Scheme

Кевролетин В.В. 236гр.

11 марта 2011 г.

Задание

Условие

Имеются монеты определённых достоинств. Необходимо из них составить всеми возможными способами определённую сумму.

Решение1

Данный вариант реализации выводит все возможные наборы. Используется рекурсивный алгоритм: задача делится на 2 подзадачи:
а) ищутся все наборы, которые не содержат первый символ из алфавита
б) идутся все наборы, которые содержат в себе первый символ из алфавита

```
(define (solve-rec alph sum curr-res tot-res)
  (if (null? alph)
      (if (= sum 0) (cons curr-res tot-res) tot-res)
      (if (< sum 0) tot-res
          (let ((num (car alph)))
            (let ((part-res (solve-rec alph (- sum num) (cons num curr-res) tot-res)))
              (solve-rec (cdr alph) sum curr-res part-res))))))

(define (solve alph sum)
  (solve-rec alph sum '() '()))

(solve '(10 25 50) 100) ; =>

;((50 50)
; (50 25 25)
; (25 25 25 25)
; (50 10 10 10 10 10)
; (25 25 10 10 10 10 10)
; (10 10 10 10 10 10 10 10 10 10))
```

Решение2

Второй алгоритм генерирует по текущему набору следующий, увеличивая количество элементов в наборе так, чтобы полученная сумма на превосходила той суммы, которую надо набрать. [формат вывода результата отличается от предыдущей реализации: в списках хранятся не сами достоинства монет, а количество монет соответствующего достоинтва]

```
(define (calc-sum-rec alph curr-res tot-res)
  (if (null? curr-res) tot-res
      (calc-sum-rec
       (cdr alph)
       (cdr curr-res)
       (+ (* (car alph) (car curr-res)) tot-res))))

(define (calc-sum alph curr-res)
  (calc-sum-rec alph curr-res 0))
```

```
(define (add-one alph sum curr-res)
  (if (null? curr-res) '()
      (let ((new-res (cons (+ 1 (car curr-res)) (cdr curr-res))))
        (let ((new-sum (calc-sum alph new-res)))
          (if (> new-sum sum) (cons 0 (add-one (cdr alph) sum (cdr curr-res)))
              new-res)))))

(define (next alph sum curr-res)
  (let ((new-res (add-one alph sum curr-res)))
    (let ((new-sum (calc-sum alph new-res)))
      (cond
        ((= new-sum sum) new-res)
        ((= new-sum 0) '())
        (#t (next alph sum new-res))))))

(define (solve-rec alph sum curr-res tot-res)
  (let ((next-res (next alph sum curr-res)))
    (if (null? next-res) tot-res
        (solve-rec alph sum next-res (cons next-res tot-res)))))

(define (solve alph sum)
  (let ((zeros (map (lambda (x) 0) alph)))
    (solve-rec alph sum zeros '())))

; some tests

(solve '(2 3) 7)     ; ((2 1))
(solve '(1 2 3) 6)
; ((0 0 2) (1 1 1) (3 0 1) (0 3 0) (2 2 0) (4 1 0) (6 0 0))
(solve '(10 25 50) 100)
; ((0 0 2) (0 2 1) (5 0 1) (0 4 0) (5 2 0) (10 0 0))
```