Лабораторная работа №4

Table of Contents

1. Цели 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.

2. Задания

2.1. 1. Продолжения.

Утверждение (assertion) — проверка на истинность некоторого условия, заданного программистом. По традиции осуществляется процедурой (функцией) с именем `assert`. Включается в код во время написания кода и отладки с целью установки ограничений на значения и выявления недопустимых значений. Если в процессе выполнения программы указанное условие нарушается, то программа завершается с выводом диагностического сообщения о том, какое условие было нарушено. Если условие соблюдено, то выполнение программы продолжается, никаких сообщений не выводится.

Реализуйте каркас (фреймворк) для отладки с помощью утверждений. Пусть Ваш каркас перед использованием инициализируется вызовом `(use-assertions)`, а сами утверждения записываются в коде ниже в виде `(assert <statement>)`. Если условие не выполнено, происходит завершение работы программы без возникновения ошибки выполнения и вывод в консоль диагностического сообщения вида `FAILED: <statement>`. Пример использования каркаса:

```
(use-assertions) ; Инициализация вашего каркаса перед использованием

; 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
```

Сообщение, которое должно быть выведено при выполнении примера, показанного выше:

```
FAILED: (not (zero? x))
```

Важно! Если в программе используются гигиенические макросы и эта программа будет выполнена в среде guile 1.8.x (в том числе на сервере тестирования), то следует подключить модуль поддержки таких макросов, написав в начале программы следующую строку:

```
  (use-syntax (ice-9 syncase))
```

2.1.1. решение

```
(define (exit) (display "please call (use-assertions) before using (assert ...)"))

(define-syntax use-assertions
  (syntax-rules ()
    ((use-assertions)
     (call-with-current-continuation
       (lambda (cont)
         (set! exit cont))))))

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

2.1.2. тесты

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

(use-assertions)
```

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

(1 1/2 1/3 1/4 1/5)

```
(map 1/x '(-2 -1 0 1 2)) ; ВЫВЕДЕТ в консоль сообщение и завершит работу программы
```

2.2. 2. Код как данные. Порты ввода-вывода.

- *Сериализация данных*. Реализуйте процедуры для записи данных из переменной в файл по заданному пути (т.е. для сериализации) и последующего чтения данных (десериализации) из такого файла:

```
(save-data данные путь-к-файлу)
(load-data путь-к-файлу) ⇒ данные
```

- решение

```
(define (save-data data path)
(with-output-to-file path (lambda ()
                            (write data))))

(define (load-data path)
(with-input-from-file path (lambda ()
                            (read))))
```

- *Подсчет строк в текстовом файле.* Реализуйте процедуру, принимающую в качестве аргумента путь к текстовому файлу и возвращающую число *непустых* строк в этом файле. Используйте процедуры, разработанные вами ранее в рамках выполнения домашних заданий.

  - решение

```
(define (file-nonempty-lines-count path)
  (with-input-from-file path (lambda ()
                               (let loop ((prev #\newline)
                                          (k 0))
                                 (let ((cur (read-char)))
                                   (if (eof-object? cur)
                                       (if (equal? #\newline prev)
                                           k
                                           (+ k 1))
                                       (if (and (equal? #\newline cur)
                                                (not (equal? #\newline prev)))
                                           (loop cur (+ k 1))
                                           (loop cur k)))))))))
```

  - тесты

```
(display (file-nonempty-lines-count "./lab4.org"))
```

```
688
```

2.3. 3. Мемоизация результатов вычислений.

Реализуйте функцию вычисления n-го “числа трибоначчи” (последовательности чисел, которой первые три числа равны соответственно 0, 0 и 1, а каждое последующее число — сумме предыдущих трех чисел):

$$t(n) = \begin{cases} 0, & n \leqslant 1; \\ 1, & n = 2; \\ t(n-1) + t(n-2) + t(n-3), & n > 2; \end{cases}$$

Figure 1: Функция

$$n \in \mathbb{Z}, n \geqslant 0.$$

Figure 2: Область определения функции

Реализуйте версию этой функции с мемоизацией результатов вычислений. Сравните время вычисления значения функций для разных (умеренно больших) значений её аргументов без мемоизации и с мемоизацией. Для точного измерения вычисления рекомендуется использовать команду REPL Guile `time` (Guile 2.x).

2.3.1. Решение без мемоизации

```
(define (tribonacci n)
  (cond ((= n 0) 0)
        ((<= n 2) 1)
        (else (let loop ((n (- n 2))
                         (first 0)
                         (second 1)
                         (third 1))
                (if (= n 0)
                    third
                    (loop (- n 1) second third (+ first second third))))))))
```

1. Тесты

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

```
(define tests
  (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)
(use-modules (ice-9 time))
(time (tribonacci 9000))
(time (tribonacci 90000))
(time (tribonacci 90050))
```

```
Test 1: (tribonacci 0) ok
Test 2: (tribonacci 1) ok
Test 3: (tribonacci 2) ok
Test 4: (tribonacci 3) ok
Test 5: (tribonacci 4) ok
Test 6: (tribonacci 4) ok
Test 7: (tribonacci 5) ok
clock utime stime cutime cstime gctime
 0.01  0.04  0.00   0.00   0.00   0.04
clock utime stime cutime cstime gctime
 0.59  2.90  0.07   0.00   0.00   2.77
clock utime stime cutime cstime gctime
 0.48  2.23  0.03   0.00   0.00   2.09
```

2.3.2. Решение с мемоизацией

```
(define tribonacci-memo
  (let ((known-results '()))
    (lambda (n)
      (let* ((args n)
             (res (assoc args known-results)))
        (if res
            (cadr res)
            (let ((res (cond ((= n 0) 0)
                             ((<= n 2) 1)
```

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

1. Тесты

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

```
(define tests
  (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)))

(run-tests tests)
(use-modules (ice-9 time))
(time (tribonacci-memo 9000))
(time (tribonacci-memo 90000))
(time (tribonacci-memo 90050))
```

```
Test 1: (tribonacci-memo 0) ok
Test 2: (tribonacci-memo 1) ok
Test 3: (tribonacci-memo 2) ok
Test 4: (tribonacci-memo 3) ok
Test 5: (tribonacci-memo 4) ok
Test 6: (tribonacci-memo 4) ok
Test 7: (tribonacci-memo 5) ok
clock utime stime cutime cstime gctime
 0.06  0.05  0.00   0.00   0.00   0.00
clock utime stime cutime cstime gctime
 7.29  7.71  0.09   0.00   0.00   0.83
clock utime stime cutime cstime gctime
 0.02  0.01  0.00   0.00   0.00   0.00
```

2.4. 4. Отложенные вычисления.

Используя примитивы для отложенных вычислений `delay` и `force`, реализуйте макрос `my-if`, который полностью воспроизводит поведение встроенной условной конструкции (специальной формы) `if` для выражений, возвращающих значения. Например, такие примеры должны вычисляться корректно:

```
(my-if #t 1 (/ 1 0)) ⇒ 1
(my-if #f (/ 1 0) 1) ⇒ 1
```

Запрещается использовать встроенные условные конструкции `if`, `cond`, `case` и перехват исключений.

2.4.1. Решение

```
(define-syntax my-if
  (syntax-rules ()
    ((my-if condition statement1 statement2)
     (let ((promise1 (delay statement1))
           (promise2 (delay statement2)))
       (force (or (and condition promise1) promise2))))))
```

2.4.2. Тесты

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

```
Test 1: (my-if #t 1 (/ 1 0)) ok
Test 2: (my-if #f (/ 1 0) 1) ok
Test 3: (my-if (= (+ 1 1) 2) 2 (/ 1 0)) ok
```

2.5. 5. Локальные определения.

Реализуйте макросы `my-let` и `my-let*`, полностью воспроизводящие поведение встроенных макросов `let` и `let*`.

2.5.1. Решение

1. let

   первое приближение:

```
(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))))
```

   второе приближение:

```
(define-syntax my-let
  (syntax-rules ()
    ((my-let ((var val) ...)
             expr)
     ((lambda (var ...)
        expr)
      val ...))))
```

2. let*

```
(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))))
```

2.5.2. Тесты

1. let

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

(run-tests tests)
```

```
Test 1: (my-let ((x 5) (y 7)) (+ x 1 y)) ok
Test 2: (my-let ((=> #f)) (cond (#t => (quote ok)))) ok
```

2. let*

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

(run-tests tests)
```

```
Test 1: (my-let* ((x 5) (y (+ x 7))) y) ok
```

2.6. 6. Управляющие конструкции.

Используя *гигиенические* макросы языка Scheme, реализуйте управляющие конструкции, свойственные императивным языкам программирования.

2.6.1. A. Условия *when* и *unless*

Напишите макросы:

- *(when cond? expr1 expr2 … exprn)*, который *выполняет* последовательность выражений *expr1 expr2 … exprn*, если условие *cond?* истинно.

- *(unless cond? expr1 expr2 … exprn)*, который *выполняет* последовательность выражений *expr1 expr2 … exprn*, если условие *cond?* ложно.

Предполагается, что *when* и *unless* возвращают результат последнего вычисленного в них выражения. *When* и *unless* могут быть вложенными.

Пример:

```
; Пусть x = 1
;
(when   (> x 0) (display "x > 0")  (newline))
(unless (= x 0) (display "x != 0") (newline))
```

В стандартный поток будет выведено:

```
x > 0
x != 0
```

1. Решение

```
(define-syntax when
  (syntax-rules ()
    ((when test expr ...)
     (if test
         (begin expr ...)))))

(define-syntax unless
  (syntax-rules ()
    ((unless test expr ...)
     (if (not test)
         (begin expr ...)))))
```

2. Тесты

```
(define x 1)
(when   (> x 0) (display "x > 0")  (newline))
(unless (= x 0) (display "x != 0") (newline))
```

```
x > 0
x != 0
```

2.6.2. Б. Циклы *for*

Реализуйте макрос *for*, который позволит организовывать циклы с переменной — параметром цикла. Определение должно допускать две различных формы записи:

- *(for x in xs expr1 expr2 … exprn)* и
- *(for xs as x expr1 expr2 … exprn),*

где *x* — переменная, *xs* — список значений, которые должна принимать, переменная на каждой итерации, *expr1 expr2 … exprn* — последовательность инструкций, которые должны быть выполнены в теле цикла.

Примеры применения:

```
(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. Решение

```
(define-syntax for
  (syntax-rules (in as)
    ((for x in xs expr ...)
     (let loop ((values xs))
       (if (not (null? values))
           (let ((x (car values)))
             (begin expr ...)
             (loop (cdr values))))))
    ((for xs as x expr ...)
     (for x in xs expr ...))))
```

2. Тесты

```
(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)
```

2.6.3. B. Цикл *while*

Реализуйте макрос *while*, который позволит организовывать циклы с предусловием:

(while cond? expr1 expr2 … exprn),

где *cond?* — условие, *expr1 expr2 … exprn* — последовательность инструкций, которые должны быть выполнены в теле цикла. Проверка условия осуществляется перед каждой итерацией, тело цикла выполняется, если условие выполняется. Если при входе в цикл условие не выполняется, то тело цикла не будет выполнено ни разу.

Пример применения:

```
(let ((p 0)
      (q 0))
  (while (< p 3)
         (set! q 0)
```

```
          (while (< q 3)
                 (display (list p q))
                 (newline)
                 (set! q (+ q 1)))
          (set! p (+ p 1))))
```

Выведет:

```
(0 0)
(0 1)
(0 2)
(1 0)
(1 1)
(1 2)
(2 0)
(2 1)
(2 2)
```

Рекомендация. Целесообразно разворачивать макрос в вызов анонимной процедуры без аргументов со статической переменной, содержащей анонимную процедуру с проверкой условия, рекурсивным вызовом и телом цикла. Для краткой записи такой процедуры и ее вызова можно использовать встроенную конструкцию *letrec*, которая аналогична *let* и let*, но допускает рекурсивные определения, например:

```
(letrec ((iter (lambda (i)
                 (if (= i 10)
                     '()
                     (cons i (iter (+ i 1)))))))
  (iter 0))
  => (0 1 2 3 4 5 6 7 8 9)
```

1. Решение

```
(define-syntax while
  (syntax-rules ()
    ((while cond? expr ...)
     (let loop ()
       (if cond?
           (begin expr ...
                  (loop)))))))
```

2. Тесты

```
(let ((p 0)
      (q 0))
  (while (< p 3)
    (set! q 0)
    (while (< q 3)
      (display (list p q))
      (newline)
      (set! q (+ q 1)))
    (set! p (+ p 1))))
```

```
(0 0)
(0 1)
(0 2)
(1 0)
(1 1)
(1 2)
(2 0)
(2 1)
(2 2)
```

2.6.4. Г. Цикл *repeat..until*

Реализуйте макрос *repeat..until*, который позволит организовывать циклы с предусловием:

(repeat (expr1 expr2 … exprn) until cond?),

где *cond?* — условие, *expr1 expr2 … exprn* — последовательность инструкций, которые должны быть выполнены в теле цикла. Проверка условия осуществляется после каждой итерации. Если условие возвращает истину, цикл завершается, иначе цикл выполняется снова. Таким образом, тело цикла выполняется по меньшей мере 1 раз.

Например:

```
(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)))
```

Выведет:

```
(0 0)(0 1)(0 2)
(1 0)(1 1)(1 2)
(2 0)(2 1)(2 2)
```

1. Решение

```
(define-syntax repeat
  (syntax-rules (until)
    ((repeat (expr ...) until cond?)
     (let loop ()
       (begin expr ...
              (if (not cond?) (loop)))))))
```

2. Тесты

```
(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)
```

3. Без тела цикла в скобках

Подумайте, зачем требуется заключать тело цикла в круглые скобки? Как изменится макрос, если отказаться от этих скобок?

```
(define-syntax repeat
  (syntax-rules (until)
    ((repeat expr ... until cond?)
     (let loop ()
       (begin expr ...
```

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

(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)
```

2.6.5. Д. Вывод «в стиле C++»

Реализуйте макрос для последовательного вывода значений в стандартный поток вывода вида:

```
(cout << "a = " << 1 << endl << "b = " << 2 << endl)
```

Здесь *cout* — имя макроса, указывающее, что будет осуществляться вывод в консоль (от console output), символы << разделяют значения, *endl* означает переход на новую строку.

Данный пример выведет следующий текст:

```
a = 1
b = 2
```

1. Решение

```
(define-syntax cout
  (syntax-rules (<< endl)
    ((cout << endl)
     (newline))
    ((cout << elem)
     (display elem))
    ((cout << endl others ...)
     (begin (newline)
```

```
            (cout others ...)))
    ((cout << elem others ...)
     (begin (display elem)
            (cout others ...)))))
```

2. Тесты

```
(cout << "a = " << 1 << endl << "b = " << 2 << endl)
```

```
a = 1
b = 2
```

Author: Starovoytov Alexandr
Created: 2021-11-28 Sun 23:00