Scheme

Кевролетин В.В. 236гр.

28 марта 2011 г.

Задание18

Условие

map, append, length через accumulate

Решение

```
(define (map p sequence)
  (accumulate (lambda (x y) (cons (p x) y)) '() sequence))

(define (append seq1 seq2)
  (accumulate cons seq2 seq1))

(define (length sequence)
  (accumulate (lambda (x y) (+ 1 y)) 0 sequence))
```

Задание19

Условие

accumulate-n

Решение

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      '()
      (cons (accumulate op init (map car seqs))
            (accumulate-n op init (map cdr seqs)))))
```

Задание20

Условие

dot-product, matrix-*-vector, transpose mat, matrix-*-matrix

Решение

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

```
(define (matrix-*-vector m v)
  (map (lambda (x) (dot-product x v)) m))

(define (transpose mat)
  (accumulate-n (lambda (x y) (cons x y)) '() mat))

(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (lambda (row) (matrix-*-vector n row) ) m)))
```

Задание21

Условие

fold-right and fold-left

Решение

```
(define (fold-right op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (fold-right op initial (cdr sequence)))))

(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))
```

Задание22

Условие

reverse через fold-right and fold-left

Решение

```
(define (reverse sequence)
  (fold-right (lambda (x y) (append y (list x))) '() sequence))

(define (reverse sequence)
  (fold-left (lambda (x y) (cons y x)) '() sequence))
```

Задание23

Условие

Two lists are said to be equal? if they contain equal elements arranged in the same order.

Решение

```
(define (equal? a b)
  (cond
   ((and (pair? a) (pair? b))
    (and (eq? (car a) (car b)) (equal? (cdr a) (cdr b))))
   ((and (not (pair? a)) (not (pair? b)))
    (eq? a b))
   (else '())))
```

Задание24

Условие

implement the differentiation rule for u^n...

Решение

```
(define (deriv exp var)
  (cond ...
        ((power? exp)
         (make-product
           (make-power (power-get-arg exp) (- (power-get-pow exp) 1))
           (deriv (power-get-arg exp) var)))
        ...)

(define (make-power num pow) (list '^ num pow))

(define (power? x)
  (and (pair? x) (eq? (car x) '^)))

(define (power-get-arg p) (cadr p))
(define  (power-get-pow p)  (caddr p))

;;usage
(deriv (make-power (make-sum 'x 1) 2) 'x)
;; > (* (^ (+ x 1) 1) (+ 1 0))
```

Задание25

Условие

Extend the differentiation program to handle sums and products of arbitrary numbers of (two or more) terms

Решение

```
(define (deriv exp var)
  (cond ...
        ((sum? exp)
```

```
     (cons '+
           (foldr (lambda (x y) (cons (deriv x var) y))
                  '() (sum-args exp))))
    ((product? exp)
     (make-sum
       (make-product (deriv (product-first-arg exp) var)
                     (product-last-args exp))
       (make-product (product-first-arg exp)
                     (deriv (product-last-args exp) var))))
    ...)

(define (make-sum a1 . a2) (append (list '+ a1) a2))
(define (sum-args s) (cdr s))

(define (make-product m1 . m2) (append (list '* m1) m2))
(define (product-args p) (cdr p))
(define (product-first-arg s) (car (product-args s)))
(define (product-last-args s)
  (let ((tail (product-args s)))
    (if (> (length tail) 2)
        (cons '* tail)
        (car tail))))

;;usage

(define s (make-sum 1 2 3))
(sum-args s)                                  ;; > (1 2 3)

(define p (make-product 1 2 3))
(product-args p)                              ;; > (1 2 3)
(product-first-arg p)                         ;; > 1
(product-last-args p)                         ;; > (* 2 3)
(product-last-args (product-last-args p))     ;; > 3

(deriv (make-sum 'x 'x 1) 'x)                 ;; > (+ 1 1 0)
(deriv (make-product 'x 'x) 'x)               ;; > (+ (* 1 x) (* x 1))
```

Задание26

Условие

Suppose we want to modify the differentiation program so that it works with ordinary mathematical notation, in which + and * are infix rather than prefix operators a. Show how to do this in order to differentiate algebraic expressions presented in infix form, such as (x + (3 * (x + (y + 2)))). To simplify the task, assume that + and * always take two arguments and that expressions are fully parenthesized. b. The problem becomes substantially harder if we allow standard algebraic notation, such as (x + 3 * (x + y + 2)), which drops unnecessary parentheses and assumes that multiplication is done before addition. Can you design appropriate predicates, selectors, and constructors for this notation such that our derivative program still works?

Решение

а)

```
(define (make-sum a1 a2) (list a1 '+ a2))
(define (make-product m1 m2) (list m1 '* m2))

(define (sum? x)
  (and (pair? x) (eq? (cadr x) '+)))
(define (addend s) (car s))
(define (augend s) (caddr s))

(define (product? x)
  (and (pair? x) (eq? (cadr x) '*)))
(define (multiplier p) (car p))
(define (multiplicand p) (caddr p))
```

б) Ответ: да, дейвствительно, можно. Для этого потребуется добавить более сложный селектор, который выбирает из списка не один элемент, а все сомножители произведения. И селектор, выбирающий оставшуюся часть.

```
(define (make-sum l1 l2)
  (append l1 (list '+) l2))

(define (make-mult l1 l2)
  (append l1 (list '*) l2))

(define (check-for-oper x op)
  (and
   (not (null? x))
   (not (null? (cdr x)))
   (eq? (cadr x) op)))

(define (mult? x)
  (check-for-oper x '*))

(define (sum? x)
  (or (check-for-oper x '+)
      (and (pair? x)
      (null? (cdr x)))))

(define (select-mult p)
  (define (iter l res)
    (if (not (mult? l)) (cons (car l) res)
        (iter (cddr l) (cons '* (cons (car l) res)))))
  (iter p '()))

(define (select-after-mult p)
  (define (iter l res)
    (if (not (mult? l)) (cddr p)
        (iter (cddr l) (cons '* (cons (car l) res)))))
  (iter p '()))
```