6.945 Adventures in Advanced Symbolic Programming

Spring 2009

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945 Spring 2008
Problem Set 4

Issued: Wed. 27 Feb. 2008 **Due: Wed. 5 Mar. 2008**

Reading: MIT Scheme Reference Manual, section 2.11: Macros
This is complicated stuff, so don't try to read it until you need to in the compilation part of the problem set.

Code: load.scm, rule-compiler.scm, matcher.scm, rule-simplifier.scm, rules.scm, all attached.

Pattern Matching and Instantiation, continued

In this problem set we extend our pattern matching system to build a primitive algebraic simplifier, based on pattern matching and instantiation.

In rules.scm there are two elementary rule systems. A rule has three parts: a pattern to match a subexpression, a predicate expression that must be true for the rule to be applicable, and a skeleton to be instantiated and replace the matched subexpression.

The rules are assembled into a list and handed to the rule-simplifier procedure. The result is a simplifier procedure that can be applied to an algebraic expression.

The first rule system demonstrates only elementary features. It does not use segment variables or restricted variables. The first system has three rules: The first rule implements the associative law of addition, the second implements the commutative law of multiplication, and the third implements the distributive law of multiplication over addition.

The commutative law looks like:

```
(rule (* (? b) (? a))
      (expr<? a b)
      (* (? a) (? b)))
```

Notice the rule-restriction predicate expression in the rule for the commutative law. The restriction predicate expr<? imposes an ordering on algebraic expressions.

Problem 4.1:

Why is the (expr<? a b) restriction necessary in the commutative law? What would go wrong if the there was no restriction? (Indicated by the symbol "none" in the restriction slot of the rule.)

The second system of rules is far more interesting. It is built with the assumption that addition and multiplication are n-ary operations: it needs segment variables to make this work. It also uses variable restrictions to allow rules for simplifying numerical terms and prefactors.

Problem 4.2:

In the second system how does the use of the ordering on expressions imposed on the commutative laws make the numerical simplification rules effective?

Suppose that the commutative laws did not force an ordering, how would we have to write the numerical simplification rules? Explain why numerical simplification would become very expensive.

Problem 4.3:

The ordering in the commutative laws evolves an n^2 bubble sort on the terms of a sum and the factors of a product. This can get pretty bad if there are many terms, as in a serious algebra problem. Is there some way in this system to make a more efficient sort? If not, why not? If so, how would you arrange it?

Problem 4.4:

The system we have described does not collect like terms. For example:

```
(algebra-2 '(+ (* 4 x) (* 3 x)))
;Value (+ (* 3 x) (* 4 x))
```

Add rules that cause the collection of like terms, leaving the result as a sum of terms. Demonstrate your solution. Your solution must be able to handle problems like:

```
(algebra-3
  '(+ y (* x -2 w) (* x 4 y) (* w x) z (* 5 z) (* x w) (* x y 3)))
;Value: (+ y (* 6 z) (* 7 x y))
```

Now that we have some experience with the use of such a rule system, let's dive in to see how it works.

The center of the simplifier is in the file rule-simplifier.scm. It is composed of three parts. The first one, rule-simplifier, is a simple recursive simplifier constructor. It produces a procedure, simplify-expression, that takes an expression and uses the rules to simplify the expression. It recursively simplifies all the subexpressions of an expression, and then applies the rules to simplify the resulting expression. It does this repeatedly until the process converges and the expression returned is a fixed point of the simplification process.

```
(define (rule-simplifier the-rules)
  (define (simplify-expression expression)
    (let ((ssubs
           (if (list? expression)
               (map simplify-expression expression)
               expression)))
      (let ((result (try-rules ssubs the-rules)))
        (if result
            (simplify-expression result)
            ssubs))))
  (rule-memoize simplify-expression))
```

The procedure rule-memoize may be thought of as an identity function:

```
(define (rule-memoize f) f)      ; CAVEAT:  Defined in "load.scm"
```

but we can change this to be a memoizer that can greatly reduce the computational complexity of the process.

A rule returns #f if it cannot be applied to an expression. The procedure try-rules just scans the list of rules, returning the result of the first rule that applies, or the #f if no rule applies:

```
(define (try-rules expression the-rules)
  (define (scan rules)
    (if (null? rules)
        #f
        (or ((car rules) expression)
            (scan (cdr rules)))))
  (scan the-rules))
```

A rule is made from a matcher combinator procedure, a restriction predicate procedure and an instantiator procedure. These are put together by the procedure rule:make. If the matcher succeeds it produces a dictionary and a count of items eaten by the matcher from the list containing the expression to match. If this number is 1 then if either there is no restriction predicate or the restriction predicate is satisfied then the instantiator is called. The restriction predicate procedure and the instantiator procedure take as arguments the values of the variables bound in the match. These procedures are compiled from the rule patterns and skeletons, by a process we will see later.

```
(define (rule:make matcher restriction instantiator)
  (define (the-rule expression)
    (matcher (list expression)
             '()
             (lambda (dictionary n)
               (and (= n 1)
                    (let ((args (map match:value dictionary)))
                      (and (or (not restriction)
                               (apply restriction args))
                           (apply instantiator args)))))))
    the-rule)
```

Problem 4.5:

Problem 3.27 in SICP (pp. 272--273) shows the basic idea of memoization. Can this help the simplifier? How?

Write a memoizer that may be useful in dealing with expressions in the rule-simplifier procedure. One problem is that we don't want to store a table with an unbounded number of big expression. However, most of the advantage in this kind of memoizer comes from using it as a cache for recent results. Implement an LRU memoizer mechanism that stores only a limited number of entries and throws away the Least-Recently Used one. Demonstrate your program.

Magic Macrology

The call to rule:make is composed from the expression representing a rule by the rule compiler, found in rule-compiler.scm. There are complications here, imposed by the use of a macro implementing syntax for rules. Consider the rule:

```
(rule (* (?? a) (? y) (? x) (?? b))
      (expr<? x y)
      (* (?? a) (? x) (? y) (?? b)))
```

The rule macro turns this into a call to compile-rule:

```
(compile-rule '(* (?? a) (? y) (? x) (?? b))
              '(expr<? x y)
              '(* (?? a) (? x) (? y) (?? b))
              some-environment)
```

We can see the expression this expands into with the following magic incantation:

```
(pp (syntax
     (compile-rule '(* (?? a) (? y) (? x) (?? b))
                   '(expr<? x y)
                   '(* (?? a) (? x) (? y) (?? b))
                   (the-environment))
     (the-environment)))
==>
(rule:make
 (match:list (match:eqv (quote *))
             (match:segment (quote a))
             (match:element (quote y))
             (match:element (quote x))
             (match:segment (quote b)))
 (lambda (b x y a)
   (expr<? x y))
 (lambda (b x y a)
   (cons (quote *) (append a (cons x (cons y b))))))
```

We see that the rule expands into a call to rule:make with arguments that construct the matcher combinator, the predicate procedure, and the instantiation procedure. This is the expression that is evaluated to make the rule. In more conventional languages macros expand directly into code that is substituted for the macro call. However this process is not referentially transparent, because the macro expansion may use symbols that conflict with the user's symbols. In Scheme we try to avoid this problem, allowing a user to write "hygienic macros" that cannot cause conflicts. However this is a bit more complicated than just substituting one expression for another. We will not try to explain the problems or the solutions here, but we will just use the solutions described in the MIT Scheme reference manual, section 2.11.

```
;;;; File:  load.scm -- Loader for rule system

(load "rule-compiler")
(load "matcher")
(load "rule-simplifier")

(define (rule-memoize x) x)   ;;; NB:  Scaffolding stub for prob 4.5

(load "rules")
```

```
;;;; File:  rule-compiler.scm

(define-syntax rule
  (sc-macro-transformer
   (lambda (form env)
     (if (syntax-match? '(DATUM EXPRESSION DATUM) (cdr form))
         (compile-rule (cadr form) (caddr form) (cadddr form) env)
         (ill-formed-syntax form)))))

(define (compile-rule pattern restriction template env)
  (let ((names (pattern-names pattern)))
    `(rule:make ,(compile-pattern pattern env)
                ,(compile-restriction restriction env names)
                ,(compile-instantiator template env names))))
```

```
(define (pattern-names pattern)
  (let loop ((pattern pattern) (names '()))
    (cond ((or (match:element? pattern)
               (match:segment? pattern))
           (let ((name (match:variable-name pattern)))
             (if (memq name names)
                 names
                 (cons name names))))
          ((list? pattern)
           (let elt-loop ((elts pattern) (names names))
             (if (pair? elts)
                 (elt-loop (cdr elts) (loop (car elts) names))
                 names)))
          (else names))))

(define (compile-pattern pattern env)
  (let loop ((pattern pattern))
    (cond ((match:element? pattern)
           (if (match:restricted? pattern)
               `(match:element ',(match:variable-name pattern)
                               ,(match:restriction pattern))
               `(match:element ',(match:variable-name pattern))))
          ((match:segment? pattern)
           `(match:segment ',(match:variable-name pattern)))
          ((null? pattern)
           `(match:eqv '()))
          ((list? pattern)
           `(match:list ,@(map loop pattern)))
          (else
           `(match:eqv ',pattern)))))


(define (match:element? pattern)
  (and (pair? pattern)
       (eq? (car pattern) '?)))

(define (match:segment? pattern)
  (and (pair? pattern)
       (eq? (car pattern) '??)))

(define (match:variable-name pattern)
  (cadr pattern))


;;; These restrictions are for variable elements.

(define (match:restricted? pattern)
  (not (null? (cddr pattern))))

(define (match:restriction pattern)
  (caddr pattern))
```

```
;;; The restriction is a predicate that must be true for the rule to
;;; be applicable.  This is not the same as a variable element
;;; restriction.

(define (compile-restriction expr env names)
  (if (eq? expr 'none)
      '#f
      (make-lambda names env
        (lambda (env)
          (close-syntax expr env)))))


(define (compile-instantiator skel env names)
  (make-lambda names env
    (lambda (env)
      (list 'quasiquote
            (let ((wrap (lambda (expr) (close-syntax expr env))))
              (let loop ((skel skel))
                (cond ((skel:element? skel)
                       (list 'unquote
                             (wrap (skel:element-expression skel))))
                      ((skel:segment? skel)
                       (list 'unquote-splicing
                             (wrap (skel:segment-expression skel))))
                      ((list? skel) (map loop skel))
                      (else skel)))))))))


(define (skel:constant? skeleton)
  (not (pair? skeleton)))


(define (skel:element? skeleton)
  (and (pair? skeleton)
       (eq? (car skeleton) '?)))

(define (skel:element-expression skeleton)
  (cadr skeleton))


(define (skel:segment? skeleton)
  (and (pair? skeleton)
       (eq? (car skeleton) '??)))

(define (skel:segment-expression skeleton)
  (cadr skeleton))
```

```
;; Magic!
(define (make-lambda bvl use-env generate-body)
  (capture-syntactic-environment
   (lambda (transform-env)
     (close-syntax `(,(close-syntax 'lambda transform-env)
                     ,bvl
                     ,(capture-syntactic-environment
                       (lambda (use-env*)
                         (close-syntax (generate-body use-env*)
                                       transform-env))))
                   use-env))))

#|
;;; For example

(pp (syntax '(rule (+ (? a) (+ (? b) (? c)))
                   none
                   (+ (+ (? a) (? b)) (? c)) )
            (the-environment)))
(rule:make
 (match:list
  (match:eqv (quote +))
  (match:element (quote a))
  (match:list (match:eqv (quote +))
              (match:element (quote b))
              (match:element (quote c))))
 #f
 (lambda (c b a)
   (list (quote +) (list (quote +) a b) c)))

(pp (syntax '(rule (+ (? a) (+ (? b) (? c)))
                   (> a 3)
                   (+ (+ (? a) (? b)) (? c)) )
            (the-environment)))
(rule:make
 (match:list
  (match:eqv (quote +))
  (match:element (quote a))
  (match:list (match:eqv (quote +))
              (match:element (quote b))
              (match:element (quote c))))
 (lambda (c b a)
   (> a 3))
 (lambda (c b a)
   (list (quote +) (list (quote +) a b) c)))

|#
```

```
;;;; File:  matcher.scm

;;;; Matcher based on match combinators, CPH/GJS style.
;;;     Idea is in Hewitt's PhD thesis (1969).

(declare (usual-integrations))

;;; There are match procedures that can be applied to data items.  A
;;; match procedure either accepts or rejects the data it is applied
;;; to.  Match procedures can be combined to apply to compound data
;;; items.

;;; A match procedure takes a list containing a data item, a
;;; dictionary, and a success continuation.  The dictionary
;;; accumulates the assignments of match variables to values found in
;;; the data.  The success continuation takes two arguments: the new
;;; dictionary, and the number of items absorbed from the list by the
;;; match.  If a match procedure fails it returns #f.

;;; Primitive match procedures:

(define (match:eqv pattern-constant)
  (define (eqv-match data dictionary succeed)
    (and (pair? data)
         (eqv? (car data) pattern-constant)
         (succeed dictionary 1)))
  eqv-match)


;;; Here we have added an optional restriction argument to allow
;;; conditional matches.

(define (match:element variable #!optional restriction?)
  (if (default-object? restriction?)
      (set! restriction? (lambda (x) #t)))
  (define (element-match data dictionary succeed)
    (and (pair? data)
         ;; NB:  might be many distinct restrictions
         (restriction? (car data))
         (let ((vcell (match:lookup variable dictionary)))
           (if vcell
               (and (equal? (match:value vcell) (car data))
                    (succeed dictionary 1))
               (succeed (match:bind variable (car data) dictionary)
                        1)))))
  element-match)
```

```
(define (match:segment variable)
  (define (segment-match data dictionary succeed)
    (and (list? data)
         (let ((vcell (match:lookup variable dictionary)))
           (if vcell
               (let lp ((data data)
                        (pattern (match:value vcell))
                        (n 0))
                 (cond ((pair? pattern)
                        (if (and (pair? data)
                                 (equal? (car data) (car pattern)))
                            (lp (cdr data) (cdr pattern) (+ n 1))
                            #f))
                       ((not (null? pattern)) #f)
                       (else (succeed dictionary n))))
               (let ((n (length data)))
                 (let lp ((i 0))
                   (if (<= i n)
                       (or (succeed (match:bind variable
                                                (list-head data i)
                                                dictionary)
                                    i)
                           (lp (+ i 1)))
                       #f)))))))
  segment-match)

(define (match:list . match-combinators)
  (define (list-match data dictionary succeed)
    (and (pair? data)
         (let lp ((data (car data))
                  (matchers match-combinators)
                  (dictionary dictionary))
           (cond ((pair? matchers)
                  ((car matchers) data dictionary
                      (lambda (new-dictionary n)
                        (if (> n (length data))
                            (error "Matcher ate too much." n))
                        (lp (list-tail data n)
                            (cdr matchers)
                            new-dictionary))))
                 ((pair? data) #f)
                 ((null? data)
                  (succeed dictionary 1))
                 (else #f)))))
  list-match)
```

```
;;; Syntax of matching is determined here.

(define (match:->combinators pattern)
  (define (compile pattern)
    (cond ((match:element? pattern)
           (if (match:restricted? pattern)
               (match:element (match:variable-name pattern)
                              (match:restriction   pattern))
               (match:element (match:variable-name pattern))))
          ((match:segment? pattern)
           (match:segment (match:variable-name pattern)))
          ((list? pattern)
           (apply match:list (map compile pattern)))
          (else (match:eqv pattern))))
  (compile pattern))

(define (match:element? pattern)
  (and (pair? pattern) (eq? (car pattern) '?)))

(define (match:segment? pattern)
  (and (pair? pattern) (eq? (car pattern) '??)))

(define (match:variable-name pattern)
  (cadr pattern))

;;; These restrictions are for variable elements.

(define (match:restricted? pattern)
  (not (null? (cddr pattern))))

(define (match:restriction pattern)
  (caddr pattern))

(define (matcher pattern)
  (let ((match-combinator (match:->combinators pattern)))
    (lambda (datum)
      (match-combinator (list datum) '()
       (lambda (dictionary number-of-items-eaten)
         (and (= number-of-items-eaten 1)
              dictionary))))))


;;; Support for the dictionary.

(define (match:bind variable data-object dictionary)
  (cons (list variable data-object) dictionary))

(define (match:lookup variable dictionary)
  (assq variable dictionary))

(define (match:value vcell)
  (cadr vcell))
```

```
#|
((match:->combinators '(a ((? b) 2 3) 1 c))
 '((a (1 2 3) 1 c))
 '()
  (lambda (x y) `(succeed ,x ,y)))
;Value: (succeed ((b 1)) 1)

((match:->combinators '(a ((? b) 2 3) (? b) c))
 '((a (1 2 3) 2 c))
 '()
  (lambda (x y) `(succeed ,x ,y)))
;Value: #f

((match:->combinators '(a ((? b) 2 3) (? b) c))
 '((a (1 2 3) 1 c))
 '()
  (lambda (x y) `(succeed ,x ,y)))
;Value: (succeed ((b 1)) 1)


((match:->combinators '(a (?? x) (?? y) (?? x) c))
 '((a b b b b b b c))
 '()
 (lambda (x y)
   (pp `(succeed ,x ,y))
   #f))
(succeed ((y (b b b b b b)) (x ())) 1)
(succeed ((y (b b b b)) (x (b))) 1)
(succeed ((y (b b)) (x (b b))) 1)
(succeed ((y ()) (x (b b b))) 1)
;Value: #f

((matcher '(a ((? b) 2 3) (? b) c))
 '(a (1 2 3) 1 c))
;Value: ((b 1))
|#
```

```
;;;; File:  rule-simplifier.scm

;;;;         Match and Substitution Language Interpreter

(declare (usual-integrations))

;;;   This is a descendent of the infamous 6.001 rule interpreter,
;;; originally written by GJS for a lecture in the faculty course held
;;; at MIT in the summer of 1983, and subsequently used and tweaked
;;; from time to time.  This subsystem has been a serious pain in the
;;; ass, because of its expressive limitations, but I have not had the
;;; guts to seriously improve it since its first appearance. -- GJS

;;; January 2006.  I have the guts now! The new matcher is based on
;;; combinators and is in matcher.scm.  -- GJS


(define (rule-simplifier the-rules)
  (define (simplify-expression expression)
    (let ((ssubs
           (if (list? expression)
               (map simplify-expression expression)
               expression)))
      (let ((result (try-rules ssubs the-rules)))
        (if result
            (simplify-expression result)
            ssubs))))
  (rule-memoize simplify-expression))

(define (try-rules expression the-rules)
  (define (scan rules)
    (if (null? rules)
        #f
        (or ((car rules) expression)
            (scan (cdr rules)))))
  (scan the-rules))



;;;;  Rule applicator, using combinator-based matcher.

(define (rule:make matcher restriction instantiator)
  (define (the-rule expression)
    (matcher (list expression)
             '()
             (lambda (dictionary n)
               (and (= n 1)
                    (let ((args (map match:value dictionary)))
                      (and (or (not restriction)
                               (apply restriction args))
                           (apply instantiator args)))))))
    the-rule)
```

```
;;; File:  rules.scm -- Some sample algebraic simplification rules

(define algebra-1
  (rule-simplifier
   (list

    ;; Associative law of addition
    (rule (+ (? a) (+ (? b) (? c)))
          none
          (+ (+ (? a) (? b)) (? c)))

    ;; Commutative law of multiplication
    (rule (* (? b) (? a))
          (expr<? a b)
          (* (? a) (? b)))

    ;; Distributive law of multiplication over addition
    (rule (* (? a) (+ (? b) (? c)))
          none
          (+ (* (? a) (? b)) (* (? a) (? c))))

    )))

(define (expr<? x y)
  (cond ((null? x)
         (if (null? y) #f #t))
        ((null? y) #f)
        ((number? x)
         (if (number? y) (< x y) #t))
        ((number? y) #f)
        ((symbol? x)
         (if (symbol? y) (symbol<? x y) #t))
        ((symbol? y) #f)
        ((list? x)
         (if (list? y)
             (let ((nx (length x)) (ny (length y)))
               (cond ((< nx ny) #t)
                     ((> nx ny) #f)
                     (else
                      (let lp ((x x) (y y))
                        (cond ((null? x) #f) ; same
                              ((expr<? (car x) (car y)) #t)
                              ((expr<? (car y) (car x)) #f)
                              (else (lp (cdr x) (cdr y)))))))))
        ((list? y) #f)
        (else
         (error "Unknown expression type -- expr<?"
                x y))))

#|
(algebra-1 '(* (+ y (+ z w)) x))
;Value: (+ (+ (* x y) (* x z)) (* w x))
|#
```

```
(define algebra-2
  (rule-simplifier
   (list

    ;; Sums

    (rule (+ (? a)) none (? a))

    (rule (+ (?? a) (+ (?? b)))
          none
          (+ (?? a) (?? b)))

    (rule (+ (+ (?? a)) (?? b))
          none
          (+ (?? a) (?? b)))

    (rule (+ (?? a) (? y) (? x) (?? b))
          (expr<? x y)
          (+ (?? a) (? x) (? y) (?? b)))


    ;; Products

    (rule (* (? a)) none (? a))

    (rule (* (?? a) (* (?? b)))
          none
          (* (?? a) (?? b)))

    (rule (* (* (?? a)) (?? b))
          none
          (* (?? a) (?? b)))

    (rule (* (?? a) (? y) (? x) (?? b))
          (expr<? x y)
          (* (?? a) (? x) (? y) (?? b)))


    ;; Distributive law

    (rule (* (? a) (+ (?? b)))
          none
          (+ (?? (map (lambda (x) `(* ,a ,x)) b))))
```

```
      ;; Numerical simplifications below

      (rule (+ 0 (?? x)) none (+ (?? x)))

      (rule (+ (? x number?) (? y number?) (?? z))
            none
            (+ (? (+ x y)) (?? z)))


      (rule (* 0 (?? x)) none 0)

      (rule (* 1 (?? x)) none (* (?? x)))

      (rule (* (? x number?) (? y number?) (?? z))
            none
            (* (? (* x y)) (?? z)))

      )))

  #|
  (algebra-2 '(* (+ y (+ z w)) x))
  ;Value: (+ (* w x) (* x y) (* x z))

  (algebra-2 '(+ (* 3 (+ x 1)) -3))
  ;Value: (* 3 x)
  |#
```