УДК 004.42

МЕСТО ЯЗЫКА LISP ПРИ ПРЕПОДАВАНИИ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ

Б. Л. Файфель[1] [0000-0002-1674-2135], Л. В. Городняя[2] [0000-0002-4639-9032]

[1]*Саратовский государственный технический университет им. Ю. А. Гагарина, г. Саратов, Россия*

[2]*Институт систем информатики им. А. П. Ершова Сибирского отделения РАН, г. Новосибирск, Россия*

[2]*Новосибирский государственный университет, г. Новосибирск, Россия*

[1]catstail@ya.ru, [2]lidvas@gmail.com

Аннотация

Рассмотрены ключевые проблемы преподавания функционального программирования обучаемым, уже знакомым с императивной парадигмой. Описаны модель обучаемого и основные сложности, возникающие при преподавании функционального программирования в этом случае (изменяемые переменные, циклы, последовательные вычисления). Приведен развернутый пример перехода от императивной к функциональной парадигме. Подробно рассмотрен возврат функционального значения на примерах численного дифференцирования и интерполяции. Рассмотрена реализация отложенных вычислений, основанная на анонимных функциях. Показано, что использование мультипарадигменного языка Lisp удобно для первого знакомства с функциональной парадигмой.

Ключевые слова*: язык программирования, Lisp, Common Lisp, HomeLisp, функциональное программирование.*

ВВЕДЕНИЕ

Основные трудности, встающие перед преподавателями, ведущими курсы по функциональному программированию (ФП), состоят в том, что подавляющее число студентов, изучающих ФП, уже «впитали» концепцию императивного программирования, и преподавателям приходится не столько учить, сколько переучивать. При этом естественно, что сам преподаватель должен

иметь четкое представление о функциональной парадигме программирования. В работе описана модель обучаемого и рассмотрены сложности, возникающие при преподавании ФП в этом случае.

МОДЕЛЬ ОБУЧАЕМОГО

По нашему мнению, студент, изучающий в настоящее время чистое функциональное программирование, уже знаком с каким-либо «традиционным» языком программирования (C, Pascal, Java, Python). Это порождает следующие проблемы:

- трудности вызывает слишком разнообразный синтаксис таких языков, как Хаскелл, отвлекающий внимание от принципов ФП в пользу выбора между формами представления;
- сознание обучаемого естественным образом сопротивляется идеям ФП (поскольку трудно отказаться от изменяемых переменных и циклов, хотя наличие присваиваний и циклов для ФП не является принципиальным, их при желании можно моделировать средствами ФП, присваивания проще всего выполнять через локализацию, а циклы – через функции);
- обучаемый (хорошо знакомый с императивным программированием) уже способен создавать программные системы, и он не ощущает необходимости в ФП при решении учебных задач[1].

Таким образом, при преподавании основ ФП акценты, по нашему мнению, нужно расставлять следующим образом:

- показать (постоянно подчеркивать) преимущества ФП (лаконичность, наглядность, простота) и обязательно указать, для каких задач эти свойства особенно важны;
- показать, что переход к функциональной парадигме не так сложен, как это кажется на первый взгляд.

[1] Потребность в ФП возникает при переходе к сложным задачам, например, параллелизму.

СРАВНИТЕЛЬНАЯ ХАРАКТЕРИСТИКА ИМПЕРАТИВНОЙ И ФУНКЦИОНАЛЬНОЙ ПАРАДИГМ

Далее кратко рассмотрим основные характеристики двух парадигм программирования: императивной и функциональной [1].

Императивный подход характеризуется следующим:

- программная единица представляет собой набор операторов;
- активно используются изменяемые переменные, основное назначение которых – это предотвращение повторного вычисления;
- активно используются циклы, назначение которых – также повторное использование кода;
- активно используются последовательные вычисления;
- функции не являются объектами[2] первого класса [2].

Функциональный подход характеризуется следующим:

- программная единица представляет собой набор функций, связанных взаимными вызовами;
- изменяемые переменные значения не используются, а для предотвращения повторного вычисления используется специальный синтаксис;
- явные циклы не используются, для повторения используется рекурсия;
- последовательные вычисления в чистых функциональных языках присутствуют как упорядочение иерархии формул изнутри наружу, что преодолевается при необходимости ленивыми вычислениями;
- представления функций являются объектами первого класса: функция может быть параметром другой функции и функция может быть возвращена как результат.
- существует универсальная функция, способная вычислять результат любого правильно устроенного выражения.

Императивное программирование завладело умами неслучайно – пошаговый подход ближе к мышлению большинства людей. Даже машина Тьюринга императивна по своей сути.

[2] В статье слово «объект» используется в широком методологическом смысле этого слова (предмет, на который направлена деятельность), а не в том смысле, который придается этому термину в объектно-ориентированной парадигме.

ОСНОВНЫЕ ПРОБЛЕМЫ, ВОЗНИКАЮЩИЕ ПРИ ПРЕПОДАВАНИИ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ

Опыт преподавания показывает, что для «традиционного» программиста (привыкшего, скажем, к Java) переход сразу на чистый функциональный язык (типа Haskell [3]) вызывает значительные трудности. Поэтому представляется методически обоснованным использовать для первоначального обучения функциональному языку мультипарадигменный язык (в котором возможны разные подходы к решению одной и той же задачи). Таких мультипарадигменных языков в настоящее время известно несколько. Это, в первую очередь, разумеется, классический Lisp (Common Lisp [4] или Scheme [2] – не столь важно). Уже первые реализации языка Lisp поддерживали четыре парадигмы: кроме функциональной парадигмы это: метапрограммирование, декларативное и локально-императивное. Последнее не производит внешнего побочного эффекта. Кроме языка Lisp следует отметить такие языки, как Scala [5] и Erlang [6].

Lisp является весьма «влиятельным» языком, породившим большое число диалектов и языков-наследников, а также через функциональную парадигму – много производных и гибридных парадигм (Lisp – корень огромного дерева). Таким образом, знакомство с языком Lisp расширит кругозор обучаемого более значительно, чем при использовании в качестве базы какого-либо другого языка.

Современное промышленное программирование не мыслится без объектно-ориентированной парадигмы (ООП). Базовые идеи ООП прекрасно «вписаны» в Lisp (унаследованы языком Lisp от lambda-исчисления; инкапсуляции соответствует именование переменных, полиморфизму – многократные локальные определения; наследованию – свободные переменные). Поэтому Lisp не нуждается в идее ООП[3], она уже реализована как свойства или значения атомов.

[3] Об этом любит писать Пол Грэм [4].

Другими преимуществами языка Lisp в описываемом контексте являются его простота, прозрачность и низкий порог вхождения. В настоящей работе в качестве базового языка будет использован Lisp (в реализации HomeLisp [7, 8]). Все приводимые ниже примеры могут быть легко переведены в любую версию языка Lisp.

К основным проблемам традиционного императивного программиста при переходе на «функциональное поле» можно отнести запрет на использование глобальных изменяемых переменных (явных присвоений) и циклов, а также традиционных последовательных вычислений. Как известно, для реализации последовательных вычислений в некоторых чистых функциональных языках служит концепция монад [3]. Монады появились в языке Хаскелл, но сейчас этот подход появляется и в других языках (JavaScript, Java)[4]. Возможно использование монадических вычислений и в языке Lisp, но это достаточно сложный вопрос, и в настоящей статье он не рассматривается[5].

Время от времени раздаются призывы в пользу того, что ФП следует преподавать перед императивным. Особенно, когда обучаемые – математики (математика функциональна по своему духу). С точки зрения методики преподавания, это обосновано и рационально (переход от ФП к императивному проще, чем обратный переход). Но воплотить такой подход в реальность можно было лет шестьдесят назад (когда в программе средней школы не было предмета «информатика»; сейчас все студенты первого курса в той или иной мере уже «инфицированы» императивной парадигмой). Поэтому проблема перехода от императивной парадигмы к функциональной остается актуальной.

Таким образом, задача преподавателя при обучении основам функционального программирования состоит в смягчении трудностей перехода. Ниже рассмотрено несколько примеров того, как можно, действуя достаточно «мягко», превратить императивный итеративный код в функциональный, хотя легче осуществить обратный переход.

---

[4]URL: https://habr.com/ru/companies/otus/articles/800957/

[5]Примером монадического подхода можно считать функцию Prog, поддерживающую локально-императивные вычисления.

СОПОСТАВЛЕНИЕ ИМПЕРАТИВНОГО И ФУНКЦИОНАЛЬНОГО КОДОВ

Начнем с очень простого примера: вычисления суммы элементов числового списка.

Для «императивного» программиста решение этой задачи выглядит достаточно просто и традиционно:

- завести переменную s для будущей суммы;
- просматривать список элемент за элементом, до его окончания;
- прибавлять значение очередного элемента к значению переменной s;
- после исчерпания списка переменная s будет содержать требуемый результат.

Вот как выглядит это решение на Lisp в чисто императивном стиле[6]:

```
(defun sum-list (list)
 (prog (s)
   (setq s 0)
@lab_1: (cond ((null list) (return s)))
   (setq s (+ s (car list)))
   (setq list (cdr list))
   (go @lab_1:)))

(sum-list '(1 2 3 4 5))
==> 15
```

Разумеется, этот код несколько карикатурен (он даже использует явный переход goto[7]). В языке Lisp имеются различные реализации циклов (от простейших до весьма изощренных). Например, с помощью конструкции dolist функция вычисления суммы элементов списка может быть несколько «облагорожена»:

```
(defun sum-list (list)
 (let ((s 0))
  (dolist (a list s)
```

[6] Можно сказать, в монаде императивного программирования

[7] Неодобряемый методикой структурного программирования

```
    (setq s (+ a s)) )))

(sum-list '(1 2 3 4 5))
==> 15
```

Этот код выглядит значительно лучше предыдущего, однако он при этом не перестает быть императивным – содержит явное присвоение[8].

Использовав более продвинутую циклическую конструкцию iter[9], рассматриваемую задачу можно решить буквально двухстрочным кодом:

```
(defun sum-list (list)
 (iter (for a in list) (summing a)))

(sum-list '(1 2 3 4 5))
==> 15
```

Приведенный выше код уже не выглядит императивным, т. к. не содержит явного присвоения. Реальная функциональная чистота этого кода зависит от реализации iter.

Теперь рассмотрим рекурсивное решение этой задачи. Оно базируется на двух фактах:

- сумма элементов пустого списка равна нулю;
- сумма элементов непустого списка равна сумме первого элемента плюс сумма элементов хвоста списка.

Эти соображения порождают следующий код:

```
(defun sum-list (list)
  (if (null list) 0 (+ (car list) (sum-list (cdr list)))))

(sum-list '(1 2 3 4 5))
==> 15
```

[8] Впрочем, в языках Scheme и Racket присваивание унифицировано с объявлением функции.
[9] URL: https://dspace.mit.edu/bitstream/handle/1721.1/41498/AI_WP_324.pdf?sequence=4

Такой код уже не содержит никаких явных присвоений и является «чистым». Хотя этот код практически столь же лаконичен, как и код, использующий iter, он имеет очевидный недостаток – большой расход памяти на рекурсивные вызовы (что, разумеется, может быть исправлено переходом к хвостовой рекурсии).

Впрочем, обычно в языках ФП, начиная с языка Lisp, арифметические операции обычно реализованы как мультифункции, использующие неявные циклы при обработке серийных аргументов.

```
(+ 1 2 3 4 5)
==> 15
```

Такое выражение можно строить при вычислении и вычислять его, используя универсальную функцию eval, что дает выход на метапрограммирование.

```
(defun ar-list (op list)
  (eval (cons op list)))
(ar-list '+ '(1 2 3 4 5))
==> 15
```

Более того, выбор операции можно производить как ввод данного при вычислении, хотя это выводит за границы чисто функционального стиля, понимаемого как константные вычисления.

```
(defun ar-list (list)
   (eval (cons (read) list)))  ;; представление операции будет введено
(ar-list '(1 2 3 4 5))
==> 15  ;; если был введен «+»
==> 120 ;; если был введен «*»
```

Имеется и неочевидное, на первый взгляд, достоинство рекурсивного подхода. Чтобы выяснить это, чуть усложним задачу. Пусть наш список станет многоуровневым. Для таких списков ни один из приведенных выше кодов работать не будет.

Рассмотрим, каким образом можно доработать императивный код. Один из путей такой доработки может быть таким:

- завести стек и занести в него исходный список;
- завести аккумулятор и занести в него нуль;
- до исчерпания стека выполнять следующие действия:
- брать из стека его вершину (это будет список);
- просматривать этот список элемент за элементом;
- если очередной элемент – число, то прибавить его к аккумулятору;
- если очередной элемент – список, то занести его в стек.

Это порождает следующий код:

```
(defun sum-list (list)
 (let ((stack (list list))
    (q nil)
    (s 0))
  (loop
   (when (null stack) (return s))
   (setq q (pop stack))
   (iter (for a in q) (if (numberp a) (summing a into s) (push a stack))))))

(sum-list '(1 (2 3) ((4)) 5))
==> 15
```

Как можно убедиться, код значительно проиграл в лаконичности и прозрачности. Между тем рекурсивное решение может быть очень легко доработано для случая многоуровневых списков. Для этого нужно просто добавить обработку еще одного условия: если голова списка есть число, то следует прибавить это число к результату рекурсивного вызова функции на хвосте списка. Иначе говоря, результат равен сумме рекурсивных вызовов на голове и хвосте:

```
(defun sum-list (list)
 (if (null list) 0
   (if (numberp (car list))
      (+ (car list) (sum-list (cdr list)))
      (+ (sum-list (car list)) (sum-list (cdr list))))))
```

```
(sum-list '(1 (2 3) ((4)) 5))
==> 15
```

Приведенный пример показывает, что рекурсивный код оказывается более универсальным, чем императивный, и он легче поддается модификации.

Рассмотрим еще один классический пример, показывающий «изжитие» из кода явных присвоений. Речь пойдет о расчете чисел Фибоначчи. Как хорошо известно, каноническая последовательность Фибоначчи начинается с двух единиц, а каждый последующий член равен сумме двух предыдущих. Это определение порождает следующий очевидный код (регулярно переходящий из пособия в пособие):

```
(defun fib (n)
  (if (< n 2) 1 (+ (fib (- n 1) (- n 2)))
```

Хорошо известно, что этот код крайне неэффективен, т. к. порождает древовидную рекурсию, что вызывает экспоненциальное увеличение объема вычислений с ростом n. Полезным упражнением для студентов является подсчет количества рекурсивных вызовов функции fib в зависимости от n. Самое простое решение в данном случае – это завести глобальную переменную, предварительно обнулить ее, и при каждом вызове fib увеличивать эту переменную:

```
(setq *c* 0)
(defun fiba (n)
   (setq *c* (+ 1 *c*))
   (if (<= n 1) 1 (+ (fiba (- n 1)) (fiba (- n 2)))))

(fiba 16)
==> 1597
*c*
==> 3193
```

Однако использование глобальных переменных[10] – это «вопиющее нарушение» функциональной чистоты (не поощряемое и в чисто императивных языках[11]). Как же можно организовать подсчет числа вызовов без использования глобальных переменных (и без явных присвоений)? Введем в интерфейс функции еще один (накопительный) параметр:

```
(defun fibb (n &optional (c 1))
 (if (<= n 1) (list 1 c)
  (let* ((t1 (fibb (- n 1) (+ c 1)))
      (t2 (fibb (- n 2) (+ (cadr t1) 1))))
    (list (+ (car t1) (car t2))
      (cadr t2)))))

(fibb 16)
==> (1597 3193)
```

Этот код несколько более громоздок, чем код, использующий глобальную переменную, однако он является почти функционально чистым. Слово «почти» здесь не случайно – код содержит последовательное вычисление (форма let*).

В качестве другого классического примера рассмотрим алгоритм «Ханойской башни» – имеется доска с тремя стержнями, на одном из них нанизано n дисков с диаметрами, уменьшающимися снизу навверх. Необходимо построить алгоритм, перемещающий все диски с исходного стержня на заданный. При этом за один ход можно перемещать только один диск и нельзя класть больший диск на меньший.

Нерекурсивное решение этой задачи является, на наш взгляд, весьма нетривиальным [9].

Рекурсивная же реализация этого алгоритма очень проста (стержни имеют номера 1, 2 и 3):

[10] Хотя сохраняются глобальные определения функций.

[11] Первый механизм оптимизации программ.

```
(defun move (from to)
  (print[12] from)
  (prints " -> ")
  (printline to))

(defun hanoi (n from to)
 (cond ((= n 1) (move from to))
    (t (hanoi (- n 1) from (- 6 from to))
           (move from to)
    (hanoi (- n 1) (- 6 from to) to))))
```

Снова, как и в примере о числах Фибоначчи, попробуем подсчитать число выполненных шагов (или, что то же самое, перенумеровать шаги). Способ с использованием глобальной переменной реализуется вполне очевидным образом – переработаем функцию move:

```
(defun move (from to)
  (setq *c* (+ *c* 1))
  (print *c* )
  (prints " ")
  (print from)
  (prints " -> ")
  (printline to))
```

А для решения проблемы в функциональном стиле нужно снова завести еще один накопительный параметр у функций hanoi и move:

```
(defun move (from to step)
  (prints (fix2str (+ step 1)))
  (prints ". ")
  (print from)
  (prints " -> ")
  (printline to))
```

[12] Вывод данных часто исключают из чисто функционального стиля, хотя он не противоречит его принципам, не нарушает соответствия между аргументами и результатами.

```
(defun hanoi (n from to &optional (step 0))
 (cond ((= n 1) (move from to step) (+ step 1))
   (t (let* ((c1 (hanoi (- n 1) from (- 6 from to) step))
        (_ (move from to c1)))
      (hanoi (- n 1) (- 6 from to) to (+ c1 1))))))
```

Здесь новая функция hanoi возвращает номер следующего шага. Как можно убедиться, «явные присвоения» устранены, однако присутствуют формы, неявно использующие последовательные вычисления (let*, cond).

Однако не следует считать, что рекурсия есть единственный способ превращения императивного кода в функциональный. Другим способом является использование стандартных функционалов (map, filter, reduce).

Так, суммирование списка с помощью reduce выполняется одним вызовом:

```
(defun sum-list (list)
  (reduce '+ list))
```

Суммирование же многоуровневого списка и при использовании функционалов тоже требует рекурсии, но решается достаточно просто:

```
(defun sum-list (list)
  (reduce '+ (mapcar (lambda (x) (if (numberp x) x (sum-list x))) list)))
```

Убедив аудиторию в естественности и удобстве функционального подхода, можно перейти к специфическим возможностям, присущим только функциональной парадигме.

ВОЗВРАТ ИЗ ФУНКЦИИ ФУНКЦИОНАЛЬНОГО ЗНАЧЕНИЯ (ЗАМЫКАНИЯ)

Передача в функцию функционального значения – это в настоящее время достаточно популярный прием программирования. Такая возможность может присутствовать и в традиционных языках (например, в C – это передача указателя на функцию). Другое дело – возврат функционального значения. Для этого язык программирования должен поддерживать безымянные функции (или локальные функции, как это позволяет Python). В действительности поддержка

безымянных функций вполне достаточна. Ниже рассмотрены два не слишком тривиальных примера возврата функционального значения.

Первый пример – это построение интерполяционного полинома Лагранжа. Как хорошо известно [10], если имеется *n* попарно различных точек плоскости с координатами (x_i, y_i), то существует единственный полином степени *n*–1, который в точках x_i будет принимать значения y_i. Этот полином может быть представлен в форме Лагранжа.

$$P(x) = y_1 \frac{(x-x_2)\dots(x-x_n)}{(x_1-x_2)\dots(x_1-x_n)} + y_2 \frac{(x-x_1)\dots(x-x_n)}{(x_2-x_1)\dots(x_2-x_n)} + \dots + y_n \frac{(x-x_1)\dots(x-x_{n-1})}{(x_n-x_1)\dots(x_n-x_{n-1})}.$$

Обычно студентам дают задания вида «протабулировать функцию с помощью интерполяционного полинома». Интерфейс такой функции при традиционном подходе предполагает, что на вход передаются массивы {x_i}, {y_i} и аргумент *x*, а на выходе получается значение интерполяционного полинома. Если эта функция потом применяется для табуляции, т. е. вызывается в цикле, то становится очевидным, что ее интерфейс «перегружен» – на каждом витке цикла нужно передавать на вход два массива.

Использование функционального подхода позволяет построить функцию, которая вернет полином Лагранжа как функциональный объект. Этот функциональный объект может быть использован где угодно (например, для табуляции) уже без громоздкого задания массивов коэффициентов при каждом вызове. Ниже приведен пример реализации функции, принимающей на вход списки коэффициентов, а возвращающей полином Лагранжа как функциональный объект.

```
;; Исходные списки
(setq *x* '(0 1 2 3 4 5 6))
(setq *y* '(-7 0 11 13 16 19 22))

;; Интерполяционный полином Лагранжа как функциональный объект
(defun lagrange (xs ys)
  (lambda (a)
   (apply '+ (mapcar (lambda (xa y)
       (* y (apply '* (mapcar (lambda (xb)
              (if (/= xa xb) (/ (- a xb) (- xa xb)) 1)) xs)))) xs ys))))
```

```
;; lagr – функциональный объект
(setq lagr (lagrange *x* *y*))

;; Проверка
 (iter (for x from -1 to 8) (printline (list x (lagr x))))

;; Результат вывода:
(-1 106)
(0 -7)
(1 0)
(2 11)
(3 13)
(4 16)
(5 19)
(6 22)
(7 84)
(8 427)
```

Легко убедиться, что значения полученной функции в опорных точках интерполяции в точности совпадают со значениями, заданными в списке *y*. Преимуществами такого подхода, на наш взгляд, являются более внятный интерфейс и высокая наглядность. Кстати, чисто функциональное решение, использующее стандартные функционалы Lisp apply и mapcar, выглядит весьма лаконичным.

В качестве второго примера рассмотрим задачу прямого численного дифференцирования функции. Этот алгоритм основан на вычислении предела разностного отношения приращения функции к приращению аргумента.

Снова, как и выше, построим функцию, которая будет принимать на вход дифференцируемую функцию (как функциональный объект), а возвращать ее производную (тоже как функциональный объект). Разумеется, приведенный подход не следует путать с аналитическим символьным дифференцированием функции (хотя результаты оказываются близкими).

```
;; Вспомогательная функция, вычисляющая предел разностного
;; отношения с точностью eps
(defun hderive (f pdf x dx &optional (eps 1.0e-5))
  (let ((df (/ (- (f (+ x dx)) (f x)) dx)))
   (if (<= (abs (- df pdf)) eps) df (hderive f df x (* 0.5 dx) eps))))

;; Функция, возвращающая производную заданной
(defun derive (f &optional (epsilon 1.0e-5))
 (lambda (x &optional (dx 0.1) (eps epsilon))
  (let* ((pdf (/ (- (f (+ x dx dx)) (f x)) dx)))
           (hderive f pdf x dx eps))))

;; Служебная функция для выравнивания печати
(defun strAlign (stri n)
  (let ((sz (strLen stri)))
    (if (> sz n) (strLeft stri n) (strCat (strSpace (- n sz)) stri))))

;; Тестовая функция
(defun test nil
 (let ((f1 (derive 'sin)) ;; f1 = cos
   (f2 (derive 'log))) ;; f2 = 1/x
  (printsline "+----------+----------+----------+----------+----------+")
  (printsline "|  x   | sin'(x) | cos(x) | log'(x) |  1/x  |")
  (printsline "+----------+----------+----------+----------+----------+")
  (iter (for x from 0.1 to 2 by 0.1)
    (printsline (strCat "| " (strAlign (format x "0.00000") 8)
                " | " (strAlign (format (f1 x) "0.00000") 8)
                " | " (strAlign (format (cos x) "0.00000") 8)
                " | " (strAlign (format (f2 x) "0.00000") 8)
                " | " (strAlign (format (/ 1 x) "0.00000") 8)
                " |")))
  (printsline "+----------+----------+----------+----------+----------+")))
```

Запустив эту функцию, убедимся, что результат оказывается вполне предсказуемым:

```
+----------+----------+----------+----------+----------+
|  x   | sin'(x) | cos(x) | log'(x) |  1/x  |
+----------+----------+----------+----------+----------+
| 0.10000 | 0.99499 | 0.99500 | 9.99999 | 10.00000 |
| 0.20000 | 0.98006 | 0.98007 | 4.99999 | 5.00000 |
| 0.30000 | 0.95533 | 0.95534 | 3.33332 | 3.33333 |
| 0.40000 | 0.92105 | 0.92106 | 2.49999 | 2.50000 |
| 0.50000 | 0.87758 | 0.87758 | 1.99999 | 2.00000 |
…    …    …    …    …    …
| 1.90000 | -0.32330 | -0.32329 | 0.52631 | 0.52632 |
| 2.00000 | -0.41615 | -0.41615 | 0.49999 | 0.50000 |
+----------+----------+----------+----------+----------+
```

Теперь можно рассмотреть и пример символьного дифференцирования[13]. Для начала построим простой макет для бинарных формул суммирования и произведений в префиксной форме.

```
(defun diff (e x) ; формула и переменная
  (cond ((atom e)
       (cond ((eq e x) 1) ; простая формула
            ( T 0 ) ) )
      ((eq (car e) '+) (list '+ (diff (cadr e) x) (diff (caddr e) x)) )
      ((eq (car e) '*) (list '+ (list '* (caddr e) (diff (cadr e) x) )
                  (list '* (cadr e) (diff (caddr e) x) )))))

(diff '(+ x (* x x) ) 'x)
==> (+ 1 (+ (* X 1) (* X 1)))
```

Расширение класса дифференцируемых формул сводится к вставке независимых клауз в форму cond. Если нужна инфиксная форма результата, следует определить преобразования префиксной формы в инфиксную и обратно.

[13] В первых описаниях Lisp от группы Маккарти эта задача предлагалась среди 15-минутных задач.

```
defun pref2inf (aexpr)
 (cond ((atom aexpr) aexpr)
     ((eq '+ (car aexpr)) (list (pref2inf (cadr aexpr)) '+ (pref2inf (caddr aexpr))))
     ((eq '- (car aexpr)) (list (pref2inf (cadr aexpr)) '- (pref2inf (caddr aexpr))))
     ((eq '* (car aexpr)) (list (pref2inf (cadr aexpr)) '* (pref2inf (caddr aexpr))))
     ((eq '/ (car aexpr)) (list (pref2inf (cadr aexpr)) '/ (pref2inf (caddr aexpr))))
     ((eq '^ (car aexpr)) (list (pref2inf (cadr aexpr)) '^ (pref2inf (caddr aexpr))))
     ((eq 'sin (car aexpr)) (list 'sin (list (pref2inf (cadr aexpr)))))
     ((eq 'cos (car aexpr)) (list 'cos (list (pref2inf (cadr aexpr)))))
     ((eq 'log (car aexpr)) (list 'log (list (pref2inf (cadr aexpr)))))
     ((eq 'exp (car aexpr)) (list 'exp (list (pref2inf (cadr aexpr)))))))

(pref2inf '(+ (sin x) (cos x)))
==> ((SIN (X)) + (COS (X)))
(pref2inf '(* (sin (* x 3)) (cos (log x))))
==> ((SIN ((X * 3))) * (COS ((LOG (X)))))
```

Определение можно сделать более компактным и гибким:

```
(defun pref2inf (aexpr)
  (cond ((atom aexpr) aexpr)
      ((member (car aexpr) '(+ - * / ^))
       (list (pref2inf (cadr aexpr)) (car aexpr) (pref2inf (caddr aexpr))))
      ((member (car aexpr) '(sin cos log exp ))
       (list (car aexpr) (list (pref2inf (cadr aexpr)))))))
```

Эффективность вычисления можно повысить исключением дублирования формул «(car aexpr)» и «(cadr aexpr)» через промежуточную безымянную функцию:

```
(defun pref2inf (aexpr)
   (if (atom aexpr) aexpr
     ((lambda (x y) ; x = (car aexpr) и y = (cadr aexpr))
            (cond
              ((member x '(+ - * / ^))
       (list (pref2inf y) x (pref2inf (caddr aexpr))))
              ((member x '(sin cos log exp ))
```

```
        (list x (list (pref2inf y))))))
       (car aexpr) (cadr aexpr))))
```

Переход от инфиксной формы к префиксной можно определить примерно как функция «inf2pref»[14]:

```
(defun inf2pref (e)
  (cond  ((atom e) e)
     ((eq (cadr e) '+) (list '+ ( inf2pref (car e))
                 ( inf2pref (caddr e)) ))
     ((eq (cadr e) '*) (list '* ( inf2pref (car e))
                 ( inf2pref (caddr e)) ))
     ((eq (car e) '-) (list '- ( inf2pref (cadr e)) ))
     (T ”неизвестная операция”)))

(inf2pref '((x * z) + (- y)) )
==> (+ (* X Z) (- Y))
```

Как можно убедиться, задача аналитического дифференцирования (весьма нетривиальная для начинающих) решается в Lisp достаточно просто.

ОТЛОЖЕННЫЕ (ЛЕНИВЫЕ) ВЫЧИСЛЕНИЯ

Отложенные вычисления – это стратегия вычислений, при которой запрос на вычисления не исполняется сразу, а как бы «запоминается». Реальное исполнение запроса происходит только в тот момент, когда результат вычислений становится необходимым (например, когда результат нужно напечатать). Первым языком, в котором была реализована концепция отложенных вычислений в качестве основной[15], были языки ML и Hope. Впоследствии эта модель была унаследована языком Haskell.

[14] Приведенная далее функция inf2pref является модельной, т. к. не учитывает приоритеты операций. Более полная версия этой функции более громоздка.

[15] Хотя концепция отложенных вычислений полностью реализуема в языке Lisp с помощью функций Eval и Quote, там она не является основой языка.

Использование отложенных вычислений позволяет экономить ресурсы за счет сокращения вычислений[16]. Другим плюсом этой концепции является возможность работы с бесконечными объектами.

При изучении ФП студентов обычно знакомят с отложенными вычислениями на примере Haskell [3]. Как было сказано, в Haskell отложенные вычисления «встроены» в язык. Это обстоятельство может навести обучаемого на мысль о том, что отложенные вычисления нельзя реализовать, оставаясь в рамках «энергичного»[17] языка (которым отчасти является Lisp[18]).

Между тем языка Lisp позволяет достаточно легко организовать отложенные вычисления [2]. Демонстрация такой реализации имеет, на наш взгляд, большой методический смысл. В классическом учебнике [2] использована Scheme, но все выкладки можно без большого труда перенести в Common Lisp, что и будет показано ниже.

Основой реализации отложенных вычислений служит макро delay:

```
(defmacro delay (aexpr)
  (if (null aexpr) nil `(lambda nil ,aexpr)))
```

Этот макрос принимает выражение, а возвращает это же выражение, но «обернутое» в анонимную функцию. Понятно, что это действие не может быть выполнено обычной функцией, поскольку при вычислении последней ядро языка Lisp предварительно вычислило бы выражение aexpr (в силу энергичности языка Lisp). Обратный апостроф (`), употребленный в теле макро delay, отличается по действию от обычного апострофа тем, что при вычислении такой формы символы, перед которыми стоит запятая, будут вычислены. В макро delay символ aexpr будет заменен своим значением, получаемым при вызове макро.

---

[16] Впрочем, «платить» за этот выигрыш иногда приходится повышенным расходом оперативной памяти.

[17] Энергичным принято называть язык программирования, в котором вычисления выполняются «сразу же» (в частности, при вызове функций их аргументы предварительно вычисляются).

[18] В языке Lisp часть функций являются энергичными (Eval), другие – макросы, позволяющие выполнять и отложенные действия (Quote).

Если на вход delay подать, например, выражение (+ 3 4 5), то макрос вернет следующий результат:

```
(delay (+ 3 4 5))
==> (CLOSURE NIL ((+ 3 4 5)) NIL)
```

Как видим, результатом является замыкание, внутри которого содержится сохраненное выражение, вычисление которого отложено.

Для выполнения (форсирования) отложенного вычисления, служит функция force:

```
(defun force (delayed-object)
   (if (null delayed-object) nil (funcall delayed-object)))
```

Как видим, действие функции force очень просто – применение funcall вызывает вычисление сохраненного замыкания:

```
(setq *u* (delay (+ 3 4 5)))
==> (CLOSURE NIL ((+ 3 4 5)) NIL)

;; Создана глобальная переменная *u*
*u*
==> (CLOSURE NIL ((+ 3 4 5)) NIL)

(force *u*)
==> 12
```

Макро delay и функция force образуют базу реализации отложенных вычислений. На основе этих конструкциях можно построить списки отложенных вычислений. Понятно, что для построения списков нужна модифицированная функция cons, которую можно обозначить как cons!:

```
(defmacro cons! (a b)
   `(cons ,a (delay ,b)))
```

Снова приходится применять макро, поскольку при использовании обычной функции будет вычислено значение параметров.

Для «разбора» подобных списков понадобятся аналоги функций car и cdr, при этом функцию car можно даже не изменять. Однако в целях наглядности эту пару новых функций можно обозначить как car! и cdr!:

```
(defun car! (x) (car x))
(defun cdr! (x) (force (cdr x)))
```

Как можно убедиться, функция car! не отличается от обычной реализации car. А вот cdr! отличается от cdr тем, что к хвосту списка применяется функция force.

Далее, опираясь на разработанные функции, можно достаточно просто реализовать следующие функции:

map! – применить функциональный объект к списку отложенных вычислений и получить на выходе новый такой список;

for-each! – применить функциональный объект к каждому элементу списка отложенных вычислений;

take! – эта функция выполняет первые n отложенных вычислений в цепочке и собирает их в обычный список;

filter! – применить фильтрующий предикат к списку отложенных вычислений.

Реализация описанных функций может быть, например, такой:

```
(defun map! (f s)
 (if (null s) nil (cons! (funcall f (car! s)) (map! f (cdr! s)))))

(defun for-each! (f s)
 (if (null s) 'ok (progn (funcall f (car! s)) (for-each! f (cdr! s)))))

(defun take! (n s)
 (if (zerop n) nil (cons (car s) (take! (- n 1) (cdr! s)))))

(defun filter! (pred s)
 (cond ((null s) nil)
```

```
((funcall pred (car! s)) (cons! (car! s) (filter! pred (cdr! s))))
(t (filter! pred (cdr! s)))))
```

Следует обратить внимание на то, что если первый параметр функции for-each! не выполняет вывода на печать (или каких-то подобных действий), то внешний эффект отсутствует – for-each! в любом случае возвращает атом ok.

Описанная техника позволяет строить списки отложенных вычислений (в том числе и бесконечные). Вот как это может быть реализовано:

```
(defun enum-interval! (l h)
  (if (> l h) nil (cons! l (enum-interval! (+ 1 l) h))))

(defun enum-inf! (p)
  (cons! p (enum-inf! (+ 1 p))))
```

Первая функция представляет собой реализацию «ленивого интервала», а вторая – «ленивое» перечисление целых чисел, начиная с заданного p. Рассмотрим далее манипулирование создаваемыми объектами.

Создадим “ленивый интервал” чисел от 1 до 100:

```
(setq *jj* (enum-interval! 1 100))
==> (1 CLOSURE NIL ((ENUM-INTERVAL! (+ 1 L) H)) ((H 100) (L 1)))
```

Как видим, начало интервала (1) уже готово к использованию, а вычисление остальных чисел отложено (алгоритм вычисления и все необходимые переменные содержатся в замыкании). Результат радикально отличается от того, что вернула бы функция range (она вернула бы полный список чисел от единицы до ста).

Можно построить «ленивый» список квадратов этих чисел с помощью функции map!

```
(setq *qq* (map! (lambda (x) (* x x)) *jj*))
==> (1 CLOSURE NIL ((MAP! F (CDR! S)))
    ((S (1 CLOSURE NIL ((ENUM-INTERVAL! (+ 1 L) H))
    ((H 100) (L 1)))) (F (CLOSURE (X) ((* X X)) NIL))))
```

В отличие от «классической» mapcr, которая принимает список и возвращает список, функция map! обеспечивает применение функции к очередному элементу «ленивого» списка, но не выполняет это действие до тех пор, пока результат не будет запрошен:

```
(take! 10 *qq*)
==> (1 4 9 16 25 36 49 64 81 100)

(take! 10 *jj*)
==> (1 2 3 4 5 6 7 8 9 10)
```

В первом случае мы запросили десять первых элементов списка квадратов, во втором – десять первых элементов исходного списка.

К конечному «ленивому» списку можно применить функцию for-each!:

```
(for-each! (lambda (x) (printline (/ 1 x))) *jj*)
1
1/2
1/3
1/4
1/5
…
1/99
1/100
==> OK
```

Здесь функциональный параметр вычисляет и печатает обратные к элементам «ленивого» списка.

Можно создать и бесконечный интервал, а затем применить к нему map!:

```
(setq *uu* (enum-inf! 1))
==> (1 CLOSURE NIL ((ENUM-INF! (+ 1 L))) ((L 1)))

(setq *vv* (map! (lambda (x) (* x x x)) *uu*))
==> (1 CLOSURE NIL ((MAP! F (CDR! S)))
    ((S (1 CLOSURE NIL ((ENUM-INF! (+ 1 L)))
    ((L 1)))) (F (CLOSURE (X) ((* X X X)) NIL))))
```

Теперь можно получить отрезок результирующего списка (списка кубов целых чисел):

```
(take! 10 *vv*)
==> (1 8 27 64 125 216 343 512 729 1000)
```

Однако применение функции for-each! к бесконечным спискам напрямую уже невозможно (вычисление потребует бесконечного времени). Но можно получить конечный отрезок бесконечного списка и распечатать его традиционным способом:

```
(iter (for a in (take! 6 *vv*)) (printline a))
1
8
27
64
125
216
```

Описываемая техника позволяет записать компактный и эффективный код. Ниже приведен код, берущий на вход бесконечный список целых и строящий бесконечный список простых чисел.

```
(defun is-prime (n) ;; является ли число простым
 (let ((res t))
 (cond ((< n 2) nil)
     ((<= n 3) t) ((zerop (mod n 2)) nil)
(t (iter (for i from 3 to (\ n 2))
     (when (zerop (rem n i)) (return (setq res nil)))) res))))
 ==> IS-PRIME

(defun enum-prime! nil
  (filter! 'is-prime (enum-inf! 1)))
==> ENUM-PRIME!
```

```
(setq *pp* (enum-prime!))
==> (2 CLOSURE NIL ((FILTER! PRED (CDR! S)))
    ((S (2 CLOSURE NIL ((ENUM-INF! (+ 1 L)))
    ((L 2)))) (PRED IS-PRIME)))

(take! 10 *pp*)
==> (2 3 5 7 11 13 17 19 23 29)
```

В качестве последнего примера рассмотрим построение бесконечного списка чисел Фибоначчи:

```
(defun enum-fib! (p c)
 (cons! p (enum-fib! c (+ p c))))
==> ENUM-FIB!

(setq *ff* (enum-fib! 1 1))
==> (1 CLOSURE NIL ((ENUM-FIB! C (+ P C))) ((C 1) (P 1)))

 (take! 6 *ff*)
==> (1 1 2 3 5 8)
```

Как можно убедиться, языка Lisp позволяет реализовать отложенные вычисления достаточно простыми средствами, что особенно важно при обучении.

ЗАКЛЮЧЕНИЕ

Таким образом, показано и обосновано приведенными примерами, что язык Lisp (несмотря на почтенный возраст[19]) хорошо подходит для первоначального обучения основам ФП, включая переход к метапрограммированию. Удобство языка Lisp состоит в простом синтаксисе и низком пороге вхождения. Приведен пример «мягкого перехода» от чисто императивного кода к функциональному. Количество таких примеров можно было без труда увеличить. Мы выражаем надежду, что приведенный материал будет полезен преподавателям функционального программирования.

[19] Может быть, благодаря длительной проверке, показавшей достоинства его решений.

Благодарности

Авторы искренне благодарны Андрею Валентиновичу Климову за ценные рекомендации по улучшению стиля изложения, Николаю Вячеславовичу Шилову за интерес к языку Lisp и стимулирующие вопросы.

СПИСОК ЛИТЕРАТУРЫ

1. *Городняя Л.В.* Функциональное программирование. Парадигма, модели и методы. Новосибирск, СО РАН, 2022. 482 с.

2. *Абельсон Х., Сассман Дж.Дж.* Структура и интерпретация компьютерных программ. М.: Добросвет, 2010. 608 с.

3. *Липовача М.* Изучай Haskell во имя добра. М.: ДМК Пресс, 2012. 490 с.

4. *Грэм П.* ANSII-Common Lisp. СПб.: Символ-Плюс, 2012. 448 с.

5. *Хостман К.* Scala для нетерпеливых. М.: ДМК Пресс, 2013. 408 с.

6. *Хеберт Ф.* Изучай Эрланг во имя добра. М.: ДМК Пресс, 2015. 688 с.

7. *Файфель Б.Л.* HomeLisp – простая реализация Лисп 1.5 для целей обучения // Вестник НГУ, Серия Информационные технологии. 2012. Т. 10. Вып. 3. С. 105–116.

8. *Файфель Б.Л.* Новые возможности системы HomeLisp // Языки программирования и компиляторы – 2017: труды всерос. науч. конф. памяти А.Л. Фуксмана, г. Ростов-на-Дону, 3–5 апр. 2017 г. 2017. С. 252–254.

9. *Ламуатье Ж-П.* Упражнения по программированию на Фортране-IV. М.: Мир, 1978. 162 с.

10. *Курош А.Г.* Курс высшей алгебры. СПб: Лань, 2025. 432 с.

THE PLACE OF LISP IN TEACHING FUNCTIONAL PROGRAMMING

B. L. Faifel[1] [0000-0002-1674-2135], L. V. Gorodnyaya[2] [0000-0002-4639-9032]

[1]Saratov State Technical University named after Yuri Gagarin, Saratov, Russia

[2]Institute of Informatics Systems named after A. P. Ershov, Siberian Branch of the Russian Academy of Sciences, Novosibirsk, Russia

[2]Novosibirsk State University, Novosibirsk, Russia

[1]catstail@ya.ru, [2]lidvas@gmail.com

Abstract

This article examines the main problems of teaching functional programming to students already familiar with the imperative paradigm. The learner model and the main problems that arise when teaching functional programming in this case (mutable variables, loops, sequential calculations) are described. A detailed example of the transition from an imperative to a functional paradigm is given. The return of a functional value is examined in detail using examples of numerical differentiation and interpolation. An implementation of lazy evaluation based on anonymous functions is discussed. It is shown that the multi-paradigm Lisp language is a convenient introduction to the functional paradigm.

***Keywords**:* programming language, Lisp, Common Lisp, HomeLisp, functional programming.

REFERENCES

1. *Gorodnyaya L.V.* Funcionalnoe programmirovanie. Paradigma, modeli i metody. Novosibirsk, SO RAN, 2022. 482 p.
2. *Abelson X, Sussman Jh.* Stuctura i interpretaciya komputernych program/ M.: Dobrosvet, 2010. 608 p.
3. *Lipovacha M.* Izuchai Haskell vo imya dobra. M.: DMK Press, 2012. 490 p.
4. *Gaham P.* ANSII-Common Lisp. SPb.: Cymvol-Plus, 2012. 448 p.
5. *Chostman R.* Scala dlya neterpelivych. M.: DMK Press, 2013. 408 p.
6. *Chebert F.* Izuchai Erlang vo imya dobra. M.: DMK Press, 2015. 688 p.

7. *Faifel B.L.* HomeLisp – prostaya realizaciya Lisp 1.5 dlya celei obychenia // Vestnik NGU, Seriya Informacionnye technologii. 2012. Vol. 10. Num. 3. P. 105–116.

8. *Faifel B.L.* Novye vosmoznosti cictemy HomeLisp. // Yasiki programmirovaniya i kompilyatory – 2017: Trudy vseros. nauch. konf. pamyati A.L. Fuksmana, g. Rostov-na-Donu, 3–5 apr. 2017 г. 2017. P. 252–254.

9. *Lamuatie Z-P.* Upraznenia po programmirovaniu na Fortran-IV. M.: Mir, 1978. 162 p.

10. *Kurosh A.G.* Kurs vishei algerby. SPb: Lan', 2025. 432 p.

СВЕДЕНИЯ ОБ АВТОРАХ

ФАЙФЕЛЬ Борис Леонидович – к. ф.-м. н., доцент Саратовского технического университета, специалист в области образовательной информатики.

Boris Leonidovich FAIFEL – associate professor at Saratov State Technical University, specialist in educational informatics.

email: catstail@ya.ru

ORCID: 0000-0002-1674-2135

ГОРОДНЯЯ Лидия Васильевна – к. ф.-м. н., старший научный сотрудник Института систем информатики имени акад. А.П. Ершова СО РАН, доцент Новосибирского государственного университета, специалист в области системного программирования и образовательной информатики.

Lidia Vasiljevna GORODNYAYA – senior researcher at A. P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences, associate professor at the Novosibirsk State University, specialist in system programming and educational informatics.

email: gorod@iis.nsk.su

ORCID: 0000-0002-4639-9032

Материал поступил в редакцию 17 января 2026 года