Lisp

Lisp — это один из старейших среди существующих языков программирования. Пожалуй, старше только Фортран, но разница состоит в том, что у Лиспа есть будущее, которого у Фортрана, к сожалению, не осталось.

Для начала, пожалуй, следует объяснить фундаментальное отличие функционального программирования от так называемого процедурного, или императивного.

Императивная программа представляет собой набор инструкций, последовательно выполняя которые, исполнитель (компьютер) сможет прийти к определенной цели, причем исполнитель наделен возможностью хранения промежуточных результатов.

Ясно, что при данной схеме программирование сводится к изобретению того, как данная задача может быть решена на компьютере. Таким образом, несмотря на все современные возможности языков программирования, императивная программа не очень сильно отличается от простой последовательности машинных кодов.

Казалось бы, это неплохо, но возникает одна основная проблема при написании кода программист уделяет слишком большое внимание особенностям реализации задачи в данных условиях (например, на архитектуре i386) в ущерб собственно самой задаче. Принципиально иным подходом к программированию является так называемое функциональное программирование.

Основная идея его такова программы строятся из «чистых» математических функций, каждая из которых свободна от побочного эффекта, то есть не в состоянии изменить среду вычислений. Рассмотрим следующий достаточно абстрактный пример. Пусть у нас есть множество. Задав над этим множеством отображение, мы получим его множество значений. Так вот, естественно представить получившееся множество значений в виде совокупности отображения и исходного множества. Таким образом, любое множество представимо в виде совокупности исходного множества и композиции отображений.

Причем, что важно нас не интересуют «промежуточные» результаты действия какой-либо подпоследовательности отображений на исходное множество — они заданы той самой композицией. На каждом шаге в действительности производится одна и та же операция, только в качестве исходного множества берется результат действия предыдущих отображений на самое первое множество, что естественно приводит нас к рекурсии.

В этом случае выполнение программы будет заключаться в изменении формы требуемой итоговой величины так, что, скажем, «8+1», можно заменить на «9», и при этом обе эти формы означают одну и ту же величину.

Подобная идея очень хорошо подходит для решения задач, в которых основную роль играют исходные данные, и задача состоит в замене формы представления данных на какую-либо иную. В эту сферу попадает программирование экспертных систем, логическое программирование, символьные вычисления и т.д.

Существует очень большое количество функциональных языков программирования, но наиболее известным из них, безусловно, является Lisp. Lisp был реализован Джоном Мак-Карти, сотрудником лаборатории искусственного интеллекта Стэнфордского университета, в 1958 году.

Так в чем же заключаются основные возможности Lisp?

Символы и их значения

Для представления объектов окружающего мира мы используем их названия, то есть некоторые символы, ссылающиеся на реальные объекты. Например, «кошка» и тот объект, который этим символом описывается, — не одно и то же, но называем мы его именно так.

Значением символа можно назвать тот самый объект, который мы описываем символом. Очень важно заметить, что значением является именно объект, а не еще один символ для его описания. Безусловно, мы можем получить не сам объект, а, например, его видимую форму, но тем не менее, это будет являться свойством самого объекта, а не привнесенным нами его обозначением.

Лисп оперирует символами (или именами переменных в привычном представлении, что не совсем верно, конечно), причем каждый из символов имеет значение. Таким образом, каждый (!) из символов Лиспа — это отображение из пространства объектов в пространство имен, которым можно воспользоваться в обратную сторону, то есть «вычислить» символ.

Программирование на Лисп предполагает построение некоторой модели предметной области, сущностями которой оперирует программа. Но все же «изнутри» они представлены обычными структурами данных, поэтому мы можем говорить о структурах данных, представляемых символами.

Пусть, скажем, мы каким-либо образом (на самом деле, с помощью макроса SET) сделали так, что символ a стал обозначать число 4. Если мы далее подадим на ввод Лиспа символ a, его значением в любом контексте будет 4.

Символ, значением которого является всегда он сам, называется атомом. Примерами атомов могут служить T и NIL (логическая «истина» и логическая «ложь»), а также число 4 (как, впрочем, и любое другое число). Для того чтобы воспользоваться самим символом a (как реальным объектом), существует возможность блокирования вычисления символа с помощью макроса QUOTE. Примеры использования SET и QUOTE я дам чуть позже, когда вы получите небольшое представление о синтаксисе Lisp.

Структуры данных

Основной и всегда используемой структурой данных Lisp является список. Список строится перечислением символов через пробел и заключением получившегося выражения в скобки, например, (A B C). Поскольку сам список, безусловно, представляется символом, то он может являться элементом других списков — (A (B C (D (E F)))). В этом случае список должен представлять собой правильно построенное скобочное выражение, в котором каждой открывающей скобке соответствует закрывающая. Атомы и списки (то есть то, что раньше мы называли символами) называются еще и символьными выражениями, или s-выражениями.

Внутреннее представление списка в Lisp классическое, то есть список состоит из набора связанных так называемых списочных ячеек — структуры из двух элементов значения и указателя на следующую ячейку. Последняя ячейка ссылается на пустой список (), то есть значением поля указателя является NIL. Следует отметить, что в Лиспе понятия пустого списка и булевского значения «ложь» совпадают, то есть () эквивалентно NIL. Безусловно, кроме списков в Lisp реализованы все обычно используемые типы и структуры данных массивы, хэш-массивы, вектора, структуры, классы и т.п.

Функции. Запись функций и программ

Как и всякое символьное выражение, список должен иметь значение. Вот здесь-то мы и подходим к воистину фундаментальной особенности Лисп, которая резко выделяет его из строя «обычных» языков программирования. Но, впрочем, все по порядку.

Читателю, безусловно, известен так называемый «польский», или префиксный, метод записи функций, при котором функциональный символ ставится перед аргументами. Рассмотрим

польскую запись выражения 2+2: + 2 2. Не правда ли, что-то напоминает? Ну-ка, поставим скобки... Voila! Перед нами лисповский список из трех элементов — (+ 2 2).

Прозорливый читатель уже, конечно, догадался, как записываются функции в Лиспе. Да да, именно в виде списков. Помните, я говорил о значении списка? Так вот, значением списка является результат вызова функции с именем, представляемым в виде первого элемента списка, и остальными элементами списка в качестве аргументов.

Продемонстрируем сказанное (в дальнейшем левая часть строки — это то, что подается на ввод интерпретатора Lisp, а правая часть, после знака =>, — то, что было выдано интерпретатором):

```
(+ 1 2) => 3
```

Теперь можно продемонстрировать упомянутые выше SET и QUOTE:

```
(quote a) => a
```

Лисп-функции могут как вычислять, так и не вычислять свои аргументы прежде чем использовать их. Как правило, функции вычисляют аргументы и используют для работы вычисленные значения. В частности, так работает функция (ну, не совсем функция, но сейчас это не особенно важно) SET.

```
(set (quote a) 1) => 1
a => 1
```

Значение атома 1 (то есть 1) было присвоено значению списка (quote a), то есть результату вызова функции QUOTE, которым, в свою очередь, являлось a.

Приведем еще один пример:

```
(set (quote a) 1) => 1
(set (quote b) 2) => 2
(set b a) => Wrong type argument: symbolp, 2
```

В последнем вызове интерпретатор попытался присвоить значение символа a (то есть 1) значению символа b (то есть 2), которое, в свою очередь, является атомом.

Поскольку вызов QUOTE применяется очень часто, существует его сокращенная запись в виде 'a, что означает именно (quote a), и поскольку вызов (SET (QUOTE также очень част, то его сокращенная запись выглядит как (SETQ.

При выполнении любой из вышеперечисленных операций возвращается новый символ, то есть физическая структура списков не меняется. Опять таки, теоретически любая чисто функциональная программа не должна изменять свое окружение, а вместо этого возвращает новое значение которое в свою очередь, будет аргументом следующей функции (помните наши отображения?). На практике же иногда приходится (либо для ускорения работы, либо для упрощения кода) пользоваться так называемыми разрушающими функциями которые изменяют физическую структуру заданного списка. Примерами таких функций могут служить rplaca и rplacd заменяющие поле значения и поле указателя ячейки соответственно, а также setf выполняющая «обобщенное» присваивание то есть заносящая значение непосредственно в ячейку памяти занимаемую символом.

И наконец финальный аккорд в этом разделе. Как вы помните, я говорил, что функциональная программа может быть записана в виде одной функции (на самом деле, являющейся композицией многих), принимающей исходные данные и возвращающей целевые. Естественно, запись этой функции происходит с помощью списков так как показано выше, а список, в свою очередь является формой представления данных, которыми можно манипулировать, — и в результате, мы получаем, что программы и данные представляются в Лиспе одинаково!

Этот впечатляющий момент дал возможность развития так называемого программирования управляемого данными (data driven programming), при котором поступающие на вход данные обрабатываются путем интерпретации их в качестве элементов кода программы.

Кстати, еще одной из возможностей Lisp, позволяющей реализовать подобное программирование, является возможность обращения к интерпретатору для любого вычислимого выражения внутри программы с помощью вызова EVAL:

```
(eval '(+ 1 2)) => 3
```

Наверное некоторые из читателей заметили что мы выполняя функции иногда не очень заботились о результате который был несколько странным. Дело в том что функции обладают как воз вращаемым значением, так и побочным эффектом (side-effect), который заключается в некотором изменении среды вычислений. Аналогия с императивным программированием прозрачна: функции, в которых нас интересует возвращаемое значение, — это «настоящие» функции, а если нас интересует побочный эффект — это процедуры.

Примером подобной функции с основным результатом в виде побочного эффекта может служить SET, которая на самом деле присваивает символу значение, а возвращает присвоенное значение.

Казалось бы зачем? Ведь раньше я говорил о том, что «чистые» функции не изменяют окружения и с их помощью можно построить все функциональное программирование. Безусловно, это так. Но тут встает вопрос удобства. При написании действительно больших программ приходится жертвовать чистотой идеологии ради комфорта скорости и расширяемости работы. Представляете, что было бы если бы пришлось писать программы лишенные присваивании? Вот именно.

Базовые функции

Во всякой математической системе есть набор аксиом исходя из которых, доказываются шаг за шагом усложняющиеся теоремы которые тем не менее базируются только на этом «минимальном» наборе утверждений принимающихся без доказательства. В Лиспе существует минимальный набор функций с помощью которых, вообще говоря, может быть построена любая программа (то есть более сложная функция) Рассмотрим их на примерах.

CAR/CDR

Функции предназначенные для получения первого элемента («головы» или значения первой списочной ячейки) и списка всех элементов, кроме первого («хвоста» или значения поля указателя первой ячейки):

```
(car '(a b c)) => a
(cdr '(a b c)) => (b c)
(cdr '(a)) => nil
```

Последний пример демонстрирует то, что последняя ячейка указывает на nil. Вызовы car/cdr можно объединять:

```
(car (cdr '(a b c))) => b (cadr '(a b c)) => b ;; car от cdr
```

CONS

Составление нового списка из «головы» и «хвоста», то есть, фактически, создание новой списочной ячейки:

```
(cons 'a '(b c)) => (a b c)
(cons 'a ( )) => (a)
(cons (car '(a b c)) (cdr '(a b c))) => (a b c)
```

ATOM

Предикатом называется выражение, принимающее (для каждого аргумента) одно из двух значений истина или ложь. В Лиспе предикатами называются функции, возвращающие либо T, либо NIL.

Предикат ATOM, как несложно догадаться, возвращает истинное значение, если его аргумент является атомом, и ложь — в противном случае:

```
(atom 'x) => T
(atom '(a b c)) => NIL
(atom nil) => T
(atom '( )) => T                ;; Пустой список - это nil
(atom '(nil)) => NIL            ;; (nil) - одноэлементный список
(atom (atom (+ 2 1))) => T      ;; T - атом
```

В Lisp существует масса предикатов в частности, проверяющих тип аргументов. Например предикат LISTP истинен, если аргумент — список а ARRAYP — если аргумент — массив и т.д.

EO/EQL/EQUAL

Предикат EQ сравнивает два символа и возвращает T, если они идентичны, и NIL — в противном случае:

```
(eq 'x 'x) => T
(eq 'x (car '(x y z))) => T
(eq ( ) nil) => T
```

Им следует пользоваться с большой осторожностью, поскольку он не проверяет логического равенства объектов, а лишь смотрит представлены ли они в памяти одной структурой. Одноименные символы представлены в памяти одной той же структурой, поэтому предикатом EQ можно пользоваться в этом случае. Одинаковые внешне списки (то есть логически равные) физически могут быть представлены разными последовательностями списочных ячеек:

```
(eq '(a b c) '(a b c)) => NIL
(eq 3.1 3.1) => NIL
```

Для сравнения чисел одного типа применяется предикат EQL:

```
(eql 3.1 3.1) => T
```

EQ неприменим для сравнения чисел различных типов (например, 3 и 30)

```
(eql 3 3.0) => NIL
```

Для логического сравнения чисел применяется предикат =:

```
(= 3.14 0.314E1) => T
(= 3 3.0000) => T
```

Предикат, работающий точно так же, как EQL, но проверяющий еще и равенство двух списков, называется EQUAL:

```
(equal '(a b c) '(a b c)) => T
(equal () '(())) => NIL
```

Таким образом, можно говорить, что EQUAL проверяет равенство «записи» аргументов:

```
(equal 3.0 3) => NIL
```

Для наиболее общего логического сравнения объектов различных типов применяется предикат EQUALP. Как правило, если вы не совсем уверены в том, какой из вышеперечисленных предикатов необходимо выбрать, нужно применять этот, хотя он медленней, чем остальные.

Предикаты равенства я рассмотрел столь подробно затем, чтобы читатель получил ясное представление о том, какое большое значение в Lisp уделяется вопросам представления и содержания данных. (Помните, я говорил о кошке как об объекте и о слове «кошка» как о символе, обозначающем кошка? Равны ли «кошка» и кошка?)

Итак, перед вами весь Лисп. Да, да, да, именно на вышеперечисленных функциях можно построить любую программу на Lisp. Безусловно, программирование только с помощью этих функций было бы весьма неудобным, и поэтому Lisp предлагает еще массу других примитивов, в частности nth, выделяющий n-ый элемент списка, list, составляющий список своих аргументов, и т.д.

Определения функций. Формализм их задания

Механизм объявления функций заимствован из лямбда-исчисления Черча, то есть основан на методе представления «безымянных функций» с помощью вычислимых выражений. «Безымянная» функция (или лямбда-выражение) представляет собой совокупность набора формальных переменных и действия с ними.

Пример лямбда-выражения:

```
(lambda (x y) (+ x y))
```

Само по себе лямбда-выражение не воспринимается интерпретатором, а вот лямбда-выражение, примененное к фактическим аргументам, уже вычислимо:

```
((lambda (x y) (+ x y)) 1 2) => 3
```

При вызове лямбда-выражения фактические параметры по определенным правилам связываются с формальными и над ними выполняется заданное действие.

Таким образом, каждый раз, когда нам нужна функция, мы можем записать необходимое лямбда-выражение, примененное к фактическим параметрам. Ясно, что это весьма неудобно (хотя и правильно с «идейной» точки зрения), поэтому Lisp предоставляет возможность именования конкретных лямбда-выражений для их последующего повторного использования.

Функцией в Lisp называется именованное следующим образом лямбда-выражение (defun *имя-функции лямбда-выражение*), например:

```
(defun myplus (x y) (+ x y))
```

Может показаться странным, что я говорю «лямбда-выражение» и при этом нигде не видно ключевого слова lambda. На самом деле, оно здесь подразумевается. Определяя символ как функцию, мы записываем лямбда-выражение в его особое системное свойство (о свойствах речь пойдет чуть ниже), при этом можно считать, что при его записи lambda вставляется.

В некоторых диалектах Lisp возможна такая запись, эквивалентная написанному выше:

```
(setq myplus '(lambda (x y) (+ x y)))   ;; Вот она где прячется!
```

Common Lisp не поддерживает подобной возможности (что, с моей точки зрения, является его существенным недостатком).

Лексические переменные (формальные параметры функций) связаны лишь в пределах той формы, в которой они определены, то есть изменение их значений не влияет на значения одноименных внешних переменных.

Безусловно, в силу рекурсивной природы списков как таковых для определения функций можно (пожалуй, и нужно) использовать рекурсию, хотя возможности Лиспа не исключают и обычные, императивные определения.

На этом этапе можно в качестве примера привести следующее рекурсивное определение функции вычисления факториала:

```
(defun myfactor (x)
(cond ((equal x 1) 1)
(t [AS13](* x (myfactor (- x 1))))))
```

Некоторых программистов раздражает наличие огромного количества круглых скобок в Lisp-коде, но по прошествии некоторого времени к ним очень привыкаешь.

Фундаментальное значение рекурсии состоит в ее применении всюду, где требуются циклические вычисления, поскольку «чистая» функциональная программа не содержит операторов цикла.

По поводу применения рекурсии ведутся большие споры в ста не ученых от computer science, которые, как это часто бывает, делятся на два лагеря — тех, кто считает, что рекурсия необходима везде, и тех, кто считает, что от нее нужно отказаться. Дело в том, что классическое функциональное программирование не использует операторы цикла (так же, как и присваивания), и поэтому для организации итераций рекурсия просто необходима. Конечно, Лисп позволяет использовать циклы, но все же иногда построение чисто рекурсивной программы полезнее хотя бы с точки зрения упражнения.

Кроме обычных функций в Lisp существует также возможность объявления функций более высокого порядка, или функционалов. Функционал — это функция, одним из аргументов которой, в свою очередь, является функция.

В различных случаях функциональный аргумент может использоваться и как данные (в том случае, если он не влияет на вычисления), и как обычная функция (в том случае, если он используется как средство, определяющее вычисления):

```
(car '(lambda (x) (list x))) => lambda ;; CAR - функция, лямбда-выражение - данные;
((lambda (x) (list x)) 'car) => (car) ;; CAR - данные, лямбда-выражение - функция
```

Функционал — это функция, в которой функциональный аргумент используется в позиции и в роли функции.

Кажется, я уже основательно вас запутал, да? Пример пояснит ситуацию. Часто используются так называемые отображающие функционалы MAP (где x — CAR или CDR), применяющие функциональные аргументы последовательно к CAR или CDR списка и составляющие список результатов:

```
(mapcar '(lambda (x) (print x)) '(1 2 3)) => 1\n2\n3\n
(mapcdr '(lambda (x) (print (car x))) '(1 2 3)) => 2\n3\n nil\n
```

Свойства символов

В Lisp каждый символ имеет так называемые свойства. Удобно представлять символы в виде структуры из нескольких значений: непосредственно самого значения символа, а также некоторых присвоенных ему именованных значений, называемых свойствами.

На самом деле свойства представляются списком, хранящимся вместе с символом (список свойств, property list, proplist), следующего вида: (*имя1 значение1 имя2 значение2...*), например, у символа «кошка» может быть такой список свойств:

```
(шерсть густая цвет-шерсти белый цвет-глаз голубой)
```

В Коммон Лиспе получение определенного свойства символа выполняется с помощью функции get:

```
(get 'кошка 'цвет-шерсти) => белый
```

Присваивание же выполняется с помощью объединения обобщенного присваивания setf и get. Причем подобным образом может быть и заведено новое свойство:

```
(setf (get 'кошка 'любимая-еда) 'рыба) => рыба
(get 'кошка 'любимая-еда) => рыба
```

Полный список пользовательских свойств может быть получен с помощью функции symbol-plist:

```
(symbol-plist 'кошка) => (любимая-еда рыба)
```

Каждый символ в Lisp помимо определенных пользователем свойств содержит также список системных свойств, таких, как, например, лямбда-выражение (если с данным символом связано определение функции), тип символа и т. д. Эти свойства изменяются с помощью описанных выше функций set, defun и пр.

Таким образом, является ли данный символ функцией или нет, принадлежит ли он тому или иному типу и т. п., является свойством самого символа, что позволяет легко манипулировать этими свойствами (попробуйте в C убрать у main()свойство быть функцией).

С моей точки зрения, свойства — это великолепная возможность Lisp, позволяющая заметно облегчить написание программ, управляемых данными, семантических сетей и, конечно, объектно-ориентированных программ.

Прочее

Практически любой язык функционального программирования содержит так называемый сборщик мусора, то есть механизм освобождения памяти от неиспользуемых объектов. Алгоритмам работы «сборщиков» посвящены целые тома и многие научные работы, поэтому на них мы останавливаться не будем. Скажу лишь, что, к сожалению, большинство нынешних языков императивного программирования не содержат этого механизма (помните, наверное, жуткие мучения с распределением памяти на C?), хотя в этой области наметилась положительная тенденция (Perl и Java, например, содержат сборщики мусора).

И еще немного об очень сильно пропагандируемом механизме обработки исключений. Кроме классического механизма catch/throw Lisp предлагает возможность выхода по исключению из любого места статического именованного блока (почти аналогично действию break в C) и макрос unwind-protect, который позволяет при любом исключении выполнять определенное действие. Что любопытно: при использовании catch/throw контекст выхода определяется динамически, то есть имеется возможность использовать throw вне пределов объявления catch.

. . . Да, наверное, следует остановиться. В Лиспе содержится невообразимое количество возможностей, и описать их все в узких рамках журнальной статьи просто нельзя.

Безусловно, Lisp не лишен некоторых недостатков. Во-первых, переход к более высокому уровню абстракции в программировании осуществляется ценой снижения (иногда значительного) скорости исполнения программ. Еще одно заметное влияние на скорость оказывает то, что Lisp все же является интерпретируемым, а не компилируемым языком. Хотя практически любой из достаточно развитых трансляторов имеет возможность компилирования в байт-код с последующей его записью, что несколько ускоряет процесс исполнения. Некоторые интерпретаторы, такие, как, например, GNU Common Lisp, могут записывать и исполняемый код (с помощью достаточно остроумного механизма замены частей Lisp-кода на C-код).

Во-вторых, существенным недостатком стоит считать огромное число различных диалектов Lisp. Дело в том, что возможность легкого расширения Lisp сыграла с ним злую шутку. На сегодняшний день существует около 90 различных (совместимых, безусловно, на программах, использующих «чистый» Lisp, но иногда отличающихся поведением даже в рамках и этих функций) вариантов Lisp, включая достаточно отдельные, но все же происшедшие из него языки, такие как Scheme. Наиболее известными можно считать диалекты MacLisp, FranzLisp и ZetaLisp. Последний применялся, кстати, на «машинах искусственного интеллекта» — лисп-машинах, то есть в компьютерах, программное обеспечение которых было практически полностью написано на Lisp, предназначенных для разработки программ на Lisp.

Определенные надежды внушает принятый в 1984 году стандарт Common Lisp, разработанный Гаем Стилом. Теперь можно писать на Common Lisp, не слишком заботясь о совместимости с иными диалектами, — но все же следует быть очень осторожным, поскольку даже интерпретаторы Common Lisp не всегда ведут себя одинаково в тех или иных ситуациях.

Не следует считать, что Lisp — это что-то вроде игрушки седовласых ученых-теоретиков от computer science. На самом деле, на нем сейчас написана масса достаточно критичных приложений, таких как управление спутниками (в частности, например, Hubble Space Telescope), программное обеспечение технологических компьютеров атомных электростанций. Есть написанный на Allegro Common Lisp великолепный web-сервер CL-HTTPD, работающий, кстати, на серверах в Белом доме. Если же говорить о «несерьезных» приложениях, то следует отметить, что почти все игры для приставок Nintendo пишутся на Lisp (кстати, достаточно известная игра Abuse тоже написана на нем). Нельзя также забывать об AutoCAD, Mathematica (использующей функциональный язык, похожий на Lisp) и, конечно, о любимом мною Emacs.

В качестве заключения я бы очень советовал читателю отбросить прочь все то, что он изучал про программирование (на Бейсике в школе и на Паскале — в институте), и войти в мир Лиспа не испорченным ничем — только тогда можно понять всю красоту и богатство Лиспа. Я уверен — вам никогда не захочется писать на чем-либо еще.

Вартан Хачатуров, wart@softhome.net
журнал Byte/Россия №3 2000