MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945 Spring 2013 Problem Set 6

Issued: Wed. 13 March 2013   **Due: Wed. 20 March 2013**

```
Readings:
  SICP second edition
     Section 4.3: AGAIN!
        Variations on a Scheme--Nondeterministic Programming

  Online MIT/GNU Scheme Documentation,
     Section  2.3: Dynamic Binding - fluid-let
     Section 12.4: Continuations   - call-with-current-continuation
                                     & within-continuation

  There is an entire bibliography of stuff about this on:
     http://library.readscheme.org/page6.html

Code:  load.scm, funco.scm, ambsch.scm (attached), stack-queue.scm,
       examples.scm (attached)
```

Generate and Test

We normally think of generate and test, and its extreme use in search, as an AI technique. However, it can be viewed as a way of making systems that are modular and independently evolvable, as in the exploratory behavior of biological systems. Consider a very simple example: suppose we have to solve a quadratic equation. There are two roots to a quadratic. We could return both, and assume that the user of the solution knows how to deal with that, or we could return one and hope for the best. (The canonical sqrt routine returns the positive square root, even though there are two square roots!) The disadvantage of returning both solutions is that the receiver of that result must know to try the computation with both and either reject one, for good reason, or return both results of the computation, which may itself have made some choices. The disadvantage of returning only one solution is that it may not be the right one for the receiver's purpose.

Perhaps a better way to handle this is to build a backtracking mechanism into the infrastructure. The square-root procedure should return one of the roots, with the option to change its mind and return the other one if the first choice is determined to be inappropriate by the receiver. It is, and should be, the receiver's responsibility to determine if the ingredients to its computation are appropriate and acceptable. This may itself require a complex computation, involving choices whose consequences may not be apparent without further computation, so the process is recursive. Of course, this gets us into potentially deadly exponential searches through all possible assignments to all the choices that have been made in the program. As usual, modular flexibility can be dangerous.

Linguistically Implicit Search

It is important to consider the extent to which a search strategy can be separated from the other parts of a program, so that one can interchange search strategies without greatly modifying the program. In this problem set we take the further step of pushing search and search control into the infrastructure that is supported by the language, without explicitly building search into our program at all.

This idea has considerable history. In 1961 John McCarthy had the idea of a nondeterministic operator AMB, which could be useful for representing nondeterministic automata. In 1967 Bob Floyd had the idea of building backtracking search into a computer language as part of the linguistic glue. In 1969 Carl Hewitt proposed a language, PLANNER, that embodied these ideas. In the early 1970s Colmerauer, Kowalski, Roussel, and Warren developed Prolog, a language based on a limited form of first-order predicate calculus, which made backtracking search implicit.

In this problem set we will learn how to implement and how to use linguistic nondeterminism. Before proceeding we recommend that you carefully reread section 4.3, up to but not including 4.3.3 of SICP (pages 412--426). This material introduces AMB and shows how it can be used to formalize some classes of search problems. Section 4.3.3 describes how to compile a language that includes AMB into a combinator structure. We touched on this in Problem Set 4. In this problem set we will see a different way to implement AMB, worked out in the file "ambsch.scm", which allows ordinary Scheme programs to freely intermix with code that includes nondeterministic search.

But before we try to understand the implementation, it is useful to review what can be done with AMB. If you load "ambsch.scm" into MIT Scheme you can run examples such as the ones in the comments at the end of the "ambsch.scm" file, and you can solve the following puzzle.

Problem 6.1: Warmup (From SICP Exercise 4.43, p.420)

Formalize and solve the following puzzle with AMB:

> Mary Ann Moore’s father has a yacht and so has each of his four friends: Colonel Downing, Mr. Hall, Sir Barnacle Hood, and Dr. Parker. Each of the five also has one daughter and each has named his yacht after a daughter of one of the others. Sir Barnacle’s yacht is the Gabrielle, Mr. Moore owns the Lorna; Mr. Hall the Rosalind. The Melissa, owned by Colonel Downing, is named after Sir Barnacle’s daughter. Gabrielle’s father owns the yacht that is named after Dr. Parker’s daughter. Who is Lorna’s father?

You must use AMB to specify the alternatives that are possible for each choice. Also determine how many solutions there are if we are not told that Mary Ann’s last name is Moore.

Fun with Current Continuation

Before we can understand how the ambsch mechanism works we have to get deeper into continuations. Explicit underlying continuations are one of the most powerful (and the most dangerous) tools of a programmer. Scheme provides the ability for a programmer to get the underlying continuation of an expression. But most other languages do not support the use of first-class continuations. (Some other languages that do have first-class continuations include SML, Ruby, and Smalltalk.)

Whenever a Scheme expression is evaluated, a continuation exists that wants the result of the expression. The continuation represents an entire (default) future for the computation. If the expression is evaluated at top level, for example, the continuation will take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top-level continuation to be printed. Normally these ubiquitous continuations are hidden behind the scenes and programmers don’t think much about them. On the rare occasions that you may need to deal explicitly with continuations, call-with-current-continuation lets you do so by creating a procedure that acts just like the current continuation.

See the on-line MIT/GNU Scheme Reference Manual, Section 12.4, for a detailed description of CALL-WITH-CURRENT-CONTINUATION.

Explicit continuations may be powerful and sometimes useful, but they are rarely necessary. One common usage case is for non-local exits. Another is for resuming a suspended computation for backtracking. Yet another is coroutining (which we will explore in a later problem set).

Continuations as Non-Local Exits

Consider the following simple example of a non-local exit continuation (adapted from the MIT/GNU Scheme Reference Manual [Section 12.4]):

```
(call-with-current-continuation
 (lambda (exit)
   (for-each (lambda (x)
               (if (negative? x) (exit x)))
             '(54 0 37 -3 245 -19))     ; **
   #t))
;Value: -3
```

Because Scheme’s for-each procedure walks the list in left-to-right order, the first negative element encountered is -3, which is immediately returned. Had the list contained no negative numbers, the result would have been #t (since the body of the lambda form is a sequence of two expressions, the for-each expression followed by #t).

In a larger context, this might appear within some other form, like the following definition (explained below) in file "funco.scm":

```
     (define (funco:first-negative list-of-numbers)
       (call-with-current-continuation
        (lambda (k_exit)
          (or (call-with-current-continuation
               (lambda (k_shortcut)
                 (for-each (lambda (n)
                             (cond ((not (number? n))
                                    (pp `(not-a-number: ,n))
                                    (k_exit #f))
                                   ((negative? n)
                                    (k_shortcut n))
                                   (else
                                    ':keep-looking)))
                           list-of-numbers)
                 #f ;; Fall-through sentinel:  no negatives found.
                 ))
              ':no-negatives-found))))

     #|
     (funco:first-negative '(54 0 37 -3 245 -19))
     ;Value: -3

     (funco:first-negative '(54 0 37  3 245  19))
     ;Value: :no-negatives-found

     (funco:first-negative '(54 0 37 no 245 boo))
     (not-a-number: no)
     ;Value: #f
     |#
```

This demonstrates nested continuations, where the outermost k_exit continuation exits the entire call to funco:first-negative while the inner k_shortcut continuation exits only to the enclosing disjunction (or), then continues from there.

In short, if a continuation captured by call-with-current-continuation is ever invoked (with value V), then the computation will continue by returning V as the value of the call to call-with-current-continuation and resuming execution normally from there. [This is a bit tricky so look at the code above and re-read this last sentence a couple times until it makes sense... and please suggest alternative wording that might be less quixotically obtuse.]

```
-------------
Problem 6.2:

A. Define a simple procedure, snark-hunt, that takes a tree of symbols
   as argument and recursively descends it looking for the symbol
   'snark at any leaf.  It should immediately halt and return #t if
   one is found; #f otherwise.  Use call-with-current-continuation.

   If it helps, feel free to assume that all input trees will be valid
   non-null lists of tree-or-symbol elements, or whatever other data
   representation you find convenient.

   E.g.,

    (snark-hunt '(((a b c) d (e f)) g (((snark . "oops") h) (i . j))))
    ;Value: #t

   Note that the dotted pairs in the above violate (intentionally) the
   assumption that the input is comprised solely of proper lists of
   tree-or-symbol elements, so overruns may well result in errors.

B. How might you verify that it exits immediately rather than silently
   returning through multiple return levels?  Define a new procedure,
   snark-hunt/instrumented, to demonstrate this.  [Hint:  setting an
   exit status flag then signaling an error on wayward return paths
   might work if placed carefully, but simply tracing via pp may be
   easier.  Whatever quick and dirty hack that works will do.  The
   goal here is to build your intuition about continuations, not to
   ship product-quality code.  Briefly explain your strategy.]
-------------
```

Continuations for Backtracking

The preceding was somewhat simplistic since the continuations captured were used only for non-local exits. Specifically, they were not used for backtracking. Moreover, they were never re-entered once invoked.

Now consider the following slightly more interesting scenario:

```
(define *k_re-funco*)
(define       funco)

#|
(begin
  (set! funco (+ 2 (call-with-current-continuation
                     (lambda (k_re-funco)
                       (set! *k_re-funco* k_re-funco)
                       3))))
  ':ok)
;Value: :ok

funco
;Value: 5

(*k_re-funco* 4)
;Value: :ok

funco
;Value: 6

(*k_re-funco* 5)
;Value: :ok

funco
;Value: 7
|#
```

Note carefully how re-entering this captured continuation returns control to the point before the add and, therefore, before assigning variable funco and returning the symbol ':ok. This is why invoking it always returns the symbol ':ok, not the value passed to the exported continuation being re-entered (obviously) and not the new value to which that variable is re-assigned nor its old value nor unspecific.

This and the other examples in file "funco.scm" (attached) demonstrate how to re-enter a captured continuation to proceed from intermediate return points. This mechanism is used for backtracking in "ambsch.scm".

Continuations and Dynamic Contexts

We’ve already seen a few instances of dynamic binding via FLUID-LET in lecture. Although assignment violates referential transparency, fluid binding can be handy for locally overriding a free variable’s value.

For example, consider the following code fragment:

```
(define *trace?* #f)

(define (foo x)
  (set! *trace?* #t)
  (let ((result (bar x))) ;; bar may pp status when *trace?* set
    (set! *trace?* #f)
    result))
```

This works as expected only so long as bar does not capture and export a continuation that can be used to re-enter bar’s body. Moreover, if bar exits by invoking a continuation that bypasses the normal return mechanism that LET-binds result, the *trace?* flag may not be reset on the way out. Worse, this presumes *trace?* is always #f on entry.

To handle side-effects like this in the face of (possibly hidden) first-class continuations, a new dynamic binding form named FLUID-LET is provided that assigns (rather than LET-binds) variables on entry and reassigns them to their previous values upon exit, whether exiting via the normal return mechanism or through some captured continuation.

Thus, FLUID-LET allows parameterization of subsystems with a condition that is in effect over a controlled time interval (an extent) rather than over a lexically apparent textual region of code (a scope).

The FLUID-LET special form is documented in the on-line MIT/GNU Scheme Reference Manual, Section 2.3 Dynamic Binding (q.v.).

In this case, for example, the expected behavior can be achieved by rewriting the above code fragment as:

```
(define (foo x)
  (fluid-let ((*trace?* #t))
    (bar x)))
```

This mechanism is used in a few places in "ambsch.scm" to allow arbitrary nesting of depth-first verse breadth-first scheduling. It is also used by the mildly hackish amb-collect-values device.

Dynamic Contexts and Within-Continuation

The story gets really interesting when we define a thunk (a procedure of no arguments) at some control point in order to delay evaluation of its body, but we wish to invoke it in the dynamic context of its definition’s control point, not the dynamic context in flight at its eventual point of call.

For example, consider the following slightly contrived code fragment:

```
(define (funco:test-k-thunk k-thunk)
  (let ((*foo* 2))                       ;---------------------.
    (define (foo-thunk) *foo*)           ; *foo* is 2 out here. :
    (call-with-current-continuation      ;                      :
     (lambda (k)                         ;                      :
       (fluid-let ((*foo* 3))   ;--------------------.          :
         (k-thunk k foo-thunk)  ; *foo* is 3 in here. :         :
         )                      ;--------------------'          :
       ))                                ; *foo* is 2 out here. :
    ))                                   ;---------------------'

#|
(funco:test-k-thunk (lambda (k thunk)
                      (k (thunk))))
;Value: 3

(funco:test-k-thunk (lambda (k thunk)
                      (within-continuation k thunk)))
;Value: 2
|#
```

The WITHIN-CONTINUATION procedure is documented in the MIT/GNU Scheme Reference Manual [Section 12.4]). In short, it unrolls the dynamic context to that of the continuation, k, before invoking the thunk, the result of which is then passed to the continuation, k.

In "ambsch.scm", WITHIN-CONTINUATION is used to ensure that sibling AMB arguments are called in the dynamic context in which they were introduced, not the dynamic context in which they are eventually invoked.

The use of WITHIN-CONTINUATION ensures that each AMB alternative backtracks to appropriate nested search strategy. It also avoids the unnecessary accumulation of control state when exploring alternatives.

From Continuations to AMB

Now that we have had experience with explicit expression continuations we can begin to understand the code in "ambsch.scm". The heart of the backtracker is amb-list, which takes a sequence of sibling thunks, each representing an alternative value for the amb expression. The thunks were produced by the amb macro, which syntactically transforms amb expressions into amb-list expressions, as follows:

```
(amb <e1> ... <en>) ==>
  (amb-list (list (lambda () <e1>) ... (lambda () <en>)))
```

The search schedule maintains an agenda of thunks that proceed the computation when it is necessary for an amb expression to return with a new alternative value. For a particular amb expression these thunks are constructed so as to return from that amb expression, using the continuation, k, captured at the entrance to its enclosing amb-list. The within-continuation expression, which is almost equivalent to the call (k (alternative)), prevents the capture of pieces of the control stack that are unnecessary for continuing the computation correctly.

Amb-list first adds the returners for its alternative values to the search schedule and then yields control to the first pending returner.

```
(define (amb-list alternatives)
  (if (null? alternatives)
      (set! *number-of-calls-to-fail*
            (+ *number-of-calls-to-fail* 1)))
  (call-with-current-continuation
   (lambda (k)
     (add-to-search-schedule
      (map (lambda (alternative)
             (lambda ()
               (within-continuation k alternative)))
           alternatives))
     (yield))))


(define (yield)
  (if (stack&queue-empty? *search-schedule*)
      (*top-level* #f)
      ((pop! *search-schedule*))))
```

Note that procedure add-to-search-schedule is fluid bound either to add-to-depth-first-search-schedule (the default behavior) or else to add-to-breadth-first-search-schedule. See "ambsch.scm" for details.

Breadth -v- Depth

Consider the following experiment:

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (let ((j (an-integer-between i high)))
      (let ((k (an-integer-between j high)))
        (set! count (+ count 1))                   ; **
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))

(define count 0)
#|
(begin
  (init-amb)
  (set! count 0)
  (with-breadth-first-schedule
    (lambda () (pp (a-pythagorean-triple-between 10 20)))))
(12 16 20)

count
;Value: 246

*number-of-calls-to-fail*
;Value: 282


(begin
  (init-amb)
  (set! count 0)
  (with-depth-first-schedule
    (lambda () (pp (a-pythagorean-triple-between 10 20)))))
(12 16 20)

count
;Value: 156

*number-of-calls-to-fail*
;Value: 182
|#
```

Problem 6.3:

Explain the different counts between depth-first and breadth-first
(in rough terms, not the exact counts).

Also, where are the extra calls to fail coming from?

Considering that the breadth-first search does more work, why is the
a-pythagorean-triple-from search [AX 3.f in "ambsch.scm"] not usable
under the depth-first search strategy?

Less Deterministic Non-Determinism

Eva Lu Ator chides that a criticism one might make of our AMB implementation is that it's not as non-deterministic as one might sometimes like. Specifically, given a list of alternatives in an AMB form, we always choose the leftmost alternative first then the second leftmost and so on in left-to-right order.

She suggests that one might wish to override this choice, say, with right-to-left alternation or even in random order. Specifically, she'd like something like:

```
(with-left-to-right-alternation <thunk>)
(with-right-to-left-alternation <thunk>)
(with-random-order-alternation  <thunk>)
```

She's quick to point out that this choice is independent of the choice of depth-first or breadth-first (or whatever else) search order one might choose.

Problem 6.4:

A. Under what circumstances might you want an unordered (random) AMB? Craft a specific short example to use as a test case below.

B. Implement these three alternatives and give an example use of each. For simplicity and uniformity, model your code after that for with-depth-first-schedule, add-to-depth-first-search-schedule, etc. [Hint: Feel free to use the native MIT Scheme RANDOM procedure.]

Neurological Origami

Consider the following brain twister:

```
(define moby-brain-twister-test
  (lambda ()
    (let ((x) (y) (z))
      (set! x (amb 1 2 3))
      (pp (list x))
      (set! y (amb 'a 'b))
      (pp (list x y))
      (set! z (amb #t #f))
      (pp (list x y z))
      (amb))))
#|
(with-breadth-first-schedule moby-brain-twister-test)
(1)
(2)
(3)
(3 a)
(3 b)
(3 a)
(3 b)
(3 a)
(3 b)
(3 b #t)
(3 b #f)
(3 b #t)
(3 b #f)
(3 b #t)
(3 b #f)
(3 b #t)
(3 b #f)
(3 b #t)
(3 b #f)
(3 b #t)
(3 b #f)
;Value: #f
|#
```

Contrast this trace with the breadth-first elementary backtrack test
AMB example from "ambsch.scm" [viz., AX 1.b].

Problem 6.5:

Why does this weird thing happen?

The explanation is very simple, but this took us many hours to
understand.

[Hint: Look at (with-depth-first-schedule moby-brain-twister-test).]

Potential Project Topics

Problem 6.6: (optional!)

In the ‘‘Continuations and Dynamic Contexts’’ discussion section above, it was claimed that the breadth-first and depth-first search strategies can be arbitrarily nested within AMB forms.

Does the nesting of depth-first and breadth-first scheduling work correctly as currently implemented in "ambsch.scm"? Specifically, design an experiment that exposes the bug (if there is one) or that demonstrates anecdotally that it does work correctly (if it does). Explain your rationale.

This involves crafting a couple experiments that distinguish between depth-first and breadth-first search strategies then composing them in interesting ways to demonstrate local control over nested searches.

Identifying a natural class of problems for which this flexibility is useful--- not just hacked together to prove a point--- might be a fine topic for an independent project. Don’t spend too much time on it yet.

Problem 6.7: (optional!)

At the end of the examples.scm file (attached) you will find a simple parser of "natural language." Of course, this is really pretty crude. It is fun to see how much real language you could parse this way. One possible extension, besides adding more rules and more words is to incorporate probabilistic information, such as the probability that a word appears as a particular part of speech or that common idiomatic expressions appear together.

Unfortunately, there is no semantic attachment mechanism in this grammar. This is the real stuff, but it is very hard (and rather a matter of research.) If you are interested in this kind of work you should certainly look at the papers of Ray Jackendoff:

      http://en.wikipedia.org/wiki/Ray_Jackendoff

Make some interesting extensions to this grammar, or make a similar grammar for some other language you may know.

```
;;;; File:  ambsch.scm
;;;; Extension of Scheme for amb
;;;   amb is the ambiguous operator of McCarthy.

;;; (load "stack-queue.scm")

(define-syntax amb
  (sc-macro-transformer
   (lambda (form uenv)
     `(amb-list
       (list ,@(map (lambda (arg)
                      `(lambda ()
                         ,(close-syntax arg uenv)))
                    (cdr form)))))))

(define *number-of-calls-to-fail* 0)    ;for metering.

(define (amb-list alternatives)
  (if (null? alternatives)
      (set! *number-of-calls-to-fail*
            (+ *number-of-calls-to-fail* 1)))
  (call-with-current-continuation
   (lambda (k)
     (add-to-search-schedule
      (map (lambda (alternative)
             (lambda ()
               (within-continuation k alternative)))
           alternatives))
     (yield))))


;;; amb-set! is an assignment operator
;;;  that gets undone on backtracking.

(define-syntax amb-set!
  (sc-macro-transformer
   (lambda (form uenv)
     (compile-amb-set (cadr form) (caddr form) uenv))))

(define (compile-amb-set var val-expr uenv)
  (let ((var (close-syntax var uenv))
        (val (close-syntax val-expr uenv)))
    `(let ((old-value ,var))
       (effect-wrapper
        (lambda ()
          (set! ,var ,val))
        (lambda ()
          (set! ,var old-value))))))


;;; A general wrapper for undoable effects

(define (effect-wrapper doer undoer)
  (force-next
   (lambda () (undoer) (yield)))
  (doer))
```

```
;;; Alternative search strategy wrappers

(define (with-depth-first-schedule thunk)
  (call-with-current-continuation
   (lambda (k)
     (fluid-let ((add-to-search-schedule
                  add-to-depth-first-search-schedule)
                 (*search-schedule* (empty-search-schedule))
                 (*top-level* k))
       (thunk)))))

(define (with-breadth-first-schedule thunk)
  (call-with-current-continuation
   (lambda (k)
     (fluid-let ((add-to-search-schedule
                  add-to-breadth-first-search-schedule)
                 (*search-schedule* (empty-search-schedule))
                 (*top-level* k))
       (thunk)))))


;;; Representation of the search schedule

(define *search-schedule*)

(define (empty-search-schedule)
  (make-stack&queue))

(define (yield)
  (if (stack&queue-empty? *search-schedule*)
      (*top-level* #f)
      ((pop! *search-schedule*))))

(define (force-next thunk)
  (push! *search-schedule* thunk))

;;; Alternative search strategies

(define (add-to-depth-first-search-schedule alternatives)
  (for-each (lambda (alternative)
              (push! *search-schedule* alternative))
            (reverse alternatives)))

(define (add-to-breadth-first-search-schedule alternatives)
  (for-each (lambda (alternative)
              (add-to-end! *search-schedule* alternative))
            alternatives))
```

```
;;; For incremental interactive experiments from REPL.

(define (init-amb)
  (set! *search-schedule* (empty-search-schedule))
  (set! *number-of-calls-to-fail* 0)
  'done)

(define add-to-search-schedule ;; Default is depth 1st
  add-to-depth-first-search-schedule)

(define *top-level*
  (lambda (ignore)
    (display ";No more alternatives\n")
    (abort->top-level unspecific)))
```

```
;;; AX 1 - Elementary backtrack test.

(define elementary-backtrack-test
  (lambda ()
    (let ((x (amb 1 2 3)))
      (pp (list x))
      (let ((y (amb 'a 'b)))
        (pp (list x y))
        (let ((z (amb #t #f)))
          (pp (list x y z)))))
    (amb)))
#|
;; AX 1.d - Elementary backtrack test.  [Depth First]

(with-depth-first-schedule elementary-backtrack-test)
(1)
(1 a)
(1 a #t)
(1 a #f)
(1 b)
(1 b #t)
(1 b #f)
(2)
(2 a)
(2 a #t)
(2 a #f)
(2 b)
(2 b #t)
(2 b #f)
(3)
(3 a)
(3 a #t)
(3 a #f)
(3 b)
(3 b #t)
(3 b #f)
;Value: #f
```

```
;; AX 1.b - Elementary backtrack test.  [Breadth First]

(with-breadth-first-schedule elementary-backtrack-test)
(1)
(2)
(3)
(1 a)
(1 b)
(2 a)
(2 b)
(3 a)
(3 b)
(1 a #t)
(1 a #f)
(1 b #t)
(1 b #f)
(2 a #t)
(2 a #f)
(2 b #t)
(2 b #f)
(3 a #t)
(3 a #f)
(3 b #t)
(3 b #f)
;Value: #f
|#
```

```
;;; AX 2 - Testing undoable assignment.

(define testing-undoable-assignment
  (lambda ()
    (let ((x (amb 1 2 3)) (y 0) (z 0))
      (pp `(before ,x ,y ,z))
      (amb-set! y x)
      (pp `(after ,x ,y ,z))
      (amb-set! z (amb 3.14 2.718))
      (pp `(zset ,x ,y ,z))
      (amb-set! x (+ y z))
      (pp `(xset ,x ,y ,z))
      (amb))))
#|
;;; AX 2.d - Testing undoable assignment.  [Depth First]

(with-depth-first-schedule testing-undoable-assignment)
(before 1 0 0)
(after 1 1 0)
(zset 1 1 3.14)
(xset 4.140000000000001 1 3.14)
(zset 1 1 2.718)
(xset 3.718 1 2.718)
(before 2 0 0)
(after 2 2 0)
(zset 2 2 3.14)
(xset 5.140000000000001 2 3.14)
(zset 2 2 2.718)
(xset 4.718 2 2.718)
(before 3 0 0)
(after 3 3 0)
(zset 3 3 3.14)
(xset 6.140000000000001 3 3.14)
(zset 3 3 2.718)
(xset 5.718 3 2.718)
;Value: #f
|#
```

```
;;; AX 3 - Pythagorean triples

;; In breadth-first we get useful results here.
;; None from depth-first.

;; AX 3.f - A Pythagorean triple from...

(define (a-pythagorean-triple-from low)
  (let ((i (an-integer-from low)))
    (let ((j (an-integer-from i)))
      (let ((k (an-integer-from j)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))

(define (require p)
  (if (not p) (amb)))

(define (an-integer-from low)
  (amb low (an-integer-from (+ low 1))))

#|
(with-breadth-first-schedule
    (lambda ()
      (pp (a-pythagorean-triple-from 1))
      (amb)))
(3 4 5)
(6 8 10)
(5 12 13)
(9 12 15)
(8 15 17)
(12 16 20)
(7 24 25)
(15 20 25)
(10 24 26)
(20 21 29)
(18 24 30)
(16 30 34)
(21 28 35)
(12 35 37)
(15 36 39)
(24 32 40)
(9 40 41)
(27 36 45)
(14 48 50)
(30 40 50)
(24 45 51)
(20 48 52)
(28 45 53)
(33 44 55)
(40 42 58)
(36 48 60)
(11 60 61)
(16 63 65)
;Quit!
|#
```

```
;; AX 3.b - A Pythagorean triple between...

;; For example, for controlling search:

(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (let ((j (an-integer-between i high)))
      (let ((k (an-integer-between j high)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))

(define (an-integer-between low high)
  (require (<= low high))
  (amb low
       (an-integer-between (+ low 1) high)))

;; A useful device:

(define (amb-collect-values result-thunk #!optional limit)
  (call-with-current-continuation
   (lambda (k)
     (let ((values '()) (count 0))
       (fluid-let ((*top-level* (lambda (ignore) (k values)))
                   (*search-schedule* (empty-search-schedule)))
         (let ((value (result-thunk)))
           (set! values (cons value values))
           (set! count (+ count 1))
           (if (and (not (default-object? limit))
                    (>= count limit))
               (k values))
           (amb)))))))
#|
(with-depth-first-schedule
    (lambda ()
      (let ((mid (amb-collect-values
                  (lambda ()
                    (a-pythagorean-triple-between 1 20))
                  ;; I want only 3, and
                  ;; I don't want to backtrack into this.
                  3)))
        (pp (list (a-pythagorean-triple-between 1 10)
                  mid
                  (a-pythagorean-triple-between 10 30)))
        (amb))))
((3 4 5) ((6 8 10) (5 12 13) (3 4 5)) (10 24 26))
((6 8 10) ((6 8 10) (5 12 13) (3 4 5)) (10 24 26))
((3 4 5) ((6 8 10) (5 12 13) (3 4 5)) (12 16 20))
((6 8 10) ((6 8 10) (5 12 13) (3 4 5)) (12 16 20))
((3 4 5) ((6 8 10) (5 12 13) (3 4 5)) (15 20 25))
((6 8 10) ((6 8 10) (5 12 13) (3 4 5)) (15 20 25))
((3 4 5) ((6 8 10) (5 12 13) (3 4 5)) (18 24 30))
((6 8 10) ((6 8 10) (5 12 13) (3 4 5)) (18 24 30))
((3 4 5) ((6 8 10) (5 12 13) (3 4 5)) (20 21 29))
((6 8 10) ((6 8 10) (5 12 13) (3 4 5)) (20 21 29))
;Value: #f
|#
```

```
;;;; File:  examples.scm

;;; SICP Section 4.3.2 : Logic Puzzles
;;;
;;; Baker, Cooper, Fletcher, Miller, and Smith live on
;;; different floors of a building that has only five
;;; floors.  Baker does not live on the top floor.
;;; Cooper does not live on the bottom floor.  Fletcher
;;; does not live on either the top or the bottom
;;; floor.  Miller lives on a higher floor than does
;;; Cooper.  Smith does not live on a floor adjacent to
;;; Fletcher’s.  Fletcher does not live on a floor
;;; adjacent to Cooper’s.  Where does everyone live?
;;;      (From Dinesman, 1968)


(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require
     (distinct?
      (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require
     (not (= (abs (- smith fletcher)) 1)))
    (require
     (not (= (abs (- fletcher cooper)) 1)))
    (list (list ’baker baker)
          (list ’cooper cooper)
          (list ’fletcher fletcher)
          (list ’miller miller)
          (list ’smith smith))))

(define (distinct? items)
  (cond ((null? items) #t)
        ((null? (cdr items)) #t)
        ((member (car items) (cdr items)) #f)
        (else (distinct? (cdr items)))))

#|
(init-amb)
;Value: done

(with-depth-first-schedule multiple-dwelling)
;Value: ((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))

(amb)
;No more alternatives
|#
```

```
;;; From SICP Section 4.3.2
;;; Parsing natural language

(define (parse input)
  (amb-set! *unparsed* input)
  (let ((sent (parse-sentence)))
    (require (null? *unparsed*))
    sent))

(define *unparsed* '())

(define (parse-sentence)
  (let* ((np (parse-noun-phrase))
         (verb (parse-verb-phrase)))
    (list 'sentence np verb)))

(define (parse-noun-phrase)
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
         (maybe-extend
          (list 'noun-phrase
                noun-phrase
                (parse-prepositional-phrase)))))
  (maybe-extend (parse-s-noun-phrase)))

(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    (amb verb-phrase
         (maybe-extend
          (list 'verb-phrase
                verb-phrase
                (parse-prepositional-phrase)))))
  (maybe-extend (parse-word verbs)))
```

```
(define (parse-s-noun-phrase)
  (let* ((article (parse-word articles))
         (noun (parse-word nouns)))
    (list 's-noun-phrase article noun)))

(define (parse-prepositional-phrase)
  (let* ((preposition
          (parse-word prepositions))
         (np (parse-noun-phrase)))
    (list 'prep-phrase preposition np)))

(define (parse-word word-list)
  (require (not (null? *unparsed*)))
  (require (memq (car *unparsed*)
                 (cdr word-list)))
  (let ((found-word (car *unparsed*)))
    (amb-set! *unparsed* (cdr *unparsed*))
    (list (car word-list) found-word)))

(define nouns
  '(noun student professor cat class))

(define verbs
  '(verb studies lectures eats sleeps))

(define articles
  '(article the a))

(define prepositions
  '(prep for to in by with))
```

```
#|
(init-amb)
;Value: done

(pp
 (parse
  '(the student with the cat sleeps in the class)))

(sentence
 (noun-phrase
  (s-noun-phrase (article the) (noun student))
  (prep-phrase (prep with)
             (s-noun-phrase (article the)
                            (noun cat))))
 (verb-phrase
  (verb sleeps)
  (prep-phrase (prep in)
             (s-noun-phrase (article the)
                            (noun class)))))
;Unspecified return value

(amb)
;No more alternatives
|#
```

```
#|
(init-amb)
;Value: done

(pp
 (parse
  '(the professor lectures
        to the student with the cat)))

(sentence
 (s-noun-phrase (article the) (noun professor))
 (verb-phrase
  (verb-phrase
   (verb lectures)
   (prep-phrase (prep to)
                (s-noun-phrase (article the)
                               (noun student))))
  (prep-phrase (prep with)
               (s-noun-phrase (article the)
                              (noun cat)))))
;Unspecified return value

(amb)

(sentence
 (s-noun-phrase (article the) (noun professor))
 (verb-phrase
  (verb lectures)
  (prep-phrase
   (prep to)
   (noun-phrase
    (s-noun-phrase (article the)
                   (noun student))
    (prep-phrase (prep with)
                 (s-noun-phrase (article the)
                                (noun cat)))))))
;Unspecified return value

(amb)
;No more alternatives
|#
```