Символьные вычисления и макросы

Коновалов А. В.

10 октября 2022 г.

Символьные вычисления

Символьные вычисления

Символьные вычисления — это преобразования символьных данных: алгебраические выкладки, трансляция языков программирования и т.д.

ЛИСП был одним из первых языков, ориентированных на символьные вычисления.

Средства языка Scheme для символьных вычислений

Как мы помним, один из постулатов языков семейства ЛИСП — единство кода и данных. Т.е. основным типом данных в ЛИСП является список, и сама программа представлена в виде вложенных списков.

Это не случайно. Программы на ЛИСПе могут преобразовывать программы на ЛИСПЕ как данные, т.е. ЛИСП естественным образом реализует парадигму метавычислений.

Символьный тип данных (1)

Символьный тип данных — это «зацитированное», «замороженное» имя:

```
'hello
```

Имеет известный нам предикат типа и процедуры преобразования:

```
(symbol? 'hello)                    → #t
(symbol? "hello")                   → #f
(symbol? #\a)                       → #f
(symbol→string 'hello)              → "hello"
(string→symbol "hello")             → hello
(string→symbol
  (string-append
    (symbol→string 'hell)
    (symbol→string 'o))))           → hello
```

Символьный тип данных (2)

Нельзя сказать, что у символов есть свои литералы. Но символы создаются операцией цитирования:

```
'hello                          → hello
(quote hello)                   → hello
'(hello world)                  → (hello world)
(quote (hello world))           → (hello world)
```

Операция цитирования — это особая форма (`qoute …`), которая принимает терм и «цитирует» его — все идентификаторы в нём становятся символами, остальные атомарные значения остаются как есть, выражения превращаются в списки.

Операция цитирования (1)

У операции цитирования (`qoute X`) имеется синтаксический сахар `'X`, что демонстрирует пример выше. Ещё пример:

```
(car ''x)                                   → ‹что тут будет?›
```

Операция цитирования (1)

У операции цитирования `(qoute X)` имеется синтаксический сахар `'X`, что демонстрирует пример выше. Ещё пример:

```
(car ''x)                                  → ‹что тут будет?›
```

Раскроем сахар:

```
(car (quote (quote x)))                    → quote
(equal? (car (qoute (quote x)))
        'qoute)                            → #t
(equal? (car (quote (quote x)))
        (string→symbol "quote"))           → #t
```

Это не ошибка — запись ключевого слова в цитате. Т.е. `'if` — корректная запись. Получится символ `if`.

Операция цитирования (2)

Литерал вектора подразумевает неявное цитирование:

```
(vector-ref #(a b c) 1)                    → b
```

Получим символ b.

```
#(10 (+ 5 5) 100)                          → #(10 (+ 5 5) 100)
```

В первом элементе зацитирован список (+ 5 5).

Квазицитирование (1)

Квазицитирование позволяет внутри цитаты вычислять какие-то значения.

Обозначается она обратной кавычкой (на клавиатуре на букве ё). Для «расцитирования» внутри квазицитаты перед термом записывается знак «запятая».

```
`(10 20 ,(+ 15 15) 40)                  → (10 20 30 40)
        ^^^^^^^^^^ – эта часть будет вычислена
```

Квазицитирование (2)

В квазицитату можно вставлять списки двумя способами. Просто запятая перед переменной вставит список как значение. Знак ,@ (запятая-собачка) вклеит список:

```
(define xs '(a b c d))

`(1 2 ,xs 3 4)          → (1 2 (a b c d) 3 4)
`(1 2 ,@xs 3 4)         → (1 2 a b c d 3 4)
```

Т.е. если внутри списка имеем ,@, то списки будут сконкатенированы.

Конкатенацию двух списков можно написать так:

```
(define (my-append xs ys)
  `(,@xs ,@ys))
```

Квазицитирование (3)

Квазицитирование тоже является синтаксическим сахаром для особых форм:

```
`expr           ≡ (quasiquote expr)
,expr           ≡ (unquote expr)
,@expr          ≡ (unquote-splicing expr)
```

Квазицитирование как интерполяция переменных

Во многих современных языках программирования присутствует похожий на квазицитирование механизм: интерполяция переменных внутри строковых констант. Например, в Python

```
>>> x = 100
>>> y = f"Square of {x} is {x*x}"
>>> y
"Square of 100 is 10000"
```

Квазицитирование: программа → данные

При помощи цитирования мы можем делать из программы данные.

```
;; Это выражение:
(lambda (x) (* x x))

;; А это цитата, список из трёх элементов, причём два последн
;; тоже списки
'(lambda (x) (* x x))
```

Процедура `eval`: данные → программа (1)

Но возможен ли обратный механизм? Можно ли выражение, записанное в цитаты, выполнить как выражение?

Процедура `eval`: данные → программа (1)

Но возможен ли обратный механизм? Можно ли выражение, записанное в цитаты, выполнить как выражение?

Можно. Для этого используется встроенная процедура `eval`. Её синтаксис:

```
(eval ‹цитата› ‹окружение›)          → ‹результат вычисления
                                        ‹цитаты› в ‹окружении››
```

Здесь `‹цитата›` — некоторое зацитированное выражение, `‹окружение›` — множество значений переменных, которые могут использоваться в `‹цитате›`.

Процедура `eval`: данные → программа (2)

Окружение вручную создать нельзя, его можно запросить другими встроенными процедура:

```
(scheme-report-environment 5)       ; встроенные процедуры
                                    ; и макросы среды R5RS

(null-environment 5)                ; только встроенные
                                    ; макросы R5RS

(interaction-environment)           ; текущие глобальные
                                    ; переменные среды
```

Последнее наиболее употребительное.

Процедура `eval`: данные → программа (3)

Заметим, что `eval` может менять среду, в частности добавлять новые переменные:

```
(eval (list '+ 5 7)
      (interaction-environment))   → 12

(define x 100)
(define y 500)
(eval (list '* 'x 'y)
      (interaction-environment))   → 5000000
```

Процедура `eval`: данные → программа (4)

```
(define ie (interaction-environment))

z                                          → ОШИБКА «Нет переменной z
```

```
(eval '(define z 100500) ie)
z                                          → 100500
```

Процедура `eval`: данные → программа (5)

`interaction-environment` хранит только глобальные переменные. Локальные в нём не видны:

```
(define a 100)

(let ((a 200))
  (eval '(* a a)
        (interaction-environment))) → 10000
```

Процедуры `member` и `assoc`, ассоциативные списки

Процедура `member`

Процедура `member` ищет элемент в списке. Если найден — возвращает хвост списка, начиная с этого элемента. Если нет — `#f`.

```
(member 'b '(a b c d))          → (b c d)
(member 'c '(a b c d))          → (c d)
(member 'z '(a b c d))          → #f
```

Ассоциативный список

Ассоциативный список — это способ реализации ассоциативного массива (т.е. структуры данных, отображающей ключи на значения) при помощи списка, это список пар (cons-ячеек), где в `car` находится ключ, а в `cdr` — связанное значение. Частный случай — список списков, где `car`'ы — ключи, а хвосты — значения. Чаще всего это частный случай и встречается, т.к. правильные списки просто удобнее.

Пример:

```
'((a 1) (b 2) (c 3))
```

отображает имена на некоторые числа.

Процедура assoc

Процедура assoc принимает ключ и ассоциативный список
и возвращает первый элемент с заданным ключом:

```
(assoc 'b '((a 1) (b 2) (c 3)))    → (b 2)
(assoc 'x '((a 1) (b 2) (c 3)))    → #f
```

Синтаксис cond со стрелкой используется с assoc:

```
(cond ((assoc key table) → (lambda (val) (cadr val))
      (else 'not-found))
```

`member`, `assoc` × `equal?`, `eqv?`, `eq?`

У `member` и `assoc` есть «процедуры-сёстры», которые отличаются предикатом сравнения:

Поиск	Ассоц. список	Предикат
`member`	`assoc`	`equal?`
`memv`	`assv`	`eqv?`
`memq`	`assq`	`eq?`

Макросы

Лирическое отступление (1)

Лирическое отступление. На взгляд лектора, можно выделить три уровня познания языков программирования:

1. Ученик почти наугад подбирает последовательность инструкций, дающую вроде как верный результат. Ну или не почти наугад.
2. Программист выражает свои мысли на языке программирования.
3. Программист выбирает или даже сам создаёт язык программирования, наиболее подходящий для выражения решения задачи.

Лирическое отступление (2)

Язык предметной области (domain specific language, DSL) — это некоторый ограниченный по средствам язык, предназначенный для решения конкретной задачи. Это язык, на котором решение данной задачи лучше всего выражается. Язык предметной области определяется или как интерпретатор (внешний DSL), либо как библиотека для некоторого имеющегося языка (внутренний DSL).

В Scheme для определения DSL'ей часто используются макросы.

Макросы (1)

Макрос — это инструмент переписывания кода. Т.е. способ создать новую языковую конструкцию на основании имеющихся.

Процесс выполнения выражений на Scheme. Пусть у нас есть выражение вида

`(‹имя› ‹термы…›)`

1. Если `‹имя›` — ключевое слово языка (`if`, `define`, `quote`, `lambda`, и т.д.), то выражение интерпретируется как особая форма.
2. Если `‹имя›` — имя макроса, то данное выражение перезаписывается согласно определению макроса.
3. Если `‹имя›` — имя переменной, то в переменной должна быть процедура, эта процедура вызывается.

Макросы (2)

Т.е. можно считать, что вычисление выражения состоит из двух этапов:

1. Раскрытие макросов.
2. Собственно вычисления (выполнения особых форм, вызовы процедур).

Макросы (3)

Синтаксис определения макроса:

```
(define-syntax ‹имя›
  (syntax-rules (‹ключевые слова›)
    (‹образец› ‹шаблон›)
    (‹образец› ‹шаблон›)
    (‹pattern› ‹template›)))
```

‹образец› (‹pattern›) — вид, который должно иметь обращение к макросу. ‹шаблон› (‹template›) — то, на что макрос заменяется.

Образцы проверяются сверху вниз и выбирается тот, который первым подходит.

Макросы (4)

Что значит: *образец подходит к обращению к макросу (применению макроса)?*

Макросы (4)

Что значит: *образец подходит к обращению к макросу (применению макроса)?*

В правилах макроса могут быть переменные (правильнее сказать, **метапеременные**), которым соответствуют фрагменты кода на Scheme. Если в ‹образце› мы можем вместо вхождений переменных подставить фрагменты кода на Scheme таким образом, что получим запись применения макроса, то считаем, что применение макроса с образцом сопоставилось успешно, и правило применяется.

Макросы: примеры (1)

Рассмотрим примеры некоторых макросов. Макрос, имитирующий встроенный макрос `begin`:

```
(define-syntax my-begin
  (syntax-rules ()
    ((my-begin one-action) one-action)
    ((my-begin action . other-actions)
     (let ((x action))
       (my-begin . other-actions)))))
```

Как это определение читается?

Макросы: примеры (2)

Если обращение к макросу имеет вид `(my-begin ‹одно какое-то действие›)`, то это дейтвие результатом раскрытия макроса (первое правило).

```
(my-begin (display 'hello))          ≡ (display 'hello)
```

В этом примере вместо метапеременной `my-begin` может быть подставлено имя макроса `my-begin`, вместо метапеременной `one-action` может быть подставлено подвыражение `(display 'hello)`, поэтому первое правило применимо. Первое правило выполнится, макрос заменится на шаблон правила, состоящий из одной переменной `one-action`, вместо неё будет подставлено `(display 'hello)`.

Макросы: примеры (3)

Второе правило применимо, когда не применимо первое правило и обращение к макросу может быть сопоставлено с образцом (`my-begin action . other-actions`). Образец состоит из трёх переменных, последняя в позиции после точки, т.е. она будет сопоставлена с концом списка. Образец говорит о том, что обращение к макросу должно быть списком из минимум двух элементов: первый будет отображён на переменную `my-begin`, второй — на `action`, все остальные — на `other-actions`.

Поскольку случай ровно двух элементов в списке перехватывается предшествующим правилом, в `other-actions` у нас всегда будет непустой список.

Макросы: примеры (4)

По смыслу это означает, что второе правило описывает конструкцию `my-begin`, в которой не менее двух выражений: имя макроса `my-begin` сопоставится с первой переменной, первое выражение сопоставится с `action`, последующие как список — с `other-actions`.

Макросы: примеры (5)

Рассмотрим правую часть (шаблон, template) второго правила:

```
((my-begin action . other-actions)
 (let ((x action))
   (my-begin . other-actions)))
```

По шаблону будет построено let-выражение с одной переменной, с переменной свяжется значение выражения, попавшего в `action`, внутри let'а будет рекурсивно применён макрос `my-begin` со всеми остальными выражениями. Чтобы из списка `other-actions` получить список, где первым элементом будет `begin`, а хвостом — `other-actions`, мы строим cons-пару при помощи точечной нотации (в макросах точечная нотация используется вместо `cons`).

Макросы: примеры (6)

Если аргументом макроса будет несколько каких-то выражений (т.е. список выражений), то строится let-выражение, результат первого действия связывается с переменной и тем самым вычисляется до всех остальных. Остальные (их может быть несколько) заворачиваются в `my-begin`, который обеспечит их последовательное выполнение (т.е. макрос вызывается рекурсивно).

Рассмотрим последовательные раскрытия макроса `my-begin` на примере:

```
(my-begin
  (display 'hello)
  (display 'my)
  (display 'world))
```

```
(let ((x (display 'hello)))
   (my-begin
     (display 'my)
     (display 'world)))
```

Макросы: примеры (7)

В первом применении макроса первое правило не применимо, т.к. невозможно отобразить список из 4 элементов на список из двух элементов (`my-begin one-action`). Второе правило применимо: можно отобразить список из четырёх элементов на (`my-begin action . other-actions`), получим следующие подстановки для переменных:

- `my-begin` ← `my-begin`,
- `(display 'hello)` ← `action`,
- `((display 'my) (display 'world))` ← `other-actions`.

Макросы: примеры (8)

Подстановка их в правую часть

```
(let ((x action))
  (my-begin . other-actions))
```

даст

```
(let ((x (display 'hello)))
   (my-begin
     (display 'my)
     (display 'world)))
```

Макросы: примеры (9)

Полное раскрытие приведёт к выражению:

```
(let ((x (display 'hello)))
   (let ((x1 (display 'my)))
      (my-begin (display 'world))))
```

↓ ↓ ↓

```
(let ((x (display 'hello)))
  (let ((x1 (display 'my)))
    (display 'world)))
```

Макросы: примеры (10)

Подробнее о следующих шагах раскрытия.

Второй шаг раскрытия тоже задействует второе правило (т.к. список из трёх элементов, переменные будут следующие:

- `my-begin` ← `my-begin`,
- `(display 'my)` ← `action`,
- `((display 'world))` ← `other-actions`.

Их подстановка даст

```
(let ((x1 (display 'my)))
   (my-begin (display 'world))))
```

Макросы: примеры (11)

Третье раскрытие будет по первому правилу, т.к. список из двух элементов можно отобразить на список двух переменных `(my-begin one-action)`:

- `my-begin` ← `my-begin`,
- `(display 'world)` ← `one-action`.

Подстановка в правую часть

```
((my-begin one-action) one-action)
```

даст одно `one-action`, т.е. `(display 'world)`.

Гигиенические макросы (1)

Макросы в Scheme гигиенические, т.е. о конфликте имён при их раскрытии беспокоиться не нужно. В примере выше для различных раскрытий макроса сгенерированы разные имена `let`'ов: `x` и `x1`.

Ключевые слова в макросе не являются метапеременными и трактуются буквально.

Макросы: примеры (12)

Пример. Определим макрос `my-cond`, частично имитирующий встроенный макрос `cond`:

```
(define-syntax my-cond
  (syntax-rules (else)
    ;; последняя ветка else
    ((my-cond (else . actions)) (begin . actions))

    ;; последняя ветка не else
    ((my-cond (condition . actions))
     (if condition
         (begin . actions)
         #f))  ;; когда нам нечего вернуть, возвращаем #f

    …))
```

Макросы: примеры (13)

```
(define-syntax my-cond
  (syntax-rules (else)
    …
    ;; не последняя ветка
    ((my-cond (condition . actions) . branches)
     (if condition
         (begin . actions)
         (my-cond . branches)))))
```

Макросы: примеры (14)

Исходный код

```
(my-cond ((> x 0) (display 'pos) (newline))
         ((< x 0) (display 'neg) (newline)))
```

Здесь сработает третье правило:

- `condition` → `(> x 0)`
- `actions` → `((display 'pos) (newline))`
- `branches` → `(((< x 0) (display 'neg) (newline)))`

Макросы: примеры (15)

Код перепишется в

```
(if (> x 0)
    (begin (display 'pos) (newline))
    (my-cond ((< x 0) (display 'neg) (newline))))
```

На рекурсивном обращении к макросу сработает вторая ветка, в результате макрос раскроется в

```
(if (> x 0)
    (begin (display 'pos) (newline))
    (if (< x 0)
        (begin (display 'neg) (newline))
        #f))
```

Макросы: примеры (16)

Пример. Цикл со счётчиком.

```
(define-syntax for
  (syntax-rules (:= to downto do)
    ((for var := start to end do . actions)
     (let ((limit end))
       (let loop ((var start))
         (and (⩽ var limit)
              (begin
                (begin . actions)
                (loop (+ var 1)))))))

    …))
```

Макросы: примеры (17)

```
(define-syntax for
  (syntax-rules (:= to downto do)
    …

    ((for var := start downto end do . actions)
     (let ((limit end))
       (let loop ((var start))
         (and (⩾ var limit)
              (begin
                (begin . actions)
                (loop (- var 1))))))))))

(for x := 1 to 10 do
  (display x)
  (newline))
```

Гигиенические макросы (2)

Макросы в Scheme **гигиенические.** Это означает, что для каждого раскрытия макроса имена переменных в `let`, `letrec`, `let*`, параметрах `lambda` и `define` генерируются новые. А значит, конфликт имён исключён.

Макросы (5)

В образцах макросов можно использовать вместо имени переменной знак _, означающий безымянную переменную. Он используется, когда конкретное значение не нужно (игнорируется). Чаще всего он используется для имени самого макроса:

```
(define-syntax my-begin
  (syntax-rules ()
    ((_ one-action) one-action)
    ((_ action . other-actions)
     (let ((x action))
       (my-begin . other-actions)))))
```

В макросах можно использовать т.н. «эллипсис», т.е. `...` . Эллипсис в макросах оставляется вам на самостоятельное изучение.

Разработка через тестирование

Разработка через тестирование (1)

Разработка через тестирование — способ разработки программы, предполагающий написание **модульных тестов** (unit tests) до написания кода, который они проверяют.

Модульный тест — автоматизированный тест, проверяющий корректность работы небольшого фрагмента программы (процедуры, функции, класса и т.д.). Модульный тест обязательно должен быть самопроверяющимся, т.е. без контроля пользователя запускает тестируемую часть программы и проверяет, что результат соответствует ожидаемому.

Разработка через тестирование (2)

Цикл разработки через тестирование:

1. Пишем тест для нереализованной функциональности. Этот тест при запуске *проходить не должен.*
2. Пишем функциональность, *но ровно на столько,* чтобы новый тест проходил. При этом все остальные тесты тоже должны проходить (не сломаться).
3. **Рефакторинг** — это эквивалентное преобразование программы, направленное на улучшение её внутренней структуры (повышение ясности программы, её расширяемости, эффективности). *В процессе рефакторинга ни один из модульных тестов сломаться не должен.*

Продолжительность одного цикла — около минуты.

Дополнительные материалы

Дополнительные материалы (1)

Ещё к лабораторной работе

```
(load ‹имя файла›)
```

Эта процедура читает и выполняет указанный файл. Её можно считать примерным аналогом `#include` в языке Си.

```
(write expr)
(display expr)
(newline)
```

Дополнительные материалы (2)

Процедура `write` печатает машиночитаемом формате, т.е., например, строки выводит в кавычках и с escape-последовательностями. Процедура `display` — в человекочитаемом, т.е. символы строк выводит буквально. `newline` печатает перевод на новую строчку.

`(display "1234")` и `(display 1234)` выведут идентичный текст.