Лабораторная работа № 4. Метапрограммирование. Отложенные вычисления

15 декабря 2023 г.

Бойко Роман, ИУ9-12Б

Цель работы

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

В работе также предлагается разработать дополнительное средство отладки программ — каркас для отладки с помощью утверждений. На этом примере предлагается ознакомится с типичным применением программирования с использованием продолжений.

Реализация

```
(load "./unit-test.scm")


;; Number 1
(define call/cc call-with-current-continuation)

(define (exit) (display "You need include assert using use-assertions macros"))

(define-syntax use-assertions
  (syntax-rules ()
    ((use-assertions)
     (call/cc
      (lambda (cc)
        (set! exit cc))))))

(use-assertions)
```

```
(define-syntax assert
  (syntax-rules ()
    ((assert condition)
     (if (not condition)
         (begin
           (write "FAILED: ")
           (write 'condition)
           (exit))))))


;; Number 2
(define (save-data data path)
  (call-with-output-file path
    (lambda (port)
      (write data port))))

(define (load-data path)
  (call-with-input-file path
    (lambda (port)
      (read port))))

(define (count-lines path)
  (call-with-input-file path
    (lambda (port)
      (let loop ((port port)
                 (count 0)
                 (symbol (read-char port)))
        (cond
          ((eof-object? symbol) (newline) count)
          ((and (equal? #\newline symbol)
                (not (equal? #\newline (peek-char port))))
           (loop port (+ 1 count) (read-char port)))
          (else (begin
                  (display symbol)
                  (loop port count (read-char port)))))))))


;; Number 3
(define tribonacci-memo
  (let ((known-results '()))
    (lambda (n)
      (let* ((args (list n))
             (res (assoc args known-results)))
        (if res
            (cadr res)
            (let ((res
```

```
                    (cond
                      ((<= n 1) 0)
                      ((= n 2) 1)
                      (else (+
                             (tribonacci-memo (- n 1))
                             (tribonacci-memo (- n 2))
                             (tribonacci-memo (- n 3)))))))
                (set! known-results (cons (list args res) known-results))
                res))))))

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


;; Number 4
(define-syntax my-if
  (syntax-rules ()
    ((my-if condition true-state false-state)
     (let
         ((promise1 (delay true-state))
          (promise2 (delay false-state)))
       (force (or (and condition promise1) promise2))))))


;; Number 5
(define-syntax my-let
  (syntax-rules ()
    ((my-let ((var val) ...) expr)
     ((lambda (var ...) expr) val ...))))

(define-syntax my-let*
  (syntax-rules ()
    ((my-let* () expr) expr)
    ((my-let* ((var1 expr1) (varn exprn) ...) expr)
     ((lambda (var1)
        (my-let* ((varn exprn) ...)
                 expr))
      expr1))))


;; Number 6

;; A
```

```
(define-syntax when
  (syntax-rules ()
    ((when cond? . actions)
     (if cond?
         (begin . actions)))))

(define-syntax unless
  (syntax-rules ()
    ((when cond? . actions)
     (if (not cond?)
         (begin . actions)))))

;; Б
(define-syntax for
  (syntax-rules (in as)
    ((for x in xs . actions)
     (if (not (null? xs))
         (let loop ((x (car xs))
                    (new_xs (cdr xs)))
           (cond
             ((null? new_xs) (begin . actions))
             (else
              (begin
                (begin . actions)
                (loop (car new_xs) (cdr new_xs)))))))))
    ((for xs as x . actions)
     (for x in xs . actions))))

;; В
(define-syntax while
  (syntax-rules ()
    ((while cond? . actions)
     (let loop ((continue? cond?))
       (if continue?
           (begin
             (begin . actions)
             (loop cond?)))))))

;; Г
(define-syntax repeat
  (syntax-rules (until)
    ((repeat (expr1 . others) until cond?)
     (let loop ()
       (begin
         (begin expr1)
         (begin . others)
```

```
        (if (not cond?)
            (loop cond?)))))))

;; Д
(define-syntax cout
  (syntax-rules (<< endl)
    ((cout << endl) (newline))
    ((count << expr) (display expr))
    ((cout << endl . actions)
     (begin
       (newline)
       (cout . actions)))
    ((cout << expr . actions)
     (begin
       (display expr)
       (cout . actions)))))
```

Тестирование

```
;; Test 3
(define tests-tribonacci-memo
  (list (test (tribonacci-memo 0) 0)
        (test (tribonacci-memo 1) 1)
        (test (tribonacci-memo 2) 1)
        (test (tribonacci-memo 3) 2)
        (test (tribonacci-memo 4) 4)
        (test (tribonacci-memo 4) 4)
        (test (tribonacci-memo 5) 7)))

(define tests-tribonacci
  (list (test (tribonacci 0) 0)
        (test (tribonacci 1) 1)
        (test (tribonacci 2) 1)
        (test (tribonacci 3) 2)
        (test (tribonacci 4) 4)
        (test (tribonacci 4) 4)
        (test (tribonacci 5) 7)))

(run-tests tests-tribonacci-memo)
(run-tests tests-tribonacci)

;; Test 4
(define tests-my-if
  (list (test (my-if #t 1 (/ 1 0)) 1)
        (test (my-if #f (/ 1 0) 1) 1)
```

```
        (test (my-if (= (+ 1 1) 2) 2 (/ 1 0)) 2)))

(run-tests tests-my-if)

;; Test 5
(define tests-my-let
  (list
   (test (my-let ((x 5) (y 7))
                 (+ x 1 y))
         13)
   (test (my-let ((=> #f))
                 (cond (#t => 'ok)))
         ok)))

(define tests-my-let*
  (list
   (test (my-let* ((x 5) (y (+ x 7)))
                  y) 12)))

(run-tests tests-my-let)
(run-tests tests-my-let*)
```

Вывод

Ознакомиться со средствами метапрограммирования («код как данные», макросы) и подходами к оптимизации вычислений (мемоизация результатов вычислений, отложенные вычисления) на примере языка Scheme . Ознакомился с типичным применением программирования с использованием продолжений.