MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 1997

Problem Set 5

Issued: Tuesday, October 14, 1997

Due: Wednesday, October 22, in recitation.

Tutorial preparation for: Week of October 20.
SICP Exercises 2.74, 2.76, 3.1, 3.2, 3.3

Reading for this Problem Set: *SICP* Sections 2–2.2.3, 2.3
Reading for Lectures: *SICP* Sections 2.4–2.5.2, 3.1–2

A Pattern Matching System

There is much more code with this problem set than you have seen before. Don't be daunted: you do not need to understand, or even read, it all. Programmers are often confronted with systems which are too large for any one person to learn in detail. But well written, modular systems can be understood in small pieces; the challenge is to identify the relevant small pieces for the task at hand. That's what you should try to do here: the procedures you are asked to define are variants of a few small procedures in the accompanying code files; these are the only ones you really need to understand.

1 Eliza

The *ELIZA* or *DOCTOR* program, one of the most famous early interactive programs, is an amusing example of programming based on pattern matching rules. Here is a sample dialog with the program:

1.1 Eliza sample session

```
(doctor)                    ;START THE SESSION

(hello, what can i do for you?)         ;ELIZA
(i need help)                           ;CLIENT

(why do you say you need help ?)        ;ELIZA
(because my mother is driving me crazy) ;CLIENT

(tell me more about your mother)
(she makes me feel guilty)

(is it usual for you to feel guilty)
(not really)

(what makes you say not really ?)
(because I am ok when she is not around)

(are you feeling defensive?)
(maybe)

(why are you being short with me?)
(will you stop asking me so many questions)

(would you ask that of anyone else besides me?)
(your questions make me start to hate this)

(when did you start to hate ?)
quit
;Value: "Your session is over.  Good luck."
```

And here are the rules which determined ELIZA's behavior above:

```
(define eliza-rules
  (list
   (make-simple-rule
    '(~ mother ~)
    '(tell me more about your mother))
   (make-simple-rule
    '(~ because ~)
    '(are you feeling defensive?))
   (make-simple-rule
    '(~ you ~phrase)
    '(would you ask that of anyone else besides me?))
   (make-simple-rule
```

```
  '(?)
  '(Why are you being short with me?))
 (make-simple-rule
  '(~ feel ?adverb ~)
  '(Is it usual for you to feel ?adverb))
 (make-simple-rule
  '(I ~phrase)
  '(why do you say you ~phrase ?))
 (make-simple-rule
  '(~ to ?%verb ~)
  '(When did you start to ?%verb ?))
 (make-simple-rule
  '(~phrase)
  '(what makes you say ~phrase ?))
 (make-simple-rule
  'yes
  '(you sound confident))
 (make-simple-rule
  'no
  '(how negative do you feel?))
 (make-simple-rule
  '?
  '(say a little more please))))
```

1.2 Programming with Pattern Matching Rules

The Eliza program illustrates how using pattern matching rules can produce readable, useful, and easily modifiable programs. You can extend the power of Eliza simply by adding rules; the real Eliza program grew to have hundreds of them.

Rule-based programs are easy to develop and maintain because they insulate the programmer from managing program control. Program control is handled automatically by the pattern-matching, rule-applying system. Of course there is a downside to losing control: as rules are added to the system, even apparently innocuous ones, the behavior of the system may change unpredictably, and the search time for an applicable rule may grow prohibitively. Real pattern-matching systems use sophisticated procedures to search through the possible rule-applications as rapidly and, with luck, as productively as possible.

Although practical search procedures can be complex, the basic mechanism behind pattern matching is easy to understand. In this problem set we will now develop a simple pattern matching system written in Scheme. This system is designed to illustrate the important ideas behind pattern matching. We will also look at some further simple examples of symbolic computation by pattern matching.

2 Patterns, Matches, and Rules

A *pattern* is a mix of *atoms* which must be matched exactly, and *variables.* Our system has two kinds of variables: *question-mark variables* which start with "?" can match a single datum; and *tilde-variables* which start with "~" can match a (possibly empty) sequence of data items.

For example, the pattern

```
(?F + ?G)
```

will match any expression which is a parenthesized sum of two things. (The "+" is an atom which must match exactly.) The result of the match is a "dictionary" which indicates that the question-mark variable `?F` is matched to the first thing, and `?G` is matched to the second thing. So matching this pattern against the expression `(2 + (x - y))` would return a *dictionary* in which the meaning, or value, of the variable `?F` is 2, and the value of `?G` is `(x - y)`.

A pattern can also serve as a *template.* If a dictionary gives values to all the variables in a template, then the values can be plugged in—the variables can be *instantiated* with the values—to yield a "pure" data item without variables. For example, we could instantiate the template `((0 - ?G) - ?F)` with the dictionary above:

```
(let ((dict (make-dictionary
              '((?F 2) (?G (x - y))))))
  (instantiate '((0 - ?G) - ?F) dict))
;Value: ((0 - (x - y)) - 2)
```

The process for matching something against the pattern `(?F + ?G)` and then instantiating into the template `((0 - ?G) - ?F)` can be described by a *rule*

```
(define rule:negate-and-commute
 (make-simple-rule
  '(?F + ?G)
  '((0 - ?G) - ?F)))
```

If a pattern contains tilde-variables, there may be many ways to match it against a single datum. For example, the pattern

```
(~a ~b ?c ~d)
```

can be matched against the expression

```
(1 2 3 4)
```

in ten different ways resulting in ten distinct dictionaries:

```
(match '(~a ~b ?c ~d) '(1 2 3 4))
; Value:
(((~a ()) (~b ()) (?c 1) (~d (2 3 4)))
 ((~a ()) (~b (1)) (?c 2) (~d (3 4)))
 ((~a ()) (~b (1 2)) (?c 3) (~d (4)))
 ((~a ()) (~b (1 2 3)) (?c 4) (~d ()))
 ((~a (1)) (~b ()) (?c 2) (~d (3 4)))
 ((~a (1)) (~b (2)) (?c 3) (~d (4)))
 ((~a (1)) (~b (2 3)) (?c 4) (~d ()))
 ((~a (1 2)) (~b ()) (?c 3) (~d (4)))
 ((~a (1 2)) (~b (3)) (?c 4) (~d ()))
 ((~a (1 2 3)) (~b ()) (?c 4) (~d ())))
```

so applying the rule whose pattern is

```
(~a ~b ?c ~d)
```

and whose template, for example, is

```
(~a - ~b - ?c - ~d)
```

to the expression

```
(1 2 3 4)
```

would yield ten different expressions (see Exercise 7):

```
(all-rule-applications
   (make-simple-rule
     '(~a ~b ?c ~d)
     '(~a - ~b - ?c - ~d))
   '(1 2 3 4))
;Value:  ((- - 1 - 2 3 4)
          (- 1 - 2 - 3 4)
          (- 1 2 - 3 - 4)
          (- 1 2 3 - 4 -)
          (1 - - 2 - 3 4)
```

```
(1 - 2 - 3 - 4)
(1 - 2 3 - 4 -)
(1 2 - - 3 - 4)
(1 2 - 3 - 4 -)
(1 2 3 - - 4 -))
```

Pre-Lab Exercise 1:

How many different expressions result from applying the rule

```
(make-simple-rule
  '(~a ~b ?c ~d)
  '(~a ~b ?c ~d))
```

to the expression `(1 2 3 4)`?

The symbols `~` and `?` are special "nameless" variables, which match like ordinary tilde-variables and question-mark variables, respectively, but do not contribute entries to the dictionary of the match. For example, the pattern

```
(~ you ?word ~)
```

will match the expression

```
(you are what you are)
```

in two ways, but both ways yield the same one-entry dictionary in which the variable `?word` has the value `are`.

2.1 Variable Restrictions

A simple pattern for an Eliza script simply spots the word "to" followed by a *word*, and then responds with `(When did you start to` *word* `?))`. For example, the rule

```
(make-simple-rule
    '(~ to ?verb ~)
    '(When did you start to ?verb ?))
```

accounts for the dialog

```
(your questions make me start to hate this)

(when did you start to hate ?)
```

On the other hand, this same rule would also yield undesirable dialog like

```
(get to your point please)

(when did you start to your ?)
```

This rule can yield plausible dialog when "to" is followed by a verb, but we don't want it to apply if the variable `?verb` does not match a verb.

The need to restrict the data which a variable may match comes up in many examples. Our pattern matching system allows such restrictions to be placed on designated variables. By convention, *restricted variables* are indicated by `%` as their second character. So for example, the rule above would really be written

```
(make-simple-rule
    '(~ to ?%verb ~)
    '(When did you start to ?%verb ?))
```

and the restricted variable `?%verb` would be associated with the restriction procedure

```
(lambda (datum)
  (memq datum '(dance run sing cry laugh hate)))
```

When applied to a datum, the restriction procedure returns `#f` when `?%verb` is not allowed to match the datum.

The selector `var-restriction` can be applied to a variable to get its restriction procedure; for convenience, `var-restriction` returns the identically true procedure for variables which are not restricted. The restrictions are kept in the `restriction-table`.

```
(define (var-restriction var)
  (let ((str (symbol->string var)))
    (if (and (> (string-length str) 1)
             (char=? \#% (string-ref str 1)))
        (let ((binding (assq var restriction-table)))
          (if binding
              (cadr binding)
              (error "VAR-RESTRICTION: no entry for" var)))
        constant-true-function)))

(define restriction-table
  (list '(?%verb ?%num ...)
    (list (lambda (dat) (memq dat '(dance run sing cry laugh hate)))
          number? ...)))
```

Lab Exercise 2. Add three or four new simple-rules to *Eliza* and show the resulting dialog.

2.2 Matching

The procedure `match` takes a pattern and a datum and returns a list of all possible dictionaries by which the pattern matches the datum. If there is no match, it returns the empty list.

The simplest case is when the pattern is an atom or the empty list; the datum must match these exactly.

```
(define (is-atom? p)  ;must be matched exactly
  (or (number? p)
      (boolean? p)
      (and (symbol? p)
           (not (qmark-variable? p))
           (not (tilde-variable? p)))))
```

Another easy case is when the pattern is a question-mark variable; `match-qmark-variable` returns a list whose only element is a dictionary if the datum meets the variable's restriction, and otherwise returns the empty list.

```
(define (match-qmark-variable qvar dat)
  (cond ((qmark? qvar) (list the-empty-dictionary))
        (((var-restriction qvar) dat)
         (list (make-singleton-dictionary qvar dat)))
            ;DAT does not satisfy var-restriction:
        (else '()))        ;no match
```

A tilde-variable can match nothing, so when the datum is the empty list, it can be matched by a pattern consisting of any list of tilde-variables. The procedure `match-null` (see the attached code file `ps5match.scm`) checks whether a pattern is a list of tilde-variables, and if it is, returns a single dictionary in which each of the tilde-variables is bound to the empty list.

So the procedure `match` is

```
(define (match pat dat)
  (cond ((or (is-atom? pat) (null? pat))
         (if (equal? pat dat)           ;must match exactly
             (list the-empty-dictionary)
             '()))                      ;no match: return empty list
```

```
      ((qmark-variable? pat)
       (match-qmark-variable pat dat))
      ((null? dat) (match-null pat))
      ((is-atom? dat) '())           ;FAIL: PAT is a pair
      (else (match-pairs pat dat)))) ;PAT & DAT are pairs
```

2.2.1 Recursively Matching Pairs

The final, and main, case for `match` is when the pattern and datum are both pairs. If the `car` of the pattern is not a tilde-variable, then `match` proceeds recursively by matching the `car`'s of the pattern and datum, and then matching the `cdr`'s. It completes its task by combining, using the `dict-merge` operation, the dictionaries from the `car` match with the dictionaries of the `cdr` match.

```
(define (match-pairs pat dat)           ;PAT & DAT are pairs
  (let ((pat0 (car pat)))
    (cond
     ((tilde? pat0)
      (match-initial-tilde (cdr pat) dat))
     ((tilde-variable? pat0)
      (match-initial-tildevar pat0 (cdr pat) dat))
     (else      ;PAT does not start with a tilde-variable
      (let ((dicts1 (match pat0 (car dat))))
        (if (null? dicts1)
            '()
            (all-pairs dicts1
                       (match (cdr pat) (cdr dat))
                       dict-merge)))))))

;  (all-pairs '(x1 ... xn) '(y1 ... ym) op) =
;       (list
;         (op x1 y1) (op x1 y2) ... (op x1 ym)   --|
;                             .                   |
;                             .                   |  ONE BIG LIST
;                             .                   |
;         (op xn y1) (op xn y2) ... (op xn ym))  --|

;(tensor '(x1 ... xn) '(y1 ... ym) op) =
;   (list
;     (list (op x1 y1) (op x1 y2) ... (op x1 ym))  --|
;                         .                         |
;                         .                         |List of Lists
;                         .                         |
;     (list (op xn y1) (op xn y2) ... (op xn ym))) --|
```

```
(define (tensor x-list y-list op)
    (map (lambda (x)
           (map (lambda (y) (op x y))
                y-list))
         x-list))

(define (all-pairs x-list y-list op)
  (apply append (tensor x-list y-list op)))
```

2.2.2 Patterns Starting with Tilde

Tilde-variables are the reason a single pattern can match a single datum in many ways.

To see how matching works for tilde-variables, consider the pattern `(~var ?rest ~)` and the datum `(1 2 3)`. We can split the possible matches and their dictionaries into two separate groups: those in which `~var` matches none of the items in the datum and `(?rest ~)` matches all of `(1 2 3)`, and those in which `~var` matches at least the first item `1`. The first case happens to yield just a single dictionary

```
( (~var '()) (?rest 1) )
```

For the other case, we match `~var` against `1` and possibly `2` and `3` as well, with `(?rest ~)` matching whatever is left over. This is the same as matching the original pattern `(~var ?rest ~)` against `(2 3)`, which happens to yield two dictionaries,

```
(match '(~var ?rest ~) '(2 3))
;Value: ( ((~var ()) (?rest 2))   ((~var (2)) (?rest 3)) )
```

and then tacking the `1` onto whatever sequence `~var` matched in each of these dictionaries:

```
        ( ((~var (1)) (?rest 2))
          ((~var (1 2)) (?rest 3)) )
```

So the result of the full match is the three dictionaries:

```
(match '(~var ?rest ~) '(1 2 3))
;Value: ( ((~var ()) (?rest 1))   ;~var matches nothing
          ((~var (1)) (?rest 2))  ;~var matches at least 1
          ((~var (1 2)) (?rest 3)) ) ;~var matches at least 1
```

The procedure `match-initial-tildevar` works by using this same split into two cases.[1]

```
(define (match-initial-tildevar tilde-var pat-after-tildevar dat)
  (append
             ;dicts with TILDE-VAR matching empty:
   (map (dictionary-reviser tilde-var '())
        (match pat-after-tildevar dat))
             ;dicts with TILDE-VAR matching one or more items:
   (if (null? dat)
       '()
       (map
        (lambda (dict)
          ((dictionary-reviser
            tilde-var
            (cons (car dat) (lookup tilde-var dict)))
           dict))
        (match-initial-tildevar
         tilde-var
         pat-after-tildevar
         (cdr dat))))))
```

2.3 Restricted Rules

There are a number of familiar rewriting rules which go somewhat beyond simple rules with restrictions on individual variables. For example, we might want a rule which would rewrite any expression of the form

```
(exp + exp)
```

into

```
(2 * exp)
```

For this purpose, we allow rules which include a predicate, called the match restriction, which the *whole dictionary* of a match must satisfy before the rule can be applied. The effect of the rule above could be obtained by using the pattern

```
(?exp1 + ?exp2)
```

[1]The actual procedure `match-initial-tildevar` in the file `ps5match.scm` is slightly more complicated because it also checks that each item matched by the tilde-variable satisfies the variable's restriction.

with the match restriction that the dictionary values of `?exp1` and `?exp2` had to be the same. So the match restriction for the rule above would be

```
(define same-exps
   (lambda (dict)
     (equal? (lookup '?exp1 dict)
             (lookup '?exp2 dict))))
```

and the rule would be defined by:

```
(define rule:e+e
  (make-restricted-rule
   '(?exp1 + ?exp2)
   '(2 * ?exp1)
    same-exps))                     ; THE MATCH RESTRICTION
```

Lab Exercise 3. Add one or two new restricted-rules to *Eliza* and show the resulting dialog.

Lab Exercise 4. The `match-pairs` procedure used the `dict-merge` operation to combine a dictionary from a match of the `car`'s of pattern and datum, with a dictionary from a match of the `cdr`'s. Because we can restrict distinct variables to match the same thing, there is no need to allow more than one occurrence of a variable in a pattern. In that case, the `dict-merge` operation simply `append`'s the bindings of two dictionaries.

On the other hand, instead of using a restricted rule, a more natural way to express `rule:e+e` would be with a repeated variable in a simple rule:

```
(define rule:e+e
  (make-simple-rule
   '(?exp + ?exp)
   '(2 * ?exp)))
```

Modifying our system to handle repeated question-mark variables can be done simply by revising `dict-merge` and making a small change to the `tensor` procedure in `ps5match.scm`[2]

[2]Repeated tilde-variables can be handled this way too, with one more subtle change. In the `match-initial-tildevar` procedure, a pattern with an initial tilde-variable is used recursively to match a subdatum, and then the dictionary value of the tilde-variable is revised to reflect a match of the pattern against the whole datum. If the pattern happened to contain additional, non-initial, occurrences of the initial tilde-variable, then in the match against the whole datum, the initial occurrence matches the revised value, but the non-initial occurrences match the unrevised value. For this reason the match against the whole datum should actually fail.

To repair this, a "fresh" initial tilde-variable (one which does not occur elsewhere in the pattern) should have been used in the recursive calls of `match-initial-tildevar`. Since generating "fresh" variables involves some bletcherous (see *The Hacker's Dictionary*) string/symbol hacking, and since none of our examples require repeated tilde-variables, we are not asking you to make this modification.

- Modify the `dict-merge` procedure to handle repeated variables. That is, a match will fail unless all the occurrences of a repeated variable match `equal?` data items.
- Then modify the `tensor` so it filters out the failed dictionaries.
- Run some test examples to verify your modifications.

2.4 Instantiation

Simple rules and restricted rules operate by matching the pattern and then instantiating the template. The `instantiate` procedure can be written as a simple tree recursion:

```
(define (instantiate temp dict)
  (cond ((or (and (qmark-variable? temp)
                  (not (qmark? temp)))
             (and (tilde-variable? temp)
                  (not (tilde? temp))))
         (lookup temp dict))
        ((or (is-atom? temp) (null? temp)
                ;when ? or ~ is in a template, treat it as an atom:
             (qmark? temp) (tilde? temp))
         temp)
        (else
         ((if (tilde-variable? (car temp))
              append
              cons)
            (instantiate (car temp) dict)
            (instantiate (cdr temp) dict)))))
```

2.5 More General Pattern Matching Rules

We also want the behavior of Scheme primitives to be expressible in a rewriting rule system. For example, to capture integer addition with rewriting, we want some rules whose effect would be the same having the full, infinite set of addition rules:

```
(make-simple-rule '(0 + 0) 0)
(make-simple-rule '(0 + 1) 1)
(make-simple-rule '(1 + 0) 1)
(make-simple-rule '(1 + 1) 2)
```

```
(make-simple-rule '(0 + 2) 2)
(make-simple-rule '(2 + 0) 2)
(make-simple-rule '(1 + 2) 3)
         .
         .
         .
```

We accomplish this using *general-rules*, which have a pattern and an arbitrary Scheme "result" procedure operating on dictionaries. This procedure determines whether the match is allowed, and if it is, returns a result.

Now a single such general rule can capture addition of any two numbers Scheme numbers:

```
(make-general-rule   ;NUMBER ADDITION RULE
 '(?%num1 + ?%num2)
 (lambda (dict)       ;RESULT PROCEDURE: DICT --> DATUM
      (+ (lookup '?%num1 dict)
         (lookup '?%num2 dict)))))
```

General-rules allow a rewriting rule system to "hook" into the full power of underlying Scheme. There is no way to capture the behavior of Scheme's addition primitive with simple or restricted rules, so the general rule in this case provides a simple and necessary mechanism. But it is not hard to see how *any* procedure mapping data to lists can be formulated as a general-rule. If general rules are not used sparingly and with good taste, the attractions of rewriting-style programming can quickly be undermined.

Lab Exercise 5. Extend Eliza with one or more general rules which changes first person (I, mine, etc.) to second person (you, yours, etc.), and vice versa, everywhere in its responses. Show some dialogs in which your rule is invoked.

3 Which Rule Where?

Of course the rules given above determine only some of Eliza's behavior. There are many possible responses Eliza could make to an input, because

- there may be many ways (dictionaries) to match the pattern of a particular rule to the input datum, each yielding a different instance of the rule template, or
- there may be many rules whose patterns match the input.

An important aspect of a rewriting-rule system is the procedure for selecting a possible response among many—and there do tend to be very many—far more than can feasibly be examined exhaustively.

The procedure `some-rule-application` below applies a rule using the first match it finds whose dictionary successfully matches the datum.

```
         ;Do SOME APPLICATION of the rule to the ENTIRE DATUM
         ;(We choose the first dictionary of the match.)
(define (some-rule-application rule datum)
  (let ((dicts (match (rule.pattern rule) datum)))
    (if (null? dicts)
        (fail)
        (cond
         ((simple? rule)
          (instantiate (rule.template rule) (car dicts)))

         ((restricted? rule)
          (let ((dict-ok? (rule.restriction rule)))
            (define (search maybe-ok-dicts)
              (if (null? maybe-ok-dicts)
                  (fail)
                  (let ((dict (car maybe-ok-dicts)))
                    (if (dict-ok? dict)
                        (instantiate (rule.template rule) dict)
                        (search (cdr maybe-ok-dicts))))))
            (search dicts)))

         ((general? rule)
          (let ((proc (rule.result-procedure rule)))
            (define (search maybe-ok-dicts)
              (if (null? maybe-ok-dicts)
                  (fail)
                  (let* ((dict (car maybe-ok-dicts))
                         (result (proc dict)))
                    (if (fail? result)
                        (search (cdr maybe-ok-dicts))
                        result))))
            (search dicts)))

         (else (error "unknown rule" rule))))))
```

The procedure `some-rules-application` does some application of some rule in a list to the entire datum:

```
         ;Do some application of SOME RULE to the ENTIRE DATUM
```

```
        ;(We choose the first dictionary of the match of
        ; the first rule that has one.)
(define (some-rules-application rules datum)
  (if (null? rules)
      (fail)
      (let ((result (some-rule-application (car rules) datum)))
        (if (fail? result)
            (some-rules-application (cdr rules) datum)
            result))))
```

Lab Exercise 6.

Modify `some-rules-application` to select patterns and rules in some new way. Compare the behavior of Eliza using your new selection procedure against the one above.

Lab Exercise 7.

Define procedures `all-rule-applications` and `all-rules-applications`. Try them using the Eliza rules.

4 Repeated Rule Application

In Eliza, the underlying system only needed to select a rule and a match. In the case of rules like `rule:negate-and-commute` on page 4 which are intended to be used repeatedly, there are two further kinds of control decisions to make because:

- a rule may be applicable to more than one *sub*expression,
- it may be advantageous to vary the choice of applicable rule depending on which rules were already applied. The choice may influence the resulting "final" form of the rewritten expression (see below).

For example, we can apply `rule:negate-and-commute` to the whole expression

```
((2 + (x - y)) + z)
```

to get

```
((0 - z) - (2 + (x - y)))
```

But we could also apply the rule to the lefthand *subexpression*

```
(2 + (x - y))
```

yielding the overall expression

```
(((0 - (x - y)) - 2) + z)
```

Rule-based programming systems such as PROLOG and various "expert systems" typically support some built-in search order for selecting or enumerating possible matches and successive rule applications. Unfortunately, there is no one correct or generally appropriate search order.

The procedure `some-leftmost-outermost-application` below implements one standard way to search for a subexpression where a rule will be applied.

```
(define (some-leftmost-outermost-application rules datum)
  (let ((result (some-rules-application rules datum)))
    (if (fail? result)
        (some-leftmost-subtrees-application rules datum)
        result)))

(define (some-leftmost-subtrees-application rules datum)
  (if (or (is-atom? datum) (null? datum))
      (fail)
      (let ((dat1     (car datum))
            (dat-rest (cdr datum)))
        (let ((dat1-result
               (some-leftmost-outermost-application
                rules
                dat1)))
          (if (fail? dat1-result)
              (let ((dat-rest-result
                     (some-leftmost-subtrees-application rules (cdr datum))))
                (if (fail? dat-rest-result)
                    (fail)
                    (cons dat1 dat-rest-result)))
              (cons dat1-result dat-rest))))))
```

For such "simplification" rules, the idea is to start with some expression and *repeatedly* simplify subexpressions until an expression is obtained in which no rule is applicable *anywhere*. Such an expression is called a *final*, or *normal*, form of the starting expression. The procedure `some-leftmost-outermost-final-form` below implements one standard way to choose successive rules and places to apply them.

```
  (define (some-leftmost-outermost-final-form rules datum)
    (let ((result (some-leftmost-outermost-application rules datum)))
      (if (fail? result)
          datum
          (some-leftmost-outermost-final-form rules result))))

(define do-rules some-leftmost-outermost-final-form)
```

4.1 Sample Simplifications

Recursive procedures for symbolic differentiation of arithmetic expressions are described in *SICP* Section 2.3.2, but the most familiar way of describing differentiation is by rules like "the sum of the derivatives is the derivative of the sum":

```
(make-simple-rule
 '(deriv (?F + ?G) ?X)
 '((deriv ?F ?X) + (deriv ?G ?X)))
```

The file `ps5rules.scm` contains the remaining rules for symbolic derivatives:

```
;(define test-deriv '(deriv ((a + b) * (2 + (a * a))) a))

;(do-rules deriv-rules test-deriv)
;Value: (((a + b) * (0 + ((a * 1) + (a * 1))))
         + ((2 + (a * a)) * (1 + 0)))
```

To simplify this last result,

```
;(do-rules (append deriv-rules simplify-rules) test-deriv)
;Value: ((2 + ((((2 * a) * a) + (b * a)) + (b * a))) + (a * a))
```

Lab Exercise 8. Define an `all-final-forms` procedure (which returns a list of all final forms resulting from applying every rule in every order to every subdatum of a datum). See if you can find an arithmetic expression with more than one final form using the `simplify-rules` rules above.

The following more challenging exercises are optional:

Lab Exercise 9 (Optional). Define rules which, starting with any Scheme expression with operators + and *, and only the variable `x`, will simplify the expression to the standard final form for a polynomial, using ** for exponentiation. For example,

```
(* (+ (* x 2 x) (* x x 3) 1)
   (+ x (* x 5 x x)))
```

would simplify to

```
(+ (* 25 (x ** 5)) (* 10 (x ** 3)) x)
```

You may do this by adding simplification rules to those in `ps5rules.scm` or by inventing your own set of rules. You may also replace `some-leftmost-outermost-application` with a variant such as `some-leftmost-INNERMOST-application` which you define.

Lab Exercise 10 (Optional).

The file `ps5rules.scm` contains rules for desugaring `quote`d lists of atoms and symbols into expressions in which `quote` appears only at symbols and the empty list. For example, `'(a (b) 1)` should have final form `(list 'a (list 'b) 1)`.

Extend these rules to simplify quoted lists of atoms and symbols allowing `car` and `cdr` as well. For example, now

```
(cons (car (list (cdr '(1 2 3)))) 4)
```

should simplify to

```
(cons (list 2 3) 4)
```

(This exercise is harder than it may appear: note that `(car (list 1 (cdr '())))` must not "simplify" to 1; it should either itself be considered a final form or have `error` as a final form.)

Turn in this sheet along with your answers to the questions in the problem set:

How much time did you spend on this homework assignment?

Indicate all of the following facilities you in doing this problem set:

- The 6.001 lab in building 36.
- MIT Scheme on Project Athena.
- MIT Scheme on a Windows PC.
- EdScheme on a PC.
- Some other Scheme system (which one?).

We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).

If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the course text (such as some of the course archives), please indicate your consultants' names and your references in the space below. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.

Otherwise, write "I worked alone using only the course reference materials." and sign your statement.