Лабораторная работа № 5. Интерпретатор стекового языка программирования

16 декабря 2023 г.

Бойко Роман, ИУ9-12Б

Цель работы

Реализуйте интерпретатор стекового языка программирования, описание которого представлено ниже. Интерпретатор должен вызываться как процедура `(interpret program stack)` которая принимает программу на исходном языке `program` и начальное состояние стека данных `stack` и возвращает его состояние после вычисления программы. Программа на исходном языке задана вектором литеральных констант, соответствующих словам исходного языка. Исходное и конечное состояния стека данных являются списком, голова которого соответствует вершине стека.

При реализации интерпретатора избегайте императивных конструкций, используйте модель вычислений без состояний. Для хранения программы и состояния интерпретатора **запрещается** использовать глобальные переменные. Перечисленные ниже встроенные слова обязательны для реализации и будут проверены сервером тестирования.

Реализация

```
(define (interpret program stack)
  (let ((env '())
        (oper_with_stack '(+ - * /
                           mod neg drop swap
                           dup over rot and
                           = > < not or depth))
        (end (vector-length program)))

    (let inner ((loc_stack stack)
                (ind 0)
                (func-ignore #f)
                (return '())
                (if-ignore #f))
```

```
(if (equal? ind end)
    loc_stack
    (let* ((current_symb (vector-ref program ind))
           (new_stack (if (and (equal? func-ignore #f) (equal? if-ignore #f))
                          (case current_symb
                            ('+ (+$ loc_stack))
                            ('- (-$ loc_stack))
                            ('* (*$ loc_stack))
                            ('/ (/$ loc_stack))
                            ('mod (mod$ loc_stack))
                            ('neg (neg$ loc_stack))
                            ('drop (drop$ loc_stack))
                            ('swap (swap$ loc_stack))
                            ('dup (dup$ loc_stack))
                            ('over (over$ loc_stack))
                            ('rot (rot$ loc_stack))
                            ('depth (depth$ loc_stack))
                            ('and (and$ loc_stack))
                            ('or (or$ loc_stack))
                            ('not (not$ loc_stack))
                            ('= (=$ loc_stack))
                            ('> (>$ loc_stack))
                            ('< (<$ loc_stack))
                            (else loc_stack)) loc_stack)))
      (cond
        ;; end of definition of the article
        ((and (null? return) (equal? current_symb 'end))
         (inner new_stack (+ ind 1) #f return if-ignore))

        ;; end of execution of the article and return up the call stack
        ((and (not (null? return)) (equal? current_symb 'end))
         (inner new_stack (car return) #f (cdr return) if-ignore))

        ;; definition of the article so ignoring the program
        ((equal? func-ignore #t)
         (inner new_stack (+ ind 1) func-ignore return if-ignore))

        ;; end of if-statement
        ((equal? current_symb 'endif)
         (inner new_stack (+ ind 1) func-ignore return #f))

        ;; if-statement is false so ignoring the program
        ((equal? if-ignore #t)
         (inner new_stack (+ ind 1) func-ignore return if-ignore))

        ;; exit is outside the articles so we are completing the program
```

```
                ((and (null? return) (equal? current_symb 'exit))
                 (inner new_stack end func-ignore return if-ignore))

                ;; exit is in the articles so we are moving up the call stack
                ((and (not (null? return)) (equal? current_symb 'exit))
                 (inner new_stack (car return) func-ignore (cdr return) if-ignore))

                ;; symbol is a number so it add to stack
                ((number? current_symb)
                 (inner (cons current_symb loc_stack) (+ ind 1) func-ignore return if-ignore))

                ;; check if term is an operation with stack
                ((my-element? current_symb oper_with_stack)
                 (inner new_stack (+ ind 1) func-ignore return if-ignore))

                ;; definition of the article
                ((equal? current_symb 'define)
                 (begin
                   (set! env
                         (append env
                                 (cons
                                  (cons (vector-ref program (+ ind 1))
                                        (cons (+ ind 2) '())) '()))))
                   (inner new_stack (+ ind 1) #t return if-ignore)))

                ;; executing previously defined  article
                ((assoc current_symb env)
                 (inner new_stack (cadr (assoc current_symb env))
                        func-ignore (cons (+ ind 1) return) if-ignore))

                ;; if-statement
                ((equal? current_symb 'if)
                 (if (not (equal? (car loc_stack) 0))
                     (inner (cdr new_stack) (+ ind 1) func-ignore return if-ignore)
                     (inner (cdr new_stack) (+ ind 1) func-ignore return #t)))))))))


;; Arithmetic operations

(define (+$ stack)
  (cons (+ (cadr stack) (car stack)) (cddr stack)))

(define (-$ stack)
  (cons (- (cadr stack) (car stack)) (cddr stack)))

(define (*$ stack)
```

```
  (cons (* (cadr stack) (car stack)) (cddr stack)))

(define (/$ stack)
  (if (< (cadr stack) (car stack))
      (cons 0 (cddr stack))
      (cons (/ (cadr stack) (car stack)) (cddr stack))))

(define (mod$ stack)
  (cons (remainder (cadr stack) (car stack)) (cddr stack)))

(define (neg$ stack)
  (cons (- (car stack)) (cdr stack)))


;; Comparison Operations

(define (=$ stack)
  (if (= (cadr stack) (car stack))
      (cons -1 (cddr stack))
      (cons 0 (cddr stack))))

(define (>$ stack)
  (if (> (cadr stack) (car stack))
      (cons -1 (cddr stack))
      (cons 0 (cddr stack))))

(define (<$ stack)
  (if (< (cadr stack) (car stack))
      (cons -1 (cddr stack))
      (cons 0 (cddr stack))))


;; Logical operations

(define (not$ stack)
  (if (= (car stack) 0)
      (cons -1 (cdr stack))
      (cons 0 (cdr stack))))

(define (and$ stack)
  (if (and (not (= (car stack) 0)) (not (= (cadr stack) 0)))
      (cons -1 (cddr stack))
      (cons 0 (cddr stack))))

(define (or$ stack)
  (if (or (not (= (car stack) 0)) (not (= (cadr stack) 0)))
```

```
      (cons -1 (cddr stack))
      (cons 0 (cddr stack))))


;; Stack Operations

(define (drop$ stack)
  (cdr stack))

(define (swap$ stack)
  (cons (cadr stack)
        (cons (car stack)
              (cddr stack))))

(define (dup$ stack)
  (cons (car stack) stack))

(define (over$ stack)
  (cons (cadr stack) stack))

(define (rot$ stack)
  (cons (caddr stack)
        (cons (cadr stack)
              (cons (car stack)
                    (cdddr stack)))))

(define (depth$ stack)
  (length stack))


(define (my-element? x xs)
  (cond
    ((null? xs) #f)
    ((equal? x (car xs)) #t)
    (else (my-element? x (cdr xs)))))
```

Тестирование

```
(define tests
  (list (test (interpret #(1) '()) '(1))
        (test (interpret #(1 2) '(10)) '(2 1 10))
        (test (interpret #(1 2 +) '(10)) '(3 10))
        (test (interpret #(1 2 -) '(10)) '(-1 10))
        (test (interpret #(1 2 *) '(10)) '(2 10))
        (test (interpret #(1 2 /) '(10)) '(0 10))
```

```
(test (interpret #(1 2 mod) '(10)) '(1 10))
(test (interpret #(1 2 neg) '(10)) '(-2 1 10))
(test (interpret #(2 3 * 4 5 * +) '()) '(26))
(test (interpret #(10 10 =) '()) '(-1))
(test (interpret #(10 0 >) '()) '(-1))
(test (interpret #(0 10 <) '()) '(-1))
(test (interpret #(10 5 =) '()) '(0))
(test (interpret #(0 10 >) '()) '(0))
(test (interpret #(10 0 <) '()) '(0))
(test (interpret #(0 0 and) '()) '(0))
(test (interpret #(1000 7 -) '()) '(993))
(test (interpret #(100 100 and) '()) '(-1))
(test (interpret #(100 0 or) '()) '(-1))
(test (interpret #(100 not) '()) '(0))
(test (interpret #(0 not) '()) '(-1))
(test (interpret #(define -- 1 - exit end) '()) '())
(test (interpret #(define -- 1 - end
                    5 -- --)
                  '())
      '(3))
(test (interpret #(10 15 +
                     define -- 1 - end
                     exit
                     5 -- --)
                  '())
      '(25))
(test (interpret #(10 15 +
                     define -- exit 1 - end
                     5 -- --)
                  '())
      '(5 25))
(test (interpret #(10 4 dup) '()) '(4 4 10))
(test (interpret #(define abs
                    dup 0 <
                    if neg endif
                    end
                    9 abs
                    -9 abs
                    10 abs
                    -10 abs)
                  '())
      '(10 10 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)
                  '())
      '(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)
                  '())
      '(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)
                  '())
      '(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))
(test (interpret #(define =0? dup 0 = end =0?) '(0)) '(-1 0))
 (test (interpret #(define =0? dup 0 = end
                    define kek 0 =0? end
                    kek)
                  '())
       '(-1 0))))

(run-tests tests)
```

Вывод

Реализовал интерпретатор стекового языка программирования. Понял, как работают стековые языки программирования. Разобрался как использовать стек данных, что такое стек вызовов и как с ним работать.