**Structures and Interpretation of Computer Program**

**Exercise Chapter 2.5 Name:** Wan Huzaifah bin Wan Azhar

**Exercise 2.5.3 Representing Term List**

(define (install-=zero?-polynomial)

(define (=zero? term)

(if (empty-termlist? term)

#t

(if (=zero? (coeff (car poly)))

(=zero? (coeff (cdr poly)))

#f))

(put '=zero? 'polynomial))

1. I use generic operation sub and change add-terms to substract-terms

(define (install-substract-polynomial)

(define (negate-term term)

(define (negate-coeff cterm)

(if (null? cterm)

'()

(cons (\* -1 (car cterm)) (negate-coeff (cdr cterm)))))

(if (empty-termlist? term)

the-empty-termlist

(cons (order term) (negate-coeff (coeff term)))))

(define (substract-terms L1 L2)

(define (substract-term L1 L2)

(let ((t1 (first-term L1)) (t2 (first-term L2)))

(cond ((> (order t1) (order t2))

(adjoin-term

t1 (substract-terms (rest-terms L1) L2)))

((< (order t1) (order t2))

(adjoin-term

t2 (substract-terms L1 (rest-terms L2))))

(else

(adjoin-term

(make-term (order t1)

(sub (coeff t1) (coeff t2)))

(substract-terms (rest-terms L1)

(rest-terms L2)))))))

(cond

((empty-termlist? L1) (negate-term L2))

((empty-termlist? L2) L1)

(else (substract-term L1 L2))))

(put 'sub '(polynomial polynomial) substract-terms)

)

1. Everything else is the same except adjoin-list and first-term

(define (coeff term) (cadr term))

(define (order term) (car term))

(define (make-term order coeff) (list order coeff))

(define (first-term term-list)

(make-term (- 1 (length term-list)) (car term-list))

)

(define (adjoin-term term term-list) ; adjoin dense term to sparse term-list

(let

((order-term (order term))

(coeff-term (coeff term))

(length-termlist (length term-list)))

(cond

((< order-term length-termlist) (error "Fail to append"))

((= order-term length-termlist) (cons coeff-term term-list))

(else (cons coeff-term (adjoin-term (make-term length-termlist 0) term-list)))

)

)

)

(display (adjoin-term (list 5 2) (list 0 3 -2 -5)))

4.

(define (install-term-dense-package)

(define (make-term coeff) (coeff))

(define (order term) (- 1 (length term)))

(define (coeff term) (car term))

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

(define (first-term term-list) (list (order term-list) (car term-list)))

(define (rest-term term-list) (cdr term-list))

(define (empty-termlist? term-list) (null? term-list))

(define (adjoin-term term term-list)

(let

((order-term (order term))

(coeff-term (coeff term))

(length-termlist (length term-list)))

(cond

((< order-term length-termlist) (error "Fail to append"))

((= order-term length-termlist) (cons coeff-term term-list))

(else (cons coeff-term (adjoin-term (make-term length-termlist 0) term-list)))

)

)

)

(define (tag x) (attach-tag 'dense x))

; interface

(put 'make-term 'dense (lambda (a) (tag (make-term a))))

(put 'order '(dense) order)

(put 'coeff '(dense) coeff)

(put 'the-empty-termlist '(dense) the-empty-termlist)

(put 'first-term '(dense) first-term)

(put 'rest-term '(dense) rest-term)

(put 'empty-termlist? '(dense) empty-termlist?)

(put 'adjoin-term '(dense) adjoin-term)

)

(define (install-term-sparse-package)

(define (make-term order coeff) (list order coeff))

(define (order term) (car term)

(define (coeff term) (cadr term))

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

(define (first-term term-list) (car term-list))

(define (rest-term term-list) (cdr term-list))

(define (empty-termlist? term-list) (null? term-list))

(define (adjoin-term term term-list) ; adjoin dense term to sparse term-list

(if (=zero? (coeff term))

term-list

(cons term term-list)))

(define (tag x) (attach-tag 'sparse x))

; interface

(put 'make-term 'sparse (lambda (a) (tag (make-term a))))

(put 'order '(sparse) order)

(put 'coeff '(sparse) coeff)

(put 'the-empty-termlist '(sparse) the-empty-termlist)

(put 'first-term '(sparse) first-term)

(put 'rest-term '(sparse) rest-term)

(put 'empty-termlist? '(sparse) empty-termlist?)

(put 'adjoin-term '(sparse) adjoin-term)

)

(define (adjoin-term term term-list) ((get 'adjoin-term (type-tag term-list)) term term-list))

(define (first-term term-list) (apply-generic 'first-term term-list))

(define (rest-term term-list) (apply-generic 'rest-term term-list))

5.

(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))))

(let ((rest-of-result (div-terms

(sub-terms L1

(mul-terms-by-all-terms (make-term new-o new-c) L2))

L2)))

(list (adjoin-term (make-term new-o new-c) (car rest-of-result)) (cadr rest-of-result))))))))

(define (div-poly L1 L2)

(if (same-variable? (variable L1) (variable L2))

(let ((L1-term (term-list L1))

(L2-term (term-list L2)))

(let ((div-result (div-terms L1-term L2-term)))

(let ((div-result-remains (car (reverse div-result)))

(div-result-terms (reverse (cdr (reverse div-result)))))

(list (make-poly (variable L1) div-result-terms)

(make-poly (variable L1) div-result-remains)))))

(error "Cannot divide polynomial because of different variable")))

6.

Skipped because it is difficult.

7. Modify make-rat

(define (numer x) (car x))

(define (denom x) (cdr x))

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

(define (add-rat x y)

(make-rat (+ (\* (numer x) (denom y))

(\* (numer y) (denom x)))

(\* (denom x) (denom y))))

(define (sub-rat x y)

(make-rat (- (\* (numer x) (denom y))

(\* (numer y) (denom x)))

(\* (denom x) (denom y))))

(define (mul-rat x y)

(make-rat (\* (numer x) (numer y))

(\* (denom x) (denom y))))

(define (div-rat x y)

(make-rat (\* (numer x) (denom y))

(\* (denom x) (numer y))))

(define (tag-rat x) (attach-tag 'rational x))

(define (make-rational n d)

(tag-rat (make-rat n d)))

#define poly package here

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

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

(define rf (make-rational p2 p1))

(display rf)

8.

(define (install-gcd-package)

(define (remainder-terms t1 t2)

(cadr (div-terms t1 t2)))

(define (gcd-integer a b)

(if (= b 0)

a

(gcd b (remainder 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 (equal? (variable p1-data) (variable p2-data))

(make-polynomial (variable p1-data) (gcd-terms (term-list p1-data) (term-list p2-data)))

(error "Polynomial does not have same variable")))

(put 'gcd '(scheme-number scheme-number) (lambda (a b) (make-scheme-number (gcd-integer a b))))

(put 'gcd '(polynomial polynomial) (lambda (p1 p2) (make-polynomial (gcd-poly p1 p2))))

'done)

(define (greatest-common-divisor a b)

(apply-generic 'gcd a b))

9.

Let,

Q1 = (x^2 – 2x + 1) (11x^2 + 7)

Q2 = (x^2 – 2x + 1) (13x + 5)

Process:

(gcd q1 q2) produce 908/169 as remainder, which is noninteger.

(gcd q2 908/169) which is difficult to calculate and thus, cause problem.

10.



(define (pseudoremainder-terms t1 t2)

(let ((O1 (order (first-term t1)))

(O2 (order (first-term t2)))

(c (coeff (first-term t2))))

(let ((integrizing-factor-terms (make-term 0 (expt c (+ 1 (- O1 O2))))))

(cadr (div-terms (mul-terms t1 integrizing-factor-terms) t2)))))

(define (gcd-terms a b)

(if (empty-termlist? b)

a

(gcd-terms b (pseudoremainder-terms a b))))



I’m giving up on these. My best attempt:

(define (gcd-terms a b)

(define (remove-common-factor term-list factor)

(if (empty-termlist? term-list)

'()

(let ((term (first-term term-list)))

(cons (make-term (order term) (/ (coeff term) factor))

(remove-common-factor (rest-term term-list) factor)))))

(define (calculate-gcd a b)

(if (empty-termlist? b)

a

(calculate-gcd b (pseudoremainder-terms a b))))

(let ((gcd-int (gcd-integer a b)))

(remove-common-factor (calculate-gcd a b) gcd-int))

)

11.

(define (reduce-terms n d)

(define (calculate-gcd-all-coeff n d)

(if (empty-termlist? n)

'()

(cons (make-term 0 (gcd (coeff (first-term n) (coeff (first-term d)))))

(calculate-redund-factor (rest-term n) (rest-term d)))))

(let ((gcd-n-d (gcd-terms n d)))

(let ((integr-factor (expt (coeff (first-term gcd-n-d))

(+ 1 (- (order (first-term gcd-n-d))) (max (order (first-term n)) (order (first-term d))))))

(gcd-all-coeff (calculate-gcd-all-coeff n d)))

(let ((divided-terms-n-result ((divide-terms (mul-terms n integr-factor)

gcd-n-d)))

(divided-terms-d-result ((divide-terms (mul-terms d integr-factor)

gcd-n-d))))

(cons (divide-terms divided-terms-n-result gcd-all-coeff)

(divide-terms divided-terms-d-result gcd-all-coeff))))))

(define (reduce-poly p1 p2)

(if (same-variable? (variable p1) (variable p2))

(let ((reduced-term (reduce-terms (term-list p1) (term-list p2))))

(list (make-poly (variable p1) (car reduced-term))

(make-poly (variable p2) (cadr reduced-term))))

(error "Cannot reduce polynomial: polynomial variable is not equal")))



(define (install-reduce-pkg)

(define (reduce-terms n d)

(define (calculate-gcd-all-coeff n d)

(if (empty-termlist? n)

'()

(cons (make-term 0 (gcd (coeff (first-term n) (coeff (first-term d)))))

(calculate-redund-factor (rest-term n) (rest-term d)))))

(let ((gcd-n-d (gcd-terms n d)))

(let ((integr-factor (expt (coeff (first-term gcd-n-d))

(+ 1 (- (order (first-term gcd-n-d))) (max (order (first-term n)) (order (first-term d))))))

(gcd-all-coeff (calculate-gcd-all-coeff n d)))

(let ((divided-terms-n-result ((divide-terms (mul-terms n integr-factor)

gcd-n-d)))

(divided-terms-d-result ((divide-terms (mul-terms d integr-factor)

gcd-n-d))))

(cons (divide-terms divided-terms-n-result gcd-all-coeff)

(divide-terms divided-terms-d-result gcd-all-coeff))))))

(define (reduce-poly p1 p2)

(if (same-variable? (variable p1) (variable p2))

(let ((reduced-term (reduce-terms (term-list p1) (term-list p2))))

(list (make-poly (variable p1) (car reduced-term))

(make-poly (variable p2) (cadr reduced-term))))

(error "Cannot reduce polynomial: polynomial variable is not equal")))

(define (reduce-integers n d)

(let ((g (gcd n d)))

(list (/ n g) (/ d g))))

(put 'reduce '(scheme-number scheme-number)

(lambda (n d) (let ((reduced-int (reduce-integers n d)))

(map tag reduced-int))))

(put 'reduce '(polynomial polynomial)

(reduce-poly n d)) ; Make-poly will automatically tag

(define (reduce a b)

(apply-generic 'reduce a b))

(define (make-rat n d)

(let ((reduced (reduce n d)))

(cons (car reduced) (cadr reduced))))

'done)