Практика 10. Макросы и потоки

материалы преподавателя

Макросы

До этого момента была возможность лишь определять собственные процедуры в Scheme с использованием особой формы `define`. При выполнении этих процедур интерпретатор следует правилам обработки вызывающих выражений, которые включают вычисление всех операндов.

Тебе уже известно, что особые формы не следуют правилам вычисления вызывающих выражений. Вместо этого каждая особая форма обрабатывается по своим правилам, которые могут требовать вычисления не всех аргументов. Не правда ли было бы круто иметь возможность объявлять свои собственные особые формы с собственными правилами вычисления. Посмотри на следующий пример, в котором осуществлена попытка создать функцию вычисляющую заданное выражение дважды:

```
scm> (define (twice f) (begin f f))
twice
scm> (twice (print 'woof))
woof
```

Поскольку `twice` является обычной процедурой, её вызов будет выполняться по правилам для вызывающих выражений: сначала будет вычислено значение оператора, затем будут вычислены операнды. Это означает, что `woof` появится в выводе при вычислении операнда `(print 'woof)`. Внутри тела `twice` имя `f` будет связано со значением `undefined`, так что при вычислении выражения `(begin f f)` ничего не произойдёт.

Как же это исправить? Нужно предотвратить вычисление выражения `(print 'woof)` до попадания в тело процедуры `twice`. Именно в таких случаях может помочь особая форма `define-macro` (её синтаксис в точности совпадает с формой `define`):

```
scm> (define-macro (twice f) (list 'begin f f))
twice
```

Особая форма `define-macro` позволяет определять макросы. Макрос — это сочетание невыполненных входных выражений с другими выражениями. При вызове макроса операнды не вычисляются, а рассматриваются как Scheme-данные. То есть операнды в виде вызывающих выражений или особых форм обрабатываются как Scheme-списки.

При вызове `(twice (print 'woof))` символ `f` связан со списком `(print 'woof)`, а не со значением `undefined`. Внутри тела `define-macro` эти выражения могут быть «вставлены» в другие Scheme-выражения. В рассматриваемом примере `begin`-выражение должно выглядеть так:

```
(begin (print 'woof) (print 'woof))
```

Если рассматривать это выражение с точки зрения данных, то это просто список с

элементами `begin`, `(print 'woof)`, `(print 'woof)`. То есть это в точности значение выражения `(list 'begin f f)`. Теперь при вызове `twice` этот список будет обработан как выражение — `(print 'woof)` напечатается два раза.

```
scm> (twice (print 'woof))
woof
woof
```

Подводя итог, макросы вызываются похожим с процедурами способом, но правила вычисления макросов отличаются. Например, лямбда-процедура вычисляется таким образом:

- вычислить значение оператора;
- вычислить значения операндов;
- применить значение оператора к значениям операндов.

Правила же вычисления макросов таковы:

- вычислить значение оператора;
- применить значение оператора к невычисленным операндам;
- вычислить значение выражения возвращённого из макроса в фрейме, в котором был вызван макрос.

Квазицитирование

Припомни, что особая форма `quote` предотвращает выполнение интерпретатором Scheme указанного выражения. Похоже, цитаты удобно использовать при построении сложных Scheme-выражений со множеством вложенных списков.

Рассмотри другую версию макроса `twice`:

```
(define-macro (twice f)
  '(begin f f))
```

Казалось бы, этот вариант идентичен предыдущему, но это не так. Особая форма `quote` предотвращает любые вычисления, то есть результирующее выражение будет `(begin f f)`. Это не то, что требуется.

Квазицитирование позволяет создавать списки похожим на `quote` способом, но эта форма дополнительно позволяет указывать, какие подвыражения следует вычислять, а какие нет.

То есть с первого взгляда особая форма `quasiquote` (или сокращённо `` ` ``) ведёт себя ровно как `quote` (сокращённо `'`). Однако использование квазицитат позволяет для выбранных выражений делать *расцитирование* — особая форма `unquote` (или сокращённо `,`). Расцитированное выражение вычисляется и его значение вставляется в квазицитату.

Использование этих форм значительно облегчает создание макросов.

Последний вариант `define-macro`:

```
(define-macro (twice f)
  `(begin ,f ,f))
```

Квазицитирование относится не только к определению макросов. Его можно использовать в любой ситуации, где требуется нерекурсивно задавать списки, зная их структуру. Например, вместо `(list x y z)` можно смело писать `(,x ,y ,z)`.

Вопрос 1

Напиши макрос, который принимает выражение и возвращает лямбда-процедуру без параметров, телом которой является исходное выражение

```
(define-macro (make-lambda expr)
```

Ответ

```
With quasiquotes:
(define-macro (make-lambda expr) `(lambda () ,expr))

With the list constructor:
(define-macro (make-lambda expr) (list 'lambda (list) expr))
```

```
scm> (make-lambda (print 'hi))
(lambda () (print (quote hi)))
scm> (make-lambda (/ 1 0))
(lambda () (/ 1 0))
scm> (define print-3 (make-lambda (print 3)))
print-3
scm> (print-3)
3
```

Вопрос 2

Напиши макрос, который принимает выражение `expr` и число `n` и повторяет входное выражение `n` раз. Например, выражение `(repeat-n expr 2)` должно вести себя так же, как `(twice expr)`. Учти, что на месте `n` может находиться комбинация, то есть `(+ 1 2)`, которую нужно вычислить, чтобы макрос не посчитал второй аргумент списком.

Дополни приведённую реализацию, используя функцию `replicate`. Эта функция принимает значение `x` и число `n` и возвращает список, в котором `x` повторено `n` раз.

```
(define (replicate x n)
    (if (= n 0) nil
        (cons x (replicate x (- n 1)))))

(define-macro (repeat-n expr n)
```

Ответ

```
(define-macro (repeat-n expr n)
    (cons 'begin (replicate expr (eval n))))
```

```
scm> (repeat-n (print '(resistance is futile)) 3)
(resistance is futile)
(resistance is futile)
(resistance is futile)
scm> (repeat-n (print (+ 3 3)) (+ 1 1)) ①
6
6
```

① Второй операнд тут вызывающее выражение.

Вопрос 3

Напиши макрос, который принимает два выражения и совершает с ними операцию `or` (вычисляя только те выражения, которые требуется). Однако это нужно сделать не используя особую форму `or`. Также можно считать, что имя `v1` не встречается нигде за пределами макроса.

```
(define-macro (or-macro expr1 expr2)

    `(let ((v1 ____________________________________________________________))

       (if ____________________________________________________________

       ____________________________________________________________)))
```

```
(define-macro (or-macro expr1 expr2)
    `(let ((v1 ,expr1))
       (if v1 v1 ,expr2)))
```

```
scm> (or-macro (print 'bork) (/ 1 0))
bork
scm> (or-macro (= 1 0) (+ 1 2))
3
```

Потоки

В Python можно использовать итераторы для представления бесконечных последовательностей (например, генератор всех натуральных чисел). Однако в Scheme нет итераторов. Посмотри, что получится, если использовать Scheme-список для представления бесконечной последовательности натуральных чисел:

```
scm> (define (naturals n)
    (cons n (naturals (+ n 1))))
naturals
scm> (naturals 0)
Error: maximum recursion depth exceeded ①
```

① Ошибка: достигнута максимальная глубина рекурсии.

Поскольку `cons` — обычная процедура и оба её операнда должны быть вычислены до создания пары, то невозможно создать бесконечную последовательность целых чисел в Scheme-списке. К счастью, интерпретатор Scheme поддерживает *потоки (streams)*, которые являются *ленивыми* Scheme-списками. В таких списках первый элемент непосредственно представлен значением, а остальная часть списка вычисляется при необходимости. Вычисление значения только в случае, когда оно необходимо, называют *ленивым вычислением*.

```
scm> (define (naturals n)
    (cons-stream n (naturals (+ n 1))))
naturals
scm> (define nat (naturals 0))
nat
scm> (car nat)
0
scm> (cdr nat)
#[promise (not forced)]
scm> (car (cdr-stream nat))
1
scm> (car (cdr-stream (cdr-stream nat))) 2
```

В примере использована особая форма для создания потока `cons-stream`:

```
(cons-stream <операнд_1> <операнд_2>)
```

Особая форма `cons-stream` является особой, поскольку второй операнд при обработке выражения не вычисляется. Интерпретатор Scheme действует так:

- Вычислить первый операнд.
- Создать *промис* (объект отложенного исполнения), содержащий второй операнд.
- Вернуть пару, содержащую значение первого операнда и промис.

Для фактического получения остальной части потока нужно вызвать `cdr-stream`, чтобы выполнился промис. Заметь, что этот аргумент будет вычисляться только единожды, поскольку результат будет храниться в промисе; последующие вызовы `cdr-stream` будут возвращать результат без перерасчёта. Такой механизм позволяет эффективно работать с бесконечными потоками. Посмотри, как он себя будет вести с нечистой функцией:

```
scm> (define (compute-rest n)
...> (print 'evaluating!)
...> (cons-stream n nil))
compute-rest
scm> (define s (cons-stream 0 (compute-rest 1)))
s
scm> (car (cdr-stream s))
evaluating!
1
scm> (car (cdr-stream s)) 1
```

В примере выражение `compute-rest 1` вычисляется только один раз при первом вызове `cons-stream`, то есть символ `evaluating!` будет выведен только один раз.

При отображении потока первый элемент отделяется от промиса точкой (это показывает, что они являются частями одной пары, в которой `cdr` — промис). Если промис ещё не был вычислен вызовом `cdr-stream`, то он не решён (not forced), иначе он считается решённым (forced)

```
scm> (define s (cons-stream 1 nil))
s
scm> s
(1 . #[promise (not forced)])
scm> (cdr-stream s) ①
scm> s
(1 . #[promise (forced)])
```

① Возвращает nil.

Потоки очень похожи на Scheme-списки тем, что тоже являются рекурсивными структурами. Точно так же, как `cdr` от списка — или другой список, или `nil`, `cdr-stream` от потока будет или другим потоком, или `nil`. Разница лишь в том, что аргументы `cons` вычисляются до создания пары, а второй аргумент `cons-stream` не вычисляется до первого вызова `cdr-stream`.

Вот краткое изложение:

- `nil` — это пустой поток;
- `cons-stream` создаёт поток, содержащий значение первого операнда и промис для вычисления второго операнда;
    - `car` возвращает первый элемент потока;

- `cdr-stream` вычисляет и возвращает остаток потока.

Вопрос 4

При выполнении следующих заданий помни, что у потоков есть два важных свойства:

- Ленивое вычисление — остаток потока не будет вычислен, пока не понадобится.
- Запоминание — всё, что вычислено, не будет перевычисляться.

Приведенные ниже примеры используют эти свойства «наполную». В большинстве практических случаев использования, переопределять функцию, вычисляющую остаток потока ни к чему.

```
scm> (define (has-even? s)
    (cond ((null? s) #f)
        ((even? (car s)) #t)
        (else (has-even? (cdr-stream s)))))
has-even?
scm> (define (f x) (* 3 x))
f
scm> (define nums (cons-stream 1 (cons-stream (f 3) (cons-stream (f 5) nil))))
nums
scm> nums
```

```
(1 . #[promise (not forced)])
```

```
scm> (cdr nums)
```

```
#[promise (not forced)]
```

```
scm> (cdr-stream nums)
```

```
(9 . #[promise (not forced)])
```

```
scm> nums
```

```
(1 . #[promise (forced)])
```

```
scm> (define (f x) (* 2 x))
f
scm> (cdr-stream nums)
```

```
(9 . #[promise (not forced)])
```

```
scm> (cdr-stream (cdr-stream nums))
```

```
(10 . #[promise (not forced)])
```

```
scm> (has-even? nums)
```

```
True
```

Вопрос 5

Применение потоков может оказаться непростым! Сравни две следующие реализации `filter-stream`. Первая из них корректная, а вторая — нет. В чём проблема со второй?

```
; Корректная
(define (filter-stream f s)
    (cond
        ((null? s) nil)
        ((f (car s)) (cons-stream (car s) (filter-stream f (cdr-stream s))))
        (else (filter-stream f (cdr-stream s)))))

; Некорректная
(define (filter-stream f s)
    (if (null? s) nil
        (let ((rest (filter-stream f (cdr-stream s))))
            (if (f (car s))
                (cons-stream (car s) rest)
                rest))))
```

Ответ

```
Evaluating rest will result in infinite recursion if s is an infinite stream! In the
body of filter-stream, rest is always computed before cons-stream can delay the
evaluation.

Another way of thinking about this is that everything in the body of the let doesn't
matter. All we will be doing is repeatedly doing the recursive call on filter-stream.
```

Вопрос 6

Напиши процедуру `map-stream`, которая принимает процедуру `f` и поток `s`. Функция должна возвращать новый поток с элементами исходного потока, к которым применили процедуру `f`.

```
(define (map-stream f s)
```

Ответ

```
    (if (null? s)
        nil
        (cons-stream (f (car s)) (map-stream f (cdr-stream s))))

Для сравнения обычный map:

(define (map f s)
    (if (null? s)
        nil
        (cons (f (car s)) (map f (cdr s)))))
```

```
scm> (define evens (map-stream (lambda (x) (* x 2)) nat))
evens
scm> (car (cdr-stream evens))
2
```

Вопрос 7

Напиши процедуру `slice`, которая принимает поток `s`, начало `start` и конец `end`. Процедура должна возвращать Scheme-список, содержащий все элементы между индексами `start` и `end` (`end` не включается). Если поток закачивается до `end`, можешь вернуть `nil`.

```
(define (slice s start end)
```

Ответ

```
    (cond
        ((or (null? s) (= end 0)) nil)
        ((> start 0)
            (slice (cdr-stream s) (- start 1) (- end 1)))
        (else
            (cons (car s)
                (slice (cdr-stream s) (- start 1) (- end 1))))))
```

```
scm> (define nat (naturals 0)) ①
nat
scm> (slice nat 4 12)
(4 5 6 7 8 9 10 11)
```

① Процедура `naturals` определена выше.

Вопрос 8

Поскольку потоки вычисляют следующий элемент только при необходимости, есть возможность комбинировать бесконечные потоки для получения интересных результатов! Попробуй использовать их для вычисления твоих любимых последовательностей. Ниже определена процедура `combine-with` вместе с примером её использования для определения потока чётных чисел.

```
(define (combine-with f xs ys)
    (if (or (null? xs) (null? ys))
        nil
        (cons-stream
          (f (car xs) (car ys))
          (combine-with f (cdr-stream xs) (cdr-stream ys)))))

scm> (define evens (combine-with + (naturals 0) (naturals 0)))
evens
scm> (slice evens 0 10)
(0 2 4 6 8 10 12 14 16 18)
```

В следующих вопросах можешь использовать поток `naturals` и процедуру `combine-with`.

A)

```
(define factorials
```

Ответ

```
   (cons-stream 1 (combine-with * (naturals 1) factorials)))
```

```
scm> (slice factorials 0 10)
(1 1 2 6 24 120 720 5040 40320 362880)
```

Б)

```
(define fibs
```

Ответ

```
    (cons-stream 0
        (cons-stream 1
            (combine-with + fibs (cdr-stream fibs)))))
```

```
scm> (slice fibs 0 10)
(0 1 1 2 3 5 8 13 21 34)
```

Вопрос 9

Используя макрос `make-lambda` (Вопрос 1), определи макрос `make-stream`, который возвращает пару элементов. Второй элемент не вычисляется до вызова `cdr-stream`. Также определи процедуру `cdr-stream`, которая принимает поток, возвращаемый `make-stream`, и возвращает результат вычисления второго элемента пары.

В отличие от встроенных потоков, `cdr-stream` в этой версии будет вычислять результат множество раз.

```
(define-macro (make-stream first second)

    ------------------------------------------------------------------------
```

```
(define-macro (make-stream first second)
    `(list ,first (make-lambda ,second)))
```

```
(define (cdr-stream stream)

    ------------------------------------------------------------------------
```

```
(define (cdr-stream stream)
    ((car (cdr stream))))
```

```
scm> (define a (make-stream (print 1) (make-stream (print 2) nil)))
1
a
scm> (define b (cdr-stream a))
2
b
scm> (cdr-stream b)
()
```

Вопрос 10

Теперь нужно представить последовательность простых чисел как бесконечный поток! Определи функцию `sieve`, которая принимает поток увеличивающихся чисел и возвращает поток, содержащий только те числа, которые не делятся на на предыдущие числа без остатка. Ты можешь определить процедуру `primes`, *просеивая* все натуральные числа, начиная с двух. Посмотри, что такое Решето Эратосфена.

Возможно тебе пригодится процедура `filter-stream`, определённая выше.

```
(define (sieve s)
```

```
    (cons-stream
        (car s)
        (sieve (sift (car s) (cdr-stream s)))))

(define (sift prime s)
    (filter-stream
        (lambda (x) (not (= 0 (modulo x prime))))
        s))
```

```
(define primes
    (sieve (naturals 2)))

scm> (slice primes 0 10)
(2 3 5 7 11 13 17 19 23 29)
```