Лекция 3. Функции высшего порядка

Значения языка Scheme

- Числа: `1`, `1.0`, `6.022e23`, `1/3`…
- Строки: `"Scheme"`
- Логический тип: `#t`, `#f`
- Литерный (character) тип: `#\a` `#\newline`…
- Символьный (symbol) тип: `'x`, `'sin`
- Списки
- Вектора
- **Процедурный тип**
- Продолжения (continuations)

Синтаксис

```
(lambda (аргументы) выражение)
```

Конструкция lambda создаёт безымянную процедуру. Эту процедуру можно вызвать:

```
((lambda (x y) (+ x y)) 10 13)
;          ↑__ формальные параметры
; фактические параметры __↑
```

При вызове процедуры создаются новые переменные, соответствующие формальным параметрам и они связываются с фактическими параметрами.

```
(define f
  (lambda (x y) (+ x y)))
(f 10 13)
```

Синтаксический сахар:

```
(define f (lambda (парам) выраж))
```

эквивалентно

```
(define (f парам) выраж)
```

Передача процедуры как параметра:

```
(define (g f)
  (f 10 13))

(g (lambda (x y) (+ x y)))
(g +)
```

Возврат процедуры из процедуры

```
(define (select n)
  (if (> n 0)
      (lambda (x y) (+ x y))
      (lambda (x y) (- x y))))

((select +1) 10 13)
((select -1) 100 50)
```

Замыкания, области видимости и захват переменных

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

(define f1 (f 1))
(define f7 (f 7))

(f1 10) → 11
(f7 10) → 17
```

Конструкции `let`, `let*` и `letrec`

Конструкции let, let* и letrec: let

```
(let ((var1 expr1)
      (var2 expr2)
      …
      (varN exprN))
  выражение)
```

В теле let-выражения можно использовать переменные var1…varN. Выражения expr1…exprN могут вычисляться в произвольном порядке — порядок их вычисления не определён.

НО! Внутри expr1…exprN нельзя использовать var1…varN.

Конструкции `let`, `let*` и `letrec`: `let*`

```
(let* ((var1 expr1)
       (var2 expr2)
       …
       (varN exprN))
  body)
```

Переменную `varK` можно использовать не только в теле `let*`, но и в `exprM`, где `M` > `K`.

Выражения `expr1`…`exprN` вычисляются *последовательно.*

Конструкции `let`, `let*` и `letrec`: `letrec`

```
(letrec ((var1 expr1)
         (var2 expr2)
         …
         (varN exprN))
 body)
```

Внутри любого `exprK` можно использовать любую переменную из `var1`…`varN`.

Конструкции let, let* и letrec: let с рекурсией

```
(let proc-name ((var1 expr1)
                (var2 expr2)
                …
                (varN exprN))
  body)
```

Внутри тела let-выражения можно вызывать процедуру `proc-name`, передавая ей `N` параметров.

Это выражение эквивалентно

```
(letrec ((proc-name
          (lambda (var1 … varN)
            body)))
  (proc-name expr1 … exprN))
```

Конструкции `let`, `let*` и `letrec`: синтаксический сахар

Все эти конструкции являются синтаксическим сахаром. Для примера:

```
(let ((var1 expr1)
      (var2 expr2)
      …
      (varN exprN))
  выражение)
```

эквивалентна

```
((lambda (var1 … varN)
   выражение)
 expr1 … exprN)
```

Рекурсия, итерация и хвостовая рекурсия

Рекурсия

$N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (N-1) \cdot N$

$0! = 1$

$N! = N \cdot (N-1)!$

Рекурсия — делим задачу на меньшие подзадачи, подобные исходной.

Итерация — задача делится на некоторое количество одинаковых подзадач, одинаковых шагов, приближающих к цели.

Рекурсия и итерация

Как итерацию выразить через рекурсию?

Итерация: пока цель не достигнута, повторять шаг вычисления.

Рекурсия:

- Цель достигнута?
  - Да — прекратить вычисления, вернуть результат.
  - Нет — выполнить один шаг вычисления, выполнить рекурсивный вызов.

Итерация → рекурсия

Факториал в терминах итерации:

```
int fact(int N) {
    int res = 1;
    int i = 1;
    while (i ⩽ N) {
        res = res * i;
        i = i + 1;
    }
    return res;
}
```

Перепишем эту функцию на язык Scheme

Итерация → рекурсия

- Для цикла заводим вспомогательную процедуру.
- Переменные цикла становятся параметрами процедуры.
- Тело цикла превращается в рекурсивный вызов.
- Инициализация переменных цикла становится вызовом рекурсивной процедуры.

Итерация → рекурсия

```c
int fact(int N) {
    int res = 1;
    int i = 1;
    while (i ⩽ N) {
        res = res * i;
        i = i + 1;
    }
    return res;
}
```

```scheme
(define (fact N)
  (define (loop i res)
    (if (⩽ i N)
        (loop (+ i 1)
              (* res i))
        res))
  (loop 1 1))
```

Итерация → рекурсия

Другой способ:

```
int fact(int N) {
    int res = 1;
    int i = 1;
    while (i ⩽ N) {
        res = res * i;
        i = i + 1;
    }
    return res;
}
```

```
(define (fact N)
  (let loop ((i 1)
             (res 1))
    (if (⩽ i N)
        (loop (+ i 1)
              (* res i))
        res)))
```

Хвостовая рекурсия

Хвостовой вызов — вызов, который является последним, результат этого вызова становится результатом работы функции.

```
(define (f x y z)
  (if (a)
      (B x (c y))
      (D (if (e)
             (g)
             (h)))))
```

(Вызовы `B` и `D` — хвостовые)

Оптимизация хвостовой рекурсии

В языке Scheme заложена оптимизация хвостового вызова, т.н. оптимизация хвостовой рекурсии. Фрейм стека (см. лекцию про продолжения) вызывающей процедуры замещается фреймом стека вызываемой процедуры.

Если хвостовой вызов является рекурсивным, фреймы стека не накапливаются.

Хвостовая рекурсия в языке Scheme эквивалента итерации по вычислительным затратам.

Пример хвостовой рекурсии

Рекурсивный факториал:

```
(define (fact N)
  (if (> N 0)
      (* (fact (- N 1)) N)
      1))
```

Итеративный факториал:

```
(define (fact N)
  (define (loop i res)
    (if (⩽ i N)
        (loop (+ i 1)
              (* res i))
        res))
  (loop 1 1))
```

Оптимизация хвостовой рекурсии изнутри

Рекурсивная функция

```
int loop(int N, int i, int res) {
    if (i ⩽ N) loop(N, i + 1, res * i);
    else return res;
}
```

преобразуется примерно в такой код

```
int loop(int N, int i, int res) {
LOOP:
    if (i ⩽ N) {
        res = res * i;
        i = i + 1;
        goto LOOP;
    } else return res;
}
```