Лекция 7. Макросы

Пример макроса. Swap

- чтобы поменять местами значения переменных a и b нужно:

```
(let ((c b))
        (set! b a)
        (set! a c))
```

- чтобы не писать подобный фрагмент каждый раз, определим макрос swap:

```
(define-syntax swap
   (syntax-rules ()
        ((swap a b)
         (let ((c b))
            (set! b a)
            (set! a c))
        )))
```

Пример макроса. Swap

- как работает swap?

```
> (define first 5)
> (define second 'a)
> (swap first second)
> first          ->      'a
> second         ->      5
> c              ->      ошибка!
```

- при подстановке c заменяется на временное имя:

```
> (define c 10)
> (swap first c)
> first          ->      10
> c              ->      'a
```

- работает!

Пример макроса. Swap

- возможный результат подстановки (swap first c):

```
(let ((__generated_symbol_1 c))
  (set! c first)
  (set! first __generated_symbol_1))
```

- описывая макрос, мы указали имя:

```
(define-syntax swap
```

- указали тип применяемого преобразования:

```
(syntax-rules ()
```

- указали правило, содержащее образец:

```
((swap a b)
```

- ... и шаблон:

```
(let ((c b)) (set! b a) (set! a c))
            )))
```

Swap c define

- перепишем без let / проверим, что гигиену дает syntax-rules, а не let:

```
(define-syntax swap2
   (syntax-rules ()
          ((swap2 a b)
           (begin (define c b)
              (set! b a)
              (set! a c)) )))
> (define c 5)
> (define x 1)
> (define y 2)
> (swap2 x y)
> (swap2 x c)
> (display (list x y c))       ->        (5 1 2)
```

Итог примера

- Макрос это:
  - специальным образом описанная трансформация кода
  - трансформация, осуществляемая без вычисления кода
  - трансформация, не использующая данные времени выполнения

- Другой пример cond-set!

```
(cond-set! (> test 4) var 15)
```

- при подстановке даёт:

```
(cond ((> test 4) (set! var 15)))
```

- не вычисляет свои аргументы!

Макроподстановка

Приблизительно подстановка происходит так:

1) Сопоставитель находит вхождение имени макроса в тексте.
2) Сопоставитель перебирает образцы, описанные в макросе, пытаясь сопоставить образец с вызовом макроса.
3) Если удаётся сопоставить с образцом, то в шаблон подставляются параметры макровызова.
4) Локальные имена из шаблона заменяются на сгенерированные символы.
5) Получившийся код вставляется на место макровызова.

Когда уместно использовать макрос

- Используйте макросы для изменения порядка вычислений (определения собственных спецформ).
- Ситуации, когда макросы необходимы:
  1) условные вычисления (аналоги cond, case, and, or)
  2) циклы (аналоги именованного let, do)
  3) связывания (аналоги set!, let, let*)
  4) используются не вычисляемые имена (=>)

Когда неуместно использовать макрос?

- Всегда, когда без него можно обойтись!
  1) Макросы в отличие от функций не являются объектами первого класса.
  2) Макросы затрудняют отладку.

cond-set!

- Вернёмся к примеру:

```
(cond-set! (> test 4) var 15)
```

- Можно ли описать cond-set! функцией?

```
(define (cond-set! test variable value)
        (cond (test (set! variable value))))
```

- не работает!

```
> (define a 5)
> (define b 10)
> (cond-set! (< a b) a b)
> a       ->       5
```

- меняется значение локальной переменной функции!
- Макрос уместен.

swap!

- Можно ли описать swap функцией?

```
(define (swap! x y)
        (let ((c x)) (set! x y) (set! y c)))
```

- не работает!

```
> (define a 5)
> (define b 10)
> (swap! a b)
> a       ->       5
```

- меняется значение локальной переменной функции!
- Макрос уместен.

Другой пример. swap-mpair!

Можно ли описать swap-mpair!, меняющий голову и хвост пары местами, функцией?

```
(define (swap-mpair! mpr)
  (if (or (null? mpr) (not (mpair? mpr)))
      mpr
      (let ((temp (mcar mpr)))
        (set-mcar! mpr (mcdr mpr))
        (set-mcdr! mpr temp))))
> (define a (mcons 1 2))
> (swap-mpair! a)
> a                    ->      {2 . 1}
> (swap-mpair! a)
> a                    ->      {1 . 2}
```

макрос неуместен!

Специальные формы и макросы

- Примитивных (настоящих) спецформ мало:
  - lambda
  - if
  - quote
  - set!

- Все остальные можно описать макросами.
- Хотя в трансляторе они могут быть реализованы напрямую.

Специальные формы и макросы

- and как макрос «на коленке»:

```
(define-syntax and
  (syntax-rules ()
          ((and) #t)
          ((and test) test)
          ((and test1 test2 ...)
                  (if test1 (and test2 ...) #f))))
```

- используется ... , чтобы указать, что аргументов 2≤ (1≤)
- описание рекурсивно
- приёмы составления рекурсивного описания макроса:
  - как и у функции сначала нерекурсивные ветки;
  - макровызов внутри шаблона короче, чем образец в левой части правила, – значит, не будет бесконечной рекурсии

Группировка пар образец-шаблон

- Одно макроопределение может содержать несколько описаний пар образец-шаблон:

```
(define-syntax and
  (syntax-rules ()
          ((and) #t)
          ((and test) test)
          ((and test1 test2 ...)
                  (if test1 (and test2 ...) #f))))
```

При сопоставлении пары просматриваются сверху вниз до первой подходящей! Неверный порядок может привести к ошибке.

Системы макросов в Scheme

- Разные реализации Scheme поддерживают разные системы макросов
  - defmacro (унаследована от Лиспа)
  - syntax-rules (стандартная R5RS, «гигиеничная»)
  - syntax-case
  - syntax-table
  - …
- Рассмотрим syntax-rules (она есть в Racket!)

Характеристики системы syntax-rules

- *Закрытость.* Макросистема отделена от Scheme. Во время подстановки нельзя запустить какой-либо код или использовать какое-то значение.
- *Гигиена*. Макрос не портит непредсказуемым образом окружения, в которых происходит подстановка.
- *Прозрачность ссылок*. Окружение, в котором происходит подстановка не портит макрос.
- *Язык образцов* и *язык шаблонов* используются для описания структуры макрокоманд и «тел» макросов, соответственно.

Гигиеничные макросы

- Все локальные переменные макроса автоматически переименовываются перед подстановкой, для того чтобы предотвратить конфликт имен.
- Пример: swap. Локальное имя c не конфликтует с c из окружения, в котором происходит подстановка.
- Гигиеничность означает, что вызов макроса не может *неожиданно* повлиять на связывания. Ожидаемые изменения возможны:

```
(define-syntax shadow
    (syntax-rules () ((shadow a b)
                      (begin (define a 5) b))))
> (define test 7)
> (shadow test test) --> (begin (define test 5) test)        ->        5
> test    ->        5        ожидаемое изменение с таким вызовом
```

Прозрачность ссылок

- Все свободные переменные из шаблона макроса имеют связывания из окружения, в котором описан макрос.
- Пример:

```
(define-syntax dec
    (syntax-rules () ((dec arg)
                      (set! arg (- arg 1)))))
> (define a 1)
> (define b 1)
> (let ((- +)) (dec a) (set! b (- b 1)))
> a       ->      0       минус «старый»!
> b       ->      2       минус «новый»!
```

Язык образцов

- Язык образцов в syntax-rule прост:
  - образец – это комбинация (список)
  - первый элемент – имя макроса
  - литералы, а также списки, векторы представляют сами себя
  - бывают спецсимволы и многоточия
  - остальное – переменные образца.
- Образец сопоставится:
  - если литералы, списки, векторы совпадают точно
  - если каждая переменная образца сопоставима одному подвыражению макрокоманды.

Язык образцов. Пример

- Дан образец:

(let1 (name value) body)

- Макрокоманда:

```
(let1     (x (read))
          (cond ((not x) (display "you said no"))))
```

- Образец сопоставится:
  - name ≡ x
  - value ≡ (read)
  - body ≡ (cond ((not x) (display "you said no")))

Язык образцов. Ещё пример

- Дан образец:

(contrived #((first . second) #(3 third)))

- Макрокоманда:

(contrived #((1 2 3 4 5) #(3 (foo))))

- Образец сопоставится:
  - first ≡ 1
  - second ≡ (2 3 4 5)
  - third ≡ (foo)

Язык шаблонов

- Шаблон – это код на Scheme, определяющий вместе с образцом, что будет подставлено.
  - литералы, а также списки, векторы представляют сами себя
  - символы, не встречающиеся в образце, представляют сами себя
  - остальное – переменные образца.
- При подстановке происходит замена внутри шаблона переменных образца, на то, с чем они сопоставились.

Язык шаблонов. Пример

- Образец:

```
(let1 (name value) body)
```

- Шаблон:

```
(let ((name value)) body)
```

- Макрокоманда:

```
(let1    (x (read))
         (cond ((not x) (display "you said no"))))
```

- Подстановка:

```
(let     ((x (read)))
         (cond ((not x) (display "you said no"))))
```

Многоточие

- Если за переменной образца следует ... , то переменная с многоточием сопоставляется с несколькими подвыражениями макрокоманды.
- Пример образца:

```
(dotimes count body ...)
```

- Пример макрокоманды:

```
(dotimes 5 (set! x (+ x 1)) (display x))
```

- Сопоставление:

count ≡ 5

body ... ≡ (set! x (+ x 1)) (display x)

Многоточие. Пример

- В шаблоне вхождение переменной образца с многоточием заменяется на все сопоставленные подвыражения
- Пример:

```
(define-syntax dotimes
  (syntax-rules ()
    ((dotimes count body ...)
     (let loop ((counter count))
       (cond ((> counter 0) (begin body ...
         (loop (- counter 1)))))))))
```

```
> (define x 5)
> (dotimes 5 (set! x (+ x 1)) (display x))
```

- в результате код, печатающий 678910

Многоточие. Пример

- … вот этот код:

```
(let loop ((counter 5))
  (cond ((> counter 0)
    (begin (set! x (+ x 1)) (display x)
      (loop (- counter 1))))))
```

Многоточие. Пример

- Многоточие в шаблоне после подвыражения с переменной образца с многоточием вызывает многократную подстановку подвыражения с каждым сопоставлением переменной.
- Пример:

```
(define-syntax thunkify
  (syntax-rules ()
          ((thunkify body ...)
           (list (lambda () body) ...))))
```

- Макрокоманда:

```
(thunkify 5 (* x x))
```

- Постановка

```
(list (lambda () 5) (lambda () (* x x)))
```

Многоточие. Ещё пример

- Многоточие в образце может стоять не после имени, а после подвыражения. Это означает, что в вызове должна быть последовательность из одного или более чем одного подвыражения, сопоставимого с подвыражением из образца.

```
(define-syntax update-if-true!
  (syntax-rules ()
          ((update-if-true! (condition variable) ...)
           (begin (let ((test condition))
                    (cond (test (set! variable test)))) ...))))
```

- Макрокоманда:

```
(update-if-true! ((> x 5) x-is-big) ((zero? y) y-is-zero))
```

Многоточия. Пример

- Постановка:

```
(begin
    (let ((test (> x 5)))
     (cond (test (set! x-is-big test))))
    (let ((test (zero? y)))
     (cond (test (set! y-is-zero test)))))
```

Многоточия. Пример

- В образце подвыражение перед многоточием также может внутри себя содержать многоточия. Это означает, что частей в подвыражении может быть больше.
- Пример:

```
(define-syntax quoted-append
  (syntax-rules ()
          ((quoted-append (arg ...) ...)
           (quote (arg ... ...)))))
```

- Макрокоманда:

```
(quoted-append (1 2 3) (a b c) (+ x y))
```

- Постановка

```
(1 2 3 a b c + x y)
```

Спецсимволы в образцах

- Пусть мы хотим, чтобы макрокоманда:

(implications (a => b) (c => d) (e => f))

- давала подстановку:

(begin (cond (a b) (else #t)) (cond (c d) (else #t)) (cond (e f) (else #t)))

- проблема в том, как указать, что => – не переменная!

(syntax-rules ()
((implications (condition => consequent) ...)
(begin (cond (condition consequent) (else #t)) ...)))

- приводит к тому, что при макровызове

(implications (test =. (set! testp #t)))

- получается ненужное нам сопоставление

condition ≡ test

=> ≡ =.

consequent ≡ (set! testp #t)

Спецсимволы в образцах

- Спецсимволы следует указывать в списке после syntax-rules:

```
(syntax-rules (=>)
((implications (condition => consequent) ...)
(begin (cond (condition consequent) (else #t)) ...)))
```

- Теперь при макровызове (implications (a => b) (c => d) (e => f)) получим только нужные сопоставления condition ≡ a, consequent ≡ b и т. д..
- При макровызове (implications (test =. (set! testp #t))) не будет сопоставления, так как *спецсимвол сопоставляется только сам с собой.*
- другой случай, когда сопоставление невозможно:

```
(let ((=> 5)) (implications (foo => bar)))
```

локальное имя => из let и => из окружения, где описан макрос, разные!

Итоги по спецсимволам в образцах

- Спецсимволы из макроопределения относятся к окружению, где было сделано макроопределение.
- Символы из макрокоманды относятся к окружению, в котором встретилась команда.
- Спецсимвол сопоставится, только если он не перекрыт в окружении макрокоманды.
- Многоточие ведёт себя не как спецсимвол, значит, оно не спецсимвол.

Итоги по syntax-rules

Сочетание гигиеничности с прозрачностью ссылок делает макросы системы syntax-rules лексически безопасными.

- Макрос не портит неожиданно окружение, в котором происходит подстановка.
- Окружение, в котором происходит подстановка, не портит макрос.
- Действует «правило наименьшего удивления».

Безопасность дополнительно повышается за счёт закрытости системы syntax-rules.

Вспомним dotimes

```
(define-syntax dotimes
  (syntax-rules ()
    ((dotimes count body ...)
     (let loop ((counter count))
       (cond ((> counter 0) (begin body ...
         (loop (- counter 1)))))))))
```

- что если?

```
> (define counter 5)
> (dotimes 5 (set! counter (+ counter 1)) (display counter))
```

- зацикливания нет, в результате код, печатающий 678910
- макрос безопасный – counter’ы разные.

Рекомендации по стилю макроопределений

- сортируйте пары образец-шаблон (сначала короткие, как в and)
- имя макроса в образце можно не писать, а ставить прочерк (так проще переименовывать макросы)

```
(define-syntax dec
    (syntax-rules () (( _ arg)
                          (set! arg (- arg 1)))))
```

- если можно выбрать между let и define внутри шаблона макроопределения, то выбирайте let
- при написании рекурсивных макроопределений действуйте по рекомендациям со слайда №13.

Итоги лекции

- Мы рассмотрели не всё. Бывают:
  - cps-style макросы
  - обход гигиеничности (Petrofsky’s find-identifier)
  - синтезирование символов
- Того, что рассмотрено, достаточно.
- На итоговой контрольной будут задачи/вопросы по этой теме.