Лабораторная работа № 3. Типы данных. Модульное тестирование

14 октября 2024 г.

Булат Валиуллин, 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

ref, if->cond

Реализация

```
(define (hello name)
  (display "Привет, ")
  (display name)
  (display "!\n"))(define-syntax trace
  (syntax-rules ()
    ((trace expr)
     (let ((val expr))
       (write 'expr)
       (display " => ")
       (write val)
       (newline)
       val)
     )))


(define-syntax test
```

```
  (syntax-rules ()
    ((test call exp) (list 'call exp))))

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

(define (run-tests tests)
  (define (loop tests allpassed?)
    (if (null? tests)
        allpassed?
        (if (run-test (car tests))
            (loop (cdr tests) allpassed?)
            (loop (cdr tests) #f))))
  (loop tests #t))


(define (ref seq num . ins)
  (if (null? ins)
      (cond
        ((list? seq) (and (< num (length seq)) (list-ref seq num)))
        ((vector? seq) (and (< num (vector-length seq)) (vector-ref seq
                                                                    num)))
        ((string? seq) (and (< num (string-length seq)) (string-ref seq num))))
      (cond
        ((vector? seq)
         (let ((list-seq (vector->list seq)))
           (list->vector (ref list-seq num (car ins)))))
        ((string? seq)
         (or
```

```
          (and (char? (car ins)) (= (string-length seq) num)
               (string-append seq (string (car ins))))
          (and (char? (car ins))
               (not (= (string-length seq) num))
               (<= 0 num)
               (<= num (string-length seq))
               (let ((list-seq (string->list seq)))
                 (list->string (ref list-seq num (car ins)))))))
        ((list? seq)
         (if (and (<= 0 num) (<= num (length seq)))
             (cond
               ((null? seq) (list ins))
               ((= num 0) (cons (car ins) seq))
               (else (cons (car seq) (ref (cdr seq) (- num 1) (car ins))))))))))


(define (if->cond chain . nested?)
  (if (and (list? chain) (eq? (car chain) 'if))
      (let* ((condit (cadr chain))
             (true_branch (caddr chain))
             (false_branch (if (null? (cdddr chain)) #f (cadddr chain)))
             (cond-statement (list condit true_branch)))
        (if (eq? nested? '())
            (cons 'cond
                  (if (not false_branch)
                      (list cond-statement)

                      (if (and (list? false_branch) (eq? (car false_branch)
                                                         'if))
                          (cons cond-statement
                                (if->cond false_branch #t))
                          (cons cond-statement
                                (list (cons 'else (cons false_branch
                                                        '()))))))))
            (cond
              ((not false_branch) (cons cond-statement '()))

              ((and (list? false_branch) (eq? (car false_branch) 'if))
               (cons cond-statement
                     (if->cond false_branch #t)))
              (else
               (cons cond-statement
                     (list (cons 'else (cons false_branch '())))))))))
```

Тестирование

```
Language: R5RS; memory limit: 128 MB.
> (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))
> (define (signum x)
  (cond
    ((< x 0) -1)
    ((= x 0)  1) ; Ошибка здесь!
    (else     1)))
> (define the-tests
  (list (test (signum -2) -1)
        (test (signum  0)  0)
        (test (signum  2)  1)))
> (run-tests the-tests)
(signum -2) ok
(signum 0) FAIL
  Expected: 0
  Returned: 1
(signum 2) ok
#f
> (ref '(1 2 3) 1)
2
> (ref #(1 2 3) 1)
2
> (ref "123" 1)
#\2
> (ref "123" 3)
#f
> (ref '(1 2 3) 1 0)
(1 0 2 3)
> (ref #(1 2 3) 1 0)
#(1 0 2 3)
```

```
> (ref #(1 2 3) 1 #\0)
#(1 #\0 2 3)
> (ref "123" 1 #\0)
"1023"
> (ref "123" 1 0)
#f
> (ref "123" 3 #\4)
"1234"
> (ref "123" 5 #\4)
#f
> (if->cond '(if (> x 0)
               +1
               (if (< x 0)
                   -1
                   0)))
(cond ((> x 0) 1) ((< x 0) -1) (else 0))
> (if->cond '(if (equal? (car expr) 'lambda)
               (compile-lambda expr)
               (if (equal? (car expr) 'define)
                   (compile-define expr)
                   (if (equal (car expr) 'if)
                       (compile-if expr)))))
(cond
 ((equal? (car expr) 'lambda) (compile-lambda expr))
 ((equal? (car expr) 'define) (compile-define expr))
 ((equal (car expr) 'if) (compile-if expr)))
> (if->cond '(if a
               aa
       (if b
           bb
           (if c
               cc
           dd))))
(cond (a aa) (b bb) (c cc) (else dd))
> (if->cond '(if a
               aa
       (if b
           bb
           (if c
               cc))))

(cond (a aa) (b bb) (c cc))
> (if->cond '(if (> x 0)
               (dislay '+)
       (if (= x 0)
           (display 0)
```

```
           (display '-))))
(cond ((> x 0) (dislay '+)) ((= x 0) (display 0)) (else (display '-)))
> (if->cond '(if (< d 0)
                 (list)
       (if (= d 0)
           (list x)
           (list x1 x2))))
(cond ((< d 0) (list)) ((= d 0) (list x)) (else (list x1 x2)))
> (define counter
  (let ((n 0))
    (lambda ()
      (set! n (+ n 1))
      n)))
> (+ (trace (counter))
   (trace (counter)))
(counter) => 1
(counter) => 2
3
> (define counter-tests
  (list (test (counter) 3)
        (test (counter) 77) ; ошибка
        (test (counter) 5)))
> (run-tests counter-tests)
(counter) ok
(counter) FAIL
  Expected: 77
  Returned: 4
(counter) ok
#f
```

Вывод

Даа, это были вовсе не скучные 3 недели, которые уж точно запомнятся. Запомнятся, конечно, не только знакомством с системой типов, юнит-тестированием, разработкой средств отладки программ, средствами метапрограммирования, но и незабываемыми эмоциями от “Что?! От меня правда ожидается, что я напишу это?!”, “Да что ж такое, почему опять скобка не там появляется?”, “Ах да, не стоит ведь возвращать логические значения из условий, это будет интересный опыт написать всё через and и or”, до “Что, серьёзно, кажется, или оно наконец работает взрывмозга.png!”.