ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ДИСТАНЦИОННОГО ОБУЧЕНИЯ И ПОВЫШЕНИЯ КВАЛИФИКАЦИИ

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

Учебно-методическое пособие

по дисциплине

«ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ»

Авторы
Долгов В.В.
Кузин А.П.

Ростов-на-Дону, 2018

Аннотация

Учебно-методическое пособие предназначено для студентов очной формы обучения по направлениям 02.03.03 «Математическое обеспечение и администрирование информационных систем», 09.03.04 «Программная инженерия».

Авторы

доцент, к.т.н.,
зав.каф. ПОВТиАС
Долгов В.В.
Ст.преподаватель
А.П. Кузин

Оглавление

1. ВВЕДЕНИЕ

Идея функционального программирования опирается на интуитивное понятие о функциях как о достаточно общем механизме представления и анализа решений сложных задач. Механизм функций основательно изучен математиками, и это позволяет программистам наследовать выверенные построения, обладающие предельно высокой моделирующей силой. Систематическое применение функционального программирования впервые достаточно ярко было продемонстрировано Джоном Маккарти и его учениками в методах реализации языка Лисп и программирования на этом языке. Наиболее очевидные из этих методов были успешно ассимилированы другими языками и системами программирования. Обычно про функциональное программирование вспоминают при смене технологий, когда возрастает роль аналитики и исследовательских задач. В настоящее время часто употребляют термин «функциональность» при сравнительной характеристике информационных систем, что, видимо, свидетельствует о проявлении новой метрики, заслуживающей отдельного рассмотрения.

Функциональный стиль объединяет разные подходы к определению процессов вычисления на основе достаточно строгих абстрактных понятий и методов символьной обработки данных. Связь функционального программирования с математическими основами позволяет в тексте программы наследовать доказательность построения результата, если она достигнута, причем с использованием разных методов абстрагирования решаемой задачи.

Сложность решения задач с помощью функциональных определений преодолевается чисто алгебраически: нацеленностью на формализацию основного множества объектов и определения полной семантической системы операций над ними. Это позволяет представлять классы задач и их решений строгими формулами, для наглядности упрощаемыми введением дополнительных функциональных символов. При необходимости такие символы вносятся в определение алгебраической системы, что приводит к ее расширению. Вводятся новые функции, подобные леммам и другим вспомогательным построениям в математике. Активно используется рекурсия и символьные обозначения как данных, так и действий, удобных при определении функций.

Формально такое расширение является консервативным (новый символ определен с помощью старых), оно гарантирует сохранение всех функциональных свойств исходной системы. Минимальный набор обозначений, к которым можно свести все пра-

вильные, т.е. вычислимые формулы системы, играет роль базиса системы, реализация которого является минимальной версией всей системы.

Следует отметить, что при формальной эквивалентности разные расширения одного и того же базиса могут поддерживать различные применения и восприниматься как совершенно разные системы. Программисты вынуждены строить расширения, которые выглядят неконсервативными, так как целью их работы является именно изменение ряда характеристик функционирования исходной системы (компьютера, системы программирования и т.п.) с частичным сохранением заданных функциональных свойств.

Большинство систем программирования разработано с ориентацией на расширение, уточнение и настройку пользователем реализованных программных средств, свойства которых определены и обеспечены в процессе разработки. Такое разделение труда естественно при ориентации на решение задач с исчерпанным или четко ограниченным исследовательским компонентом. Но исходная разработка любой системы включает фазу формирования базиса и наполнения ядра системы в терминах, которые не сводятся к ее языку. Это позволяет независимо рассматривать один еще более общий уровень — аппликативные системы, в которые можно включать любые символы с определенным смыслом. Поведение такой системы будет обусловлено набором включенных в нее символов.

Основная трудность перехода к функциональному программированию — соблазн легкого пути, т.е. стремление быстро смоделировать привычные средства и методы программирования. Более надежный путь — исследовать функциональное программирование как незнакомый мир. Идеи функционального программирования легче воспринять как самостоятельную теорию или интеллектуальную игру, которая новыми путями непременно приведет к знакомым и интересным задачам, но обеспечит преимущество — изящные решения и глубину понимания.

Джон Маккарти предложил проект языка Лисп (LISP - LISt Processing) в качестве средства исследования границ применимости компьютеров, в частности, методом решения задач искусственного интеллекта. Идеи этого языка вызвали не утихающие по сей день дискуссии о приоритетах в программировании и сущности программирования. Лисп послужил эффективным инструментом экспериментальной поддержки теории программирования и развития сферы его применения. Рост интереса к Лиспу коррелирует с улучшением элементной базы, повышением эксплуата-

ционных характеристик оборудования и появлением новых сфер применения ИТ.

Существует и активно применяется более трехсот диалектов Лиспа и родственных ему языков: Clojure, Interlisp, muLisp, Clisp, Scheme, Cmucl, Logo, Hope, Sisal, Haskell, Miranda и др.

Цель лабораторного практикума – практическое изучение функциональной методологии программирования на примере использования диалекта языка Common Lisp в реализации SBCL (Steel Bank Common Lisp).

2. ИНТЕРПРЕТАТОР ЯЗЫКА LISP И ВЫЧИСЛЕНИЯ В LISP-Е

2.1. Общие сведения о языке Lisp

Лисп ориентирован на обработку нечисловых задач. Он основан на алгебре списочных структур, лямбда-исчислении и теории рекурсий. Так как язык имеет функциональную направленность, то любое предложение, заключенное в круглые скобки, введенное вне редактора считается функцией и выполняется сразу после попадания в интерпретатор.

Чтобы предотвратить вычисление значения выражения, нужно перед этим выражением поставить апостроф «’» или использовать функцию «quote». В Лиспе формы представления программы и обрабатываемых ею данных одинаковы. И то и другое представляется списочной структурой, имеющей одинаковую форму как записи, так и хранения. Типы данных не связаны с именами объектов данных, а сопровождают сами объекты. Переменные могут в различные моменты времени представлять объекты различных типов (такие языки называют языками с динамической типизацией). Основные базовые типы данных языка – атомы и списки.

Атомы – это символы (symbol) и числа.

Список – это упорядоченная последовательность, элементами которой являются атомы либо списки. Списки заключаются в круглые скобки, элементы списка разделяются пробелами. Несколько пробелов между символами эквивалентны одному пробелу. Первый элемент списка называется «головой», а остальная часть списка, т. е. список без первого элемента, называется «хвостом». Хвост списка, соответственно может являться списком. Если список состоит из одного элемента, то его хвост пустой список.

Список, в котором нет ни одного элемента, называется пустым и обозначается «()» либо NIL.

Символ – это имя, состоящее из букв, цифр и специальных знаков, которое обозначает какой-нибудь предмет, объект, действие. В Лиспе символы обозначают числа, другие символы или более сложные структуры, программы (функции) и другие объекты языка. Символы могут состоять как из прописных, так и из строчных букв, хотя в большинстве Лисп-систем, как и в описываемой здесь версии, прописные и строчные буквы отождествляются и представляются прописными буквами.

Символы T и NIL имеют в Лиспе специальное назначение: «T» обозначает логическое значение истина, а «NIL» – логическое значение ложь.

Код, написанный на языке Lisp, может храниться в файлах и загружаться в интерпретатор, используя специальную функцию *LOAD*, как в примере ниже.

(load <имя_файла_с_программой_или_функцией>)

Эта функция загружает файл выражений в интерпретатор языка и выполняет их. Если операция успешно завершена, LOAD возвращает имя последней функции, определенной в файле. Если операция не выполнена, LOAD возвращает имя файла в виде строкового выражения. Функция LOAD не может вызываться из другой функции LISP. Она должна вызываться непосредственно с клавиатуры, в то время как ни одна другая функция LISP не находится в процессе выполнения.

2.2. Символы и свойства символов

Основные функции определения и присваивания значений символам: *SET*, *SETQ*, *SETF*.

Функция *SET* присваивает символу или связывает с ним некоторое значение. Причем она вычисляет оба своих аргумента. Установленная связь действительна до конца работы интерпретатора, если этому имени не будет присвоено новое значение.

Например,

```
 (SET 'a '(b c d))              // ⇨ (b c d)
a                               // ⇨ (b c d)
(SET (CAR a) (CDR (o f g))) // ⇨ (f g)
a                               // ⇨ (b c d)
(CAR a)                         // ⇨ b
b                               // ⇨ (f g)
                          // ⇨ (f g)
```

Значение символа вычисляется с помощью специальной функции *symbol-value*, которая возвращает в качестве значения значение своего аргумента.

```
(symbol-value (CAR a))      // ⇨ (f g)
```

Функция *SETQ* – связывает имя, не вычисляя его. Эта функция отличается от SET тем, что вычисляет только свой второй аргумент (значение, связываемое с символом).

```
(SETQ d '(l m n))           // ⇨ (l m n)
```

Функция *SETF* – обобщенная функция присваивания. SETF используется для занесения значения в уже существующую ячейку памяти. Ее общие синтаксис *(SETF ячейка-памяти значение)*, где ячейкой памяти может быть практически любое значение или выражение языка, способное хранить какое-нибудь значение.

```
(SETF ячейка '(a b c))      // ⇨ (a b c)
ячейка                      // ⇨ (a b c)
(SETQ a '(1 4 5))           // ⇨ (1 4 5)
a                           // ⇨ (1 4 5)
(SETF (car a) '(1 2 3))     // ⇨ (1 2 3)
a                           // ⇨ ((1 2 3) 4
5)
(SETF (cdr a) '(6 7))       // ⇨ (6 7)
a                           // ⇨ ((1 2 3) 6
7)
```

В языке Лисп с символом можно связать именованные свойства. Свойства символа записываются в хранимый вместе с сим-

волом список свойств. Свойство имеет имя и значение. Список свойств может быть пуст и его можно изменять или удалять без ограничений.

Встроенная функция *GET* – возвращает значение свойства, связанного с символом.

```
(GET символ свойство)
```

При отсутствии свойства функция *GET* возвращает *NIL* в качестве ответа.

```
(GET 'студент 'имя)          // ⇨ Иван
(GET 'студент 'группа)       // ⇨ NIL
```

Для присваивания символу свойств в языке Common Lisp отдельной функции нет. Для этого используются уже известные нам функции:

```
(SETF (GET символ свойство) значение)
```

Например,

```
(SETF (GET 'студент 'группа) 'РВ-90-1)
   // ⇨ РВ-90-1
(GET 'студент 'группа)
   // ⇨ РВ-90-1
```

Удаление свойства и его значения осуществляется псевдо функцией *REMPROP*. Она возвращает в качестве значения имя удаляемого свойства. Если удаляемого свойства нет, то возвращается NIL.

```
  (REMPROP символ свойство)
```

Например,

```
(REMPROP 'студент 'группа)          // ⇨
группа
(GET 'студент 'группа)              // ⇨
NIL
(REMPROP 'студент 'ср_бал)          // ⇨
```

```
NIL
```

Для просмотра всего списка свойств используют функцию *SYMBOL-PLIST*. Значением функции является весь список свойств.

```
(SYMBOL-PLIST 'СИМВОЛ)
```

Например,

```
(SYMBOL-PLIST 'студент)     // ⇨ (имя Иван
отчество Иванович
                                        фамилия
Иванов)
```

Свойства символов независимо от их значений доступны из всех контекстов пока не будут явно изменены или удалены. Изменение значения символа не влияет на другие свойства. Свойства символа передаются другому символу с помощью функции SETQ.

2.3. Средства языка для работы с числами

В языке Лисп как для вызова функций, так и для записи выражения принята единообразная префиксная форма записи, при которой имя функции или действия, а также их аргументы записываются внутри круглых скобок:

```
(f x), (g x y), (h x (g y z)) и т. д.
```

В языке Lisp доступны следующие основные арифметические действия:

```
(+ числа)            // сложение чисел
(- число числа)      // вычитание чисел из
числа
(* числа)            // умножение чисел
(/ число числа)      // деление числа на
числа
```

Например,

```
(+ 5 7 4)                    // ⇨ 16
(- 10 3 4 1)                 // ⇨ 2
```

```
(/ 15 3)                    // ⇨ 5
```

Для сравнения чисел можно использовать стандартные операторы:

```
(= число числа)             // ⇨ все числа
равны между собой
(< число числа)             // ⇨ число
меньше для всех чисел
(> число числа)             // ⇨ число
больше для всех чисел
(>= число числа)            // ⇨ число
больше или равно всех чисел
(<= число числа)            // ⇨ число
меньше или равно всех чисел
```

Также можно использовать специальные числовые предикаты:

```
(ZEROP число)               // ⇨ проверка
на ноль
(MINUSP число)              // ⇨ проверка
на отрицательность
(PLUSP число)               // ⇨ проверка
на положительность
```

Логические операции представлены следующими операциями:

```
(NOT объект)                // ⇨ логическое
отрицание
(AND (формы))               // ⇨ логическое
И
(OR (формы))                // ⇨ логическое
ИЛИ
```

Например,

```
(AND (ATOM NIL) (NULL NIL) (EQ NIL NIL))
   // ⇨ T
```

```
(NOT (NULL NIL))
   // ⇨ NIL
```

2.4. Задание к лабораторной работе

1) В командной строке интерпретаторы языка Lisp вычислите нижеприведенные выражения и объясните полученные результаты:

- 3.234 * (45.6 + 2.43)
- 55 + 21.3 + 1.54 * 2.5432 - 32
- (34 - 21.5676 - 43) / (342 + 32 * 4.1)

2) Определите значения следующих выражений и проверьте себя, вычислив их в командной строке интерпретатора:

- ‘(+ 2 (* 3 5))
- (+ 2 ‘(* 3 5))
- (+ 2 (’ * 3 5))
- (+ 2 (* 3 ’5))
- (quote ‘quote)
- (quote 6)

2.5. Контрольные вопросы

1) Перечислите базовые функции.
2) Каковы типы аргументов базовых функций?
3) Какие значения они возвращают?
4) Что такое символ?
5) Различия функций SET, SETQ, SETF?
6) Особенности свойств символов?

3. ОСНОВЫ СИНТАКСИСА ЯЗЫКА. ВСТРОЕННЫЕ ТИПЫ ДАННЫХ И СПИСКИ

В данной лабораторной работе мы познакомиться со средой исполнения Steel Bank Common Lisp, изучим способ записи списков, базовые функции языка для работы со списками, символы и их свойства.

3.1. Базовые функции языка

3.1.1 Функции разбора

Функция CAR возвращает в качестве значения первый элемент списка, который может быть как атомом так и списком.

```
(CAR список)                    // ⇨ S - выражение
(атом либо список)
```

Например,

```
(CAR '(a b c d))                // ⇨ a
(CAR '((a b) c d))              // ⇨ (a b)
(CAR '(a))                      // ⇨ a
(CAR NIL)                       // ⇨ NIL
(CAR ())                        // ⇨ NIL
(CAR '(nil))                    // ⇨ NIL
                                //Голова пустого
списка – пустой список
```

Вызов функции CAR с аргументом (a b c d) без апострофа был бы проинтерпретирован как вызов функции «a» с аргументом «b c d», и было бы получено сообщение об ошибке.

Функция CAR имеет смысл только для аргументов, являющихся списками.

```
(CAR 'a)                        // ⇨ Ошибка
выполнения
```

Функция CDR возвращает в качестве значения хвостовую часть списка, т.е. список, получаемый из исходного списка после удаления из него головного элемента:

```
(CDR список)              // ⇨ хвост (список
без первого элемента)
```

Функция CDR также как и CAR определена только для списков. Примеры выполнения:

```
(CDR '(a b c d))          // ⇨ (b c d)
(CDR '((a b) c d))        // ⇨ (c d)
(CDR '(a (b c d)))        // ⇨ ((b c d))
(CDR '(a))                // ⇨ NIL
(CDR NIL)                 // ⇨ NIL
(CDR ())                  // ⇨ NIL
(CDR 'a)                  // ⇨ Ошибка
выполнения
```

Также последовательность вызовов функций CAR и CDR можно заменить одной составной функции, имя которой будет начинаться с «C», а заканчиваться «R», между этими двумя символами может идти последовательность из не более чем 4 символов «A» или «D», где «A» - обозначает функцию CAR, а «D» - обозначает функцию CDR. Порядок вызов функций справа налево. Пример:

```
(SET a '(a (b c)))              // ⇨ (a (b c))
(CAR (CAR (CDR (a))))           // ⇨ b
(CAADR a)                       // ⇨ b
                                //(CAR (CAR
(CDR (a))))
                                //
эквивалентно(CAADR a)
```

3.1.2 Функция создания CONS

Функция CONS строит новый список из переданных ей в качестве аргументов головы и хвоста.

```
(CONS голова хвост)       // ⇨ новый список
```

Для того чтобы можно было включить первый элемент функции CONS в качестве первого элемента значения второго аргумента этой функции, второй аргумент должен быть списком. Значением функции CONS все- гда будет список:

```
(CONS 'a '(b c))          // ⇨ (a b c)
(CONS '(a b) '(c d))      // ⇨ ((a b) c d)
(CONS (+ 1 2) '(+ 3))     // ⇨ (3 + 3)
(CONS '(a b c) NIL)       // ⇨ ((a b c))
(CONS NIL '(a b c))       // ⇨ (NIL a b c)
```

В случае, когда значением второго аргумента функции CONS является атом, результатом будет точечная пара:

```
(CONS 'a 'b)              // ⇨ (a.b)
(CAR '(a.b))              // ⇨ (a.b)
(CDR '(a.b))              // ⇨ nil
```

3.1.3 Функция создания LIST

Функция LIST строит новый список из произвольного количества переданных ей в качестве аргументов списков или атомов.

```
(LIST элемент_1 элемент_2 ... элемент_n)
   // ⇨ новый список
```

Значением функции LIST всегда будет список:

```
(LIST 'a 'b 'c)           // ⇨ (a b c)
(LIST 'a '(B C))          // ⇨ (a (b c))
(LIST 'a)                 // ⇨ (a)
```

3.1.4 Предикаты ATOM, EQ, EQL, EQUAL

Предикат – это функция, которая определяет, обладает ли аргумент определенным свойством, и возвращает в качестве значения NIL (ложное значение) или T (истинное значение).

Предикат ATOM – проверяет, является ли аргумент атомом:

```
(ATOM s-выражение)
```

Значением вызова ATOM будет T, если аргументом является атом, и NIL в противном случае.

```
(ATOM 'a)                 // ⇨ T
(ATOM '(a b c))           // ⇨ NIL
(ATOM NIL)                // ⇨ T
```

```
(ATOM '(NIL))              // ⇨ NIL
```

Предикат EQ сравнивает два символа и возвращает значение T, если они идентичны, и NIL в противном случае. С помощью EQ сравнивают только символы или константы.

```
(EQ 'a 'b)                 // ⇨ NIL
(EQ 'a (CAR '(a b c)))     // ⇨ T
(EQ NIL ())                // ⇨ T
```

Предикат EQL работает также как и EQ, но дополнительно позволяет сравнивать однотипные числа.

```
(EQL 2 2)                  // ⇨ T
(EQL 2.0 2.0)              // ⇨ T
(EQL 2 2.0)                // ⇨ NIL
```

Для сравнения чисел различных типов используют предикат «=». Значением предиката «=» является T в случае равенства чисел независимо от их типов и внешнего вида записи.

```
(= 2 2.0)                  // ⇨ T
```

Предикат EQUAL проверяет идентичность записей. Он работает также, как и EQL, но дополнительно проверяет одинаковость двух списков. Если внешняя структура двух объектов одинакова, то результатом EQUAL будет T.

```
(EQUAL 'a 'a)                          // ⇨ T
(EQUAL '(a b c) '(a b c))              // ⇨ T
(EQUAL '(a b c) '(CONS 'a '(b c)))     // ⇨ T
(EQUAL 1.0 1)                          // ⇨
NIL
```

Функция NULL проверяет список на пустоту.

```
(NULL '())                 // ⇨ T
(NULL NIL)                 // ⇨ T
(NULL '(1 2 3))            // ⇨ NIL
```

3.2. Задание к лабораторной работе

1) Запишите последовательности вызовов CAR и CDR, выделяющие из приведенных ниже списков символ «a». Упростите эти вызовы с помощью функций C...R.

- (1 2 3 a 4)
- (1 2 3 4 a)
- ((1) (2 3) (a 4))
- ((1) ((2 3 a) (4)))
- ((1) ((2 3 a 4)))
- (1 (2 ((3 4 (5 (6 a))))))

2) Каково значение каждого из следующих выражений?

- (ATOM (CAR (QUOTE ((1 2) 3 4))));
- (NULL (CDDR (QUOTE ((5 6) (7 8)))));
- (EQUAL (CAR (QUOTE ((7)))) (CDR (QUOTE (5 7))));
- (ZEROP (CADDDR (QUOTE (3 2 1 0))));

3) Составьте список студентов своей группы в виде (ФИО ФИО ... ФИО)

4) Для каждого студента

4.а) С помощью функции LIST составьте следующие списки:
Для самого студента - (дата рождения), (адрес), (средний бал по лекционным занятиям), (средний бал по практическим занятиям), (средний бал по лабораторным работам). Для отца и матери - (ФИО), (дата рождения), (адрес), (место работы).

б) с помощью функций CONS и SETQ объедините полученные списки и присвойте их в виде значений символам, означающим ФИО каждого студента:
ФИО ст. – (((дата рождения ст.) (адрес ст.) ((ср. балл (до десятых) по лекционным занятиям) (ср. балл по практическим

занятиям) (ср. бал по лабораторным работам))) (((ФИО отца) (дата рождения отца) (адрес) (место работы отца)) ((ФИО матери) (дата рождения матери) (адрес) (место работы матери)))).

5) Для произвольно выбранных студентов с помощью базовых функций сравните:

а) год рождения;
б) успеваемость (с учетом того, что число, характеризующее средний бал, может быть как целым, так и дробным);
в) выясните, не являются ли они родственниками;
г) выясните, живут ли они с родителями.

3.3. Контрольные вопросы

1) Назначение функции CAR и CDR.
2) Назовите отличия функций CONS и LIST.
3) Что такое предикат.
4) Отличия предикатов «eq», «eql» и «=».

4. ПОЛЬЗОВАТЕЛЬСКИЕ И АНОНИМНЫЕ ФУНКЦИИ

В данной работе мы рассмотрим описание анонимных функций (лямбд), именованных функций и способы их вызова.

4.1. Функции определяемые пользователем

Определение функций и их вычисление в Лиспе основано на лямбда-исчислении Черча. В Лиспе лямбда-выражение имеет вид

```
(LAMBDA (x1 x2 ... xn) fn)
```

, где (x1 x2 ... xn) – параметры, получаемые лямбда-выражением, а fn – тело выражения.

Символ LAMBDA означает, что мы имеем дело с определением функции. Символы x_i являются формальными параметрами

определения, которые принимают значения аргументов в описывающем вычисления теле функции fn. Входящий в состав формы список, образованный из параметров, называют лямбда-списком.

Телом функции является произвольная форма, значение которой может вычислить интерпретатор Лиспа. Например, простейшим лямбда-выражением может служить выражение, складывающее два числа:

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

Формальность параметров означает, что их можно заменить на любые другие символы, и это не отразится на вычислениях, определяемых функцией.

Лямбда-выражение – это определение вычислений и параметров функции в чистом виде без фактических параметров, или аргументов. Для того чтобы применить такую функцию к некоторым аргументам, нужно в вызове функции поставить лямбда-определение на место имени функции:

```
(лямбда-выражение a1 a2 ... an)
```

Здесь a_i – формы, задающие фактические параметры, которые вычисляются как обычно. Например,

```
((lambda (x y) (+ x y)) 1 2)    // ⇨ 3
```

Лямбда-вызовы можно свободно объединять между собой и другими формами. Вложенные лямбда-вызовы можно ставить как на место тела лямбда-выражения, так и на место фактических параметров.

```
((lambda (x)
   ((lambda (y) (list x y)) 'b)) 'a)   //
⇨ (a b) лямбда-вызов
```

Записывать вызовы функций полностью с помощью лямбда-вызовов не разумно, поскольку очень скоро выражения в вызове пришлось бы повторять, хотя разные вызовы одной функции отличаются лишь в части фактических параметров. Проблема разрешима путем именования лямбда-выражений и использования в вызове лишь имени.

Дать имя и определить новую функцию можно с помощью функции DEFUN:

```
(DEFUN имя лямбда-список тело)
```

DEFUN соединяет символ с лямбда-выражением, и символ начинает представлять определенные этим лямбда-выражением вычисления. Значением этой формы является имя новой функции.

После именования функции ее вызов осуществляется по имени и параметрам.

```
(defun list1 (x y)
   (cons x (cons y nil)))  // ⇨ list1
(list1 'c 'n)              // ⇨ (c n)
```

Специальная функция EVAL позволяет вычислить результат любого выражения языка LISP. Например,

```
(setq a 123)
(setq b 'a)
(eval 4.0)              // ⇨ 4.000000
(eval (abs -10))        // ⇨ 10
(eval a)                // ⇨ 123
(eval b)                // ⇨ 123
(set 'a '(+ 2 3))       // ⇨ (+ 2 3)
(eval a)                // ⇨ 5
```

4.2. Применяющие функционалы

Функции, которые позволяют вызывать другие функции, т.е. применять функциональный аргумент к его параметрам называют применяющими функционалами. Они дают возможность интерпретировать и преобразовывать данные в программу и применять ее в вычислениях.

Функция APPLY является функцией двух аргументов, из которых первый аргумент представляет собой функцию, которая применяется к элементам списка, составляющим второй аргумент функции APPLY:

```
(APPLY fn список-аргументов)
```

Например,

```
(SETQ a '+)             // ⇨ +
(APPLY a '(1 2 3))      // ⇨ 6
```

```
(APPLY '+ '(4 5 6))     // ⇨ 15
```

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

```
(FUNCALL fn x1 x2 ... xn)
```

Например,

```
(FUNCALL '+ 4 5 6)      // ⇨ 15
```

FUNCALL и APPLY позволяют задавать вычисления (функцию) произвольной формой, например, как в вызове функции, или символом, значением которого является функциональный объект. Таким образом, появляется возможность использовать синонимы имени функции. С другой стороны, имя функции можно использовать как обыкновенную переменную, например, для хранения другой функции (имени или лямбда-выражения), и эти два смысла (значение и определение) не будут мешать друг другу:

```
(SETQ list '+)          // ⇨ +
(FUNCALL list 1 2)      // ⇨ 3
(LIST 1 2)              // ⇨ (1 2)
```

4.3. Примеры функция на языке LISP

Рассмотрим несколько примеров пользовательских функций на языке LISP.

4.2.1 Функция BEGIN

Функция BEGIN будет возвращать начало списка без последнего элемента.

```
(defun begin
    (lambda (L)
        (cond
            ((NULL (CDR L)) nil)
            (T (CONS (car L) (BEGIN (CDR
L)))      )
        )
    )
```

```
)
```

4.2.2 Функция LEN

Функция LEN будет возвращать длину переданного в качестве параметра списка.

```
(defun LEN
    (lambda (L)
        (cond
            ((NULL L ) 0 )
            (T (+ 1 (LEN (CDR L)))      )
        )
    )
)
```

4.2.3 Функция PRINT-LIST

Функция PRINT-LIST распечатает поэлементно список.

```
(defun PRLR
    (lambda (L)
        (cond
            ((NULL  L) nil)
            (T (PRINT (car L)) (PRLR (CDR
L))      )
        )
    )
)
```

4.2.4 Функция CENTER

Функция CENTER находит среднее из трех чисел.

```
(DEFUN center (x y z)
    (COND
        ((> x y)
            (IF (< x z)
                (PROGN
                    (PRINT x)
                    (PRINT «среднее (1)»))
                (IF (> y z)
                    (PROGN
```

```
                              (PRINT y)
                              (TERPRI)
                              (PRINT «среднее
(2)»))
                         (PROGN
                              (PRIN1 z)
                              (PRIN1«среднее
(3)»)))))
          ((< x y)
               (IF (< y z)
                   (PROGN
                        (PRIN1 y)
                        (TERPRI)
                        (PRIN1 «среднее (4)»))
               (IF (> x z)
                   (PROGN
                        (PRINC x)
                        (PRINC «среднее (5)»))
                   (PROGN
                        (PRINC z)
                        (TERPRI)
                        (PRINC «среднее
(6)»)))))
     (T (PRINC «ошибка ввода»))))
```

4.4. Задание к лабораторной работе

1. Определите с помощью лямбда-выражения функцию, вычисляющую:

a) +y-x*y;
b) x*x+y*y;
c) x*y/(x+y)-5*y;

2. Определите функции (NULL x), (CADDR x) и (LIST x1 x2 x3) с помощью базовых функций. (Используйте имена NULL1, CADDR1 и LIST1, чтобы не переопределять одноименные встроенные функции системы.

3. Напишите функцию вычисления корней квадратного. Для вычисления дискриминанта должна использоваться отдельная функция.

4. Написать функцию, реализующую для десятичной дроби вывод либо ее целой части, либо десятичной части (задается через параметр функции).

5. Написать функцию сравнения длины двух списков. Если первый список больше второго, вывести сообщение на сколько больше и наоборот. Если списки равны, вывести сообщение об этом.

6. Написать свою функцию взятия модуля от числа. Предусмотреть возможность ввода некорректных значений.

7. Написать функцию, которая возвращает максимальное или минимальное значение из двух чисел (задается через параметр функции).

5. ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ И ПРЕДИКАТЫ

5.1. Функции консольного ввода/вывода

Функция чтения READ обрабатывает выражение целиком. Вызов функции осуществляется в виде

```
(READ)
```

Функция не показывает, что она ждет ввода выражения. Она лишь читает выражение и возвращает в качестве значения само это выражение, после чего вычисления продолжаются.

Если прочитанное выражение необходимо сохранить для дальнейшего использования, то вызов READ должен быть аргументом какой-нибудь формы, например присваивания (SETQ), которая свяжет полученное выражение:

```
(SETQ input (READ))
(+ 1 2)                    // введенное
выражение
(+ 1 2)                    // значение
input                      // ⇨ (+1 2)
```

Для вывода выражений используют несколько функций: PRINT, PRIN1, PRINC.

Функция PRINT с одним аргументом, сначала вычисляет значение аргумента, а затем выводит это значение на экран. Функция PRINT перед выводом аргумента переходит на новую

строку, а после него выводит пробел. Таким образом, значение выводится всегда на новую строку. Формальным результатом работы функции также является выведенное на экран значение.

```
_(PRINT (+ 1 2))
3                    // вывод произведенный
функцией
3                    // результат вычисления
функции
```

PRINT является псевдофункцией, у которой есть как побочный эффект, так и значение. Значением функции является значение ее аргумента, а побочным эффектом – печать этого значения.

Функции PRIN1 и PRINC работают так же, как PRINT, но не переходят на новую строку и не выводят пробел:

```
(PRIN1 5)                  // ⇨ 55
(PRINC 4)                  // ⇨ 44
```

Обеими функциями можно выводить кроме атомов и списков и другие типы данных которые мы рассмотрим позже:

```
(PRIN1 «CHG»)              // ⇨ «CHG»«CHG»
(PRINC «tfd»)              // ⇨ tfd«tfd»
;вывод без кавычек,
                           // результат -
значение аргумента
```

С помощью функция PRINC можно получить более приятный вид. Она выводит лисповские объекты в том же виде, как и PRIN1, но преобразует некоторые типы данных в более простую на вид форму.

Функция TERPRI переводит курсор на новую строку. У функции TERPRI нет аргументов и в качестве значения она возвращает NIL:

```
(DEFUN out (x y)
   (PRIN1 x) (PRINC y)
   (TERPRI) (PRINC (LIST 'x 'y))   // ⇨
out

(out 1 2)                          // ⇨ 12
```

```
(1 2)
```

5.2. Выражения изменяющие структуру

Основными функциями, изменяющими физическую структуру списков, являются RPLACA и RPLACD, которые уничтожают прежние и записывают новые значения в поля CAR и CDR списочной ячейки:

```
(RPLACA ячейка значение-поля)  // ⇨ поле
CAR
(RPLACD ячейка значение-поля)  // ⇨ поле
CDR
```

Первым аргументом является указатель на списочную ячейку, вторым – записываемое в нее новое значение поля CAR или CDR. Обе функции возвращают в качестве результата указатель на измененную списочную ячейку:

```
(SETQ a '(b c d))          // ⇨ (b c d)
(RPLACA a 'd)              // ⇨ (d c d)
(RPLACD a '(o n m))        // ⇨ (d o n m)
a                          // ⇨ (d o n m)
```

5.3. Задание к лабораторной работе

1. Используя композицию, напишите функции, которые осуществляют обращение списка из 2, 3, ... , n элементов.

2.1 Используя композицию описанных выше предикатов и логических связок, постройте функцию, которая проверяет, является ли ее аргумент:

a) списком из 2, 3, ... элементов;
b) списком из 2, 3, ... атомов;

2.2 Напишите функцию:

a) такую, что P(n) для произвольного целого n есть список из трех элементов, а именно: квадрата, куба и четвертой степени числа n;
b) для двух аргументов значением которой является список из двух элементов (разности и остатка от деления);
c) такую, что A(n) есть список (The answer is n). Так,

значением (A 12) будет (The answer is 12);

d) семи аргументов, значением которой служит сумма всех семи аргументов.

3. Для каждого из следующих условий определить функцию одного аргумента L , которая имеет значение T, если условие удовлетворяется, и NIL в противном случае:

a) n-ый элемент L есть 12;
b) n-ый элемент L есть атом;
c) L имеет не более n элементов (атомов или подсписков).

4. Напишите функцию, которая вводит фразу на естественном языке и преобразует ее в список.

5. Напишите функцию, которая спрашивает у пользователя ФИО студента из группы (список группы составлен раньше) и выдает следующие данные о нем:

a) год рождения;
b) средний бал;
c) родителей;
d) списки свойств, присвоенные ему раньше.

6. Напишите функцию:

a) от одного аргумента (ФИО любого студента), замещающую в списке с данными о нем (написанном раньше) подсписки со средними балами на списки свойств;
b) вычисляющую средние балы, беря данные из списков свойств.

7. Каковы будут значения выражений (RPLACA x x) и (RPLACD x x), если:

a) x = ′(a b);
b) x = ′(a);

8. Вычислите значение следующих выражений:

a) (RPLACD ‘(a) ‘b);
b) (RPLACA ‘(a) ‘b);
c) (RPLACD (CDDR ‘(a b x)) ‘c);
d) (RPLACD ‘(nil) nil)

6. УПРАВЛЯЮЩИЕ СТРУКТУРЫ

6.1. Конструкции LET и LET*

Предложение LET создает локальную связь внутри формы:

```
(LET ((m1 знач1) (m2 знач2)...)
   форма1 форма2 ...)
```

Вначале статические переменные m1, m2, ... связываются (одновременно) с соответствующими значениями знач1, знач2, ... Затем слева на право вычисляются значения форма1, форма2, ... Значение последней формы возвращается в качестве значения всей формы. После вычисления связи статических переменных ликвидируются.

Предложения LET можно делать вложенными одно в другое.

```
(LET ((x 'a) (y 'b))
   (LET ((z 'c)) (LIST x y z)))          //
⇨ (a b c)

(LET ((x (LET ((z 'a)) z)) (y 'b))
   (LIST x y))                           //
⇨ (a b)

(LET ((x 1) (y (+ x 1)))
   (LIST x y))                           //
⇨ Ошибка выполнения
```

При вычислении у «У» и «Х» еще нет связи. Значения переменным присваиваются одновременно. Это означает, что значения всех переменных mi вычисляются до того, как осуществляется связывание с формальными параметрами.

Подобной ошибки можно избежать с помощью формы LET*:

```
(LET* ((x 1) (y (+ x 1)))
   (LIST x y))                 // ⇨ (1 2)
```

6.2. Последовательные вычисления и ветвления

6.2.1 Последовательные вычисления

Предложения PROG1 и PROGN позволяют работать с несколькими вычисляемыми формами:

```
(PROG1 форма1 ... формаN)
(PROGN форма1 ... формаN)
```

Эти специальные формы последовательно вычисляют свои аргументы и в качестве значения возвращают значение первого (PROG1) или последнего (PROGN) аргумента.

```
(PROG1 (SETQ x 1) (SETQ y 5))       // ⇨ 1
(PROGN (SETQ j 8) (SETQ z (+x j))) // ⇨ 9
```

6.2.2 Оператор COND

Условное выражение COND:

```
(COND (p1 a1)
   ...
    (pn an))
```

Предикатами pi и результирующими выражениями ai могут быть произвольные формы. Выражения pi вычисляются последовательно до тех пор, пока не встретится выражение, значением которого является T. Вычисляется результирующее выражение, соответствующее этому предикату, и полученное значение возвращается в качестве значения всего предложения COND. Если истинного предиката нет, то значением COND будет NIL.

Рекомендуется в качестве последнего предиката использовать символ T. Тогда соответствующее ему an будет вычисляться в том случае, если другие условия не выполняются.

Если условию не ставится в соответствие результирующее выражение, то в качестве результата выдается само значение предиката. Если же условию соответствуют несколько форм, то при его истинности формы вычисляются последовательно слева направо и результатом предложения COND будет значение последней формы.

Предложения COND можно комбинировать. Например,

```
(COND
    ((> x 0)  (SETQ рез x))
    ((< x 0)  (SETQ x -x)  (SETQ рез x))
    ((= x 0))
    (T 'ошибка))
```

6.2.3 Оператор IF

Выражение IF записывается как

```
(IF условие то-форма иначе-форма)
```

Например,

```
(IF (> x 0) (SETQ y (+ y x)) (SETQ y (- y
x)))
```

Если выполняется условие (т.е. x>0), то к значению «y» прибавляется значение «x», иначе (x<0) от y отнимается отрицательное значение «x», т. е. прибавляется абсолютное его значение.

Когда у условного оператора отсутствует вычислений при ложном условии можно использовать сокращенную форму WHEN.

```
(WHEN условие форма1 форма2 ... )
```

Выбирающее предложение CASE:

```
(CASE ключ
    (список-ключей1 m11 m12 ... )
    (список-ключей2 m21 m22 ... )
    ...)
```

Сначала вычисляется значение ключевой формы – ключ. Затем его сравнивают с элементами списка-ключейi. Когда в списке найдено значение ключевой формы, начинают вычисляться соответствующие формы mi1, mi2, … . Значение последней формы возвращается в качестве значения всего предложения CASE.

Например,

```
(SETQ ключ 3)                          // ⇨ 3
(CASE ключ
    (1 'one)
    (2 '(one + one) 'two)
    (3 '(two + one) 'three)      // ⇨ three
```

6.3. Задание к лабораторной работе

1. Запишите следующие лямбда-вызовы с использованием формы LET и вычислите их на машине:

a) ((LAMBDA (x y) (LIST x y)
'(+ 1 2) 'c);
b) ((LAMBDA (x y) ((LAMBDA (z) (LIST x y z)) 'c)
'a 'b);
c) ((LAMBDA (x y) (LIST x y))
((LAMBDA (z) z) 'a)
'b).

2. Напишите с помощью композиции условных выражений функции от четырех аргументов AND4(x1 x2 x3 x4) и OR4(x1 x2 x3 x4), совпадающие с функциями AND и OR от четырех аргументов.

3. Пусть L1 и L2 - списки. Напишите функцию, которая возвращала бы T, если N-ые два элемента этих функций соответственно равны друг другу, и NIL в противном случае.

4. Написать условное выражение (используя COND), которое:

a) дает NIL, если L атом, и T в противном случае;
b) выдает для списка L, состоящего из трех элементов, первый из этих трех, который является атомом, или список, если в списке нет элементов атомов.

5. С помощью предложений COND или CASE определите функцию, которая возвращает в качестве значения столицу заданного аргументом государства.

6. Напишите с помощью условного предложения функцию, которая возвращает из трех числовых аргументов значение большего, меньшего по величине числа.

7. ЦИКЛИЧЕСКИЕ И РЕКУРСИВНЫЕ ВЫЧИСЛЕНИЯ

7.1. Циклические вычисления

Самым общим и мощным циклическим оператором в языке Лисп является оператор DO.

```
(DO ((var1 знач1 шаг1) (var2 знач2 шаг2)
...)
    (условие-окончания форма11 форма12 ...)
    форма21 форма22
    ...)
```

Первый аргумент описы- вает внутренние переменные

var1, var2, ..., их начальные значения – знач1, знач2, ... и формы обновления – шаг1, шаг2, ...

Вначале вычисления предложения DO внутренним переменным присваиваются начальные значения, если значения не присваиваются, то по умолчанию переменным присваивается NIL. Затем проверяется условие-окончания. Если оно действительно, то последовательно выполняются формы1i и значение последней из них возвращается в качестве значения всего предложения DO, иначе последовательно вычисляются формы2i.

На следующем цикле переменным vari одновременно присваиваются значения форм – шагi, вычисляемых в текущем контексте, проверяется условие-окончания и т.д.

```
(DO ((x 5 (+ x 1)) (y 8 (+ y 2)) (рез 0))
    ((< x 10) рез)
    (SETQ рез (+ рез x y))
```

Более частным случаем циклических функций является DOLIST.

```
(DOLIST (переменная_итератор (список)
результат_цикла)
    (
        (форма_1)
        (форма_2)
        (форма_n)
    )
```

В этом случае переменная итератор не имеет заданного шага, а по очереди принимает все значения из переданного списка. Пример:

```
(setq z 0)
(DOLIST (x '(1 2 3 4 5) z)
    (
        (print x)
        (setq z (+ z x))
    )
// ⇨ 1
// ⇨ 2
// ⇨ 3
```

```
// ⇨ 4
// ⇨ 5
// ⇨ 15
```

7.2. Безусловная передача правления

На Лиспе можно писать программы и в обычном операторном стиле с использованием передачи управления. Однако во многих системах не рекомендуется использовать эти предложения, так как их можно заменить другими предложениями (например DO) и, как правило, в более понятной форме. Но мы рассмотрим предложения передачи управления, хотя использовать их не следует.

```
(PROG (m1 m2 ... mn)
   оператор1
   оператор2
   ...
   операторm)
```

Перечисленные в начале формы переменные mi являются локальными статическими переменными формы, которые можно использовать для хранения промежуточных результатов. Если переменных нет, то на месте списка переменных нужно ставить NIL. Если какая-нибудь форма операторі является символом или целым числом, то это метка перехода. На такую метку можно передать управление оператором GO:

```
(GO метка)
```

GO не вычисляет значение своего «аргумента».

Кроме этого, в PROG-механизм входит оператор окончания вычисления и возврата значения:

```
(RETURN результат)
```

Операторы предложения PROG вычисляются слева направо (сверху вниз), пропуская метки перехода. Оператор RETURN прекращает выполнение предложения PROG и в качестве значения всего предложения возвращается значение аргумента оператора PROG. Если во время вычисления оператор RETURN не встретился, то значением PROG после вычисления его последнего оператора станет NIL .

После вычисления значения формы связи программных переменных исчезают.

7.3. Рекурсия. Различные формы рекурсии

Основная идея рекурсивного определения заключается в том, что функцию можно с помощью рекуррентных формул свести к некоторым начальным значениям, к ранее определенным функциям или к самой определяемой функции, но с более «простыми» аргументами. Вычисление такой функции заканчивается в тот момент, когда оно сводится к известным начальным значениям.

Рекурсивная процедура, во-первых, всегда содержит по крайней мере одну терминальную ветвь и условие окончания. Во-вторых, когда процедура доходит до рекурсивной ветви, то функционирующий процесс приостанавливается, и новый такой же процесс запускается сначала, но уже на новом уровне. Прерванный процесс каким-нибудь образом запоминается. Он будет ждать и начнет исполняться лишь после окончания нового процесса. В свою очередь, новый процесс может приостановиться, ожидать и т.д.

Будем говорить о рекурсии по значению и рекурсии по аргументам. В первом случае вызов является выражением, определяющим результат функции. Во втором – в качестве результата функции возвращается значение некоторой другой функции, и рекурсивный вызов участвует в вычислении аргументов этой функции. Аргументом рекурсивного вызова может быть вновь рекурсивный вызов и таких вызовов может быть много.

Рассмотрим следующие формы рекурсии:

- простая рекурсия;
- параллельная рекурсия;
- взаимная рекурсия.

Рекурсия называется простой, если вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обыкновенный цикл.

Для примера напишем функцию вычисления чисел Фибоначчи (F(1)=1; F(2)=1; F(n)=F(n-1)+F(n-2) при n>2):

```
(DEFUN FIB (N)
    (IF (> N 0)
        (IF (OR (= N 1) (= N 2)) 1
        (+ (FIB (- N 1)) (FIB (- N 2))))
    'ОШИБКА_ВВОДА))
```

Рекурсию называют па- раллельной, если она встреча-

ется одновременно в нескольких аргументах функции:

```
(DEFUN f ...
    ...(g ... (f ...) (f ...) ...)
    ...)
```

Рассмотрим использование параллельной рекурсии на примере преобразования списочной структуры в одноуровневый список:

```
(DEFUN FLAT (L)
    (COND
        ((NULL L) NIL)
        ((ATOM L) (CONS (CAR L) NIL))
        (T (APPEND
            (PREOBR (CAR L))
            (PREOBR (CDR L))))))
```

Рекурсия является взаимной между двумя и более функциями, если они вызывают друг друга:

```
(DEFUN f ...
    ...(g ...) ...)
(DEFUN g ...
    ...(f ...) ...)
```

Для примера напишем функцию обращения или зеркального отражения в виде двух взаимно рекурсивных функций следующим образом:

```
(DEFUN obr (l)
    (COND
        ((ATOM l) l)
        (T (per l nil))))

(DEFUN per (l res)
    (COND
        ((NULL l) res)
        (T (per
                (CDR l)
                (CONS (obr (CAR l))
```

```
res)))))
```

7.4. Задание к лабораторной работе

1) Запрограммируйте с помощью предложения DO функцию факториал.

2) Запишите с помощью предложения PROG функцию (аналог встроенной функции LENGTH), которая возвращает в качестве значения длину списка (количество элементов на верхнем уровне).

3) Используя функцию COND, напишите функцию, которая спрашивает у пользователя ФИО двух студентов из группы (список группы составлен раньше) для которых:

a) сравнивает год рождения и выдает результат (кто старше или что они ровесники);

b) сравнивает средний бал и выдает сообщение о результатах сравнения;

c) проверяет родственные связи (если одни и те же родители, то они родственники) и выдает об этом сообщение.

4) Напишите подобные функции, но уже используя функцию IF. Для двух последних заданий, вывод информации осуществить при помощи функций PRINT, PRIN1, PRINC.

Напишите рекурсивную функцию, определяющую сколько раз функция FIB вызывает сама себя. Очевидно, что FIB(1) и FIB(2) не вызывают функцию FIB.

5) Напишите функцию для вычисления полиномов Лежандра (P0(x)=1, P1(x)=x, Pn+1(x)= ((2*n+1)*x*Pn(x)-n*Pn-1(x))/(n+1) при n>1).

6) Напишите функцию:

вычисляющую число атомов на верхнем уровне списка (Для

списка (a в ((a) c) e) оно равно трем.);

определяющую число подсписков на верхнем уровне списка;

вычисляющую полное число подсписков, входящих в данный список на любом уровне.

7) Напишите функцию:

- от двух аргументов X и N, которая создает список из N раз повторенных элементов X;
- удаляющую повторные вхождения элементов в список;
- которая из данного списка строит список списков его элементов, например, (a b) ⇨ ((a) (b));
- вычисляющую максимальный уровень вложения подсписков в списке;
- единственным аргументом которой являлся бы список списков, объединяющую все эти списки в один;
- зависящую от трех аргументов X, N и V, добавляющую X на N-е место в список V.

8) Напишите функцию:

- аналогичную функции SUBST, но в которой третий аргумент W обязательно должен быть списком;
- которая должна производить замены X на Y только на верхнем уровне W;
- заменяющую Y на число, равное глубине вложения Y в W, например Y=A, W=((A B) A (C (A (A D)))) ⇨ ((2 B) 1 (C (3 (4 D))));
- аналогичную функции SUBST, но производящую взаимную замену X на Y, т. е. X ⇨ Y, Y ⇨ X.

8. МАКРОСЫ И ИХ ПРИМЕНЕНИЕ

8.1. Макросы

Программное формирование выражений наиболее естественно осуществляется с помощью макросов. Макросы дают возможность писать компактные, ориентированные на задачу программы, которые автоматически преобразуются в более сложный, но более близкий машине эффективный лисповский код. При наличии макросредств некоторые функции в языке могут быть определены в виде макрофункций. Такое определение фактически задает закон предварительного построения тела функции непосредственно перед фазой интерпретации.

Синтаксис определения макроса выглядит так же, как синтаксис используемой при определении функций формы DEFUN:

```
(DEFMACRO имя лямбда-список тело)
```

Вызов макроса совпадает по форме с вызовом функции, но его вычисление отличается от вычисления вызова функции. Первое отличие состоит в том, что в макросе не вычисляются аргументы. Тело макроса вычисляется с аргументами в том виде, как они записаны.

Второе отличие состоит в том, что интерпретация функций, определенных как макро, производится в два этапа. На первом, называемом макрорасширением, происходит формирование лямбда-определения функции в зависимости от текущего контекста, на втором осуществляется интерпретация созданного лямбда-выражения.

```
(DEFMACRO setqq (x y)
   (LIST 'SETQ x (LIST 'QUOTE y)))     //
⇨ setqq
(setqq a (b c))                        //
⇨ (b c)
a                                      //
⇨ (b c)
```

Макросы отличаются от функций и в отношении контекста вычислений. Во время расширения макроса доступны синтаксические связи из контекста определения. Вычисление же полученной в результате расширения формы производится вне контекста

макровызова, и поэтому статические связи из макроса не действуют. Использование макрофункций облегчает построение языка с лиспоподобной структурой, имеющего свой синтаксис, более удобный для пользователя. Чрезмерное использование макросредств затрудняет чтение и понимание программ.

8.2. Задание к лабораторной работе

1. Определите макрос (A-APPLY f x), который применяет каждую функцию f_i списка f = (f1 f2 ... fn) к соответствующему элементу xi списка x = (x1 x2 ... xn) и возвращает список, сформированный из результатов.
2. Определите в виде макроса функциональный предикат (КАЖДЫЙ пред список), который истинен в том и только в том случае, когда, являющийся функциональным аргументом предикат «пред» истинен для всех элементов списка список.
3. Определите в виде макроса функциональный предикат (НЕКОТОРЫЙ «пред» список), который истинен, когда предикат истинен хотя бы для одного элемента списка.
4. Определите функционал (MAPLIST fn список) для одного списочного аргумента.
5. Определите макрос, который возвращает свой вызов.
6. Определите лисповскую форму (IF условие p q) в виде макроса.

СПИСОК ЛИТЕРАТУРЫ

1. Городняя Л.В., Березин Н.А. Введение в программирование на Лиспе. – Интуит НОУ. – 2016. – URL: https://www.book.ru/book/917654

2. Фельдфебелев В. Е. Особенности языка Лисп. – М.: Лаборатория книги. – 2011. – URL: http://biblioclub.ru/index.php?page=book&id=140277&sr=1