МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт информационных технологий и управления в технических системах

Кафедра «Информационные технологии и компьютерные системы»

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Методические указания
к выполнению лабораторных работ
по дисциплине «Функциональное и логическое программирование»
для студентов направления подготовки
09.03.01 «Информатика и вычислительная техника»

Севастополь
СевГУ
2019

УДК 004.056.5

Р е ц е н з е н т:
С.Г. Лелеков – доцент кафедры «Информационные технологии и компьютерные системы», канд. физ.-мат. наук, доцент

Составители: Брюховецкий А.А., Ткаченко К.С.

Методические указания для выполнения лабораторных работ по дисциплине «Функциональное и логическое программирование» для бакалавров по направлению 09.03.01 «Информатика и вычислительная техника» / Сост. А.А. Брюховецкий, К.С. Ткаченко. – Электрон. дан. – Севастополь: СевГУ, 2019 г. – Режим доступа: свободный после авторизации. – Загл. с экрана. – 35 с.

Целью методических указаний является оказание методической помощи бакалаврам при выполнении лабораторных работ по дисциплине «Функциональное и логическое программирование».

Методические указания содержат:
- краткие теоретические сведения;
- постановку задачи для выполнения работы;
- содержание отчета;
- список литературы;
- индивидуальные задания для выполнения работы.

УДК 004.056.5

Методические указания рассмотрены и утверждены на заседании кафедры «Информационные технологии и компьютерные системы» (протокол № 4 от 23 января 2019 г.)

Текстовое (символьное) издание (1,5 Мб).

Системные требования: Intel, 3,4 GHz; 150 Мб; Windows XP/Vista/7; DVD-ROM; 1 Гб свободного места на жестком диске; программа для чтения pdf-файлов: Adobe Acrobat Reader, Foxit Reader.

Дата подписания к использованию: 23.01.2019.

РИИЦМ СевГУ. 299015, г. Севастополь, ул. Курчатова, 7.

СОДЕРЖАНИЕ

Введение

Символьные и алгебраические вычисления представляют собой одну из основных областей применения символьной обработки. В рамках этого направления разрабатываются специальные языки для решения интеллектуальных задач, в которых традиционно упор делается на преобладание логической и символьной обработки над вычислительными процедурами. Достаточно популярно также создание так называемых пустых экспертных систем или оболочек, базы знаний которых можно наполнять конкретными знаниями, создавая различные прикладные системы.

Идентификация цепочек символов входит как составная часть во многие задачи, связанные с редактированием текстов, поиском данных и символьной обработкой. Множество образов часто является регулярным множеством, заданным регулярным выражением. В настоящей главе мы обсудим несколько приемов решения такого рода задач идентификации цепочек. Инструментарий, применяемый для разработки экспертных систем, может быть разделен на две категории: высокоуровневые языки символьной обработки и экспертные системы широкого профиля, или оболочки. К числу наиболее распространенных высокоуровневых языков обработки символьных данных относятся ЛИСП и ПРОЛОГ.

С ростом количества знаний и возможностей, заложенных в различных системах машинного проектирования и автоматизированного управления, возникла потребность в применении методов технологии знаний и символьной обработки в этих системах. В машинном проектировании, ориентированном на знания, предполагается использование глубоких профессиональных знаний в нескольких областях, однако независимо от содержательной части для работы в каждой из этих областей можно применять методы символьной обработки.

Common Lisp (сокращённо — CL) — диалект языка программирования Лисп, стандартизированный ANSI[1]. Был разработан с целью объединения разрозненных на момент начала 1980-х годов диалектов Лиспа; доступно несколько реализаций Common Lisp, как коммерческих, так и свободно распространяемых.

Стандарт фиксирует язык как мультипарадигменный: поддерживается комбинация процедурного, функционального и объектно-ориентированного программирования. В частности, объектно-ориентированное программирование обеспечивается входящей в язык системой CLOS; а система лисп-макросов позволяет вводить в язык новые синтаксические конструкции, использовать техники метапрограммирования и обобщённого программирования.

Термин S-выражение или sexp (для символического выражения) относится к соглашению о способе записи полуструктурированных данных в доступной для человеческого понимания текстовой форме. Символические выражения создаются, в основном, из символов и списков. S-выражения наиболее известны благодаря их использованию в языках программирования семейства Лисп. Также S-выражения применяют в языках-наследниках Лиспа, таких как DSSSL, и в разметке коммуникационных протоколов вроде IMAP и CBCL Джона Маккарти. Детали синтаксиса и поддерживаемых типов данных отличаются в различных языках, но общая особенность — использование S-выражений как префиксной нотации с использованием скобок (известных как кембриджская польская нотация).

S-выражения используются в Лиспе как для кода, так и для данных. S-выражения были первоначально предназначены только для представления данных, которыми должны были манипулировать M-выражения, но первая реализация Лиспа была интерпретатором S-выражений, в которые планировалось переводить M-выражения, и программисты Lisp вскоре привыкли к использованию S-выражений как для данных, так и для кода.

S-выражения могут быть как отдельными объектами (атомами), такими как числа, Символ (Lisp), включая специальные символы nil и t, или точечными парами, в виде (x.y). Более длинные списки, состоящие из вложенных точечных пар, например (1 . (2 . (3 . nil))), можно написать более привычным способом, как (1 2 3). Вложенные списки также могут

быть записаны в виде S-выражений: ((молоко сок) (мёд мармелад)). S-выражения не зависят от пробелов и разрывов строк, пробелы используются только в качестве разграничителей между атомами.

S-выражения в Лиспе читаются с помощью функции READ. Эта функция читает текстовое представление S-выражения и возвращает Lisp-данные. Функция PRINT может быть использована для вывода S-выражения. То, что возвращает PRINT, можно прочитать с помощью функции READ при условии, что все выводимые объекты данных имеют представление для ввода-вывода. Lisp имеет такое представление для чисел, строк, символов, списков и ещё многих типов данных. Программный код может быть представлен в виде аккуратно форматированного (pretty printed) S-выражения с помощью функции PPRINT.

Lisp программы — это корректные S-выражения, но не все S-выражения являются правильными программами на Lisp. (1.0 + 3.1) — это корректное S-выражение, но не корректная Lisp-программа, Lisp использует префиксную нотацию, поэтому число с плавающей точкой (1.0) не может быть распознано как операция (первый элемент выражения).

Лабораторная работа № 1. Базовые сведения для интерактивной работы в среде Лисп

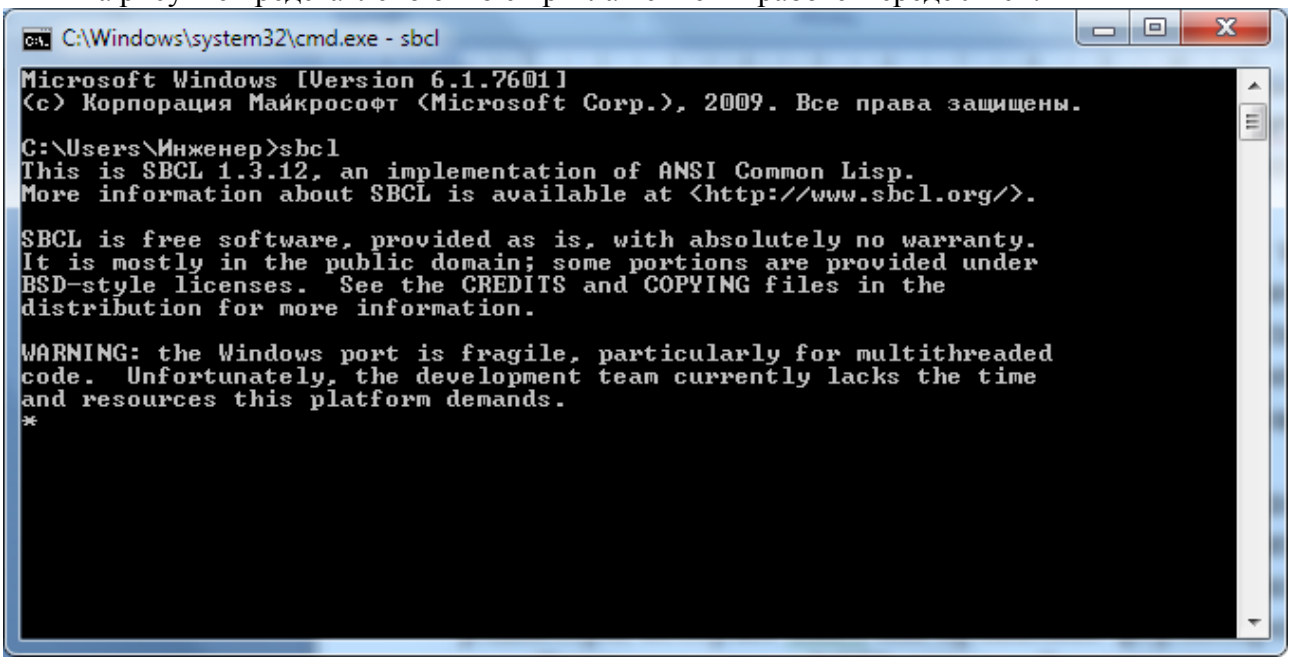
Цель работы: изучение технологии практической работы в среде Лисп (Steel Bank Common Lisp — свободная реализация языка программирования Common Lisp) и правил записи функциональных зависимостей. Материалы по изучению LISP доступны по ссылкам https://www.cliki.net/ , http://www.sbcl.org/manual/index.html

Интерактивное программирование

При запуске SBCL, вы должны увидеть приглашение, которое может выглядеть примерно так :

*

На рисунке представлено окно с приглашением к работе в среде Лисп.



Это приглашение Lisp. Как и приглашение оболочки DOS или UNIX, приглашение Lisp — это место, куда вы можете вводить выражения, которые заставляют что-либо делать

компьютер. Однако вместо того, чтобы считывать и выполнять строку команд оболочки, Lisp считывает Lisp выражения, вычисляет их согласно правилам Lisp и печатает результат. Потом он (Lisp) повторяет свои действия со следующим введенным вами выражением. Вот вам бесконечный цикл: считывания, вычисления, и печати (вывода на экран), поэтому он называется цикл-чтение-вычисление-печать (по-английски read-eval-print-loop), или сокращённо REPL . Этот процесс может также называться top-level, top-level listener, или Lisp listener.

Через окружение, предоставленное REPL'ом, вы можете определять и переопределять элементы программ такие как переменные, функции, классы и методы; вычислять выражения Lisp; загружать файлы, содержащие исходные тексты Lisp или скомпилированные программы; компилировать целые файлы или отдельные функции; входить в отладчик; пошагово выполнять программы; и проверять состояние отдельных объектов Lisp.

Все эти возможности встроены в язык, и доступны через функции, определённые в стандарте языка. Если вы захотите, вы можете построить достаточно приемлемую среду разработки только из REPL и текстового редактора, который знает как правильно форматировать код Lisp.

Для знакомства с REPL, вам необходимо выражение Lisp, которое может быть прочитано, вычислено и выведено на экран. Простейшее выражение Lisp - это число. Если вы наберете 10 в приглашении Lisp и нажмете ВВОД, то сможете увидите что-то наподобие:

```
* 10
10
```

Первая 10 - это то, что вы набрали. Считыватель Lisp, R в REPL, считывает текст "10" и создаёт объект Lisp, представляющий число 10. Этот объект - самовычисляемый объект, это означает, что такой объект при передаче в вычислитель, E в REPL, вычисляется сам в себя. Это значение подаётся на устройство вывода REPL, которое напечатает объект "10" в отдельной строке. Хотя это и похоже на сизифов труд, можно получить что-то поинтереснее, если дать интерпретатору Lisp пищу для размышлений. Например, вы можете набрать (+ 2 3) в приглашении Lisp:

```
* (+ 2 3)
5
```

Все что в скобках - это список, в данном случае список из трех элементов: символ +, и числа 2 и 3. Lisp, в общем случае, вычисляет списки, считая первый элемент именем функции, а остальные - выражениями для вычисления и передачи в качестве аргументов этой функции. В нашем случае, символ «+» - название функции которая вычисляет сумму. 2 и 3 вычисляются сами в себя и передаются в функцию суммирования, которая возвращает 5. Значение 5 отправляется на устройство вывода, которое отображает его. Lisp может вычислять выражения и другими способами, например:

"Здравствуй, мир" .

Нет законченной книги по программированию без программы "Здравствуй, мир"("hello, world."). После того как интерпретатор запущен, нет ничего проще чем набрать строку "Здравствуй, мир".

```
* "Здравствуй, мир"
```

"Здравствуй, мир"

Это работает, поскольку строки, также как и числа, имеют символьный синтаксис, понимаемый считывателем Lisp, и являются самовычисляемыми объектами: Lisp считывает строку в двойных кавычках и создает в памяти строковой объект, который при вычислении вычисляется сам в себя и потом печатается в том же символьном представлении. Кавычки не являются частью строкового объекта в памяти - это просто синтаксис, который позволяет считывателю определить, что этот объект - строка. Устройство вывода REPL напечатает кавычки тоже, потому что оно пытается выводить объекты в таком же виде, в каком понимает их считыватель.

Однако, наш пример не может квалифицироваться как программа "Здравствуй мир". Это, скорее, значение "Здравствуй мир".

Вы можете сделать шаг к настоящей программе, напечатав код, который, в качестве побочного эффекта, отправит на стандартный вывод строку "Здравствуй, мир". Common Lisp предоставляет несколько путей для вывода данных, но самый гибкий - это функция FORMAT. FORMAT получает переменное количество параметров, но только два из них обязательны: указание, куда осуществлять вывод, и строка для вывода. В следующей главе Вы увидите, как строка может содержать встроенные директивы, которые позволяют вставлять в строку последующие параметры функции (а-ля printf или строка % из Python). До тех пор, пока строка не содержит символа ~, она будет выводиться как есть. Если вы передадите t в качестве первого параметра, функция FORMAT направит отформатированную строку на стандартный вывод. Итак, выражение FORMAT для печати "Здравствуй, мир" выглядит примерно так:

```
* (format t "Здравствуй, мир")
Здравствуй, мир
NIL
```

Стоит заметить, что результатом выражения FORMAT является NIL в строке после вывода "Здравствуй, мир". Этот NIL является результатом вычисления выражения FORMAT, напечатанного REPL. (NIL – это Lisp-версия false и/или null). В отличие от других выражений, рассмотренных ранее, нас больше интересует побочный эффект выражения FORMAT (в данном случае, печать на стандартный вывод), чем возвращаемое им значение. Но каждое выражение в Lisp вычисляется в некоторый результат.

Однако, до сих пор остается спорным, написали ли мы настоящую программу. Вы видите восходящий стиль программирования, поддерживаемый REPL: вы можете экспериментировать с различными подходами и строить решения из уже протестированных частей. Теперь, когда у вас есть простое выражение, которое делает то, что вы хотите, нужно просто упаковать его в функцию. Функции являются одним из основных строительных материалов в Lisp и могут быть определены с помощью выражения DEFUN подобным образом:

```
* (defun hello-world () (format t "hello, world"))
HELLO-WORLD
```

Выражение hello-world, следующее за DEFUN, является именем функции. Далее мы рассмотрим, какие именно символы могут использоваться в именах, но сейчас будет достаточно сказать, что многие символы, такие как <<-", недопустимые в именах в других языках можно использовать Common Lisp. Это стандартный стиль Lisp – "not to mention more in line with normal English typography" – формирование составных имен с помощью дефисов, как в hello-world, вместо использования знаков подчеркивания, как в hello_world, или

использованием заглавных букв внутри имени, как helloWorld. Скобки () после имени отделяют список параметров, который в данном случае пуст, так как функция не принимает аргументов. Остальное - это тело функции.

В какой-то мере это выражение подобно всем другим, которые вы видели, всего лишь еще одно выражение для чтения, вычисления и печати, осуществляемых REPL. Возвращаемое значение в этом случае - это имя только что определенной функции. Но, подобно выражению FORMAT, это выражение более интересно своими побочными эффектами, нежели возвращаемым значением. Однако, в отличие от выражения FORMAT, побочные эффекты невидимы: после вычисления этого выражения создается новая функция, не принимающая аргументов, с телом (format t "hello, world") и ей дается имя HELLO-WORLD.

Теперь, после определения функции, вы можете вызвать ее следующим образом:

```
* (hello-world)
hello, world
NIL
```

Вы можете видеть, что вывод в точности такой же, как при вычислении выражения FORMAT напрямую, включая значение NIL, напечатанное REPL. Функции в Common Lisp автоматически возвращают значение последнего вычисленного выражения.

Краткие теоретические сведения

Функциональная программа

Функциональная программа состоит из совокупности определений функций. Функции, в свою очередь, представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов. Вычисления начинаются с вызова некоторой функции. Она в свою очередь вызывает функции, входящие в ее определение и т.д. в соответствии с иерархией определений и структурой условных предложений.

Функции часто либо прямо, либо опосредованно вызывают сами себя. Каждый вызов возвращает некоторое значение в вызвавшую его функцию, вычисление которой после этого продолжается. Этот процесс повторяется до тех пор, пока запустившая вычисления функция не вернет конечный результат пользователю.

Чистое функциональное программирование не признает присваивания и передач управления. Разветвление вычислений основано на механизме обработки аргументов условного предложения.

Повторные вычисления осуществляются через рекурсию. Она является основным средством функционального программирования.

Понятие функции и формы ее записи

Функцией в математике называется отображение, которое однозначно отображает одни значения на другие.

Например, в выражении

$$y = f(x),$$

любому *x* из области определения функции ставится в соответствие единственное значение *y* из области допустимых значений функции.

В математике и обычных языках программирования вызов функции записывается в, так называемой, *префиксной нотации* (записи). В ней имя функции стоит перед круглыми скобками, окружающими аргументы. Например: $f(x, y)$, $f(z)$.

В арифметических выражениях используется также инфиксная запись, где имя функции или действие располагается между аргументами: $x + y$, $x * (y + z)$.

В функциональном программировании как для вызова функции, так и для записи выражений принята *единообразная* префиксная форма записи. В этой записи как имя функции (действие), так и сами аргументы записываются внутри скобок. Таким же образом записываются и арифметические действия.

Пример. Выражение $x + y$ записывается как: $(+\ x\ y)$.

Пример. Записать выражение: *sin(x-3)+7* примет вид: *(+ (sin (- x 3)) 7).*

Диалог с интерпретатором Лиспа

Транслятор **Лиспа** работает как правило в режиме интерпретатора.

Read-eval-print цикл:

loop { read
evaluate
print}

В **Лиспе** сразу читается , затем вычисляется (*evaluate*) значение функции и выдается значение.

Функция QUOTE

В некоторых случаях не требуется вычисления значений выражений, а требуется само выражение. Если прямо ввести *(+ 2 3)* , то *5* получится как значение. Но можно понимать *(+ 2 3)* не как функцию, а как список. *S*-выражения, которые не надо вычислять, помечают для интерпретатора апострофом " ' " (*quote*).

Пример.

' (+ 2 3)

Ответ интерпретатора: (+ 2 3)

Пример.

(QUOTE *y*)

Ответ интерпретатора: *y*

Использование символов в качестве переменных

Изначально символы в Лиспе не имеют значения. Значения имеют только константы. Для связывания символов используются функции *SET, SETQ* и *SETF*.

Функция *SET* связывает символ со значением, предварительно вычисляя значения аргументов. В качестве значения *SET* возвращает значение второго аргумента.

Пример.

(SET 'd '(x y z))

Ответ интерпретатора: *(x y z)*

d

Ответ интерпретатора: *(x y z)*

Пример.

(SET 'a 'b)

Ответ интерпретатора: *b*

a

Ответ интерпретатора: *b*

Пример. Здесь вычисляется последовательность операторов: a = 1; b = a+2.

(SET 'a 1)

Ответ интерпретатора: 1

(SET 'b (+ a 2))

Ответ интерпретатора: 3

b

Ответ интерпретатора: 3

Если перед первым аргументом нет апострофа, то значение будет присвоено значению этого аргумента.

Пример.

(set 'a 'b)

Ответ интерпретатора: *b*

(set a 'e)
Ответ интерпретатора: *e*
a
Ответ интерпретатора: *b*
b
Ответ интерпретатора: *e*

Функция *SETQ* аналогична *SET*, но не вычисляет значения первого аргумента. Буква *Q* в имени *SETQ* означает блокировку.

Пример.
(setq m 'k)
Ответ интерпретатора: *k*
m
Ответ интерпретатора: *k*

Пример.
(setq x 1)
Ответ интерпретатора: 1
x
Ответ интерпретатора: 1
(sin x)
Ответ интерпретатора: 0.841471

Арифметические функции

Арифметические функции могут быть использованы с целыми или действительными аргументами.

Число аргументов для большинства арифметических функций может быть разным.
(+ x1 x2 ... xn) возвращает x1 + x2 + x3 + ... + xn.
(- x1 x2 ... xn) возвращает x1 - x2 - x3 - ... - xn.
(y1 y2 ... yn)* возвращает y1 x y2 * y3 * ... * yn.
(/ x1 x2 ... xn) возвращает x1/x2/... /xn.

Примеры.
(+ 5 7 4) ⇒ 16
(- 10 3 4 1) ⇒ 2
(/ 15 3) ⇒ 5

Сравнение чисел:
(= число числа) ⇒ равны (все)
(< число числа) ⇒ меньше (для всех)
(> число числа) ⇒ больше (для всех)
и т. д.

В языке Лисп как для вызова функций, так и для записи выражения принята единообразная префиксная форма записи, при которой как имя функции или действия, так и сами аргументы записываются внутри скобок:
(f x), (g x y), (h x (g y z)) и т. д.

Примечание:
Комплексные числа (тип complex) представляются в алгебраической форме, с действительной и мнимой частями, каждая из которых является не комплексным числом (целым, дробным, или с плавающей точкой). Комплексные числа могут быть обозначены с помощью записи символа #C с последующим списком действительной и мнимой частей. Если две части не принадлежат одному типу, тогда они будут преобразованы в соответствие с правилами преобразования чисел с плавающей точкой.

Задание

Вычислить на Лиспе значение заданного выражения.

Пример выполнения задания

Пусть задана зависимость: $y = 5x^3 + \sin^2(3x) + 2$. Значение переменной *x* задать при помощи *SET* или *SETQ*.

Программа на Лиспе приняла вид:

(setq x 1)

Ответ интерпретатора: 1

(setq y (+ (5 x x x) (* (sin (* 3 x)) (sin (* 3 x))) 2))*

Ответ интерпретатора: 7.01991.

Варианты заданий

1. $y = 7x^4 + \sin^3(2x+1) + 4$	6. $y = 2x^2(x-1) + 7\sin^2(x/3)$
2. $y = -3x^2 + \sin^4(\mathrm{x}/2) - 5$	7. $y = 5(x-3)(x^2-1) + x\cos^2(x)$
3. $z = 8x^5 - 2\cos^2(4x+3) + 1$	8. $y = 11(x-4)(x+5) + \sin^4(x)$
4. $t = 6x^4 - 3\sin^3(1-3x) + 7$	9. $v = (x^2-3)(\sin(4x)) + 5x^2$
5. $p = -9x^3 + 6\cos^2(3x/2) + 2$	10. $y = 18(1-2x)x^3(x+1) + \cos(3x)$
11. $y = \dfrac{5\cos(2x) + x(x+1)}{3x^2 - 1}$	20. $y = \dfrac{\cos^2(2x/3) + x^2(1-x)}{2x+1}$
12. $y = \dfrac{5x\sin(3x) + x(x^2+1)}{3x^3 - 1}$	21. $y = \dfrac{\sin^2(2/3x) - 4x(1-6x)}{-6x+1}$
13. $y = \dfrac{x^2\cos(2x/3) + x(2x-1)}{7x+1}$	22. $y = \dfrac{\cos^3(x) + x^2(1-x)}{9x+4}$
14. $y = \dfrac{5\cos(2x) + x(4x+1)}{x^2 - \cos(x)}$	23. $y = \dfrac{\cos(3x/4)/x + 2x/(1-5x)}{2x+1}$
15. $y = \dfrac{\sin(x/2)/3 + 2x(x-5)}{4x-1}$	24. $y = \dfrac{x\cos^2(2x/5) + 2x(1-3x)}{1+7x}$
16. $y = 4x^2/(1-x) + \sin^2(2x) + 6$	25. $y = 4(x-3)/x^2 + \sin^2(4x) + 7$
17. $y = 3x^3/(2-x) + \sin^2(4x) + 5$	26. $y = 9(x-7)/2x^2 + \cos^2(2x-5)$
18. $y = 5x/(1-x^2) + \cos^2(5x-1)$	27. $y = 5(x^2+6)/3x + \sin^2(4-3x)$
19. $y = 7x^4/(3-x) + \sin(4x) + 13$	28. $y = 3(7-x^2)/4x^2 + \sin^2(4-x^2)$

Контрольные вопросы

1. Что такое функциональная программа.
2. Приведите примеры формы записи функции.
3. Назначение функции QUOTE.
4. Назначение функции SET.
5. Назначение функции SETQ.
6. Разработать программу вычисления заданной преподавателем функциональной зависимости на языке Лисп.

Содержание отчета

Отчет о проделанной работе должен содержать:
Титульный лист.
1.Цель работы.
2.Постановка задачи.
3.Вариант индивидуального задания.
4.Описание программы.
5.Результаты выполнения программы.
Выводы.

Лабораторная работа № 2. Разработка и исследование функций обработки арифметических выражений на Лиспе

Цель работы: изучение правил создания и вызова пользовательских функций на Лиспе.

Краткие теоретические сведения.

Определение функций

Функцию можно определить самим и использовать как встроенную. Для определения функции необходимо:
1) дать имя функции;
2) определить параметры функции;
3) определить действия, выполняемые функцией.

Для задания новых функций в Лиспе используется специальная форма *defun*:

(defun < имя функции > < параметры > < тело функции >)

Имя функции должно являться символом. Параметры представляют собой список аргументов, разделенных пробелами.

Пример 1. Написать функцию, складывающую сумму двух чисел.

(defun sum (a b) (+ a b))
Ответ интерпретатора: *sum*

В рассмотренном примере: *sum* – имя функции, *a* и *b* – ее параметры, *(+ a b)* – тело функции.

Вызов функции *sum* для параметров *1* и *2* осуществляется следующим образом:

(sum 1 2)
Ответ интерпретатора: *3*

Пример 2. Вызвать функцию *sum* для вычисления суммы чисел *x* и *y*.

(setq x 4)
Ответ интерпретатора: *4*
(setq y 5)
Ответ интерпретатора: *5*
(sum x y)
Ответ интерпретатора: *9*

Передача параметров. Локальные переменные

В Лиспе передача параметров производится в функцию по значению, т.е. формальный параметр в функции связывается с тем же значением, что и значение фактического параметра.

Изменение значения формального параметра не оказывает влияния на значения фактических параметров. После вычисления функции, созданные на это время связи параметров ликвидируются и происходит возврат к тому состоянию, которое было до вызова функции. Параметры функции являются локальными переменными, и имеют значение только внутри функции.

Пример 3.

(defun f (x) (setq x 'new))
Ответ интерпретатора: *f*
(setq x 'old)
Ответ интерпретатора: *old*
x
Ответ интерпретатора: *old*
(f x)
Ответ интерпретатора: *new*

Пример 4.

(defun double (num) (num 2)*
Ответ интерпретатора: *double*
(setq num 5)
Ответ интерпретатора: *5*
(double 2)
Ответ интерпретатора: *4*
num
Ответ интерпретатора: *5*

Свободные переменные

Если в теле функции есть переменные, не входящие в число ее формальных параметров - они называются свободными. Значения свободных переменных остается в силе после ее выполнения.

Пример 5.

(defun f1 (y) (setq x 3))
Ответ интерпретатора: *f1*
(f1 5)
Ответ интерпретатора: *3*
x
Ответ интерпретатора: *3*

Пример выполнения задания

Вычислить зависимость $y = \frac{x^3 + (1+x)^3}{3x^3 + (1+2x+x^2)^3}$ на основе функции $f(x) = x^3 = x \cdot x \cdot x$. Значение переменной *x* задать при помощи *SET* или *SETQ*.

Программа на Лиспе приняла вид:

(defun cub(a) (a a a))*
Ответ интерпретатора: *defun*
(setq x 1)
Ответ интерпретатора: *1*
(setq y (/ (+ (cub x) (cub (+ x 1))) (cub (+ 1 (2 x) (* x x)))))*
Ответ интерпретатора: *9/64*
y
Ответ интерпретатора: *9/64*

В CL есть специальный оператор function, аргументом которого может быть символ или не вычисляемая форма вида (lambda ...).
Результатом вычисления function является функция-объект, к которой можно применить apply или funcall.
#' — стандартный макрос Lisp считывателя (reader macro), который раскрывается в (function ...).
Поэтому #'(lambda (x) (expr x)) есть ни что иное, как (function (lambda (x) (expr x))).
Также стандартом определён макрос lambda, который раскрывается в (function (lambda ...)) или кратко #'(lambda ...).

Строго говоря, следующие выражения семантически равнозначны:
(lambda (x) (expr x))
#'(lambda (x) (expr x))
(function (lambda (x) (expr x)))
Запись вида #'(lambda (x) (expr x)) симметрична записи #'f.
Один подход:

```
(let ((xs (list 1 2 3 4 5 6)))
  (mapcar #'(lambda (x)
              (+ x 2))
          (remove-if #'oddp xs)))
```

Другой подход:

```
(let ((xs (list 1 2 3 4 5)))
  (mapcar (lambda (x)
            (+ x 2))
          (remove-if (lambda-match
                       (1 nil)
                       (2 nil)
                       (x  x))
                     xs)))
```

Варианты заданий

Вычислить заданную зависимость $y(x)$ на основе функции $f(x)$. Функция $f(x)$ должна быть определена с помощью *defun*.

Номер варианта	Зависимость $y(x)$	Функция $f(x)$
1	$y=\frac{\sin(2x+1)+\sin(3x+0{,}5)}{1+\sin\frac{\pi}{8}}$	$f(a,b,x)=\sin(ax+b)$
2	$y=4\sin x+\frac{5\sin x+3\cos x}{8\sin x+4\cos x}$	$f(a,b,x)=a\sin x+b\cos x$
3	$y=\frac{2{,}5x^2+3{,}4x+8{,}1}{3{,}6x^2-1{,}8x-5{,}2}$	$f(a,b,c,x)=ax^2+bx+c$
4	$y=\frac{2+\cos\frac{3\pi}{5}+\cos\frac{4\pi}{7}}{8+\cos\frac{3\pi}{7}+\cos\frac{5\pi}{11}}$	$f(m,n)=\cos\frac{\pi\cdot m}{n}$
5	$y=\frac{\sin\frac{4\pi}{3}+7\sin\frac{5\pi}{2}}{5\sin\frac{8\pi}{5}+\sin\frac{7\pi}{3}}$	$f(n,k)=\sin\frac{\pi\cdot n}{k}$
6	$y=\frac{2tg\frac{x}{2}+3ctg\frac{x}{2}}{4+tg\frac{x}{4}}$	$tgx=\frac{\sin(x)}{\cos(x)}$

7	$y=\frac{3,7x^3+2,4x+6,1}{3,9x^3-1,8x-9,2}$	$f(a,b,c,x)=ax^3+bx+c$
8	$y=\frac{1,5x^4+3,4x^2+8,1}{5,6x^4-1,8x^2-5,1}$	$f(a,b,c,x)=ax^4+bx^2+c$
9	$y=\frac{0,5x^3+1,4x^2+0,1}{3,8x^3-1,7x^2+4,9}$	$f(a,b,c,x)=ax^3+bx^2+c$
10	$y=\frac{\sin(3x+2)+\sin(3x+1)}{1-\sin\frac{\pi}{4}}$	$f(a,b,x)=\sin(ax-b)$
11	$y=4\cos 2x+\frac{4\sin x+7\cos x}{5\sin 3x-9\cos x}$	$f(a,b,x)=a\sin x-b\cos x$
12	$y=\frac{2,2x^2-3,1x+8,6}{3,9x^2+1,8x-5,7}$	$f(a,b,c,x)=ax^2-bx-c$
13	$y=\frac{1+\cos\left(\frac{3\pi}{7}+1\right)+2\cos\frac{4\pi}{7}}{11+\cos\frac{3\pi}{7}+3\cos\left(\frac{5\pi}{9}-0,5\right)}$	$f(m,n)=\cos\left(\frac{\pi\cdot m}{n}+k\right)$
14	$y=\frac{9\sin\frac{4\pi}{11}-6\sin\left(\frac{5\pi}{2}-2\right)}{5\sin\left(\frac{8\pi}{5}+0,2\right)+2\sin\frac{13\pi}{3}+0,8}$	$f(n,k)=\sin\left(\frac{\pi\cdot n}{k}+m\right)$
15	$y=\frac{4,2x^3+2,7x-6,9}{3,9x^3-1,8x+7,3}$	$f(a,b,c,x)=ax^3-bx+c$
16	$y=\frac{9,4x^4-2,8x^2+3,1}{6,6x^4+2,3x^2-5,4}$	$f(a,b,c,x)=ax^4+bx^2-c$
17	$y=\frac{0,7x^3+1,3x^2+0,6}{4,5x^3-8,7x^2+4,1}$	$f(a,b,c,x)=ax^3-bx^2-c$
18	$y=\frac{2x^5+4x^2+9}{4x^5-8x^2+7}$	$f(a,b,c,x)=ax^5-bx^2+c$
19	$y=\frac{6x^5+5x^2+4}{3x^5+2x^2+1}$	$f(a,b,c,x)=ax^5+bx^2-c$
20	$y=\frac{-3x^5+4x^3+5}{6x^5-7x^3+8}$	$f(a,b,c,x)=ax^5+bx^3+c$
21	$y=\frac{2x^6+4x^2-6}{3x^6+5x^2-7}$	$f(a,b,c,x)=ax^6-bx^2+c$
22	$y=\frac{9x^6-8x^3-7}{6x^6-5x^3-4}$	$f(a,b,c,x)=ax^6+bx^3+c$

23	$y=\frac{1-\cos\left(\frac{3\pi}{7}+1\right)+2\cos\frac{4\pi}{7}}{9+\cos\frac{3\pi}{7}-4\cos\left(\frac{5\pi}{9}-0{,}5\right)}$	$f(m,n)=\cos\left(\frac{\pi\cdot m}{n}-k\right)$
24	$y=\frac{7\sin\frac{4\pi}{11}-5\sin\left(\frac{5\pi}{2}-2\right)}{5\sin\left(\frac{8\pi}{5}+0{,}3\right)+2\sin\frac{13\pi}{3}+0{,}8}$	$f(n,k)=\sin\left(\frac{\pi\cdot n}{k}-m\right)$
25	$y=\frac{2tg\frac{3x}{2}-5ctg\frac{5x}{2}}{3+tg\frac{5x}{4}}$	$ctgx=\frac{\cos(x)}{\sin(x)}$
26	$y=3\sin 2x+\frac{5\sin 3x+3\cos x}{8\sin x-4\cos 4x}$	$f(a,b,n,m,x)=a\sin nx+b\cos mx$
27	$y=\frac{2\sin(3x+2)+7\sin(3x+1)}{3-4\sin\frac{3\pi}{8}}$	$f(a,b,k,x)=k\sin(ax-b)$

Пример операции с вещественными числами:
(setq x 0.2)
Ответ интерпретатора: *0.2*

Пример операции вычисления синуса:
(sin 1.5)
Ответ интерпретатора: *0.997495*

Контрольные вопросы

1. Форма *defun* определения функции пользователя.
2. Передача параметров в функцию.
3. Дайте определение «Свободные переменные».
4. По заданию преподавателя написать функцию и вызвать ее для вычисления требуемого выражения.

Содержание отчета

Отчет о проделанной работе должен содержать:
Титульный лист.
1.Цель работы.
2.Постановка задачи.
3.Вариант индивидуального задания.
4.Описание программы.
5.Результаты выполнения программы.
Выводы.

Лабораторная работа № 3. Разработка и исследование логических функций на Лиспе

Цель работы: Изучить логические функции Лиспа.

Теоретические сведения

Булева функция (или логическая функция, или функция алгебры логики)] от n аргументов — в дискретной математике — отображение $B^n \to B$, где $B = \{0,1\}$ — булево множество. Элементы булева множества {1, 0} обычно интерпретируют как логические значения «истинно» и «ложно», хотя в общем случае они рассматриваются как формальные символы, не несущие определённого смысла. Неотрицательное целое число n называют арностью или местностью функции, в случае n = 0 булева функция превращается в булеву константу. Элементы декартова произведения (n-я прямая степень) B^n называют булевыми векторами. При работе с булевыми функциями происходит полное абстрагирование от содержательного смысла, который имелся в виду в алгебре высказываний. Тем не менее, между булевыми функциями и формулами алгебры высказываний можно установить взаимно-однозначное соответствие, если: установить взаимно-однозначное соответствие между булевыми переменными и пропозициональными переменными, установить связь между булевыми функциями и логическими связками, оставить расстановку скобок без изменений.

Каждая булева функция арности n полностью определяется заданием своих значений на своей области определения, то есть на всех булевых векторах длины n. Число таких векторов равно 2^n. Поскольку на каждом векторе булева функция может принимать значение либо 0, либо 1, то количество всех n-арных булевых функций равно $2^{(2^{**}n)}$. Поэтому только простейшие и важнейшие булевы функции. Практически все булевы функции малых арностей (0, 1, 2 и 3) сложились исторически и имеют конкретные имена. Если значение функции не зависит от одной из переменных (то есть строго говоря для любых двух булевых векторов, отличающихся лишь в значении этой переменной, значение функции на них совпадает), то эта переменная называется фиктивной.

Булева функция задаётся конечным набором значений, что позволяет представить её в виде таблицы истинности.

Нульарные функции

При n = 0 количество булевых функций сводится к двум 220 = 21 = 2, первая из них тождественно равна 0, а вторая 1. Их называют булевыми константами — тождественный нуль и тождественная единица.

Таблица значений и названий нульарных булевых функций:

Значение	Обозначение	Название
0	F0,0 = 0	тождественный ноль
1	F0,1 = 1	тождественная единица, тавтология

Унарные функции

При $n = 1$ число булевых функций равно $2^{21} = 2^2 = 4$. Определение этих функций содержится в следующей таблице.

Таблица значений и названий булевых функций от одной переменной:

x_0=x	**1**	**0**	**Обозначение**	**Название**
0	0	0	F1,0 = 0	тождественный ноль
1	0	1	F1,1 = $x = \neg x = x' =$ NOT(x)	отрицание, логическое «НЕТ», «НЕ», «НИ», инвертор, SWAP (обмен)
2	1	0	F1,2 = x	тождественная функция, логическое "ДА", повторитель
3	1	1	F1,3 = 1	тождественная единица, тавтология

Бинарные функции

При $n = 2$ число булевых функций равно $2^{2^{**}2} = 2^4 = 16$.

Таблица значений и названий булевых функций от двух переменных:

x_0=x	**1**	**0**	**1**	**0**		
x_1=y	**1**	**1**	**0**	**0**	**Обозначение функции**	**Название функции**
0	0	0	0	0	F2,0 = 0	тождественный ноль

1	0	0	0	0	1	F2,1 = $x \downarrow y = x$ NOR y = NOR(x,y) = x НЕ-ИЛИ y = НЕ-ИЛИ(x,y) = NOT(MAX(X,Y))	стрелка Пи́рса - "↓" (кинжал Куайна - "†"), функция Ве́бба - "°", НЕ-ИЛИ, 2ИЛИ-НЕ, антидизъюнкция, инверсия максимума
2	0	0	1	0		F2,2 = $x > y = x$ GT y = GT(x,y) = $x \to y = x \not\to y$	функция сравнения "первый операнд больше второго операнда", инверсия прямой импликации
3	0	0	1	1		F2,3 = $y = y' = \neg y$ = NOT2(x,y) = НЕ2(x,y)	отрицание (негация, инверсия) второго операнда
4	0	1	0	0		F2,4 = $x < y = x$ LT y = LT(x,y) = $x \leftarrow y = x \not\leftarrow y$	функция сравнения "первый операнд меньше второго операнда", инверсия обратной импликации
5	0	1	0	1		F2,5 = $x = x' = \neg x$ = NOT1(x,y) = НЕ1(x,y)	отрицание (негация, инверсия) первого операнда
6	0	1	1	0		F2,6 = $x >< y = x <> y = x$ NE y = NE(x, y) = $x \oplus y = x$ XOR y = XOR(x,y)	функция сравнения "операнды не равны", сложение по модулю 2, исключающее «или», сумма Жегалкина
7	0	1	1	1		F2,7 = $x \mid y = x$ NAND y = NAND(x,y) = x НЕ-И y = НЕ-И(x,y) = NOT(MIN(X,Y))	штрих Ше́ффера, НЕ-И, 2И-НЕ, антиконъюнкция, инверсия минимума
8	1	0	0	0		F2,8 = $x \wedge y = x \cdot y = xy = x$ & $y = x$ AND y = AND(x,y) = x И y = И(x,y) = min(x,y)	конъюнкция, 2И, минимум
9	1	0	0	1		F2,9 = $(x \equiv y) = x \sim y = x \leftrightarrow y = x$ EQV y = EQV(x,y)	функция сравнения "операнды равны", эквивалентность
10	1	0	1	0		F2,10 = YES1(x,y) = ДА1(x,y) = x	первый операнд
11	1	0	1	1		F2,11 = $x \geq y = x >= y = x$ GE y = GE(x,y) = $x \leftarrow y = x \subset y$	функция сравнения "первый операнд не меньше второго операнда", обратная импликация (от второго аргумента к первому)
12	1	1	0	0		F2,12 = YES2(x,y) = ДА2(x,y) = y	второй операнд
13	1	1	0	1		F2,13 = $x \leq y = x <= y = x$ LE y = LE(x,y) = $x \to y = x \supset y$	функция сравнения "первый операнд не больше второго операнда", прямая (материальная) импликация (от первого аргумента ко второму)
14	1	1	1	0		F2,14 = $x \vee y = x + y = x$ OR y = OR(x,y) = x ИЛИ y = ИЛИ(x,y) = max(x,y)	дизъюнкция, 2ИЛИ, максимум
15	1	1	1	1		F2,15 = 1	тождественная единица, тавтология

Объяснение:
NOR – НЕ-ИЛИ;
GT – >;
NOT2 – отрицание второго операнда;
LT – <;
NOT1 – отрицание первого операнда;
XOR – исключающее «или»;
NAND – НЕ-И;
EQV – эквивалентность;
GE – первый операнд не меньше второго операнда;
LE – первый операнд не больше второго операнда.

Стандартные логические функции Common Lisp
(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

Задание к лабораторной работе

Реализовать программу по варианту, в которой описана функция:

№ варианта	Функция
1.	Стрелка Пирса
2.	Инверсия прямой импликации
3.	Негация второго операнда
4.	Инверсия обратной импликации
5.	Негация первого операнда
6.	Исключающее «или»
7.	Штрих Шеффера
8.	Эквивалентность
9.	Функция сравнения «первый операнд не меньше второго операнда»
10.	Функция сравнения «первый операнд не больше второго операнда»

Содержание отчета

Отчет о проделанной работе должен содержать:
Титульный лист.
1.Цель работы.
2.Постановка задачи.
3.Вариант индивидуального задания.
4.Описание программы.
5.Результаты выполнения программы.
Выводы.

Лабораторная работа № 4. Разработка и исследование функций обработки символьных выражений на Лиспе

Цель работы: Изучить базовые функции Лиспа, символы и их свойства, а также средства для работы с числами.

Теоретические сведения

Основные положения программирования на Лиспе.

Лисп ориентирован на обработку нечисловых задач. Он основан на алгебре списочных структур, лямбда-исчислении и теории рекурсий.

Язык имеет функциональную направленность, т. е. любое предложение заключенное в скобки, введенное вне редактора считается функцией и выполняется сразу после нажатия «ENTER».

Чтобы предотвратить вычисление значения выражения, нужно перед этим выражением поставить апостроф «’». Апостроф перед выражением - это на самом деле сокращение лисповской функции QUOTE.

В Лиспе формы представления программы и обрабатываемых ею данных одинаковы. И то и другое представляется списочной структурой имеющей одинаковую форму.

Типы данных не связаны с именами объектов данных, а сопровождают сами объекты. Переменные могут в различные моменты времени представлять различные объекты.

Основные типы данных языка - атомы и списки.

Атомы - это символы и числа.

Список - 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 остаток , т. е. список без первого элемента, называется «хвостом. 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 лисповские объекты. Символы могут состоять как из прописных, так и из строчных букв, хотя в большинстве Лисп-систем, как и в описываемой здесь версии Lisp, 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.

При генерации или считывании Lispом нового символа, за его величину принимается он сам. Такая ссылка символа на себя называется автоссылкой.

Создание программы на Лиспе - написание некоторой функции, возможно сложной, при вычислении использующей другие функции либо рекурсивно саму себя. На практике, написание программ осуществляется записью в файл определений функций, данных и других объектов с помощью имеющегося в программном окружении редактора. Файлу присваивается расширение LSP.

Необязательно делать отступы в строках выражений, входящих в ваши функции. На самом деле, по желанию, вы можете написать всю программу в одну строку. Однако отступы в строках и пустые строки делают структуру программы понятней и более читабельней. Так же вы-равнивание начальных и конечных скобок основных выражений помогают убедиться в балансе ваших скобок.

Определения функций могут храниться в файлах и загружаться используя функцию LOAD:

(load <имя файла>)

Эта функция загружает файл выражений и выполняет эти выражения. <Имя файла> - это строковая константа, которая представляет собой имя файла без расширения (подразумевается расширение ".lsp"). 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.

Базовые функции обработки списков.

Функция CAR возвращает в качестве значения первый элемент списка.

(CAR список) ⇒ S - выражение (атом либо список).

(CAR '(a b c d)) ⇒ a
(CAR '((a b) c d)) ⇒ (a b)
(CAR '(a)) ⇒ a
(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) ⇒ Error

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

(CDR список) ⇒ список

Функция CDR определена только для списков.

(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 'a) ⇒ Error

Функция создания списка – CONS.

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

(CONS голова хвост)

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 всегда будет список:

(CONS s-выражение список) ⇒ 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)

Вложенные вызовы CAR и CDR.

Комбинации вызовов CAR и CDR образуют уходящие в глубину списка обращения, в Лиспе для этого используется более короткая запись. Желаемую комбинацию вызовов CAR и CDR можно записать в виде одного вызова функции:

(C...R список)

Вместо многоточия записывается нужная комбинация из букв A и D (для CAR и CDR соответственно). В один вызов можно объединять не более четырех функций CAR и CDR.

(CADAR x) ⇒ (CAR (CDR (CAR x)))

(CDDAR '((a b c d) e)) ⇒ (c d)
(CDDR '(k l m)) ⇒ (M)

Функция LIST - создает список из элементов. Она возвращает в качестве своего

значения список из значений аргументов. Количество аргументов произвольно.

(LIST 'a 'b 'c) ⇒ (a b c)
(LIST 'a 'b (+ 1 2)) ⇒ (a b 3)

Предикаты ATOM, EQ, EQL, EQUAL.

Предикат - функция, которая определяет, обладает ли аргумент определенным свойством, и возвращает в качестве значения NIL или T.

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, в противном случае - NIL. С помощью EQ сравнивают только символы или константы T и NIL.

(EQ 'a 'b) ⇒ NIL
(EQ 'a (CAR '(a b c))) ⇒ T
(EQ NIL ()) ⇒ T

Предикат EQL работает так же 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 как EQL , но дополнительно проверяет одинаковость двух списков. Если внешняя структура двух лисповских 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 на пустой список.

(NULL '()) ⇒ T

Числовые предикаты.
(ZEROP число) ⇒ проверка на ноль
(MINUSP число) ⇒ проверка на отрицательность

Символы, свойства символов.
Функции присваивания: 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, если этому имени не будет присвоено новое значение функцией SET.

(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)

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 второй аргумент.

(SETQ d '(l m n)) ⇒ (l m n)

Функция SETF - обобщенная функция присваивания. SETF используется для занесения значения в ячейку памяти.
(SETF ячейка-памяти значение)

(SETF ячейка '(a b c)) ⇒ (a b c)
ячейка ⇒ (a b c)

Переменная «ячейка» без апострофа указывает на ячейку памяти, куда помещается в качестве значения список (a b c).
В Лиспе с символом можно связать именованные свойства. Свойства символа записываются в хранимый вместе с символом список свойств. Свойство имеет **имя** и **значение**. Список свойств может быть пуст. Его можно изменять или удалять без ограничений.
(имя1 знач1 имя2 знач2 ... имяN значN)

Списки свойств
В Common Lisp каждый символ имеет собственный список свойств (property-list, plist). Функция get принимает символ и ключ, возвращая значение, связанное с этим ключом, из списка свойств:
> (get 'alizarin 'color)
NIL
Для сопоставления ключей используется eql. Если указанное свойство не задано, get возвращает nil. Чтобы ассоциировать значение с ключом, можно использовать setf вместе с get:

> (setf (get 'alizarin 'color) 'red)
RED
> (get 'alizarin 'color)
RED

Теперь свойство color (цвет) символа alizarin имеет значение red (красный). Функция symbol-plist возвращает список свойств символа:

> (setf (get 'alizarin 'transparency) 'high)
HIGH
> (symbol-plist 'alizarin)
(TRANSPARENCY HIGH COLOR RED)

Заметьте, что списки свойств не представляются как ассоциативные списки, хотя и работают похожим образом.

Функция remprop с обращением (remprop a name) служит для удаления свойства name у атома a, оба аргумента функции вычисляются. Если свойства с заданным именем у этого атома нет, то значение функции равноNIL.

(remprop 'APPLE 'TASTE) => SWEET
(get 'APPLE 'TASTE) => NIL

В в случае успешного удаления свойства атома функция remprop выдает T.

Представление списков в памяти компьютера.

Произвольное S-выражение можно изобразить в виде бинарного дерева, у которого листья атомы этого выражения, а поддеревья-точечные пары. Поэтому считают, что логической структурой S-выражения является бинарное дерево.

Например, в памяти компьютера x ссылается на (A (B C) D) , а y – на (A .((B . C). D)))

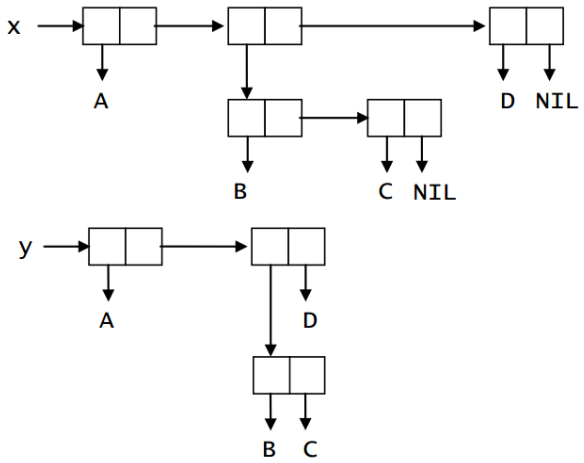


Рис. Внутреннее представление списочных структур

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.

APPLY

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)

(SETQ a '+) ⇒ +

(APPLY a '(1 2 3)) ⇒ 6
(APPLY '+ '(4 5 6)) ⇒ 15

FUNCALL.

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

(FUNCALL fn x1 x2 ... xn)

(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 дру-гой 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)

Отображающие функционалы.

Отображающие или MAP-функционалы являются функциями, которые являются функциями, которые некоторым образом отображают список (последовательность) в новую последовательность или порождают побочный эффект, связанный с этой последовательностью. Каждая из них имеет более двух аргументов, значением первого должно быть имя определенной ранее или базовой функции, или лямбда-выражение, вызываемое MAP-функцией итерационно, а остальные аргументы служат для задания аргументов на каждой итерации. Естественно, что количество аргументов в обращении к MAP-функции должно быть согласовано с предусмотренным количеством аргументов у аргумента-функции. Различие между всеми MAP-функциями состоит в правилах формирования возвращаемого значения и механизме выбора аргументов итерирующей функции на каждом шаге.

Рассмотрим основные типы MAP-функций.

MAPCAR.

Значение этой функции вычисляется путем применения функции fn к последовательным элементам xi списка, являющегося вторым аргументом функции. Например в случае одного спи-ска получается следующее выражение:

(MAPCAR fn '(x1 x2 ... xn))

В качестве значения функционала возвращается список, построенный из результатов вызовов функционального аргумента MAPCAR.

(MAPCAR 'LISTP '((f) h k (i u))) ⇒ (T NIL NIL T)
(SETQ x '(a b c)) ⇒ (a b c)
(MAPCAR 'CONS x '(1 2 3)) ⇒ ((a . 1) (b . 2) (c . 3))

MAPLIST.

MAPLIST действует подобно MAPCAR, но действия осуществляет не над элементами списка, а над последовательными CDR этого списка.

(MAPLIST 'LIST '((f) h k (i u)) ⇒ (T T T T)
(MAPLIST 'CONS '(a b c) '(1 2 3)) ⇒ (((a b c) 1 2 3) ((b c) 2 3) ((c) 3))

Функционалы MAPCAR и MAPLIST используются для программирования циклов специального вида и в определении других функций, поскольку с их помощью можно сократить запись повторяющихся вычислений.

Функции MAPCAN и MAPCON являются аналогами функций MAPCAR и MAPLIST. Отличие состоит в том, что MAPCAN и MAPCON не строят, используя LIST, новый список из результатов, а объединяют списки, являющиеся результатами, в один список.

Индивидуальное задание и методика выполнения работы

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.

а) (1 2 3 a 4)
б) (1 2 3 4 a)
в) ((1) (2 3) (a 4))
г) ((1) ((2 3 a) (4)))
д) ((1) ((2 3 a 4)))
е) (1 (2 ((3 4 (5 (6 a))))))

2. Каково значение каждого из следующих выражений:

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

3. Проделайте следующие вычисления с помощью интерпретатора Лиспа:

а) (car '(1 2))
б) (car (cons 2 3))
в) (cdr '(1 2))
г) (cdr '(1))

4. Определите значения следующих выражений:

а) (car '(rose violet daisy buttercup))
б) (car '(pine fir oak maple))
в) (cdr '(pine fir oak maple))
г) (car '((lion tiger cheetah)
(gazelle antelope zebra)
(whale dolphin seal)))
д) (cdr '((lion tiger cheetah)
(gazelle antelope zebra)
(whale dolphin seal)))

5.1 Составьте список студентов своей группы
(ФИО1 ФИО2 ... ФИОN)

5.2 Составьте список свойств, характеризующих Вашу успеваемость в прошлом семестре

6. Вычислите значения следующих вызовов:

a) (APPLY 'LIST '(a b));
b) (FUNCALL 'LIST '(a b));

c) (FUNCALL 'APPLY 'LIST '(a b));
d) (FUNCALL 'LIST 'APPLY '(a b));

Содержание отчета

Отчет о проделанной работе должен содержать:
Титульный лист.
1.Цель работы.
2.Постановка задачи.
3.Вариант индивидуального задания.
4.Описание программы.
5.Результаты выполнения программы.
Выводы.

Лабораторная работа № 5. Разработка и исследование ввода-вывода на Лиспе

Цели: Изучить основы программирования ввода-вывода.

Теоретические сведения

Программируемый парсер

Парсер лиспа позволяет легко разбирать входные данные. Он получает текст из входного потока и создаёт лисповские объекты, которые обычно называют S-выражениями. Это очень сильно упрощает разбор входных данных.

Парсер можно использовать посредством нескольких функций, таких как READ, READ-CHAR, READ-LINE, READ-FROM-STRING и т.д. Входной поток может быть файлом, вводом с клавиатуры и так далее, но, кроме того, мы можем читать данные из строк или последовательностей символов при помощи соответствующих функций.

Вот простейший пример чтения при помощи READ-FROM-STRING, который создаёт объект (400 500 600), то есть список, из строки "(400 500 600)".

```
> (read-from-string "(400 500 600)")
(400 500 600)
13
> (type-of (read-from-string "t"))
BOOLEAN
```

Макросы чтения (reader macros) позволяют определить специальную семантику для заданного синтаксиса. Это возможно потому, что парсер лиспа является программируемым. Макросы чтения — это ещё один способ расширить синтаксис языка (они обычно используются, чтобы добавить синтаксический сахар).

Некоторые стандартные макросы чтения:

#'foo — функции,
#\\ — символы (characters),
#c(4 3) — комплексные числа,
#p"/path/" — пути к файлам.

Парсер может сгенерировать любой объект, для которого определены правила чтения; в частности, эти правила можно задать при помощи макросов чтения. На самом деле парсер, о котором идёт речь, используется и для интерактивных интерпретаторов (read-eval-print loop, REPL).

Вот так мы можем прочитать число в шестнадцатеричной записи при помощи стандартного макроса чтения:

```
> (read-from-string "#xBB")
187
```

Программируемая печать

Система текстового вывода в лиспе предоставляет возможности для печати структур, объектов или каких-либо ещё данных в разном виде.

PRINT-OBJECT — это встроенная обобщённая функция, которая принимает в качестве аргументов объект и поток, и соответствующий метод выводит в поток текстовое представление данного объекта. В любом случае, когда нужно текстовое представление объекта, используется эта функция, в том числе в FORMAT, PRINT и в REPL.

Рассмотрим класс JOURNEY:

```
(defclass journey ()
  ((%from :initarg :from :accessor from)
   (%to :initarg :to :accessor to)
   (%period :initarg :period :accessor period)
   (%mode :initarg :mode :accessor mode)))
```

Если мы попытаемся распечатать объект класса JOURNEY, мы увидим нечто подобное:

```
> (defvar *journey*
    (make-instance 'journey
                   :from "Christchurch" :to "Dunedin"
                   :period 20 :mode "bicycle"))
*JOURNEY*
> (format nil "~a" *journey*)
"#<JOURNEY {10044DCCA1}>"
```

Можно определить метод PRINT-OBJECT для класса JOURNEY, и с его помощью задать какое-то текстовое представление объекта:

```
(defmethod print-object ((j journey) (s stream))
  (format s "~A to ~A (~A hours) by ~A."
          (from j) (to j) (period j) (mode j)))
```

Наш объект теперь будет использовать новое текстовое представление:

```
> (format nil "~a" *journey*)
"Christchurch to Dunedin (20 hours) by bicycle."
```

Индивидуальное задание и методика выполнения работы

Задания к лабораторной работе

Выполнить лабораторную работу № 2, исходные данные вводятся с клавиатуры, результат выводится на экран.

Пример программы

```
defun quadratic-roots-2 (A B C)
  (cond ((= A 0) (string "Not a quadratic equation."))
    (t
    (let ((D (- (* B B) (* 4 A C))))
      (cond ((= D 0) (concatenate 'string "x = " (write-to-string (/ (+ (- B) (sqrt D)) (* 2 A)))))
        (t
        (concatenate 'string (concatenate 'string "x1 = " (write-to-string (/ (+ (- B) (sqrt D)) (* 2
A))))
                             (concatenate 'string "~%x2 = " (write-to-string (/ (- (- B) (sqrt D)) (* 2
A)))))))))))

(let ((A (read))
    (B (read))
    (C (read)))
(format t (quadratic-roots-2 A B C)))
```

Лабораторная работа № 6. Разработка и исследование файлового ввода-вывода на Лиспе

Цели: Изучить основы программирования файлового ввода-вывода.

Теоретические сведения

Common Lisp предоставляет мощную библиотеку для работы с файлами. Средства Common Lisp, предназначенные для ввода/вывода, аналогичны имеющимся в других языках программирования. Common Lisp предоставляет потоковую абстракцию операций чтения/записи, а для манипулирования файловыми объектами в независимом от операционной системы формате - абстракцию файловых путей. Кроме того, Common Lisp предоставляет некоторое количество уникальных возможностей, такие как чтение и запись s-выражений.

Чтение данных из файлов

Самая фундаментальная задача ввода/вывода - чтение содержимого файла. Для того, чтобы получить поток, из которого вы можете прочитать содержимое файла, используется функция OPEN. По умолчанию, OPEN возвращает посимвольный поток ввода данных, который можно передать множеству функций, считывающих один или несколько символов текста: READ-CHAR считывает одиночный символ; READ-LINE считывает строку текста, возвращая ее как строку без символа конца строки; функция READ считывает одиночное s-выражение, возвращая объект Lisp. Когда работа с потоком завершена, вы можете закрыть его с помощью функции CLOSE.

Функция OPEN требует имя файла как единственный обязательный аргумент. Как можно увидеть в секции "Имена файлов", Common Lisp предоставляет два пути для представления имени файла, но наиболее простой способ - использовать строку, содержащую имя в формате, используемом в файловой системе. Так, предполагая, что /some/file/name.txt это файл, возможно открыть его следующим образом:

```
(open "/some/file/name.txt")
```

Вы можете использовать объект, возвращаемый функцией, как первый аргумент любой функции, осуществляющей чтение. Например, для того, чтобы напечатать первую строку файла, вы можете комбинировать OPEN, READ-LINE, CLOSE следующим образом:

```
(let ((in (open "/some/file/name.txt")))
  (format t "~a~%" (read-line in))
  (close in))
```

Конечно, при открытии и чтении данных может произойти ряд ошибок. Файл может не существовать, или вы можете непредвиденно достигнуть конца файла в процессе его чтения. По умолчанию, OPEN и READ-* будут сигнализировать об ошибках в данных ситуациях.

Если вы хотите открыть файл, который возможно не существует, без генерирования ошибки функцией OPEN, вы можете использовать аргумент :if-does-not-exists для того, чтобы указать другое поведение. Три различных значения допустимы для данного аргумента - :error, по умолчанию; :create, что указывает на необходимость создания файла и повторное его открытие как существующего и NIL, что означает возврат NIL (при неуспешном открытии) вместо потока. Итак, возможно изменить предыдущий пример таким образом, чтобы обработать несуществующий файл.

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (format t "~a~%" (read-line in))
    (close in)))
```

Все функции чтения - READ-CHAR, READ-LINE, READ - принимают опциональный аргумент, по умолчанию "истина", который указывает должны ли они сигнализировать об ошибке, если они достигли конца файла. Если этот аргумент установлен в NIL, то они возвращают значение их 3го аргумента, который по умолчанию NIL, вместо ошибки. Таким

образом, вывести на печать все строки файла можно следующим способом:

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (loop for line = (read-line in nil)
        while line do (format t "~a~%" line))
    (close in)))
```

Среди трёх функций чтения, READ – уникальна. Это – та самая функция, которая представляет букву "R" в "REPL", и которая используется для того, чтобы читать исходный код Lisp. Во время вызова она читает одиночное s-выражение, пропуская пробельные символы и комментарии, и возвращает объект Lisp, представляемый s-выражением. Например, предположим, что файл /some/file/name.txt содержит следующие строки:

```
(1 2 3)
456
"строка" ; это комментарий
((a b)
 (c d))
```

Другими словами, он содержит 4 s-выражения: список чисел, число, строку, и список списков. Вы можете считать эти выражения следующим образом:

```
CL-USER> (defparameter *s* (open "/some/file/name.txt"))
*S*
CL-USER> (read *s*)
(1 2 3)
CL-USER> (read *s*)
456
CL-USER> (read *s*)
"строка"
CL-USER> (read *s*)
((A B) (C D))
CL-USER> (close *s*)
T
```

Итак, когда необходимо хранить данные в файлах, PRINT и READ предоставляют простой способ делать это без создания специального формата данных и парсера для их прочтения. Вы даже можете использовать комментарии без ограничений. И, поскольку s-выражения создавались для того, чтобы быть редактируемыми людьми, то они так же хорошо подходят для использования в качестве формата конфигурационных файлов.

Файловый вывод

Для записи данных в файл необходим поток вывода, который можно получить вызовом функции OPEN с ключевым аргументом :direction :output. Когда файл открывается для записи, OPEN предполагает, что файл не должен существовать, и будет сообщать об ошибке в противном случае. Однако, возможно изменить это поведение с помощью ключевого аргумента :if-exists. Передавая значение :supersede можно вызвать замену существующего файла. Значение :append позволяет осуществлять запись таким образом, что новые данные будут помещены в конец файла, а значение :overwrite возвращает поток, который будет переписывать существующие данные с начала файла. Если же передать NIL, то OPEN вернет NIL вместо потока, если файл уже существует. Характерное использование OPEN для вывода данных выглядит следующим образом:

```
(open "/some/file/name.txt" :direction :output :if-exists :supersede)
```

Common Lisp также предоставляет некоторые функции для записи данных: WRITE-CHAR пишет одиночный символ в поток. WRITE-LINE пишет строку, за которой следует символ конца строки, с учетом реализации для конкретной платформы. Другая функция, WRITE-STRING пишет строку, не добавляя символ конца строки. Две разные функции могут использоваться для того чтобы вывести символ конца строки: TERPRI - сокращение для

"TERminate PRInt" (закончить печать) безусловно печатает символ конца строки, а FRESH-LINE печатает символ конца строки только в том случае, если текущая позиция печати не совпадает с началом строки. FRESH-LINE удобна в том случае, когда желательно избежать паразитных пустых строк в текстовом выводе, генерируемом другими последовательно вызываемыми функциями. Допустим, например, что есть одна функция, которая генерирует вывод и после которой обязательно должен идти перенос строки и другая, которая должна начинаться с новой строки. Но, предположим, что если функции вызываются последовательно, то необходимо обеспечить отсутствие лишних пустых строк в их выводе. Если в начале второй функции используется FRESH-LINE, ее вывод будет постоянно начинать с новой строки, но если она вызывается непосредственно после первой функции, то не будет выводиться лишний перевод строки.

Некоторые функции позволяют вывести данные Lisp в форме s-выражений: PRINT печатает s-выражение, предваряя его символом начала строки, и пробельным символом после. PRIN1 печатает только s-выражение. А функция PPRINT печатает s-выражения аналогично PRINT и PRIN1, но использует "красивую печать", которая пытается печатать s-выражения в эстетически красивом виде.

Однако, не все объекты могут быть напечатаны в том формате, который понимает READ. Переменная *PRINT-READABLY* контролирует поведение при попытке напечатать подобный объект с помощью PRINT, PRIN1 или PPRINT. Когда она равна NIL, эти функции напечатают объект в таком формате, что READ при попытке чтения гарантировано сообщит об ошибке; в ином случае они просигнализируют об ошибке вместо того, чтобы напечатать объект.

Еще одна функция, PRINC, также печатает объекты Лиспа, но в виде, удобном для человеческого восприятия. Например, PRINC печатает строки без кавычек. Текстовый вывод может быть еще более замысловатым, если задействовать потрясающе гибкую, и в некоторой степени загадочную функцию FORMAT.

Для того, чтобы записать двоичные данные в файл, следует открыть файл функцией OPEN с тем же самым аргументом :element-type, который использовался при чтении данных: '(unsigned-byte 8). После этого можно записывать в поток отдельные байты функцией WRITE-BYTE.

Функция блочного вывода WRITE-SEQUENCE принимает как двоичные, так и символьные потоки до тех пор, пока элементы последовательности имеют подходящий тип: символы или байты. Так же как и READ-SEQUENCE, эта функция наверняка более эффективна, чем запись элементов последовательности поодиночке.

Закрытие файлов

Любой, кто писал программы, взаимодействующие с файлами, знает, что важно закрывать файлы, когда работа с ними закончена, так как дескрипторы норовят быть дефицитным ресурсом. Если открывают файлы и забывают их закрывать, вскоре обнаруживают, что больше нельзя открыть ни одного файла5). На первый взгляд может показаться, что достаточно каждый вызов OPEN сопоставить с вызовом CLOSE. Например, можно всегда обрамлять код, использующий файл, как показано ниже:

```
(let ((stream (open "/some/file/name.txt")))
  ;; работа с потоком
  (close stream))
```

Однако этом метод имеет две проблемы. Первая — он предрасположен к ошибкам: если забыть написать CLOSE, то будет происходить утечка дескрипторов при каждом вызове этого кода. Вторая – наиболее значительная – нет гарантии, что CLOSE будет достигнут. Например, если в коде, расположенном до CLOSE, есть RETURN или RETURN-FROM, возвращение из LET произойдет без закрытия потока. Или, как вы увидите в 19 главе, если какая-либо часть кода до CLOSE сигнализирует об ошибке, управление может перейти за пределы LET обработчику ошибки и никогда не вернется, чтобы закрыть поток.

Common Lisp предоставляет общее решение того, как удостовериться, что

определенный код всегда исполняется: специальный оператор UNWIND-PROTECT. Так как открытие файла, работа с ним и последующее закрытие очень часто употребляются, Common Lisp предлагает макрос, WITH-OPEN-FILE, основанный на UNWIND-PROTECT, для скрытия этих действий. Ниже — основная форма:

```
(with-open-file (stream-var open-argument*)
  body-form*)
```

Выражения в body-form* вычисляются с stream-var, связанной с файловым потоком, открытым вызовом OPEN с аргументами open-argument*. WITH-OPEN-FILE удостоверяется, что поток stream-var закрывается до того, как из WITH-OPEN-FILE вернется управление. Поэтому читать файл можно следующим образом:

```
(with-open-file (stream "/some/file/name.txt")
  (format t "~a~%" (read-line stream)))
```

Создать файл можно так:

```
(with-open-file (stream "/some/file/name.txt" :direction :output)
  (format stream "Какой-то текст."))
```

Как правило, WITH-OPEN-FILE используется в 90-99 процентах файлового ввода/вывода. Вызовы OPEN и CLOSE понадобятся, если файл нужно открыть в какой-либо функции и оставить поток открытым при возврате из нее. В таком случае вы должны позаботиться о закрытии потока самостоятельно, иначе произойдет утечка файловых дескрипторов и, в конце концов, вы больше не сможете открыть ни одного файла.

Индивидуальное задание и методика выполнения работы

Задания к лабораторной работе

Выполнить лабораторную работу № 2, исходные данные вводятся из файла, результат выводится в файл и на консоль.

Пример: чтение из файла

```
(defun load_my_data (file_name)
(setq my_data nil)
(setq stream (open file_name :direction :input))
(loop (if (not(listen stream)) (return my_data))
(setq my_data (cons (read stream) my_data)))
(close stream))
```

Пример: вызов (load_my_data "h:/test_file/test.dat") в качестве побочного эффекта дает формирование в памяти списка my_data с содержимым, считанным из файла "h:/test_file/test.dat”.

Содержание отчета

Отчет о проделанной работе должен содержать:
Титульный лист.
1.Цель работы.
2.Постановка задачи.
3.Вариант индивидуального задания.
4.Описание программы.
5.Результаты выполнения программы.
Выводы.

Лабораторная работа № 7. Разработка и исследование рекурсивных функций на Лиспе

Цели: Изучить основы программирования с применением рекурсии.

Теоретические сведения

1. Рекурсия. 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 содержит всегда 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))))
            NIL))
```

Рекурсию называют параллельной, если она встречается одновременно в нескольких аргументах функции:

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

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

```
(DEFUN PREOBR (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)))))
```

Трассировка — это отслеживание информации о ходе выполнения программы (например, о вызовах и аргументах функций) в целях ее отладки. В Lisp для этого можно воспользоваться макросом trace. Пример:

```
(defun fact (n)
 (if (zerop n)
    1
    (* n (fact (1- n)))))


CL-USER> (trace fact)
(FACT)
CL-USER> (fact 5)
 0: (FACT 5)
  1: (FACT 4)
   2: (FACT 3)
    3: (FACT 2)
     4: (FACT 1)
      5: (FACT 0)
      5: FACT returned 1
     4: FACT returned 1
    3: FACT returned 2
   2: FACT returned 6
  1: FACT returned 24
 0: FACT returned 120
120
CL-USER>
```

Вместо CL-USER> может быть звездочка.

Индивидуальное задание и методика выполнения работы

Задания к лабораторной работе

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

2. Разработать 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).

3. Разработать функцию:

a) вычисляющую число атомов на верхнем уровне списка (Для списка (a в ((a) c) e) оно равно трем.);

b) определяющую число подсписков на верхнем уровне списка;

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

4. Разработать функцию:

a) от двух аргументов X и N, которая создает список из N раз повторенных элементов X;

b) удаляющую повторные вхождения элементов в список;

c) которая из данного списка строит список списков его элементов, например, (a b) ⇒ ((a) (b));

d) вычисляющую максимальный уровень вложения подсписков в списке;

e) единственным аргументом которой являлся бы список списков, объединяющую все эти списки в один;

f) зависящую от трех аргументов X, N и V, добавляющую X на N-е место в список V.

5. Разработать функцию:

a) аналогичную функции SUBST, но в которой третий аргумент W обязательно должен быть списком;

b) которая должна производить замены X на Y только на верхнем уровне W;

c) 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))));

d) аналогичную функции SUBST, но производящую взаимную замену X на Y, т. е. X ⇒ Y, Y ⇒ X.

Содержание отчета

Отчет о проделанной работе должен содержать:
Титульный лист.
1.Цель работы.
2.Постановка задачи.
3.Вариант индивидуального задания.
4.Описание программы.
5.Результаты выполнения программы.
Выводы.

ПРИЛОЖЕНИЕ А. Примеры для выполнения самостоятельной работы

В примерах для выполнения самостоятельной работы требуется исследовать схему рекурсивного процесса, подобрать корректные входные данные для функции. Описать процесс выполнения рекурсивных функций.

```
(defun F0(a b c)
  (F1 (* 4 a c) (* b b))
)
```

```
(defun F1(d e)
  (F2 (+ d e))
   )

(setq g 2.0)
(setq f 100.0)
(defun F2(f)
    (F3 (sqrt f) g))

(setq b 10.0)
(defun F3(b g)

    (F4 (+ (- b) g) g)
    (F4 (+ (- b) g) g)
   )
(setq h 10.0)
(setq s 2.0)
(defun F4(h s)
  (let ((h 7) (s 3))(print(/ h (* 2 s))))
)
(let ((a1 7) (b1 3))(* a1 b1))
(defun FR1(a1 b1 a2 b2 a3 b3 a4 b4)
  (F2 (* (+ a1 b1) (- a2 b2) (/ a3 b3) (* a4 b4)))
)

(defun FR2(x y z m c d)
  (cond
    ((> y 0) (F4 (* x m) (+ c d)))
  (T (F3 (- c d) (+ x z)))
  )
   )
(defun fib(n)
     (if (<= n 1)
       1
        (+ (fib (- n 1)) (fib (- n 2)))))
  (FIB 3)
(DEFUN FIB1 (N)
     (IF (> N 0)
       (IF (OR (= N 1)(= N 2))
                1
          (+ (FIB1 (- N 1)) (FIB1 (- N 2))))))
      (FIB1 3)
 (trace fib) ; включить трассировку функции (fib 7 ) / untrace

(defun PREOBR (l); выдает список из первых элементов всех подсписков
    (COND
       ((NULL l) nil)
       ((ATOM l) nil)
            ((atom (car l))(CONS (CAR l) nil))
       (T (append
            (PREOBR (CAR l))
```

```
            (PREOBR (CDR l))))))

(defun double1(lst); повторяет каждый элемент списка дважды
  (cond
    ((null lst)NIL)
    (T(cons(car lst)(cons(car lst)(double1(cdr lst)))))))


(defun SUM(Lst)
  (COND
    ((NULL Lst) 0)
    (T (+ (CAR Lst) (SUM (CDR Lst)) ))
))

(defun M(Lst); max значение из списка
 (COND
   ((NULL Lst) NIL)
   ((NULL (CDR Lst)) (CAR Lst))
   (T (max (CAR Lst) (M (CDR Lst))) )
))
(defun DF(Lst) ;список из подсписков - из каждого удаляется первый элемент
  (COND
    ((NULL Lst) NIL)
  ((atom lst) nil)
    (T (CONS (CDR (CAR Lst)) (DF (CDR Lst)) ))
))

;((1 2)( 7 9))

(defun DL (Lst);список без последнего элемента
   (COND
    ((NULL (CDR Lst)) NIL)
    (T (CONS (CAR Lst) (DL (CDR Lst)) ))
))
;(1 2 5)


(defun SFSUM(Rez Lst); вспомогательная для fsum, можно обратиться с res=0
  (COND
    ((NULL Lst) Rez)
    (T (SFSUM (+ (CAR Lst) Rez) (CDR Lst) ))
))
 (defun FSUM(Lst) ;сумма элементов списка
  (SFSUM 0 Lst))

 (defun SFM(MAXX Lst); макс элемент списка
  (COND
    ((NULL Lst) MAXX)
    (T (SFM (max MAXX (CAR Lst)) (CDR Lst) ))
))
(defun FM(Lst)  ;макс элемент списка через sfm
  (SFM (CAR Lst)(CDR Lst))
```

```
)

(defun SFR(Rev Lst) ; реверс списка
  (COND
    ((NULL Lst) Rev)
    (T (SFR(cons (CAR Lst)Rev) (CDR Lst)))))
(defun FR(Lst) ;реверс списка
 (SFR nil Lst)
)

(trace ins) ; включить трассировку функции (ins 7 '(1 2 3))
(defun Ins(In Lst) ; вставляет последним в список
  (COND
    ((NULL Lst)(List In))
    (T (cons (CAR Lst)(Ins In (CDR Lst))))
))

(defun R(Lst) ; меняет местами первый и последний элементы списка
  (COND
    ((NULL Lst) nil)
    (T (INS (CAR Lst)(R (CDR Lst))))
))

(defun TSum(Tree)   ;(7 (4 nil nil)(9 nil nil)) --> 20
  (COND
    ((NULL Tree) 0)
    (T (+ (CAR TREE) (TSUM (CADR Tree)) (TSUM (CADDR Tree))))
  )
)
;Функция APPEND1 соединяет два списка в один
 (defun append1 (l p)
       (if (null l) p     ;L пуст - вернуть P (условие окончания),
          (cons (car l)   ;иначе - создать список,
               (append1 (cdr l) p)))) ;используя рекурсию.

           (append1 '(a b c) '(d e f g))
(defun append_l(L1 L2)
  (cond
    ((NULL L1) L2)
    ((NULL L2) L1)
    (T (cons (car L1) (append_l (cdr L1) L2)))))

  (append_l '(a b c) '(w e r))

(defun INS1 (El Lst) ; вставляет  el в упорядоченный список не нарушая упорядочивания
  (COND
    ((NULL Lst) (List El))
    ((<= El (CAR Lst)) (CONS EL Lst))
     (T(CONS(CAR LST)(INS1 EL (CDR LST))))))

(defun SORT1(LST) ; по возрастанию
  (COND
```

```
   ((NULL LST) NIL)
   (T(INS1(CAR LST)(SORT1(CDR LST))))))

  ;DELETE1 - удаляет элемент X из списка L
(defun delete1 (x l)
      (cond ((null l) nil)
            ((equal (car l) x) (delete1 x (cdr l)))
            (t (cons (car l) (delete1 x (cdr l))))))
      (delete1 'a '(s a d f a g e a))

(defun LELst (El Lst) ;удаляет все, которые меньше El
  (COND
    ((NULL LST) NIL)
    ((< El (CAR LST)) (LELst El (CDR Lst)))
    (T (CONS (CAR LST) (LELst El (CDR Lst))))
  )
)

(defun GLst (El Lst) ;удаляет все, которые больше или равно El
  (COND
    ((NULL LST) NIL)
    ((>= El (CAR LST)) (GLst El (CDR Lst)))
    (T (CONS (CAR LST) (GLst El (CDR Lst))))
  )
)
(defun QSORT (LST) ;сортирует список по возрастанию
  (COND
    ((NULL LST) NIL)
    (T
     (Append (qsort (LeLst (CAR LST) (CDR LST)))
       (cons (car lst) (qsort (GLst (CAR LST) (CDR LST))))))))

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

  (akk 2 2)
  (akk 2 3)
```