Computer Science 21b (Spring Term, 2013)
Structure and Interpretation of Computer Programs

Problem Set 3: The Stability of Marriage

Due Friday, March 15

Reading Assignment: Chapter 3, Sections 3.1–3.3.

1 Homework Exercises

Exercise 3.12 : `append!`

Exercise 3.14 : `Mystery` procedure.

Exercise 3.17 : `Count-pairs`, the right way.

2 Laboratory Assignment

Daisy, Daisy, give me your answer, do.
I'm half crazy, all for the love of you.
It won't be a stylish marriage! I can't afford a carriage,
But you'll look sweet, upon the seat of a bicycle built for two.[1]

Marriage as a societal institution has been with us for many years, though in times of social flux or upheaval the stability of the institution is often subject to question, if not shaken outright. God made light, heavens and earth; He filled the seas with great whales; He created Man and Woman and commanded them to "be fruitful and multiply."[2] First there was Adam and Eve... sometime later Abraham and Sarah, Isaac and Rebekah, and a long time afterwards Henry VIII and his succession of marriages, Elizabeth Taylor and Richard Burton (twice! or was it three times...), and so on to the current day. Without question, love is as powerful an emotion as it was in the good old days, but how love is manifested in marriage is as varied as the number of marriages that have taken place.

It is generally not recognized that, despite the emotive aspects of love and romance, marriage is fundamentally an engineering problem. As such, it is amenable to a careful mathematical and computational analysis, emplying principles of programming and finite combinatorics. In this problem set, we will use the paradigm of message-passing programming, combined with the idea of encoding state information in procedures, to provide, for the first time in human history, *a definitive solution to the problem of stability in marriage.*

Let's assume, for the sake of an argument, that you are a village matchmaker given the task of marrying 100 men and 100 women. Each of the men has ranked the women from 1 to 100 in the order of his preference; each woman, not to be outdone, has similarly ranked the 100 men. As a matchmaker, it is clear that in producing 100 couples, you will not be able to give each person his first choice: for example, if Alan is the first choice of all the women, only one will get him, and all other women will be left with their second (or worse) choices.

[1] Words and music by Harry Dacre, 1892. "When Dacre, an English popular composer, first came to the United States, he brought with him a bicycle, for which he was charged duty. His friend (the songwriter William Jerome) remarked lightly: 'It's lucky you didn't bring a bicycle built for two, otherwise you'd have to pay double duty.' Dacre was so taken with the phrase 'bicycle built for two' that he decide to use it in a song. That song, *Daisy Bell,* first became successful in a London music hall, in a performance by Kate Lawrence. Tony Pastor was the first one to sing it in the United States. Its success in America began when Jennie Lindsay brought down the house with it at the Atlantic Gardens on the Bowery early in 1892." (David Ewen, **American Popular Songs**. Random House, 1973.)

[2] We wish to emphasize the mathematical and computational aspects of this divine imperative.

Rather than insist that everyone get their first choice, you are instead charged with creating 100 *stable* marriages. A set of marriages is said to be stable if there exists no man m and woman w such that m likes w better than his wife, and w likes m better than her husband. The notion is called "stability" as m's and w's marriages are unstable, since (albeit sordid and tawdry) m and w could optimize their sorry lot in life by leaving their mates and running off together.

You are presented with several problems. Does a set of stable marriages always exist, given any group of preference lists? How can you go about finding one? What is an *algorithm*, a method for doing so? How can it be encoded in Scheme? Happily, the answer to the first question is "Yes." As to the rest of the questions: read on!

3 The Algorithm

We first describe in English an algorithm to find stable marriages. Each person has a *preference list* of members of the opposite sex. The first time a person proposes, it is to their number one choice, the second time to their number two choice, and so on. We will assume, without loss of non-sexist generality, that men always propose to women (although there is no reason that it couldn't happen the other way around: see Problem 7), and that the proposals occur in *rounds*. At any moment in time, each person is either *engaged* or *unengaged*, and of course initially everyone is unengaged. (Assuming heterosexual alliances only, we can further— in the interest of mathematical simplicity—assert that the number of engaged men equals precisely the number of engaged women.)

At the start of a round, each *unengaged* man is ordered by you, the matchmaker, to propose to the woman he loves most but as yet to whom he has not proposed. Each woman responds according to her status as given in the following case analysis:

- **Unengaged, received no proposals:** Sit tight, life is bound to improve. Your prince will be arriving shortly.
- **Unengaged, received one proposal:** Accept the proposal (the man and woman are now engaged).
- **Unengaged, received more than one proposal:** Accept the proposal you like the best based on *your* preference list, telling the other suitors to buzz off (Wonder Man and woman now engaged).
- **Engaged, received no proposals:** Declare your eternal and undying love to your sweetie.
- **Engaged, received one proposal:** If you like the man who proposed more than your current main squeeze, dump your fiancé and reengage yourself to this more inviting gentleman. You and the proposer are now engaged; Mr. Dumped is now unengaged. Otherwise, say "I'm flattered but I'm not interested," and declare your undying love for your intended.
- **Engaged, received more than one proposal:** Choose the proposal you like best by looking at your preference list, and act as if you are currently engaged and this gentleman alone has proposed.

At the end of a round and the ensuing musical chairs, some men are engaged and others are not. If everyone is engaged, stop: the current pairs are the marriages that the matchmaker seeks. Otherwise, do another round.

Of course, no assurances have been made that the marriages produced are stable, or even that everyone is engaged at the end. For example, is it possible that a man proposes 100 times, exhausting his proposal list, and gets rejected every time? We'll address these questions later on; for the moment, let's assume that the method works correctly.

4 A Scheme Implementation of People

Do you really believe that people are just computers, brain meat is nothing but a central processing unit, and the thoughts, hopes, and dreams that keep our life alive are nothing but software being executed? Of course not. (However, I'm sad to say, there are computer scientists and cognitive psychologists who do.) Nonetheless, for the purposes of this problem set, we will *model* certain salient features of people by a procedure:

```
  ==> (define mickey-mouse (make-person 'Mickey))
  mickey-mouse
  ==> (define minnie-mouse (make-person 'Minnie))
  minnie-mouse
  ==> ((minnie-mouse 'i-love-you) mickey-mouse)
  i-love-you-too
  ==> ((minnie-mouse 'intended) 'name)
  Mickey
```

In this example, `make-person` is a procedure evaluating to a procedure `me` that *represents a person.* As an example:

```
(define (make-person my-name)
  (let ((preference-list '())
        (possible-mates '())
        (current-intended '()))
    (define (me message)
      (cond ((eq? message 'name) my-name)
            ((eq? message 'intended) current-intended)
            ((eq? message 'loves) preference-list)
            ((eq? message 'possible) possible-mates)
            ((eq? message 'reset) ... )
            ((eq? message 'load-preferences) ... )
            ((eq? message 'propose) ... )
            ((eq? message 'i-love-you) ... )
            ((eq? message 'i-changed-my-mind) ... )
            (else
              (error "Bad message to a person " (list my-name message)))))
      me))
```

Each person, then, will be able to respond to certain fixed messages. For example, in order to tell Mickey to propose, we evaluate `(mickey-mouse 'propose)`, and for Minnie to evaluate Mickey's proposal of marriage we evaluate `((minnie-mouse 'i-love-you) mickey-mouse)`. In analyzing the functionality of the previous expression, we see that `(minnie-mouse 'i-love-you)` evaluates to a *procedure* that is Minnie's "proposal consideration" procedure: it is then *applied* to `mickey-mouse`, and state information is then modified accordingly. This style of programming is called *message passing*, because evaluation is controlled by sending messages to procedures, which then act as semi-autonomous computational processes.

Here is a list of possible messages and what they are supposed to do:

- `name` evaluates to the atom given as the name of the person/procedure, i.e.,

```
    ==> (define mrs-flintstone (make-person 'wilma))
    mrs-flintstone
    ==> (mrs-flintstone 'name)
    wilma
```

- `intended` evaluates to the *procedure* denoting the intended significant other. As a conceivable example:

```
    ==> (mrs-flintstone 'intended)
    #(procedure mr-flintstone)
    ==> ((mrs-flintstone 'intended) 'name)
    fred
```

- `loves` evaluates to the list of people the person would marry, in order of preference.
- `possible` evaluates to the list of people the person has not yet proposed to, but is still interested in, given in order of preference.
- `load-preferences` assigns to `preference-list` a list of mates (in order of preference) to a person:

  ```
  ==> ((minnie 'load-preferences) (list mickey donald goofy pluto))
  ```

  This message also does the equivalent of a `reset` *(see below)*.
- `reset` reinitializes a person's parameters, setting the current `intended` to `'()` and `possible-mates` to `preference-list`: this allows the person to start proposing again beginning with his or her first choice, or start considering proposals with no obligations, unconstrained by previous romantic entanglements.
- `propose` causes a procedure to propose to his or her "next" choice, i.e., the most desired individual to whom the person has not yet proposed.

  ```
  ==> (Harry 'propose)
  we-are-engaged
  ```

  Harry proposed to his next choice, and it was accepted. If his proposal was rejected, the expression would evaluate to `no-one-loves-me`.
- `i-love-you` is a message sent to a beloved: it evaluates to a *procedure* P that is then applied to the sender of the proposal. The procedure P can access and change the same state information as the procedure representing the beloved. For example:

  ```
  ==> ((Anne 'i-love-you) Harry)
  i-love-you-too
  ```

  Note that P can change state information for Anne and Harry, *and* evaluates to the atom `i-love-you-too`, indicating that the proposal was accepted. On the other hand,

  ```
  ==> ((Anne 'i-love-you) Tarzan)
  buzz-off-creep
  ```

  indicates that Tarzan's proposal was rejected, and that state information was not changed. Remember that if Anne was previously engaged to someone when Harry proposed but liked Harry better, she must break her current engagement *(see below). Editor's note: any resemblance of these characters to real people is purely coincidental.*
- `i-changed-my-mind` is the message sent to a Mr. Right who in the course of the stable marriage algorithm has become Mr. Wrong, in order that he become Mr. Dumped. For example,

  ```
  ==> ((Harry 'i-changed-my-mind) Lyn)
  dumped!
  ```

  indicates that procedure `Lyn`, once involved with procedure `Harry`, is moving on to greener pastures. If indeed Harry and Lyn are involved, i.e.,

  ```
          (and (eq? (Harry 'intended) Lyn)
               (eq? (Lyn 'intended) Harry))
  ```

  then their `intended` state information is reset to `'()`, and the expression evaluates to `dumped!`; otherwise expression evaluates to `there-must-be-some-misunderstanding`, and nothing is changed.

Remember that if anyone ever says about the recent break-up of their romance, "It was a mutual decision," you are talking to the one who got dumped.

—Mary James

The code you need to complete this assignment can be found from the home page for the course. Included is a "test file" of men and women with preference lists, which you should feel free to try out, and change. Maybe you could use this code, once you fix it up, to start up a computer dating service among your friends: observe that no pair of people, unmatched to each other, can conspire to "circumvent" the pairings you compute.

Note: As you implement and debug code, be aware that since your code modifies *state*—and modifies the *environment*—you may need to *reload* new code as you debug. This way you wipe out whatever weird state you may have created (a person/procedure with a messed-up internal memory), and begin with a new *tabula rasa.*

Problem 0. In the description of the stable marriage algorithm, we consider *separately* the cases of a person receiving one, or more than one, proposal of marriage. No such "separate case" treatment is apparent in the Scheme implementation. Explain why.

Problem 1. Write the following procedures: [3]

`(courtship` ⟨*unengaged_proposers*⟩ ⟨*proposers*⟩ ⟨*proposees*⟩`)` an implementation of the Stable Marriage algorithm described above, where a subset of the proposers (namely, the unengaged proposers) propose marriage to the proposees. Note that while we for the moment assume that ⟨*proposers*⟩ is a list of men and ⟨*proposees*⟩ is a list of women, we are *not* assuming that men-procedures are structurally different from women-procedures, i.e., they are both produced by `make-person`. Do we really need this list of proposees? Maybe not...

`(couple?` ⟨*person*$_1$⟩ ⟨*person*$_2$⟩`)` a predicate telling whether two people are engaged to each other.

`(currently-unengaged` ⟨*list*⟩`)` a procedure returning a list of the people appearing as elements of ⟨*list*⟩ who are not engaged.

`(send` ⟨*list*⟩ ⟨*message*⟩`)` sends a given message to each person in the list, for example `(send list-of-men 'propose)`.

`(i-like-more?` ⟨*person*$_1$⟩ ⟨*person*$_2$⟩`)` a predicate telling whether the first argument is liked more than the second argument by a particular person/procedure. *Note: this procedure has to be defined in the correct lexical context to be able to access and manipulate the proper data structures.*

Problem 2. Complete the part of `make-person` that handles the message `i-love-you`.

Problem 3. Add dialogue to the algorithm, so that (say) proposals, acceptances, and engagement breakings are printed at the terminal as the stable marriage algorithm continues.

Problem 4. Prove that when the stable marriage algorithm ends, everyone is engaged to somebody. *Hint: prove by contradiction.*

Problem 5. Give arguments that if P is a proposer, then every time P proposes, it is to someone liked less, and if A is a proposal acceptor, then every time A accepts a proposal, it is from someone liked more.

Problem 6. Prove that the marriages produced by the stable marriage algorithm are indeed stable. Suppose not: then let m and w be a man and woman not matched together. Show m and w could not both wish to run off together. *Hint: show m had to have proposed to w at some time: where did their love go wrong?*

[3] In adding procedures, be careful not to confuse `=`, `eq?`, `equal?`, and related functions: use `eq?` to test whether two atoms or procedures are the same, but `equal?` to test if two list structures are the same: `(eq? (cons 1 2) (cons 1 2))` evaluates to *false:* the rationale is the same that makes you say "identical twins are not equivalent since they are physically different"—i.e., the `cons` cells are different. On the other hand, `equal?` applied to the same arguments evaluates to *true*, since this predicate means "isomorphic"—that they are similar structures even though they are physically different.

Problem 7. Using the "test file" of people, evaluate `(match-make men women)` and `(match-make women men)`. Then try the following:

```
==> (define bob (make-person 'Bob))
bob
==> (define carol (make-person 'Carol))
carol
==> (define ted (make-person 'Ted))
ted
==> (define alice (make-person 'Alice))
alice
==> ((bob 'load-preferences) (list carol alice))
preferences-loaded
==> ((ted 'load-preferences) (list alice carol))
preferences-loaded
==> ((carol 'load-preferences) (list ted bob))
preferences-loaded
==> ((alice 'load-preferences) (list bob ted))
preferences-loaded
==> (define men (list bob ted))
men
==> (define women (list carol alice))
women
==> (match-make men women)
...
==> (match-make women men)
...
```

Explain what has happened. Would you rather propose marriage or have it proposed to you?

```
;; This is the code for -- Stable Marriage

(define (match-make proposers proposees)
  (send proposers 'reset)
  (send proposees 'reset)
  (courtship proposers proposers proposees)
  (zip-together (send proposers 'name)
                (send (send proposers 'intended) 'name)))

(define (courtship unengaged-proposers proposers proposees) ... )

(define (currently-unengaged list-of-people) ... )

(define (send list-of-people message) ... )

(define (couple? person1 person2) ...)

(define (zip-together list1 list2)
  (if (null? list1)
      '()
      (cons (list (car list1) (car list2))
            (zip-together (cdr list1) (cdr list2)))))

(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst)) (cons (car lst) (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

```
(define (make-person my-name)
  (let ((preference-list '())
        (possible-mates '())
        (current-intended '()))
    (define (me message)
      (cond ((eq? message 'name) my-name)
            ((eq? message 'intended) current-intended)
            ((eq? message 'loves) preference-list)
            ((eq? message 'possible) possible-mates)
            ((eq? message 'reset)
               (set! current-intended '())
               (set! possible-mates preference-list)
               'reset-done)
            ((eq? message 'load-preferences)
               (lambda (plist)
                  (set! preference-list plist)
                  (set! possible-mates plist)
                  (set! current-intended '())
                  'preferences-loaded))
            ((eq? message 'propose)
               (let ((beloved (car possible-mates)))
                 (set! possible-mates (cdr possible-mates))
                 (if (eq? ((beloved 'i-love-you) me)
                          'i-love-you-too)
                     (begin (set! current-intended beloved)
                            'we-are-engaged)
                     'no-one-loves-me)))
            ((eq? message 'i-love-you) ... )
            ((eq? message 'i-changed-my-mind)
               (lambda (lost-love)
                  (cond ((eq? current-intended lost-love)
                            (set! current-intended '())
                            'dumped!)
                        (else
                            'there-must-be-some-misunderstanding))))
            (else
              (error "Bad message to a person " (list me my-name message)))))
      me))
```

```
;; This is a test file for -- Stable Marriage

(define alan (make-person 'Alan))
(define bob (make-person 'Bob))
(define charles (make-person 'Chuck))
(define david (make-person 'Dave))
(define ernest (make-person 'Ernie))
(define franklin (make-person 'Frank))
(define agnes (make-person 'Agnes))
(define bertha (make-person 'Bertha))
(define carol (make-person 'Carol))
(define deborah (make-person 'Debbie))
(define ellen (make-person 'Ellen))
(define francine (make-person 'Fran))

((alan 'load-preferences)
   (list agnes carol francine bertha deborah ellen))
((bob 'load-preferences)
   (list carol francine bertha deborah agnes ellen))
((charles 'load-preferences)
   (list agnes francine carol deborah bertha ellen))
((david 'load-preferences)
   (list francine ellen deborah agnes carol bertha))
((ernest 'load-preferences)
   (list ellen carol francine agnes deborah bertha))
((franklin 'load-preferences)
   (list ellen carol francine bertha agnes deborah))
((agnes 'load-preferences)
   (list charles alan bob david ernest franklin))
((bertha 'load-preferences)
   (list charles alan bob david ernest franklin))
((carol 'load-preferences)
   (list franklin charles bob alan ernest david))
((deborah 'load-preferences)
   (list bob alan charles franklin david ernest))
((ellen 'load-preferences)
   (list franklin charles bob alan ernest david))
((francine 'load-preferences)
   (list alan bob charles david franklin ernest))

(define men (list alan bob charles david ernest franklin))
(define women (list agnes bertha carol deborah ellen francine))
```

"She's lovely," John thinks, almost staring.
They shake hands. John's heart gives a lurch.
"Handsome, all right, and what he's wearing
Suggests he's just returned from church...
Sound, solid, practical, and active,"
Thinks Liz, "I find him quite attractive.
Perhaps..." All this has been inferred
Before the first substantive word
Has passed between the two. John orders
A croissant and espresso; she
A sponge cake and a cup of tea.
They sit, but do not breach the borders
Of discourse till, at the same time,
They each break silence with, "Well, I'm—"

Both stop, confused. Both start together:
"I'm sorry—" Each again stops dead.
They laugh. "It hardly matters whether
You speak or I," says John: "I said,
Or meant to say— I'm glad we're meeting."
Liz quietly smiles, without completing
What she began. "Not fair," says John.
"Come clean. What was it now? Come on:
One confidence deserves another."
"No need," says Liz. "You've said what I
Would have admitted in reply."
They look, half smiling, at each other
Half puzzled too, as if to say,
"I don't know why I feel this way."

— Vikram Seth, *The Golden Gate*