Лабораторная работа № 3. Типы данных. Модульное тестирование

12 декабря 2023 г.

Семён Чайкин, ИУ9-11Б

Цели работы

- На практике ознакомиться с системой типов языка `Scheme`.
- На практике ознакомиться с юнит-тестированием.
- Разработать свои средства отладки программ на языке `Scheme`.
- На практике ознакомиться со средствами метапрограммирования языка `Scheme`.

Задание 1

Реализуйте макрос `trace` для трассировки. Трассировка — способ отладки, при котором отслеживаются значения переменных или выражений на каждом шаге выполнения программы. Необходимость и вывести значение в консоль, и вернуть его в программу нередко требует существенной модификации кода, что может стать источником дополнительных ошибок. Реализуйте макрос, который позволяет ценой небольшой вставки, не нарушающей декларативность кода, выполнить и вывод значения в консоль с комментарием в виде текста выражения, которое было вычислено, и возврат его значения в программу.

Реализация

```
(define-syntax trace
  (syntax-rules ()
    ((trace x)
     ((lambda ()
        (write 'x)
        (display " => ")
        (define v x)
        (write v)
        (newline)
        v)))))
```

Тестирование

```
> (define (zip . xss)
    (if (or (null? xss)
            (null? (trace (car xss))))
        '()
        (cons (map car xss)
              (apply zip (map cdr (trace xss))))))

> (zip '(1 2 3) '(one two three))
(car xss) => (1 2 3)
xss => ((1 2 3) (one two three))
(car xss) => (2 3)
xss => ((2 3) (two three))
(car xss) => (3)
xss => ((3) (three))
(car xss) => ()
((1 one) (2 two) (3 three))
```

Задание 2

Юнит-тестирование — способ проверки корректности отдельных относительно независимых частей программы. При таком подходе для каждой функции (процедуры) пишется набор тестов — пар “выражение — значение, которое должно получиться”. Процесс тестирования заключается в вычислении выражений тестов и автоматизированном сопоставлении результата вычислений с ожидаемым результатом. При несовпадении выдается сообщение об ошибках. Реализуйте свой каркас для юнит-тестирования. Пусть каркас включает следующие компоненты:

- Макрос `test` — конструктор теста вида (выражение ожидаемый-результат).
- Процедуру `run-test`, выполняющую отдельный тест. Если вычисленный результат совпадает с ожидаемым, то в консоль выводятся выражение и признак того, что тест пройден. В противном случае выводится выражение, признак того, что тест не пройден, а также ожидаемый и фактический результаты. Функция возвращает `#t`, если тест пройден и `#f` в противном случае. Вывод цитаты выражения в консоль должен выполняться до вычисления его значения, чтобы при аварийном завершении программы последним в консоль было бы выведено выражение, в котором произошла ошибка.
- Процедуру `run-tests`, выполняющую серию тестов, переданную ей в виде списка. Эта процедура должна выполнять все тесты в списке и возвращает `#t`, если все они были успешными, в противном случае процедура возвращает `#f`.

Реализация

```
(define-syntax test
  (syntax-rules ()
    ((test expr correct) (list 'expr correct))))

(define (run-test test)
  (write (car test))
  (define v (eval (car test) (interaction-environment)))
  (if (equal? v (cadr test))
      (begin
        (display " ok")
        (newline)
        #t)
      (begin
        (display " FAIL")
        (newline)
        (display "  Expected: ")
        (write (cadr test))
        (newline)
        (display "  Returned: ")
        (write v)
        (newline)
        #f)))

(define (run-tests tests)
  (define (run-tests-rec tests fl)
    (if (null? tests)
        fl
        (run-tests-rec (cdr tests)
                       (and (run-test (car tests)) fl))))
  (run-tests-rec tests #t))
```

Тестирование

Пример тестирования:

```
(define counter
  (let ((n 0))
    (lambda ()
      (set! n (+ 1 n))
      n)))
```

```
(define counter-tests
  (list (test (counter) 2)
        (test (counter) 7)
        (test (counter) 4)))

(trace (counter))
(run-tests counter-tests)
```

Результат в консоли:

```
(counter) => 1
1
(counter) ok
(counter) FAIL
  Expected: 7
  Returned: 3
(counter) ok
#f
```

Задание 3

Реализуйте процедуру доступа к произвольному элементу последовательности (правильного списка, вектора или строки) по индексу. Пусть процедура возвращает `#f` если получение элемента не возможно.

Реализуйте процедуру “вставки” произвольного элемента в последовательность, в позицию с заданным индексом (процедура возвращает новую последовательность). Пусть процедура возвращает `#f` если вставка не может быть выполнена.

Реализация

```
(define (ref array index . value)
  (define (f list index)
    (and (not (null? list))
         (if (= index 0)
             (car list)
             (f (cdr list) (- index 1)))))
  (define (g-rec list index value)
    (if (= index 0)
        (cons value list)
        (cons (car list) (g-rec (cdr list) (- index 1) value))))
  (define (g list index value . type)
    (and (not (> index (length list)))
         (if (null? type)
             (g-rec list index value)
             ((car type) (g-rec list index value)))))
  (if (null? value)
      (cond
        ((list? array) (f array index))
        ((vector? array) (f (vector->list array) index))
        ((string? array) (f (string->list array) index))
        (else #f))
      (cond
        ((list? array) (g array index (car value)))
        ((vector? array) (g (vector->list array) index (car value) list->vector))
        ((string? array) (and (char? (car value))
                              (g (string->list array) index (car value) list->string)))
        (else #f))))
```

Тестирование

Пример тестирования:

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

(define ref-tests
  (list (test (ref '(1 2 3) 1) 2)
        (test (ref #(1 2 3) 1) 2)
        (test (ref "123" 1) #\2)
        (test (ref "123" 3) #f)
        (test (ref '(1 2 3) 1 0) '(1 0 2 3))
        (test (ref #(1 2 3) 1 0) #(1 0 2 3))
        (test (ref "123" 1 #\0) "1023")
        (test (ref "123" 1 0) #f)
        (test (ref "123" 3 #\4) "1234")
        (test (ref "123" 5 #\4) #f)
        ))

(run-tests ref-tests)
```

Результат в консоли:

```
(ref '(1 2 3) 1) ok
(ref #(1 2 3) 1) ok
(ref "123" 1) ok
(ref "123" 3) ok
(ref '(1 2 3) 1 0) ok
(ref #(1 2 3) 1 0) ok
(ref "123" 1 #\0) ok
(ref "123" 1 0) ok
(ref "123" 3 #\4) ok
(ref "123" 5 #\4) ok
#t
```

Задание 4

Разработайте наборы юнит-тестов и используйте эти тесты для разработки процедуры, выполняющей разложение на множители.

Реализуйте процедуру `factorize`, выполняющую разложение многочленов вида $a^2 - b^2$, $a^3 - b^3$ и $a^3 + b^3$ по формулам.

Пусть процедура принимает единственный аргумент — выражение на языке `Scheme`, которое следует разложить на множители, и возвращает преобразованное выражение. Возведение в степень в исходных выражениях пусть будет реализовано с помощью встроенной процедуры `expt`. Получаемое выражение должно быть пригодно для выполнения в среде интерпретатора с помощью встроенной процедуры `eval`. Упрощение выражений не требуется.

Реализация

```
(define (factorize expr)
  (define +? (equal? (car expr) '+))
  (define expt (car (reverse (cadr expr))))
  (define x (cadr (cadr expr)))
  (define y (cadr (caddr expr)))
  (cond
    ((and (not +?) (= expt 2)) `(* (- ,x ,y) (+ ,x ,y)))
    ((and (not +?) (= expt 3)) `(* (- ,x ,y) (+ (expt ,x 2) (* ,x ,y) (expt ,y 2))))
    ((and  +? (= expt 3)) `(* (+ ,x ,y) (+ (expt ,x 2) (- (* ,x ,y)) (expt ,y 2))))
    (else #f)))
```

Тестирование

Пример тестирования:

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

```
(define factorize-tests
  `(,(test ((lambda ()
              (eval
               `((lambda
                     (x y)
                   ,(factorize '(- (expt x 2) (expt y 2)))) 1 2)
               (interaction-environment)))) -3)
    ,(test ((lambda ()
              (eval
               `((lambda
                     (x y)
                   ,(factorize '(- (expt x 3) (expt (- y 2) 3)))) 1 2)
               (interaction-environment)))) 1)
    ,(test ((lambda ()
              (eval
               `((lambda
                     (x y)
                   ,(factorize '(+ (expt (+ x 12) 3) (expt (- y 2) 3)))) 5 2)
               (interaction-environment)))) 4913)))

(run-tests factorize-tests)
```

Результат в консоли:

```
((lambda () (eval `((lambda (x y)
                   ,(factorize '(- (expt x 2) (expt y 2)))) 1 2)
                   (interaction-environment)))) ok
((lambda () (eval `((lambda (x y)
                   ,(factorize '(- (expt x 3) (expt (- y 2) 3)))) 1 2)
                   (interaction-environment)))) ok
((lambda () (eval `((lambda (x y)
                   ,(factorize '(+ (expt (+ x 12) 3) (expt (- y 2) 3)))) 5 2)
                   (interaction-environment)))) ok
#t
```

Вывод

Узнал про систему типов языка Scheme, узнал про юнит-тестирование, понял как разрабатывать средства отладки и тестирования программ на языке Scheme, а также ознакомился со средствами метапрограммирования языка Scheme.