MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945 Spring 2013
Problem Set 7

Issued: Wed. 20 March 2013 **Due: Wed. 3 April 2013**

Readings:

Radul & Sussman, "The Art of the Propagator,"
http://dspace.mit.edu/handle/1721.1/44215.
This is a preliminary paper about the ideas in the propagator system. It is NOT about the system we will be using. However, this paper accurately captures the philosophy of the system and explains a simple implementation.

Alexey Radul's PhD thesis dissertation:
Propagation Networks: A Flexible and Expressive Substrate for Computation
http://web.mit.edu/~axch/www/phd-thesis.pdf
This is more detail and more worked out than the "Art" paper above, but it is longer.

Radul & Sussman, "Revised Report on the Propagator Model"
http://groups.csail.mit.edu/mac/users/gjs/propagators/
This is the documentation for the system we will be using.

Code:

http://groups.csail.mit.edu/mac/users/gjs/propagators/propagator.tar
This is a tarball of the prototype propagator system that we will be using. It is written in MIT/GNU Scheme. It is lots of code, so don't try to read through it.

load.scm, ui.scm, extra.scm, family.scm

Propagation

This problem set introduces propagation. Propagation is a computational model built on the idea that the basic computational elements are autonomous machines interconnected by shared cells through which they communicate. Each machine continuously examines the cells it is interested in, and adds information to some based on deductions it can make from information from the others. This model makes it easy to smoothly combine expression-oriented and constraint-based programming; it also easily accommodates implicit incremental distributed search in ordinary programs.

Getting Started

You can obtain the prototype propagator system from the URL given. You should download it and unpack it in some appropriate directory. Get an MIT/GNU Scheme and change your working directory to the propagator directory, and load the propagator system, then go back to your problem set 7 directory and load the problem-set software:

```
(cd <your propagator directory>)
(load "load")
(cd <your ps07 directory>)
(load "load")
```

Your propagator system is initialized by incanting:

```
(initialize-scheduler)
```

This clears out all cells and wiring previously done. It does not clear out Scheme or propagator definitions.

Cells Store Partial Information

In this example, the content of cells will be numeric intervals. For example, suppose the cell's content is the interval [3, 10], which you can think of as saying that the value described is between 3 and 10. If you add the interval [0, 5] as more content, then the cell content will be be [3, 5]. And if you now add the interval [6, 10] that produces a contradiction.

Although propagation is a more general process, in this problem set, we will be keeping track of the assumptions that justify the content we put into a cell. The propagator system keeps track of the reasons and can say which sets of reasons are contradictory.

Here's an example. John is a student doing term-time work. There are various opinions about his earnings. We need a cell to store information about John's earnings:

```
(define-cell john-earnings)
```

Harry estimates that John's earnings are between $20K and $27K. This estimate is represented as an interval and it is dependent on the premise named harry-estimate. We may find that harry-estimate is not true, but let's start out by believing it:

```
(tell! john-earnings (make-interval 20 27) 'harry-estimate)
```

So at this point all we know about John's earnings is:

```
(content john-earnings)
#(tms (#(value=#[interval 20 27],
   premises=(harry-estimate),
   informants=(user))))
```

TMS Maintains Provenance

This is a data structure called a TMS (Truth Maintenance System). It is a set of statements, with the premises that support those statements. A cell can hold a TMS that keeps track of what is "true" based on which assumptions are currently believed. Initally, every assumption is believed to be true.

Suppose Mary also estimates of John's earnings. She believes it to be between $15K and $21K.

```
(tell! john-earnings (make-interval 15 21) 'Mary-estimate)
```

Now, if we look at what is known about John's earnings we see three statements. There is Harry's estimate, Mary's estimate, and the consequence of believing both:

```
(pp (content john-earnings))
#(tms
  (#(value=#[interval 20 21],
   premises=(harry-estimate mary-estimate),
   informants=(user))
   #(value=#[interval 15 21],
   premises=(mary-estimate),
   informants=(user))
   #(value=#[interval 20 27],
   premises=(harry-estimate),
   informants=(user))))
```

At this time the best estimate (most informative, taking into account all the sources of information) is

```
(inquire john-earnings)
#(value=#[interval 20 21],
   premises=(harry-estimate mary-estimate),
   informants=(user))
```

The bank (which wants to give John a student credit card) esimates his earnings to be between 25K and 30K. However, this estimate contradicts the previous best estimate, so something must be wrong:

```
(tell! john-earnings (make-interval 25 30) 'bank-estimate)
;Value: (contradiction (Mary-estimate harry-estimate bank-estimate))
```

Here's the information about John's earnings now. If we believe Mary and Harry and the bank, we get a contradiction.

```
(content john-earnings)
#(tms (#(value=#[contradictory-interval 25 21],
   premises=(mary-estimate harry-estimate bank-estimate),
   informants=(user)) #(value=#[interval 25 30],
   premises=(bank-estimate),
   informants=(user)) #(value=#[interval 20 21],
   premises=(harry-estimate mary-estimate),
   informants=(user)) #(value=#[interval 15 21],
   premises=(mary-estimate),
   informants=(user)) #(value=#[interval 20 27],
   premises=(harry-estimate),
   informants=(user))))
```

Indeed, the best information we have is contradictory.

```
(inquire john-earnings)
#(value=#[contradictory-interval 25 21],
   premises=(mary-estimate harry-estimate bank-estimate),
   informants=(user))
```

Controlling the Worldview

We can choose to disregard Harry’s opinion:

```
(retract! ’harry-estimate)
```

But there is still a contradiction between the bank and Mary.

```
(inquire john-earnings)
#(value=#[contradictory-interval 25 21],
   premises=(mary-estimate bank-estimate),
   informants=(user))
```

The contradiction did not depend on Harry, so let’s reinstate him.

```
(assert! ’harry-estimate)
```

Nothing has changed, except that the system now knows that the contradiction did not depend on Harry.

```
(inquire john-earnings)
#(value=#[contradictory-interval 25 21],
   premises=(mary-estimate bank-estimate),
   informants=(user))
```

Let’s withdraw our our belief in Mary’s estimate:

```
(retract! ’Mary-estimate)
```

Now John's earnings are between $25K and $27K, based on Harry's estimate together with the bank's estimate:

```
(inquire john-earnings)
#(value=#[interval 25 27],
   premises=(harry-estimate bank-estimate),
   informants=(user))
```

Propagators Relate Information in Cells

Let us attach symbolic descriptions to the earnings levels. For example, imagine that MIT Financial services considers students to be eligible for loans if their earnings are less than 20K. (Not really!) First, we will define a compound propagator that takes a value cell, an interval, and a boolean output cell. It tells the boolean output cell to be true only if the contents of the value cell is within the

range specified by the range.

```
(define-propagator (p:in-range? value interval bool)
  (p:and (e:<= (e:interval-low interval) value)
         (e:<= value (e:interval-high interval))
         bool))
```

This is made up of several propagators. It depends on the syntax of p: and e: propagator constructors, as described in class and in the propagator documentation.

We'll introduce a property of an earnings-estimate cell called loan-eligible that will be true when the earnings are below 20K.

The following procedure adds another cell as a symbolic property of a cell that contains an estimate. The status of the property will be true or false (or unknown or contradictory) depending on whether the cell's content is within the designated interval.

```
;;; estimate is the cell to give a symbolic property.  Interval is the
;;; range for which the property will be true.  property-name is the
;;; symbol used to access the symbolic property cell (with eq-get)
;;; from the estimate cell.

(define (add-interval-property estimate interval property-name)
  ;; Is there already such a property on the estimate?
  (let ((status-cell (eq-get estimate property-name))) ;Already defined?
    (if status-cell
        ;; Property already exists, get the range cell.
        (let ((range (eq-get estimate (symbol property-name ':range))))
          (if (not range)
              (error "Interval property has no range"
                     (name estimate) property-name))
          (p:== interval range)
          'range-updated)
        ;; New definition: Create internal cells to hold the status of
        ;; the symbolic property and its defining range (initialized
        ;; to the given interval).
        (let-cells (status-cell range)
          ;; Initialize the range cell.
          (p:== interval range)
          ;; Make the status and the range properties of the estimate.
          (eq-put! estimate (symbol property-name ':range) range)
          (eq-put! estimate property-name status-cell)
          ;; If the cell content is within the interval
          ;; then propagate #t to the status-cell.
          (p:in-range? estimate range status-cell)
          ;; If the status is true then propagate the content of the
          ;; interval-call to the estimate.
          (p:switch status-cell range estimate)
          'property-added))))
```

We use this to add a loan-eligible property to the john-earnings cell.

```
(add-interval-property john-earnings (make-interval 0 20) 'loan-eligible)
```

Suppose that the MIT claims that John is loan-eligible:

```
(tell! (eq-get john-earnings 'loan-eligible) #t 'mit-financial)
;Value: (contradiction (bank-estimate harry-estimate mit-financial))
```

Whoops! That contradicts the other estimates. Let's get rid of them:

```
(for-each retract! '(harry-estimate bank-estimate))
```

OK, now the MIT estimate has propagated to john-earnings.

```
(inquire john-earnings)
#(value=#[interval 0 20],
   premises=(mit-financial),
   informants=((switch:p status-cell range)))
```

Suppose MIT is worried and retracts its belief

```
(retract! 'mit-financial)
```

Then, nothing is known about John's financial status.

```
(inquire john-earnings)
;Value: #(*the-nothing*)

(inquire (eq-get john-earnings 'loan-eligible))
;Value: #(*the-nothing*)
```

But perhaps Mary was right all along:

```
(assert! 'Mary-estimate)
```

Then we have her estimate, but it is not low enough to make MIT happy.

```
(inquire john-earnings)
#(value=#[interval 15 21],
   premises=(mary-estimate),
   informants=(user))

(inquire (eq-get john-earnings 'loan-eligible))
;Value: #(*the-nothing*)
```

But if Debbie also provides a low-ball estimate:

```
(tell! john-earnings (make-interval 5 18) 'debby-estimate)
```

Then John's earnings fall within the MIT guidelines and he is now eligible for a loan!

```
(inquire john-earnings)
#(value=#[interval 15 18],
   premises=(mary-estimate debby-estimate),
   informants=(user))

(inquire (eq-get john-earnings 'loan-eligible))
#(value=#t,
   premises=(debby-estimate mary-estimate),
   informants=((and:p cell4 cell2)))
```

Example: Modeling Financial Status

So now we have seen propagation with two cells, JOHN-EARNINGS and (EQ-GET JOHN-EARNINGS 'LOAN-ELIGIBLE), interconnected with propagators. We see how provenance of factoids is carried by named assumptions, such as Mary-ESTIMATE, and how beliefs depend on the assumptions. So it is now time to build something more interesting.

We can generalize the interval-property classifier to attach multiple symbolic ranges to a cell containing an interval. We will allow the symbolic ranges to overlap.

```
(define ((c:bins named-ranges) numeric-interval)
  (for-each
   (lambda (named-range)
     (add-interval-property numeric-interval
                            (cadr named-range)
                            (car named-range)))
   named-ranges))
```

Using this, on the authority of gjs, we can make multiple symbolic ranges associated with the numerical range cell foo:

```
((c:bins (named-ranges 'gjs
            '(low    ,(make-interval 3 6))
            '(medium ,(make-interval 5 8))
            '(high   ,(make-interval 7 9))))
 foo)
```

The intervals will depend on the GJS premise. If GJS is retracted, the intervals will become unknown, but the named cells will remain, and new intervals can be placed in them.

Now, imagine that we have a population of people, each of which has a variety of properties, some of which are numerical, some of which are symbolic, and some of which are symbolic descriptions of ranges of numerical properties. For example, Joe may be tall and Harry may be of medium height; Joe may have a high income and Harry a low income; but Harry may be frugal and Joe may be spending beyond his means.

Problem 7.1: Warmup

a. Given a list of names of imaginary people, make a program that attaches to each name the numerical properties "height", "weight", "income", "expenses", (and any others you may please to invent). For each such numerical property assign further symbolic properties. For symbolically named ranges your code should use the c:bins propagator-network constructor we supplied above. It uses the eq-put! and eq-get! mechanism for attaching things together. We provide a path mechanism to make these chains easy to work with.

Problem 7.1: Warmup continued

For example:

```
(eq-put! 'Joe 'height 'Joes-height-cell)

(eq-put! 'Joes-height-cell 'tall 'Joes-tall-cell)

((eq-path '(tall height)) 'Joe)
;Value: Joes-tall-cell
```

Of course, 'Joes-tall-cell and 'Joes-height-cell should actually be appropriate cells, not the atomic symbols used here for illustration.

b. Add to your program some interesting cells and constraints. For an example, add a cell to the data about each person that carries an estimate of whether he is living beyond his/her means: So, if the "expenses" interval is strictly higher than the "income" interval then the "living-beyond-means" property should become true, decorated with the assumptions on which this judgement depended. Some of these may be higher-order combinations, for example, if the symbolic properties "rich" and "living-beyond-means" are both true this should make the "profligate" property become true. Your dossier on each person should include several such interrelated properties.

Make sure that the assumptions are correctly tracked by your program. Demonstrate your code on some fun data.

A Financial Model

Real people are not isolated individuals. They are bound to other people by relationships. For example, there are families: Alyssa P. Hacker and Ben Bitdiddle are married; as a consequence there is a family income and family expenses. Ben and Harry Reasoner belong to a club together. Part of Ben's expenses are dues for the club; so the club has income and expenses. Income and expenses are a kind of flow. Unless an activity generates value (such as farming) or destroys it (such as eating) there is a conservation law. In every isolated group of interacting individuals the difference of the sum of the incomes and the sum of the expenses must be equal to the rate of creation (or destruction, if negative) of value. If a group is not isolated we may violate the conservation to model income or expenses originating outside the group (e.g. interest, dividends, taxes). This gives us the beginnings of a way to model an economy.

In the file family.scm we construct a small model of a family consisting of Ben and Alyssa. Part of that model is a simple database structure made up using eq-properties. There are only a few phenomena modeled here: income and expenses.

Problem 7.2: Building more constraints

The procedures BREAKDOWN and COMBINE-FINANCIAL-ENTITIES should be able to combine more than two parts. This requires building up the sums for many parts. Please fix these programs so that they correctly build a tree of adder constraints for any number of parts.

A population may have many such groupings and conservation laws. It is fun to build a model and see the consequences that arise from alternate world views.

Problem 7.3: A small family complication

Alyssa and Ben are married. Elaborate this model. For example, Harry Reasoner and Eva Lu Ator are also married, and Harry and Ben are members of a club. Add financial entities representing those relationships and elaborate the story line.

How accurate are the inferences based on partial information? Do you see any cases where the intervals are much wider than they ought to be? When might this happen?

Make a situation where, because of some wrong information there are contradictions, and the results of an INQUIRE query differ depending on what worldview (set of assumptions) is adopted. Use that to track down the bad data.

Problem 7.4: Your turn

Come up with an interesting application of the propagator framework. Build it and demonstrate your work.

Some possible suggesgtions:

a. Build a town, with about twenty or thirty individuals connected by a variety of overlapping financial relationships. The inhabitants can be built using random numbers to build their financial states (but make them "plausible").

It is interesting to add in some qualitative constraints, as we demonstrated in the introduction and warmup. There are rich and poor, there are profligate and frugal, there are honest people and scam artists.

Which of your people have no visible means of support? Who is living beyond their means? How can we audit them?

b. Build an interactive puzzle game, such as Sudoku, Minesweeper, or Mastermind, where the game rule enforcement and player feedback are implemented with propagators.

Better yet, build an AI module that can do a half-decent job at playing the game, making deductions about appropriate moves using constraints and partial information, but without search backtracking.

The games suggested above all have the interesting property that information about the final state of the game accumulates monotonically on the board. Other games may have state that is highly dynamic, changing from turn to turn (e.g. checkers) as game pieces move or disappear. Because of this "stateful" nature, such games require more abstraction to implement with propagators and will probably be more challenging.

Also observe that Sudoku and Minesweeper have the property that safe moves are commutative, unlike turn-based games such as Mastermind. This impacts what kinds of constraints you need to build and how you structure an AI.

Tip: Although the most obvious way to represent game cell states may be by defining a custom data type, this involves quite a bit of trouble that we won’t be covering in class. Instead, you can typically get away with representing states using intervals (interpreted over the integers) and boolean values (by breaking up a single cell that would’ve contained a complex enumerated type into several associated cells containing only boolean indicators). You will likely need to re-use the strategy for variadic constraints you developed back in problem set 2.

```
;;;; This is the ps07 file extra.scm

;;; These make e: and p: propagators out of the scheme procedures
(propagatify interval-low)
(propagatify interval-high)

(define-propagator (p:in-range? value interval bool)
  (p:and (e:<= (e:interval-low interval) value)
         (e:<= value (e:interval-high interval))
         bool))

(define (add-interval-property estimate interval property-name)
  ;; Is there already such a property on the estimate?
  (let ((status-cell (eq-get estimate property-name))) ;Already defined?
    (if status-cell
        ;; Property already exists, get the range cell.
        (let ((range (eq-get estimate (symbol property-name ':range))))
          (if (not range)
              (error "Interval property has no range"
                     (name estimate) property-name))
          (p:== interval range)
          'range-updated)
        ;; New definition:
        ;; Create internal cells to hold the status of the symbolic
        ;; property and its defining range (initialized to the given interval).
        (let-cells (status-cell range)
          ;; Initialize the range cell.
          (p:== interval range)
          ;; Make the status cell and the range named properties of
          ;; the estimate cell.
          (eq-put! estimate (symbol property-name ':range) range)
          (eq-put! estimate property-name status-cell)
          ;; If the cell content is within the interval
          ;; then propagate #t to the status-cell.
          (p:in-range? estimate range status-cell)
          ;; If the status is true then propagate the content of the
          ;; interval-call to the estimate.
          (p:switch status-cell range estimate)
          'property-added))))
```

```
    (define ((c:bins named-ranges) numeric-interval)
      (for-each
       (lambda (named-range)
         (add-interval-property numeric-interval
                                (cadr named-range)
                                (car named-range)))
       named-ranges))


    ;;; This can be used to support named ranges with a premise
    ;;; representing the range-defining authority:

    (define (named-ranges authority . named-ranges)
      (map (lambda (named-range)
             (list (car named-range)
                   (depends-on (cadr named-range) authority)))
           named-ranges))
```

```
;;;; This is the ps07 file ui.scm

;;; This removes those annoying hash numbers after ;Value:
(set! repl:write-result-hash-numbers? #f)

;;; This is part of paranoid programming.
(define (assert p #!optional error-comment irritant)
  (if (not p)
      (begin
        (if (not (default-object? irritant))
            (pp irritant))
        (error
         (if (default-object? error-comment)
             "Failed assertion"
             error-comment)))))

;;; This abstracts an annoying composition
(define (depends-on information . premises)
  (make-tms (contingent information premises)))

;;; This is required because (run) returns old value if there is
;;; nothing to do.  This is a problem if a contradiction is resolved
;;; by a kick-out! with no propagation.

(define (tell! cell information . informants)
  (assert (cell? cell) "Can only tell something to a cell.")
  (set! *last-value-of-run* 'done)
  (add-content cell (make-tms (contingent information informants)))
  (run))

(define (retract! premise)
  (set! *last-value-of-run* 'done)
  (kick-out! premise)
  (run))

(define (assert! premise)
  (set! *last-value-of-run* 'done)
  (bring-in! premise)
  (run))
```

```
(define (inquire cell)
  (assert (cell? cell) "Can only inquire of a cell.")
  (let ((c (content cell)))
    (if (tms? c)
        (let ((v (tms-query c)))
          (cond ((nothing? v) v)
                ((contingent? v) v)
                (else
                 (error
                  "Bug: TMS contains non-contingent statement"
                  cell))))
        c)))
```

```
;;;; A Small Financial Example -- this file is family.scm

;;; First, we need a small database mechanism
;;;  Parent and child here do not refer to biological
;;;  things, but rather the relationships of parts
;;;  of a database.

(define (add-branch! parent child name)
  (eq-put! parent name child)
  (eq-put! child 'parent parent)
  (eq-put! child 'given-name name)
  'done)

(define (name-of thing)
  (let ((n (eq-get thing 'given-name)))
    (if n
        (let ((p (eq-get thing 'parent)))
          (if p
              (cons n (name-of p))
              (list n)))
        (list thing))))

;;; e.g. (thing-of Gaggle-salary gross-income Ben)

(define (thing-of name-path)
  (let lp ((path name-path))
    (cond ((= (length path) 1) (car path))
          (else
           (eq-get (lp (cdr path))
                   (car path))))))



;;; A financial entity has three cells

(define (make-financial-entity entity)
  (eq-put! entity 'kind-of-entity 'financial)

  (let-cells (gross-income expenses net-income)

    (add-branch! entity gross-income 'gross-income)
    (add-branch! entity net-income 'net-income)
    (add-branch! entity expenses 'expenses)

    (c:+ expenses net-income gross-income)
    'done
    ))
```

```
(define (financial-entity? thing)
  (eq? (eq-get thing 'kind-of-entity) 'financial))

(define (gross-income entity)
  (assert (financial-entity? entity))
  (eq-get entity 'gross-income))

(define (net-income entity)
  (assert (financial-entity? entity))
  (eq-get entity 'net-income))

(define (expenses entity)
  (assert (financial-entity? entity))
  (eq-get entity 'expenses))

(define (breakdown sum-node . part-names)
  (for-each (lambda (part-name)
              (let-cell part
                        (add-branch! sum-node part part-name)))
            part-names)
  (cond ((= (length part-names) 2)
         (c:+ (eq-get sum-node (car part-names))
              (eq-get sum-node (cadr part-names))
              sum-node)
         'done)
        (else
         (error "I don't know how to sum multiple parts"))))

(define (combine-financial-entities compound . parts)
  (assert (every financial-entity? parts))
  (cond ((= (length parts) 2)
         (let ((p1 (car parts)) (p2 (cadr parts)))
           (c:+ (gross-income p1) (gross-income p2) (gross-income compound))
           (c:+ (net-income p1) (net-income p2) (net-income compound))
           (c:+ (expenses p1) (expenses p2) (expenses compound))
           'done))
        (else
         (error "I don't know how to combine multiple parts"))))
```

```
#|
(initialize-scheduler)

(make-financial-entity 'Alyssa)
(make-financial-entity 'Ben)

;;; Ben and Alyssa are married
(make-financial-entity 'Ben-Alyssa)
(combine-financial-entities 'Ben-Alyssa 'Ben 'Alyssa)

;;; Ben and Alyssa file income tax jointly
(tell! (gross-income 'Ben-Alyssa) 427000 'IRS)

;;; Ben works at Gaggle as a software engineer.
(breakdown (gross-income 'Ben) 'Gaggle-salary 'investments)

;;; He gets paid alot to make good apps.
(tell! (thing-of '(Gaggle-salary gross-income Ben)) 200000 'Gaggle)

;;; Alyssa works as a PhD biochemist in big pharma.
(breakdown (gross-income 'Alyssa) 'GeneScam-salary 'investments)

;;; Biochemists are paid poorly.
(tell! (thing-of '(GeneScam-salary gross-income Alyssa)) 70000 'GeneScam)

(tell! (thing-of '(investments gross-income Alyssa))
       (make-interval 30000 40000) 'Alyssa)

(inquire (thing-of '(investments gross-income Ben)))
;Value: #(supported #[interval 117000 127000] (gaggle genescam alyssa irs))

;;; Ben is a tightwad
(tell! (thing-of '(expenses Ben)) (make-interval 10000 20000) 'Ben)

(inquire (thing-of '(net-income Ben)))
;Value: #(supported #[interval 297000 317000] (ben genescam alyssa irs))

;;; But Alyssa is not cheap.  She likes luxury.
(tell! (thing-of '(expenses Alyssa)) (make-interval 200000 215000) 'Alyssa)

(inquire (thing-of '(net-income Alyssa)))
;Value: #(supported #[interval -115000 -90000] (alyssa genescam))

;;; But they are doing OK anyway!
(inquire (thing-of '(net-income Ben-Alyssa)))
;Value: #(supported #[interval 192000 217000] (ben alyssa irs))

;;; Notice that this conclusion does not depend on the details, such
;;; as Gaggle or GeneScam!
|#
```