Лабораторная работа № 5. Интерпретатор стекового языка программирования

30 ноября 2023 г.

Иван Гордеев, Семён Чайкин ИУ9-11Б

Цель работы

Создать интерпретатор стекового языка программирования

Реализация

```scheme
(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)
       0)
     (begin
       (display " FAIL")
       (newline)
       (display "  Expected: ")
       (write (cadr test))
       (newline)
       (display "  Returned: ")
       (write v)
       (newline)
       1)))

(define (run-tests tests)
 (define (run-tests-rec tests)
   (if (null? tests)
       0
       (+ (run-test (car tests)) (run-tests-rec (cdr tests)))))
 (= (run-tests-rec tests) 0))


(define (->tf v) (if v -1 0))
(define (tf-> v) (= v -1))

(define (interpret-rec program program-size stack ip ret table skip-define? skip-if?)
 (if (< ip program-size)
     (begin
       (let ((term (vector-ref program ip)))
         (cond
           ((equal? term 'define)
            (begin
              (set! skip-define? #t)
              (set! table (cons (list (vector-ref program (+ ip 1)) (+ ip 1)) table))))
           ((equal? term 'end)
            (begin
              (set! skip-define? #f)
              (and (not (null? ret))
                   (begin
                     (set! ip (car ret))
                     (set! ret (cdr ret))))))
           ((not skip-define?)
            (cond
              ((equal? term 'if)
               (begin
                 (set! skip-if? (not (tf-> (car stack))))
                 (set! stack (cdr stack))))
              ((equal? term 'endif)
               (begin
                 (set! skip-if? #f)))
              ((not skip-if?)
               (cond
                 ((number? term) (set! stack (cons term stack)))
                 ((equal? term '+) (set! stack (cons (+ (cadr stack) (car stack)) (cddr stack))))
                 ((equal? term '-) (set! stack (cons (- (cadr stack) (car stack)) (cddr stack))))
                 ((equal? term '*) (set! stack (cons (* (cadr stack) (car stack)) (cddr stack))))
                 ((equal? term '/) (set! stack (cons (/ (cadr stack) (car stack)) (cddr stack))))
                 ((equal? term 'mod)
                   (set! stack (cons (modulo (cadr stack) (car stack)) (cddr stack))))
                 ((equal? term 'neg) (set! stack (cons (- (car stack)) (cdr stack))))
                 ((equal? term '=)
                  (set! stack (cons (->tf (= (cadr stack) (car stack))) (cddr stack))))
                 ((equal? term '>)
                  (set! stack (cons (->tf (> (cadr stack) (car stack))) (cddr stack))))
                 ((equal? term '<)
                  (set! stack (cons (->tf (< (cadr stack) (car stack))) (cddr stack))))
                 ((equal? term 'dup) (set! stack (cons (car stack) stack)))
                 ((equal? term 'swap)
                  (set! stack (append (list (cadr stack) (car stack)) (cddr stack))))
                 ((equal? term 'over) (set! stack (cons (cadr stack) stack)))
                 ((equal? term 'rot) (set! stack
                  (append (list (caddr stack) (cadr stack) (car stack)) (cdddr stack))))
                 ((equal? term 'depth) (set! stack (cons (length stack) stack)))
                 ((equal? term 'not) (set! stack (cons (->tf (zero? (car stack) )) (cdr stack))))
                 ((equal? term 'and) (set! stack (cons
                 (->tf (and (not (zero? (car stack))) (not (zero? (cadr stack))))) (cddr stack))))
                 ((equal? term 'or) (set! stack (cons
                 (->tf (or (not (zero? (car stack))) (not (zero? (cadr stack))))) (cddr stack))))
                 ((equal? term 'exit)
                  (begin
                    (set! ip (car ret))
                    (set! ret (cdr ret))))
                 ((equal? term 'drop) (set! stack (cdr stack)))
                 (else
                  (begin
                    (set! ret (cons ip ret))
                    (set! ip (cadr (assoc term table))))))))))
       (interpret-rec program program-size stack (+ ip 1) ret table skip-define? skip-if?))
     stack))


(define (interpret program stack)
 (interpret-rec program (vector-length program) stack 0 '() '() #f #f))

(define tests (list
              (test (interpret #(2 3 * 4 5 * +) '()) '(26))
              (test (interpret #( define abs
                                   dup 0 <
                                   if neg exit endif
                                   end
                                   9 abs
                                   -9 abs
                                   )
                               '()) '(9 9))
              (test (interpret #(neg) '(-9)) '(9))
              (test (interpret #(dup 0) '(-9)) '(0 -9 -9))
              (test (interpret #(1 not 0 not or) '()) '(-1))
              (test (interpret #(0 not 0 not and) '()) '(-1))
              (test (interpret #(   define abs
                                     dup 0 <
                                     if neg endif
                                     end
                                     9 abs
                                     -9 abs      ) (quote ())) '(9 9))
              (test (interpret #(   define =0? dup 0 = end
                                     define <0? dup 0 < end
                                     define signum
                                     =0? if exit endif
                                     <0? if drop -1 exit endif
                                     drop
                                     1
                                     end
                                     0 signum
                                     -5 signum
                                     10 signum      ) (quote ())) '(1 -1 0))
              (test (interpret #(   define -- 1 - end
                                     define =0? dup 0 = end
                                     define =1? dup 1 = end
                                     define factorial
                                     =0? if drop 1 exit endif
                                     =1? if drop 1 exit endif
                                     dup --
                                     factorial
                                     *
                                     end
                                     0 factorial
                                     1 factorial
                                     2 factorial
                                     3 factorial
                                     4 factorial    ) (quote ())) '(24 6 2 1 1))
              (test (interpret #(   define =0? dup 0 = end
                                     define =1? dup 1 = end
                                     define -- 1 - end
                                     define fib
                                     =0? if drop 0 exit endif
                                     =1? if drop 1 exit endif
                                     -- dup
                                     -- fib
                                     swap fib
                                     +
                                     end
                                     define make-fib
                                     dup 0 < if drop exit endif
                                     dup fib
                                     swap --
                                     make-fib
                                     end
                                     10 make-fib    ) (quote ())) '(0 1 1 2 3 5 8 13 21 34 55))
              (test (interpret #(   define =0? dup 0 = end
                                     define gcd
                                     =0? if drop exit endif
                                     swap over mod
                                     gcd
                                     end
                                     90 99 gcd
                                     234 8100 gcd    ) '()) '(18 9))
              ))

(run-tests tests)
```

Тестирование

```
Welcome to DrRacket, version 8.10 [cs].
Language: R5RS; memory limit: 128 MB.
> (run-tests tests)
(interpret #(2 3 * 4 5 * +) '()) ok
(interpret #(define abs dup 0 < if neg exit endif end 9 abs -9 abs) '()) ok
(interpret #(neg) '(-9)) ok
(interpret #(dup 0) '(-9)) ok
(interpret #(1 not 0 not or) '()) ok
(interpret #(0 not 0 not and) '()) ok
(interpret #(define abs dup 0 < if neg endif end 9 abs -9 abs) '()) ok
(interpret #(define =0? dup 0 = end
             define <0? dup 0 < end
             define signum =0? if exit endif <0? if drop -1 exit endif drop 1 end
             0 signum -5 signum 10 signum) '()) ok
(interpret #(define -- 1 - end
             define =0? dup 0 = end
             define =1? dup 1 = end
             define factorial =0? if drop 1 exit endif =1?
                                  if drop 1 exit endif dup -- factorial * end
             0 factorial 1 factorial 2 factorial 3 factorial 4 factorial) '()) ok
(interpret #(define =0? dup 0 = end
             define =1? dup 1 = end
             define -- 1 - end define fib =0? if drop 0 exit endif =1?
                                            if drop 1 exit endif -- dup -- fib swap fib + end
             define make-fib dup 0 < if drop exit endif dup fib swap -- make-fib end
             10 make-fib) '()) ok
(interpret #(define =0? dup 0 = end
             define gcd =0? if drop exit endif swap over mod gcd end
             90 99 gcd 234 8100 gcd) '()) ok
#t
```

Вывод

Мы научились создавать собственный интерпретатор стекового языка программирования. Мы узнали, что состояние интерпретатора должно описываться вектором слов, счетчиком слов (индекс текущего слова), стеком данных, стеком возвратов и словарём (ассоциативный список). Несмотря на то что работа оказалась достаточно объёмной, мы считаем её несложной и интересной.