Лабораторная работа № 4. Метапрограммирование. Отложенные вычисления

30 декабря 2023 г.

Семён Чайкин, 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).

Задание 1

Утверждение (assertion) — проверка на истинность некоторого условия, заданного программистом. По традиции осуществляется процедурой (функцией) с именем `assert`. Включается в код во время написания кода и отладки с целью установки ограничений на значения и выявления недопустимых значений. Если в процессе выполнения программы указанное условие нарушается, то программа завершается с выводом диагностического сообщения о том, какое условие было нарушено. Если условие соблюдено, то выполнение программы продолжается, никаких сообщений не выводится.

Реализуйте каркас (фреймворк) для отладки с помощью утверждений. Пусть Ваш каркас перед использованием инициализируется вызовом (`use-assertions`), а сами утверждения записываются в коде ниже в виде (`assert` условие). Если условие не выполнено, происходит завершение работы программы без возникновения ошибки выполнения и вывод в консоль диагностического сообщения вида `FAILED:` условие.

Реализация

```
(define *assertion* #f)

(define call/cc call-with-current-continuation)

(define (use-assertions)
  (call/cc (lambda (cc) (set! *assertion* cc))))

(define-syntax assert
  (syntax-rules ()
    ((assert cond?)
     (and *assertion*
          (or cond?
              ((begin
                 (display "FAILED: ")
                 (write 'cond?)
                 (newline)
                 (*assertion*))))))))
```

Тестирование

Пример тестирования:

```
(use-assertions)

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

```
(map 1/x '(1 2 3 4 5))
(map 1/x '(-2 -1 0 1 2))
```

Результат в консоли:

```
(1 1/2 1/3 1/4 1/5)
FAILED: (not (zero? x))
```

Задание 2

- *Сериализация данных*. Реализуйте процедуры для записи данных из переменной в файл по заданному пути (т.е. для сериализации) и последующего чтения данных (десериализации) из такого файла:

  ```
  (save-data данные путь-к-файлу)
  (load-data путь-к-файлу) ;; данные
  ```

- *Подсчет строк в текстовом файле*. Реализуйте процедуру, принимающую в качестве аргумента путь к текстовому файлу и возвращающую число непустых строк в этом файле. Используйте процедуры, разработанные вами ранее в рамках выполнения домашних заданий.

Реализация

```
(define (load-data filename)
  (call-with-input-file filename
    (lambda (p)
      (let loop
        ((line (read p)) (result '()))
        (if (eof-object? line)
            (reverse result)
            (loop (read p) (cons line result)))))))

(define (save-data data filename)
  (call-with-output-file filename
    (lambda (p)
      (write data p))))

(define (count-lines filename)
  (with-input-from-file filename
    (lambda ()
      (begin
        (let loop ((prev #f) (count 0))
          (let ((ch (read-char)))
            (cond
              ((eof-object? ch) count)
              (else
               (loop (equal? ch #\newline)
                     (+ count (if (and (equal? ch #\newline) (not prev)) 1 0)))))))))))
```

Тестирование

```
> (save-data '(1 2 3 4 5) "test.txt") ;; создался файл "test.txt"
> (load-data "test.txt")
((1 2 3 4 5))
> (count-lines "test.txt")
1
```

Задание 3

Реализуйте функцию вычисления n-го “числа трибоначчи” (последовательности чисел, которой первые три числа равны соответственно 0, 0 и 1, а каждое последующее число — сумме предыдущих трех чисел):

Функция:

$$t(n) = \begin{cases} 0, & n \leqslant 1; \\ 1, & n = 2; \\ t(n-1) + t(n-2) + t(n-3), & n > 2; \end{cases}$$

Область определения: $D_t = \{n : n \in \mathbb{Z}, n \geqslant 0\}$.

Реализация

```
;; Обычная версия функции
(define (t n)
  (cond
    ((<= n 1) 0)
    ((= n 2) 1)
    (else (+ (t (- n 1)) (t (- n 2)) (t (- n 3))))))

;; Мемоизированная версия
(define t-memo
  (let ((table '()))
    (lambda (n)
      (if (assoc n table)
          (cadr (assoc n table))
          (begin

            (cond
              ((<= n 1) (set! table (cons `(,n 0) table)))
              ((= n 2) (set! table (cons `(,n 1) table)))
              (else
               (set! table (cons `(,n ,(+ (t-memo (- n 1))
                                          (t-memo (- n 2))
                                          (t-memo (- n 3)))) table))))
            (cadr (assoc n table)))))))
```

Тестирование

```
> (t 35)
334745777
;; real 4.207s
;; user 4.088s

> (t-memo 50)
334745777
;; real 0.374s
;; user 0.292s
```

Задание 4

Используя примитивы для отложенных вычислений `delay` и `force`, реализуйте макрос `my-if`, который полностью воспроизводит поведение встроенной условной конструкции (специальной формы) `if` для выражений, возвращающих значения. Например, такие примеры должны вычисляться корректно:

```
(my-if #t 1 (/ 1 0)) ;; 1
(my-if #f (/ 1 0) 1) ;; 1
```

Запрещается использовать встроенные условные конструкции `if`, `cond`, `case` и перехват исключений.

Реализация

```
(define-syntax my-if
  (syntax-rules ()
```

```
    ((_ true? opt1 opt2)
     (let ((f1 (delay opt1))
           (f2 (delay opt2)))
       (force (or (and true? f1) f2))))))
```

Тестирование

```
> (my-if #t #f (/ 1 0))
#f
> (my-if #f (/ 1 0) 1)
1
```

Задание 5

Реализуйте макросы `my-let` и `my-let*`, полностью воспроизводящие поведение встроенных макросов `let` и `let*`.

Реализация

```
(define-syntax my-let
  (syntax-rules ()
    ((my-let ((var val) ...) body)
     ((lambda (var ...) body) val ...))))

(define-syntax my-let*
  (syntax-rules ()
    ((my-let* () body) body)
    ((my-let* ((var val)) body)
     (my-let ((var val)) body))
    ((my-let* ((var val) . xs) body)
     (my-let ((var val)) (my-let* xs body)))))
```

Тестирование

Пример тестирования:

```
(define (f1 x y)
  (my-let ((x (+ y 12))
           (y (* x 3)))
          (+ x y)))

(define (f2 x y)
  (let ((x (+ y 12))
        (y (* x 3)))
    (+ x y)))
```

Результат в консоли:

```
> (f1 12 2)
50
> (f2 12 2)
50
```

Задание 6

Используя *гигиенические* макросы языка Scheme, реализуйте управляющие конструкции, свойственные императивным языкам программирования.

Реализация

```
;; A
(define-syntax when
```

```
  (syntax-rules ()
    ((_ cond? . actions) (if cond? (begin . actions)))))

(define-syntax unless
  (syntax-rules ()
    ((_ cond? . actions) (if (not cond?) (begin . actions)))))

;; Б
(define-syntax for
  (syntax-rules (as in)
    ((for x in xs . actions)
     (let loop ((xs-copy xs))
       (if (not (null? xs-copy))
           (let ((x (car xs-copy)))
             (begin . actions)
             (loop (cdr xs-copy))))))
    ((for xs as x . actions)
     (for x in xs . actions))))

;; В
(define-syntax while
  (syntax-rules ()
    ((_ cond? . actions)
     (let loop ()
       (if cond?
           (begin
             (begin . actions)
             (loop)))))))

;; Г
(define-syntax repeat
  (syntax-rules (until)
    ((repeat actions until cond?)
     (let loop ()
       (begin
         (begin . actions)
         (if (not cond?)
             (loop)))))))

;; Д
(define-syntax cout
  (syntax-rules (<< endl)
    ((cout << endl)
     (newline))
    ((cout << endl . exprs)
     (begin
       (newline)
       (cout . exprs)))
    ((cout << expr)
     (display expr))
    ((cout << expr . exprs)
     (begin
       (display expr)
       (cout . exprs)))))
```

Тестирование

Пример тестирования:

```
;; A
(let ((x 1))
```

```
  (when (> x 0) (display "x > 0") (newline))
  (unless (= x 0) (display "x != 0") (newline)))

;; Б
(for i in '(1 2 3)
  (for '(4 5 6) as j
    (display (list i j))
    (newline)))

;; В
(let ((p 0)
      (q 0))
  (while (< p 3)
         (set! q 0)
         (while (< q 3)
                (display (list p q))
                (newline)
                (set! q (+ q 1)))
         (set! p (+ p 1))))

;; Г
(let ((i 0)
      (j 0))
  (repeat ((set! j 0)
           (repeat ((display (list i j))
                    (set! j (+ j 1)))
                   until (= j 3))
           (set! i (+ i 1))
           (newline))
          until (= i 3)))

;; Д
(cout << "a = " << 1 << endl << "b = " << 2 << endl)
```

Результат в консоли:

```
;; А
x > 0
x != 0

;; Б
(1 4)... (1 5)... (1 6)...
(2 4)... (2 5)... (2 6)...
(3 4)... (3 5)... (3 6)...

;; В
(0 0)... (0 1)... (0 2)...
(1 0)... (1 1)... (1 2)...
(2 0)... (2 1)... (2 2)...

;; Г
(0 0)(0 1)(0 2)
(1 0)(1 1)(1 2)
(2 0)(2 1)(2 2)

;; Д
a = 1
b = 2
```

Вывод

В ходе выполнения этой лабораторной работы я ознакомился со средствами метапрограммирования языка `Scheme`, а именно с принципом “код как данные”. Также очень интересным оказалось создание собственных макросов, потому что позволило посмотреть на программирование с новой стороны, где код не только производит вычисления, но и сам становится объектом обработки.

Также интересным оказалось изучение методов оптимизации вычислений. До этой лабораторной я знал только про мемоизацию (и принципы её работы в языке `Python`), но задачи позволили мне посмотреть на неё с новой стороны, а также познакомили с новым для меня концептом “ленивых вычислений”.

Я думаю, что знания, приобретённые во время выполнения этой лабораторой работы, будут полезными для дальнейшего развития в программировании.