

Домашнее задание №4

Table of Contents

- [1. 1. Мемоизация](#)
 - [1.1. Решение](#)
 - [1.2. Тесты](#)
- [2. 2. Отложенные вычисления](#)
- [3. 3. Чтение из потока](#)
 - [3.1. Решение](#)
 - [3.2. Тесты](#)
- [4. 4. Структуры \(записи\)](#)
 - [4.1. Решение](#)
 - [4.2. Тесты](#)
- [5. 5. Алгебраические типы данных](#)
 - [5.1. Решение](#)
 - [5.2. Тесты](#)
- [6. «Ачивки»](#)

1. 1. Мемоизация

Реализуйте рекурсивные вычисления с использованием оптимизационных техник — мемоизации результатов вычислений и отложенных вычислений.

Важно! Если в программе используются гигиенические макросы и эта программа будет выполнена в среде guile 1.8.x (в том числе на сервере тестирования), то следует подключить модуль поддержки таких макросов, написав в начале программы следующую строку:

```
(use-syntax (ice-9 syncase))
```

Реализуйте процедуру *memoized-factorial* для вычисления факториала по рекурсивной формуле с мемоизацией результатов вычислений. Для мемоизации используйте ассоциативный список (словарь), который храните в статической переменной. Использовать для этой цели глобальную переменную *запрещается*.

Примеры вызова процедур сервером тестирования:

```
(begin
  (display (memoized-factorial 10)) (newline)
  (display (memoized-factorial 50)) (newline))

3628800
3041409320171337804361260816606476884437764156896051200000000000
```

1.1. Решение

```
(define memoized-factorial
  (let ((known-results '()))
    (lambda (n)
      (cond ((= n 0) 1)
            ((assoc n known-results) -> (lambda (res) res))
            (else (let ((res (* n (memoized-factorial (- n 1)))))
                     (begin (cons '(n res) known-results)
                             res)))))))
```

1.2. Тесты

```
(begin
  (display (memoized-factorial 10)) (newline)
  (display (memoized-factorial 20)) (newline)
  (display (memoized-factorial 50)) (newline))
```

```
3628800
2432902008176640000
3041409320171337804361260816606476884437764156896051200000000000
```

2. 2. Отложенные вычисления

Используя средства языка Scheme для отложенных вычислений, реализуйте средства для работы с бесконечными «ленивыми» точечными парами и списками:

- Гигиенический макрос (*lazy-cons a b*), конструирующий ленивую точечную пару вида (*значение-a . обещание-вычислить-значение-b*). Почему макрос в данном случае предпочтительнее процедуры?
 - Из-за порядка вычислений

```
(define-syntax lazy-cons
  (syntax-rules ()
    ((lazy-cons a b)
     (delay (cons a b))))))
```

- Процедуру (*lazy-car p*), возвращающую значение 1-го элемента «ленивой» точечной пары *p*.

```
(define (lazy-car p)
  (car (force p)))
```

- Процедуру (*lazy-cdr p*), возвращающую значение 2-го элемента «ленивой» точечной пары *p*.

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

На их основе определите:

- Процедуру (*lazy-head xs k*), возвращающую значение *k* первых элементов «ленивого» списка *xs* в виде списка.

```
(define (lazy-head xs k)
  (let loop ((xs xs) (k k) (res '()))
    (if (= k 0)
        (reverse res)
        (loop (lazy-cdr xs) (- k 1) (cons (lazy-car xs) res)))))
```

- Процедуру (*lazy-ref xs k*), возвращающую значение *k*-го элемента «ленивого» списка *xs*.

```
(define (lazy-ref xs k)
  (let loop ((xs xs) (k k))
    (if (= k 0)
        (lazy-car xs)
        (loop (lazy-cdr xs) (- k 1)))))
```

Продемонстрируйте работу процедур на примере бесконечного списка натуральных чисел. Для этого определите процедуру-генератор (*naturals start*), возвращающую бесконечный «ленивый» список натуральных чисел, начиная с числа *start*.

Примеры вызова процедур сервером тестирования:

```
(display (lazy-head (naturals 10) 12))  
(10 11 12 13 14 15 16 17 18 19 20 21)
```

```
(define (naturals start)  
  (lazy-cons start (naturals (+ start 1))))  
  
(display (lazy-head (naturals 10) 12))
```

```
(10 11 12 13 14 15 16 17 18 19 20 21)
```

Реализуйте процедуру (lazy-factorial n), которая возвращает значение $n!$, получая его из n -го элемента бесконечного списка факториалов.

Примеры вызова процедур сервером тестирования:

```
(begin  
  (display (lazy-factorial 10)) (newline)  
  (display (lazy-factorial 50)) (newline))  
  
3628800  
30414093201713378043612608166064768844377641568960512000000000000
```

```
(define (factorials start val)  
  (lazy-cons val (factorials (+ start 1) (* val (+ start 1)))))  
  
(define (lazy-factorial n)  
  (lazy-ref (factorials 0 1) n))  
  
(begin  
  (display (lazy-factorial 10)) (newline)  
  (display (lazy-factorial 50)) (newline)  
  (display (lazy-factorial 1)) (newline)  
  (display (lazy-factorial 2)) (newline)  
  (display (lazy-factorial 0)) (newline))
```

```
3628800  
30414093201713378043612608166064768844377641568960512000000000000  
1  
2  
1
```

3. 3. Чтение из потока

Напишите процедуру (read-words), осуществляющую чтение слов, разделенных пробельными символами, из стандартного потока (порта) ввода (сервер тестирования направляет в этот поток текстовый файл с примером). Слова в потоке разделены одним или более пробельными символами. Также пробельные символы могут присутствовать перед первым словом и после последнего слова, такие пробельные символы должны игнорироваться. Признак конца файла означает окончание ввода. Процедура должна возвращать список строк (один элемент списка — одно слово в строке). Для пустого файла или файла, содержащего только пробелы, процедура должна возвращать пустой список.

Анализ потока символов осуществляйте непосредственно при чтении файла, для чего используйте встроенные процедуры read-char, peek-char, eof-object? и встроенные предикаты классификации символов.

Пример входных данных (· — пробел, ¶ — конец строки):

```
·one·two·three·¶ four·five·six·¶
```

Результат вызова процедуры для этих входных данных:

```
(read-words) ⇒ ("one" "two" "three" "four" "five" "six")
```

3.1. Решение

```
(define (read-words)
  (let loop ((res '())
            (word '()))
    (let ((cur (read-char)))
      (cond ((eof-object? cur)
             (reverse res)
             (if (null? word)
                 (loop res word)
                 (loop (cons (list->string (reverse word)) res)
                       '())))
            (else (loop res (cons cur word)))))))
```

3.2. Тесты

```
(with-output-to-file "home4.test" (lambda ()
                                     (display " one two three\n four five six \n")))
```

```
(with-input-from-file "home4.test" (lambda ()  
                                     (write (read-words))))
```

```
("one" "two" "three" "four" "five" "six")
```

4. 4. Структуры (записи)

Используя средства языка Scheme для метапрограммирования, реализуйте каркас поддержки типа данных «структура» («запись»). Пусть объявление нового типа «структура» осуществляется с помощью вызова (define-struct тип-структуры (имя-поля-1 имя-поля-2 ... имя-поля-n)). Тогда после объявления структуры программисту становятся доступны:

- Процедура — конструктор структуры вида (make-тип-структуры значение-поля-1 значение-поля-2 ... значение-поля-n), возвращающий структуру, поля которой инициализированы перечисленными значениями.
- Предикат типа вида (тип-структуры? объект), возвращающая #t если объект является структурой типа тип-структуры и #f в противном случае.
- Процедуры для получения значения каждого из полей структуры вида (тип-структуры-имя-поля объект).
- Процедуры модификации каждого из полей структуры вида (set-тип-структуры-имя-поля! объект новое-значение).

Пример использования каркаса:

```
(define-struct pos (row col)) ; Объявление типа pos  
(define p (make-pos 1 2))    ; Создание значения типа pos  
  
(pos? p)    ⇒ #t  
  
(pos-row p) ⇒ 1  
(pos-col p) ⇒ 2  
  
(set-pos-row! p 3) ; Изменение значения в поле row  
(set-pos-col! p 4) ; Изменение значения в поле col  
  
(pos-row p) ⇒ 3  
(pos-col p) ⇒ 4
```

Рекомендация. Для более короткой записи решения можно (но не обязательно) использовать квазицитирование (quasiquote).

Важно! Если в программе используются гигиенические макросы и эта программа будет выполнена в среде guile 1.8.x (в том числе на сервере тестирования), то следует подключить модуль поддержки таких макросов, написав в начале программы следующую строку:

```
(use-syntax (ice-9 syncase))
```

4.1. Решение

```
(define-syntax define-struct
  (syntax-rules ()
    ((_ sym-name sym-fields)
      (let loop ((name (symbol->string 'sym-name))
                 (fields (map symbol->string 'sym-fields))
                 (i 2))
        (if (null? fields)
            (eval `(begin (define (,(string->symbol (string-append "make-"
                                                                    name))
                                                                    . vals)
                              (list->vector (cons '_struct (cons 'sym-name vals))))
              (define (,(string->symbol (string-append name
                                                         "?"))
                      obj)
                (and (vector? obj)
                     (eqv? '_struct (vector-ref obj 0))
                     (eqv? 'sym-name (vector-ref obj 1)))))
            (interaction-environment))
          (begin (eval `(begin (define (,(string->symbol (string-append name
                                                                    "_")
                                                                    (car fields)))
                                                                    obj)
                              (vector-ref obj ,i))
                    (define (,(string->symbol (string-append "set-"
                                                                name
                                                                "_")
                                                                (car fields)
                                                                "!"))
                              obj
                              val)
                    (vector-set! obj ,i val)))
                (interaction-environment))
            (loop name (cdr fields) (+ i 1))))))
```

4.2. Тесты

```
(load "./unit-test.scm")

(define-struct pos (row col)) ; Объявление типа pos
(define p (make-pos 1 2))    ; Создание значения типа pos
```

```

(define struct-tests
  (list (test (pos? p) #t)
        (test (pos-row p) 1)
        (test (pos-col p) 2)
        (test (begin
                (set-pos-row! p 3)
                (set-pos-col! p 4)
                (pos-row p))
              3)
        (test (pos-col p) 4)))

(run-tests struct-tests)

```

```

Test 1: (pos? p) ok
Test 2: (pos-row p) ok
Test 3: (pos-col p) ok
Test 4: (begin (set-pos-row! p 3) (set-pos-col! p 4) (pos-row p)) ok
Test 5: (pos-col p) ok

```

5. 5. Алгебраические типы данных

Используя средства языка Scheme для метапрограммирования, реализуйте каркас поддержки алгебраических типов данных.

Алгебраический тип данных — составной тип, получаемый путем комбинации значений других типов (полей) с помощью функций-конструкторов. Такой тип допускает различные комбинации полей — варианты. Для каждого из вариантов предусматривается свой конструктор. Все варианты типа рассматриваются как один полиморфный тип. Функции (процедуры), работающие с алгебраическим типом, предусматривают отдельные ветви вычислений для каждого из вариантов.

Пример. Необходимо вычислять периметры геометрических фигур (квадратов, прямоугольников, треугольников) и длины окружностей. Для этого в программе определен тип фигура, который может принимать значения квадрат, прямоугольник, треугольник, окружность. Значения создаются с помощью конструкторов (для каждой фигуры — свой конструктор) — процедур, принимающих в качестве аргументов длины сторон (1, 2 или 3 аргумента соответственно) или радиус (для окружности) и возвращающих значение типа фигура:

```

; Определяем тип
;
(define-data figure ((square a)
                    (rectangle a b)
                    (triangle a b c)
                    (circle r)))

; Определяем значения типа
;

```



```
(define s (square 10))
(define r (rectangle 10 20))
(define t (triangle 10 20 30))
(define c (circle 10))

; Пусть определение алгебраического типа вводит
; не только конструкторы, но и предикат этого типа:
;
(and (figure? s)
     (figure? r)
     (figure? t)
     (figure? c)) ⇒ #t
```

Функция расчета периметра или длины окружности — единая для всех фигур, принимает значение типа фигура и возвращает значение, вычисленное по формуле, выбранной в соответствии с вариантом фигуры:

```
(define pi (acos -1)) ; Для окружности

(define (perim f)
  (match f
    ((square a)      (* 4 a))
    ((rectangle a b) (* 2 (+ a b)))
    ((triangle a b c) (+ a b c))
    ((circle r)      (* 2 pi r))))

(perim s) ⇒ 40
(perim r) ⇒ 60
(perim t) ⇒ 60
```

Здесь `match` — сопоставление с образцом. В данном примере при вычислении `(perim s)` значение `s` будет сопоставлено с образцом `(square a)`. При этом будет осуществлена подстановка фактического значения `a`, содержащегося в `s`, на место `a` в выражении `(* 4 a)` справа от образца. Вычисленное значение будет возвращено из конструкции `match`.

Рекомендации. Для более короткой записи решения можно (но не обязательно) использовать квазицитирование (quasiquotation). По литературе и ресурсам Интернет ознакомьтесь с тем, как работает сопоставление с образцом в других языках программирования.

5.1. Решение

```
(define-syntax define-data
  (syntax-rules ()
    ((_ name _constructors)
     (let loop ((constructors '_constructors))
       (if (null? constructors)
```

```

;; определим предикат
(eval `(define (,(string->symbol (string-append (symbol->string 'name)
                                                    "?"))
              obj)
      (and (list? obj)
            (eqv? '_data (car obj))
            (eqv? 'name (cadr obj))))
      (interaction-environment))
;; определим конструктор
(begin (eval `(define (,(caar constructors)
                        . params)
          (append (list '_data
                        'name
                        ',(caar constructors))
                  params))
        (interaction-environment))
      (loop (cdr constructors))))))

(define-syntax match
  (syntax-rules ()
    ((_ val ((name params ...) expr)
      (apply (lambda (params ...)
                expr)
              (cdddr val))))
    ((_ val ((name params ...) expr) patterns ...)
      (if (eqv? 'name (caddr val))
          (apply (lambda (params ...)
                    expr)
                  (cdddr val))
          (match val patterns ...))))))

```

5.2. Тесты

```

;; Определяем тип
(define-data figure ((square a)
                    (rectangle a b)
                    (triangle a b c)
                    (circle r)))

;; Определяем значения типа
(define s (square 10))
(define r (rectangle 10 20))
(define t (triangle 10 20 30))
(define c (circle 10))

(display (and (figure? s)

```

```

      (figure? r)
      (figure? t)
      (figure? c)))
(newline)

(define pi (acos -1)) ;; Для окружности

(define (perim f)
  (match f
    ((square a)      (* 4 a))
    ((rectangle a b) (* 2 (+ a b)))
    ((triangle a b c) (+ a b c))
    ((circle r)      (* 2 pi r))))

(display (perim s))
(newline)
(display (perim r))
(newline)
(display (perim t))
(newline)
(display (perim c))
(newline)

```

```

#t
40
60
60
62.83185307179586

```

6. «Ачивки»

- Объяснить, как и почему работает следующий фрагмент кода:

```

((call-with-current-continuation
  (lambda (c) c))
 (lambda (x) x))
'hello)

```

```

(<продолжение> (lambda (x) x))
'hello)

```

```
((lambda (x) x) (lambda (x) x))
'hello)
```

```
((lambda (x) x) 'hello)
```

```
'hello
```

Можно догадаться, что этот код печатает hello, но нужно объяснить почему. **+1 балл.**

- Написать макросы `my-let` и `=my-let*=` без использования эллипсисов (многоточий, ...) (и, конечно, встроенных `let`, `let*`, `letrec`) **+1 балл** за оба.

```
(define-syntax my-let
  (syntax-rules ()
    ((_ ((var val)) expr)
      ((lambda (var)
         expr) val))
    ((_ ((var val) . others) expr)
      ((lambda (var)
         (my-let others
                  expr)) val))))

(define-syntax my-let*
  (syntax-rules ()
    ((_ ((var val)) expr)
      ((lambda (var)
         expr) val))
    ((_ ((var val) . others) expr)
      ((lambda (var)
         (my-let* others
                  expr)) val))))

(load "./unit-test.scm")

(define tests
  (list
    (test (my-let ((x 5) (y 7))
                (+ x 1 y))
          13)
    (test (my-let ((=> #f))
              (cond (#t => 'ok)))
          ok)))
```

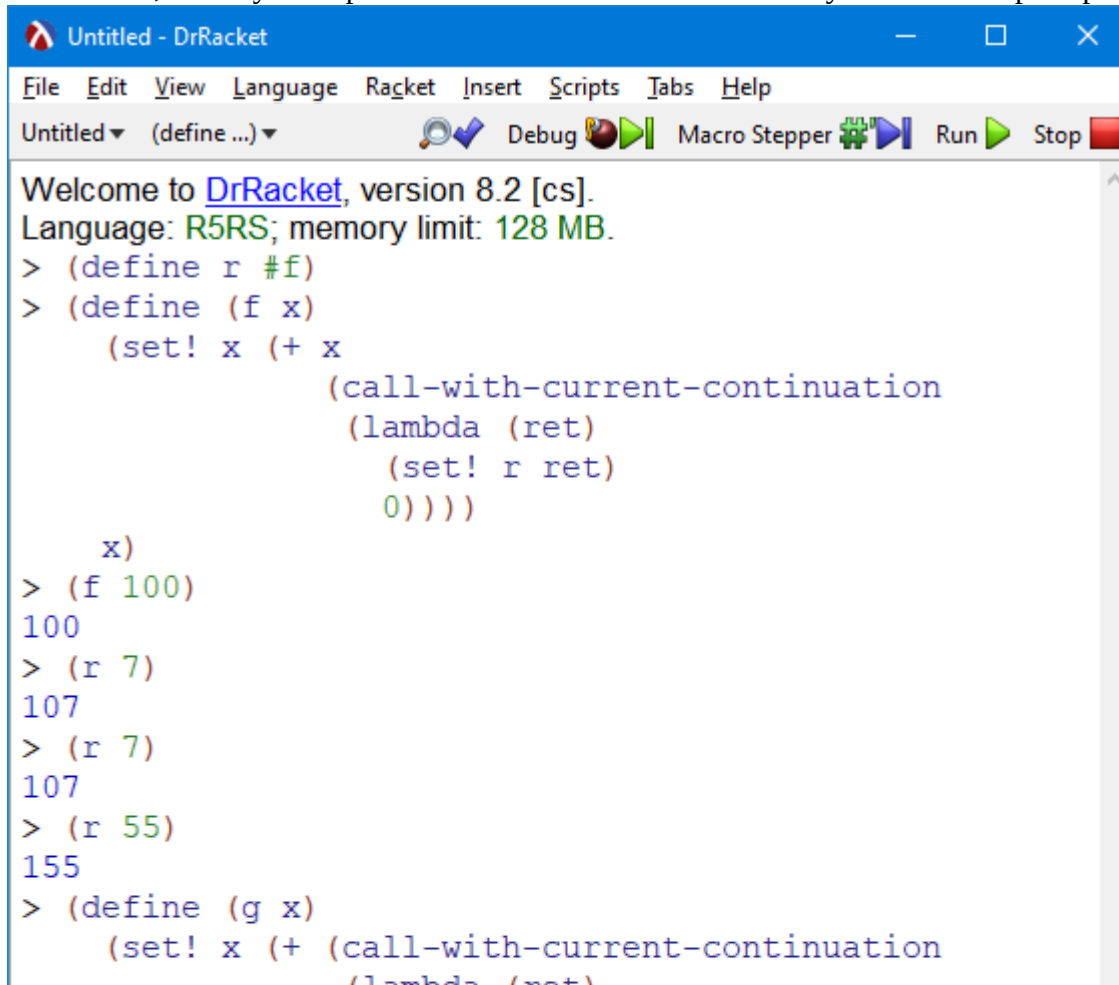
```
(run-tests tests)

(define tests
  (list
    (test (my-let* ((x 5) (y (+ x 7)))
            y) 12)))

(run-tests tests)
```

```
Test 1: (my-let ((x 5) (y 7)) (+ x 1 y)) ok
Test 2: (my-let ((=> #f)) (cond (#t => (quote ok)))) ok
Test 1: (my-let* ((x 5) (y (+ x 7))) y) ok
```

- Объяснить, почему от перемены мест слагаемых меняется сумма в этом примере (1 балл):




The screenshot shows the DrRacket IDE interface. The title bar reads "Untitled - DrRacket". The menu bar includes "File", "Edit", "View", "Language", "Racket", "Insert", "Scripts", "Tabs", and "Help". The toolbar contains icons for "Untitled", "define ...", "Debug", "Macro Stepper", "Run", and "Stop". The main text area displays the following Scheme code:

```
Welcome to DrRacket, version 8.2 [cs].
Language: R5RS; memory limit: 128 MB.
> (define r #f)
> (define (f x)
  (set! x (+ x
            (call-with-current-continuation
              (lambda (ret)
                (set! r ret)
                0))))
  x)
> (f 100)
100
> (r 7)
107
> (r 7)
107
> (r 55)
155
> (define (g x)
  (set! x (+ (call-with-current-continuation
              (lambda (ret)
```

```
        (lambda (ret)
          (set! r ret)
          0))
      x))

x)
> (g 100)
100
> (r 7)
107
> (r 7)
114
> (r 55)
169
>
```

R5RS ▾ 34:2 P 406.06 MB 

```
(define r #f)

(define (g x)
  (set! x (+
    x ;; x из первого вызова g, т.к. будет сохранено в стеке
    (call-with-current-continuation
      (lambda (ret)
        (set! r ret)
        0))
    x ;; x из последнего вызова, т.к. set! изменит переменную сохраненную в континуации
  )))

x)
```

Author: Starovoytov Alexandr

Created: 2021-12-10 Fri 00:33