SICP section 2.5.3

September 21, 2007 at 20:11 **Tags** SICP

The code for this section is in Scheme.

Arithmetic on polynomials

Here is the whole code written by the authors before the exercises commence, collected into the `polynomial` package:

```
(define (install-polynomial-package)
  (define (variable? x) (symbol? x))

  (define (same-variable? v1 v2)
    (and (variable? v1) (variable? v2) (eq? v1 v2)))

  (define (make-poly variable term-list)
    (cons variable term-list))

  (define (variable p) (car p))

  (define (term-list p) (cdr p))

  (define (adjoin-term term term-list)
    (if (=zero? (coeff term))
        term-list
        (cons term term-list)))

  (define (the-empty-termlist) '())
  (define (first-term term-list) (car term-list))
  (define (rest-terms term-list) (cdr term-list))
  (define (empty-termlist? term-list) (null? term-list))
  (define (make-term order coeff) (list order coeff))
  (define (order term) (car term))
  (define (coeff term) (cadr term))

  (define (add-terms L1 L2)
    (cond ((empty-termlist? L1) L2)
          ((empty-termlist? L2) L1)
          (else
           (let ((t1 (first-term L1)) (t2 (first-term L2)))
             (cond ((> (order t1) (order t2))
                    (adjoin-term
```

```
                  t1 (add-terms (rest-terms L1) L2)))
                ((< (order t1) (order t2))
                 (adjoin-term
                  t2 (add-terms L1 (rest-terms L2))))
                (else
                 (adjoin-term
                  (make-term (order t1)
                             (add (coeff t1) (coeff t2)))
                  (add-terms (rest-terms L1)
                             (rest-terms L2))))))))

  (define (mul-terms L1 L2)
    (if (empty-termlist? L1)
        (the-empty-termlist)
        (add-terms (mul-term-by-all-terms (first-term L1) L2)
                   (mul-terms (rest-terms L1) L2))))
  (define (mul-term-by-all-terms t1 L)
    (if (empty-termlist? L)
        (the-empty-termlist)
        (let ((t2 (first-term L)))
          (adjoin-term
           (make-term (+ (order t1) (order t2))
                      (mul (coeff t1) (coeff t2)))
           (mul-term-by-all-terms t1 (rest-terms L))))))

  (define (add-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (make-poly (variable p1)
                   (add-terms (term-list p1)
                              (term-list p2)))
        (error "Polys not in same var -- ADD-POLY"
               (list p1 p2))))
  (define (mul-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (make-poly (variable p1)
                   (mul-terms (term-list p1)
                              (term-list p2)))
        (error "Polys not in same var -- MUL-POLY"
               (list p1 p2))))

  ;; Interface to the outer world
  ;;
  (define (tag p) (attach-tag 'polynomial p))
  (put 'add '(polynomial polynomial)
    (lambda (p1 p2) (tag (add-poly p1 p2))))
  (put 'mul '(polynomial polynomial)
    (lambda (p1 p2) (tag (mul-poly p1 p2))))
  (put 'make 'polynomial
    (lambda (var terms) (tag (make-poly var terms))))
  'done)

(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms))
```

```
(load "generic_arithmetic_combining_types.scm")
```

And now I have all the generic functions `add`, `apply-generic`, etc.) available, and the polynomial package can be installed without affecting the other types. When something like this happens, you know it's good software design.

Exercise 2.87

```
;; This goes inside "install-polynomial-package":
  (define (zero-poly? p)
    (let ((terms (term-list p)))
      (if (empty-termlist? terms)
        #t
        (and (=zero? (coeff (first-term terms)))
             (zero-poly?
              (make-poly
                (variable p)
                (rest-terms terms)))))))

  (put '=zero? '(polynomial)
    (lambda (p) (zero-poly? p)))
```

The function `zero-poly?` is implemented in a clever recursive way. A polynomial is zero if either its terms list is empty (i.e. no terms), or its first term's coefficient is 0 and the rest of the polynomial is zero.

However, after writing this, I felt not so good with it. After all, it doesn't look nice to create the polynomial anew with the reduced terms list. Can't we just work on the terms list itself ?

So I've rewritten the function to look like this:

```
  (define (zero-poly? p)
    (define (zero-terms? terms)
      (or (empty-termlist? terms)
          (and  (=zero? (coeff (first-term terms)))
                (zero-terms? (rest-terms terms)))))

    (zero-terms? (term-list p)))
```

It looks cleaner to me, but that's a matter of taste.

Exercise 2.88

First, here is the negation for the scheme number package, to allow testing:

```
(put 'neg '(scheme-number)
  (lambda (x) (tag (- x))))
```

And the generic operation will be defined as:

```
(define (neg x) (apply-generic 'neg x))
```

When I came to adding negation to the polynomial package, I ran into an abstraction problem. Generally, the most natural way to negate a polynomial is to map over all its terms and negate each. But the representation of the terms list is abstracted away by `first-term` and `rest-terms`, so I can only walk the list in order. However, to create a new list, I must use `adjoin-term` function which adds a new term to the beginning of the list. Therefore I can create the terms list only in reverse. So, it appears there's not clean iterative solution to this problem without breaking the abstraction, and a recursive one will have to do (although I find it terribly inefficient, being proportional in depth to the amount of terms in the polynomial):

```
(define (negate-terms terms)
  (if (empty-termlist? terms)
    (the-empty-termlist)
    (let ((first (first-term terms)))
      (adjoin-term
        (make-term (order first) (neg (coeff first)))
        (negate-terms (rest-terms terms))))))

(define (negate-poly p)
  (make-poly
    (variable p)
    (negate-terms (term-list p))))

(put 'neg '(polynomial)
  (lambda (p) (tag (negate-poly p))))
```

`negate-terms` does its trick by reversing the list of terms implicitly in the Scheme program stack. Look at the call to `adjoin-term` – it obviously always has its elements in the correct order. The first term is created, and then `adjoin-term` "waits" for the recursive call to return the rest of the negated terms and prepends the first one to them. This way, the correct order is preserved.

Anyway, I almost forgot our real goal here is to subtract polynomials. Given the negation, it is trivial:

```
(define (sub-poly p1 p2)
  (add-poly p1 (negate-poly p2)))

(put 'sub '(polynomial polynomial)
  (lambda (p1 p2) (tag (sub-poly p1 p2))))
```

Exercise 2.89

There is a slight trick here. Note that in the sparse polynomial term-list, the order of each coefficient is explicit. In a dense term-list this is not so, however, and the order is implicit in the coefficient's location in the term-list.

So my proposed implementation is this:

```
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons (coeff term) term-list)))

(define (the-empty-termlist) '())

(define (first-term term-list)
  (cons
    (car term-list)
    (- (length term-list) 1)))

(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))
(define (make-term order coeff) (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))
```

Note that `first-term` and `adjoin-term` are different from the sparse versions. The strategy is as follows: while the term list is a simple list of coefficients (dense representation), a single term is an `(order coefficient)` pair. The order is determined in `first-term` by computing the length of the list, and `adjoin-term` removes the order to attach only the coefficient to a term list.

Exercise 2.90

After struggling with this for half an hour, I reached the conclusion this is too much work. Repeatable work. So, I'll skip it now and will perhaps get back to it later.

Exercise 2.91

This is the complete `div-terms`:

```
  (define (div-terms L1 L2)
    (if (empty-termlist? L1)
        (list (the-empty-termlist) (the-empty-termlist))
        (let ((t1 (first-term L1))
              (t2 (first-term L2)))
          (if (> (order t2) (order t1))
              (list (the-empty-termlist) L1)
              (let* ( (new-c (div (coeff t1) (coeff t2)))
                      (new-o (- (order t1) (order t2)))
                      (new-t (make-term new-o new-c))
                      (mult (mul-terms L2 (list new-t)))
                      (diff (add-terms
                              L1
                              (negate-terms mult))))
                (let ((rest-of-result
                        (div-terms diff L2)))
                  (list
                    (cons new-t (car rest-of-result))
                    (cadr rest-of-result)))))))))
```

Note the order of computations in the `let*` form – it carefully follows the algorithm outlined in the

text.

And this is the rest of the code to be added to `install-polynomial-package`:

```
(define (div-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
    (let ((div-result (div-terms
                        (term-list p1)
                        (term-list p2))))
      (list
        (make-poly (variable p1) (car div-result))
        (make-poly (variable p2) (cadr div-result))))
    (error "Polys not in same var -- DIV-POLY"
      (list p1 p2))))

(put 'div '(polynomial polynomial)
  (lambda (p1 p2)
    (let ((div-result (div-poly p1 p2)))
      (list
        (tag (car div-result))
        (tag (cadr div-result))))))
```

Exercise 2.92

Skipping.

Exercise 2.93

```
(define (install-rational-package)
  (define (tag x) (attach-tag 'rational x))

  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (equal-rat? x y)
    (and
      (equ? (numer x) (numer y))
      (equ? (denom x) (denom y))))

  (define (make-rat n d)
    (cons n d))

  (define (add-rat x y)
    (make-rat (add  (mul (numer x) (denom y))
                    (mul (numer y) (denom x)))
              (mul (denom x) (denom y))))

  (define (sub-rat x y)
    (make-rat (sub (mul (numer x) (denom y))
                   (mul (numer y) (denom x)))
              (mul (denom x) (denom y))))

  (define (mul-rat x y)
    (make-rat (mul (numer x) (numer y))
              (mul (denom x) (denom y))))

  (define (div-rat x y)
    (make-rat (mul (numer x) (denom y))
              (mul (denom x) (numer y))))

  ;; interface to rest of the system
  ;;
  (put 'equ? '(rational rational)
    (lambda (x y) (equal-rat? x y)))
  (put '=zero? '(rational)
    (lambda (x) (=zero? (numer x))))
  (put 'add '(rational rational)
    (lambda (x y) (tag (add-rat x y))))
  (put 'sub '(rational rational)
    (lambda (x y) (tag (sub-rat x y))))
  (put 'mul '(rational rational)
    (lambda (x y) (tag (mul-rat x y))))
  (put 'div '(rational rational)
    (lambda (x y) (tag (div-rat x y))))
  (put 'make 'rational
    (lambda (n d) (tag (make-rat n d))))
  'done)
```

Now executing:

```
(define p1 (make-polynomial 'x '((2 1) (0 1))))
(define p2 (make-polynomial 'x '((3 1) (0 1))))

(define rf (make-rational p1 p2))
(define rf2 (add rf rf))

(printf "~a~%" rf)
(printf "~a~%" rf2)
```

Produces the expected results.

Exercise 2.94

First, I'll add this to the `scheme-number` package:

```
  (put 'greatest-common-divisor '(scheme-number scheme-number)
    (lambda (a b) (gcd a b)))
```

And this to the outer code for the generic operation:

```
(define (greatest-common-divisor a b)
  (apply-generic 'greatest-common-divisor a b))
```

The additions to the polynomial package are:

```
  (define (remainder-terms a b)
    (cadr (div-terms a b)))

  (define (gcd-terms a b)
    (if (empty-termlist? b)
        a
        (gcd-terms b (remainder-terms a b))))

  (define (gcd-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
      (make-poly
        (variable p1)
        (gcd-terms (term-list p1) (term-list p2)))
      (error "Polys not in same var -- GCD-POLY"
        (list p1 p2))))

  (put 'greatest-common-divisor '(polynomial polynomial)
    (lambda (p1 p2) (tag (gcd-poly p1 p2))))
```

Exercise 2.95

```
(define p1 (make-polynomial 'x '((2 1) (1 -2) (0 1))))
(define p2 (make-polynomial 'x '((2 11) (0 7))))
(define p3 (make-polynomial 'x '((1 13) (0 5))))

(define q1 (mul p1 p2))
(define q2 (mul p1 p3))

(define gq (greatest-common-divisor q1 q2))

(printf "~a~%" gq)
=>
(polynomial x (2 1458/169) (1 -2916/169) (0 1458/169))
```

Which obviously isn't `p1`, although there is some consistency in it. Note that if 1458/169 represents 1, it is exactly the same as `p1`. So how did 1 turn into 1458/169 ?

We can find out by tracing some of the functions. In Scheme, tracing is not part of the language, but most implementations provide it. PLT Scheme has a `trace` function in the `trace.ss` library of MzLib. First it's imported with:

```
(require (lib "trace.ss"))
```

Then, I've added these lines to the polynomial package:

```
  (trace div-terms)
  (trace gcd-terms)
```

And when running the `greatest-common-divisor` call, the result is:

```
|(gcd-terms ((4 11) (3 -22) (2 18) (1 -14) (0 7)) ((3 13) (2 -21) (1 3) (0 5)))
| (div-terms
    ((4 11) (3 -22) (2 18) (1 -14) (0 7))
    ((3 13) (2 -21) (1 3) (0 5)))
| |(div-terms
     ((3 -55/13) (2 201/13) (1 -237/13) (0 7))
     ((3 13) (2 -21) (1 3) (0 5)))
| | (div-terms
      ((2 1458/169) (1 -2916/169) (0 1458/169))
      ((3 13) (2 -21) (1 3) (0 5)))
| | (() ((2 1458/169) (1 -2916/169) (0 1458/169)))
| |(((0 -55/169)) ((2 1458/169) (1 -2916/169) (0 1458/169)))
| (((1 11/13) (0 -55/169)) ((2 1458/169) (1 -2916/169) (0 1458/169)))
|(gcd-terms
   ((3 13) (2 -21) (1 3) (0 5))
   ((2 1458/169) (1 -2916/169) (0 1458/169)))
| (div-terms
    ((3 13) (2 -21) (1 3) (0 5))
    ((2 1458/169) (1 -2916/169) (0 1458/169)))
| |(div-terms ((2 5) (1 -10) (0 5)) ((2 1458/169) (1 -2916/169) (0 1458/169)))
| | (div-terms () ((2 1458/169) (1 -2916/169) (0 1458/169)))
| | (() ())
| |(((0 845/1458)) ())
| (((1 2197/1458) (0 845/1458)) ())
|(gcd-terms ((2 1458/169) (1 -2916/169) (0 1458/169)) ())
|((2 1458/169) (1 -2916/169) (0 1458/169))
(polynomial x (2 1458/169) (1 -2916/169) (0 1458/169))
```

Let's see what's happening here. `gcd-terms` is called with `q1` and `q2` and divides one by another to find the remainder. The quotient of this division is: `(((1 11/13) (0 -55/169))`, and the remainder is: `((2 1458/169) (1 -2916/169) (0 1458/169)))`, so this is where the fractions are coming from.

Exercise 2.96

a.

```
(define (pseudoremainder-terms a b)
  (let*  ((t1 (first-term a))
          (o1 (order t1))
          (t2 (first-term b))
          (o2 (order t2))
          (c (coeff t2))
          (ic (expt c (+ 1 o1 (- o2)))))
    (cadr
      (div-terms
        (map
          (lambda (t)
            (make-term (order t) (* ic (coeff t))))
          a)
        b))))

(define (gcd-terms a b)
  (if (empty-termlist? b)
      a
      (gcd-terms b (pseudoremainder-terms a b))))
```

Now the result is:

```
(polynomial x (2 1458) (1 -2916) (0 1458))
```

b.

Luckily, Scheme's built-in `gcd` function knows how to compute the answer for an arbitrary amount of arguments. This allows us to rewrite `gcd-terms` as:

```
  (define (gcd-terms a b)
    (if (empty-termlist? b)
        a
        (let*  ((gcd-res (gcd-terms b (pseudoremainder-terms a b)))
                (coeff-list (map cadr gcd-res))
                (coeff-gcd (apply gcd coeff-list)))
          (map
            (lambda (t)
              (make-term  (order t)
                          (/ (coeff t) coeff-gcd)))
            gcd-res))))
```

Exercise 2.97

a.

I took some common code out into a separate function, so here is the full code for this extended exercise:

```
  (define (map-coeffs fn term-list)
    "fn is a function that will be called on each
    coefficient in term-list, and is expected to
    return a new coefficient."
    (map
```

```
    (lambda (t)
      (make-term (order t) (fn (coeff t))))
    term-list))

(define (reduce-terms n d)
  (let*  ((nd-gcd (gcd-terms n d))
          (o1 (max  (order (first-term n))
                    (order (first-term d))))
          (o2 (order (first-term nd-gcd)))
          (c (coeff (first-term nd-gcd)))
          (ic (expt c (+ 1 o1 (- o2))))
          (ni (map-coeffs
                (lambda (c) (* c ic)) n))
          (di (map-coeffs
                (lambda (c) (* c ic)) d))
          (nn (quotient-terms ni nd-gcd))
          (dd (quotient-terms di nd-gcd)))
    (list nn dd)))

(define (quotient-terms a b)
  (car (div-terms a b)))

(define (remainder-terms a b)
  (cadr (div-terms a b)))

(define (pseudoremainder-terms a b)
  (let*  ((t1 (first-term a))
          (o1 (order t1))
          (t2 (first-term b))
          (o2 (order t2))
          (c (coeff t2))
          (ic (expt c (+ 1 o1 (- o2)))))
    (cadr
      (div-terms
        (map-coeffs (lambda (c) (* c ic)) a)
        b))))

(define (gcd-terms a b)
  (if (empty-termlist? b)
      a
      (let*  ((gcd-res (gcd-terms b (pseudoremainder-terms a b)))
              (coeff-list (map cadr gcd-res))
              (coeff-gcd (apply gcd coeff-list)))
        (map
          (lambda (t)
            (make-term  (order t)
                        (/ (coeff t) coeff-gcd)))
          gcd-res))))

(define (div-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
    (let ((div-result (div-terms
```

```
                          (term-list p1)
                          (term-list p2))))
        (list
          (make-poly (variable p1) (car div-result))
          (make-poly (variable p2) (cadr div-result))))
      (error "Polys not in same var -- DIV-POLY"
        (list p1 p2))))

  (define (gcd-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
      (make-poly
        (variable p1)
        (gcd-terms (term-list p1) (term-list p2)))
      (error "Polys not in same var -- GCD-POLY"
        (list p1 p2))))

  (define (reduce-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
      (let ((result (reduce-terms
                      (term-list p1)
                      (term-list p2))))
        (list
          (make-poly (variable p1) (car result))
          (make-poly (variable p2) (cadr result))))
      (error "Polys not in same var -- REDUCE-POLY"
        (list p1 p2))))
```

b.

```
;; Added to the scheme-number package
;;
(define (reduce-integers n d)
  (let ((g (gcd n d)))
    (list (/ n g) (/ d g))))

(put 'reduce '(scheme-number scheme-number)
  (lambda (n d) (reduce-integers n d)))

;; Added as an outer function
;;
(define (reduce n d) (apply-generic 'reduce n d))

;; Added to the polynomial package
;;
(put 'reduce '(polynomial polynomial)
  (lambda (p1 p2)
    (let ((result (reduce-poly p1 p2)))
      (list
        (tag (car result))
        (tag (cadr result))))))
```

Now, the answer of `(add rf1 rf2)` is:

```
(rational
  (polynomial x (3 1) (2 2) (1 3) (0 1))
  polynomial x (4 1) (3 1) (1 -1) (0 -1))
```

As expected.

Afterword

This concludes chapter 2 of SICP. Phew, that was lots of work. Especially the last few sections with the generic arithmetic packages. I feel it got a little repetitive and tiresome towards the end, so I even skipped a couple of exercises. Nevertheless, I learned a lot from this chapter and am looking forward to chapter 3.

For comments, please send me an email.

Back to top