Massachusetts Institute of Technology

6.034 Artificial Intelligence

Assignment # **1** Due: **15/Sept/98** 6.034 Item # **1**

Be sure to indicate on your solution the time, room number and recitation instructor for the recitation section that you normally attend. Also, write BOTH partners name on the solution.

Goals

This assignment is intended to

- encourage you to (re)familiarize yourself with Scheme.
- serve as a diagnostic. This problem set involves understanding, manipulating and changing a Scheme program of the complexity that we will be using in the course. If you have trouble doing this assignment, you will likely have trouble with the subsequent assignments.

Look at the file `/mit/6.034/doc/hints.txt` for hints on running MIT Scheme on Athena. If you think of anything we should add to the file, let us know.

The two problems in this problem set are independent. The program in Problem 1 is not used in Problem 2. If you detect some subtle problem in the specification of these problems, just go ahead and make a reasonable choice. We are not going to be picky.

Problem 1

Let's warm up with a very simple Scheme program to count the number of "atoms" in an arbitrary Scheme expression. An "atom" in this context is anything that does not satisfy the `pair?` predicate. We do want to count the final null list `()` in a list as a separate atom. For example:

```
(count '()) => 1
(count 'a) => 1
(count '(the (quick brown) fox)) => 6
(count '(a (b 1 (c . #t) d 23 #f))) => 10
(count '(a (b . c) . d)) => 4
(count '(a (b . c) () . d)) => 5
```

Problem 2

The file /mit/6.034/ps1/match.scm (included at the end of this handout) has Scheme code for a simple list matcher. The matcher is given two lists as input, one (the pattern) contains variables and the other (the data) does not, and it returns a list of variable bindings that would make the pattern match the data. We denote simple variables by a list whose first element is the symbol ? and whose second element is the name of the variable, e.g. (? x) represents the variable named x. So:

```
(match '(a (? x) c) '(a b c)) => (bindings (x b))
```

Note that the return value is a list, whose car is the symbol bindings and whose cdr is a list of lists (possibly null). The first element of the component lists is the name of a variable and the second element of these lists is the matched value. Here are some more examples:

```
(match '(a ((? x) c) d) '(a (b c) d)) => (bindings (x b))
(match '(a ((? x) c) (? y)) '(a (b c) c)) => (bindings (y c) (x b))
(match '(a (b c) d) '(a (b c) d)) => (bindings)
```

As you can see above, variables can occur anywhere in the pattern, not necessarily at top level. Also note that when a match returns the a list whose cdr is an empty list (bindings), this indicates a succesful match, that is, a match that did not require binding any variables. A failed match is indicated by returning #f – which unfortunately prints as () in MIT Scheme.

```
(match '(a b c d) '(a (b c) c)) => ()
```

If a variable with the same name occurs more than once in a pattern, it must match equal parts of the data:

```
(match '(a (? x) c (? x) e) '(a b c b e)) => (bindings (x b))
(match '(a (? x) c (? x) e) '(a b c d e)) => ()
(match '(a (? x) c (? y) e) '(a b c d e)) => (bindings (y d) (x b))
```

The code also supports nameless variables (? _), which simply match but whose matching values are not returned and not constrained to be equal:

```
(match '(a (? _) c (? _) e) '(a b c b e)) => (bindings)
(match '(a (? _) c (? _) e) '(a b c d e)) => (bindings)
nnn
```

A simple variable can match any component of the data list, but it will not match a sublist of the data:

```
(match '(a (? x) d) '(a b c d)) => ()
```

What we want to do in this problem is extend the matcher to handle a new class of variable – called "segment variables" – that can match sublists of the data:

```
(match '(a (* x) d) '(a b c d)) => (bindings (x (b c)))
(match '(a (* x) d) '(a d)) => (bindings (x ()))
(match '((* x) a (* x) c) '(a c)) => (bindings (x ()))
(match '((* x) a (* x) c) '(b a b c)) => (bindings (x (b)))
(match '((* x) a (* x) c) '(b a d c)) => ()
(match '((a (* x)) d e) '((a b c) d e)) => (bindings (x (b c)))
```

Note that the binding value of a segment variable is always a list, possibly a null list. Note also that there is, in general, no unique assignment to the segment variables to produce a match. We just want to produce SOME assignment if one exists. **Hint:** the simple implementation will be very inefficient, but don't worry about that.

It is an error to have a simple variable and a segment variable of the same name in a pattern, but don't bother checking for this error explicitely since it will complicate your code. Also, feel free to ignore the case of standalone variables that are not part of a list, that is, `(match '(?  x) ...)` and `(match '(* x) ...)` are illegal.

You might notice that the current implementation, without segment variables, is a little more complex than necessary. This is because the code was designed to make adding segment variables easier...

Please hand in only the code YOU wrote, and not the whole file. You should only need to modify the top-level `match` function and then define a couple of new functions. If you think you need to write pages of code, think again!

```
----------------------------------------------------------------------
;;; UTILITIES
(define rest cdr)               ;; a more mnemonic name for CDR
(define *null-bindings* '(bindings)) ;; initial value of bindings

;;; Something that prints as a single symbol
(define (atom? x) (or (symbol? x) (number? x) (boolean? x) (null? x)))

;;; A test for a (? x) type of variable.
(define (simple-variable? x) (and (pair? x) (eq? '? (first x))))

;;;; MATCHER

;; The top-level call, just calls the recursive function do-match with
;; initial values for the bindings list.
(define (match p d) (do-match p d *null-bindings*))

(define (do-match p d bindings)
  "
  Purpose:      Matching
  Arguments:    Pattern, datum, optional bindings.
  Returns:      A list of bindings (whose car is the symbol 'bindings)
                or #f if no match is possible.
  Remarks:      Pattern variables are indicated by (? <variable name>)
                Prolog's nameless variable is indicated by (? _)
  "
  (cond ((and (null? p) (null? d))
         bindings)
        ((or (atom? p) (atom? d))       ; This is not allowed
         #f)
        ((and (atom? (first p)) (atom? (first d)))
         (match-atoms p d bindings))
        ((simple-variable? (first p))
         (match-simple-variable p d bindings))
        ((and (pair? (first p)) (pair? (first d)))
         (match-pieces p d bindings))
        (else #f)))
```

```
(define (match-atoms p d bindings)
  ;;Are the pattern and datum the same:
  (if (eqv? (first p) (first d))        ; eqv? not eq? because of (non-integer) numbers.
      ;;If so, return the value of BINDINGS:
      (do-match (rest p) (rest d) bindings)
      ;;Otherwise, return #f
      #f))

(define (match-simple-variable p d bindings)
  ;; Check for either consistent match of previously bound variable
  ;; or bind a new value and proceed with the match.
  (let ((binding (find-binding (first p) bindings)))
    ;;Is the pattern variable on the list of bindings:
    (if binding
        ;;If it is, substitute its value and try match again:
        (do-match (cons (extract-value binding) (rest p))
                  d
                  bindings)
        ;;Otherwise, add new binding and match the rest:
        (do-match (rest p)
                  (rest d)
                  (add-binding (first p) (first d) bindings)))))

(define (match-pieces p d bindings)
  ;; This assumes that the CAR's of both p and d are pairs.
  (let ((result (do-match (first p) (first d) bindings)))
    ;;See if the FIRST parts match producing new bindings:
    (if result
        ;;If they do match, try the REST parts using the resulting bindings:
        (do-match (rest p) (rest d) result)
        ;; Else fail
        #f)))

;;;; AUXILIARY PROCEDURES

;;; adds a new binding for a variable to a bindings list.
;;; A variable expression is one of (? x) or (* x)
(define (add-binding variable-expression datum bindings)
 (if (eq? '_ (second variable-expression))
     ;; If it is a nameless variable, it matches anything but does not add bindings
     bindings
     ;; add a new binding, remembering that the car of the list is the symbol 'bindings
     (cons (first bindings)
           (cons (list (second variable-expression) datum)
                 (rest bindings)))))

;;; returns the binding for a variable.
;;; A variable expression is one of (? x) or (* x)
(define (find-binding variable-expression binding)
  (if (not (eq? '_ (second variable-expression)))
    ;; If, it's a nameless variable, don't look for binding.  Otherwise, use
    ;; assoc to find the binding.  (assoc x y) looks for a list in y whose car
    ;; is x and returns that, which is just what we need.
    (assoc (second variable-expression) (rest binding))
    #f))

;;; a binding is just a list (key value)
(define (extract-key binding) (first binding))
(define (extract-value binding) (second binding))
```