

Э. В. Харитонов

РЕАЛИЗАЦИЯ СОПОСТАВЛЕНИЯ С ОБРАЗЦОМ В ЯЗЫКЕ LISP НА ОСНОВЕ АНАЛОГИЧНЫХ СРЕДСТВ В ЯЗЫКАХ REFAL И HASKELL¹

ВВЕДЕНИЕ

В языках Refal [1], ML, Haskell [2] разбор структурированных данных удачно поддержан механизмом сопоставления с образцом. Lisp [3, 4] — язык, хорошо приспособленный к обработке структурных данных, но не содержащий в наборе стандартных возможностей средств сопоставления с образцом. Тем не менее базовые средства языка Lisp вполне достаточны для реализации подобных возможностей. Более того, было бы естественно расширить Lisp несколькими модификациями новых средств одновременно и оценить разные варианты расширения языка в общей программной среде.

В данной работе рассматриваются некоторые способы введения в язык Lisp средств сопоставления с образцом и оцениваются удобства применения этих средств для обработки структурированных данных.

Lisp, расширенный средствами сопоставления с образцом, обозначим Lisp*.

1. ФОРМА FUNC — ОПРЕДЕЛЕНИЕ ФУНКЦИИ ПРИ ПОМОЩИ СОПОСТАВЛЕНИЯ С ОБРАЗЦАМИ

При традиционном программировании на языке Lisp, с разбором вариантов и рекурсией, функции, обрабатывающие структурные выражения (S-выражения), выделяют элементы структуры параметров и в зависимости от их значений возвращают разные результаты. Например, функция вычисления длины списка имеет следующий вид:

```
(defun length1 (L)
  (cond
    ((null L) 0)
    (t (+ 1 (length1 (cdr L))))
  ))
```

¹Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 01-01-794) и Министерства образования РФ.

Аналогичное определение на языке Haskell выглядит так:

```
length2 []          -> 0
length2 (x:xs)     -> 1 + length2 xs
```

Здесь первая строка — длина пустого списка, вторая — длина списка с головой `x` и хвостом `xs`.

Для языка Lisp можно было бы определить вместо `defun` новую форму — `func`, позволяющую оформить определение функции `length` в стиле приведенного Haskell-фрагмента. Определение на языке Lisp с использованием сопоставления с образцом может выглядеть, например, так:

```
(func length3
  (nil)          0
  ((x . xs))    (+ 1 (length3 xs))
)
```

Это означает, что если список параметров функции `length3` совпадает с `(nil)`, то результат равен 0 (длина пустого списка). Если же параметр `length3` возможно представить в виде `(cons x xs)`, то значение вычисляется рекурсивным вызовом `(+ 1 (length3 xs))`.

Пусть форма `func` определяет новую функцию и имеет вид:

```
(func ИМЯ_ФУНКЦИИ
  ОБРАЗЕЦ1      ВЫРАЖЕНИЕ1
  ...
  ОБРАЗЕЦN      ВЫРАЖЕНИЕN
)
```

Здесь `ИМЯ_ФУНКЦИИ` — символьный атом, пополняющий глобальную среду имен, `ОБРАЗЕЦi` — S-выражение, задающее вариант структуры списка параметров, `ВЫРАЖЕНИЕi` — выражение, вычисляющее результат функции при успешном сопоставлении списка параметров с `i`-м образцом. При этом если в образце содержатся переменные, то при успешном сопоставлении эти переменные получают значения соответствующих фрагментов списка параметров. Например: при вызове `(length3 '(1 2 3))` список параметров, очевидно, `((1 2 3))`, и сопоставление с образцом `(nil)` неуспешно, поскольку `(1 2 3)` и `nil` — разные значения, а сопоставление с образцом `((x . xs))` успешно при `x = 1`, `xs = (2 3)`, поэтому будет возвращен результат `(+ 1 (length3 '(2 3)))`, из которого в конце концов получится число 3.

Заметим, что переменная `x` не используется в выражении `(+ 1 (length3 xs))`. Если переменная, входящая в образец, не используется

в выражении, ее предпочтительнее обозначать подчеркиком:

```
(func length4
  (nil)          0
  ((_ . xs))    (+ 1 (length4 xs))
)
```

Рассмотрим несколько примеров применения введенной формы `func`. Тестовые вызовы определяемых ниже функций будут обозначаться **ВЫРАЖЕНИЕ => ЗНАЧЕНИЕ**, например, “`(length nil) => 0`”.

Пример. Функция, возвращающая подсписок из первых N элементов данного списка L .

Lisp:

```
(defun take1 (L N)
  (cond
    ((null L) nil)
    ((= N 0) nil)
    (t (cons (car L) (take1 (cdr L) (- N 1))))
  ))
```

Haskell:

```
take2 []      _   -> []
take2 _      0   -> []
take2 (x:xs) N -> x : take2 xs (N-1)
```

Lisp*:

```
(func take3
  (nil      _)  nil
  (_       0)  nil
  ((x . xs) N) (cons x (take3 xs (- N 1)))
)
```

```
(take3 '(A B C (1 2) 3 4) 4) => (A B C (1 2))
```

Пример. Функционал, выполняющий данную функцию F над каждым элементом списка L и выдающий список из результатов

Lisp:

```
(defun map1 (F L)
  (cond
    ((null L) nil)
    (t (cons (funcall F (car L)) (map1 F (cdr L))))
  ))
```

Haskell:

```
map2 _ []      -> []
map2 f (x:xs) -> (f x) : map2 f xs
```

Lisp*:

```
(func map3
  (_ nil)      nil
  (F (x . xs)) (cons (funcall F x) (map3 F xs))
)
```

```
(map3 'sqrt '(1 4 9 16)) => (1 2 3 4)
```

2. ПОИСК ОБЩИХ ПОДВЫРАЖЕНИЙ В ОБРАЗЦАХ

В языке Refal, в отличие от Haskell'а, в образце разрешается несколько вхождений одной переменной, что означает требование совпадения данных, соответствующих разным вхождениям переменной при сопоставлении с таким образцом.

Пример. Поиск в ассоциативном списке ((E1 . V1) ... (En . Vn)).

Lisp:

```
(defun sass1 (E L)
  (cond
    ((null L) nil)
    ((eq E (caar L)) (cdr L))
    (t (sass1 E (cdr L))))
)
```

Lisp*:

```
(func sass2
  (_ nil)      nil
  (E ((E . V) . _))  V
  (E (_ . L))      (sass2 E L)
)
```

```
(sass2 y '((x . 1) (y . -3) (z . A))) => (y . -3)
```

Пример. Свойство из списка (E1 V1 ... EN VN).

Lisp:

```
(defun get1 (E L)
  (cond
    ((null L) nil)
    ((eq E (car L)) (cadr L))
    (t (get1 E (cddr L))))
  ))
```

Lisp*:

```
(func get2
  (_ nil)          nil
  (E (E V . _))   V
  (E (_ _ . L))   (get2 E L)
)
```

```
(get2 'name '(apval 5 name A)) => A
```

3. ФУНКЦИИ С ПЕРЕМЕННЫМ КОЛИЧЕСТВОМ ПАРАМЕТРОВ

Количество параметров функции может быть переменным, поскольку в образцах фигурирует весь список параметров.

Пример. Вычисление суммы квадратов от переменного количества аргументов.

Lisp:

```
(defun sumsqr1 (&rest Args)
  (cond
    ((null Args) 0)
    (t (+ (expt (car Args) 2) (apply 'sumsqr1 (cdr Args)))))
  ))
```

Lisp*:

```
(func sumsqr2
  nil      0
  (a . b) (+ (* a a) (apply 'sumsqr2 b))
)
```

Здесь строка “nil 0” означает, что если функцию `sumsqr2` вызвали без параметров, то она возвращает 0. Если же первый параметр связать

с переменной `a`, а список всех остальных параметров — с переменной `b`, то результат — сумма от `(* a a)` и вызова `sumsq2` со списком параметров из `b`.

```
(sumsq2 1 -2 3) => 14
```

Пример. Объединение переменного количества списков.

Lisp:

```
(defun append1 (&rest l)
  (cond
    ((null l) nil)
    ((null (cdr l)) (car l))
    ((null (car l)) (apply 'append1 (cdr l)))
    (t (cons (caar l) (apply 'append1 (cons (cdar l) (cdr l))))))
)
```

Lisp*:

```
(func append2
  nil          nil
  (a)         a
  (nil . a)   (apply 'append2 a)
  ((a . b) . c) (cons a (apply 'append2 (cons b c)))
)
```

```
(append2 '(1 2) '(a b) '((x)) => (1 2 a b (x))
```

Из приведенных примеров видно, что форма `func` синтаксически и семантически сходна с комбинацией форм `defun` и `cond`. Применение формы `func` наиболее удачно в примерах, описывающих функции `sassoc2`, `get2` и `append2`. Это связано с большей сложностью структуры образцов в указанных примерах, и, следовательно, с более заметным переносом на образцы действий по разбору структурных выражений. Таким образом, форма `func`, реализующая сопоставление с образцами, наиболее адекватно применима при создании функций обработки сложных структур.

4. ФУНКЦИОНАЛЬНЫЕ ВЫРАЖЕНИЯ С ОБРАЗЦАМИ

Форма `func` пополняет глобальную среду новым именем функции. Для локального определения функций нужен аналог лямбда-выражения. Форма “ \rightarrow ” определяет новую функцию без имени и возвращает ее в качестве значения:

```
(mapcar
  (-> (x) (+ 1 x))
  '(1 2 3 4)
)
; Результат: (2 3 4 5)
```

```
(let*
  ((length5
    (->
      (nil)      nil
      ((x . xs)) (+ 1 (funcall length5 xs))
    )))
  ; Функция ‘длина списка’ связана с временной
  ; переменной length5
  (mapcar length5 '(nil (q) (3 4 5)))
)
; Результат: (0 1 3), действие локального значения length5
; прекратилось
```

5. МОДИФИКАЦИИ СИНТАКСИСА

Формы `func` и “ \rightarrow ” должны быть более подвержены ошибкам при программировании, чем, например, форма `cond`, потому что ветви “образец выражение” не обособлены дополнительными скобками. Отчасти это сделано для того, чтобы не перегружать функциональные определения дополнительными скобками. Поскольку в языке Lisp есть возможность вводить синтаксические расширения, можно было бы определить более наглядный вид конструкций сопоставления с образцами, например:

```
(func length6
  {nil      -> nil}
  {(x . xs) -> (+ 1 (funcall length6 xs))}
)
```

```
(func append3
  {      -> nil}
  {a     -> a}
  {nil . a -> (apply 'append3 a)}
  {(a . b) . c -> (cons a (apply 'append3 (cons b c)))}
)
```

6. НЕКОТОРОЕ ОБОБЩЕНИЕ ПОНЯТИЯ СТРУКТУРЫ И ОБРАЗЦА

Фактически образец содержит утверждение, при помощи каких средств сконструированы сопоставляемые данные, а для сопоставления с образцом требуется информация о том, как “разобрать” данные.

Функции для конструирования данных назовем конструкторами, а функции для доступа к частям, из которых данные сконструированы, — реструктурами. Реструктуры являются обратными функциями по отношению к конструкторам.

Другой подход мог бы состоять в том, что реструктор (пусть он в этом случае называется полным реструктором) — функция, возвращающая весь список параметров конструктора.

Если F — конструктор, и $Y = F(X_1, \dots, X_n)$ — сконструированные данные, то обозначим реструкторы как $F'_1(Y), \dots, F'_n(Y)$. Полный реструктор будет иметь вид $(F'_1(Y) \dots F'_n(Y))$. В случае неоднозначности конструктора неоднозначными могут быть и $F'_i(Y)$. Тогда может потребоваться вычисление всех возможных вариантов значений. Например, если в качестве конструктора используется операция умножения целых чисел, для разбора полученной “структуры” понадобится нахождение делителей числа. Для однозначности будем предполагать, что образец $(* x y)$ обозначает: x — наименьший простой делитель в сопоставляемом этому образцу числе.

; Список простых делителей числа

```
(func primes-list
  { 1          -> nil }
  { (* x y)    -> (cons x (primes-list y)) }
)
```

```
(func length6
  { nil        -> 0 }
  { (cons _ xs) -> (+ 1 (length6 xs)) }
)
```

Итак, чтобы функцию можно было применять в качестве конструктора, необходимо задать для нее полный реструктор.

Реструктор для функции CONS:

```
(lambda (x) (list (car x) (cdr x)))
```


Реструктор для функции LIST:

```
(lambda (x) x)
```

Реструктор для функции VECTOR:

```
(lambda (v) (coerce v 'list))
```

Реструктор для функции “*” (вариант):

```
; (lsd x) --- наименьший простой делитель от x  
(lambda (x) (let ((y (lsd x))) (list y (/ x y))))
```

Поскольку имя функции-конструктора — символьный атом, то реструктор может быть задан, например, как свойство `RESTR` для символьного атома — имени конструктора:

```
(setf (get 'cons 'restr)  
      (lambda (x) (list (car x) (cdr x))))  
)  
(setf (get 'list 'restr)  
      (lambda (x) x))  
)  
(setf (get 'vector 'restr)  
      (lambda (v) (coerce v 'list)))  
)  
(setf (get '* 'restr)  
      (lambda (x) (let ((y (lsd x))) (list y (/ x y))))  
)
```

Рассмотренный материал позволяет заключить, что расширение языка Lisp средствами сопоставления с образцом может способствовать повышению компактности программ, а также некоторому упрощению программирования.

СПИСОК ЛИТЕРАТУРЫ

1. **Turchin V.F.** Refal-5, Programming Guide and Reference Manual — Holyoke: New England Publishing Co., 1989.
2. **The Haskell 98 Report.** — 1999. — <http://haskell.org>
3. **McCarthy J.** Lisp 1.5 programming manual. — Cambridge: The MIT Press., 1963.
4. **Henderson P.** Functional Programming Application and Implementation. — London: Prentice-Hall International Inc., 1980.