СОДЕРЖАНИЕ

1 Лекция 1 — 10 февраля 2025

1.1 Методологии

5

1. императивная — пошаговая работа ПК;
2. функциональное;
3. логическое;
4. ООП;
5. параллельное.

1.2 Коротко о Lisp

4

1. Идея Lisp(List Processing — обработчик списков) — математика + функции, символьная обработка информации;
2. аппрективный язык;
3. безтиповый язык;
4. нет синтаксической разницы между данными и программой.

1.3 Функциональное программирование

Функциональное программирование определяет процессы вычисления на основе достаточно строгих абстрактных понятий и методов символьной обработки данных, а значит допускает высокий уровень абстракции, позволяя выполнить формализацию множества объектов и определить полную семантическую систему операций над объектами.

При этом классы задач и их решений могут быть представлены строгими формулами. Формулы могут быть упрощены введением новых функциональных символов(данных, действий, формул), то есть система может быть консервативно расширена.

Базис — минимальный набор обозначений, к которым можно свести все правильные вычислительные формулы системы, реализация которого является минимальной версией системы. Разные расширения базиса могут восприниматься как разные системы.

4 Базис Lisp образуют:

1. атомы;
2. точечные пары;
3. базовые функции;
4. базовые функционалы.

Системы, в которых можно включать любые символы, наделяя их смыслом, называются аппрекативными.

Разработка системы включает фазу формирования базиса и наполнение ядра системы в терминах, которые не сводятся к её языку.

1.4 S-выражения, атом, точечная пара

Lisp использует S-выражения(символьные).

S-выражение ::= <атом>|<точечная пара>

Атом — неделимая последовательность символов.

3 Атомы делятся на:

1. символы(идентификаторы);
2. специальные символы(T, Nil);
3. самоопределимые атомы — числа, строки.

S-выражения — все виды атомов + списки.

Более сложно организованные данные — списки и точечные пары, в памяти строятся из унифицированных структур — бинарных узлов.

<Точечная пара(тп)> ::= (<атом>.<атом>) || (<тп>.<атом>) ||
(<атом>.<тп>) || (<тп>.<тп>)

<Список> ::= <пустой список> | <непустой список>, где:

— <пустой список> ::= () | Nil;

— <непустой список> ::= (<первый элемент>.<хвост>);

— <первый элемент> ::= <S-выражение>;

— <хвост> ::= <список>.

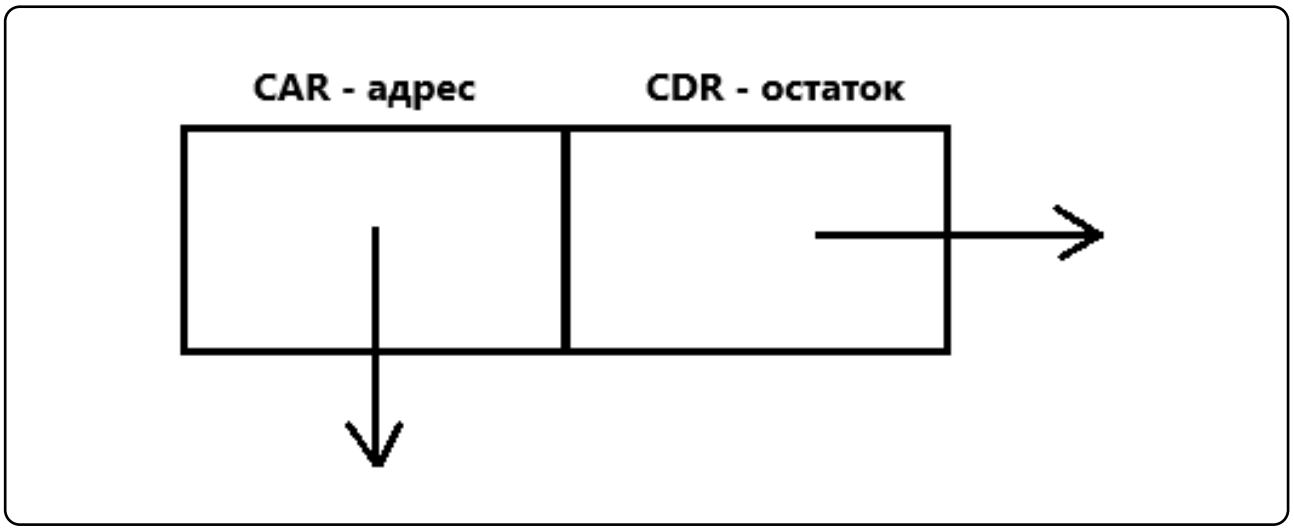

Рисунок 1.1 – Расположение точечной пары в памяти.

Функции car/cdr — переход по car/cdr указателю.

1.5 Структуры

() — признак структуры.
Список (A B C), где A — имя функции.
Пример: (car '(A B C))

2 Лекция 2 — 17 февраля 2025

2.1 Функции

Программа на Lisp представляет собой вызов функции на верхнем уровне. Синтаксически программа оформляется в виде S-выражения.

При создании программы используются стандартные функции Lisp и функции пользователя. Стандартные функции Lisp носят частичный характер, то есть работают не всегда корректно с поданными аргументами. Функции всюду определимы в Lisp, то есть если функция приняла аргументы, то что-то должна вернуть в любом случае.

2 Способы определения функций:

1. lambda-нотация (функция без имени);
2. с использованием макро определения(спец функции) DEFUN.

```
(defun sum (a b) (+ a b))

(sum 2 3)

((lambda (a b) (+ a b)) 2 3)
```

λ-выражение = (lambda λ-список форма).
Вызов: (λ-выражение форма1 ... формаN)

Рисунок 2.1 – Примеры вызова defun и lambda функций

Вычисление функций и выполнение программ осуществляет интерпретатор eval. Вычисление функций без имени может быть выполнено с помощью функционала apply или funcall. Вместо циклов и операторов используются рекурсивные функции.

2.2 Классификация функций

6

1. чистые – (так, как принято в математике, вычисляются все аргументы);
2. специальные(формы) – переменное кол-во аргументов или они обрабатываются по-разному, могут вычисляться не все (cond, if, and, or, quote, eval, lambda).
3. псевдофункции – создают спецэффекты (например – вывод на экран);
4. с вариантыми значениями;
5. высших порядков(функционалы) – в качестве аргумента используют другие функции или возвращают функцию в качестве результата(apply, funcall);
6. базисные функции.

2.3 Классификация базисных функций

3

1. Селекторы – car, cdr;
2. Функции-конструкторы:
   - cons – создает списковую ячейку и расставляет указатели, передается 2 S-выражения;
   - list – (не базисная форма) создает столько списковых ячеек, сколько аргументов, менее эффективная;
3. Предикаты – логические функции:
   - null – пустая ли структура;
   - atom – является ли атомом;
   - consp – является ли структура точечной парой;
   - listp – список или нет(более качественная проверка относительно consp);
   - eq – проверяет, ссылаются ли два объекта на одно и то же место в памяти. Подходит для символов, точечных пар.
   - eql – сравнивает атомы и числа одного типа (= применима только к числам, любого типа), не сравнивает списки;
   - equal – eql + списки, но не сравнивает числа разных типов (2 и 2.0);
   - equalp – сравнивает всё.

2.4 Символ апостроф

Апостроф эквивалентен вызову функции quote. Она позволяет заблокировать вычисление своего единственного аргумента:

```
'(A B C) == (quote (A B C))
```

Рисунок 2.2 – Синтаксис апострофа

2.5 eval

Функция EVAL - это функция, которая может вычислить любое правильно составленное выражение языка LISP.

```
(print (eval '(+ 2 3))) ; 5

(print '(+ 2 3))        ; (+ 2 3)
```

Рисунок 2.3 – Пример работы функции eval

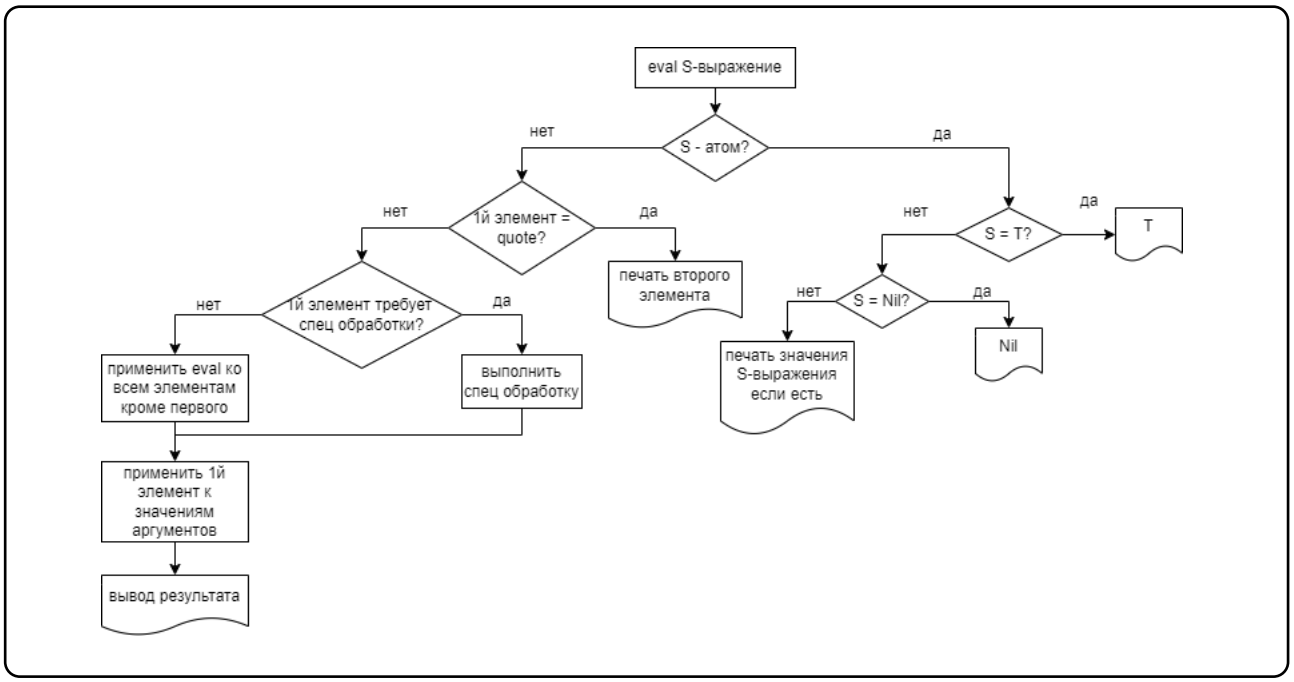

Рисунок 2.4 – Схема работы функции eval

3 Лекция 3 — 24 февраля 2025

3.1 Представление атома в памяти

Атом представлен в памяти пятью указателями. Последний из них – package, был добавлен на будущее развитие языка, первоначально представлял заглушку.

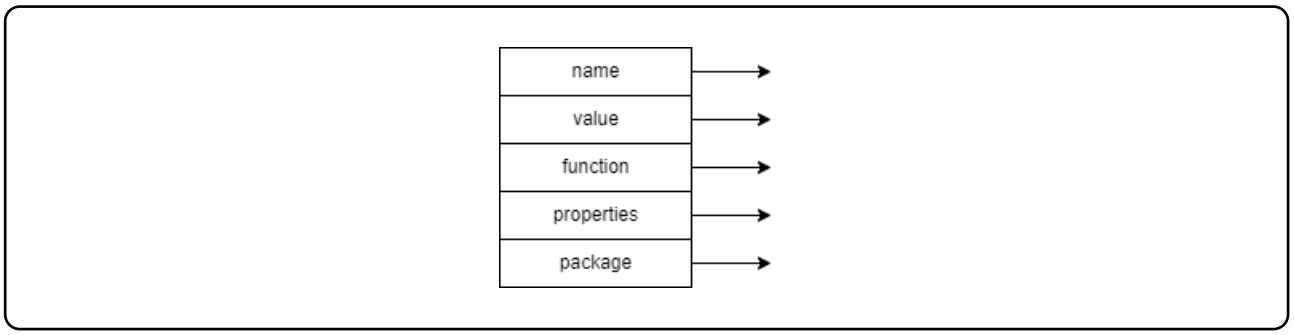

Рисунок 3.1 – Представление атома в памяти

3.2 set, setq

SET требует ровно два аргумента. Значением первого аргумента должен быть атом, а значением второго - произвольное S-выражение. Функция присваивает атому, являющемуся значением первого аргумента, значение второго аргумента. Это значение возвращается в качестве результата.

Функция SETQ отличается от SET тем, что не вычисляет значение первого аргумента. Поэтому первый аргумент при вызове SETQ не нужно квотировать. В остальном функция SETQ эквивалентна функции SET.

```
(set 'a 4)
(setq b 5)

(set 'c '(1 2 3))
(setq d '(1 2 3))
```

Рисунок 3.2 – Пример работы функций set и setq

3.3 setf

SETF — это макрос. Он позволяет делать присваивания в произвольное место. Первым аргументом SETF может быть как выражение, так и имя переменной. SETF позволяет устанавливать значение сразу нескольким переменным.

```
(setf a 3)

(setf b 4
      c 5)

(setf x '(1 2 3))    ; (1 2 3)
(setf (cadr x) 10)   ; (1 10 3)
```

Рисунок 3.3 – Пример работы setf

3.4 cond, if

В cond отбирается первое подвыражение, чья форма условия вычисляется в не-nil. Все остальные подвыражения игнорируются. Формы отобранного подвыражения последовательно выполняются.

В if выполняется форма test. Если результат не равен nil, тогда выбирается форма then. Иначе выбирается форма else. Выбранная ранее форма выполняется, и if возвращает то, что вернула это форма.

```
; (if test then else) == (cond (test then) (t else))

(setf a nil
      b nil)

(cond (a a)
      (b b)
      (t "a and b == nil")) ; "a and b == nil"

(if (equalp a nil)
"equal" "not equal")        ; "equal"
```

Рисунок 3.4 – Пример работы if и cond

3.5 Логические операторы and, or, not

NOT возвращает t, если аргумент является nil, иначе возвращает nil. Функции NOT и null полностью эквиваленты. По соглашению принято использовать null, когда надо проверить пустой ли список, и NOT, когда надо инвертировать булево значение.

AND последовательно слева направо вычисляет формы. Если какая-либо форма formN вычислилась в nil, тогда немедленно возвращается значение nil без выполнения оставшихся форм. Если все формы кроме последней вычисляются в не-nil значение, AND возвращает то, что вернула последняя форма.

OR последовательно выполняет каждую форму слева направо. Если какая-либо непоследняя форма выполняется в что-либо отличное от nil, OR немедленно возвращает это не-nil значение без выполнения оставшихся форм. Если все формы кроме последней, вычисляются в nil, OR возвращает то, что вернула последняя форма.

```
(setf a 1 b 2)

(not nil)                ; T

; (and form1 form2 ... )
(and (eq a 1) (eq b 2)) ; T

; (or form1 form2 ... )
(or (eq a 10) (eq b 2)) ; T
```

Рисунок 3.5 – Пример работы and, or, not

3.6 let и let*

LET используют, чтобы связать значение с символом таким образом, чтобы интерпретатор не спутал переменную с таким же именем, но определенную вне функции.

Локальные переменные, создаваемые выражением LET, сохраняют свои значения только внутри самого выражения LET.

Символы в списке переменных – это переменные, которым особая форма LET присваивает начальное значение. Если символ только один, то ему

присваивается значение nil; если символ входит в состав двухэлементного списка, то ему назначается значение, которое возвращается после вычисления второго элемента списка.

Функция LET* отличается от LET только тем, что при вычислении значения очередной переменной из списка доступны значения уже ранее вычисленных переменных (последовательное вычисление).

```
;(let ((переменная значение)
;      (переменная значение)
;      ...)
;      (форма1) ... (формаN))

(setf a 3)

(defun myfunc ()
    (let ((a 4)
          b) ; b присвоится nil
    a)
)

(myfunc) ; 4
```

Рисунок 3.6 – Пример работы let

4 Лекция 4 — 3 марта 2025

4.1 Функции работы со списками

2 Функции работы со списками бывают:

1. разрущающие структуру списков;
2. неразрущающие.

2 Примечания:

1. все стандартные функции Lisp работают только с верхнем уровнем списков;
2. n в начале имени функции означает, что функция не создаёт копию списка.

4.2 append и nconc

Функция APPEND выстраивает в одну цепочку элементы всех списков, поставляемых в качестве аргументов. APPEND объединяет элементы списков, расположенные лишь на самом верхнем уровне. APPEND создаёт копии все списков, кроме последнего.

```
(setf l1 '(a b)
      l2 '(c d)
      l3 '((e f)))

(print (append l1 l2 l3)) ; (A B C D (E F))
(print l1)                ; (A B)
(print l2)                ; (C D)
(print l3)                ; ((E F))
```

Рисунок 4.1 – Пример работы функции append

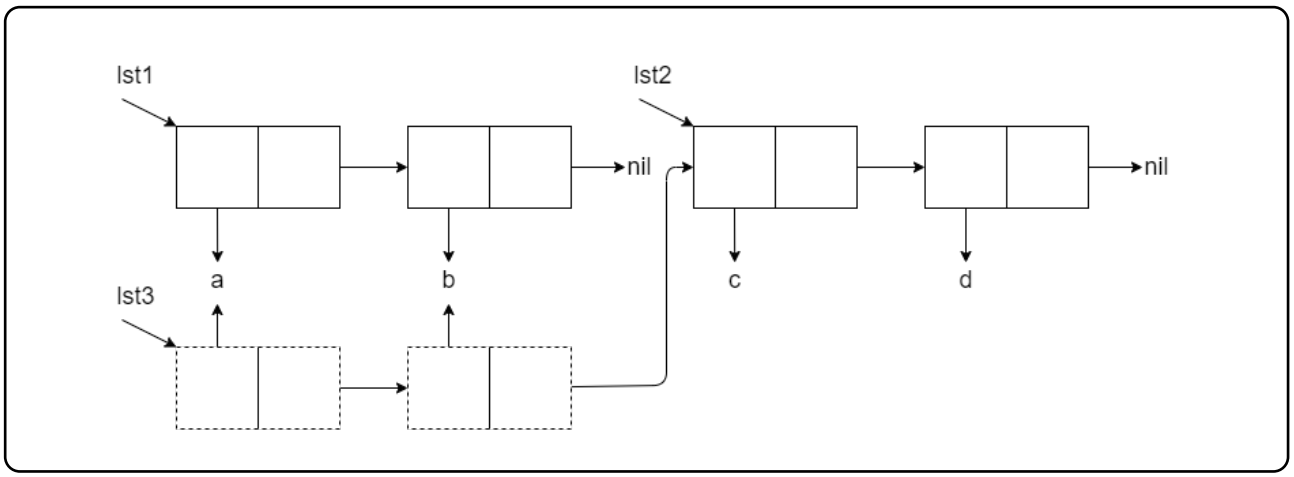

Рисунок 4.2 – Схема работы функции append

Работа NCONC аналогична APPEND, только она не создаёт копий, а переставляет указатели.

```
(setf l1 '(a b)
      l2 '(c d)
      l3 '((e f)))

(print (nconc l1 l2 l3)) ; (A B C D (E F))
(print l1)               ; (A B C D (E F))
(print l2)               ; (C D (E F))
(print l3)               ; ((E F))
```

Рисунок 4.3 – Пример работы функции nconc

4.3 reverse и nreverse

Функции переставляют элементы в обратном порядке.

```
(setf l1 '(a b c)
      l2 '(d e f))

(setf l3 (reverse l1))
(print l1) ; (A B C)
(print l3) ; (C B A)

(setf l4 (nreverse l2))
(print l2) ; (D)
(print l4) ; (F E D)
```

Рисунок 4.4 – Пример работы функций reverse, nreverse

4.4 nth, nthcdr

NTH возвращает n-нный элемент списка list. Индексация с 0. NTHCDR выполняет для списка операцию cdr n раз, и возвращает результат. Если N больше длины списка, то возвращается nil. Отрицательное N вызовет ошибку.

```
(setf l1 '(a b c))

(print (nth 0 l1))     ; A
(print (nthcdr 1 l1)) ; (B C)
```

Рисунок 4.5 – Пример работы функций nth, nthcdr

4.5 last, length

LAST возвращает последние N ячеек списка. Если N не указано, возвращается последняя списковая ячейка.

LENGTH возвращает количество списковых ячеек. Может работать некорректно на циклических списках.

```
(setf l1 '(a (b c)))

(print (length l1))    ; 2
(print (last l1))      ; ((B C))
(print (last l1 2))    ; (A (B C))
```

Рисунок 4.6 – Пример работы функций last, length

4.6 remove

REMOVE удаляет все вхождения элемента, указанного первым параметром, из списка, указанного вторым параметром. По умолчанию внутри для сравнения элементов используется функция EQL, которая не позволяет сравнивать вложенные списки. С помощью оператора двоеточие можно явно указать функцию сравнения.

```
(setf l1 '(b a (c a) d))

(print (remove 'a l1))                    ; (B (C A) D)
(print (remove '(c a) l1))                ; (B A (C A) D)
(print (remove '(c a) l1 :test #'equal))  ; (B A D)
```

Рисунок 4.7 – Пример работы функции remove

4.7 rplaca, rplacd

Данные функции изменяют соответственно car и cdr элементы cons-ячеек первого аргумента-списка на значение второго аргумента, и возвращают модифицированный первый аргумент.

```
; rplaca (cons, s_exp)
; rplacd (cons, s_exp)

(setf l1 '(a b c))
(setf l2 '(d e f))
(setf l3 '(h i j))

(print (rplaca (cdr l1) 'z)) ; (Z C)
(print l1)                   ; (A Z C)

(print (rplacd (cdr l1) l2)) ; (Z D E F)
(print l1)                   ; (A Z D E F)

(print (rplaca l3 '(k l)))   ; ((K L) I J)
(print l3)                   ; ((K L) I J)
```

Рисунок 4.8 – Пример работы функций rplaca, rplacd

4.8 subst, nsubst

SUBST выполняет замену в выражении, заданном третьим аргументом все вхождения значения второго аргумента НА ВСЕХ УРОВНЯХ на значение первого аргумента.

```
; (subst new old lst)
(setf l1 '((a b) c))

(print (subst 'a 'b l1))  ; ((A A) C)
(print l1)                ; ((A B) C)

(print (nsubst 'a 'b l1)) ; ((A A) C)
(print l1)                ; ((A A) C)
```

Рисунок 4.9 – Пример работы функций subst, nsubst

4.9 member

MEMBER осуществляет поиск элемента, удовлетворяющего условию, в списке. Если элемент не найдёт, возвращается nil. Иначе возвращается часть списка, начинающаяся с искомого элемента.

```
(setf l1 '(a b c))

(print (member 'b l1)) ; (B C)
(print (member 'd l1)) ; NIL
(print l1)             ; (A B C)
```

Рисунок 4.10 – Пример работы функции member

4.10 union, intersection, set-difference

UNION строит список, содержащий объединение элементов, входящих в значения двух ее списков-аргументов.

INTERSECTION возвращает список, состоящий из элементов, входящих в оба списка.

SET-DIFFERENCE возвращает набор, содержащий элементы первого списка, которые отсутствуют во втором списке.

Все три функции сохраняют неизменными исходные списки.

```
(setf l1 '(a b c)
      l2 '(d e a))

(print (union l1 l2))           ; (C B D E A)
(print (intersection l1 l2))    ; (A)
(print (set-difference l1 l2)) ; (C B)
```

Рисунок 4.11 – Пример работы функций union, intersection, set-difference

4.11 Ассоциативные таблицы

Список Lisp можно рассматривать как таблицу: множество точечных пар, где первый элемент можно воспринимать как ключ, а второй как значение.

5 Основные функции для работы с таблицами:

1. pairlis – принимает два списка и создаёт ассоциативный список, который связывает элементы первого списка с соответствующими элементами второго;
2. assoc – возвращает первую пару, удовлетворяющую условию, или nil, если такой пары не было найдено;
3. rassoc – реверсивный assoc;
4. acons – создаёт новый ассоциативный список, с помощью добавления пары к старому a-list;
5. sublis –принимает два аргумента. Значением первого аргумента должен быть ассоциативный список. Значением второго аргумента может быть произвольное S-выражение. Функция берет каждый атом, входящий в значение второго аргумента, и заменяет его на ассоциацию из ассоциативного списка. Функция возвращает результат замены.

```
(setf l1 '((Москва.Россия) (Лондон.Британия) (Берлин.Германия)))
(setf l1 (pairlis
          '(Москва Лондон Берлин) '(Россия Британия Германия)))

(print (assoc 'Москва l1))      ; (МОСКВА . РОССИЯ)
(print (assoc 'Вашингтон l1))   ; NIL
(print (rassoc 'Германия l1))   ; (БЕРЛИН . ГЕРМАНИЯ)

(acons 'Минск 'Беларусь l1)

(print (sublis '((a . 1) (b . 2) (c . 3)) '(a b c))) ; (1 2 3)
(print (sublis '((a . 1) (b . 2) (c . 3)) 'a))       ; 1

; Пример замены значения по ключу
(rplacd (assoc 'Москва l1) 'МИР)
(print l1)    ; ((БЕРЛИН . ГЕРМАНИЯ)
              ;  (ЛОНДОН . БРИТАНИЯ) (МОСКВА . МИР))
```

Рисунок 4.12 – Пример работы функций pairlis, assoc, rassoc, acons, sublis

5 Лекция 5 — 10 марта 2025

5.1 Функционалы

2 Функционалы делятся на два типа:

1. отображающие – применяют функцию многократно к элементам списка, имя часто начинается на map;
2. применяющие – apply, funcall.

5.2 apply

APPLY принимает 2 аргумента, из которых первый аргумент представляет собой функцию, которая применяется к элементам списка, составляющим второй аргумент функции. Должно быть выдержано соответствие между структурой элементов списка и функцией. Функция может быть определена через lambda. Символ # называется функциональной блокировкой. Необходим для явного указанию интерпретатору, что первый аргумент APPLY – функция, а не переменная. В противном случае возникнет ошибка несвязанной переменной.

```
(apply #' лямбда-выражение (arg1 arg2 ... argN))
(print (apply #'list '(1 2 3)))          ; (1 2 3)
(print (apply #'+ '(1 2 3)))             ; 6
(print (apply (function +) '(1 2 3)))    ; 6
```

Рисунок 5.1 – Пример работы функции apply

5.3 funcall

FUNCALL по своему действию аналогичен APPLY, но аргументы для вызываемой функции он принимает не списком, а по отдельности.

```
(funcall #'func_name arg1 arg2 ... argN)

(print (funcall #'list 1 2 3))        ; (1 2 3)

(print (funcall #'+ 1 2 3))           ; 6

(print (funcall (function +) 1 2 3)) ; 6
```

Рисунок 5.2 – Пример работы функции funcall

5.4 mapcar

MAPCAR возвращает список, полученный из результатов применения функции к элементам списка. Список формируется с помощью LIST. Количество аргументов у функции должно соответствовать количеству передаваемых в MAPCAR списков. На каждом шаге работы MAPCAR выделяет i-е элементы списков и передаёт в функцию. Если передаваемые списки разной длины, на выходе получится список длины минимального списка.

```
; (mapcar #'fun lst)
; (mapcar #'fun lst1 lst2 ... lstN)

(defun sum(a b) (+ a b))

(print (mapcar #'(lambda (x) (* x x)) '(1 2 5))) ; (1 4 25)
(print (mapcar #'sum '(1 2 3) '(4 5 6)))         ; (5 7 9)
(print (mapcar #'sum '(1 2 3) '(4)))             ; (5)
```

Рисунок 5.3 – Пример работы функции mapcar

5.5 maplist

MAPLIST применяет переданную функцию сначала ко всему списку, потом к нему без первого элемента и т.д. Список формируется с помощью LIST. Работа MAPLIST с несколькими списками аналогична MAPCAR, только возвращается список списков.

```
; (maplist #'fun lst)
; (maplist #'fun lst1 lst2 ... lstN)

(defun intersect(l1 l2) (intersection l1 l2))

(print (maplist #'(lambda (l) (length l)) '(1 2 5))) ; (3 2 1)
(print (maplist #'intersect '(1 2 3 4)
                            '(4 3 2 1)))    ; ((4 3 2 1) (3 2) NIL NIL)
(print (maplist #'intersect '(1 2 3) '(1))) ; ((1))
```

Рисунок 5.4 – Пример работы функции maplist

5.6 mapcan, mapcon

MAPCAN и MAPCON аналогичны MAPCAR и MAPLIST соответственно, за исключением того, что результат формируется с помощью функции NCONC, что разрушает искомый список.

5.7 find-if

FIND-IF применяет функцию к car элементам списка, возвращает первый car элемент, удовлетворивший условию. В противном случае возврат NIL.

```
(print (find-if #'numberp '(a 2 3))) ; 2
```

Рисунок 5.5 – Пример работы функции find-if

5.8 remove-if, remove-if-not

REMOVE-IF возвращает новый список с элементами, для которых предикат вернул NIL.

REMOVE-IF-NOT возвращает новый список с элементами, для которых предикат вернул T.

```
(setf l1 '(1 (2) 3))
(print (remove-if #'numberp l1))      ; ((2))
(print (remove-if-not #'numberp l1)) ; (1 3)
```

Рисунок 5.6 – Пример работы функций remove-if, remove-if-not

5.9 reduce

REDUCE каскадно преобразует элементы списка, используя двухаргументную функцию. Сначала к первому и второму, потом к образовавшемуся элементу и 3 элементу списка и т.д.

```
(setf l1 '(1 2 3 4 5))

(print (reduce #'cons l1))    ; ((((1 . 2) . 3) . 4) . 5)
(print (reduce #'+ l1 :initial-value -15)) ; 0
(print l1)  ; (1 2 3 4 5)
```

Рисунок 5.7 – Пример работы функции reduce

5.10 every, some

EVERY применяет предикат к каждому элементу списка. Если предикат возвращает NIL, EVERY возвращает NIL. Если дошла до конца списка, то возвращается T.

SOME применяет предикат к элементам до тех пор, пока он не вернёт T. Если дошла до конца списка, то возвращает NIL.

По аналогии с MAPCAR можно передавать сразу несколько списков в EVERY и SOME.

```
(setf l1 '(1 2 3 4 5))
(setf l2 '(-1 -2 -3 -4 -5))

(defun check(x y) (= (+ x y) 0))

(print (every #'numberp l1))  ; T
(print (every #'check l1 l2)) ; T

(print (some #'listp l1))  ; NIL
(print (some #'(lambda(x y) (> (+ x y) 0)) l1 l2)) ; NIL
```

Рисунок 5.8 – Пример работы функций every, some

6 Лекция 6 — 17 марта 2025

6.1 Основные понятия о рекурсии

2 Методы организации повторных вычислений:

1. функционалы;
2. рекурсия.

3 Классификация рекурсии:

1. простая – один рекурсивный вызов;
2. первого порядка – несколько рекурсивных вызовов;
3. взаимная – функции рекурсивно вызывают друг друга.

2 Классификация рекурсии в рамках классификации:

1. хвостовая – рекурсия, при завершении которой результат УЖЕ сформирован, не отстается отложенных вычислений;
2. дополняемая – при обращении к рекурсии используется дополнительная функция ВНЕ рекурсивного вызова;
3. множественная – на одной ветке несколько рекурсивных вызовов.

3 Проблемы рекурсии:

1. как совершить первый вызов;
2. как остановить выполнение;
3. как организовать новый шаг.

6.2 Примеры построения рекурсий разных видов

```
; Общий вид дополняемой рекурсии
(defun fun(lst)
    (cond (end_test end_value)
          (t (add_fun add_value (fun reduced_lst)))))

(defun my_len(lst)
    (cond ((null lst) 0)
          (t (+ 1 (my_len (cdr lst))))))

(print (my_len '(1 2 3 4 5 33)))
```

Рисунок 6.1 – Пример дополняемой рекурсии

```
; Общий вид хвостовой рекурсии
(defun fun (x)
  (cond
    (end_test end_value)
    (t (fun changed_x))))

(defun my_numberp(lst)
    (cond
        ((numberp (car lst)) (car lst))
        (t (my_numberp (cdr lst)))))
```

Рисунок 6.2 – Пример хвостовой рекурсии

```
; Общий вид множественной рекурсии
(defun fun(lst)
    (cond (end_test end_value)
          (t (combiner (fun reduced_1_lst)
                       (fun reduced_2_lst)))))

; Поиск первого числа на любом уровне списка
(defun first_number (lst)
    (cond ((numberp lst) lst)
          ((atom lst) nil)
          (t (or (first_number (car lst))
                 (first_number (cdr lst))))))
```

Рисунок 6.3 – Пример множественной рекурсии

```
(defun insert-help (x lst)
    (cond ((null lst)
                (list x))
          ((<= x (car lst))
                (cons x lst))
          (t (cons (car lst)
                   (insert-help x (cdr lst))))))

(defun sort-help(lst1 lst2)
    (cond ((null lst1) lst2)
        (t (sort-help (cdr lst1)
                      (insert-help (car lst1)
                                   lst2)))))

(sort-help '(3 2 4 1) nil) ; (1 2 3 4)
```

Рисунок 6.4 – Пример сортировки вставками с помощью рекурсии