Лекция 5. Императивное программирование на языке Scheme

Коновалов А. В.

26 сентября 2022 г.

Императивное программирование на языке Scheme

До этого мы рассматривали декларативное программирование, в котором у нас не было:

- присваиваний,
- циклов,
- процедур с побочными эффектами,
- недетерминированных процедур — процедур, результат которых определяется не только значениями аргументов.

В Scheme есть средства не только декларативного программирования, но и императивного. Т.е. можно и присваивать переменным новые значения, и пользоваться процедурами, которые вызываются не только ради возвращаемого значения, но и дополнительных действий (побочного эффекта).

В Scheme не определён порядок вычисления аргументов в вызове процедуры. Но в императивном программировании порядок вычисления (а вернее, выполнения) операций существенен. Поэтому в первую очередь нам нужно средство упорядочивания выполнения операций.

1. `begin` (1)

Если мы имеем вызов вида

```
(f (g …))
```

то в Scheme гарантируется, что сначала вычислится `(g …)`,
а потом `(f …)`. (В Haskell не гарантируется.)

Но если мы имеем вызов вида

```
(f (g …) (h …))
```

то, что выполнится раньше — `g` или `h` — зависит от реализации.
Разные реализации Scheme могут вычислять аргументы справа налево или слева направо.

1. begin (2)

Но если нужно вывести на печать несколько значений, то порядок вызова будет существенен: функции должны вызваться в правильном порядке. Можно изловчиться, например, конструкцией `let*`:

```
(let* ((x (display "Hello, "))
       (y (display "World!")))
  #f)
```

Но это избыточно, т.к. в Scheme уже есть особая форма (`begin …`), гарантирующая порядок вычисления:

```
(begin
  (display "Hello, ")
  (display "World!"))
```

(На самом деле `begin` может быть библиотечным макросом, который неявно трансформируется в тот же `let*`).

1. begin (3)

`begin` выполняет действия в том порядке, в котором они записаны.

Результатом `begin`'а является результат последнего действия.

```
(begin (* 7 3) (+ 6 4))        → 10
```

Результат умножения будет отброшен, умножение тут вообще бессмысленно.

Неявный `begin` (1)

Некоторые конструкции Scheme позволяют записывать несколько действий подряд, например `lambda`, `define`, определяющий процедуру, `cond`, `let`, `let*`, `letrec`.

```
; синтаксический сахар

(lambda (x y)
  (display x)
  (display y))


(define (f x y)
  (display x)
  (display y)
  (+ x y))
```

```
; эквивалентен

(lambda (x y)
   (begin
      (display x)
      (display y)))


(define (f x y)
   (begin
      (display x)
      (display y)
      (+ x y)))
```

Неявный begin (2)

```
; синтаксический сахар
(let ((x 100)
      (y 200))
  (display x)
  (display y)
  (* x y))
```

```
; эквивалентен
(let ((x 100)
      (y 200))
  (begin
    (display x)
    (display y)
    (* x y)))
```

```
; синтаксический сахар
(cond ((> x y)  (display x) (- x y))
      …)
; эквивалентен
(cond ((> x y)  (begin (display x) (- x y)))
      …)
```

2. Присваивания, `set!` (1)

Синтаксис:

```
(set! ‹имя переменной› ‹выражение›)
```

Переменной может быть как имя, объявленное при помощи `define`, так и параметр процедуры или имя, определённое `let`, `let*`, `letrec`.

Например

```
(define counter 0)
counter                                  → 0
(set! counter 100)
counter                                  → 100
```

2. Присваивания, set! (2)

```
(define counter 0)

(define (next)
  (set! counter (+ counter 1))
  counter)

(next)                          → 1
(next)                          → 2
(next)                          → 3
counter                         → 3
(set! counter 7)
(next)                          → 8
```

2. Присваивания, `set!` (3)

Статические переменные в Scheme

В языке Си есть понятие **статическая переменная** — глобальная переменная, видимость которой ограничена одной функцией. Объявляется она с использованием ключевого слова `static`:

```c
void f() {
  int x = 0;
  static int y = 0;

  x = x + 1;
  y = y + 1;

  printf("x = %d\n", x);
  printf("y = %d\n", y);
}
```

```c
int main(int argc, char **argv)
  f();
  f();
  f();

  return 0;
}
```

Напечатается:

```
x = 1
y = 1
x = 1
y = 2
x = 1
y = 3
```

Значение статической переменной сохраняется между вызовами функции (сравните выше поведение x и y).

В языке Scheme статических переменных нет, но есть идиома (приём программирования), позволяющая их имитировать: т.е. создавать переменные, видимые только внутри функции, но при этом сохраняющие значение между вызовами.

Вспомним, что конструкция

```
(define (f x y)
  ‹тело процедуры›)
```

есть синтаксический сахар для

```
(define f
  (lambda (x y)
    ‹тело процедуры›))
```

Что будет, если мы эту лямбду обернём в let-конструкцию?

```
(define f
  (let (‹объявления каких-то переменных›)
    (lambda (x y)
      ‹тело процедуры›)))
```

Let-конструкция свяжет с переменными значения и вернёт лямбду как свой результат. Переменная `f` будет связана с лямбдой. Что же будет с переменными?

Эти переменные будут видимы внутри лямбды, не будут видимы вне конструкции `let`, их значения будут сохраняться между вызовами.

Эти переменные будут вести себя как статические переменные в Си.

Перепишем пример с `(next)`, чтобы переменная `counter` была статической.

```
(define next
  (let ((counter 0))
    (lambda ()
      (set! counter (+ counter 1))
      counter)))

(next)                          ; → 1
(next)                          ; → 2
(next)                          ; → 3
```

3. Цикл do (1)

Цикл do используется редко, выглядит он вот так:

```
(do ((‹перем› ‹нач› ‹модиф [необ.]›)   ; почти как в let
     ...
     (‹перем› ‹нач› ‹модиф [необ.]›))
    (‹усл. выраж.› ‹возврат [необ.]›)
    ‹выраж›
    ...
    ‹выраж›)
```

3. Цикл do (2)

Пример:

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
   ((= i 5) vec)
 (vector-set! vec i i))                    → #(0 1 2 3 4)
```

Две переменные цикла: `vec` и `i`. `vec` присваивается новый вектор, `i` — `0`, `vec` не меняется (модификация переменной отсутствует), `i` увеличиваетя на 1, условие выхода — `(= i 5)`, возвращаемое значение — `vec`. В теле цикла в `i`-ю позицию вектора присваивается число `i`.

4. Изменяемые структуры данных (1)

В Scheme некоторые значения в памяти можно менять. Прежде всего это вектор — его элементам можно присваивать новые значения при помощи `vector-set!`. Но можно менять и cons-ячейки.

Есть такие функции:

```
(set-car! ‹cons-ячейка› ‹значение›)
(set-cdr! ‹cons-ячейка› ‹значение›)
```

4. Изменяемые структуры данных (2)

Например, можно создать кольцевой список:

```
(define loop-xs '(a b c))
(set-cdr (cdr loop-xs) loop-xs)
```

Получится кольцевой список вида `(a b a b a b …)`.

```
(list-ref loop-xs 0)          ; → a
(list-ref loop-xs 37)         ; → b
(length loop-xs)              ; → зависло
```

4. Изменяемые структуры данных (3)

Вообще, рекомендуется работать со списками как с неизменяемыми данными. Если содержимое списков менять на месте при помощи `set-car!` или `set-cdr!`, то можно сильно запутать программу, поскольку разные списки могут разделять общий хвост.

```
(define xs '(a b c d))
(define ys (append '(1 2 3) xs))
(define zs (cons 'x xs))
(define us (append xs xs))

(set-car! xs 'hello)

ys                                  ; → (1 2 3 hello b c d)
zs                                  ; → (x hello b c d)
us                                  ; → (a b c d hello b c d)
```

Наиболее ожидаемым было изменение `zs`. Наиболее неожиданным — `us`.

4. Изменяемые структуры данных (4)

Вспомним, что семантику `append` можно описать примерно так:

```
(define (append xs ys)
  (if (null xs)
      ys
      (cons (car xs) (append (cdr xs) ys))))
```

Т.е. второй аргумент `ys` используется как есть, ему в начало приписываются (`cons`'ами) звенья со значениями элементов списка первого аргумента. Таким образом, результатом (`append as bs`) будет список, часть звеньев которого будет общей со списком `bs`.