Знакомство с Common Lisp

Лектор: Артамонов Юрий Николаевич

Университет "Дубна"
филиал Котельники

Общий концептуальный подход

На самом общем уровне любой язык программирования должен иметь три механизма:

- Элементарные выражения - это минимальные сущности, с которыми мы можем работать;
- Средства комбинирования - это средства, позволяющие из простых сущностей строить более сложные;
- Средства абстракции - это средства, с помощью которых сложные сущности можно называть и обращаться с ними как с единым целым, т.е. начинать считать их простыми.

Элементарные выражения (символьные выражения) Common Lisp

В Common Lisp используются символы. Символ - это имя, состоящее из букв, цифр и специальных знаков, которое обозначает некоторую сущность. Символы преобразуются к верхнему регистру независимо от того, как вы их ввели:

>'Hello

HELLO

В данном случае мы ввели символ «Hello». Символы, как правило не являются самовычисляемыми, поэтому, чтобы сослаться на символ, его необходимо цитировать - мы поставили перед ним знак '. Существует более полная форма такого цитирования:

>(quote Hello)

HELLO

Запись (quote Hello) на самом деле - это список, в котором на вход специальной функции quote, призванной блокировать вычисления, подали символ Hello.

Элементарные выражения (символьные выражения) Common Lisp

Таким образом, все, что мы вводим в интерпретатор будет вычислено, и так в бесконечном цикле, что называется RELP (read-eval-print-loop). Если попытаться ввести символ Hello без блокировки вычислений, то возникнет ошибка, т.к. у Hello нет никакого значения. Эту ситуацию можно исправить:
>(setf Hello 'Yes)
Отныне у символа HELLO будет значение YES, и его можно вычислить:
>Hello
YES

Элементарные выражения Common Lisp

На самом деле это механизм присваивания переменным значения, но об этом мы на время изучения функционального подхода забудем. Можно поступить и иначе:
>(defun Hello ())
Этим действием мы определили функцию с именем Hello, однако сама функция довольно странная - она ничего не вычисляет и работает так:
>(hello)
NIL - Её вычисление вернуло пустой список. А что она могла вернут ещё?

Элементарные выражения Common Lisp

Кроме символов, к элементарным выражениям относятся числа. Числа - самовычисляемый объект:

```
>777
777
>666
666
```

Самовычисляемыми являются также два специальных символа: T - истина; NIL - ложь (пустой список и ложь в Common Lisp - это одно и тоже).

```
>T
T
>Nil
NIL
>()
NIL
```

Элементарные выражения Common Lisp

Символы и числа в Common Lisp называются атомами. Кроме атомов, как было указано в Common Lisp используются списки. Причем списки являются основной структурой данных в Common Lisp и выступают средством объединения программ и данных. Любой список для Common Lisp - это указание что-то вычислить, любая программа - это список.

Список - это всё, что угодно, записанное в круглых скобках через пробелы:

(1 2 3) - список из чисел

(1 2 a s d (1 2 3)) - список из чисел, символов и списка

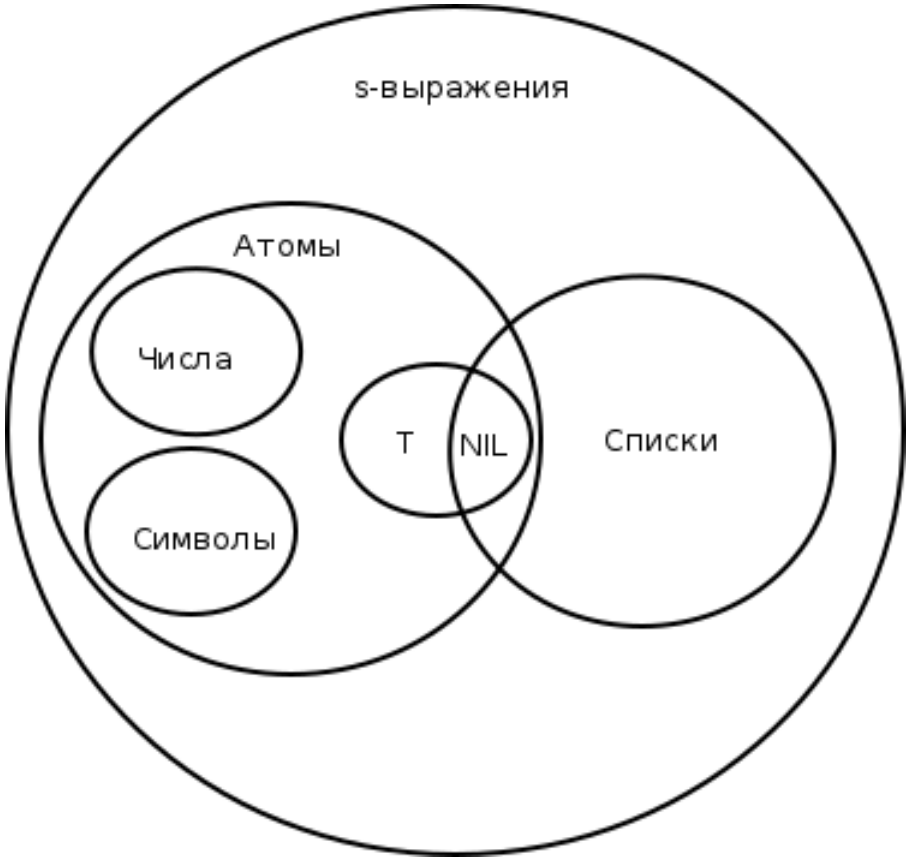
NIL - пустой список

() - пустой список

(NIL ()) - список из двух пустых списков

Атомы и списки называются символьными выражениями (s-выражениями).

Элементарные выражения Common Lisp



Пустой список - единственный является одновременно атомом и списком.

Средства комбинирования Common Lisp

Поскольку списки являются основной структурой данных в Common Lisp, для них есть примитивы - селекторы - позволяют разбирать сущность на части, конструкторы - позволяют соединять части и получать сущность, предикаты - позволяют проверять принадлежность сущности к заданному классу сущностей.

Селекторы для списков:

- Функция CAR - отделяет голову от списка
- Функция CDR - отделяет хвост от списка

```
>(car '(1 2 3))
1
>(cdr '(1 2 3))
(2 3)
```

Существует договоренность использования составной конструкции

```
>(cadr '(1 2 3))
```

2 - сначала берем хвост, потом от него голову

>(caddr '(1 2 3))
3
(cdar '(1 2 3 4)) - ошибка, т.к. получим атом 1, а от него хвост уже взять нельзя.
Конструктор списков - функция cons:
>(cons 0 '(1 2 3))
(0 1 2 3)
>(cons '(1 2 3) '(4 5 6))
((1 2 3) 4 5 6)
Еще один конструктор, без которого можно обойтись уже существующими средствами, но для которого удобно создать отдельную абстракцию (синтаксичекий сахар):
>(append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
Соединяет списки в один.

Средства комбинирования Common Lisp

Еще один сахарный элемент)))
>(list '(1 2 3) '(4 5 6))
((1 2 3) (4 5 6))
>(list 1 2)
(1 2)
Предикаты в Common Lisp:

- Atom - проверяет, является ли сущность атомом
- Null - проверяет, является ли список пустым
- Eq - проверяет тождественность двух символов
- Eql - сравнивает числа одинаковых типов
- Equal - проверяет идентичность записей
- numberp - проверяет, является ли сущность числом
- listp - проверяет, является ли сущность списком
- symbolp - проверяет, является ли сущность символом

Средства комбинирования Common Lisp

```
>(atom nil)
T
>(atom '(1 2 3))
NIL
>(Null '(1 2 3))
NIL
>(Null ())
T
>(eq nil ())
T
>(eq 'a 'b)
NIL
>(eq T (atom 'bed))
T
```

Средства комбинирования Common Lisp

```
>(eql 3 4)
NIL
>(eql 3 3)
T
>(eql 4 4.0)
NIL
>(equal '(1 2 3) '(1 2 3))
T
>(equal Nil '(nil))
NIL
>(nubmerp 6)
T
```

Средства комбинирования Common Lisp

```
>(numberp '(1 2 3))
NIL
>(listp '(1 2 3))
T
>(listp 9)
NIL
>(symbolp 'p)
T
>(symbolp '(1 2 3))
NIL
```

Средства абстракции Common Lisp

С одним средством абстракции мы уже познакомились - можно вводить новые функции - называть длинную последовательность действий одним именем и многократно ее использовать:

```
>(defun f (x) (+ x 1))
F
>(f 4)
5
```

Еще одним средством, позволяющим строить новые абстракции, является условная конструкция:

(if <condition> (eval-if-condition-t) (eval-if-condition-NIL))

```
>(if (not (atom T)) (+ 6 7) 'Error)
ERROR
>(if (numberp 888) (+ 6 7) 'Error)
13
```

Более расширенным вариантом условной конструкции является Cond:

```
(cond
(<condition1> (eval-if-condition1))
(<condition2> (eval-if-condition2))
...
(t (eval-else)))
```

Используя условную конструкцию и идею рекурсивного вызова функций со списками можно делать все, что угодно!

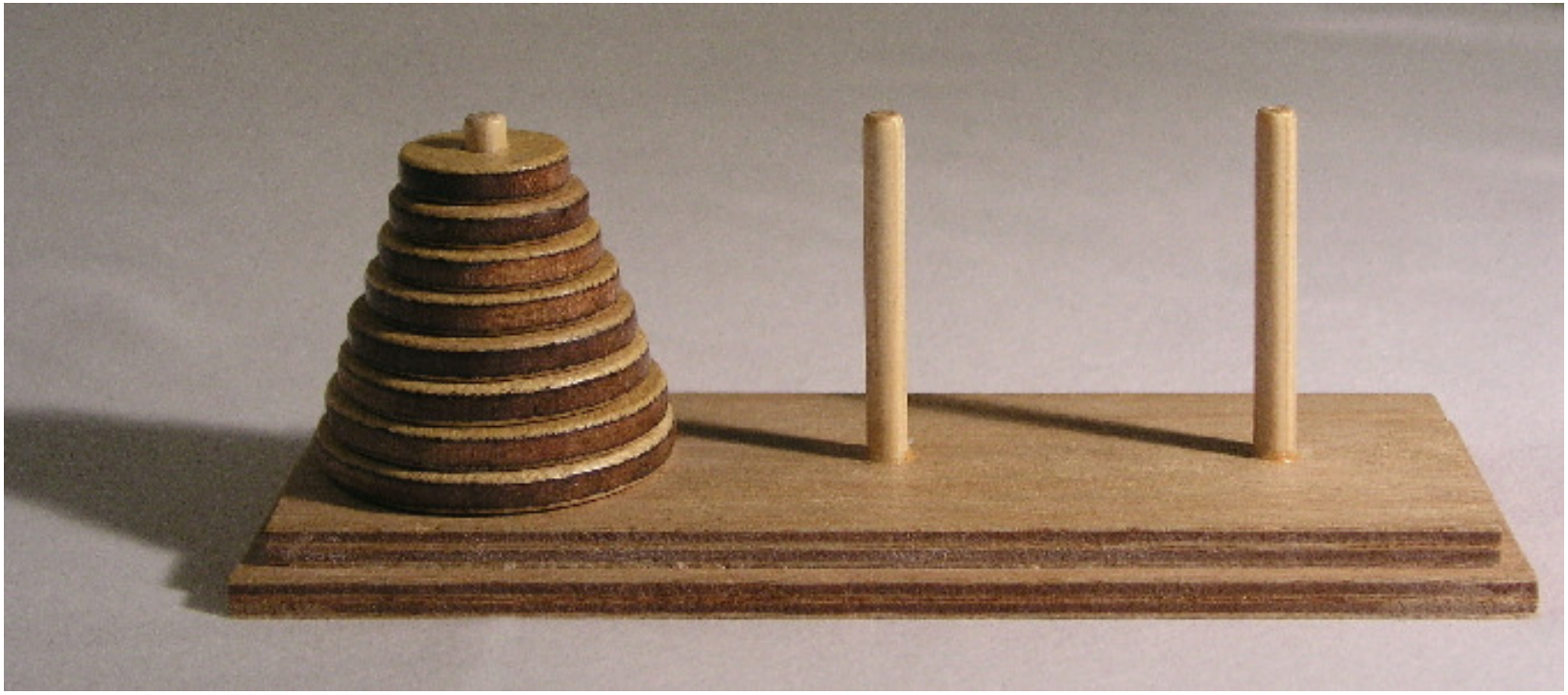
Докажем это на практике.

Рекурсия

Функция является рекурсивной, если в ее определении содержится вызов этой функции. На рекурсии основана декомпозиция задачи на подзадачи, решение которых, насколько это возможно, пытаются свести к уже решенным или к решаемой в настоящий момент задаче. Рекурсия близка к аппарату математической индукции, что определяет удобство реализации индуктивных определений. Как правило, рекурсивная функция реализует разбор случаев, некоторые из них влекут продолжение процесса вычисления через вызов рекурсивной функции, другие не прибегают к рекурсии, они называются базис рекурсии. Феномен рекурсии довольно сложен. И иногда он существенно помогает в решении задач.

Ханойская башня

Для примера рассмотрим задачу о ханойской башне.



Ханойская башня (продолжение)

Перенести пирамиду из восьми колец. За один раз разрешается переносить только одно кольцо, причём нельзя класть большее кольцо на меньшее.

Предположим, что мы уже научились перекладывать n-1 колец, тогда алгоритм перекладывания n колец такой:

- перекладываем на средний колышек n-1 колец
- перекладываем на крайний правый колышек самое большо кольцо
- перекладываем на крайний правый колышек n-1 колец
- Последовательно уменьшая n мы приходим к одному кольцу, алгоритм перекладывания которого очевиден. После чего легко восстанавливается весь алгоритм.

Вычисление факториала

Следующая задача - вычисление факториала :

```
(defun factorial (n)
  (if (= n 1) 1 (* n (factorial (- n 1)))))
```

Вычисление факториала

При вычислении используется так называемая подстановочная модель:

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Виды рекурсии

Выделяют следующие способы организации рекурсии:

- Простая рекурсия
- Параллельная рекурсия
- Взаимная рекурсия
- Рекурсия высоких порядков

Простая рекурсия

Общий вид простой рекурсии:
(defun f (...) (cond ((...) (...)) ((...)(...(f ...) ...)) ((...)(...))))
Говорят о простой рекурсии, если вызов функции встречается в некоторой ветви лишь один раз. Простая рекурсия часто эквивалентна циклу.
Определение функции factorial является простой рекурсией.

Простая рекурсия - другой пример

```
(defun append (X Y)
  (if (Null X) Y (cons (car X) (append (cdr X) Y))))
```

Параллельная рекурсия

Общий вид параллельной рекурсии:
(defun f ... (g ... (f ...) ... (f ...) ...) ...)
Таким образом, при параллельной рекурсии тело определения функции f содержит вызов некоторой функции g, несколько аргументов, которой является рекурсивными вызовами функции f. Пример такой функции — вычисление чисел Фибоначчи.
Общее правило для чисел Фибоначчи можно сформулировать так:
если $n = 0$, то $Fib(n) = 0$
если $n = 1$, то $Fib(n) = 1$
в остальных случаях $Fib(n)=Fib(n-1) + Fib(n-2)$
Можно немедленно преобразовать это определение в процедуру Лисп:

Параллельная рекурсия - числа Фибоначчи

```
(defun fib (n)
  (cond
    ((= n 0) 0)
    ((= n 1) 1)
    (t (+ (fib (- n 1)) (fib (- n 2))))))
```

Взаимная рекурсия

Общий вид взаимной рекурсии:
(defun f ... (g ...)...)
(defun g ... (f ...)...)
О взаимной рекурсии говорят, если определение функции f содержит вызов некоторой другой функции g, которая в свою очередь содержит вызов функции f. Ясно, что в общем случае, количество функций, вовлеченных в организацию взаимной рекурсии, может быть больше, чем две.
Для примера взаимной рекурсии реализуем функцию turn зеркального отображения списка и всех вложенных в него подсписков.

Взаимная рекурсия - функция turn

```
(defun turn (L)
  (cond
    ((Atom L) L)
    (t (permulate L nil))))

(defun permulate (L Result)
  (cond
    ((Null L) result)
    (t (permulate (cdr L) (cons (turn (car L)) result)))
```

Рекурсия высоких порядков

Общий вид рекурсии высших порядков:
(defun f ... (f ...(f ...)...) ...)
В качестве классического примера рекурсии более высокого порядка часто приводится известная из теории рекурсивных функций функция Аккермана,пользующаяся славой "плохой"функции.

Рекурсия высоких порядков- функция Аккермана

```
(defun akkerman (m n)
  (cond
    ((= m 0) (+ n 1))
    ((= n 0) ( akkerman (- m 1) 1))
    (t ( akkerman (- m 1) (akkerman m (- n 1))))))
```

```
>(аккерман 2 2)
7
```

Вычисление функции Аккермана довольно сложно, и время вычисления растет лавинообразно уже при малых значениях аргумента.