Лисп как программная основа при преподавании функционального программирования

Б.Л. Файфель, СГТУ им. Гагарина Ю.А., Саратов,

Л.В. Городняя, Институт систем информатики им. А.П. Ершова СО РАН, Новосибирск

Аннотация. В статье рассматриваются основные проблемы преподавания функционального программирования для обучаемых, уже знакомых с императивной парадигмой. Описана модель обучаемого и основные проблемы, 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 примере численного дифференцирования. Показано, что использование мультипарадигменного языка Лисп удобно для первого знакомства и функциональной парадигмой.

Ключевые слова: функциональное программирование, Лисп, HomeLisp

Lisp as a programming base for teaching functional programming

B.L. Faifel, SSTU named after Gagarin Yu.A., Saratov

L. V. Gorodnyaya, A.P. Ershov Institute of Informatics Systems

Abstract. The article considers the main problems of teaching functional programming to students already familiar with the imperative paradigm. The model of the student and the main problems that arise when teaching functional programming in this case (changeable variables, cycles, sequential calculations) are described. A detailed example of the transition from the imperative to the functional paradigm is given. The return of a functional value is considered in detail using the example of numerical differentiation. It is shown that the use of the multi-paradigm language Lisp is convenient for the first acquaintance with the functional paradigm.

Keywords: Functional programming, Lisp, HomeLisp

Введение

Настоящая заметка рассчитана на начинающих преподавателей, преподающих функциональное программирование. Естественно, что преподаватель должен сам иметь четкое представление о функциональной парадигме программирования. Основные трудности, встающие перед таким преподавателем, состоят в том, что подавляющее число студентов, изучающих функциональное программирование, уже “впитали” концепцию императивного программирования и преподавателю приходится не столько учить, сколько переучивать.

Модель обучаемого

По мнению авторов настоящей заметки, студент, изучающий сейчас 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 функционального программирования в пользу выбора между формами представления;
- Сознание обучаемого естественным образом сопротивляется идеям функционального программирования (поскольку трудно отказаться от изменяемых переменных и циклов, хотя наличие присваиваний и циклов для ФП не является принципиальным, их при желании можно моделировать средствами функционального программирования, присваивания проще всего через локализацию, а циклы через функции);
- Обучаемый (хорошо знакомый с императивным программированием) уже способен создавать программные системы и он не ощущает необходимости в функциональном программировании.

Таким образом, в преподавании основ функционального программирования акценты, по мнению авторов, нужно расставить следующим образом:

- Показать (постоянно подчеркивать) преимущества функционального подхода (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.

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 – предотвращение повторного вычисления;
- Активно используются циклы, назначение которых – также повторное использование кода;
- Активно используются последовательные вычисления;
- Функции не являются объектами[1] первого класса [2].

Функциональный подход характеризуется следующим:

- Программная единица представляет собой набор функций, связанных взаимными вызовами;
- Изменяемые переменные 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;
- Функции 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 не случайно – императивный подход ближе к мышлению большинства людей. Даже машина Тьюринга императивна по своей сути.

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

---

[1] В статье слово “объект” используется в широком методологическом смысле этого слова (предмет, на который направлена деятельность), а не в смысле, который придаётся этому термину в ООП.

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 Лисп (Common Lisp [4] или Scheme [2] - не столь важно). Уже первые реализации Лиспа поддерживали четыре парадигмы: кроме функциональной парадигмы поддерживались: метапрограммирование, декларативное и локально-императивное. Последнее не производит внешнего побочного эффекта. Кроме Лиспа следует отметить такие языки как Scala [5] и Erlang [6].

Лисп является весьма “влиятельным” языком, породившим большое число диалектов и языков-наследников, а также через функциональную парадигму – много производных и гибридных парадигм (Лисп – корень огромного дерева). Таким образом, знакомство с Лиспом 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 Лисп (унаследованы Лиспом от lambda-исчисления; инкапсуляции соответствует именование переменных, полиморфизму – многократные локальные определения; наследованию – свободные переменные). Поэтому Лисп не нуждается в идее ООП. Она уже реализована как свойства или значения атомов.[2]

Другим преимуществом Лиспа в описываемом контексте является его простота, прозрачность и низкий порог вхождения. В этой заметке в качестве базового языка будет использоваться Лисп (в реализации HomeLisp [7-8]). Все приводимые ниже примеры могут быть легко переведены в любую версию Лиспа.

К основным проблемам традиционного императивного программиста при переходе на “функциональное поле” можно отнести запрет на использование изменяемых переменных (явных присвоений) и циклов, а также традиционных последовательных вычислений. (Как известно, для реализации последовательных вычислений в чистых функциональных языках служит концепция монад [3]. Монады появилась в языке Хаскелл, но сейчас этот

---

[2] Об этом любит писать Пол Грэм [4]

подход появляется и в других языках (JavaScript, Java)[3]. Возможно использование монадических вычислений и в Лиспе, но это достаточно сложный вопрос и в настоящей статье он не рассматривается.[4]

Время от времени раздаются голоса в пользу того, что функциональное программирование следует преподавать перед императивным. Особенно, когда обучаемые – математики (математика функциональна по своему духу). С точки зрения методики преподавания, это обосновано и рационально (переход от функционального программирования 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 будет рассмотрено несколько примеров того, как можно, действуя достаточно "мягко", превратить императивный итеративный код в функциональный, хотя легче обратный переход.

Сопоставление императивного и функционального кода

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 Лиспе в чисто императивном стиле[5]:

---

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

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

[5] Можно сказать, в монаде императивного программирования

```
(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[6]). В Лиспе 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)
      (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[7].

Используя более продвинутую циклическую конструкцию iter[8], рассматриваемую задачу можно решить буквально двухстрочным кодом:

---

[6] Неодобряемый методикой структурного программирования

[7] Впрочем, в языках Scheme и Racket присваивание унифицировано с объявлением функции.

[8] URL: https://iterate.common-lisp.dev/doc/Don_0027t-Loop-Iterate.html

```
(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. Рекурсивное решение 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
```

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

Впрочем, обычно в языках функционального программирования, начиная с Лиспа, 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 даёт 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 введён «+»
==> 120 ; если был введён «*»
```

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

Рассмотрим, каким образом можно доработать императивный код. Одним 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 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 (давно кочующий 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 её, 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[9] – это “вопиющее нарушение” функциональной чистоты (не поощряемое и в чисто императивных языках[10]). Как же можно организовать подсчёт 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)))))
```

[9] Хотя сохраняются глобальные определения функций

[10] Первый механизм оптимизации программ.

```
(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 нанизано n дисков с уменьшающимися снизу вверх диаметрами. Необходимо построить алгоритм, перемещающие 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 взгляд авторов настоящей статьи, весьма нетривиальным [9].

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

```
(defun move (from to)
   (print[11] 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:

---

[11] Вывод данных выводит за границы чисто функционального стиля

```
(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))

(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” изжиты, однако присутствуют формы, неявно использующие последовательные вычисления (let*, cond).

Но 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 традиционный 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 Питон). В действительности поддержка безымянных функций вполне достаточна.

Ниже будут рассмотрены два не слишком тривиальных примера возврата функционального значения.

Первый пример – 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 её 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 приводится 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*. Преимуществом такого подхода, на взгляд авторов, является более внятный интерфейс и высокая наглядность. Кстати, чисто функциональное решение, использующее стандартные функционалы Лиспа 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 её 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))))

;; Тестовая функция

(defun test nil
 (let ((f1 (derive 'sin))  ;; f1 = cos
       (f2 (derive 'log))) ;; f2 = 1/x
  (iter (for x from 0.1 to 2 by 0.1)
      (printline (list x (f1 x) (cos x) (f2 x) (/ 1 x))))))
```

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

```
(0.1   0.9949  0.9950   9.9999   10.0)
(0.2   0.9800  0.9800   4.9999   5.0)
(0.3   0.9553  0.9553   3.3333   3.3333)
(0.4   0.9210  0.9211   2.4999   2.5)
(0.5   0.8776  0.8776   1.9999   2.0)
```

Здесь в каждой строке первое значение – это аргумент, следующая пара – это значение производной синуса, возвращаемое нашей функцией и точное значение (косинус). Последняя пара – это значение производной логарифма, возвращаемое нашей функцией и соответствующее точное значение.

Теперь можно рассмотреть и пример символьного дифференцирования[12]. 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 ) )  )
```

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

```
        ((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.

```
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»[13]:

```
(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))
```

Заключение

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

Приведенный выше материал показывает, что язык Лисп (несмотря на почтенный возраст) хорошо подходит для первоначального обучения основам функционального программирования, включая переход к метапрограммированию. Удобство Лиспа состоит в простом синтаксисе и низком пороге вхождения. Приводится пример “мягкого перехода” от чисто императивного кода к функциональному. Количество таких примеров можно было без труда увеличить. Авторы выражает надежду, что приведенный материал будет полезен преподавателю функционального программирования и пользуется приятной возможностью выразить самую искреннюю благодарность Н.В. Шилову (университет Иннополис, г. Казань) за интерес к работе.

Литература

1. Городняя Л.В. “Функциональное программирование. Парадигма, модели и методы” Новосибирск, СО РАН, 2022. - 482 с. ISBN 978-5-6047823-0-9
2. Абельсон Х., Сассман Дж.Дж. “Структура и интерпретация компьютерных программ” М.: Добросвет, 2010. - 608 с. ISBN-978-5-98227-829-6
3. Липовача М. “Изучай Haskell во имя добра” М.: ДМК Пресс, 2012. - 490 с. ISBN-978-5-94074-749-9
4. Грэм П. “ANSII-Common Lisp” СПб.: Символ-Плюс, 2012. – 448 с.
5. Хостман К. “Scala для нетерпеливых” М.: ДМК Пресс, 2013. – 408 с. ISBN-978-5-94074-920-2
6. Хеберт Ф. “Изучай Эрланг во имя добра” М.: ДМК Пресс, 2015. – 688 с. ISBN-978-5-97060-086-3
7. Файфель Б.Л. HomeLisp – простая реализация Лисп 1.5 для целей обучения // Вестник НГУ, Серия Информационные технологии, 2012, том 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. — ISBN 978-5-507-52215-6