10/10 You are a superhero!

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 6.891 PS4
;;;
;;; Dan R. K. Ports <drkp@mit.edu>
;;;
;;; $Date: 2007-03-07 12:57:16 -0500 (Wed, 07 Mar 2007) $ $Rev: 2286 $
;;;
;;; Collaborators: '(iyzhang amdragon)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load "load.scm")

;;; Some helper functions

(define (assert x . msg)
  (if (not x) (apply error msg)))

(define (assert-equal a b)
  (assert (equal? a b)
          "Test failed"
          "Expected: "
          b
          " got: "
          a)
  #t)


;;;
;;; Problem 1
;;;
;;; Without this restriction, the rule system would happily apply the
;;; commutative law willy-nilly, replacing (* a b) with (* b a) and
;;; vice versa; it would never terminate.
;;;
;;; With the restriction, the rule system terminates. Here's a proof
;;; sketch. Define an ordering over algebraic expressions as follows:
;;;   A < B iff:
;;;     - the number of addition subexpressions in A that contain an
;;;       addition subexpression that isn't the first element is less
;;;       than the corresponding number of subexpressions in A, or
;;;       they're equal and
;;;     - the sum over all multiplication subexpressions in A of the
;;;       product of the number of addition subexpressions in each term
;;;       is less than the corresponding sum for B, or they're equal and
;;;     - (expr<? A B)
;;;
;;; Then it's easy to see that each application of one of the rules
;;; causes the new subexpression to be ordered strictly less than the
;;; previous one, because each application of one of the rules causes
;;; one of the comparisons to decrease without affecting the
;;; higher-precedence comparisons. The details of the proof are left
;;; as an exercise for the reader. (The interested reader. In other
;;; words, not me.)
```

7 Well done!! 2/2

"ping-ponging"

Woo Hoo!

```
;;;
;;; Problem 2
;;;
;;; The commutative laws ensure that the terms in a multiplication or
;;; addition subexpression are always sorted in expr<? order. The
;;; numerical simplification rules use this assumption by assuming
;;; that numeric terms appear next to each other at the beginning of a
;;; multiplication, not scattered about. The rule
;;;   (* (? x number?) (? y number?) (?? z)) would not match, for
;;; example (* y 3 x 2).
;;;
;;; If we didn't have the ordering, we'd have to  write something like
;;;    (rule (* (?? a) (? x number?) (?? b) (? y number?) (?? c))
;;;       none
;;;       (* (? (* x y)) (?? a) (?? b) (?? c)))
;;; This would work, but would be much less efficient, since it
;;; involves matching multiple segment variables, which requires
;;; (potentially quite costly) backtracking in the matcher.
;;;
```

7 2/2 Good!

"canonicalization"

Good

```
;;;
;;; Problem 3
;;;
;;; Sure. There's no reason that the sort has to be a bubble
;;; sort. When we detect two elements out of order in the terms of a
;;; sum, we can sort the entire contents of the sum using a more
;;; efficient algorithm, as shown in the rule that follows:
;;;

(define sorter
  (rule-simplifier
   (list
    (rule (+ (?? a) (? y) (? x) (?? b))
          (expr<? x y)
          (+ (?? (sort (cons x (cons y (append a b))) expr<?)))))))

(assert-equal
 (sorter '(+ 7 5 2 9 6 8 1 3 0 4))
 '(+ 0 1 2 3 4 5 6 7 8 9))

(assert-equal
 (sorter '(+ 7 5 2 9 (* 2 x) (* x y)))
 '(+ 2 5 7 9 (* 2 x) (* x y)))

;;;
;;; (Of course, we can do the same for products too.)
;;;
;;; But this feels like cheating. Why not just implement merge sort in
;;; the rule system? It'd look something like this:
;;;

(define another-sorter
  (rule-simplifier
   (list
    (rule (sort (? a))
          none
          (sorted (? a)))
    (rule (sort (? a) (? b) (?? c))
          none
          (split () () (? a) (? b) (?? c)))
    (rule (split (? a) (? b) (? c))
          none
          (merge () (sort (? c) (?? a)) (sort (?? b))))
    (rule (split (? a) (? b) (? c) (?? d))
          none
          (split (? b) ((? c) (?? a)) (?? d)))
    (rule (merge (? r) (sorted (? a) (?? x)) (sorted (? b) (?? y)))
          (expr<? a b)
          (merge ((?? r) (? a)) (sorted (?? x)) (sorted (? b) (?? y))))
    (rule (merge (? r) (sorted (? a) (?? x)) (sorted (? b) (?? y)))
          (not (expr<? a b))
          (merge ((?? r) (? b)) (sorted (? a) (?? x)) (sorted (?? y))))
    (rule (merge (? r) (sorted (?? x)) (sorted))
          none
          (sorted (?? r) (?? x)))
    (rule (merge (? r) (sorted) (sorted (?? y)))
          none
          (sorted (?? r) (?? y)))
    )))

(assert-equal
 (another-sorter '(sort 7 5 2 9 6 8 1 3 0 4))
 (sorted 0 1 2 3 4 5 6 7 8 9))

(assert-equal
 (another-sorter '(sort 7 5 2 9 (* 2 x) (* x y)))
 '(sorted 2 5 7 9 (* 2 x) (* x y)))


;;;
;;; Of course, this is probably much less efficient than just escaping
;;; to Scheme and sorting, given that we have so many new rules that
;;; need to be applied, and adding all these new rules creates new
;;; matchers that need to test the expression at each stage. Actually,
;;; despite being O(n lg n), I wouldn't be too surprised if it were
;;; slower than the bubble sort for most reasonable cases. But isn't
;;; that a small price to pay for theoretical efficiency and
;;; ideological purity?
;;;
```

7 2/2 EXTRAORDINARILY WELL DONE!

YIKES!

YOU ARE A SUPER HERO!

EXACTLY! ☺

P.O.E.!

```
;;;
;;; Problem 4
;;;
;;; We'll add a few new rules for collecting like
;;; terms. Unfortunately, this seems to require four separate rules to
;;; handle a bunch of different cases: besides handling the obvious
;;; case of (+ (* 2 x) (* 3 x)) -> (+ (* 5 x)), we need to handle the
;;; cases in which one or both of the terms is actually a single
;;; variable (like 'x'). Also, we need to handle the case in which we
;;; have terms like (* x y) and (* 2 x y).
;;;

(define algebra-3
  (rule-simplifier
   (list

    ;; New rules for collection of like terms

    (rule (+ (?? a) (? x) (?? b) (? x) (?? c))
          none
          (+ (?? a) (?? b) (?? c) (* 2 (? x))))

    (rule (+ (?? a) (? v) (?? b) (* (? x) (? v)) (?? c))
          (number? x)
          (+ (?? a) (* (? (+ 1 x)) (? v)) (?? b) (?? c)))

    (rule (+ (?? a) (* (?? v)) (?? b) (* (? x) (?? v)) (?? c))
          (number? x)
          (+ (?? a) (* (? (+ 1 x)) (?? v)) (?? b) (?? c)))

    (rule (+ (?? a) (* (? x) (?? v)) (?? b) (* (? y) (?? v)) (?? c))
          (and (number? x) (number? y))
          (+ (?? a) (* (? (+ x y)) (?? v)) (?? b) (?? c)))

    ;; The rest of algebra-2 follows:
    ;; 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)))

    )))

(assert-equal
 (algebra-3 '(+ x x x))
 '(* 3 x))

(assert-equal
 (algebra-3
  '(+ y (* x -2 w) (* x 4 y) (* w x) z (* 5 z) (* x w) (* x y 3)))
 '(+ y (* 6 z) (* 7 x y)))

(assert-equal
 (algebra-3
  '(+ y (* x -2 w) (* x 4 y) (* w x) z (* 5 z) (* x w) (* x y 3)
     (* x y)))
 '(+ y (* 6 z) (* 8 x y)))
```

7 2/2 WELL DONE!

Good!

) IF YOU SORT FIRST, YOU WOULDN'T NEED (??b), RIGHT?

IT'S GENERALLY BETTER TO TEST SIMPLE LOCAL SYNTACTIC RESTRICTS LIKE THESE EARLY, LIKE A VARIABLE RESTRICTS IN THE MATCH PATTERNS... BUT OK.

```
;;;
;;; Problem 5
;;;
;;; Memoizing is useful for the rule simplifier not just because we
;;; can memoize the results of simplification in case we later
;;; apply the same rules to the same expression again, but also for
;;; memoizing subexpressions during the process of simplification. If
;;; a subexpression occurs repeatedly --- for example (* 2 x), or just
;;; 'x' or '3' --- we won't need to repeatedly try all of the rule
;;; matchers against it, which could be quite costly if there are a
;;; lot of rules.
;;;
;;; To implement the memoizer, we'll start by building a queue data
;;; type that we'll later use to handle expiration in the LRU
;;; cache. It supports the ability to append to the end of the queue,
;;; pop from the front, or remove an element from the middle, all in
;;; O(1) time. It's implemented essentially as a doubly-linked list.
;;;

;; Internal representation
(define-record-type :queue
  (queue head tail)
  queue?
  (head queue-head set-queue-head!)
  (tail queue-tail set-queue-tail!))

(define-record-type :queue-item
  (queue-item data prev)
  queue-item?
  (data queue-item-data set-queue-item-data!)
  (prev queue-item-prev set-queue-item-prev!))

;; Public interface
(define (make-queue)
  (queue '() '()))

(define (queue-length q)
  (length (queue-head q)))

(define (append-to-queue! q data)
  (let* ((old-tail (queue-tail q))
         (item (queue-item data old-tail))
         (item-list (list item)))
    (set-queue-tail! q item-list)
    (append! old-tail item-list)
    (if (null? (queue-head q))
        (set-queue-head! q item-list))
    item-list))

(define (pop-from-queue! q)
  (if (null? (queue-head q))
      #f
      (let* ((old-head (queue-head q))
             (data (queue-item-data (car old-head)))
             (new-head (cdr old-head)))
        (set-queue-head! q new-head)
        (if (null? new-head)
            (set-queue-tail! q '())
            (set-queue-item-prev! (car new-head) '()))
        data)))

(define (delete-from-queue! q item)
  (let ((prev (queue-item-prev (car item)))
        (next (cdr item)))
    (if (null? prev)
        (set-queue-head! q next)
        (set-cdr! prev next))
    (if (null? next)
        (set-queue-tail! q prev)
        (set-queue-item-prev! (car next) prev))))

(define (queue-data q)
  (let lp ((head (queue-head q)))
    (if (null? head)
        '()
        (cons (queue-item-data (car head))
              (lp (cdr head))))))

;; Test cases
(let ((q (make-queue)))
  (define i1 (append-to-queue! q 1))
  (assert-equal (queue-data q) '(1))
  (define i2 (append-to-queue! q 2))
  (assert-equal (queue-data q) '(1 2))
  (define i3 (append-to-queue! q 3))
  (assert-equal (queue-data q) '(1 2 3))
  (define i4 (append-to-queue! q 4))
  (assert-equal (queue-data q) '(1 2 3 4))
  (define i5 (append-to-queue! q 5))
  (assert-equal (queue-data q) '(1 2 3 4 5))
  (assert-equal (pop-from-queue! q) 1)
  (assert-equal (queue-data q) '(2 3 4 5))
  (assert-equal (pop-from-queue! q) 2)
  (assert-equal (queue-data q) '(3 4 5))
  (delete-from-queue! q i4)
  (assert-equal (queue-data q) '(3 5))
  (delete-from-queue! q i3)
  (assert-equal (queue-data q) '(5))
  (define i6 (append-to-queue! q 6))
  (assert-equal (queue-data q) '(5 6))
  (delete-from-queue! q i6)
  (assert-equal (queue-data q) '(5))
  (delete-from-queue! q i5)
  (assert-equal (queue-data q) '()))

;;;
;;; And we'll combine that queue with a hash table to implement a LRU
;;; cache. It provides essentially a map interface, except that
;;; entries are automatically removed when it fills up. The queue
;;; is used to determine which entries to expire, and the hash table
;;; contains pointers to the queue elements indexed by their key.
;;;

;; Internal representation
(define-record-type :lru
  (%lru queue hash n)
  %lru?
  (queue %lru-queue %set-lru-queue!)
  (hash %lru-hash %hash-lru-queue!)
  (n %lru-size %set-lru-size!))

;; Public interface
(define (make-lru size)
  (%lru (make-queue) (make-equal-hash-table) size))

(define (lru-add lru k v)
  (let ((qitem (append-to-queue! (%lru-queue lru) (list k v))))
    (hash-table/put! (%lru-hash lru) k qitem)
    (if (> (queue-length (%lru-queue lru)) (%lru-size lru))
```

7 2/2 VERY WELL DONE!

Good

COULD ALSO MAINTAIN A count BUT THIS IS FINE TOO.

I THINK YOU MEAN datum, BUT OK.

Good!

NICE

Good

```
        (hash-table/remove! (%lru-hash lru)
                            (car (pop-from-queue! (%lru-queue lru)))))))

(define (lru-lookup lru k)
  (let ((qitem (hash-table/get (%lru-hash lru) k #f)))
    (if (not qitem)
        #f
        (begin
          (delete-from-queue! (%lru-queue lru) qitem)
          (hash-table/put! (%lru-hash lru) k
                           (append-to-queue! (%lru-queue lru)
                                             (queue-item-data (car qitem))))
          (queue-item-data (car qitem))))))

;; Test cases
(let ((l (make-lru 5)))
  (lru-add l 'a 1)
  (lru-add l 'b 2)
  (lru-add l 'c 3)
  (lru-add l 'd 4)
  (lru-add l 'e 5)
  (assert-equal (lru-lookup l 'c) '(c 3))
  (assert-equal (lru-lookup l 'e) '(e 5))
  (assert-equal (lru-lookup l 'b) '(b 2))
  (assert-equal (lru-lookup l 'c) '(c 3))
  (lru-add l 'e 6)
  (assert-equal (lru-lookup l 'a) #f)
  (lru-add l 'f 7)
  (assert-equal (lru-lookup l 'd) #f)
  (lru-add l 'g 8)
  (assert-equal (lru-lookup l 'e) #f)
  (assert-equal (lru-lookup l 'b) '(b 2))
  (assert-equal (lru-lookup l 'c) '(c 3)))

;;;
;;; Finally, we'll implement the memoizer using the LRU:
;;;

(define *lru-size* 1024)

(define (rule-memoize f)
  (let ((lru (make-lru *lru-size*)))
    (lambda (x)
      (let ((lru-val (lru-lookup lru x)))
        (if lru-val
            (begin
;;              (pp "Using memoized value from LRU")
              (cadr lru-val))
            (begin
;;              (pp "Executing and adding to LRU")
              (let ((val (f x)))
                (lru-add lru x val)
                val)))))))

;; Note that we need to modify the rule simplifier to ensure that the
;; memoized simplify-expression is called recursively, allowing us to
;; memoize the results of subexpression simplification.

(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))))
  (set! simplify-expression (rule-memoize simplify-expression))
  simplify-expression)

;;;
;;; Test cases. The tests shown below verify that the rule systems
;;; from before still work correctly. Further testing (not shown)
;;; using the commented out debugging statements in rule-simplifier
;;; shows that memoized values are actually used.
;;;

(define sorter
  (rule-simplifier
   (list

    (rule (+ (?? a) (? y) (? x) (?? b))
          (expr<? x y)
          (+ (?? (sort (cons x (cons y (append a b))) expr<?)))))))

(assert-equal
 (sorter '(+ 7 5 2 9 6 8 1 3 0 4))
 '(+ 0 1 2 3 4 5 6 7 8 9))

(assert-equal
 (sorter '(+ 7 5 2 9 (* 2 x) (* x y)))
 '(+ 2 5 7 9 (* 2 x) (* x y)))


(load "rules.scm")

(assert-equal
 (algebra-1 '(* (+ y (+ z w)) x))
 '(+ (+ (* x y) (* x z)) (* w x)))

(assert-equal
 (algebra-2 '(* (+ y (+ z w)) x))
 '(+ (* w x) (* x y) (* x z)))

(assert-equal
 (algebra-2 '(+ (* 3 (+ x 1)) -3))
 '(* 3 x)
```