Лабораторная работа № 4. Метапрограммирование. Отложенные вычисления

28 октября 2024 г.

Александр Яннаев, ИУ9-11Б

Цель работы

На примере языка Scheme ознакомиться со средствами метапрограммирования («код как данные», макросы) и подходами к оптимизации вычислений (мемоизация результатов вычислений, отложенные вычисления). Также требуется разработать средство отладки программ с использованием утверждений (assert) и изучить программирование с продолжениями.

Индивидуальный вариант

- Процедура: `select-rec`
- Процедура: `text-stat`
- Реализация ленивого списка факториалов
- Макрос: `unless`
- Макрос: `while`
- Макрос: `cout << ... << endl`

Реализация

`iu9_lab4.scm`

```
(define (terminate) (display "#f"))
; enable-assertions
(define-syntax enable-assertions
  (syntax-rules ()
    ((enable-assertions)
     (call-with-current-continuation
      (lambda (cont)
        (set! terminate cont))))))
```

```
(define-syntax assert
  (syntax-rules ()
    ((assert condition)
     (if (not condition)
         (begin
           (display "fail ")
           (write 'condition)
           (newline)
           (terminate))))))

(enable-assertions)

(define (inverse x)
  (assert (not (zero? x)))
  (/ 1 x))

(display (map inverse '(1 2 3 4 5)))
(newline)

(display (map inverse '(-2 -1 0 1 2)))

; without memoization
(define (tribonacci n)
  (cond
    ((= n 0) 0)
    ((= n 1) 0)
    ((= n 2) 1)
    (else (+ (tribonacci (- n 1))
             (tribonacci (- n 2))
             (tribonacci (- n 3))))))

; with memoization
(define tribonacci-memo
  (let ((memo (make-vector 1000 #f)))
    (define (trib n)
      (if (not (vector-ref memo n))
          (begin
            (vector-set! memo n
                         (cond
                           ((= n 0) 0)
                           ((= n 1) 0)
                           ((= n 2) 1)
                           (else (+ (trib (- n 1))
                                    (trib (- n 2))
                                    (trib (- n 3))))))
```

```
            (vector-ref memo n)
            )
          (vector-ref memo n)))
    trib))

; lazy-funcs
(define-syntax lazy-cons
  (syntax-rules ()
    ((_ a b)
     (cons a (delay b)))))


(define (lazy-car p)
  (car p))


(define (lazy-cdr p)
  (force (cdr p)))

; lazy-head
(define (lazy-head xs k)
  (if (zero? k)
      '()
      (cons (lazy-car xs) (lazy-head (lazy-cdr xs) (- k 1)))))

; lazy-ref
(define (lazy-ref xs k)
  (if (zero? k)
      (lazy-car xs)
      (lazy-ref (lazy-cdr xs) (- k 1))))

; lazy-map
(define (lazy-map proc xs)
  (if (null? xs)
      '()
      (lazy-cons (proc (lazy-car xs)) (lazy-map proc (lazy-cdr xs)))))

; lazy-zip
(define (lazy-zip xs ys)
  (if (or (null? xs) (null? ys))
      '()
      (lazy-cons (list (lazy-car xs) (lazy-car ys))
                 (lazy-zip (lazy-cdr xs) (lazy-cdr ys)))))

(define ones (lazy-cons 1 ones))
```

```
(display (lazy-head ones 5))
(newline)

; natural numbers
(define naturals
  (lambda (start)
    (lazy-cons start (naturals (+ start 1)))))


(define factorials
  (lazy-cons 1
             (lazy-map (lambda (n-f)
                         (let ((n (car n-f))
                               (f (cadr n-f)))
                           (* n f)))
                       (lazy-zip (naturals 1) factorials))))


(display (lazy-head factorials 12))
(newline)

; unless
(define-syntax unless
  (syntax-rules ()
    ((unless condition? expr1 expr2 ...)
     (if (not condition?)
         (begin expr1 expr2 ...)))))

; while
(define-syntax while
  (syntax-rules ()
    ((while condition? expr1 expr2 ...)
     (letrec ((loop (lambda ()
                      (if condition?
                          (begin
                            expr1
                            expr2 ...
                            (loop))))))
       (loop)))))
; C++
(define << " ")
(define endl 'endl)

(define (cout . args)
  (for-each print-helper args))
```

```
(define (print-helper elem)
  (if (eq? elem endl)
      (newline)
      (display elem)))

;text-stat, select-rec
(define (read-file filename)
  (let ((input-port (open-input-file filename)))
    (let loop ((character (read-char input-port)))
      (if (eof-object? character)
          (begin (close-input-port input-port) '())
          (cons character (loop (read-char input-port)))))))

(define (chars->words char-list)
  (define (helper chars current-word result)
    (cond
      ((null? chars)
       (if (null? current-word)
           (reverse result)
           (reverse (cons (list->string (reverse current-word)) result))))
      ((or (char=? (car chars) #\space)
           (char=? (car chars) #\newline))
       (if (null? current-word)
           (helper (cdr chars) current-word result)
           (helper (cdr chars) '() (cons (list->string (reverse current-word)) result))))
      (else
       (helper (cdr chars) (cons (car chars) current-word) result))))

  (helper char-list '() '()))

(define (is-number? ch)
  (and (char? ch) (char<=? #\0 ch #\9)))

(define (is-letter? ch)
  (and (char? ch) (char<=? #\A ch #\z)))

(define (check-type str)
  (define (loop chars result)
    (cond ((null? chars) result)
          ((equal? (car chars) #\') 1)
          ((or (equal? (car chars) #\=)
               (equal? (car chars) #\+)
               (equal? (car chars) #\*)
               (equal? (car chars) #\\)
               (equal? (car chars) #\%)
```

```
                (equal? (car chars) #\>)
                (equal? (car chars) #\<)) 3)
            ((and (or (= result 0) (= result 2)) (is-number? (car chars))) (loop (cdr chars) 2
            ((and (= result 1) (is-number? (car chars))) 3)
            ((is-letter? (car chars)) (loop (cdr chars) 1))))
  (loop (string->list str) 0))

(define (text-stat source)
  (define (count-types source)
    (define (helper words counts)
      (if (null? words)
          counts
          (let ((word (car words)))
            (cond ((= (check-type word) 2)
                   (begin (vector-set! counts 1 (+ 1 (vector-ref counts 1)))
                          (helper (cdr words) counts)))
                  ((= (check-type word) 3)
                   (begin (vector-set! counts 2 (+ 1 (vector-ref counts 2)))
                          (helper (cdr words) counts)))
                  (else
                   (begin (vector-set! counts 0 (+ 1 (vector-ref counts 0)))
                          (helper (cdr words) counts)))))))
    (helper source (vector 0 0 0)))
  (count-types (chars->words (read-file source))))

(text-stat "1.txt")

(define (read-functions filename)
  (define (loop port acc)
    (let ((text (read port)))
      (if (eof-object? text)
          acc
          (loop port (append acc (list text))))))
  (let ((port (open-input-file filename))) (loop port '())))

(define (my-flatten xs)
  (define (loop xs acc)
    (if (null? xs)
        acc
        (if (null? (car xs))
            (loop (cdr xs) acc)
            (if (list? (car xs))
                (loop (cons (caar xs) (cons (cdar xs) (cdr xs))) acc)
                (loop (cdr xs) (append acc (list (car xs))))))))
  (loop xs '()))
```

```
(define (my-element? x xs)
  (and (not (null? xs))
       (or (equal? x (car xs))
           (my-element? x (cdr xs)))))

(define (write-rec file val)
  (with-output-to-file file
    (lambda ()
      (for-each (lambda (x) (write x) (newline)) val))))

(define (is-recursive? fn)
  (and (list? (cadr fn)) (my-element? (caadr fn) (my-flatten (cddr fn)))))

(define (select-rec source dest)
  (define (find-rec expr)
    (cond
      ((null? expr) '())
      ((not (list? expr)) '())
      ((equal? 'define (caar expr))
       (if (is-recursive? (car expr))
           (append (list (car expr)) (find-rec (cdr expr)))
           (find-rec (cdr expr))))
      (else (find-rec (cdr expr)))))

  (let ((expr (read-functions source)))
    (let ((rec-procs (find-rec expr)))
      (write-rec dest rec-procs))))

;(select-rec "source.scm" "output.scm")

; testing...
(cout << "a =" << 1 << endl << "b =" << 2 << endl)

(display (lazy-head factorials 10))

(newline)
(tribonacci 30)
(tribonacci-memo 30)
```

source.scm

```
(define (factorial n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
(define (fibonacci n)
  (if (<= n 1)
      n
      (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))


(define (add a b)
  (+ a b))


(define (square x)
  (* x x))


(define (count-items lst)
  (if (null? lst)
      0
      (+ 1 (count-items (cdr lst)))))


(define (sum3 a b c)
  (+ a b c))
```

1.txt

```
E=mc^2 hello 123
```

Тестирование

iu9_lab4.scm

```
Welcome to DrRacket, version 8.7 [3m].
Language: R5RS; memory limit: 256 MB.
(1 1/2 1/3 1/4 1/5)
fail (not (zero? x))
(1 1 1 1 1)
(1 1 2 6 24 120 720 5040 40320 362880 3628800 39916800)
#(1 1 1)
 a = 1
 b = 2
(1 1 2 6 24 120 720 5040 40320 362880)
15902591
15902591
```

output.scm

```
(define (factorial n) (if (<= n 1) 1 (* n (factorial (- n 1)))))
(define (fibonacci n) (if (<= n 1) n (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))
(define (count-items lst) (if (null? lst) 0 (+ 1 (count-items (cdr lst)))))
```

Вывод

В ходе работы я углубился в изучение метапрограммирования, освоил подходы к написанию кода, который может генерировать или изменять другие программы. Также я научился оптимизировать рекурсивные процедуры, что позволило мне лучше понять, как эффективно управлять памятью и вычислительными ресурсами, особенно в контексте функционального программирования на языке Scheme. Эти знания помогут мне создавать более гибкие и производительные программы в будущем.