

Содержание

1	Лекция 1	2
2	Лекция 2	3
	2.1 Представление атома в памяти	3
	2.2 Классификация функций	3
	2.3 Именованные функции	4
	2.4 Классификация по операциям работы со списку	4
	2.5 Вычисление функций	4
3	Лекция 3	6
	3.1 Функции	6
	3.2 Специальные функции	6
	3.3 Функции, реализующие операции со списками	7
	3.4 Функционалы	8
	3.5 Среда	9
4	Лекция 4	10
	4.1 Функции работы со списками-	10
5	Лекция 5	12
	5.1 Рекурсия	12
6	Лекция 6	14
	6.1 Рекурсия	14
7	Лекция 6	16
	7.1 Предпосылки появления Prolog	16
	7.2 Prolog	16

1 Лекция 1

– Lisp - безтиповый язык. Атомы - символы, которые могут обозначать любые объекты. Само-вычислительные атомы - T, Nil, числа, строки.

Точечная пара - (A.B):

$\square\square \rightarrow cdr$

↓

car

Пустая конструкция - () - Nil.

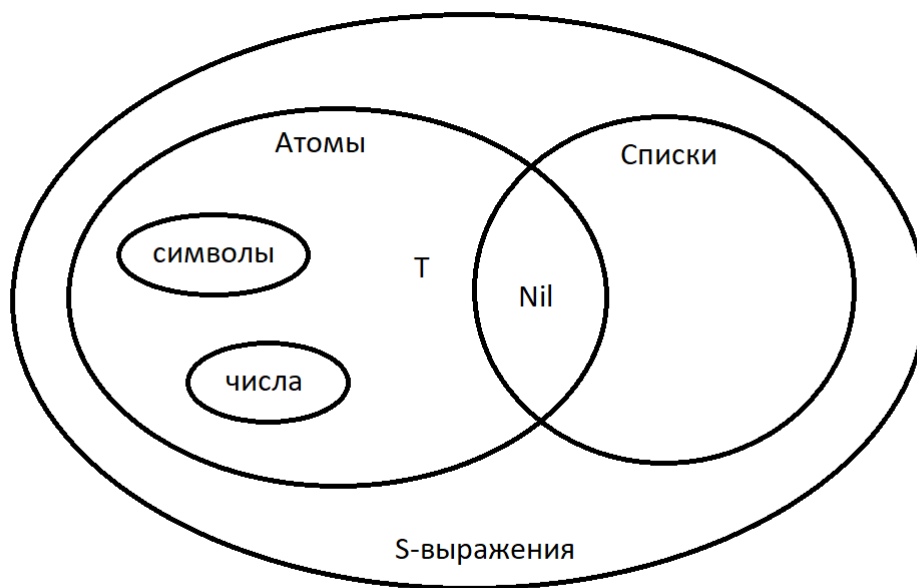


Рисунок 2.1 — Виды структур в lisp

2.1 Представление атома в памяти

Представление атома в памяти - атом представляется в памяти в виде структуры, содержащей пять указателей:

- name - символьный идентификатор атома
- value - самоопределяемое значение атома
- function - лямбда-выражение
- properties - список свойств
- package - пакет, указатель на начало области, связывающий значение атома и пространство имён

2.2 Классификация функций

- чистые функции (математические)
- формы (специальные функции), могут принимать различное число аргументов
- псевдо-функции (вывод на экран)
- функции с вариантами значений
- функционалы - принимают функцию в качестве параметров, либо возвращаемым значением является функция
- базисные функции - car, cdr, cons, atom, cond, quote, eval, lambda, apply, funcall

Выяснить атом - перейти по указателю.

Лямбда-выражение - (lambda (параметры) (тело)).

Вызов: (лямбда-выражение аргументы)

2.3 Именованные функции

Синтаксис:

(defun имя лямбда-выражение)

(defun f (x_1, \dots, x_k) форма)

Лямбда-определение может оказаться более эффективным. Пример лямбда-определения:

$(let(x_1p_1)(x_2p_2)\dots(x_kp_k)e) = ((lambda(x_1, x_2, \dots, x_k)e)p_1p_2\dots p_k)$

2.4 Классификация по операциям работы со списку

1. Селекторы - car, cdr

2. Функции-конструкторы - cons, list

3. Функции-предикаты - atom, null, listp, consp, eql (eql сравнивает атомы и числа одного типа, применим только к числам) equal (как eql и списки), equalp (equal + "-").

2.5 Вычисление функций

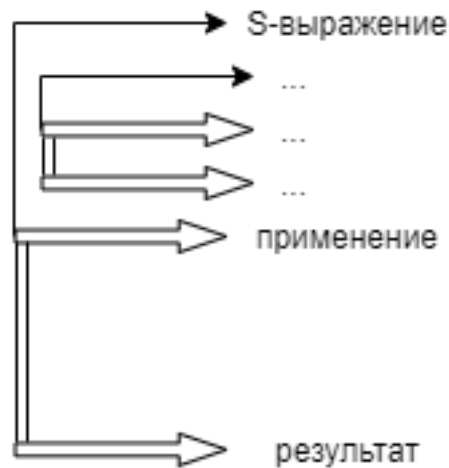


Рисунок 2.2 — Пример построения схемы вычислений

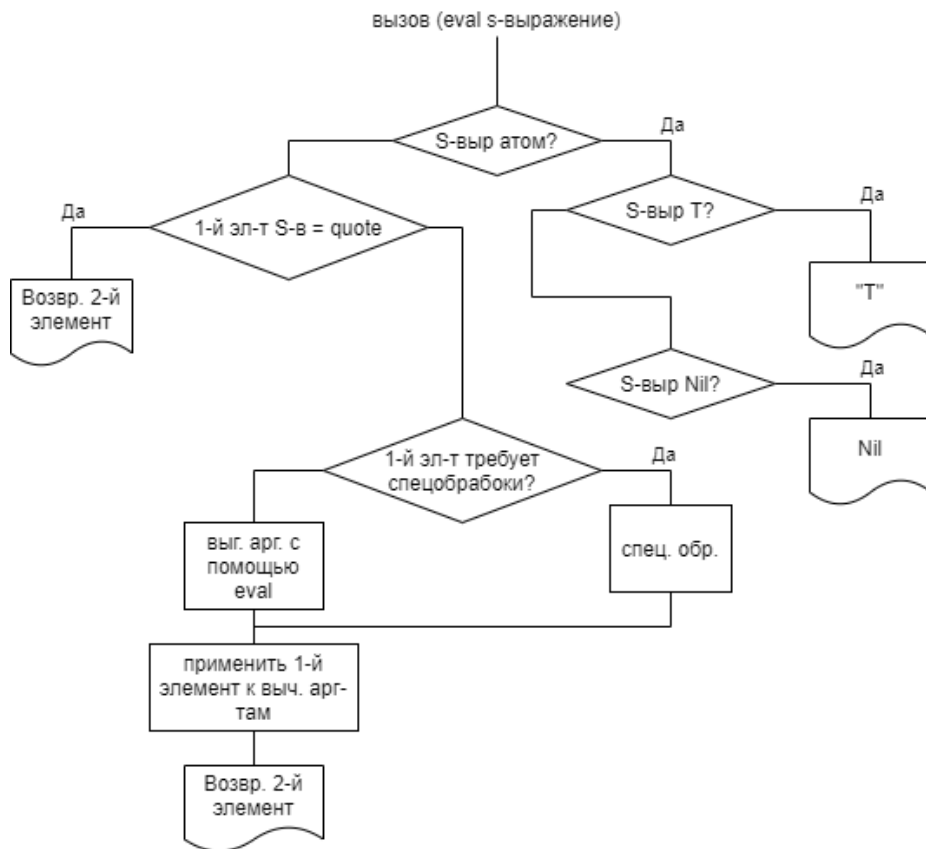


Рисунок 2.3 — Пример построения блок-схемы работы функции

3 Лекция 3

Как функция отличает символы от функций? Как система запоминает символы и на какое время? Придумать метод реализации учитывая, что Lisp реализуется с помощью указателей.

3.1 Функции

Мн-во всех функций можно классифицировать:

1. как базисные и небазисные
2. по методу написания:
 - чистые функции имеют фиксированное количество аргументов и возвращают результат
 - формы - функция, которая вычисляет не все свои аргументы
 - функционалы - принимают на вход функцию или возвращают функцию в качестве результата

3.2 Специальные функции

- базисная функция `(cond (test_1 body_1) (test_2 body_2) ... (test_i body_i) [(t body_i+1)])`, `test_i` - тестирующееся выражение, `body_i` - выполняется, если `test_i` корректный. - `if` - более удобная реализация `cond`, `(if test then else)/(if test then)` - если нет `else`, то автоматически возвращается `nil`

Диаграммы во второй лабораторной. При вызове своей функции указываются фактические параметры, в определении функции присутствуют формальные параметры. Если формальный параметр получает какое-то значение, то к этому значению необходимо иметь доступ, поэтому этому символу должна быть выделена память. Для каждого атома - 5 указателей. В процессе работы тела указатели могут переставляться и могут меняться значения, на которые указывают эти указатели. И поэтому на каждом шаге использования этого параметра система вынуждена каждый раз вычислять значение этого атома.

Формальные параметры существуют только во время работы функции -> в рекурсивной функции есть опасность потерять значение.

- логические выражения (формы `and/or`, `not`, `null`)

Символьному атому можно установить значение. Есть локальные атомы - существуют в функции, есть глобальные атомы - существуют на протяжении всей работы программы. Можно установить значение символьному атому до момента, пока это значение не поменяется. Установка происходит с помощью функций `setf` и `setq`. Эти функции имеют два аргумента, первый не вычисляется, второй вычисляется и является значением первого аргумента. Разница между `setf` и `setq` - вычисляются оба аргумента или один.

Если в тексте программе используется установленный уникальный атом, то система использует значение установленного ранее атома. Список свойств - динамическая структура, состоящая из списков вида (имя свойства, свойство).

Функция `let` - является формой, имеет два аргумента (`let ((name1 val1) (name2 val2) ... (namen valn)) body`) - `namen` - имя указателя, `valn` - значение указателя `namen`, `body` - выражение, где можно использовать указатели `name1` - `namen`.

Работа `let` - сначала выделяется память, потом готовятся значения, после чего происходит связывание в произвольном порядке. При этом в `name2` нельзя использовать `name1`, поскольку ещё не произошло связывание, поэтому при вычислении очередного значения `name1` нельзя сослаться на значение предыдущих атомов.

`let*` - позволяет сослаться на значения предыдущих атомов, но работает менее эффективно.

3.3 Функции, реализующие операции со списками

Данные функции можно разбить на две группы - разрушающие структуру и не разрушающие структуру.

Для работы со списком его необходимо создать, получить доступ и модифицировать.

Функции, разрушающие структуру - изменяют структуру списка. Функции не разрушающую структуру - производят какие-то операции без изменения поданного на вход списка.

Функция `append` - функция объединяет списки. Количество аргументов может быть произвольным. Функция `append` работает с копией списка -> не разрушается структура списка, поданного на вход.

```
1 (setf lst1 '(a b))
2 (setf lst2 '(c d))
3
4 lst1 -> [][] -> [][] -> nil
5         |         |
6         a         b
7
8 lst1 -> [][] -> [][] -> nil
9         |         |
10        c         d
11
12 (setf lst3 (append lst1 lst2))
13
14 lst3 -> [][] -> [][] -> [][] -> [][] -> nil
15         |         |         |         |
16        a         b         c         d
```

Создаются копии списков, кроме последнего. После чего копии сцепляются последними указателями, последний прицепляется к `lst2`. В результате можно работать со списками `lst1` и `lst3`, однако `lst2` одновременно входит в `lst3`, что приводит к тому, что его изменении происходит и изменение `lst3`.

`cons` - дубль `append`, `cons` переставляет указатели, в результате чего списки сшиваются без создания дублей

`reverse` - развернуть с копиями `nreverse` - развернуть без копий

last - на вход подаётся структура, состоящая из списковых ячеек, возвращает последнюю списковую ячейку.

Список может быть смешанным - числа и символы могут быть в списке. Может быть одноуровневым и многоуровневым. Структурированный/неструктурированный.

Список числовой - только из чисел.

<!> Все стандартные функции работают только с верхним уровнем списка.

(nth N lst) - n-й элемент. (nthcdr N lst) - n-й хвост. (remove elem lst) - удаление элемента, не обрабатывает внутренности списка.

В стандартных функциях для сравнения использует функцию equal, принимающую на вход только атомы, не списки.

member - проверка присутствия элемента в списке. (member elem lst) - по верхнему уровню ячеек, используется функция eql, не сравнивается список. Пример (member '(a, b) '(c (a b) d)) - в случае подачи, система не обнаружит, поскольку eql не сравнивает списки. Можно повлиять на работу функций с помощью механизма ключевых слов. При создании функции используется набор ключевых параметров, которые могут быть изменены для изменения поведения функции. В данном случае изменение ключевого параметра test позволит включить сравнение списков - (member '(a, b) '(c (a b) d): test #'equal). Результат работы будет ((a b) d).

Для возврата списочных ячеек по элементу - assoc и rassoc.

Работа со множествами - работа со множествами, представленными в виде списком. Существуют стандартные функции для работы с множествами.

Ассоциативные таблицы - самый простой вариант реализации - список точечных пар, где в каждой паре точечная пара состоит из ключа и значения. Для работы с ассоциативными таблицами существует несколько стандартных функций.

3.4 Функционалы

Какие типы алгоритмов существуют? - линейный, разветвлённый, циклический.

Для организации повторных вычислений используются либо функционалы, либо рекурсии.

Функции применяющие и отображающие.

Значения глобальных атомов могут влиять на работу функции, поэтому необходимо его заблокировать. # - функциональная блокировка, фиксирует значения глобальных атомов (фиксирует состояние в памяти окружения данной функции). # - function - функция, вызывающая функциональную блокировку.

Функция apply - применяет лямбда-функцию к какому-либо выражению (apply #'fun lst), fun - функция или лямбда-выражение, lst - список аргументов. (funcall #'fun arg1 ... argn) - применение формы, аналогично apply но без списка.

Отображающие функционалы - функционалы, позволяющие реализовать многократные или повторные вычисления. Результаты организуются в виде списка результатов для каждого вызова функции.

Основные отображающие функционалы: - (`mapcar #'fun lst1`) - функция, указанная `fun`, применяется к каждому элементу списка из верхнего уровня ячеек, элементы могут быть как указателями так и списками, поэтому необходимо учитывать это при написании функции. - (`maplist #'fun lst2`) - функция применяется целиком к списку, к хвосту списка, к хвосту хвоста... пока список не станет пустым. В данной форме функция `fun` должна быть одноаргументная, в функции `maplist` - функция `fun` должна уметь работать со списком.

3.5 Среда

Термин "среда" употребим, когда используется какой-то пакет. При загрузке пакета становится доступен интерфейс среды - набор возможностей, облегчающих программирование.

4 Лекция 4

4.1 Функции работы со списками-

Вызов функции `mapcar` для функции нескольких аргументов: `(mapcar #'fun lst1 lst2 ... lstk)`. В данном случае `mapcar` выбирает из каждого списка `car` аргументы и применяет их к функции. `mapcar` не контролирует длину списков, поданных на вход. Если списки имеют разную длину, когда заканчиваются элементы по верхнему уровню самого короткого списка.

`(maplist #'fun lst1 lst2 ... lstk)`, функция `fun` должна работать со списками.

При работе данных функций образуются несколько результатов, которые объединяются в список с помощью `list`. Существуют дубли, где объединение происходит с помощью функции `pconc` - `mapcar`, `mapcon`.

`(find-if #'fun lst)` - функционал `find-if` проходит только по верхнему уровню списковых ячеек, возвращает первый элемент списка, удовлетворяющий данной функции (`#'fun = #'predicat`). Пример: `(find-if #'odd '(2 4 7 5))` → 7. `(find-if-not #'predicat lst)` - первый элемент, `fun` от которого возвращает `Nil`.

`(remove-if #'predicat lst)/(remove-if-not #'predicat lst)` - удаление элемента с условием.

Если определение предиката написано прямо в функции, то при выполнении программы не происходит множественного перехода по указателю от имени к определению функции, поэтому данный вариант предпочтительнее, так как работает быстрее.

`(reduce #'fun lst)` - каскадная функция, `fun` должна иметь не менее двух элементов. Сначала применяет к первым двум элементам, потом к результату и третьему, потом к результату и четвёртому.

`(every #'fun lst)`, `(some #'fun lst)` также работают каскадным образом.

Примеры применения функционалов:

```
1 (defun consist-of (lst)
2   (if (member (car lst) (cdr lst)) 1 0))
3
4 (defun all-last-element (lst)
5   (if (eql (consist-of lst) 0) (lst (car lst)) ()))
6
7 (defun collection-to-set (lst)
8   (mapcon \#'all-last-element lst))
9
10 (collection-to-set '(i t i g t k s i f k))
11 > (g t s i f k)
```

Смысл примера - порядок результата может зависеть от порядка аргументов в исходном списке.

```
1 (defun dcart (lstx, lsty)
2   (mapcar #'(lambda (x)
```

```
3      (mapcar #'(lambda (x) (list x y)) lsty))
4          lstx))
5
6 (decart '(a b) '(1 2))
7 > ((a 1)(a 2)(b 1)(b 2))
```

Смысл примера - применение лямбда-функций и использование вложенного вызова функционала с фиксацией аргумента x.

5 Лекция 5

5.1 Рекурсия

Рекурсия - повторный вызов некой функции этой функци. Рекурсия - ссылка на некий объект в описании этого объекта. Основные вопросы при создании рекурсии - как войти, как выйти, как передать аргументы.

Классификация рекурсий в Lisp:

- Простая рекурсия - один вызов в теле функции
- Рекурсия первого подярка - когда в теле функции вызов функции производится несколько раз
- Взаимная рекурсия - в теле функции вызывается несколько разных рекурсивных функций

Первые ветки рекурсии реализуются cond - базисный случай. Золотое правило создание рекурсии - сначала проверка, нужно ли выйти из рекурсии и только потом уход на следующий шаг.

Для эффективной реализации рекурсии необходимо заранее выполнить отложенные вычисления и только затем уходить в рекурсию для того, чтобы вернуть только промежуточный результат.

Хвостовая рекурсия - один из способов эффективной организации рекурсии.

lisp, append, cons - какую из этих функций нужно использовать, чтобы реализовать рекурсию эффективно (точно не append, поскольку append делает копии).

"Хорошая" функция - эффективность с точки зрения данных и реализации.

```
1 (defun my-number (el lst)
2   (cond ((null lst) Nil)
3         ((equal el (car lst)) t)
4         (t (my-member el (cdr lst) )))))
5
6 (my-member 'a (b a c))
7 > t
8 (my-member nil ())
9 > nil
```

В последнем вызове вывод не совсем корректный из-за порядка следования проверок cond. Переставив их местами мы можем добиться правильного вывода.

```
1 (defun my-number (el lst)
2   ((equal el (car lst)) t)
3   (cond ((null lst) Nil)
4         (t (my-member el (cdr lst) )))))
5
6 (my-member nil ())
7 > t
```

Реализация собственной функции reverse с помощью append - неэффективный способ. Данная рекурсия нехвостовая, так как результат не вычисляется на входе.

```
1 (defun my-reverse (lst)
2   (cond ((null lst) lst)
3         (t (append (my-reverse (cdr lst))
4                     (cons (car lst) lst)))))
```

Реализация собственной функции reverse с помощью своей функции - эффективный способ.

```
1 (defun my-reverse (lst)
2   ...)
3
4 (defun move-to (lst result)
5   (cond ((null lst) result)
6         (t (move-to (cdr lst) (cons (car lst) result)))))
```

6 Лекция 6

6.1 Рекурсия

Эффективный вариант реализации рекурсии - при входе не остаётся невыполненных команд, т.е. рекурсивный вызов последний.

Количество веток реализации рекурсии зависит от конкретной задачи.

Может быть несколько выходов из рекурсии и несколько входов в рекурсию.

Общий вид дополняемой рекурсии

```
1 (defun fn (x)
2   (cond (cnd-test end-value)
3         (t (cons add_val(fn changed_x))))))
```

Внутри дополняемых рекурсий существует вариант дополняемой рекурсии, имеющей преобразование с условием.

```
1 (defun fn (x)
2   (cond (cnd-test end-value)
3         (add-test add-func (fn chaged1_x))
4         (t (fn changed2_x))))
```

Пример:

```
1 (defun extract_symbols (lst)
2   (cond ((null lst) nil)
3         ((symbolp (car lst))
4          (cons (car lst)
5                (extract-symbols (cdr lst))))))
```

Множественная рекурсия - может быть дополнительная функция, обрабатывающая два рекурсивных вызова, по голове и хвосту.

Пример:

```
1 (defun cons_sells (lst)
2   (if (atom lst) 0
3       (+ (length lst)
4          (reduce #'+
5                  (mapcar #'cons_sells lst))))))
```

Пример:

```
1 (defun into_one_level (lst rst)
2   (cond ((null lst) rst)
3         ((atom lst) (cons lst rst))
4         (t (into_one_level (car lst)
5                             (into_one_level (cdr lst) rst)))))
```

boundp - связь атома со значением, fboundp - связь атома с функционалом. Свойства - упорядоченный список чётного количества элементов. putprop - назначение свойства. remprop, symb-plist.

Механизм ключевых слов:

`&optional` - не указанный параметр, если не введён, то заменяется на `nil`.

`&rest` Пример:

```
1 (defun f1 (x &optional y) (list x y))
```

7 Лекция 6

7.1 Предпосылки появления Prolog

Процесс совершенствования техники застопорился из-за отсутствия новой элементной базы. Пролог - реализация логики предикатов в виде языка программирования. При доказательстве теорем не используются данные в привычном смысле слова.

Пролог работает со знаниями.

Основной элемент математической логики - высказывание, которое может быть либо истинно либо ложно.

Задача пролога - автоматизировать процесс доказательства теорем. Для решения данной задачи необходимо было доказать принцип резолюции. Данный принцип был доказан в 1969 году. Позволяет сделать обоснованный вывод из утверждения.

Пролог поддерживает декларативный способ программирования.

Всё что нужно для функционирования искусственного интеллекта - обладание знаниями.

7.2 Prolog

Программа на прологе представляет из себя базу знаний. Данная база состоит из аксиом (фактов) и теорем. База знаний фиксируется в разделе, имеющем заголовок `clauses` - предложения. Активизация производится в разделе `go` (цель).

Единственная конструкция пролога - `term`, либо константа, либо переменная, либо составной терм. Логическая система возвращает либо да либо нет и попутно информация о том, как прийти для нужного результата.

Основная единица - символьный атом, использующегося для обозначения предикатов. С маленькой буквы - константа, с большой буквы - переменная.

Строковые константы - в кавычках. Именованные переменные - переменные использующие имена. Prolog использует анонимные переменные, обозначающиеся как `_`.

Составные термы используются для того, чтобы зафиксировать, что между какими-то объектами есть связь. $F(t_1, t_2, \dots, t_n)$, F - главный функтор, t_1-t_n - термы.

Пример:

`student(ivanov, mgtu)`. - конкретный студент конкретного вуза

`student(X, mgtu)`. - все студенты конкретного вуза

`student(X, Y)`. - все студенты всех вузов

Все предложения заканчиваются точкой. Количество аргументов в функторе - арность.

`student(ivanov)` - арность 1, поэтому разные знания.

В момент фиксации утверждение с переменной значение не имеет. Поэтому процесс работы системы заключается в том, чтобы найти решение данного предиката.

Когда система подбирает для переменной значение говорят, что система конкретизирует переменную значением. При этом при подборе система может ошибиться, поэтому система ведёт доказательство путём проб и ошибок. Работа системы - процесс доказательства, работающий на основе фактов. Активация происходит путём задания вопроса.

Синтаксическая форма правила:

$A :- B_1, B_2, \dots, B_k.$

Факты, содержащие переменные - основные, не содержащие - неосновные. A - заголовок правила. B₁-B_k - тело.

$student(X, mgtu) :- документы(X, att), \dots$

Заголовок является фиксацией знания о том, что между аргументами возможна истинная связь (X действительно студент, если...)

Особенный способ работы с переменными:

Вместо того, чтобы сначала задавать значения переменной, задаётся условие и система ищет такие значения переменных, чтобы на вопрос ответить да. При этом сохраняются и возвращаются значения для этого ответа.

Переменные нужны для передачи данных во времени и пространстве.

Во времени - переменная получила значение и через какое-то время оно было использовано

В пространство - передача между областями данных (памяти) - перенос в физическом пространстве

Во время фиксации переменная не имеет значения.