# Лабораторная работа № 3. Типы данных. Модульное тестирование

7 Октября 2024г.

Александр Яннаев, ИУ9-11Б

## Цель 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.

## Индивидуальный вариант

1. Реализация процедуры доступа к произвольному элементу последовательности (список, вектор или строка) по индексу.
2. Реализация процедуры «вставки» произвольного элемента в последовательность.
3. Реализация процедуры `if->cond`, преобразующей цепочку условных конструкций `if` в `cond`.

## Реализация

```
; Макрос trace позволяет выводить выражение и его значение для отладки.
(define-syntax trace
  (syntax-rules ()
    ((_ expr)
     (let ((value expr))
       (write 'expr)
       (display " => ")
       (display value)
       (newline)
       value))))

; Функция zip обрабатывает несколько списков параллельно.
(define (zip . xss)
```

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

; Макрос test создаёт тесты с указанным выражением и ожидаемым результатом.
(define-syntax test
  (syntax-rules ()
    ((test action res) '(action res))))

; Функция run-test выполняет тест и сравнивает результат с ожидаемым.
(define (run-test test)
  (write (car test))
  (define c-res (eval (car test) (interaction-environment)))
  (if (equal? c-res (cadr test))
      (begin
        (display " ok")
        (newline)
        #t)
      (begin
        (display " FAIL")
        (newline)
        (display "Expected: ")
        (write (cadr test))
        (newline)
        (display "Returned: ")
        (write c-res)
        (newline)
        #f)))

; Функция run-tests выполняет несколько тестов.
(define (run-tests xs)
  (if (null? (cdr xs))
      (run-test (car xs))
      ((lambda (x y)
         (and x y))
       (run-test (car xs))
       (run-tests (cdr xs)))))

; Функция signum определяет знак числа.
(define (signum x)
  (cond
    ((< x 0) -1)
    ((= x 0)  1)
    (else     1)))
```

```
; Список тестов для функции signum.
(define the-tests
  (list (test (signum -2) -1)
        (test (signum  0)  0)
        (test (signum  2)  1)))

; Функция counter возвращает счётчик, который увеличивает значение при каждом вызове.
(define counter
  (let ((n 0))
    (lambda ()
      (set! n (+ n 1))
      n)))

; Список тестов для функции counter.
(define counter-tests
  (list (test (counter) 3)
        (test (counter) 77) ; ошибка
        (test (counter) 5)))

; Функция if->cond преобразует выражение if в cond.
(define (if->cond f)
  (if (and (list? f) (eq? (car f) 'if))
      (let ((condition (cadr f))
            (true-branch (caddr f))
            (false-branch (if (null? (cdddr f)) '() (cadddr f))))
        (if (null? false-branch)
            `(cond (,condition ,true-branch))
            (if (and (list? false-branch) (eq? (car false-branch) 'if))
                `(cond (,condition ,true-branch) ,@(cdr (if->cond false-branch)))
                `(cond (,condition ,true-branch) (else ,false-branch)))))
      f))

; if->cond testing
(define if-cond-tests
  (list
   (test (if->cond '(if (> value 0)
                        increase
                        (if (< value 0)
                            decrease
                            neutral)))
         (cond ((> value 0) increase)
               ((< value 0) decrease)
               (else neutral)))
   (test (if->cond '(if (equal? (car input) 'lambda)
                        (process-lambda input)
```

```
                    (if (equal? (car input) 'define)
                        (process-define input)
                        (if (equal? (car input) 'if)
                            (process-if input)))))
      (cond ((equal? (car input) 'lambda) (process-lambda input))
            ((equal? (car input) 'define) (process-define input))
            ((equal? (car input) 'if) (process-if input))))
(test (if->cond '(if condition1
            action1
    (if condition2
        action2
        (if condition3
            action3
        fallback))))
      (cond (condition1 action1)
            (condition2 action2)
            (condition3 action3)
            (else fallback)))
(test (if->cond '(if condition1
            action1
    (if condition2
        action2
        (if condition3
            action3))))
      (cond (condition1 action1)
            (condition2 action2)
            (condition3 action3)))
(test (if->cond '(if (> value 0)
            (display positive)
    (if (= value 0)
        (display zero)
        (display negative))))
      (cond ((> value 0) (display positive))
            ((= value 0) (display zero))
            (else (display negative))))
(test (if->cond '(if (< delta 0)
            (create-list)
    (if (= delta 0)
        (create-list x)
        (create-list x1 x2))))
      (cond ((< delta 0) (create-list))
            ((= delta 0) (create-list x))
            (else (create-list x1 x2))))))

; Функция ref получает или изменяет элемент в списке, векторе или строке по индексу.
(define (ref elem idx . new-elem)
```

```
  (define (insert lst i val)
    (if (= i 0)
        (cons val lst)
        (cons (car lst) (insert (cdr lst) (- i 1) val))))

  (and
   (>= idx 0)
   (cond
     ((pair? elem)
      (if (null? new-elem)
          (and (< idx (length elem)) (list-ref elem idx))
          (and (<= idx (length elem))
               (insert elem idx (car new-elem)))))
     ((vector? elem)
      (if (null? new-elem)
          (and (< idx (vector-length elem)) (vector-ref elem idx))
          (let ((lst-rep (vector->list elem)))
            (and
             (<= idx (length lst-rep))
             (let ((res-vec (list->vector (insert lst-rep idx (car new-elem)))))
               res-vec)))))
     ((string? elem)
      (if (null? new-elem)
          (and (< idx (string-length elem)) (string-ref elem idx))
          (and
           (char? (car new-elem))
           (let ((lst-rep (string->list elem)))
             (and
              (<= idx (length lst-rep))
              (let ((res-str (list->string (insert lst-rep idx (car new-elem)))))
                res-str)))))))))

; ref testing
(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)))
```

```
; Тестирование работы счётчика с помощью trace.
(+ (trace (counter))
   (trace (counter)))

; Запуск всех тестов.
(zip '(1 2 3) '(one two three))
(run-tests the-tests)
(run-tests counter-tests)
(run-tests ref-tests)
(run-tests if-cond-tests)
```

## Тестирование

```
Welcome to DrRacket, version 8.7 [3m].
Language: R5RS; memory limit: 128 MB.
(counter) => 1
(counter) => 2
3
(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))
(signum -2) ok
(signum 0) FAIL
Expected: 0
Returned: 1
(signum 2) ok
#f
(counter) ok
(counter) FAIL
Expected: 77
Returned: 4
(counter) ok
#f
(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
(if->cond '(if (> value 0) increase (if (< value 0) decrease neutral))) ok
(if->cond '(if (equal? (car input) 'lambda) (process-lambda input) (if (equal? (car input) 'defi
(process-define input) (if (equal? (car input) 'if) (process-
if input))))) ok
(if->cond '(if condition1 action1 (if condition2 action2 (if condition3 action3 fallback)))) ok
(if->cond '(if condition1 action1 (if condition2 action2 (if condition3 action3)))) ok
(if->cond '(if (> value 0) (display positive) (if (= value 0) (display zero) (display negative)))
(if->cond '(if (< delta 0) (create-list) (if (= delta 0) (create-
list x) (create-list x1 x2)))) ok
#t
>
```

## Вывод

Во время выполнения этой работы я лучше познакомился с возможностями языка Scheme, особенно с тем, как отлаживать программы и проводить модульное тестирование. Одной из ключевых частей было создание макроса `trace`, который позволяет отслеживать значения переменных в программе, выводя их на экран. Это очень удобно для поиска ошибок, так как выражения вычисляются только один раз, что снижает риск новых ошибок. Также была реализована система для тестирования функций, включая процедуры `run-test` и `run-tests`, которые проверяют, правильно ли работает программа, и выводят результаты тестов в удобной форме. Это помогло автоматизировать процесс поиска ошибок и упростить проверку работы функций. Кроме того, я разработал процедуру для доступа и изменения элементов в списках, векторах и строках. Она универсальна и работает с разными типами данных, возвращая результат или сообщение об ошибке, если операция невозможна. Также я создал процедуру `if->cond`, которая преобразует условные конструкции `if` в `cond`. Это позволило лучше понять, как работает структура условных выражений в Scheme, и использовать более оптимальные способы их написания. Модульное тестирование и трассировка сильно помогли сделать программу более надёжной и удобной для отладки. Эти приёмы будут полезны и в будущем при решении других задач на языке Scheme.