x #t))`

   $x \wedge y \wedge z$, which in μScheme is constructed by `(make-and '(x y z))`, is solved by `'((x #t) (y #t) (z #t))`

   $x \wedge \neg x$, which in μScheme is constructed by `(make-and (list2 'x (make-not 'x)))`, has no solution

   For each of these formulas, replace the ellipsis with your answer:

   (a) $(x \vee \neg x) \wedge y$, which in μScheme is constructed by
   `(make-and (list2 (make-or (list2 'x (make-not 'x))) 'y))`,
   …

   (b) $(x \vee \neg x) \wedge \neg x$, which in μScheme is constructed by
   `(make-and (list2 (make-or (list2 'x (make-not 'x))) (make-not 'x)))`,
   …

   (c) $(x \vee y \vee z) \wedge (\neg x \wedge x) \wedge (x \vee \neg y \vee \neg z)$, which in μScheme is constructed by

```
(make-and
    (list3 (make-or (list3 'x 'y 'z))
           (make-and (list2 (make-not 'x) 'x))
```

```
                (make-or (list3 'x (make-not 'y) (make-not 'z))))))
```

…

You are ready to start problems S and T.

Programming and Language Design (90 percent)

You will explore an alternative semantics for `val` (**45**), and you will solve four problems related to Boolean formulas: recognize lists (**L**), recognize formulas (**F**), evaluate formulas (**E**), and solve formulas (**S**). You will also submit test cases for the solver (**T**). Problems **L**, **F**, **E**, and **S** require algebraic laws.

Language-design problem

45. *Operational semantics and language design.* Do all parts of exercise 45 on page 226 of *Build, Prove, and Compare*. 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.

Representing Boolean formulas

This homework involves Boolean formulas. We represent a formula either as a symbol or a record ("struct"), using the following record definitions:

```
(record not [arg])
(record or  [args])
(record and [args])
```

In the context of these definitions, a formula is one of the following:

- A symbol, which stands for a variable
- A record `(make-not` *f*`)`, where *f* is a formula
- A record `(make-or` *fs*`)`, where *fs* is a list of formulas
- A record `(make-and` *fs*`)`, where *fs* is a list of formulas

Programming problems

L. *Recognizing lists.* Define a higher-order function `list-of?`, which takes two arguments:

- The first argument, `A?`, is a predicate that can be applied to any value.
- The second argument, `v`, is an *arbitrary* µScheme value.

Calling `(list-of? A? v)` returns a Boolean that is `#t` if `v` is a list of values, each of which satisfies `A?`. Otherwise, `(list-of? A? v)` returns `#f`.

Hints:

- The forms of data for arbitrary µScheme values can be deduced from the definition of values in Figure 2.1 on page page 95.

  Function `list-of?` must correctly handle fully general S-expressions like `(cons 'COMP 105)`.

- Provided `A?` is a good predicate, `(list-of? A? v)` never causes a checked run-time error, no matter what `v` is. This property distinguishes `list-of?` from `all?`.
- Every one of the primitive type predicates, like `symbol?` and `procedure?`, is a good predicate to pass to `list-of?`.
- For testing, I encourage you to define and use the following additional predicate:

  ```
  (define value? (_) #t) ;; tell if the argument is a value
  ```

  You can then identify any list of values with `(list-of? value? v)`.
- Test with several different predicates. For each one, write `check-assert` tests with both true and false results.
- Remember that for every A, the empty list is a list of A.

Related reading: In *Build, Prove, and Compare*, section 2.2, which starts on page 94. And if you need to, the second *Lesson in Program Design*[3] (Scheme values).

Laws: Function `list-of?` needs algebraic laws. The last law may include a side condition "all other forms."

F. *Recognizing formulas.* Define a function `formula?`, which when given an *arbitrary* μScheme value, returns `#t` if the value represents a Boolean formula and `#f` otherwise. Follow the above definition of formulas exactly.

A straightforward solution that uses `if` expressions and "how were you formed?" questions takes 10 lines of μScheme. If you want to get clever with short-circuit operators, you can cut that in half.

Hints:

- Function `formula?` must analyze every possible μScheme value, no matter what its form, and must always return `#t` or `#f`. Every possible form of μScheme value must therefore be handled by some case. (If it turns out that one case can handle multiple forms of input, that is OK. It is even good practice.)
- Function `formula?` must identify, out of all possible values, which ones are formulas. Every possible form of formula must therefore be handled by a case that returns `#t`.
- Just because some clown puts a list of values in a record, it doesn't mean that the record holds a list of formulas.
- Because `formula?` must analyze every possible μScheme value, it is not possible to violate its contract!

Related reading: Section 2.2, which starts on page 94. The definition of `equal?` in section 2.3. The definition of *LIST(A)* in section 2.6.

Laws: Function `formula?` needs algebraic laws. In the last law, I encourage you to include a side condition like "`v` has none of the forms above." By obviating the need to define a law for every possible form of S-expression, such a law will enable you to focus your laws on just the interesting forms.

E. *Evaluating formulas.* Define a function `eval-formula`, which takes two arguments: a formula f and an environment `env`. The environment is an association list in which each key is a symbol and each value

[3] ../design/lessons.pdf

is a Boolean. If the formula is satisfied in the given environment, `(eval-formula` f `env)` returns `#t`; otherwise it returns `#f`. Evaluation is defined by induction:

- The result of evaluating a symbol is the result of looking that symbol up in the environment.
- The result of evaluating a record `(make-not` f`)` is the complement of the result of evaluating f.
- The result of evaluating a record `(make-or` *fs*`)` is true if and only if the list of formulas *fs* contains a formula that evaluates to true.
- The result of evaluating a record `(make-and` *fs*`)` is true if and only if every formula in the list of formulas *fs* evaluates to true.

It is part of the contract of `eval-formula` that `(eval-formula f env)` may be called only when every variable in formula `f` is bound in `env`.

My solution is 10 lines, and it is structurally similar to my straightforward implementation of `formula?`.

You must document your solution with algebraic laws. As usual, we recommend you write the laws before you write the code.

Related reading: The initial basis of μScheme. And if you are uncertain about structure, the implementation of either the Impcore evaluator or the μScheme evaluator. (You could also look at the implementation of a μScheme evaluator *in μScheme*, which you would find in section 2.15.4, which starts on page 183, but that section has so much detail that it may be easier just to figure out on your own how to structure an evaluator written in μScheme.)

Laws: Function `eval` needs algebraic laws. The only permissible side condition should be something like "where `x` is a symbol."

Your laws should cover only those inputs permitted by `eval`'s contract. Do not include any laws for inputs that violate the contract. (And in the code, do not include any cases for inputs that violate the contract.)

T. *Testing SAT solvers*. Create three test cases to test solutions to problem **S**. 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 above. For example,

  ```
  (val f1 (make-not 'x))
  ```

  would be acceptable.
- 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`. For example,

  ```
  (val s1 '((x #f)))
  ```

  solves formula `f1` above.
- Values `f2`, `s2`, `f3`, and `s3` are similar: two more formulas and their respective solutions.

As another example, if I wanted to code the test formula

$$(x \vee y \vee z) \wedge (\neg z \vee \neg y \vee \neg z) \wedge (x \vee y \vee \neg z),$$

I might write

```
(val f1 (make-and
          (list3 (make-or (list3 'x 'y 'z))
```

```
                 (make-or (list3 (make-not 'x) (make-not 'y) (make-not 'z)))
                 (make-or (list3 'x 'y (make-not 'z))))))
(val s1 '((x #t) (y #f)))
```

As a second test case, I might write

```
(val f2 (make-and (list2 'x (make-not 'x))))  ; x and not x
(val s2 'no-solution)
```

Put your test cases into a file `solver-tests.scm`, which you should create initially using the template[4] at https://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.

Design test cases that will find bugs in solvers. (We will run every submitted solver on every submitted test case.)

Related reading: The example formulas and satisfying assignments on page 147 (at the very end of section 2.10.1).

Laws: This problem includes only test cases and results, not any code. No laws are needed.

S. *SAT solving using continuation-passing style*. Write a function `find-formula-true-asst` which, given a satisfiable formula, finds a *satisfying* assignment—that is, a mapping of variables to Booleans that makes the formula true. Remember De Morgan's laws, one of which is mentioned on page 134.

Function `find-formula-true-asst` must take three parameters: a formula, a failure continuation, and a success continuation. A call to

```
(find-formula-true-asst f fail succ)
```

searches for an assignment that satisfies formula `f`. If it finds a satisfying assignment, it calls `succ`, passing both the satisfying assignment (as an association list) and a resume continuation. If it fails to find a satisfying assignment, it calls `fail`. Notes:

- The failure continuation does not accept any arguments.
- The success continuation accepts two arguments: the first is the current (and perhaps partial) solution, and the second is a resume continuation. A resume continuation, like a failure continuation, does not accept any arguments.
- Formulas may be nested with one kind of operator under another.

My solution to this exercise is under 50 lines of μScheme.

You'll be able to use the ideas in section 2.10.1, but not the code. Instead, try using `letrec` to define the following mutually recursive functions:

- Calling `(find-formula-asst formula bool cur fail succeed)` extends assignment `cur` to find an assignment that makes the single `formula` equal to `bool`.

- Calling `(find-all-asst formulas bool cur fail succeed)` extends `cur` to find an assignment that makes every formula in the list `formulas` equal to `bool`.

- Calling `(find-any-asst formulas bool cur fail succeed)` extends `cur` to find an assignment that makes any one of the `formulas` equal to `bool`.

[4] ./sat_solver_template.scm

- Calling `(find-formula-symbol x bool cur fail succeed)`, where `x` is a symbol, does one of three things:
  - If `x` is bound to `bool` in `cur`, succeeds with environment `cur` and resume continuation `fail`
  - If `x` is bound to `(not bool)` in `cur`, fails
  - If `x` is not bound in `cur`, extends `cur` with a binding of `x` to `bool`, then succeeds with the extended environment and resume continuation `fail`

In all the functions above, `bool` is `#t` or `#f`. Before defining each of the functions, we recommend completing the following algebraic laws:

```
(find-formula-asst x            bool cur fail succeed) == ...,
                                                   where x is a symbol
(find-formula-asst (make-not f)  bool cur fail succeed) == ...
(find-formula-asst (make-or  fs) #t   cur fail succeed) == ...
(find-formula-asst (make-or  fs) #f   cur fail succeed) == ...
(find-formula-asst (make-and fs) #t   cur fail succeed) == ...
(find-formula-asst (make-and fs) #f   cur fail succeed) == ...

(find-all-asst '()         bool cur fail succeed) == ...
(find-all-asst (cons f fs) bool cur fail succeed) == ...

(find-any-asst '()         bool cur fail succeed) == ...
(find-any-asst (cons f fs) bool cur fail succeed) == ...

(find-formula-symbol x bool cur fail succeed) == ..., where x is not bound in cur
(find-formula-symbol x bool cur fail succeed) == ..., where x is bool in cur
(find-formula-symbol x bool cur fail succeed) == ..., where x is (not bool) in cur
```

Include the completed laws in your solution.

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

```
(check-assert (procedure? find-formula-true-asst))    ; 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
                   (make-and (list2 'x (make-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
   (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   ; (make-not 'x) can be solved
   (find-formula-true-asst (make-not 'x)
                           (lambda () 'fail)
                           (lambda (cur resume) 'succeed))
   'succeed)

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

(check-expect   ; (make-and (list2 'x (make-not 'x))) cannot be solved
   (find-formula-true-asst (make-and (list2 'x (make-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.

Laws: Complete the laws given in the problem, and include the completed laws with your solution. Place laws for each helper function near the definition of that function.

What and how to submit

You must submit five files:

[5] ./solver-interface-tests.scm

- A `README` file containing
  - The names of the people with whom you collaborated
  - The numbers of the problems that you solved
- A `cqs.continuations.txt` containing the reading-comprehension questions[6] with your answers edited in
- A PDF file `semantics.pdf` containing the solution to Exercise **45**. 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 45 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 **L**, **F**, **E**, and **S**. Each function should be accompanied by a brief contract, algebraic laws as specified, and unit tests. Precede each solution by a comment that looks like something like this:

  ```
  ;;
  ;; Problem L
  ;;
  ```
- 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

It's a common mistake to define a version of `list-of?` or `formula?` that causes a checked run-time error on some inputs. These functions must always return Booleans.[8]

The most common mistakes on this assignment have to do with the Boolean-formula solver in problem **S**. 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.

[6] ./cqs.continuations.txt
[7] http://www.latex-project.org/
[8] Function `list-of?` requires a good predicate, but I've never seen a student make a mistake with the predicate.