Лабораторная работа № 4. Метапрограммирование. Отложенные вычисления

11 ноября 2024 г.

Булат Валиуллин, ИУ9-11Б

Цель работы

На примере языка Scheme ознакомиться со средствами метапрограммирования («код как данные», макросы) и подходами к оптимизации вычислений (мемоизация результатов вычислений, отложенные вычисления).

Индивидуальный вариант

proc-desugar, count_str, fibonacci, when, for, repeat…until

Реализация

```
(define abort #f)

(define-syntax assert
   (syntax-rules ()
     ((assert cond)
      (begin
        (if (not cond)
            (begin
              (display "FAILED ")
              (write 'cond)
              (newline)
              (abort))
            )
        ))))

(call-with-current-continuation (lambda (k) (set! abort k)))
```

```
;; 2.1
(define (proc-desugar source dest)
  (call-with-input-file source
    (lambda (port1)
      (let ((in (read port1)))
        (call-with-output-file dest
          (lambda (port2)
            (display "(define " port2)
            (display (caadr in) port2)
            (display " (lambda " port2)
            (display (cdaddr in) port2)
            (display " " port2)
            (write (caddr in) port2)
            (display "))" port2)
            ))))))

;; 2.2
(define (contains_symbs? line)
  (define (symb? chars)
    (cond
      ((null? chars) #f)
      ((equal? #\space (car chars)) (symb? (cdr chars)))
      (else #t)))

  (symb? line))

(define (create_line port)
  (define (reader line)
    (let ((char (peek-char port)))
      (cond
        ((eof-object? char) (reverse line))
        ((equal? char #\newline)
         (read-char port)
         (if (null? line)
             (reader '())
             (reverse line)))
        (else
         (reader (cons (read-char port) line))))))

  (reader '()))


(define (count_str source)
  (call-with-input-file source
    (lambda (port)
      (define (counter count)
```

```
        (let ((line (create_line port)))
          (if (null? line)
              count
              (if (contains_symbs? line)
                  (counter (+ count 1))
                  (counter count)))))

      (counter 0))))


;; 3
(define known-results '())

(define (tribonacci-memo n)
  (cond
    ((<= n 1) 0)
    ((= n 2) 1)
    (else
     (let ((res (assoc n known-results)))
       (if found
           (cdr res)
           (let ((result (+ (tribonacci-memo (- n 1))
                            (tribonacci-memo (- n 2))
                            (tribonacci-memo (- n 3)))))
             (set! known-results (cons (cons n result) known-results))

             result))))))

(define (tribonacci n)
  (cond
    ((= n 1) 0)
    ((= n 2) 0)
    ((= n 3) 1)
    (else
     (+ (tribonacci (- n 1))
        (tribonacci (- n 2))
        (tribonacci (- n 3))))))


; 4.0
(define-syntax lazy-cons
  (syntax-rules ()
    ((lazy-cons a b)
     (cons a (delay b)))))

(define (lazy-car p)
```

```
  (car p))

(define (lazy-cdr p)
  (force (cdr p)))

(define (lazy-head xs k)
  (if (> k 0)
      (cons (lazy-car xs) (lazy-head (lazy-cdr xs) (- k 1)))
      '()))

(define (lazy-ref xs k)
  (if (> k 0)
      (lazy-ref (lazy-cdr xs) (- k 1))
      (car xs)))

(define (lazy-map proc xs)
  (if (not (null? xs))
      (lazy-cons (proc (lazy-car xs)) (lazy-map proc (lazy-cdr xs)))
      '()))

(define (lazy-zip xs ys)
  (if (null? xs)
      '()
      (lazy-cons (list (lazy-car xs) (lazy-car ys)) (lazy-zip (lazy-cdr xs) (lazy-cdr ys)))))
  )

;;4.3
(define fibonacci
  (lazy-cons 1
             (lazy-cons 1
                        (lazy-map (lambda (els) (+ (car els) (cadr els)))
                                  (lazy-zip fibonacci (lazy-cdr fibonacci)))
                        )
             )
  )

;; 5a
(define-syntax when
  (syntax-rules ()
    ((when cond? expr1 expr2 ...)
     (if cond?
         (begin expr1 expr2 ...)))))

;; 5b
(define-syntax for
  (syntax-rules (in as)
```

```
    ((for x in xs expr1 ...)
     (letrec ((iter (lambda (lst)
                      (if (null? lst)
                          '()
                          (let ((x (car lst)))
                            expr1
                            ...
                            (iter (cdr lst)))))))
       (iter xs)))
    ((for xs as x expr1 ...)
     (letrec ((iter (lambda (lst)
                      (if (null? lst)
                          '()
                          (let ((x (car lst)))
                            expr1
                            ...
                            (iter (cdr lst)))))))
       (iter xs)))
    ))

;; 5c
(define-syntax repeat
  (syntax-rules ()
    ((repeat (expr1 ...) until cond?)
     (let loop ()
       expr1
       ...
       (if (not cond?)
           (loop)
           )))))
```

Тестирование

```
Welcome to DrRacket, version 8.2 [cs].
Language: R5RS; memory limit: 128 MB.
> ; Определение процедуры, требующей верификации переданного ей значения:

(define (1/x x)
  (assert (not (zero? x))) ; Утверждение: x ДОЛЖЕН БЫТЬ ≠ 0
  (/ 1 x))

; Применение процедуры с утверждением:

(map 1/x '(1 2 3 4 5)) ; ВЕРНЕТ список значений в программу
```

```
(map 1/x '(-2 -1 0 1 2)) ; ВЫВЕДЕТ в консоль сообщение и завершит работу программы
(1 1/2 1/3 1/4 1/5)
FAILED (not (zero? x))
> (proc-desugar "lab4.rkt" "lab4-desugar.rkt")
> (define ones (lazy-cons 1 ones))
> (lazy-head ones 5)
(1 1 1 1 1)
> (define x 1)
> (when (> x 0) (display "x > 0")  (newline))
x > 0
> (for i in '(1 2 3)
  (for j in '(4 5 6)
    (display (list i j))
    (newline)))

(for '(1 2 3) as i
  (for '(4 5 6) as j
    (display (list i j))
    (newline)))
(1 4)
(1 5)
(1 6)
(2 4)
(2 5)
(2 6)
(3 4)
(3 5)
(3 6)
()
(1 4)
(1 5)
(1 6)
(2 4)
(2 5)
(2 6)
(3 4)
(3 5)
(3 6)
()
> (let ((i 0)
      (j 0))
  (repeat ((set! j 0)
           (repeat ((display (list i j))
                    (set! j (+ j 1)))
                   until (= j 3))
```

```
          (set! i (+ i 1))
          (newline))
         until (= i 3)))
(0 0)(0 1)(0 2)
(1 0)(1 1)(1 2)
(2 0)(2 1)(2 2)
```

Вывод

Понял, что такое call/cc (УРА!), понял, как с файликами работать, и одновременно с этим — как же я недооценивал возможность сокращать кучукаровсидиэров до потрясающих `cdaddr`. А если из более серьёзного, то стало понятно, как работает мемоизация, отложенные вычисления. Какая же благодарность тем, кто придумал добавить `...` в Scheme, как же это невероятно удобно и всё упрощает.