Лекция 4. Списки, вычислительная сложность, равенство

Коновалов А. В.

19 сентября 2022 г.

Списки

Списки

LISP — List processing, обработка списков. Список — основная структура данных языка Scheme.

`(1 2 3 4)` — список из четырёх чисел.

Создание списка

```
(list <элементы>)
```

```
(list 1 2 3 4)     →     (1 2 3 4)
(list)             →     ()
```

Операции над списками

cons — конструирование

`(cons <голова> <хвост>)` → `<список>`

Создаёт новый список из некоторого значения («головы») и другого списка («хвоста»). Первым элементом нового списка будет «голова», последующими — хвост.

```
(cons 1 (list 2 3 4))   →   (1 2 3 4)
```

Операции над списками

car, cdr, null?

```
(car <список>) → <голова списка>
(cdr <список>) → <хвост списка>
(null? <список>) → <bool>
```

Это селекторы, запрашивают голову и хвост списка, список должен быть непустым.

```
(car (list 1 2 3 4)) → 1
(cdr (list 1 2 3 4)) → (2 3 4)

(null? (list 1 2 3 4)) → #f
(null? (list)) → #t
```

Операции над списками

Запись списка через цитирование

Пустой список, запись списка через цитирование

```
'() – пустой список
```

Запись списка при помощи цитирования:

```
'(1 (2 3 4) 5 6)
```

Вложенные списки:

```
(list (list 1 2 3) (list 4 5 6)) → ((1 2 3) (4 5 6))
'((1 2 3) (4 5 6))               → ((1 2 3) (4 5 6))
```

Операции над списками

Список можно связать с переменной:

```
(define L '(1 2 3))

(define x 1)

(define (f y)
  (list x y))

(define (f y)
  (cons x (cons y '())))

'(x y)
```

Операции над списками

`length` и `append`

Встроенная функция `length`

```
(length '(1 1 2 1)) → 4
```

Встроенная функция `append` — конкатенация списков:

```
(append '(1 2 3) '(4 5 6)) → (1 2 3 4 5 6)

(append '(1 2) '(3 4) '(5 6)) → (1 2 3 4 5 6)
```

Списков не существует — cons-ячейки или пары

Объект, который строится функцией `cons` — т.н. cons-ячейка или пара. Аргументами функции `cons` могут быть любые объекты.

Правильный список — это или пустой список, или cons-пара, вторым элементом которой является правильный список.

```
(cons 1 2)                  → (1 . 2)    ;; неправильный список
(cons 1 (cons 2 '())) → (1 2)      ;; правильный список

(cons 1 (cons 2 (cons 3 4))) → (1 2 3 . 4)  ;; тоже неправиль
```

Пример. Как могла бы быть определена встроенная функция `length`:

```
(define (length xs)
  (if (null? xs)
      0
      (+ 1 (length (cdr xs)))))

(define (length xs)
  (define (loop len xs)
    (if (null? xs)
        len
        (loop (+ len 1) (cdr xs))))
  (loop 0 xs))
```

pair

Функция `pair?` возвращает истину, если аргумент — cons-ячейка.

```
(pair? '(1 . 2))          → #t
(pair? (list 1 2 3))      → #t
(pair? 100500)            → #f
```

Встроенная функция `map`

Используется для того, чтобы единообразно преобразовать все элементы списка, принимает процедуру и исходный список, строит новый список той же длины, что и исходный, каждый элемент этого списка является результатом применения процедуры к элементу исходного списка.

```
(define (square x) (* x x))
(map square '(1 2 3 4 5))  →  '(1 4 9 16 25)
```

Расширенный вариант использования `map`:

```
(map
  (lambda (x y) (+ (* 2 x) (* 3 y)))
  '(1 2 1 2)
  '(2 3 2 3 2)
) →
  '(8 13 8 13)
```

Функцию `map` («простой вариант») можно описать как:

```
(define (map f xs)
  (if (null? xs)
      '()
      (cons (f (car xs)) (map f (cdr xs)))))
```

Процедуры с переменным числом параметров

Безымянная процедура, принимающая произвольное число аргументов:

```
(lambda xs …)
```

`xs` — список аргументов

```
((lambda xs xs) 1 2 3 4) → (1 2 3 4)
```

Безымянная процедура, принимающая n+ аргументов:

```
(lambda (a b c . xs) …)

((lambda (a b c . xs)
   (list (+ a b c) xs))
 1 2 3 4 5) →
   (6 (4 5))
```

Именованная процедура:

```
(define (f <фиксированные параметры> . <список параметров>)
  …)

(define (f a b c . xs)
  (list (+ a b c) xs))

(f 1 2 3 4 5) → (6 (4 5))

(define (f . xs)
  (list xs xs))

(f 1 2 3) → ((1 2 3) (1 2 3))
```

Функцию `list` можно описать так:

```
(define (list . xs) xs)
```

Пример:

```
((lambda x x)) → ()
```

Вычислительная сложность

Вычислительная сложность

Вычислительная сложность — асимптотическая оценка времени работы программы. Асимптотическая, значит, нас интересует не конкретное время, а поведение.

`T(<данные>)` — функция, возвращающая точное значение времени работы программы на конкретных входных данных.

Асимптотическая оценка `O(f(<данные>))` показывает, что функция `T(•)` при росте входных данных ведёт себя как функция `f(•)` с точностью до некоторого постоянного сомножителя.

Т.е. существует такое `k`, что

```
lim T(data) / f(data) → k
```

при росте аргумента `data`.

Вычислительная сложность

Оценку вычислительной сложности для некоторого алгоритма и некоторого абстрактного вычислителя обычно оценивают в числе элементарных команд этого абстрактного вычислителя.

Для Scheme элементарными операциями считаются вызов функции, `cons`, `car`, `cdr`, получение значения переменной, создание процедуры (`lambda`), объявление глобальной переменной (`define`), присваивание переменной (`set!`), арифметические действия с фиксированным числом операндов (не свёртка!), call/cc (создание и переход на продолжение), `delay`, `force`, `null?` (и другие встроенные предикаты), `if`, `cond`.

Вычислительная сложность

Встроенные функции могут иметь разную сложность!

Например,

- `(map f xs)` — `O(len(xs)×T(f))`, где `T(f)` — среднее время работы `(f x)`.
- `(length xs)` — `O(len(xs))`.
- `(append xs ys)` — `O(len(xs))`.

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

Проверка на равенство

- `eqv?` — атомы сравнивает по значению, сложные типы данных (списки, векторы, lambda) — по ссылке.
- `eq?` — может и атомы сравнивать по ссылке.
- `equal?` — сравнивает аргументы по значению.
- `=` — равенство чисел. Может сравнивать числа разных типов.
- Функции сравнения отдельных типов вроде `string=?`…

Проверка на равенство

Функция `equal?` медленная, т.к. сравнивает аргументы по значению, в частности, для списков сравнивает их содержимое. Но она наиболее предсказуемая.

Функции `eq?` и `eqv?` работают быстро, но могут давать неожиданные результаты.

```
(define x …)
(define y x)

(eq? x y)      → #t
(eqv? x y)     → #t
(equal? x y)   → #t
```