Иевлев Станислав Игоревич
Ведущий разработчик ООО «Альт Линукс»,
окончил математический факультет Московского педагогического государственного университета (МПГУ) по специальности «Математика и информатика».
Автор статей в области системного программирования и защиты информации.

Ваш новый язык Scheme – часть 2

1. Повторение – мать учения

Поскольку я не рассчитываю, что читатель ознакомился с предыдущей статьёй данной серии (№ 2, 2006 год), то привожу небольшую шпаргалку по базовым конструкциям языка программирования Scheme.

Базовый синтаксис:

```
(<функция> <аргумент> <аргумент> ...)
;комментарий
```

Например:

```
(+ 1 2 3 4)    ; сумма первых четырёх натуральных чисел
(+ (+ 1 2) 5 6) ; возможно вкладывать одни вызовы функций в другие
(string-append "hello" "world") ; результат — склеенная строка
                                ;  "helloworld"
```

Задание переменных и функций:

```
(define <имя> <значение>)
(define (<имя> <аргумент>...) <тело функции>)
;     функция возвращает значение последнего вычислительного
;      выражения в теле функции.
```

Например:

```
(define a 3)
(define b 4)
(define (square x) (* x x)) ; вычисляет квадрат числа
(define (+a x) (+ x a))
; да-да, можно давать и такие имена функциям
```

Дополнительные конструкции:

```
(if <условие> <действие при успехе условия> <действие при неудаче>)
(begin <первое действие> <второе действие> ....)
```

Пример:

```
(if (> a 0)
   (display "a > 0")) ; действие при неудаче можно не указывать
(begin (display 1)
        (display 2)
        (display 3)) ; последовательно выполнятся действия, и
                     ;на экран будет выведено: 123
```

2. И опять про повторение

Ну вот, теперь можно продолжать двигаться дальше. Если вам надо выполнить какое-то действие один раз, то вы просто его делаете один раз: `(display "hello")`. Если вам надо выполнить какое-то действие два раза, то вы просто повторяете его: `(display "hello") (display "hello")`. А что, если вам нужно повторять работу снова и снова? Тогда мы сделаем функцию, которая будет повторяться требуемое количество раз. Называться эта функция будет repeat, и у неё будет один параметр — количество повторов. Алгоритм следующий:

> Если количество повторов больше нуля, то:
> 1. выполняем действие
> 2. вызываем функцию repeat с количеством повторов меньшим на 1

```
(define (repeat number)
    (if (> number 0) ; если количество повторов не нулевое
      (begin (display "hello")  ; выполняем действие
      (repeat (- number 1))))) ; повторим действие
                               ; на единицу меньшее количество раз.
```

Попробуем. Запустим один из доступных вам интерпретаторов Scheme, например mzscheme, который можно найти в интернете по адресу http://www.plt-scheme.org/software/drscheme/ и установить на вашем компьютере.

```
> (define (repeat number)
          (if (> number 0)
             (begin (display "hello")
                 (repeat (- number 1)))))
> (repeat 0)
> (repeat 1)
hello
> (repeat 2)
hellohello
> (repeat 3)
hellohellohello
```

Упражнение 1: Попробуйте написать функцию, которая будет выводить на экран цифру "1" заданное количество раз.

Упражнение 2: Попробуйте написать функцию, которая будет выводить на экран натуральные числа от "1" до заданного числа. Порядок вывода чисел не важен.

Самое время вспомнить, что в Scheme можно передавать функцию в качестве параметра. Усовершенствуем функцию `repeat` так, чтобы мы смогли повторять произвольные действия заданное количество раз. Пусть новая версия принимает два параметра: первый — количество повторов, второй — функция, которую надо запустить.

```
(define (repeat number function)
(if (> number 0)
   (begin (function)
            (repeate (- number 1)))))
```

Теперь повторять можно разные действия:

```
(define (print-one) (display "1"))
(define (print-hello) (display "hello"))
(repeat 3 print-one) ; три раза выведет на экран "1"
(repeat 5 print-hello) ; пять раз выведет на экран "hello"
```

3. Принцип наименьших усилий

Не надо делать лишних действий, если их можно избежать. Последний вариант функции repeat хорош, но ... зачем каждый раз давать функции имя, если нужно просто её выполнить? А можно ли вообще создать функцию, но не давать ей имя? Оказывается можно. В языке есть специальная инструкция, которая говорит «создать функцию».

```
(lambda (<аргументы>) <тело функции> <последнее вычисленное
            значение возвращается>)
```

Пример:

```
(lambda (x) (* x x))    ; создать функцию, которая вычисляет
                        ; квадрат числа
(lambda (x y) (+ x y)) ;создать функцию, которая вычисляет сумму
                        ;двух чисел
(define square (lambda(x) (* x x)))
; создать функцию, которая вычисляет квадрат числа и назвать её square.
```

Оказывается, последняя конструкция то же самое, что и:

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

Вот мы с вами открыли ещё один способ определять функции с именами: сначала создаём, затем даём имя. Давайте-ка теперь «повторим» действия, не давая имена функциям:

```
(repeat 3 (lambda() (display "hello")) ) ; три раза выведем на
                                         ; экран "hello"
(repeat 5 (lambda() (display "1"))) ; пять раз выведем на экран "1"
```

Для `repeat` используются функции без параметров, поэтому в конструкции `lambda` пустая пара скобок.

4. Списки

Проведём следующую работу:

```
(define a 1)
(define b 2)
(define c 3)
(+ a 1)
(+ b 1)
(+ c 1)
```

А как бы нам сразу взять три числа и увеличить их на единицу одним махом? Для этого надо «связать» эти числа вместе. Один из способов склеивания — список. Создаётся список следующей конструкцией:

```
(list <элемент1> <элемент2> <элемент3> …)
```

Пример:

```
(define abc (list 1 1 1)) ; список из трёх единиц
(define lst1 (list 1 2 3)) ; список из трёх разных чисел
(define lst2 (list "hello" "my" "world")) ; список из строк
(define lst3 (list "hello" 1 "world" 3)) ; список из строк и чисел
(define lst4 (list (+ 1 0) 2 3)); элементы списка можно
                                 ;вычислять перед его созданием
```

Scheme также предоставляет функцию:

```
(map <функция> <список>)
```

Эта функция возвращает список, в котором каждый элемент есть результат применения функции "<функция>" к элементу исходного списка. Пример:

```
(define (inc x) (+ x 1)); увеличивает число на единицу
(map inc (list 1 1 1)); возвращает список из двоек
(map square (list 1 2 3)); возвращает список из квадратов
                          ; элементов, то есть 1, 4 и 9.
```

Вспомним про lambda и решим задачу, которую поставили себе в начале этого раздела:

```
(define abc (list 1 1 1))
(map (lambda(x) (+ x 1)) abc)
```

А можно даже и список не вводить как дополнительную переменную:

```
(map (lambda(x) (+ x 1))
     (list 1 1 1))
```

Упражнение 3: Пользуясь функциями write (для печати списка на экране) и map, напишите функцию, которая будет выводить на экран список, увеличенный на заданное число, например
(print-it 5 (list 1 2 3)) выведет "(6 7 8)"

5. Работа со списками

А как написать функцию, которая последовательно будет перемещаться по заданному списку и выводить каждый элемент этого списка? Для этого нам надо только знать следующие инструкции:

- `(null? <список>)` – проверяет, а есть ли ещё какие элементы в списке,
- `(car <список>)` – возвращает первый элемент списка,
- `(cdr <список>)` – возвращает список из всех элементов, кроме первого, а если больше ничего не осталось, то – пустой список.

Пример:

```
(car (list 1 2 3)) ; вернёт 1
(cdr (list 1 2 3)) ; вернёт список из 2 и 3, то есть (list 2 3).
```

Проще говоря, функция car возвращает голову списка, а `cdr` — оставшийся хвост списка. Имя функции `print-list`. Единственный параметр — исходный список. Алгоритм работы нашей функции следующий:

> Если список не пуст, то:
> 1. выводим голову списка.
> 2. вызываем `print-list` для хвоста списка

```
(define (print-list lst)
(if (not (null? lst))
   (begin (display (car lst))
          (newline)
          (print-list (cdr lst)))))
```

Поэкспериментируем в интерпретаторе:

```
> (print-list (list 1 2 3))
1
2
3
> (print-list (list "a"))
a
```

Упражнение 4: Поэкспериментируйте в интерпретаторе с функциями `"car"`, `"cdr"` и `"null?"`.

Упражнение 5: Напишите функцию, которая будет вычислять длину списка. Длина пустого списка — ноль.

Упражнение 6: Пользуясь изученными `"car"`, `"cdr"` и `"null?"` напишите функцию, которая будет применять заданную функцию к каждому элементу данного списка, например, `(for-each-element display (list 1 2 3))` должна напечатать `"123"`. Напишите новую версию `print-list`, пользуясь только что созданной функцией.

Упражнение 7: Напишите функцию, которая будет принимать на вход два списка и возвращать список из попарных сумм элементов, например, `"(plus (list 1 2) (list 5 6))"` вернёт список из 6 и 8.

Упражнение 8: Попробуйте решить упражнение 8 с помощью функции `map` — правильный ответ вас сильно удивит.