ANSI Common Lisp

Paul Graham

Я растерял результаты моих неустанных трудов, гревших сердце,
Или ожесточение сердца и движение вперед
Но солнце снова озарило нас
И арфа твоя опять заговорила,
Вперед, заблаговременно, но не беспрерывно;
И ураган для них стал мягок как бальзам.
Без тени сомнения, обернулся он вокруг с насмешкой девять раз,
Несмотря на то, что дважды отразилось эхом,
И цепи эти уходили к преисподней, разве не ужасно?
Пока тело или что еще, и прекраснейшие райские услады
Держаться подальше от света и парить взглядом,
Или кормя, подобно духам, я появился пред тобой.

Хенли.[1]

[1] Эта строфа написана программой Хенли на основе поэмы Джона Мильтона "Потерянный рай". Можете попытаться найти тут смысл, но лучше загляните в главу 8. – *Прим. перев.*

Предисловие....7
Аудитория....7
Как пользоваться книгой....7
Код....7
On Lisp....8
Благодарности....8
Введение....10
1.1. Новые инструменты....10
1.2. Новые приемы....11
1.3. Новый подход....12
2. Лисп приветствует вас....14
2.1. Внешний вид....14
2.2. Вычисление....15
2.3. Данные....16
2.4. Операции со списками....17
2.5. Истинность....17
2.6. Функции....18
2.7. Рекурсия....19
2.8. Чтение Лиспа....20
2.9. Ввод и Вывод....21
2.10. Переменные....22
2.11. Присвоение....23
2.12. Функциональное программмрование....23
2.13. Итерации....24
2.14. Функции как объекты....25
2.15. Типы....26
2.16. Забегая вперед....27
Итоги главы....27
Упражнения....28
3. Списки....29
3.1. Ячейки....29
3.2. Сопоставление....30
3.4. Построение списков....31
3.5. Пример: сжатие....32
3.6. Доступ....33
3.7. Отображающие функции....34
3.8. Деревья....34
3.9. Чтобы понять рекурсию, нужно понять рекурсию....35
3.10. Множества....36
3.11. Последовательности....37
3.12. Стек....39
3.13. Точечные пары....40
3.14. Ассоциативные списки....40
3.15. Поиск кратчайшего пути....41
3.16. Мусор....44
Итоги главы....45
Упражнения....45
4. Специализированные структуры данных....47
4.1. Массивы....47
4.2. Пример: поиск дихотомией....48
4.3. Строки и знаки....49
4.4. Последовательности....50
4.5. Пример: разбор дат....52
4.6. Структуры....54
4.7. Пример: двоичные деревья поиска....55
4.8. Хеш-таблицы....59
Итоги главы....61
Упражнения....61
5. Управление....63
5.1. Блоки....63
5.2. Контекст....64

5.3. Условия....65
5.4. Итерации....67
5.5. Множественные значения....68
5.6. Прерывания....69
5.7. Пример: арифметические операции с датами....70
Итоги главы....74
Упражнения....74
6. Функции....75
6.1. Глобальные функции....75
6.2. Локальные функции....76
6.3. Списки параметров....77
6.4. Пример: утилиты....78
6.5. Замыкания....80
6.6. Пример: компоновщики функций....82
6.7. Динамическое окружение....84
6.8. Компиляция....85
6.9. Использование рекурсии....85
Итоги главы....87
Упражнения....87
7. Ввод и вывод....89
7.1. Потоки....89
7.2. Ввод....90
7.3. Вывод....92
7.4. Пример: замена строк....93
7.5. Макросимволы....96
Итоги главы....97
Упражнения....97
8. Символы....99
8.1. Имена символов....99
8.2. Списки свойств....99
8.3. А символы-то не маленькие....100
8.4. Создание символов....100
8.5. Использование нескольких пакетов....101
8.6. Ключевые слова....101
8.7. Символы и переменные....102
8.8. Пример: бредогенератор....102
Итоги главы....104
Упражнения....104
9. Числа....106
9.1. Типы....106
9.2. Преобразование и извлечение....106
9.3. Сравнение....108
9.4. Арифметика....108
9.5. Возведение в степень....109
9.6. Тригонометрические функции....110
9.7. Представление....110
9.8. Пример: трассировка лучей....111
Итоги главы....115
Упражнения....116
10. Макросы....117
10.1. Eval....117
10.2. Макросы....118
10.3. Обратная кавычка....119
10.4. Пример: быстрая сортировка....120
10.5. Разработка макросов....120
10.6. Обобщенный доступ....122
10.7. Пример: утилиты на макросах....123
10.8. On Lisp....126
Итоги главы....126
Упражнения....127
11. CLOS....128
11.1. Объектно-ориентированное программирование....128
11.2. Классы и экземпляры....129

11.3. Свойства слота....130
11.4. Суперклассы....131
11.5. Предшествование....132
11.6. Обобщенные функции....132
11.7. Вспомогательные методы....134
11.8. Комбинация методов....136
11.9. Инкапсуляция....137
11.10. Две модели....138
Итоги главы....138
Упражнения....139
12. Структура....140
12.1 Разделяемая структура....140
12.2. Модификация....141
12.3. Пример: очереди....142
12.4. Деструктивные функции....143
12.5. Пример: двоичные деревья поиска....144
12.6. Пример: двусвязные списки....146
12.7. Циклические списки....148
12.8. Неизменные структуры....149
Итоги главы....150
Упражнения....150
13. Скорость....151
13.1. Правило бутылочного горла....151
13.2. Компиляция....152
13.3. Декларации типов....153
13.4. Обходимся без мусора....156
13.5. Пример: пулы....159
13.6. Быстрые операторы....160
13.7. Двухстадийная разработка....161
Итоги главы....162
Упражнения....162
14. Более сложные вопросы....164
14.1. Спецификаторы типов....164
14.2. Бинарные потоки....165
14.3. Макросы чтения....165
14.4. Пакеты....167
14.5. Loop....169
14.6. Исключения....172
15. Пример: умозаключения....174
15.1. Цель....174
15.2. Соответствие....174
15.3. Ответы на вопросы....176
15.4. Анализ....179
16. Пример: генерация HTML....180
16.1. HTML....180
16.2. Утилиты HTML....181
16.3. Итерационная утилита....183
16.4. Генерация страниц....184
17. Пример: объекты....188
17.1. Наследование....188
17.2. Множественное наследование....189
17.3. Определение объектов....190
17.4. Функциональный синтаксис....191
17.5. Определение методов....192
17.6. Экземпляры....193
17.7. Новая реализация....194
17.8. Анализ....198
A. Отладка....200
Отладчик....200
Трассировка и обратная трассировка....200
Когда ничего не происходит....201
Переменные без значения/несвязанные....203
Неожиданные nil....203

Переименование....204
Ключевые слова как необязательные параметры....204
Некорректные декларации....205
Предупреждения....205
B. Лисп на Лиспе....206
C. Изменения в Common Lisp....212
Основные дополнения....212
Частные дополнения....212
Функции....212
Макросы....213
Вычисление и компиляция....213
Побочные эффекты....213
Символы....213
Списки....213
Массивы....214
Строки....214
Структуры....214
Хеш-таблицы....214
I/O....214
Числа....214
Пакеты....215
Типы....215
Изменения с 1990 года....215
Справочник по языку....217
Вычисление и компиляция....219
Типы и классы....221
Управление и потоки данных....221
Итерации....228
Объекты....231
Структуры....237
Исключения....239
Символы....242
Пакеты....243
Числа....245
Знаки....251
Ячейки....253
Массивы....257
Строки....260
Последовательности....261
Хеш-таблицы....264
Пути к файлам....265
Файлы....267
Потоки....268
Печать....272
Считыватель....277
Сборка системы....278
Окружение....279
Константы и переменные....281
Спецификаторы типов....285
Макросы чтения....287
Замечания....289
Предметный указатель....300

Предисловие

Цель данной книги – научить вас языку Common Lisp, быстро и основательно. В действительности, это даже две книги. Первая часть объясняет, с множеством примеров, основные концепции программирования на Common Lisp. Вторая часть – это современное описание стандарта ANSI Common Lisp, содержащее каждый оператор языка.

Аудитория

Книга ANSI Common Lisp предназначена как для студентов, изучающих язык, так и для профессиональных программистов. Ее чтение не требует предварительного знания Лиспа. Опыт написания программ на других языках был бы очень полезен, но также не обязателен. Книга начинается с основных концепций и уделяет особое внимание моментам, которые обычно приводят в замешательство человека, впервые знакомящегося с Лиспом.

Эта книга может использоваться самостоятельно в качестве учебного пособия по Лиспу или как составляющей часть курсов, посвященных искусственному интеллекту или теории языков программирования. Профессиональные разработчики, которые желают изучать Лисп, оценят простой, практический подход. Те, кто уже знаком с языком, найдут в ней множество полезных примеров и удобный справочник по стандарту ANSI Common Lisp.

Как пользоваться книгой

Лучший способ выучить Лисп – использовать его. Кроме того, изучать язык в процессе написания программ на нем – это еще и более интересно. Эта книга устроена так, чтобы читатель смог делать это как можно раньше. После небольшого введения предлагаются следующие главы:

- Во второй главе на двадцати одной странице объясняется все, что понадобится для создания первых Лисп-программ.
- Главы 3–9 вводят ключевые элементы программирования на Лиспе. В этих главах особое внимание уделяется самым необходимым концепциям, например, роли указателей в Лиспе, использованию рекурсии, важности функций как типа данных.

Следующие материалы предназначены для читателей, которые хотят основательно разобраться с основами:

- Главы 10–14 охватывают макросы, CLOS, операции со списками, оптимизации, а так же более сложные темы, например, пакеты и reader-макросы.
- Главы 15–17 подводят итог предыдущих глав в трех примерах реальных приложений: программы для создания логических интерфейсов, HTML-генератора и встроенного объектно-ориентированного языка.

Последняя часть книги состоит из четырех приложений, которые будут полезны всем читателям:

- Приложения A–D включают руководство по отладке, исходные коды для 58 операторов языка, описание основных отличий ANSI-стандарта и предыдущих версий языка°Ё063in, а также справочник по каждому оператору в Common Lisp.

Книга завершается комментариями, содержащими пояснения, ссылки, дополнительный код и прочие отступления. Комментарии помечаются в тексте маленьким кружочком: °.

Код

Несмотря на то, что книга рассказывает об ANSI Common Lisp, вы можете изучать по ней любую разновидность Common Lisp. Примеры, рассчитанные на новые

возможности, обычно сопровождаются комментариями, поясняющими, как они могут быть адаптированы к более ранним реализациям.

Весь код из книги доступен в сети. Вы можете найти его, вместе со ссылками на свободный софт, историческими документами, часто задаваемыми вопросами и множеством других ресурсов, на

http://www.eecs.harvard.edu/onlisp/

Также код доступен анонимно по ftp:

ftp://ftp.eecs.harvard.edu:/pub/onlisp/

Вопросы и комментарии присылайте на *pg@eecs.harvard.edu*.

On Lisp

В этой книге я постарался показать уникальные особенности, которые выделяют Лисп из множества остальных языков, а также новые возможности, которые дает вам этот язык. Взять, например, макросы, которые позволяют разработчику составлять программы, которые пишут программы. Лисп – единственный язык, который позволяет с легкостью осуществлять это, потому что только он предлагает необходимые для этого абстракции. Читателям, которым интересно узнать больше о макросах и других интересных возможностях языка, я предлагаю познакомиться с книгой «On Lisp»[1], которую можно назвать продолжением этой.

Благодарности

Из всех моих друзей, которые мне помогали в работе над книгой, я больше всего обязан Роберту Моррису, который на протяжении всего времени помогал совершенствовать ее. Некоторые примеры, включая Хенли (стр. Ё001out) и сопоставление с образцом (стр. Ё002out), взяты из написанного им кода.

Я был счастлив работать с первоклассной командой технических рецензентов: Сконом Бриттаном (Skona Brittain), Джона Фодераро (John Foderaro), Ника Левине (Nick Levine), Питера Норвига (Peter Norvig) и Дэвида Турецки (Dave Touretzky). Вряд ли найдется хотя бы одна страница, к улучшению которой они не приложили руку. Джон Фодераро даже переписал часть кода к разделу 5.7.

Нашлись люди, которые согласились прочитать рукопись целиком или частично, включая Кена Андерсона (Ken Anderson), Тома Читама (Tom Cheatham), Ричарда Фейтмана (Richard Fateman), Стива Хайна (Steve Hain), Барри Марголина (Barry Margolin), Уалдо Пачеко (Waldo Pacheco), Вилера Румла (Wheeler Ruml) и Стюарта Рассела (Stuart Russel). Кен Андерсон и Вилер Румл, в частности, сделали множество полезных замечаний.

Я благодарен Профессору Читаму, и Гарварду в целом, за предоставление возможностей, необходимых для написания книги. Также благодарю сотрудников лаборатории Айкен, в частности, Тони Хэртмана (Tony Hartman), Дейва Мазиерса (Dave Mazieres), Януша Джуду (Janusz Juda), Гарри Бохнера (Harry Bochner) и Джоанну Клис (Joanna Klys).

Я рад, что снова получил возможность поработать вместе с Аланом Аптом (Alan Apt). С сотрудниками Prentice Hall - Аланом, Моной Помпили (Mona Pompili), Ширли МакГир (Shirley McGuire) и Ширли Майклс (Shirley Michaels), действительно приятно работать.

Обложка книги выполнена превосходным мастером Гино Ли (Gino Lee) из Bow & Arrow Press, Кэмбридж.

Эта книга была набрана с помощью LaTeX, языка, созданного Лесли Лэмпортом (Leslie Lamport) поверх TeX'a Дональда Кнута (Donald Knuth), с использованием дополнительных макросов авторства Л.А.Карра (L.A.Carr), Вана Якобсона (Van Jacobson) и Гая Стила (Guy Steele). Чертежи были выполнены с помощью программы Idraw, созданной Джоном Влиссидс (John Vlissids) и Скоттом Стантоном (Skott Stanton). Предпросмотр всей книги был сделан с помощью программы

[1] Paul Graham – «On Lisp», Prentice Hall, 1993. – *Прим. перев.*

Ghostview, написанной Тимом Тейзеном (Tim Theisen) на основе интерпретатора Ghostscript, созданного Л. Питером Дойчем (L. Peter Deutch).

Я также должен поблагодарить многих других людей: Генри Бейкера (Henry Baker), Кима Барретта (Kim Barrett), Ингрид Бассет (Ingrid Basset), Тревора Блеквелла (Trevor Blackwell), Пола Беккера (Paul Becker), Гарри Бисби (Gary Bisbee), Франка Дучмана (Frank Dueschmann), Франсиса Дикли (Frances Dickey), Рича и Скотта Дрейвса (Rich and Scott Draves), Билла Дабак (Bill Dubuque), Дена Фридмана (Dan Friedman), Дженни Грэм (Jenny Graham), Эллис Хартли (Alice Hartley), Девида Хендлера (David Hendler), Майка Хьюлетта (Mike Hewlett), Гленн Холловот (Glenn Hollowat), Брэда Карпа (Brad Karp), Соню Кини (Sonya Keene), Росса Найтс (Ross Knights), Митсуми Комуро (Mutsumi Komuro), Стефи Кутзиа (Steffi Kutzia), Девида Кузника (David Kuznick), Мэди Лорд (Madi Lord), Джулию Маллози (Julie Mallozzi), Пола МакНами (Paul McNamee), Дейва Муна (Dave Moon), Говарда Миллингса (Howard Mullings), Марка Ницберга (Mark Nitzberg), Ненси Пармет (Nancy Parmet) и ее семью, Роберта Пенни (Robert Penny), Майка Плуча (Mike Plusch), Шерил Сэкс (Cheryl Sacks), Хейзэма Сейеда (Hazem Sayed), Шеннона Спирс (Shannon Spires), Лоу Штейнберга (Lou Steinberg), Пола Стоддард (Paul Stoddard), Джона Стоуна (John Stone), Гая Стила (Guy Steele), Стива Страссмана (Steve Strassmann), Джима Вейча (Jim Veitch), Дейва Уоткинса (Dave Watkins), Айдел и Джулианну Вебер (Idelle and Julian Weber), семество Вейкерсов (the Weickers), Дейва Йоста (Dave Yost) и Алана Джули (Alan Yuille).

Больше всего я хотел бы поблагодарить моих родителей и Джекки.

Дональд Кнут назвал свою известную серию книг "Искусство программирования". В своей Тьюринговской лекции он объяснил, что это было сознательным выбором, так как в программирование его привела как раз "возможность писать красивые программы".

Так же как и архитектура, программирование сочетает искусство и науку. Программа строится на математических принципах, так же как и здание держится на законах физики. Но задача архитектора – не просто построить здание, которое не разрушится. Почти всегда он стремится создать нечто прекрасное.

Многие программисты чувствуют, как и Дональд Кнут, что такова и настоящая цель программирования. Так считают почти все Лисп-хакеры. Саму суть лисп-хакерства можно выразить двумя фразами. Программирование должно доставлять радость. Программы должны быть красивыми. Таковы идеи, которые я постарался пронести через всю книгу.

Пол Грэм.

Введение

Джон Маккарти со своими студентами начал работу над первой реализацией Лиспа в 1958 году. Не считая Фортрана, Лисп это старейший из ныне используемых языков.°Ё064in Поразительно, что до сих пор он остается флагманом среди языков программирования. Специалисты, хорошо знающие Лисп, утверждают, что в нем есть нечто, делающее его особенным по сравнению с остальными языками.

Отчасти его отличает заложенная изначально возможность развиваться. Лисп позволяет определять новые операторы. Если появятся новые абстракции, которые приобретут популярность (например, объектно-ориентированное программирование), их всегда можно будет реализовать в Лиспе. Изменяясь как ДНК, такой язык никогда не выйдет из моды.

1.1. Новые инструменты

Зачем изучать Лисп? Потому что он позволяет делать то, чего не могут другие языки. Если вы захотите написать функцию, складывающую все числа, меньшие заданного, скажем, `n`, она будет очень похожа на аналогичную функцию с Си:

```
; Lisp
(defun sum (n)
  (let ((s 0))
    (dotimes (i n s)
      (incf s i))))
```

```
/* C */
int sum(int n){
  int i , s = 0;
  for(i = 0; i < n; i++)
    s += i;
  return(s);
}
```

Если вы хотите делать несложные вещи типа этой, в сущности, не имеет значения, какой язык использовать. Предположим, что вместо вышеприведенной функции мы хотим написать такую, которая принимает число `n` и возвращает функцию, которая добавляет `n` к своему аргументу.

```
; Lisp
(defun addn (n)
  #'(lambda (x)
      (+ x n)))
```

Как функция `addn` будет выглядеть на Си? Ее просто невозможно написать. Здесь вы, вероятно, спросите, зачем это может понадобиться? Языки программирования учат вас не желать того, чего не могут позволить. Раз программисту приходится думать на том языке, который он использует, ему сложно представить то, чего он не может описать. Когда я впервые занялся программированием, на Бейсике, я не огорчался отсутствию рекурсии, так как попросту не знал, что бывает такая возможность. Я думал на Бейсике и мог представить себе только итеративные алгоритмы, так должен ли я был вообще задумываться о рекурсии?

Если вы не используете лексические замыкания (пример которых приведен в предыдущем примере), сейчас просто примите на веру, что Лисп-программисты используют их постоянно. Сложно найти программу на Лиспе, написанную без использования замыканий. В разделе 6.7 вы научитесь пользоваться ими.

Замыкания – не единственные абстракции, которых нет в других языках. Есть и другая, более важная особенность. Программы на Лиспе записываются с помощью лисповых типов данных. Именно это позволяет в полной мере использовать кодогенерацию. Действительно ли люди пользуются этим? Да, это и есть макросы, и, опять же, опытные программисты используют их на каждом шагу. Вы узнаете, как создавать свои макросы, в главе 10.

С макросами, замыканиями и динамической типизацией, Лисп превосходит объектно-ориентированное программирование. Если бы вы в полной мере поняли предыдущее предложение, то, вероятно, знали бы Лисп очень хорошо и могли бы не читать эту книгу. Но это не просто слова, и вы найдете ясное объяснение тому в коде к главе 17.

Главы 2–13 поэтапно вводят все понятия, необходимые для понимания кода главы 17. Благодаря вашим стараниям вы почувствуете программирование на C++ таким же неудобным и вяжущим, каким опытный программист C++ в свою очередь считает Бейсик. Сомнительная, на первый взгляд, награда. Но это будет выглядеть более ободряющим, если посмотреть, откуда берется это чувство. Бейсик неудобен по сравнению с C++, потому что опытный программист C++ знает приемы, которые невозможны в Бейсике. Точно так же, изучение Лиспа даст вам большее, нежели еще один язык в копилку изученных технологий. Вы научитесь новым, более мощным приемам программирования.

1.2. Новые приемы

Как было показано в предыдущем разделе, Лисп предоставляет такие инструменты, которых нет в других языках. Но это еще не все. Отдельно взятые технологии, впервые появившиеся в Лиспе: автоматическое управление памятью, динамическая типизация, замыкания и т.д., значительно упрощают программирование. Взятые вместе, они создают критическую массу, которая рождает новый подход к программированию.

Лисп изначально гибок, он позволяет самостоятельно задавать новые операторы. Это возможно, потому что сам Лисп написан из таких же функций и макросов. Поэтому, расширить возможности Лиспа ничуть не сложнее, чем написать свою программу. Фактически, это так просто (и полезно), что расширение языка - это нормальная практика. Получается, что вы не только пишете программу в соответствии с языком, но и дополняете язык в соответствии с нуждами программы. Этот подход называется «снизу-вверх» (*bottom-up*).

Практически любая программа будет выигрывать, если используемый язык заточен под нее, и чем более сложна программа, тем больше значимость bottom-up подхода. В такой программе может быть несколько слоев, каждый из которых служит чем-то вроде языка для описания вышележащего слоя. Одной из первых программ, написанных таким образом, был TeX. Вы имеете возможность писать программы снизу-вверх на любом языке, но на Лиспе это делать проще всего.

Написанные снизу-вверх программы легко расширяемы. Поскольку идея расширяемости лежит в основе Лиспа, это идеальный язык для написания расширяемых программ. В качестве примера приведу три программы, написанные в 80-х годах и использовавшие расширяемость Лиспа: GNU Emacs, Autocad и Interleaf.

Кроме того, код, написанный данным методом, легко использовать многократно. Суть написания повторно используемого кода в отделении общего от частного, а эта идея лежит в самой основе метода. Вместо того, чтобы прилагать существенные усилия к созданию монолитного приложения, полезно потратить часть времени на построение своего языка, поверх которого затем реализовывать само приложение. Приложение будет находиться на вершине пирамиды языковых прослоек и будет иметь наиболее специфичное применение, а сами прослойки можно будет приспособить к повторному использованию. Действительно, что может быть более пригодным к многократному применению, чем язык программирования?

Лисп позволяет не просто создавать более хитрые приложения, но и делать это быстрее. Практика показывает, что программы на Лиспе выглядят короче, чем аналоги на други языках. Как показал Фредерик Брукс, временные затраты на написание программы зависят в первую очередь от ее длины.°Ё065in В случае Лиспа этот эффект подкрепляется его динамическим характером, за счет которого сокращается время между редактированием, компиляцией и тестированием.

Мощные абстракции и интерактивность вносят коррективы и в принцип разработки приложений. Суть Лиспа можно выразить одной фразой – быстрое прототипирование. Гораздо удобнее не составлять спецификацию, а написать прототип. Прототип служит проверкой предположения, является ли эффективным выбранный метод, а также гораздо ближе к готовой программе, чем спецификация.

Превзойти объектно-ориентированное программирование? Программировать в реальном времени? Как это? До тех пор, пока вы не познакомитесь с Лиспом поближе, все эти слова будут звучать для вас несколько противоестественно. Однако с опытом придет и понимание.

1.3. Новый подход

Одна из целей этой книги – не просто объяснить Лисп, но преподнести новый подход к программированию, который возможен благодаря Лиспу. Этот подход заменяет старую идею, предлагающую планировать всю программу заранее перед ее написанием.

Согласно этой модели, ошибок в программе быть не может. Программа, созданная по старательно разработанным заранее спецификациям, работает отлично. Неплохо звучит, вроде. К сожалению, эти спецификации разрабатываются и реализуются людьми, но люди не застрахованы от ошибок и каких-либо упущений, кроме того, тяжело учесть все нюансы на стадии проектирования. В результате такой метод часто не срабатывает.

Руководитель проекта OS/360, Фредерик Брукс, был хорошо знаком с традиционным подходом, а так же с его результатами:°Ё066in

> Любой пользователь OS/360 вскоре начинал понимать, насколько лучше могла быть система. Более того, продукт не успевал за прогрессом, использовал памяти больше запланированного, стоимость его в несколько раз превосходила ожидаемую, и он не работал стабильно до тех пор, пока было выпущено несколько релизов.

Так он описывал систему, ставшую тогда одной из наиболее успешных.

Проблема в том, что такой подход не учитывает человеческий фактор. Его идея в том, что спецификации можно каким-то образом оттранслировать в код. Опыт показывает, что это ожидание редко бывает оправдано. Гораздо полезнее предполагать, что спецификация будет реализована с ошибками, а в коде будет полно багов.

Это как раз та идея, на которой построен новый подход. Вместо того, чтобы надеяться на безошибочность программистов, она пытается минимизировать стоимость допущенных ошибок. Благодаря мощным языкам и эффективным инструментам эта стоимость может быть существенно снижена. Кроме того, разработчик больше не удерживается в рамках спецификации и может экспериментировать.

К сожалению, избежать планирования полностью не удастся, да и не к чему. Это определяется риском: если риск велик, то планированию следует уделить достаточно внимания. Мощные инструменты уменьшают риск, поэтому уменьшается и важность планирования. Устройство программы теперь может быть скорректировано с учетом уже имеющегося опыта ее разработки.

Лисп развивался в этом направлении с 1960 года. Вы можете писать прототипы на Лиспе настолько быстро, что получаете возможность испробовать несколько разных путей реализации раньше, чем закончили бы составлять ее спецификацию. О тонкостях также переживать не обязательно, ведь все тонкости реализации будут осознаны в процессе создания программы. Таким образом, переживания о багах пока можно отложить в сторону. В силу функционального подхода многие баги имеют локальный характер. Некоторые баги (переполнения буфера, висящие указатели) просто невозможны, а остальные баги легче поймать, потому что программа становится короче. Их также легко исправить благодаря интерактивной разработке, когда для запуска программы не требуется писать ее целиком.

Такой подход появился не просто так, он приносит результат. Как бы странно это не звучало, чем меньше планировать разработку, тем стройнее получится программа. Забавно, но такая тенденция наблюдается не только в программировании. В средние века, до изобретения масляных красок, художники пользовались особым материалом – темперой, которая не могла быть перекрашена или осветлена. Стоимость ошибки была настолько велика, что художники боялись экспериментировать. Изобретение масляных красок породило множество течений и стилей в живописи. Масло «позволяет думать дважды».°Ё067in

Введение в обиход масляных красок не просто облегчило жизнь художникам. Стали доступными новые, прогрессивные идеи. Янсон писал:

> Без масла покорение фламандскими мастерами визуальной реальности было бы крайне затруднительным. С технической точки зрения, они являются

прародителями современной живописи, потому что масло является неотъемлемой ее частью.°Ё068in

С точки зрения внешнего вида темпера не уступает маслу. Но широта полета фантазии является решающим фактором.

В программировании наблюдается похожая идея. Но это вовсе не означает, что через несколько лет все программисты перейдут на Лисп. Для перехода к масляным краскам тоже потребовалось существенное время. Кроме того, по разным соображениям, темпера используется и по сей день. Лисп в настоящее время используется в университетах, исследовательских лабораториях, некоторых компаниях, лидирующих в области софт-индустрии. А идеи, которые легли в основу Лиспа (интерактивность, сборка мусора, динамическая типизация, например), все больше и больше заимствуются в современные языки.

2. Лисп приветствует вас

Цель этой главы – помочь вам начать программировать как можно скорее. Прочитав ее, вы узнаете достаточно, чтобы начать писать программы на Common Lisp.

2.1. Внешний вид

Лисп интерактивен, и это еще одна причина, по которой стоит писать программы, чтобы выучить его. Любая лисп-система имеет интерактивный интерфейс, называемый *toplevel*. Программист набирает выражения в *toplevel*, а система вычисляет их значения.

Чтобы сообщить, что система готова получать новые выражения, она выводит приглашение. Часто в качестве такого приглашения используется символ `>`. Мы тоже будем пользоваться им.

Одними из наиболее простых выражений в Лиспе являются целые числа. Если ввести `1` после приглашения,

```
> 1
1
>
```

система напечатает его значение, после чего выведет очередное приглашение, ожидая новых выражений.

В данном случае введенное выражение выглядит так же, как и полученное значение. Такие выражения называют самовычисляемыми. Числа (например, `1`) - самовычисляемые объекты. Давайте посмотрим на более интересные выражения, вычисление которых требует некоторых действий. Например, желая сложить два числа, мы напишем

```
> (+ 2 3)
5
```

В выражении `(+ 2 3)` `+` - это оператор, а числа `2` и `3` - его аргументы.

В другой раз вы бы написали это выражение как `2+3`, но в Лиспе мы располагаем оператор `+` вначале, следом за ним аргументы, а все выражение заключаем в скобки. Это принято называть *префиксной* нотацией, так как оператор располагается спереди. Это может показаться странным, но именно такому способу записи выражений Лисп обязан своими возможностями.

Например, чтобы сложить три числа, вам пришлось воспользоваться оператором сложения дважды:

```
2 + 3 + 4
```

тогда как в Лиспе вы всего лишь добавляете еще один аргумент:

```
(+ 2 3 4)
```

В обычной нотации[1] `+` имеет два аргумента, один перед ним и один после. Префиксная запись дает большую гибкость, позволяя оператору иметь любое количество аргументов, или вообще ни одного.

```
> (+)
0
> (+ 2)
2
> (+ 2 3)
5
> (+ 2 3 4)
9
> (+ 2 3 4 5)
```

[1] Ее также называют *инфиксной*. – *Прим. перев.*

```
14
```

Раз аргументов может быть произвольное число, нужен способ показать, где начинается и заканчивается выражение. Для этого мы ставим скобки.

Выражения могут быть вложенными:

```
> (/ (- 7 1) (- 4 2))
3
```

Здесь мы делим разность 7 и 1 на разность 4 и 2.

Все выражения в Лиспе - либо «атомы» (например, `1`), либо списки, состоящие из произвольного количества выражений. `2`, `(+ 2 3)`, `(+ 2 3 4)`, `(/ (- 7 1) (- 4 2))` - все это нормальные выражения. В дальнейшем вы увидите, что весь код в Лиспе имеет такой вид. Языки типа Си имеют более сложный синтаксис: арифметические выражения имеют инфиксную запись, вызовы функций записываются с помощью разновидности префиксной нотации, их аргументы разделяются запятыми, выражения отделяются друг от друга с помощью точки с запятой, а блоки кода выделяются фигурными скобками. В Лиспе все мысли можно

Обработка ошибок

Если вы пытаетесь вычислить что-то, чего Лисп не понимает, вы получите сообщение об ошибке и попадете в *break loop*, а не в toplevel. Break loop дает опытному программисту возможность выяснить причину ошибки, но вам пока потребуется знать только то, как оттуда выйти. В разных реализациях это может делаться по-разному. В гипотетической реализации для это делает команда `:abort`.

```
> (/ 1 0)
Error: Division by zero.
       Options: :abort, :backtrace
>> :abort
>
```

В приложении A показано, как отлаживать программы на Лиспе, а так же приводятся примеры наиболее распространенных ошибок.

выразить одним и тем же способом.

2.2. Вычисление

В предыдущем разделе мы учились набирать выражения в *toplevel*, и смотрели, как Лисп реагирует на это. В этом разделе мы поближе познакомимся с процессом вычисления.

В Лиспе `+` - это функция, а `(+ 2 3)` - это вызов функции. Результат вызова функции вычисляется следующим образом:Ё051in

1. Сначала вычисляются аргументы, слева направо. В нашем примере все аргументы вычисляются сами в себя, и значениями аргументов являются `2` и `3`.
2. Затем аргументы применяются к функции, задаваемой оператором. В нашем случае, это сложение, которое возвращает `5`.

Аргумент может быть не только самовычисляемым объектом, но и другим вызовом функции. В последнем случае они вычисляются по тем же правилам. Посмотрим, что происходит при вычислении выражения `(/ (- 7 1) (- 4 2))`.

1. Вычисляется `(- 7 1)`: `7` вычисляется в `7`, `1` – в `1`. Эти аргументы передаются функции `-`, которая возвращает `6`.
2. Вычисляется `(- 4 2)`: `4` вычисляется в `4`, `2` – в `2`, функция `-` применяется к этим аргументам и возвращает `2`.
3. Значения `6` и `2` передаются функции `/`, которая возвращает `3`.

Большинство операторов в Common Lisp – это функции, но не все. Вызовы функций всегда обрабатываются подобным образом. Аргументы вычисляются слева-направо и затем передаются функции, которая возвращает значение всего выражения. Этот порядок называется правилом вычисления.

Тем не менее, существуют операторы, которые не следуют принятому в Common Lisp порядку вычислений. Один из них – это `quote` или оператор цитирования. Будучи специальным оператором, он подчиняется собственному правилу вычисления: ничего не делать. Фактически, `quote` берет один аргумент и возвращает его текстовую запись:

```
> (quote (+ 3 5))
(+ 3 5)
```

Ради удобства в Common Lisp можно заменять `quote` на кавычку. Просто поставив `'` перед цитируемым выражением, вы получите тот же результат.

```
> '(+ 3 5)
(+ 3 5)
```

В явном виде оператор `quote` почти не используется, уступая более короткой записи с кавычкой.

Цитирование в Лиспе дает способ предотвратить вычисление выражения. В следующей главе будет показано, чем такая защита может быть полезна.

2.3. Данные

Лисп предоставляет все типы данных, которые есть большинстве других языков, а также некоторые другие, отсутствующие где-либо еще. С одним типом данных мы уже познакомились. Это *integer* – целое число, записываемое в виде последовательности цифр: `256`. Другой тип данных, который есть в большинстве других языках – строка, представляемая как последовательность символов, окруженная двойными кавычками: `"ora et labora"`. Так же, как и целые числа, строки самовычисляемы.

Есть также и такие типы, которые редко используются в других языках - символы и списки. Символы выглядят как обычные слова. Обычно они преобразуются к верхнему регистру, как бы ни были заданы:

```
> 'Artichoke
ARTICHOKE
```

Символы, как правило, не вычисляются сами в себя, поэтому, чтобы сослаться на символ, его необходимо цитировать, как показано выше.

Список - это последовательность из нуля или более элементов, заключенных в скобки. Эти элементы могут принадлежать любому типу, в том числе, могут быть другими списками. Чтобы Лисп не счел список вызовом функции, его нужно процитировать.

```
> '(my 3 "Sons")
(MY 3 "Sons")
> '(the list (a b c) has 3 elements)
(THE LIST (A B C) HAS 3 ELEMENTS)
```

Обратите внимание, что цитирование предотвращает вычисление всего выражения, включая все его элементы.

Список может быть собран с помощью функции `list`. Как и у любой функции, ее аргументы вычисляются. В следующем примере внутри вызова функции `list` вычисляется значение функции `+`:

```
> (list 'my (+ 2 1) "Sons")
(MY 3 "Sons")
```

Вот тут самое время оценить по заслугам наиболее важную особенность Лиспа. Программы на этом языке выражаются в виде списков. Если приведенные ранее доводы о гибкости и элегантности вас не убедили, что принятая в Лиспе нотация имеет ценность, обратите внимание на этот момент. Фактически, именно эта особенность позволяет Лисп-программам генерировать Лисп-код, что дает возможность разработчику создавать программы, которые пишут программы.

Хотя такие программы не рассматриваются до главы 10, сейчас важно понять связь между выражениями и списками. Как раз для этого нам нужно цитирование. Если

список цитируется, его вычисление возвратит сам список. В противном случае, список будет расценен как код, и будет вычислено его значение.

```
> (list '(+ 2 1) (+ 2 1))
((+ 2 1) 3)
```

Первый аргумент закавычен, и он остается списком. Второй аргумент расценивается как вызов функции, и он превращается в число.

Список может быть пустым. В Common Lisp возможны два представления пустого списка: пара пустых скобок и специальный символ `nil`. Независимо от того, как вы введете пустой список, он будет отображен как `nil`.

```
> ()
NIL
> nil
NIL
```

Вам не обязательно ставить кавычку перед `()`, так как символ `nil` самовычисляем.

2.4. Операции со списками

Построением списков занимается функция `cons`. Если второй ее аргумент – список, она возвращает новый список, в котором первый аргумент становится первым элементом исходного списка.

```
> (cons 'a '(b c d))
(A B C D)
```

Список из одного элемента также может быть создан с помощью `cons` и пустого списка. Функция `list`, с которой мы уже познакомились, всего лишь более удобный способ последовательного использования `cons`.

```
> (cons 'a (cons 'b nil))
(A B)
> (list 'a 'b)
(A B)
```

Простейшие функции для получения отдельных элементов списка – `car` и `cdr`°Ё069in. `Car` списка - это его первый элемент, а `cdr` – все, за исключением него.

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
```

Произвольный элемент списка можно получить, комбинируя `car` и `cdr`. Желая получить третий элемент, вы напишете

```
> (car (cdr (cdr '(a b c d))))
C
```

Тот же результат гораздо проще получить с помощью функции `third`.

```
> (third '(a b c d))
C
```

2.5. Истинность

В Common Lisp истинность обычно представляется символом `t`. Как и `nil`, символ `t` является самовычисляемым. Например, функция `listp` возвращает истину, если ее аргумент – список:

```
> (listp '(a b c))
T
```

Функции, возвращающие истину либо ложь, называют предикатами. В Common Lisp имена предикатов часто оканчиваются на «p».

Ложность в Common Lisp представляется с помощью `nil`, пустого списка. Применяя `listp` не к списку, получим `nil`:

```
> (listp 27)
NIL
```

Поскольку `nil` имеет два значения в Common Lisp, существует функция `null`, имеющая истинное значение для пустого списка:

```
> (null nil)
T
```

и функция `not`, возвращающая истинное значение, если ее аргумент ложен.

```
> (not nil)
T
```

По сути, это одно и то же.

Простейший условный оператор в Common Lisp - `if`. обычно он принимает три аргумента: *test*-, *then*- и *else*-выражения. Сначала вычисляется тестовое выражение. Если оно истинно, следом вычисляется "то"-выражение, после чего возвращается его значение. В противном случае вычисляется "иначе"-выражение.

```
> (if (listp '(a b c))
      (+ 1 2)
      (+ 5 6))
3
> (if (listp 27)
      (+ 1 2)
      (+ 5 6))
11
```

Как и `quote`, `if` - это специальный оператор, а не функция, т.к. для функции вычисляются все аргументы, а у оператора `if` вычисляется лишь одно из двух последних выражений.

Последний аргумент `if` не обязателен. Если он пропущен, то принимается за `nil`.

```
> (if (listp 27)
      (+ 2 3))
NIL
```

Несмотря на то, что по умолчанию истина представляется в виде `t`, любое выражение, кроме `nil` также считается истинным:

```
> (if 27 1 2)
1
```

Логические операторы `and` (и) и `or` (или) действуют похожим образом. Оба они принимают любое количество аргументов, но вычисляют их до тех пор, пока не будет ясно, какое значение необходимо вернуть. Если все аргументы истинны (то есть, не `nil`), то `and` вернет значение последнего:

```
> (and t (+ 1 2))
3
```

Если вдруг один из аргументов окажется ложным, следующие за ним аргументы не будут вычислены. Так же действует и `or`, вычисляя значения аргументов до тех пор, пока не найдется хотя бы одно истинное значение.

Эти два оператора - макросы. Как и специальные операторы, макросы могут обходить общепринятый порядок вычисления. В главе 10 объясняется, как писать собственные макросы.

2.6. Функции

Собственные функции определяются с помощью `defun`. `defun` принимает три или более аргументов: имя, список параметров и одно или более выражений, составляющих тело функции. Определим с помощью `defun` функцию `third`:

```
> (defun our-third (x)
    (car (cdr (cdr x))))
OUR-THIRD
```

Первый аргумент задает имя функции, в нашем примере это `our-third`. Второй аргумент, список `(x)`, сообщает, что функция может принимать строго один аргумент: `x`. Используемый здесь символ `x` называется переменной. Помещая его в список следом за именем функции, мы считаем его ее параметром.

Оставшаяся часть, `(car (cdr (cdr x)))`, называется «телом» функции. Она определяет, какие вычисления требуется произвести, чтобы вычислить значение функции. Вызов `our-third` возвращает `(car (cdr (cdr x)))`, какое бы значение аргумента не было задано.

```
> (our-third '(a b c d))
C
```

Теперь, когда мы познакомились с переменными, будет легче понять, чем являются символы. Это попросту имена переменных, существующие «сами по себе». Как раз поэтому символы, как и списки, нуждаются в цитировании. Так же, как закавычиваются списки, чтобы не были восприняты как код, символы закавычиваются, чтобы не были восприняты как переменные.

Функцию можно рассматривать как обобщение всех Лисп-выражений. Следующее выражение проверяет, превосходит ли сумма чисел `1` и `4` число `3`:

```
> (> (+ 1 4) 3)
T
```

Заменяя конкретные числа переменными, мы можем получить функцию, выполняющую ту же проверку для трех произвольных чисел:

```
> (defun sum-greater (x y z)
    (> (+ x y) z))
SUM-GREATER
> (sum-greater 1 4 3)
T
```

В Лиспе нет различий между программой, процедурой и функцией. Все это функции (да и сам Лисп по большей части состоит из функций). Не имеет смысла определять одну главную функцию[1], любая функция может быть вызвана в toplevel. В числе прочего, это означает, что программу можно тестировать по маленьким кусочкам в процессе ее написания.

2.7. Рекурсия

Функции, рассмотренные в предыдущей главе, вызывала другие функции. Например, `sum-greater` вызывала `+` и `>`. Внутри функции может быть вызов любой функции, включая ее саму.

Функции, вызывающие сами себя, называются рекурсивными. В Common Lisp есть функция `member`, которая проверяет, есть ли в списке какой-либо объект. Ниже приведена ее упрощенная реализацияЁ014in.

```
(defun our-member (obj lst)
  (if (null lst)
      nil
      (if (eql (car lst) obj)
          lst
          (our-member obj (cdr lst)))))
```

Предикат `eql` проверяет два аргумента на идентичность. Все остальное в этом выражении вам должно быть уже знакомо.

```
> (our-member 'b '(a b c))
(B C)
> (our-member 'z '(a b c))
NIL
```

[1] Как, например, в Си, где требуется введение основной функции `main` – *Прим. перев.*

Опишем словами, что делает эта функция. Чтобы проверить, есть ли `obj` в списке `lst`, мы

1. Проверяем, пуст ли список `lst`. Если он пуст, это значит, что `obj` не присутствует в списке.
2. Если `obj` эквивалентен первому элементу `lst`, значит, он в списке есть.
3. В другом случае проверяем, есть ли `obj` в остатке списка `lst`.

Подобное описание порядка действий будет полезным, если вы захотите понять, как работает рекурсивная функция.

Поначалу понимание рекурсии может оказаться трудной задачей. Причина этому - ошибочное понимание сути функции. Часто считается, что функция – это особый агрегат, который получает параметры в качестве сырья, перепоручает часть работы другим функциям-агрегатам и в итоге собирает готовый продукт – возвращаемое значение. При таком рассмотрении возникает парадокс – как агрегат может перепоручать работу сам себе, если он уже занят?

Более подходящее сравнение для функции – процесс. Для процесса рекурсия вполне естественна. В повседневной жизни мы часто наблюдаем, не задумываясь, рекурсивные процессы. Посмотрим на историка, которому интересна динамика численности населения в Европе. Процесс изучения соответствующих документов будет выглядеть следующим образом:

1. Получить копию документа
2. Найти в нем информацию об изменении численности населения
3. Если в нем также упоминаются иные источники, также изучить и их.

Этот процесс понять легко, несмотря на то, что он рекурсивен, ведь третий его шаг может заканчиваться точно такими же процессами.

Функцию `our-member` тоже не следует считать эдаким агрегатом. Это всего лишь набор правил, позволяющих выполнить какие-либо операции со списком. В этом свете парадокс исчезает.°Ё070in

2.8. Чтение Лиспа

Определенная в предыдущем разделе функция заканчивается пятью закрывающими скобками. Более сложные функции могут заканчиваться семью-восемью скобками. Это часто отталкивает людей, начинающих изучать Лисп. Как это можно читать, не говоря уже о том, чтобы писать самому? Как искать в таком коде парные скобки?

В действительности, никто так не делает. Вместо того, чтобы искать парные скобки, Лисп-программисты используют отступы и выравнивание. Любой уважающий себя текстовый редактор умеет подсвечивать парные скобки. Если ваш редактор не делает этого, включите эту опцию, иначе писать Лисп-код в нем будет практически невозможно.[1]

С хорошем редактором, поиск парных скобок перестанет быть затруднительным. Кроме того, благодаря общепринятым соглашениям касательно выравнивания кода, читать код очень легко, не обращая внимания на скобки.

Любой Лисп-хакер с трудом разберет определение `our-member`, если оно буде записано в таком стиле:

```
(defun our-member (obj lst) (if (null lst) nil (if (eql (car lst
) obj) lst (our-member obj (cdr lst)))))
```

С другой стороны, правильно выровненный код будет легко читаться даже без скобок:

```
defun our-member (obj lst)
  if (null lst)
     nil
     if eql (car lst) obj
        lst
        our-member obj (cdr lst)
```

[1] В vi эта опция включается командой `:set sm`. В Emacs `M-x lisp-mode` будет отличным выбором.

Это может быть полезным при написании кода на бумаге. В редакторе вы сможете воспользоваться удобной возможностью поиска парных скобок.

2.9. Ввод и Вывод

До сих пор мы осуществляли ввод-вывод, вводя команды в toplevel. Чтобы программа была по-настоящему интерактивна, этого явно недостаточно. В этом разделе мы рассмотрим несколько функций ввода-вывода.

Основная функция вывода в Common Lisp – `format`. Она принимает два или более аргументов: первый определяет, куда будет печататься результат, второй – это строковый шаблон, остальные аргументы – объекты, которые будут вставляться в нужные позиции заданного шаблона. Вот типичный пример:

```
> (format t "~A plus ~A equals ~A.~%: 2 3 (+ 2 3))
2 plus 3 equals 5
NIL
```

Обратите внимание, что получено два значения. Первая строка - результат выполнения `format`, напечатанный в toplevel. Обычно функции типа `format` не вызываются напрямую в toplevel, а используются внутри программ, и их значения не видны.

Первый аргумент, `t`, означает направление вывода по умолчанию. Обычно это toplevel. Во втором аргументе - строке, `~A` определяют позиции, куда будут вставлены аргументы `format`, следующие за строкой, `~%` соответствует переносу строки.

Стандартная функция чтения – `read`. (которую также называют считывателем) Вызванная без аргументов, она выполняет чтение из умолчального источника, обычно - toplevel. Ниже приведена функция, которая предлагает ввести любое значение и затем печатает его.

```
(defun askem (string)
  (format t "~A" string)
  (read))
> (askem "How old are you? ")
How old are you? 29
29
```

Учтите, что для завершения чтения `read` будет ожидать, пока вы не нажмете `Enter`. Поэтому не стоит использовать функцию `read`, не печатая перед этим приглашение к вводу, иначе может показаться, что программа зависла, хотя она просто ожидает ввода.

Другая вещь, которую следует знать о функции `read` - это поистине мощный инструмент. Фактически, это полноценный обработчик Лисп-выражений. Она не просто читает символы и возвращает их в виде строки. Она обрабатывает введенное выражение и возвращает полученный лисповый объект. В последнем примере она возвращает число.

Несмотря на свою краткость, определение `askem` содержит нечто, чего мы до сих пор не встречали - тело функции состоит из нескольких выражений. Вообще говоря, оно может содержать любое количество выражений. При вызове функции они вычисляются последовательно, и возвращается значение последнего выражения.

Во всех предыдущих главах мы придерживались "*чистого*" (*pure*) Лиспа, то есть, Лиспа без *побочных эффектов* (*side effects*). Побочный эффект - это событие, которое любым образом изменяет состояние системы. При вычислении выражения `(+ 1 2)` возвращается значение `3`, но никаких побочных эффектов не происходит. В последнем примере побочный эффект производится функцией `format`, которая не только возвращает значение, но и печатает что-то. Это одна из разновидностей побочных эффектов.

Если мы зададимся целью писать код без побочных эффектов, то будет бессмысленно писать функции, состоящие из нескольких выражений, так как в качестве значения функции будет использоваться значение последнего аргумента, а значения предыдущих выражений будут потеряны. Если эти выражения не будут

вызывать побочных эффектов, то получить результат их вычисления просто не получится.

2.10. Переменные

Один из наиболее часто используемых операторов в Common Lisp - это `let`, который позволяет ввести локальные переменные.

```
> (let ((x 1) (y 2))
    (+ x y))
3
```

Выражение с `let` состоит из двух частей. Первая содержит инструкции, определяющие новые переменные. Каждая такая инструкция содержит имя переменной и соответствующее ей выражение. Чуть выше мы создали две новые переменные, `x` и `y`, которым были даны значения `1` и `2`. Эти переменные действительны внутри всего тела вызова `let`.

Тело `let` содержит выражения, которые будут вычислены по-порядку. В нашем случае имеется только одно выражение, `(+ x y)`. Значением всего вызова `let` будет значение последнего выражения.

Давайте напишем более селективный вариант функции `askem` с использованием `let`: Ё028in

```
(defun ask-number ()
  (format t "Please enter a number. ")
  (let ((val (read)))
    (if (numberp val)
        val
        (ask-number))))
```

Мы создаем переменную `val`, содержащую результат вызова `read`. Сохраняя это значение, мы можем проверить, что было прочитано. Как вы уже догадались, `numberp` – это предикат, проверяющий, является ли его аргумент числом.

Если введено не число, `ask-number` вызовет саму себя, чтобы пользователь повторил попытку. Так будет повторяться до тех пор, пока `read` не получит число.

```
> (ask-number)
Please enter a number. a
Please enter a number. (no hum)
Please enter a number. 52
52
```

Переменные типа `val`, создаваемые оператором `let`, называются *локальными*, то есть, действительными в определенной области. Есть также переменные, которые действительны везде[1] - *глобальные*.

Глобальная переменная может быть создана с помощью `defparameter`:

```
> (defparameter *glob* 99)
*GLOB*
```

Хотя такие переменные доступны везде, кроме создаваемых `let` локальных контекстов, в которых определена локальная переменная с таким же именем. Чтобы избегать таких случайных совпадений, принято давать глобальным переменным имена, окруженные звездочками.

Также в глобальном окружении можно задавать константы[2]:

```
(defconstant limit (+ *glob* 1))
```

Константам нет смысла давать имена, окруженные звездочками, потому что попытка определить переменную с таким же именем закончится ошибкой. Чтобы

[1] Реальная разница между локальными и глобальными переменными будет пояснена в главе 6.

[2] Как и для специальные переменных, для определяемых пользователем констант существует негласное правило, согласно которому их имена следует окружать знаками `+`. Следуя этому правилу, мы бы определили константу `+limit+`. – *Прим. перев.*

узнать, соответствует ли какое-то имя глобальной переменной или константе, воспользуйтесь boundp.

```
> (boundp '*glob*)
T
```

2.11. Присвоение

В Common Lisp наиболее общим средством присвоения значений является setf. Он может присваивать значения переменным любых типов.

```
> (setf *glob* 98)
98
> (let ((n 10))
    (setf n 2)
    n)
2
```

Если setf пытается присвоить значение переменной, которая в данном контексте не является локальной, она будет определена как глобальная.

```
> (setf x (list 'a 'b 'c))
(A B C)
```

Таким образом, глобальные переменные можно определять неявно, с помощью setf. Однако задавать их явно с помощью defparameter считается хорошим тоном.

2.12. Функциональное программирование

Ё041inТермин функциональное программирование означает, что программы не изменяют ничего, а только возвращают значения. Это основная парадигма в Лиспе. Большинство встроенных функций в Лиспе не вызывают побочных эффектов.

Ё013inНапример, функция remove принимает список и объект и возвращает новый список, состоящий из тех же элементов, что и предыдущий, за исключением этого объекта.

```
> (setf lst '(c a r a t))
(C A R A T)
> (remove 'a lst)
(C R T)
```

Почему бы не сказать, что remove просто удаляет элемент из списка? Потому что она этого не делает, оставляя исходный список неизменным:

```
> lst
(C A R A T)
```

А что если необходимо именно удалить элементы из списка? В Лиспе принято предоставлять список в качестве аргумента какой-либо функции, и присваивать возвращаемое ей значение с помощью setf. Чтобы удалить все a из списка x, мы скажем:

```
(setf x (remove 'a x))
```

В функциональном программировании избегают использования setf и других подобных вещей. Поначалу сложно вообразить, что это попросту возможно, не говоря уже о том, что желательно. Как можно создавать программы только с помощью возвращения значений?

Да, совсем без побочных эффектов работать неудобно. Тем не менее, читая дальше эту книгу, вы будете все сильнее удивляться тому, как же в сущности немного побочных эффектов необходимо. И чем меньше вы будете ими пользоваться, тем будет лучше.

Использование функционального программирования дает серьезное преимущество: интерактивное тестирование. Вы можете тестировать функции, написанные в функциональном стиле, сразу же после ее написания, и вы будете уверены, что

результат ее выполнения всегда будет одинаковым. Вы можете изменять одну часть программы, но это не повлияет на корректность работы другой ее части. Такая возможность безбоязненно изменять код открывает новый стиль программирования. Это похоже на удобство телефона, который по сравнению с письмами предоставляет новый способ общения.

2.13. Итерации

Когда вы желаете повторять что-то многократно, рекурсия не самый удобный способ. Типичный пример использования итерации – генерация таблиц. Следующая функция: Ё017in

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))
```

печатает квадраты целых чисел от `start` до `end`:

```
> (show-squares 2 5)
2 4
3 9
4 16
5 25
DONE
```

Макрос `do` - наиболее общий итерационный оператор в Common Lisp. Как и `let`, первый аргумент `do` задает переменные. Каждый элемент этого списка может выглядеть как

```
(variable initial update)
```

где `variable` - символ, а `initial` и `update` - выражения. Исходное значение каждой переменной определяется значением `initial`, и на каждой итерации это значение будет меняться в соответствии со значением `update`. В примере `show-squares` `do` использует только одну переменную, `i`. на первой итерации значение `i` равно `start`, и после каждой успешной итерации оно будет увеличиваться на один.

Второй аргумент `do` - список по меньшей мере из одного выражения, которое определяет, когда следует остановиться. В нашем примере это проверка `(> i end )`. Остальные выражения будут вычислены после завершения итераций, и будет возвращено значение последнего из них. Функция `show-squares` всегда возвращает `done`.

Оставшиеся аргументы `do` составляют тело цикла. Они будут вычисляться по-порядку на каждой итерации. На каждой итерации после модифицирования значений переменных и проверки условия окончания цикла.

Для сравнения запишем рекурсивную версию `show-squares`:Ё022in

```
(defun show-squares (i end)
  (if (> i end)
      'done
      (progn
        (format t "~A ~A~%" i (* i i))
        (show-squares (+ i 1) end))))
```

В ней мы использовали новый оператор – `progn`. Он принимает любое количество аргументов, вычисляет их один за другим, возвращая значение последнего.

Для частных случаев в Common Lisp есть более простые итерационные операторы. Вот, например, функция, определяющая длину списка.

```
(defun our-length (lst)
  (let ((len 0))
    (dolist (obj lst)
      (setf len (+ len 1)))
    len))
```

В этом примере `dolist` принимает первый аргумент вида `(variable expression)` и следующий за ним набор выражений, которые вычисляются для каждого элемента списка, значение которых по очереди присваивается переменной. В этом примере для каждого `obj` в `lst` увеличивается значение `len`.

Рекурсивная версия этой функции:

```
(defun our-length (lst)
  (if (null lst)
      0
      (+ (our-length (cdr lst)) 1)))
```

Если список пуст, его длина `0`, иначе она на `1` больше длины `cdr` списка. Эта версия `our-length` более ясная, однако она не использует хвостовую рекурсию (раздел 13.2) и поэтому неэффективна.

2.14. Функции как объекты

В Лиспе функции - самые обычные объекты, как символы, строки или списки. Давая функции имя с помощью `function`, мы получаем ассоциированный объект. `Function`, как и `quote` - это специальный оператор, и нам не нужно закавычивать его аргумент:

```
> (function +)
#<Compiled-Function + 17BA4E>
```

Таким странным образом отображаются функции в типичной реализации Common Lisp.

До сих пор мы имели дело только с такими объектами, которые при печати отображаются так же, как были введены. Это соглашение не распространяется на функции. Встроенная функция `+` обычно является куском машинного кода. В каждой реализации Common Lisp может быть свой способ отображения функций.

Аналогично использованию кавычки вместо `quote`, возможно употребление `#'` как сокращение для `function`:

```
> #'+
#<Compiled-Funxtion + 17BA4E>
```

Как и любой объект, функция может служить аргументом. Пример функции, аргументом которой является функция, служит `apply`. Она требует функцию и список ее аргументов, и возвращает результат вызова этой функции с заданными аргументами:

```
> (apply #'+ '(1 2 3))
6
> (+ 1 2 3)
6
```

`Apply` принимает любое количество аргументов, но последний из них должен быть списком.

```
> (apply #'+ 1 2 '(3 4 5))
15
```

Функция `funcall` делает то же самое, но не требует, чтобы аргументы были упакованы в список:

```
> (funcall #'+ 1 2 3)
6
```

Обычно функции создаются с помощью макроса `defun`, который также дает функции имя. Но функция не обязательно должна иметь имя, и для их определения мы не обязаны использовать `defun`. Как и к большинству других объектов, мы можем обращаться к функции побуквенно.

Чтобы побуквенно сослаться на число, мы используем последовательность цифр. Чтобы так же сослаться на функцию, мы используем *лямбда-выражение*. Лямбда-выражение - это список, содержащий символ `lambda` и следующие за ним список параметров и тело, состоящее из произвольного количества аргументов.

Ниже приведено лямбда-выражение, представляющее функцию, складывающую два числа и возвращающее их сумму:

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

Что такое Лямбда?

В лямбда-выражении `lambda` - не оператор. Это просто символ°Ë071in, В ранних диалектах Лиспа он имел свою цель: функции имели внутреннее представление в виде списков, и единственным способом отличить функцию от обычного списка была проверка, является ли первый его элемент символом `lambda.`

В Common Lisp функции задаются в виде списка, но имеют отличное от списка внутренне представление. И `lambda` больше не необходимо. Было бы вполне возможно помечать функции, например, так:

```
((x) (+ x 100))
```

вместо

```
(lambda (x) (+ x 100))
```

но Лисп-программисты привыкли начинать функции символом `lambda`, и Common Lisp также следует этой традиции.

Список `(x y)` содержит параметры, за ним следует тело функции.

Лямбда-выражение можно считать именем функции. Как и обычное имя, лямбда-выражение может быть первым элементом выражения, вызывающего функцию:

```
> ((lambda (x) (+ x 100)) 1)
101
```

а применение оператора `#'` дает соответствующую функцию:

```
> (funcall #'(lambda (x) (+ x 100))
           1)
101
```

Помимо прочего, такая запись позволяет использовать функции, не присваивая им имена.

2.15. Типы

Лисп обладает необыкновенно гибкой системой типов. Во многих языках для каждой переменной перед ее использованием необходимо задать конкретный тип. В Common Lisp типы имеют значения, но не переменные. Представьте, что каждый объект имеет метку, определяющую его тип. Такой подход называется динамической типизацией. Раз переменная может содержать объект любого типа, нет необходимости декларировать типы переменных.

Несмотря на то, что декларации типов вовсе не обязательны, вы можете задавать их из соображений производительности. Декларации типов обсуждаются в разделе 13.3.

В Common Lisp встроенные типы образуют иерархию подтипов и надтипов. Объект всегда имеет несколько типов. Например, число `27` соответствует типам `fixnum`, `integer`, `rational`, `real`, `number`, `atom` и `t`, в порядке увеличения общности. (Численные типы обсуждаются в главе 9.) Тип `t` - это самый верхний тип во всей иерархии, поэтому каждый объект имеет тип `t`.

Функция typep определяет, принадлежит ли ее первый аргумент типу, заданному вторым аргументом:

```
> (typep 27 'integer)
T
```

Мы познакомимся с самыми разными встроенными типами по мере изучения Common Lisp.

2.16. Забегая вперед

В этой главе нам удалось едва коснуться поверхности Лиспа. В дальнейшем внешность этого необычного языка будет проясняться. Начнем с того, что язык имеет единый синтаксис для записи всего кода. Синтаксис основан на списках - одном из объектов Лиспа. Функции, которые также являются лисповыми объектами, могут быть представлены в виде списков. А Лисп сам по себе - программа на Лиспе, почти полностью состоящая из встроенных Лисп-функций, которые не отличаются от тех функций, которые вы можете определить самостоятельно.

Не смущайтесь, если эта цепочка взаимоотношений пока не совсем ясна. Лисп содержит настолько много новых идей, что их осознание требует времени. Но одно должно быть ясно: некоторые эти идеи изначально довольно элегантны.

Ричард Гэбриэл однажды в полушуточной форме описал Си как язык для написания Unix°Ё072in. Также и мы можем назвал Лисп языком для написания Лиспа. Но между этими утверждениями есть существенная разница. Язык, который легко написать на самом себе, кардинально отличается от языка, годного для написания каких-то отдельных классов программ. Он открывает новый способ программирования: вы можете в равной мере писать программы на таком языке и совершенствовать язык в соответствии с требованиями этой программы. Чтобы понять суть программирования на Лиспе, вы должны начать с этой мысли.

Итоги главы

1. Лисп интерактивен. Вводя выражение в toplevel, вы тут же получаете его значение.
2. Программы на Лиспе состоят из выражений. Выражение может быть атомом или списком, состоящим из оператора и следующих за ним аргументов. Благодаря префиксной записи этих аргументов может быть сколько угодно.
3. Порядок вычисления вызовов функций в Common Lisp: сначала вычислить аргументы слева направо, затем предоставить их оператору. Оператор `quote` не подчиняется этому порядку и возвращает само выражение вместо его значения.
4. Помимо привычных типов данных в Лиспе есть символы и списки. Программы на Лиспе записываются в виде списков, поэтому легко составлять программы, которые пишут другие программы.
5. Три основные функции для обращения со списками: `cons`, строящая список; `car`, возвращающая первый элемент списка; `cdr`, возвращающая весь список за исключением его первого элемента.
6. В Common Lisp символ `t` имеет истинное значение, `nil` – ложное. В контексте логических операций все, кроме `nil`, является истинным. Основной условный оператор - `if`. Операторы `and` и `or` также могут считаться логическими.
7. Сам Лисп состоит по большей части из функций. Собственные функции можно создавать с помощью `defun`.
8. Функция, вызывающая сама себя, называется рекурсивной. Рекурсивную функцию следует понимать как процесс, но не как автомат.
9. Скобки не создают затруднений, для чтения и написания Лисп-кода пользуются выравниванием кода.
10. Основные функции ввода/вывода: `read`, являющаяся полноценным обработчиком Лисп-выражений, и `format`, использующая для вывода заданный шаблон.
11. Локальные переменные создаются с помощью `let`, глобальные – с помощью `defparameter`.
12. Для присвоения используется оператор `setf`. Его первый аргумент может быть как переменной, так и выражением.
13. Основной парадигмой Лиспа является функциональное программирование, то есть, написание кода без побочных эффектов.
14. Основной итерационный оператор – `do`.
15. Функции являются обычными объектами в Лиспе. Они могут записываться в виде лямбда-выражений, а так же использоваться как аргументы для других функций.

16. В Лиспе не переменные, а значения имеют типы.

Упражнения

1. Опишите, что происходит при вычислении следующих выражений:

```
(a) (+ (- 5 1) (+ 3 7))
(b) (list 1 (+ 2 3))
(c) (if (listp 1) (+ 1 2) (+ 3 4))
(d) (list (and (listp 3) t) (+ 1 2))
```

2. Составьте с помощью `cons` три различных выражения, создающие список `(a b c)`.
3. С помощью `car` и `cdr` определите функцию, возвращающую четвертый элемент списка.
4. Определите функцию, принимающую два аргумента и возвращающую наибольший.
5. Что делают следующие функции?Ё019in

```
(a) (defun enigma (x)
      (and (not (null x))
           (or (null (car x))
               (enigma (cdr x)))))
(b) (defun mystery (x y)
      (if (null y)
          nil
          (if (eql (car y) x)
              0
              (let ((z (mystery x (cdr y))))
                (and z (+ z 1))))))
```

6. Что может стоять на месте `x` в следующих выражениях?

```
(a) > (car (x (cdr '(a (b c) d))))
    B
(b) > (x 13 (/ 1 0))
    13
(c) > (x #'list 1 nil)
    (1)
```

7. Определите функцию, проверяющую, является ли списком хотя бы один элемент списка. Пользуйтесь только теми операторами, которые были упомянуты в этой главе.
8. Предложите итеративное и рекурсивное определение функции, которая

(a) печатает такое количество точек, которое равно заданному положительному целому числу.

(b) возвращает количество символов `a` в заданном списке.

9. Ваш товарищ пытается написать функцию, которая суммирует все значения элементы списка, кроме `nil`. Он написал две версии такой функции, но ни одна из них не работает. Объясните, что не так в каждой из них, и предложите корректную версию.

```
(a) (defun summit (lst)
      (remove nil lst)
      (apply #'+ lst))
(b) (defun summit (lst)
      (let ((x (car lst)))
        (if (null x)
            (summit (cdr lst))
            (+ x (summit (cdr lst))))))
```

3. Списки

Списки - основной тип данных в Лиспе. В ранних диалектах кроме списков не было других данных, откуда и пошло название: «LISt Processor». Нынешний Лисп уже не соответствует этому акрониму. Common Lisp является языком общего назначения и предоставляет широкий набор типов данных.

Процесс разработки Лисп-программ часто напоминает эволюцию самого Лиспа. Первоначально программа использует множество списков. Однако в более ее поздних версиях разработчик начинает использовать более сложные и эффективные типы данных.

В этой главе подробно описываются операции со списками, а также с их помощью поясняются некоторые довольно общие концепции.

3.1. Ячейки

В разделе 2.4 упомянуты `cons`, `car` и `cdr` – простейшие функции для манипуляций со списками. В действительности, `cons` объединяет два объекта в один, называемый ячейкой (`cons`). Если быть точнее, то `cons` - это пара указателей; первый указывает на `car`, второй на `cdr`.

`Cons`-ячейки дают удобный способ объединения в пару объектов любых типов, в том числе, других ячеек. Именно благодаря такой возможности с помощью `cons` можно строить произвольные списки.

Незачем представлять каждый список в виде `cons`-ячеек, но нужно помнить, что они могут быть заданы таким образом. Любой непустой список может считаться парой, содержащей первый элемент списка и его остаток без первого элемента. В Лиспе списки являются воплощением этой идеи. Именно поэтому `car` позволяет получить первый элемент списка, а `cdr` – его остаток. Таким образом, списки – это не отдельный тип объектов, а всего лишь набор связанных `cons`-ячеек.

Если мы попытаемся использовать `cons` вместе с `nil`,

```
> (setf x (cons 'a nil))
(A)
```

то мы получим список, состоящий из одной ячейки, как показано на рис. 3.1. Такое изображение ячеек называется «блочным», потому что каждая ячейка представляется в виде блока, содержащего указатели на `car` и `cdr`. Вызывая `car` или `cdr` мы получаем объект, на который указывает соответствующий указатель:

Рис. 3.1. Список, состоящий из одной ячейки

```
> (car x)
A
> (cdr x)
NIL
```

Составляя список из нескольких элементов, мы получаем цепочку ячеек.

```
> (setf y (list 'a 'b 'c))
(A B C)
```

Эта структура показана на рис. 3.2. Теперь `cdr` списка будет указывать на список из двух элементов:

Рис. 3.2. Список из трех ячеек

```
> (cdr y)
(B C)
```

Элементами списка могут быть любые объекты, в том числе, другие списки:

```
> (setf z (list 'a (list 'b 'c) 'd))
(A (B C) D)
```

Соответствующая структура показана на рис. 3.3; `car` второй ячейки указывает на другой список.

Рис. 3.3. Вложенный список

```
> (car (cdr z))
(B C)
```

В этих двух примерах списки состояли из трех элементов. Просто так сложилось, что в последнем примере один из них – тоже список. Такие списки называют «вложенными». Списки, не содержащие внутри себя подсписков, называют плоскими.

Проверить, является ли объект `cons`-ячейкой, можно с помощью `consp`. С ее помощью можно определить `listp`:

```
(defun out-listp (x)
  (or (null x) (consp x)))
```

Теперь определим предикат `atom`, учитывая, что атомами в Лиспе считается все, кроме `cons`-ячеек:

```
(defun our-atom (x) (not (consp x)))
```

Исключением из этого правила считается `nil` - атом и список одновременно.

3.2. Сопоставление

Каждый раз, когда вы вызываете `cons`, Лисп выделяет память для двух указателей. Это значит, что, вызывая `cons` дважды с одними и теми же аргументами, мы получим два значения, которые будут выглядеть идентично, но соответствовать разным объектам:

```
> (eql (cons 'a nil) (cons 'a nil))
NIL
```

Функция `eql`[1] возвращает `t`, если сравниваемые значения соответствуют одному объекту в памяти Лиспа.

```
> (setf x (cons 'a nil))
(A)
> (eql x x)
T
```

Для проверки идентичности списков (и других объектов) используется `equal`. Два одинаково выглядящих объекта равны с точки зрения предиката `equal`:

```
> (equal x (cons 'a nil))
T
```

Определим функцию `equal` для частного случая сопоставления списков[2], предполагая, что если два объекта равны для предиката `eql`, то будут равны и для `equal`.

```
(defun our-equal (x y)
  (or (eql x y)
      (and (consp x)
           (consp y)
           (our-equal (car x) (car y))
           (our-equal (cdr x) (cdr y)))))
```

3.3. Почему в Лиспе нет указателей

Чтобы понять, как устроен Лисп, необходимо осознать, что механизм присвоения значения переменным похож на построение списков из объектов. Переменной

[1] В ранних диалектах Лиспа задачу `eql` выполнял `eq`. В Common Lisp `eq` – более строгая функция, а основным предикатом проверки идентичности является `eql`. Роль `eq` разъясняется на Ё003out.

[2] Функция `our-equal` применима не к любым спискам, а только к спискам символов. Неточность в оригинале указана Биллом Стретфордом. – *Прим. перев.*

соответствует указатель на ее значение, также как cons-ячейка имеют указатели на car и cdr.

Некоторые другие языки позволяют оперировать указателями явно. Лисп же выполняет всю работу с указателями самостоятельно. Мы уже видели, как это происходит, на примере списков. Нечто похожее происходит и с переменными. Пусть две переменные указывают на один и тот же список:

```
> (setf x '(a b c))
(A B C)
> (setf y x)
(A B C)
```

Что происходит, когда мы пытаемся присвоить y значение x? Место в памяти, связанное с переменной x, содержит не сам список, а указатель на него. Чтобы присвоить переменной y то же значение, достаточно просто скопировать этот указатель (рис. 3.4). В этом примере две переменные будут одинаковыми с точки зрения equal.

Рис. 3.4. Две переменные, указывающие на один список

```
> (eql x y)
T
```

Таким образом, в Лиспе указатели явно не используются, потому что любое значение - по сути указатель. Когда вы присваиваете значение переменной или располагаете его в участке какой-либо структуры, вы присваиваете указатель на это значение. Когда вы запрашиваете содержимое какой либо переменной, Лисп возвращает данные, на которые ссылается указатель. Но это происходит неявно, и вам не приходится заботиться об указателях.

Из соображений производительности Лисп иногда использует иные представления данных, нежели указатели. Например, небольшие целые числа занимают места не более, чем указатель, и некоторые реализации Лиспа оперируют непосредственно целыми числами, а не указателям на них. Но для программиста важно лишь то, что он может без дополнительных деклараций положить что угодно куда угодно.

3.4. Построение списков

Ё006inФункция copy-list возвращает копию списка. Новый список дублируется в памяти и идентичен исходному.

```
> (setf x '(a b c)
        y (copy-list x))
(A B C)
```

Такая структура ячеек показана на рис. 3.5. Собственную функцию copy-list определим так:

Рис. 3.5. Результат копирования

```
(defun our-copy-list (lst)
  (if (atom lst)
      lst
      (cons (car lst) (our-copy-list (cdr lst)))))
```

Здесь подразумевается, что x и (copy-list x) всегда равны с точки зрения equal, и равны для eql только если x – nil.

Еще одна функция для работы со списками – append, склеивающая между собой произвольное количество списков.

```
> (append '(a b) '(c d) '(e))
(A B C D E)
```

Append копирует все аргументы, кроме последнего.

3.5. Пример: сжатие

В этом разделе приводится пример разработки простого механизма сжатия списков. Рассматриваемый ниже алгоритм принято называть *кодированием повторов* (*run-length encoding*). Представьте ситуацию в ресторане:

Официант: Что вы будете?

Первый посетитель: Принесите фирменное блюдо, пожалуйста.

Второй посетитель: И мне тоже.

Третий посетитель: Ну и я за компанию.

Все смотрят на четвертого клиента: А я бы предпочел кориандровое суфле.

Официант со вздохом разворачивается и относит повару запись: три фирменных и одно кориандровое суфле.

На рис. 3.6 показано, как можно изобразить подобный алгоритм для списков.

```
(defun compress (x)
  (if (consp x)
      (compr (car x) 1 (cdr x))
      x))

(defun compr (elt n lst)
  (if (null lst)
      (list (n-elts elt n))
      (let ((next (car lst)))
        (if (eql next elt)
            (compr elt (+ n 1) (cdr lst))
            (cons (n-elts elt n)
                  (compr next 1 (cdr lst)))))))
(defun n-elts (elt n)
  (if (> n 1)
      (list n elt)
      elt))
```

Рис. 3.6: Кодирование повторов: сжатие

Функция `compress` возвращает сжатое представление списка.

```
> (compress '(1 1 1 0 1 0 0 0 0 1))
((3 1) 0 1 (4 0) 1)
```

Если какой-либо элемент повторяется несколько раз, он заменяется на список, содержащий элемент и количество его повторений.

Основная работа выполняется рекурсивной функцией `compr`. Эта функция принимает три аргумента: `elt`, последний встреченный элемент; `n`, количество его повторений; `lst`, остаток списка, подлежащий дальнейшей компрессии. Когда список заканчивается, вызывается `n-elts`, возвращающая сжатое представление `n` элементов `elt`. Если первый элемент `lst` по-прежнему `elt`, увеличиваем `n` и идем дальше. В противном случае мы получаем сжатое представление предыдущей серии одинаковых элементов и присоединяем это к тому, что получим с помощью `compr` от остатка списка.

Чтобы реконструировать сжатый список, воспользуемся `uncompress` (рис. 3.7):

```
(defun uncompress (lst)
  (if (null lst)
      nil
      (let ((elt (car lst))
            (rest (uncompress (cdr lst))))
        (if (consp elt)
            (append (apply #'list-of elt)
                    rest)
            (cons elt rest)))))

(defun list-of (n elt)
  (if (zerop n)
      nil
      (cons elt (list-of (- n 1) elt))))
```

Рис. 3.7: Кодирование повторов: декодирование

```
> (uncompress '((3 1) 0 1 (4 0) 1))
(1 1 1 0 1 0 0 0 0 1)
```

Функция выполняется рекурсивно, копируя атомы и раскрывая списки с помощью `list-of`:

```
> (list-of 3 'ho)
(HO HO HO)
```

В действительности, нет необходимости использовать `list-of`. Встроенная `make-list` выполняет ту же работу, однако использует ключевой аргумент. Ключевых аргументов мы пока не касались.

Опытный разработчик определил бы функции `compress` и `uncompress` иным образом, сделав их более эффективными и приспособив к любым типам данных, нежели список атомов. В последующих нескольких главах мы узнаем, как можно исправить недостатки этой реализации.

3.6. Доступ

В Common Lisp имеются еще несколько функций доступа к частям списков. Для получения значения элемента с определенным индексом используется функция `nth`:

```
> (nth 0 '(a b c))
A
```

Для получения `n`-ного хвоста списка (`cdr`) есть `nthcdr`:

```
> (nthcdr 2 '(a b c))
(C)
```

Эти функции присваивают первому элементу списка нулевой индекс. Вообще говоря, в Common Lisp любая функция, обращающаяся к элементам списков, последовательностей, структур, начинает отсчет с нуля.

Эти две функции очень похожи, и вызов `nth` эквивалентен вызову `car` от `nthcdr`. Определим `nthcdr` без учета возможных ошибок:

```
(defun our-nthcdr (n lst)
  (if (zerop n)
      lst
      (our-nthcdr (- n 1) (cdr lst))))
```

Функция `zerop` всего лишь проверяет, равен ли нулю ее аргумент.

Функция `last` возвращает последнюю `cons`-ячейку списка:

```
> (last '(a b c))
(C)
```

Чтобы получить последний элемент, а не последнюю ячейку, воспользуйтесь `car` от `last`.

В Common Lisp для доступа к элементам с первого по десятый выделены специальные функции, которые получили названия от английских слов (от `first` до `tenth`). Обратите внимание, что отсчет начинается не с нуля, и `(second x)` эквивалентен `(nth 1 x)`.

Загрузка программ

Код в этом разделе впервые претендует на название отдельной программы. Если наша программа имеет достаточно большой размер, удобно сохранять ее текст в файл. Прочитать код из файла можно с помощью `load`. Если мы сохраним код с рис. 3.6 и 3.7 в файл под названием `"compress.lisp"`, то, набрав в toplevel

```
(load "compress.lisp")
```

получим такой же результат, как если бы набрали все выражения непосредственно в toplevel.

Учтите, что некоторые реализации могут использовать расширение `".lsp"`

Кроме того, в Common Lisp определены функции типа `caddr` (сокращенный вызов `car` от `cdr` от `cdr`). Определены функции вида `cxr`, где `x` – набор всех возможных сочетаний `a` и `d` длиной до четырех символов. На второй элемент можно сослаться с помощью `cadr`. В любом случае, использование таких функций не приветствуется, так как затрудняет понимание.

3.7. Отображающие функции

Часто может возникнуть необходимость применить какую-либо функцию к каждому элементу списка. Наиболее часто для этого используется функция `mapcar`, которая применяет заданную функцию поэлементно к одному или нескольким спискам и возвращающая список из значений функции.

```
> (mapcar #'(lambda (x) (+ x 10))
          '(1 2 3))
(11 12 13)
> (mapcar #'list
          '(a b c)
          '(1 2 3 4))
((A 1) (B 2) (C 3))
```

Из последнего примера видно, как `mapcar` обрабатывает случай со списками разной длины. Вычисление обрывается по окончании самого короткого списка.

Похожим образом действует `maplist`, однако применяет функцию последовательно не к `car`, а `cdr` списка.

```
> (maplist #'(lambda (x) x)
           ' (a b c))
((A B C) (B C) (C))
```

Позже, на стр. Ё004out будут описаны некоторые другие отображающие функции, например, `mapc`. На стр. Ё005out будет рассмотрена функция `mapcan`.

3.8. Деревья

`Cons`-ячейки можно рассматривать как деревья: `car` соответствует правому поддереву, а `cdr` – левому. Такое представление списка `(a (b c) d)` изображено на рис. 3.8 (Если повернуть его на 45° против часовой стрелки, то он будет напоминать рис. 3.3)

Рис. 3.8. Бинарное дерево

В Common Lisp есть встроенные функции для работы с деревьями. Например, `copy-tree` возвращает копию дерева. Определим аналогичную функцию самостоятельно:

```
(defun our-copy-tree (tr)
  (if (atom tr)
      tr
      (cons (our-copy-tree (car tr))
            (our-copy-tree (cdr tr)))))
```

Сравните ее с функцией `copy-list` (см. стр. Ё006out). В то время, как `copy-list` копирует только `cdr` списка, `copy-tree` копирует еще и `car`.

Бинарные деревья без внутренних узлов вряд ли окажутся полезными. Оперирование бинарными деревьями в Common Lisp – не самоцель, но они полезны для операций с подсписками. Например, вы имеете список

```
(and (integerp x) (zerop (mod x 2)))
```

И хотите заменить `x` на `y`. Заменить элементы в последовательности можно с помощью `substitute`:

```
> (substitute 'y 'x '(and (integerp x) (zerop (mod x2))))
(AND (INTEGERP X) (ZEROP (MOD X 2)))
```

Как видите, использование `substitute` не принесло результата. Здесь нам понадобится `subst`, работающая с деревьями:

```
> (subst 'y 'x '(and (integerp x) (zerop (mod x 2))))
(AND (INTEGERP Y) (ZEROP (MOD Y 2)))
```

Наше определение `subst` будет очень похоже на `copy-tree`:

```
(defun our-subst (new old tree)
  (if (eql tree old)
      new
      (if (atom tree)
          tree
          (cons (our-subst new old (car tree))
                (our-subst new old (cdr tree))))))
```

Любые функции, оперирующие с деревьями, будут выглядеть похожим образом, рекурсивно вызывая себя с `car` и `cdr`. Такая рекурсия называется *двойной*.

3.9. Чтобы понять рекурсию, нужно понять рекурсию

Чтобы понять, как работает рекурсия, часто предлагают изображать последовательность вызовов на бумаге. (Такая последовательность, например, изображена на стр. Ё007out) Иногда выполнение такого задания может ввести в заблуждение, так как программисту вовсе не обязательно четко представлять себе последовательность вызовов и возвращаемых результатов.

Если представлять рекурсию именно в таком виде, то ничего хорошего, кроме раздражения, от ее использования не получится. Преимущества рекурсии открываются при более абстрактном взгляде на нее.

Посмотрим, например, на рекурсивное определение длины списка:

```
(defun len (lst)
  (if (null lst)
      0
      (+ (len *cdr lst)) 1)))
```

Убедиться в корректности функции можно, проверив два случая[1]:

1. Она работает со списками нулевой длины, возвращая `0`
2. Если она работает со списками длины `n`, то будет справедлива также для списков длины `n+1`.

   Если оба случая верны, то функция ведет себя корректно на всех возможных списках.

   Первое утверждение совершенно очевидно: если `lst` - это `nil`, то функция тут же возвращает `0`. Теперь предположим, что она работает со списком длины n. Согласно определению, для списка длиной `n+1` она вернет число, на `1` большее длины `cdr` списка, т.е. `n+1`.

[1] Такой метод доказательства носит название *математической индукции*. – *Прим. перев.*

Это все, что нам понадобилось. Представлять всю последовательность вызовов вовсе не обязательно, так же, как необязательно искать парные скобки в определениях функций.

А как быть с более сложными функциями, например, с двойной рекурсией? Для функции `our-copy-tree` потребуется рассмотреть три случая: атомы, простые ячейки, ячейки, деревья, содержащие `n+1` ячеек.

Первый случай носит название базового (base case). Если рекурсивная функция ведет себя не так, как ожидалось, причина часто оказывается в некорректной проверке базового случая или же его полное отсутствие, как в примере с функцией `member`[1]:

```
(defun our-member (obj lst)              ; wrong
  (if (eql (car lst) obj)
      lst
      (our-member obj (cdr lst))))
```

В этом определении необходима проверка списка на пустоту, иначе в случае отсутствия искомого элемента в списке рекурсивный вызов будет выполняться бесконечно. В приложении А эта проблема рассматривается более детально.

Возможность оценивать корректность рекурсивной функции – ключ к ее пониманию. Однако, необходимо также научиться писать свои рекурсивные функции, а не только читать уже написанные. Этому посвящен раздел 6.9.

3.10. Множества

С помощью списков можно довольно эффективно управляться с небольшими наборами данных. Чтобы проверить, принадлежит ли элемент множеству, задаваемому списком, вы можете воспользоваться функцией `member`:

```
> (member 'b '(a b c))
(B C)
```

Если искомый элемент найден, `member` возвращает не `t`, а часть списка, начинающегося с найденного элемента. Конечно, непустой список логически соответствует истине, но такое поведение `member` позволяет получить больше информации. По умолчанию `member` сравнивает аргументы с помощью `eql`. Предикат сравнения возможно задать вручную с помощью *ключевого параметра*. Ё055in Ключевой параметры – довольно распространенный в Common Lisp способ передачи аргументов. Такие аргументы передаются не в соответствии с их положением в списке параметров, а с помощью особых тегов, называемых ключевыми словами. Ключевым словом считается любой символ, начинающийся с двоеточия.

Одним из ключевых аргументов, принимаемых `member`, является `:test`. Он позволяет использовать в качестве предиката сравнения вместо `eql` произвольную функцию, например, `equal`:

```
> (member '(a) '((a) (z)) :test #'equal)
((A) (Z))
```

Ключевые параметры не обязательны и следуют в лямбда-выражении после обязательных, причем их порядок не имеет значения.

Другой ключевой параметр `member` – `:key`. Он задает функцию, применяемую к каждому элементу перед сравнением.

```
> (member 'a '((a b) (c d)) :key #'car)
((A B) (C D))
```

В этом примере мы искали элемент, `car` которого равен `a`.

Если мы желаем использовать оба ключа, мы вольны сообщать их в произвольном порядке:

```
> (member 2 '((1) (2)) :key #'car :test #'equal)
```

[1] Слово wrong – *комментарий*. Комментариями в Лиспе считается все от символа ";" до конца текущей строки.

```
((2))
> (member 2 '((1) (2)) :test #'equal :key #'car)
((2))
```

С помощью `member-if` можем найти элемент, удовлетворяющий произвольному предикату, например, `oddp` (истинному, когда аргумент нечетен):

```
> (member-if #'oddp '(2 3 4))
(3 4)
```

Наша собственная функция `member-if` может быть записана следующим образом:

```
(defun our-member-if (fn lst)
  (and (consp lst)
          (if (funcall fn (car lst))
              lst
              (our-member-if fn (cdr lst)))))
```

Функция `adjoin` – своего рода условный `cons`. Она сцепляет заданные элемент и список, если этот элемент отсутствует в списке.

```
> (adjoin 'b '(a b c))
(A B C)
> (adjoin 'z '(a b c))
(Z A B C)
```

В общем случае `adjoin` имеет те же ключевые параметры, что и `member`.

Common Lisp предоставляет основные логические операции с множествами: объединение, пересечение, дополнение, для которых определены соответствующие функции: `union`, `intersection`, `set-difference`. Эти функции работают ровно с двумя списками и имеют те же ключевые параметры, что и `member`.

```
> (union '(a b c) '(c b s))
(A C B S)
> (intersection '(a b c) '(b b c))
(B C)
> (set-difference '(a b c d e) '(b e))
(A C D)
```

Так как результат не должен зависеть от порядка следования исходных списков, то порядок расположения элементов внутри списка-результата тоже может различаться. Например, `set-difference` может с тем же успехом вернуть `(d c a)`.

3.11. Последовательности

Списки также можно рассматривать как последовательности элементов, следующих друг за другом в фиксированном порядке. Кроме списков, последовательностями также считаются векторы. В этом разделе вы научитесь работать со списками как с последовательностями. Более детально последовательности будут рассмотрены в разделе 4.4.

Длина последовательности определяется с помощью `length`:

```
> (length '(a b c))
3
```

Ранее мы писали урезанную версию этой функции, работающую только со списками.

Скопировать часть последовательности можно с помощью `subseq`. Второй аргумент (обязательный) задает начало подпоследовательности, а третий (опционный) – индекс первого элемента, не подлежащего копированию.

```
> (subseq '(a b c d) 1 2)
(B)
> (subseq '(a b c d) 1)
(B C D)
```

Если третий аргумент пропущен, то подпоследовательность заканчивается вместе с исходной последовательностью.

Функция `reverse` возвращает последовательность, содержащую исходные элементы в обратном порядке:

```
> (reverse '(a b c))
(C B A)
```

С помощью `reverse` можно, например, искать *палиндромы*, то есть последовательности, читаемые одинаково в прямом и обратном порядке (например, `(a b b a)`). Две половины палиндрома с четным количеством аргументов будут зеркальными отражениями друг друга. Используя `length`, `subseq` и `reverse`, определим функцию Ё012in`mirror?`[1]:

```
(defun mirror? (s)
  (let ((len (length s)))
    (and (evenp len)
         (let ((mid (/ len 2)))
           (equal (subseq s 0 mid)
                  (reverse (subseq s mid)))))))
```

Эта функция определяет, является ли список таким палиндромом:

```
> (mirror? '(a b b a))
T
```

Для сортировки последовательностей в Common Lisp есть `sort`. Она принимает список, подлежащий сортировке, и функцию сравнения от двух аргументов.

```
> (sort '(0 2 1 3 8) #'>)
(8 3 2 1 0)
```

С `sort` следует быть осторожным, потому что она *деструктивна*. Из соображений производительности `sort` не создает новый список, а модифицирует исходный. Поэтому, если исходный список еще нужен, пользуйтесь `copy`°Ё073in.

Используя `sort` и `nth`, запишем функцию, которая возвращает `n`-ный элемент в порядке убывания:

```
(defun nthmost (n lst)
  (nth (- n 1)
       (sort (copy-list lst) #'>)))
```

Мы вынуждены вычитать единицу, потому что `nth` индексирует элементы, начиная с нуля, а наша `nthmost` – с единицы.

```
> (nthmost 2 '(0 2 1 3 8))
3
```

Наша реализация `nthmost` не самая эффективная, но мы пока не будем вдаваться в тонкости оптимизации.

Функции `every` и `some` получают предикат и одну или несколько последовательностей. Если задан один список, они проверяют, удовлетворяет ли каждый его элемент предикату:

```
> (every #'oddp '(1 3 5))
T
> (some #'evenp '(1 2 3))
T
```

Если задано несколько последовательностей, предикат должен принимать такое же количество аргументов:

```
> (every #'> '(1 3 5) '(0 2 4))
T
```

[1] Ричард Фейтман указал, что есть более простой способ проверки палиндрома, а именно: `(equal x (reverse x))`. – *Прим. перев.*

Если последовательности имеют разную длину, кратчайшая из них лимитирует тестирование.

3.12. Стек

Ячеечное представление списков позволяет легко реализовать идею стека. В Common Lisp есть два макроса для работы со списком как со стеком: `(push x y)` кладет объект `x` на вершину стека `y`, `(pop x)` снимает со стека верхний элемент. Оба этих макроса можно определить с помощью `setf`. Вызов

```
(push obj lst)
```

транслируется в

```
(setf lst (cons obj lst))
```

а вызов

```
(pop lst)
```

в

```
(let ((x (car lst))
      (setf lst (cdr lst))
      x)
```

Так, например,

```
> (setf x '(b))
(B)
> (push 'a x)
(A B)
> x
(A B)
> (setf y x)
(A B)
> (pop x)
A
> x
(B)
> y
(A B)
```

Структура ячеек после выполнения приведенных выражений показана на рис. 3.9.

Рис. 3.9. Эффект от `push` и `pop`

С помощью `push` можно также определить итеративный вариант `reverse` для списков:

```
(defun our-reverse (lst)
  (let ((acc nil))
    (dolist (elt lst)
      (push elt acc))
    acc))
```

Начиная с пустого списка, мы последовательно кладем на него, как на стек, элементы исходного. Последний элемент окажется на вершине стека, то есть в начале списка.

Макрос `pushnew` похож на `push`, однако использует `adjoin` вместо `cons`:

```
> (let ((x '(a b)))
     (pushnew 'c x)
     (pushnew 'a x)
     x)
(C A B)
```

Элемент `a` уже присутствует в списке, поэтому не добавляется.

3.13. Точечные пары

До сих пор мы работали с *нормальными списками* (*proper list*, *plist*). Нормальным списком считается либо `nil`, либо ячейка, `cdr` которой также нормальный список. Определим предикат для нормального списка[1]:

```
(defun proper-list? (x)
  (or (null x)
      (and (consp x)
           (proper-list? (cdr x)))))
```

Оказывается, `cons` может использоваться не только для создания нормальных списков, но и для структур, содержащих ровно два элемента.

```
> (setf pair (cons 'a 'b))
(A . B)
```

Это cons-ячейка не является нормальным списком, поэтому отображается через точку, разделяющую `car` и `cdr` этой ячейки. Такие ячейки называются *точечными парами*. Нормальные списки можно задавать и в виде набора точечных пар, но они будут отображаться в виде списков:

```
> '(a . (b . (c . nil)))
(A B C)
```

Обратите внимание, как соотносятся ячеечная и точечная нотации (рис. 3.10).

Рис. 3.10. Ячейка как точечная пара

Допустима также смешанная форма записи:

```
> (cons 'a (cons 'b (cons 'c 'd)))
(A B C . D)
```

Структура такого списка показана на рис. 3.11.

Рис. 3.11. Список точечных пар

Таким образом, список `(a b)` может быть записан аж четырьмя способами:

```
(a . (b . nil))
(a . (b))
(a b . nil)
(a b)
```

При этом Лисп отобразит их одинаково.

3.14. Ассоциативные списки

Еще одно удачное применение `cons`-ячеек – отображение. Список точечных пар называется *ассоциативным списком* (*assoc-list*, *alist*). С помощью него легко определить набор каких-либо правил и соответствий.

```
> (setf trans '((+ . "add") (- . "subtract")))
((+ . "add") (- . "subtract"))
```

Ассоциативный список – не самое эффективное решение данной задачи, особенно для больших наборов данных, но на начальном этапе создания программ могут быть очень полезными. В Common Lisp есть встроенная функция `assoc`, которая для заданного ключа получает соответствующую пару:

```
> (assoc '+ trans)
(+ . "add")
> (assoc '* trans)
NIL
```

Если `assoc` ничего не находит, он возвращает `nil`.

---

[1] Вообще говоря, это определение слегка некорректно, т.к. не всегда будет возвращать `nil` для любого объекта, не являющегося нормальным списком. Например, для циклического списка она будет работать бесконечно. Циклические списки обсуждаются в разделе 12.7.

Попробуем определить упрощенный вариант функции `assoc`:

```
(defun our-assoc (key alist)
  (and (consp alist)
       (let ((pair (car alist)))
         (if (eql key (car pair))
             pair
             (our-assoc key (cdr alist))))))
```

Как и `member`, нормальный `assoc` принимает несколько ключевых аргументов, включая `:test` и `:key`. Также Common Lisp имеет `assoc-if`, работающую по аналогии с `member-if`:

3.15. Поиск кратчайшего пути

На рис. 3.12 показана программа, вычисляющая кратчайший путь на графе (или сети). Функции `shortest-path` необходимо сообщить начальную и конечную точки, а так же саму сеть, и возвращает кратчайший путь между ними, если он вообще существует. Узлам соответствуют символы, а сама сеть представлена как ассоциативный список элементов вида `(узел . соседи)`.

Небольшая сеть, показанная на рис. 3.13, может быть записана так:

```
(defun shortest-path (start end net)
  (bfs end (list (list start)) net))

(defun bfs (end queue net)
  (if (null queue)
      nil
      (let ((path (car queue)))
        (let ((node (car path)))
          (if (eql node end)
              (reverse path)
              (bfs end
                   (append (cdr queue)
                           (new-paths path node net))
                   net))))))

(defun new-paths (path node net)
  (mapcar #'(lambda (n)
                    (cons n path))
              (cdr (assoc node net))))
```

Рис. 3.12. Поиск в ширину

Рис. 3.13. Простейшая сеть

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

Найти узлы, в которые можно попасть из узла `a`, поможет `assoc`:

```
> (cdr (assoc 'a min))
(B C)
```

Программа на рис. 3.12 реализует *поиск в ширину* (*breadth-first search*). Каждый слой сети исследуется поочередно друг за другом, пока не будет найден нужный элемент или достигнут ее конец. Последовательность исследуемых узлов представляется в виде очереди.

Приведенный код слегка усложняет эту идею, позволяя записывать пройденный путь. Таким образом, мы оперируем не с очередью узлов, а с очередью пройденных путей. Поиск выполняется функцией `bfs`. Первоначально в очереди только один элемент – путь к начальному узлу. Таким образом, `shortest-path` вызывает `bfs` с `(list (list start))` в качестве исходной очереди.

Первое, что должна сделать `bfs` – проверить, остались ли еще непроверенные узлы. Если очередь пуста, `bfs` возвращает `nil`, сигнализируя, что путь не был найден. Если же еще есть непроверенные

узлы, `bfs` берет первый из очереди. Если `car` этого узла содержит искомый элемент, значит, мы нашли путь до него, и просто возвращаем его, предварительно перевернув. В противном случае мы добавляем все его дочерние узлы в конец очереди, чтобы исследовать их, перейдя к следующему слою.

Так как `bfs` осуществляет поиск в ширину, то первый найденный путь будет одновременно самым коротким (ну, или одним из кратчайших, если есть и другие пути такой же длины).

```
> (shortest-path 'a 'd min)
(A C D)
```

А вот как выглядит соответствующая очередь во время каждого из вызовов `bfs`:

```
((A))
((B A) (C A))
((C A) (C B A))
((C B A) (D C A))
((D C A) (D C B A))
```

Второй элемент каждой очереди становится первым в следующей очереди, а первый элемент – `cdr` второго элемента следующей очереди.

Разумеется, приведенный код далек от полноценной эффективной реализации. Однако он убедительно показывает гибкость и удобство списков. В этой короткой программе мы используем списки тремя различными способами: список символов представляет путь, список путей представляет очередь[1], а ассоциативный список изображает саму сеть целиком.

3.16. Мусор

Операции со списками довольно неторопливы по ряду причин. Доступ к определенному элементу списка осуществляется не непосредственно по его номеру, а путем последовательного перебора всех предшествующих ему элементов, что может быть существенно медленнее, особенно для больших списков. Так как список является набором вложенных ячеек, то для доступа к элементу приходится преодолеть несколько переходов по указателям, в то время как для доступа к элементу массива достаточно просто увеличить позицию указателя на заданное число.

Автоматическое управление памятью – одна из наиболее ценных особенностей Лиспа. Лисп-система работает с участком памяти, называемым *куча* (*heap*). Система владеет информаций об использованной и неиспользованной памяти и позволяет использовать последнюю для размещения новых объектов (*consing*). Например, функция cons выделяет память под создаваемую ей ячейку.

Если память будет выделяться, но не освобождаться, то рано или поздно свободная память закончится. Поэтому необходим механизм поиска и освобождения участков кучи, которые более не содержат нужные данные. Память, которая становится ненужной, называется *мусором* и подлежит уборке. Этот процесс называется *сборкой мусора* (*garbge collection*, *GC*).

Как появляется мусор? Давайте помусорим немного:

```
> (setf lst (list 'a 'b 'c))
(A B C)
> (setf lst nil)
NIL
```

Первоначально мы вызвали `list`, которая трижды вызывала `cons`. После этого мы сделали `lst` пустым списком. Теперь ни одна из трех ячеек, созданных `cons`, не может быть использована[2]. Объекты, которые никаким образом не могут быть доступны для использования, и считаются мусором. Теперь система может безбоязненно использовать память, выделенную `cons`, по своему усмотрению.

Такой механизм управления памятью – огромный плюс для программиста, который не должен заботиться о явном выделении и освобождении памяти. Соответственно, исчезают баги, связанные с некорректным управлением памятью. Утечки памяти и висячие указатели просто невозможны в Лиспе.

Разумеется, за удобство надо платить. Автоматическое управление памятью работает во вред неаккуратному программисту. Затраты на работу с кучей и уборку мусора могут быть вполне существенными. Во-первых, если программа не выбрасывает ненужные объекты, память может подойти к концу. В таком случае выделение памяти и сборка мусора станут довольно затратными операциями. Конечно, прогресс не стоит на месте, автоматическое управление памятью становится более эффективным, хотя некоторые реализации Лиспа все еще не обзавелись хорошими дворниками.

Будучи неаккуратным, легко написать программу, выделяющую чрезмерно много памяти. Например, `remove` копирует ячейки, создавая новый список, чтобы не вызвать побочный эффект. Тем не менее, такого копирования можно избежать, используя *деструктивные* операции, которые модифицируют существующие данные, а не создают новые. Деструктивные функции подробно рассмотрены в разделе 12.4.

---

[1] В разделе 12.3 будет показано, как реализовать очереди более эффективным образом.

[2] Это не всегда так. В toplevel доступны глобальные переменные `*`, `**`, `***`, которым автоматически присваиваются последние три значения, вычисленные в toplevel. До тех пор, пока это значение связано с одной из вышеупомянутых переменных, мусором оно не считается.

Тем не менее, возможно написание программ, которые вообще не выделяют память в процессе выполнения. Часто программа создается так, что сначала она пишется в чисто функциональном стиле, а по мере развития копирующие функции заменяются на деструктивные, там, где это может принести бонус производительности. Здесь сложно давать конкретные советы, потому что некоторые современные реализации Лиспа управляют памятью настолько хорошо, что иногда выделение новой памяти может быть более эффективно, чем использование уже существующей.

Итоги главы

1. `Cons`-ячейка – это структура, состоящая из двух объектов. Списки создаются объединением ячеек.
2. Предикат `equal` менее строг, чем `eql`. Он сравнивает то, как отображаются объекты при печати.
3. Все объекты в Лиспе ведут себя как указатели, но вам никогда не придется управлять указателями явно.
4. Скопировать список можно с помощью `copy-list`, объединить два списка – с помощью `append`.
5. Кодирование повторов – простой алгоритм сжатия списков, легко реализуемый в Лиспе.
6. Common Lisp имеет богатый набор средств для доступа к элементам списков.
7. Отображающие функции применяют определенную функцию к каждому элементу списка.
8. Операции с вложенными списками сродни операциям с бинарными деревьями.
9. Чтобы оценить корректность рекурсивной функции, достаточно убедиться, что она соответствует нескольким требованиям.
10. Списки могут рассматриваться как множества. Для работы с множествами в Лиспе есть ряд встроенных функций.
11. Ключевые аргументы не обязательны и определяются не по положению, а по соответствующему тегу.
12. Список – подтип последовательности. Common Lisp имеет богатые возможности для работы с последовательностями.
13. `Cons`-ячейка, не являющаяся нормальным списком, называется точечной парой.
14. С помощью списков точечных пар можно представить набор правил отображения множеств. Такие списки называются ассоциативными.
15. Автоматическое управление памятью освобождает программиста от обязанности ручного выделения памяти, но генерация большого количества мусора может замедлить работу программы.

Упражнения

1. Нарисуйте следующие списки в виде ячеек:

```
(a) (a b (c d))
(b) (a (b (c (d))))
(c) (((a b) c) d)
(d) (a (b . c) . d)
```

2. Напишите свой вариант функции `union`, который сохраняет порядок следования элементов согласно исходным спискам:

```
> (new-union '(a b c) '(b a d))
(A B C D)
```

3. Определите функцию, определяющую количество повторений (с точки зрения `eql`) каждого элемента в заданном списке, и сортирующую их по убыванию встречаемости:

```
> (occurrences '(a b a d a c d c a))
((A . 4) (C . 2) (D . 2) (B . 1))
```

   Почему `(member '(a) '((A) (b)))` возвращает `nil`?

4. Функция `pos+` принимает список и возвращает новый, каждый элемент которого увеличен по сравнению с исходным на его положение в списке:

```
> (pos+ '(7 5 1 4))
(7 6 3 7)
```

5. Определите `pos+` функцию с помощью (a) рекурсии (b) итерации (c) `mapcar`.
6. После долгих лет раздумий государственная комиссия приняла постановление, согласно которому `cdr` указывает на первый элемент списка, а `car` – на его остаток. Определите следующие функции, удовлетворяющие этому постановлению:
   ```
   (a) cons
   (b) list[1]
   (c) length (для списков)
   (d) member (для списков, без ключевых параметров)
   ```
7. Измените программу на рис. 3.6 таким образом, чтобы она создавала меньшее количество ячеек. (Подсказка: используйте точечные пары)
8. Определите функцию, печатающую заданный список в точечной нотации:
   ```
   > (showdots '(a b c))
   (A . (B . (C . NIL)))
   NIL
   ```
9. Напишите программу, которая ищет наиболее длинный путь[2] в сети (раздел 3.15), которая может содержать циклы.

[1] Задачу 6b пока что не получится решить с помощью имеющихся у вас знаний. Вам потребуется остаточный параметр (`&rest`), который вводится на Ё008out. На эту оплошность указал Рикардо Феррейро де Оливьера. – *Прим. перев.*

[2] Разумеется, речь идет о пути максимальной длины, не содержащем повторений. Уточнение авторское. – *Прим. перев.*

4. Специализированные структуры данных

В предыдущей главе были рассмотрены списки – наиболее универсальные структуры для хранения данных. В этой главе будут рассмотрены другие способы хранения данных в Лиспе: массивы (а так же векторы и строки), структуры и хеш-таблицы. Они не настолько гибки, как списки, но более эффективны и занимают меньше места.

Common Lisp имеет еще один тип структур: *instance*. Об этом типе данных подробно рассказано в главе 11, описывающей CLOS.

4.1. Массивы

В Common Lisp массивы создаются с помощью функции `make-array`, сообщая ей первым аргументом список размерностей. Создадим массив 2x3:

```
> (setf arr (make-array '(2 3) :initial-element nil))
#<Simple-Array T (2 3) BFC4FE>
```

Многомерные массивы с Common Lisp могут иметь не более 7 размерностей, а в каждом измерении должно быть не более 1023 элемента.

Аргумент `:initial-element` опционален. Если он используется, то устанавливает изначальное значение каждого элемента массива. Поведение системы при попытке получить значение элемента массива, не инициализированного начальными значениями, не определено.

Чтобы получить элемент массива, пользуйтесь `aref`. Как и большинство других функций доступа в Common Lisp, `aref` начинает отсчет элементов с нуля:

```
> (aref arr 0 0)
NIL
```

Установить новое значение элемента массива можно с помощью `setf`:

```
> (setf (aref arr 0 0) 'b)
B
> (aref arr 0 0)
B
```

Аналогично спискам, массивы могут быть заданы вручную с помощью синтаксиса `#na`, где `n` – количество размерностей массива. Например, текущее состояние массива `arr` может быть задано так:

```
#2a((b nil nil) (nil nil nil))
```

Если глобальная переменная `*print-array*`[1] установлена в `t`, массивы буду печататься в таком же виде:

```
> (setf *print-array* t)
T
> arr
#2A((B NIL NIL) (NIL NIL NIL))
```

Для создания одномерного массива можно использовать желаемую длину вместо списка размерностей:

```
> (setf vec (make-array 4 :initial-element nil))
#(NIL NIL NIL NIL)
```

Одномерный массив также называют вектором. Создать и заполнить вектор можно одной функцией `vector`:

```
> (vector "a" 'b 3)
#("a" B 3)
```

[1] В вашей реализации `*print-array*` может изначально имеет значение `t`. – *Прим. перев.*

Так же как и одномерный массив, вектор может быть задан вручную с помощью синтаксиса `#()`.

Хотя доступ к элементам вектора может осуществить `aref`, специализированная функция `svref` более быстра.

```
> (svref vec 0)
```

NIL

Префикс `"sv"` расшифровывается как «simple vector»[1]. По умолчанию все векторы создаются как simple vector.

4.2. Пример: поиск дихотомией

В этом разделе в качестве примера показано, как написать функцию поиска элемента в сортированном векторе. Если нам известно, что элементы вектора расположены в определенном порядке, то поиск нужного элемента может быть выполнен быстрее, чем с помощью `find` (см стр. Ё009out). Вместо того, чтобы последовательно проверять элемент за элементом, мы сразу перемещаемся в середину списка. Если средний элемент соответствует искомому, то поиск закончен. В противном случае продолжаем поиск в правой или левой половине, в зависимости от того, больше или меньше искомого значения найденный средний элемент.

На рис. 4.1. приведена программа, реализующая такую возможность. Она состоит из двух функций: `binary-search`[2]Ё024in определяет границы поиска и передает управление функции `finder`, которая ищет соответствующий элемент между позициями `start` и `end` вектора `vec`.

```
(defun bin-search (obj vec)
  (let ((len (length vec)))
    (and (not (zerop len))
         (finder obj vec 0 (- len 1)))))

(defun finder (obj vec start end)
  (let ((range (- end start)))
    (if (zerop range)
        (if (eql obj (aref vec start))
            obj
            nil)
        (let ((mid (+ start (round (/ range 2)))))
          (let ((obj2 (aref vec mid)))
            (if (< obj obj2)
                (finder obj vec start (- mid 1))
                (if (> obj obj2)
                    (finder obj vec (+ mid 1) end)
                    obj)))))))
```

Рис. 4.1. Поиск в сортированном векторе

Когда область поиска сокращается до одного элемента, возвращается сам элемент в случае его соответствия искомому значению, либо `nil`. Если область поиска состоит из нескольких элементов, то определяется ее средний элемент – `obj2` (`round` возвращает ближайшее целое число), который сравнивается с искомым элементом `obj`. Если `obj` меньше `obj2`, поиск продолжается рекурсивно с левой половиной списка, в противном случае с правой половиной. Остается вариант `obj` = `obj2`, но это значит, что искомый элемент найден и мы просто возвращаем его.

Вставив следующую строчку в начале определения `finder`:

```
(format t "~A~%" (subseq vec start (+ end 1)))
```

Мы сможем наблюдать за процессом отсечения половин на каждом шаге.

---

[1] Вектор не может быть «простым», если он либо расширяемый (adjustable), либо предразмещенный (displaced), либо имеет указатель заполнения (fill-pointer). По умолчанию все массивы простые. Простой вектор – это одномерный простой массив, который может содержать элементы любого типа.

[2] Приведенная версия `bin-search` выпадает в осадок, если ей передать объект, меньший наименьшего из элементов вектора. Оплошность найдена Ричардом Грином. – *Прим. перев.*

```
> (bin-search 3 #(0 1 2 3 4 5 6 7 8 9))
#(0 1 2 3 4 5 6 7 8 9)
#(0 1 2 3)
#(3)
3
```

4.3. Строки и знаки

Строки – это векторы, состоящие из знаков. Строкой принято называть набор знаков, заключенный в двойные кавычки. Одиночный знак, пусть - `c`, задается так: `#\c`.

Каждый знак соответствует определенному целому числу, обычно (хотя и не обязательно) в соответствии с ASCII. В большинстве реализаций есть функция `char-code`, которая возвращает связанное со знаком число, и функция `code-char`, выполняющая обратное преобразование°Ё075in.

Для сравнения знаков используются следующие функции: `char<` (менее), `char<=` (равно или менее), `char=` (равно), `char>=` (равно или более), `char>` (более) и `char/=` (не равно). Они работают так, как будто работают с числами. Функции сравнения чисел рассматриваются на стр. Ё010out.

```
> (sort "elbow" #'char<)
"below"
```

Раз строки – это массивы, то к ним применимы все операции с массивами. Например, получить знак в конкретной позиции можно с помощью `aref`:

```
> (aref "abc" 1)
#\b
```

Однако, эта операция может быть выполнена быстрее с помощью специализированной функции `char`:

```
> (char "abc" 1)
#\b
```

Функция `char`, как и `aref`, может быть использована вместе с `setf` для замены элементов:

```
> (let ((str (copy-seq "Merlin")))
    (setf (char str 3) #\k)
    str)
"Merkin"
```

Договоренности о комментировании

В Common Lisp все, что следует за точкой с запятой, считается комментарием. Многие программисты используют последовательно несколько знаков комментирования, разделяя комментарии по уровням: четыре точки с запятой в заглавии файла, три – в описании функции или макроса, две – для пояснения последующей строки, одну – в той же строке, что и поясняемый код. Таким образом, с использованием общепринятых норм комментирования, начало кода на рис. 4.1. будет выглядеть так:

```
;;;; Инструменты для операций с сортированными списками.
;;; Находит элемент в сортированном векторе.
(defun bin-search )obj vec)
  (let ((len (length vec)))
    ;; если len действительно вектор, применяем у нему
finder
    (and (not (zerop len)) ; возвращает nil, есть len пуст
            (finder obj vec 0 (- len 1)))))
```

Многострочные комментарии удобно делать с помощью макроса чтения `#|...|#`. Все, что находится между `#|` и `|#`, игнорируется считывателем°Ё074in.

Чтобы сравнить две строки, вы можете воспользоваться известной вам `equal`, но есть также и специализированная `string-equal`, которая, к тому же, не учитывает регистр знаков (букв):

```
> (equal "fred" "fred")
T
> (equal "fred" "Fred")
NIL
> (string-equal "fred" "Fred")
T
```

Common Lisp предоставляет большой набор функций для сравнения и прочих манипуляций со строками. Они перечислены в приложении D, начиная со стр. Ё011out.

Есть несколько способов создания строк. Например, с помощью `format`. С аргументом `nil` `format` создаст строку, содержащую то, что было бы напечатано с аргументом `t`.

```
> (format nil "~A or ~A" "truth" "dare")
"truth or dare"
```

Этот способ не всегда удобен, так как рассматривает все возможные случаи. Чтобы просто соединить несколько строк, можно воспользоваться `concatenate`, которая требует указания типа получаемой последовательности:

```
> (concatenate 'string "not " "to worry")
"not to worry"
```

4.4. Последовательности

Тип *последовательность* (*sequence*) в Common Lisp включает списки и векторы (а значит, и строки). Многие функции из тех, которые мы ранее использовали для списков, могут быть использованы для любых последовательностей. Это, например, `remove`, `length`, `subseq`, `reverse`, `sort`, `every`, `some`. Таким образом, функция, определенная нами на стр. Ё012out, также может применяться к последовательностям:

```
> (mirror? "abba")
T
```

Мы уже знаем некоторые функции для доступа к элементам последовательностей: `nth` для списков, `aref` и `svref` для векторов, `char` для строк. Доступ к элементу последовательности любого типа может быть осуществлен с помощью `elt`:

```
> (elt '(a b c) 1)
B
```

Специализированные функции работают быстрее, и использовать `elt` рекомендуется только тогда, когда заранее не известен тип последовательности.

С помощью `elt` функция `mirror?` может быть оптимизирована для векторов:

```
(defun mirror? (s)
  (let ((len (length s)))
    (and (evenp len)
         (do ((forward 0 (+ forward 1))
              (back (- len 1) (- back 1)))
             ((or (> forward back)
                  (not (eql (elt s forward)
                            (elt s back))))
              (> forward back))))))
```

Эта версия по-прежнему будет работать со списками, однако, она более приспособлена для векторов. Регулярное использование последовательного доступа к элементам списка довольно затратно, а непосредственного доступа к нужному элементу они не предоставляют, в отличие от векторов, для которых стоимость доступа к любому элементу не зависит от его положения.

Многие функции, работающие с последовательностями, имеют несколько ключевых параметров: Ё049in

Параметр	Назначение	По умолчанию

:key	Функция, применяемая к каждому элементу	identity
:test	Функция, сравнивающая элементы	eql
:from-end	Если t, то функция стартует с конца	nil
:start	Номер элемента, с которого начинается выполнение	0
:end	Номер элемента, на котором заканчивается выполнение	nil

Одна из функций, использующих все эти параметры – position. Она возвращает положение определенного элемента в последовательности или nil в случае его отсутствия. Посмотрим на роль ключевых параметров на примере position:

```
> (position #\a "fantasia")
1
> (position #\a "fantasia" :start 3 :end 5)
4
```

Во втором случае поиск выполняется между четвертым и шестым элементом, :start ограничивает подпоследовательность слева, :end – справа, или же не ограничивает вовсе, если не задан.

Задавая параметр :from-end:

```
> (position #\a "fantasia" :from-end t)
7
```

мы получаем элемент, ближайший к концу последовательности. Тем не менее, в качестве ответа выдается реальное положение найденного элемента, а не расстояние от него до конца последовательности.

Параметр :key определяет функцию, применяемую к каждому элементу перед сравнением его с искомым:

```
> (position 'a '((c d) (a b)) :key #'car)
1
```

В этом примере мы поинтересовались, car которого элемента содержит букву a.

Параметр :test определяет, как будут сравниваться элементы. По умолчанию используется eql. Если вам необходимо сравнивать списки, придется воспользоваться equal:

```
> (position '(a b) '((a b) (c d )))
NIL
> (position '(a b) '((a b) (c d)) :test #'equal)
0
```

Аргумент :test может быть любой функцией двух элементов. Например, с помощью < мы можем найти первый элемент, больший заданного:

```
> (position 3 '(1 0 7 5) :test #'<)
2
```

С помощью subseq и position мы можем разделить последовательность на части. Например, функция

```
(defun second-word (str)
  (let ((p1 (+ (position #\  str) 1)))
    (subseq str p1 (position #\  str :start p1))))
```

возвращает второе слово в предложении:

```
> (second-word "Form follows function.")
"follows"
```

Поиск элементов, удовлетворяющих заданному предикату, осуществляется с помощью position-if. Он принимает функцию и последовательность, возвращая положение первого встреченного элемента, удовлетворяющего предикату:

```
> (position-if #'oddp '(2 3 4 5))
1
```

Эта функция принимает все вышеперечисленные ключевые параметры, за исключением `:test`.

Для последовательностей определены также функции, аналогичные применяемым к спискам `member` и `member-if`. Ё009inЭто `find` (использует все ключевые параметры) и `find-if` (все параметры, кроме `:test`):

```
> (find #\a "cat")
#\a
> (find-if #'characterp "ham")
#\h
```

В отличие от `member` и `member-if`, они возвращают сам найденный элемент.

Вместо `find-if` иногда использовать `find` с ключом `:key`. Например, выражение:

```
(find-if #'(lambda (x)
             (eql (car x) 'complete))
         lst)
```

будет выглядеть понятнее:

```
(find 'complete lst :key #'car)
```

Функция `remove` (см. стр. Ё013out) и `remove-if` работают с последовательностями любого типа, а разница между ними точно такая же, как между `find` и `find-if`. Связанная с ними функция `remove-duplicates` удаляет все повторяющиеся элементы, кроме последнего:

```
> (remove-duplicates "abracadabra")
"cdbra"
```

Эта функция использует все ключевые параметры, рассмотренные выше.

Функция `reduce` конденсирует последовательность в одно значение. Ей необходимо сообщить функцию двух аргументов и последовательность. Заданная функция будет первоначально применена к первым двум элементам последовательности, после чего будет последовательно применяться к полученному результату и следующему элементу. Последнее полученное значение будет возвращено как значение вызова `reduce`. Таким образом, вызов:

```
(reduce #'fn '(a b c d))
```

будет эквивалентен

```
(fn (fn (fn 'a 'b) 'c) 'd)
```

Хорошее применение `reduce` – расширение набора аргументов для функций, которые принимают только два аргумента. Например, так мы получим пересечение нескольких списков:

```
> (reduce #'intersection '((b r a d 's) (b a d) (c a t)))
(A)
```

4.5. Пример: разбор дат

В качестве примера операций с последовательностями в этом разделе приводится программа для разбора дат. Мы напишем программу, которая превращает строку типа `"16 Aug 1980"` в целые числа, соответствующие дню, месяцу, году.

Рис. 4.2. содержит некоторые функции, которые потребуются нам в дальнейшем. Первая, `tokens`Ё023in, выделяет знаки из строки. Функция `tokens` принимает строку и функцию, возвращая список подстрок, знаки которых удовлетворяют заданной функции. Приведем пример. Пусть используется функция `alpha-char-p` – предикат, справедливый для буквенных знаков. Тогда получим:

```
(defun tokens (str test start)
```

```
    (let ((p1 (position-if test str :start start)))
      (if p1
          (let ((p2 (position-if #'(lambda (c)
                                     (not (funcall test c)))
                                 str :start p1)))
            (cons (subseq str p1 p2)
                  (if p2
                      (tokens str test p2)
                      nil)))
          nil)))

(defun constituent (c)
  (and (graphic-char-p c)
       (not (char= c #\  ))))
```

Рис. 4.2. Распознавание символов

```
> (tokens "ab12 3cde.f" #'alpha-char-p 0)
("ab" "cde" "f")
```

Все остальные символы, не удовлетворяющие данному предикату, рассматриваются как разделители подпоследовательностей.

Функция `constituent` будет использоваться в качестве предиката для `tokens`. В Common Lisp к *печатным знакам* (*graphic characters*) относятся все знаки, которые видны при печати, а также пробел. Вызов `tokens` с функцией `constituent` будет выделять подстроки, состоящие из печатных символов:

```
> (tokens "ab12 3cde.f
                gh" #'constituent 0)
("ab12" "3cde.f" "gh")
```

На рис. 4.3 показаны функции, выполняющие разбор дат. Функция `parse-date` возвращает список целых чисел, соответствующих компонентам строки, представляющей дату:

```
(defun parse-date (str)
  (let ((toks (tokens str #'constituent 0)))
    (list (parse-integer (first toks))
          (parse-month (second toks))
          (parse-integer (third toks)))))

(defconstant month-names
  #("jan" "feb" "mar" "apr" "may" "jun"
    "Jul" "aug" "sep" "oct" "nov" "dec"))

(defun parse-month (str)
  (let ((p (position str month-names
                     :test #'string-equal)))
    (if p
        (+ p 1)
        nil)))
```

Рис. 4.3. Функции для разбора дат

```
> (parse-date "16 Aug 1980")
(16 8 1980)
```

Эта функция делит строку на части и применяет `parse-month` и `parse-integer` к полученным частям. Функция `parse-month` не чувствительна к регистру, так как сравнивает строки с помощью `string-equal`. Для преобразования строки, содержащей число, в само число, мы используем встроенную функцию `parse-integer`.

Однако, если бы такой функции в Common Lisp изначально не было, нам пришлось бы определить ее самостоятельно:Ё015in

```
(defun read-integer (str)
  (if (every #'digit-char-p str)
      (let ((accum 0))
```

```
    (dotimes (pos (length str))
      (setf accum (+ (* accum 10)
                     (digit-char-p (char str pos)))))
    accum)
  nil))
```

Определенная нами функция `read-integer` показывает, как в Common Lisp преобразовать набор символов в число. Она использует особенность функции `digit-char-p`, является ли аргумент цифрой, но и возвращает саму цифру.

4.6. Структуры

Структура (тип данных) – еще одна разновидность векторов. Где их использование может быть выгоднее по сравнению с обычными векторами? Представим, например, что нам нужна программа, отслеживающся положение набора геометрически правильных тел. Каждое такое тело можно представить в виде трех параметров: высота, ширина, глубина. Программа была бы куда более понятна, если использовать вместо простых `svref`'ов специальные функции:

```
(defun block-height (b) (svref b 0))
```

и так далее. Однако было бы неплохо вместо вектора, содержащего все параметры тела, использовать соответствующую структуру, для которой все подобные функции уже определены.

Определить структуру можно с помощью `defstruct`. В простейшем случае достаточно сообщить имена структуры и ее полей:

```
(defstruct point
  x
  y)
```

Мы определили структуру `point`, имеющею два поля, `x` и `y`. Помимо этого, для нее уже определены некоторые функции: `make-point`, `point-p`, `copy-point`, `point-x`, `point-y`. Ранее, в разделе 2.3, мы упоминали о способности Лисп-программ генерировать код. Это один из примеров: `defstruct` самостоятельно определяет все необходимые функции. Научившись работать с макросами, вы сможете делать похожие вещи самостоятельно. (Например, вы сможете написать свою версию `defstruct` при необходимости.)

Каждый вызов `make-point` возвращает вновь созданный экземпляр структуры `point`. Значения полей могут быть заданы изначально с помощью соответствующих ключевых параметров:

```
> (setf p (make-point :x 0 :y 0))
#S(POINT X 0 Y 0)
```

Функции доступа к полям структуры приспособлены не только к извлечению текущих значений, но и к установлению новых с помощью `setf`:

```
> (point-x p)
0
> (setf (point-y p) 2)
2
> p
#S(POINT X 0 Y 2)
```

Для каждой структуры определена также свой тип. Каждый экземпляр `point` принадлежит типу `point`, затем `structure`, затем `atom` и `t`. Таким образом, использование `point-p` равносильно проверке типа:

```
> (point-p p)
 T
> (typep p 'point)
T
```

Функция `typep` проверяет объект на принадлежность к заданному типу.

Макрос `defstruct` позволяет задавать изначальные значения полей. Для этого необходимо вместо поля сообщить список, состоящий из имени поля и исходного значения.

```
(defstruct polemic
  (type (progn
          (format t "What kind of polemic was it? ")
          (read)))
  (effect nil))
```

Вызов `make-polemic` без дополнительных аргументов установит умолчальные значения полей:

```
> (make-polemic)
What kind of polemic was it? scathing
#S(POLEMIC TYPE SCATHING EFFECT NIL)
```

Кроме того, есть возможность управлять именами создаваемых функций. Вот более продуманный вариант структуры `point`:

```
(defstruct (point (:conc-name p)
                  (:print-function print-point))
  (x 0)
  (y 0))

(defun print-point (p stream depth)
  (format stream "#<~A, ~A>" (px p) (py p)))
```

Имена функций доступа к полям структуры получаются присоединением префикса к имени соответствующих полей. По умолчанию этот префикс выглядит как `point-`, то есть, имя структуры и дефис. Параметр `:conc-name` позволяет задать другой префикс. В новом определении это просто `p`. Во многих случаев использование своего префикса делает код более читаемым, поэтому вам стоит задуматься об использовании более короткого префикса, если вам предстоит часто пользоваться функциями доступа к полям.

Параметр `:print-function` – это *имя* функции, которая будет вызываться для печати объекта в toplevel. Такая функция должна принимать три аргумента: печатаемый объект; адрес, куда он будет напечатан; третий аргумент обычно не требуется и может быть проигнорирован[1]. С потоками мы познакомимся ближе в разделе 7.1. Сейчас достаточно сказать, то второй аргумент, поток, может быть сообщен функции `format`.

Функция `print-point` будет отображать структуру следующим образом:

```
> (make-point)
#<0,0>
```

4.7. Пример: двоичные деревья поиска

В Common Lisp имеется встроенная функция `sort`, и вам, скорее всего, не придется самостоятельно писать сортирующие утилиты. В этом разделе показано, как решить сходную задачу, для которой нет встроенной функции: управление сортированным набором объектов. Мы рассмотрим метод хранения объектов в *двоичном дереве поиска* (*BST*). Сбалансированное BST позволяет искать, добавлять или удалять элементы за время, пропорциональное log (n), где n – размер набора объектов.

BST – это бинарное дерево, в котором для каждого элемента и некоторой функции упорядочения (пусть это `<`) соблюдается правило: левый дочерний элемент `<` элемента-родителя, и сам элемент `>` правого дочернего элемента. На рис. 4.4. показан пример BST, упорядоченного с помощью функции `<`.

Рис. 4.4. Двоичное дерево поиска

Рис. 4.5. содержит утилиты для вставки и поиска объектов в BST. В качестве основной структуры данных используются узлы. Каждый узел имеет три поля: в

[1] В ANSI Common Lisp вы также можете использовать с `:print-object` функцию двух параметров. Кроме того, существует макрос `print-unreadable-object`, который может быть использован для отображения объекта в виде `#<...>`.

одном хранится сам объект, в двух других – левый и правый потомки. Можно рассматривать узел как cons-ячейку с одним car и двумя cdr.

```
(defstruct (node (:print-function
                  (lambda (n s d)
                    (format s "#<~A>" (node-elt n)))))
  elt (l nil) (r nil))

(defun bst-insert (obj bst <)
  (if (null bst)
      (make-node :elt obj)
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall < obj elt)
                (make-node
                  :elt elt
                  :l   (bst-insert obj (node-l bst) <)
                  :r   (node-r bst))
                (make-node
                  :elt elt
                  :r   (bst-insert obj (node-r bst) <)
                  :l   (node-l bst)))))))

(defun bst-find (obj bst <)
  (if (null bst)
      nil
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall < obj elt)
                (bst-find obj (node-l bst) <)
                (bst-find obj (node-r bst) <))))))

(defun bst-min (bst)
  (and bst
       (or (bst-min (node-l bst)) bst)))

(defun bst-max (bst)
  (and bst
       (or (bst-max (node-r bst)) bst)))
```

Рис. 4.5. Двоичные деревья поиска: поиск и вставка

BST может быть либо nil, либо узлом, поддеревья которого также являются BST. Продолжим дальнейшую аналогию со списками. Как список может быть создан последовательностью вызовов cons, так и бинарное дерево может быть построено с помощью bst-insertЁ039in. Этой функции необходимо сообщить объект, дерево и функцию упорядочения.

```
> (setf nums nil)
 NIL
> (dolist (x '(5 8 4 2 1 9 6 7 3))
    (setf nums (bst-insert x nums #'<)))
NIL
```

Теперь дерево nums соответствует рис. 4.4.

Функция bst-find, которая ищет объекты в дереве, требует тех же аргументов, что и bst-insert. Аналогия со списками еще более прояснится, если мы сравним определение bst-find и our-member (см. стр. Ё014out).

Как и member, bst-find возвращает не сам элемент, а его поддерево.

```
> (bst-find 12 nums #'<)
NIL
> (bst-find 4 nums #'<)
#<4>
```

Такое представление позволяет нам различать случай, когда искомый элемент не найден (`nil`) и успешное нахождение элемента `nil`.

Нахождение наибольшего и наименьшего элементов BST так же не составляет особого труда. Чтобы найти минимальный элемент, мы идем по дереву, всегда выбирая левую ветвь (`bst-min`). Аналогично, следуя правым поддеревьям, мы получим наибольший элемент (`bst-max`)

```
> (bst-min nums)
#<1>
> (bst-max nums)
#<9>
```

Удаление элемента из бинарного дерева выполняется так же быстро, но соответствующий код выглядит сложнее. Он показан на рис. 4.6. Функция `bst-remove`Ё057in, которой необходимо сообщить объект , дерево и функцию упорядочения, возвращает это же дерево без заданного элемента. Как и `remove`, `bst-remove`[1] не модифицирует исходное дерево:

```
(defun bst-remove (obj bst <)
  (if (null bst)
      nil
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            (percolate bst)
            (if (funcall < obj elt)
                (make-node
                  :elt elt
                  :l (bst-remove obj (node-l bst) <)
                  :r (node-r bst))
                (make-node
                  :elt elt
                  :r (bst-remove obj (node-r bst) <)
                  :l (node-l bst)))))))

(defun percolate (bst)
  (let ((l (node-l bst)) (r (node-r bst)))
    (cond ((null l) r)
          ((null r) l)
          (t (if (zerop (random 2))
                 (make-node :elt (node-elt (bst-max l))
                            :r r
                            :l (bst-remove-max l))
                 (make-node :elt (node-elt (bst-min r))
                            :r (bst-remove-min r)
                            :l l))))))

(defun bst-remove-min (bst)
  (if (null (node-l bst))
      (node-r bst)
      (make-node :elt (node-elt bst)
                 :l   (bst-remove-min (node-l bst))
                 :r   (node-r bst))))

(defun bst-remove-max (bst)
  (if (null (node-r bst))
```

[1] Ё019inОригинальная версия `bst-remove` содержит баг.
Крис Стовер сообщает: «Задания (left child) < node < (right child) для каждого узла недостаточно. Необходимо более строгое ограничение: max(left subtree) < node < min(right subtree). Без него функция `bst-traverse`, приведенная на рис. 4.8 не обязательно перечислит элементы в порядке возрастания. (Пример: дерево с основанием 1 имеет только правого потомка 2, а он имеет только правого потомка 0.) К счастью, функция `bst-insert`, представленная на рис. 4.5, корректна. С другой стороны, корректный порядок BST после применения `bst-remove` не гарантируется. Удаляемый внутренний узел необходимо заменять либо максимальным узлом левого поддерева, либо минимальным узлом правого поддерева, а приведенная функция этого не делает».
Данный баг уже исправлен в представленном коде. – *Прим. перев.*

```
        (node-l bst)
        (make-node :elt (node-elt bst)
                   :l   (node-l bst)
                   :r   (bst-remove-max (node-r bst)))))
```

Рис. 4.6. Двоичные деревья поиска: удаление.

```
> (setf nums (bst-remove 2 nums #'<))
#<5>
> (bst-find 2 nums #'<)
NIL
```

Теперь дерево `nums` соответствует рис. 4.7. Другой возможный случай – подстановка элемента `1` на место `2`.

Рис. 4.7. Двоичное дерево поиска после удаления одного из элементов.

Удаление – более затратная процедура, так как на под удаляемым объектом появляется незанятое место, куда можно подставить одно из поддеревьев этого объекта. Этим занимается функция `percolate`. Она замещает элемент дерева одним из его поддеревьев, затем замещает это поддерево одним из *его* поддеревьев, и так далее.

Чтобы сбалансировать дерево, `percolate` случайным образом выбирает одно из двух поддеревьев. Выражение `(random 2)` вернет либо `0`, либо `1`, и `(zerop (random 2))` будет истинно в половине случаевЁ035in.

Теперь, когда мы преобразовали набор объектов в бинарное дерево, неупорядоченный обход его элементов приведет к расстановке их по возрастанию. Для этой цели определена функция `bst-traverse`, которая показана на рис. 4.8:

```
(defun bst-traverse (fn bst)
  (when bst
    (bst-traverse fn (node-l bst))
    (funcall fn (node-elt bst))
```

```
                (bst-traverse fn (node-r bst))))
```

Рис. 4.8. Двоичные деревья поиска: обход.

```
> (bst-traverse #'princ nums)
13456789
NIL
```

Функция `princ` всего лишь отображает отдельный объект.

Код, представленный в этой главе, служит основой для реализации двоичных деревьев поиска. Вероятно, вы захотите как-то усовершенствовать его в соответствии с вашими потребностями. Например, каждый узел в текущей реализации имеет лишь одно поле `elt`, в то время как может оказаться полезным введение двух полей – ключ и значение. Данная версия хотя и не поддерживает такую возможность, но позволяет ее легко реализовать.

Двоичные деревья поиска могут использоваться не только для управления сортированным набором объектов. Их применение оптимально, когда вставки и удаления узлов имеют равномерное распределение. Это значит, что для работы, например, с очередями, BST *не* лучший выбор. Хотя вставки вполне могут быть распределены равномерно, удаления будут всегда осуществляться с конца. Это будет приводить к разбалансировке дерева, и вместо ожидаемой оценки O(log n) мы получим O(n). Кроме того, для моделирования очередей удобнее использовать обычный список просто потому, что BST будет вести себя в конечном счете так же, как список.°Ё076in

4.8. Хеш-таблицы

В главе 3 было показано, что списки могут моделировать множества и отображения. Для достаточно больших массивов данных (начиная уже с 10 элементов) использование хеш-таблиц даст большую производительность. Хеш-таблицу можно создать с помощью `make-hash-table`, которая не требует обязательных аргументов:

```
> (setf ht (make-hash-table))
#<Hash-Table BF0A96>
```

Хеш-таблицы, как и функции, при печати отображаются в синтаксисе `#<...>`.

Хеш-таблица, аналогично ассоциативному списку, - способ ассоциирования пар объектов. Чтобы получить значение, связанное с заданным ключом, достаточно вызвать `gethash` с этим ключом и таблицей. По умолчанию `gethash` возвращает `nil`, если не находит искомого элемента.

```
> (gethash 'color ht)
NIL
NIL
```

Здесь мы впервые сталкиваемся с важной особенностью Common Lisp: выражение может возвращать несколько значений. Функция `gethash` возвращает два. Первое – значение, ассоциированное с ключом. Второе значение, если оно `nil`, как в нашем примере, означает, что искомый элемент не был найден. Почему мы не можем судить об этом из первого `nil`? Дело в том, что элементом, связанным с ключом `'color`, может оказаться `nil`, и `gethash` вернет его, а также `t` в качестве второго значения.

Большинство реализаций выводит последовательно все возвращаемые значения, но если результат многозначной функции используется другой функцией, то ей передается лишь первое значение. В разделе 5.5 объясняется, как получать и использовать сразу несколько значений.

Сопоставить новое значение какому-либо ключу поможет `setf`:

```
> (setf (gethash 'color ht) 'red)
RED
```

Теперь gethash вернет вновь установленное значение:

```
> (gethash 'color ht)
RED
T
```

Второе значение подтверждает, что gethash вернул реально имеющийся в таблице объект, а не умолчальное значение.

Объекты, хранимые в хеш-таблицах, могут иметь любой тип. Например, если мы захотим сопоставить каждой функции ее краткое описание, мы создадим таблицу, в которой ключами функции, а значениями – строки:

```
> (setf bugs (make-hash-table))
#<Hash-Table BF4C36>
> (push "Doesn't take keyword arguments. "
    (gethash #'our-member bugs))
("Doesn't take keyword arguments. ")
```

Так как текущее значение пока не установлено, то есть nil, о вызов push эквивалентен setf, и мы просто кладем нашу строку на пустой список. (Определение функции our-member на стр. Ё013out.)

Хеш-таблицы могут служить заменой спискам на операциях со множествами. Использование хеш-таблиц дает существенный прирост производительности по сравнению со списками на операциях с множествами больших размеров. Чтобы добавить элемент в множество, представленное в виде хеш-таблицы, используйте setf вместе с gethash:

```
> (setf fruit (make-hash-table))
#<Hash-Table BFDE76>
> (setf (gethash 'apricot fruit) t)
T
```

Проверка на принадлежность элемента множеству выполняется с помощью gethash:

```
> (gethash 'apricot fruit)
T
T
```

По умолчанию gethash возвращает nil, и вновь созданная хеш-таблица представляет пустое множество.

Чтобы удалить элемент, вы можете воспользоваться remhash:

```
> (remhash 'apricot fruit)
T
```

Возвращая t, remhash сигнализирует, что искомый элемент был найден и успешно удален.

Для итерации по хеш-таблице существует maphash, которой необходимо сообщить функцию двух аргументов и саму таблицу. Переданная функция будет применяться к каждой имеющейся в таблице паре ключ-значение, при этом порядок предъявления этих пар не регламентирован.

```
> (setf (gethash 'shape ht) 'spherical
        (gethash 'size ht) 'giant)
GIANT
> (maphash #'(lambda (k v)
               (format t "~A = ~A~%" k v))
           ht)
SHAPE = SPHERICAL
SIZE = GIANT
COLOR = RED
NIL
```

Функция maphash всегда возвращает nil, однако мы можете сохранить необходимые вам данные, изменив соответствующим образом функцию, применяемую к парам ключ-значение.

Хеш-таблицы могут накапливать любое количество вхождений, так как способны расширяться в процессе работы, когда места для хранения элементов перестанет хватать. Задать исходную емкость таблицы можно с помощью ключа `:size` функции `make-hash-table`. Используя этот параметр, вам следует помнить о двух вещах: большой размер таблицы позволит избежать частых ее расширений (это довольно затратная процедура), однако создание таблицы большого размера для малого набора данных приведет к необоснованным затратам памяти. Важный момент: параметр `:size` определяет не количество хранимых объектов, а количество пар ключ-значение. Выражение:

```
(make-hash-table :size 5)
```

вернет хеш-таблицу, которая сможет вместить пять вхождений до того, как ей придется расширяться.

Как и любая структура, подразумевающая возможность поиска элементов, хеш-таблица может использовать различные предикаты проверки эквивалентности. По умолчанию используется `eql`, но вы можете сообщить другую функцию, например, `eq`, `equal` или `equalp` с помощью ключа `:test`:

```
> (setf writers (make-hash-table :test #'equal)
#<Hash-Table C005E6>
> (setf (gethash '(ralph waldo emerson) writers) t)
T
```

Это одна из проблем, с которой нам приходится мириться ради использования эффективных хеш-таблиц. Работая со списками, мы бы воспользовались функцией `member`, которой можно каждый раз сообщать различные проверочные предикаты. При работе с хеш-таблицами мы должны определиться с используемым предикатом заранее, в момент ее создания.

С необходимостью жертвовать одним ради другого в Лисп-разработке (да и в жизни в целом) вам придется столкнуться не раз. Это часть философии Лиспа: первоначально вы миритесь с низкой производительностью ради удобства разработки, а по мере развития программы можете пожертвовать ее гибкостью ради скорости выполнения.

Итоги главы

1. Массивы в Common Lisp могут иметь до семи размерностей. Одномерные массивы называются векторами.
2. Строки – это векторы знаков. Знаки могут считаться полноценными объектами.
3. Последовательности включают списки и векторы. Большинство функций для работы с последовательностями имеют ключевые параметры.
4. Разбор последовательностей не составляет труда в Common Lisp благодаря наличию множества полезных функций для работы с ними.
5. Вызов `defstruct` создает структуру с именованными полями. Это хороший промер кодогенерации.
6. Двоичные деревья поиска удобны для хранения сортированного набора объектов.
7. Хеш-таблицы представляют эффективный способ управления множествами и отображениями.

Упражнения

1. Определите функцию, поворачивающую квадратный массив (массив с размерностями `(n n)`) на 90 градусов по часовой стрелке.:
   ```
   (quarter-turn #2A((a b) (c d)))
   #2A((C A) (D B))
   ```
   Вам потребуется `array-dimensions` (см. стр. Ё029out).
2. Разберитесь с описанием `reduce` (см. стр. Ё052out), затем определите с ее помощью:
   ```
   (a) copy-list
   (b) reverse (для списков)
   ```

3. Создайте структуру для дерева, каждый узел которого помимо некоторых данных имеет три потомка. Определите функцию:

   (a) копирующую такое дерево (каждый узел скопированного дерево не должен быть эквивалентным исходному с точки зрения `eql`)

   (b) принимающую объект и такое дерево и возвращающую истину, если этот объект встречается (с точки зрения `eql`) в поле данных хотя бы одного узла дерева.

4. Определите функцию, которая строит из BST-дерева список его объектов, отсортированный от большего к меньшему.

5. Определите `bst-adjoin`[1]. Функция работает аналогично `bst-insert`, однако встраивает новый объект лишь в том случае, когда он отсутствует в имеющемся дереве.

6. Содержимое любой хеш-таблицы может быть представлено в виде ассоциативного списка с элементами `(k . v)` для каждой пары ключ-значение. Определите функцию, строящую:

   (a) хеш-таблицу по ассоциативному списку

   (b) ассоциативный список по хеш-таблице.

[1] В действительности, определение `bst-adjoin` будет эквивалентно уже имеющемуся `bst-insert`. На это указал Ричард Грин. – *Прим. перев.*

5. Управление

В разделе 2.2 порядок вычисления был упомянут впервые, теперь же он должен быть вам хорошо знаком. В данной главе будут рассмотрены операторы, которые не подчиняются этому правилу, позволяя вам самостоятельно управлять ходом вычисления. Если обычные функции считать листьями Лисп-программы, то с помощью таких операторов можно создавать ее ветви.

5.1. Блоки

Common Lisp имеет три основных оператора для создания блоков кода: `progn`, `block` и `tagbody`. С первым мы уже знакомы. Выражения, составляющие тело оператора `progn`, вычисляются последовательно, при этом возвращается значение последнего:°Ё077in

```
> (progn
    (format t "a")
    (format t "b")
    (+ 11 12))
ab
23
```

Так как возвращается значение лишь последнего выражения, то использование `progn` целесообразно лишь при наличии побочных эффектов.

Оператор `block` похож на `progn`, однако имеет имя и может принудительно прерывать вычисления. Первый аргумент – символ, определяющий имя блока. Находясь внутри блока, вы можете в любой момент прервать вычисление, сообщив имя соответствующего блока функции `return-from`:

```
> (block head
    (format t "Here we go.")
    (return-from head 'idea)
    (format t "We'll never see this. "))
Here we go.
IDEA
```

Вызов `return-from` позволяет выйти из блока выражений при необходимости. Ее второй аргумент возвращается в качестве значения при выходе. Все выражения, находящиеся после `return-from` в соответствующем блоке, не вычисляются.

Кроме того, существует специальный макрос `return`, выполняющий выход из блока с именем `nil`:

```
> (block nil
    (return 27))
27
```

Многие операторы в Common Lisp неявным образом используют `block` с именем `nil`, например, итерационный оператор `do`:

```
> (dolist (x '(a b c d e))
    (format t "~A " x)
    (if (eql x 'c)
        (return 'done)))
A B C
DONE
```

Тело функции, создаваемой `defun`, является блоком с тем же именем, что и сама функция:

```
(defun foo ()
  (return-from foo 27))
```

Вне явного или неявного использования блока ни `return`, ни `return-from` не работают.

С помощью return-from позволяют написать более приемлемый вариант read-integer:

```
(defun read-integer (str)
  (let ((accum 0))
    (dotimes (pos (length str))
      (let ((i (digit-char-p (char str pos))))
        (if i
            (setf accum (+ (* accum 10) i))
            (return-from read-integer nil))))
    accum))
```

Предыдущий вариант read-integer (см. стр. Ё015out) выполнял проверку символов до построения целого числа. Теперь же два шага собраны воедино, так как использование return-from позволяет прервать выполнение, когда мы встретим нечисловой символ.

Нами был упомянут еще один конструктор блоков – tagbody, который допускает внутри себя использование оператора go. Атомы, встречающиеся внутри блока, расцениваются как метки, по которым может выполняться переход. Ниже приведен довольно корявый код, печатающий числа от 1 до 10:

```
> (tagbody
    (setf x 0)
   top
    (setf x (+ x 1))
    (format t "~A " x)
    (if (< x 10) (go top)))
1 2 3 4 5 6 7 8 9 10
NIL
```

Оператор tagbody один из тех, которые применяются для построения других операторов, но, как правило, не годятся для ручного использования. Большинство итеративных операторов построены поверх tagbody, поэтому иногда (не всегда) возможно использовать внутри них метки и переход по ним с помощью go.

Как выбрать наиболее подходящий конструктор блоков? Практически всегда вам подойдет progn. Если вы желаете иметь возможность экстренного выхода, используйте block. Практически никто не пользуется tagbody явным образом.

5.2. Контекст

Еще одним оператором, позволяющим группировать выражения, мы уже умеем пользоваться. Это let. Помимо набора вычисляемых выражений, он позволяет устанавливать новые переменные, которые действуют внутри его тела:

```
> (let ((x 7)
        (y 2))
    (format t "Number")
    (+ x y))
Number
9
```

Операторы типа let создают *лексический контекст* (*lexical context*). В нашем примере локальный контекст имеет две новых переменных, которые вне него становятся недоступными.

Вызов let можно понимать как вызов функции. В разделе 2.14 показано, что на функцию можно ссылаться не только по имени, но и по лямбда-выражению. Раз лямбда-выражение эквивалентно имени, то мы можем использовать его вместо имени, ставя первым элементом выражения:

```
> ((lambda (x) (+ x 1)) 3)
4
```

Пример с let, приведенный в начале раздела, может быть переписан как лямбда-вызов:

```
((lambda (x y)
   (format t "Number")
```

```
   (+ x y))
 7
 2)
```

Любой вопрос, возникающий у вас при использовании `let`, легко разрешится, если построить аналогичную конструкцию с помощью `lambda`.°Ё078in

Приведем пример ситуации, которую разъясняет наша аналогия. Как насчет выражения `let`, в котором одна из переменных зависит от другой переменной той же `let`-конструкции?

```
(let ((x 2)
      (y (+ x 1)))
  (+ x y))
```

Значение `x` в `(+ x 1)` не определено, и это видно из соответствующего лямбда-вызова:

```
((lambda (x y) (+ x y)) 2 (+ x 1))
```

Совершенно очевидно, что выражение `(+ x 1)` не может ссылаться на переменную `x` в лямбда-выражении.

Но как быть, если одна из переменных в `let`-вызова зависит от другой? Для этой цели существует оператор `let*`:

```
> (let* ((x 1)
         (y (+ x 1)))
    (+ x y))
3
```

Вызов `let*` полностью эквивалентен серии вложенных `let`. Наш пример можно переписать так:

```
(let ((x 1))
  (let ((y (+ x 1)))
    (+ x y)))
```

Как в `let`, так и в `let*`, переменные могут не иметь исходных значений. В этом случае они приобретают исходное значение `nil` и могут не заключаться в скобки:

```
> (let (x y)
    (list x y))
(NIL NIL)
```

Макрос `destructing-bind` является обобщением `let`. Вместо отдельных переменных он принимает *шаблон* (*pattern*) – одну или несколько переменных, расположенных в виде дерева, которые затем связывает с соответствующими частями реального дерева.

```
> (destructing-bind (w (x y) . z) '(a (b c) d e)
    (list w x y z))
(A B C (D E))
```

В случае несоответствия шаблона дереву, переданному во втором аргументе, возвращается ошибка.

5.3. Условия

Наиболее простым условным оператором можно считать `if`. Все остальные являются его расширениями. Оператор `when` является упрощением `if`. Его тело, которое может состоять из нескольких выражений, вычисляется лишь в случае истинности тестового выражения. Таким образом,

```
(when (oddp that)
  (format t "Hmm, that's odd. ")
  (+ that 1))
```

эквивалентно

```
(if (oddp that)
```

```
    (progn
      (format t "Hmm, that's odd. ")
      (+ that 1)))
```

Оператор `unless` действует противоположно `when`. Его тело вычисляется лишь в случае ложности тестового `выражения`.

Предком всех условных выражений является `cond`, который предлагает две новые особенности: неограниченное количество условных переходов и неявное использование `progn` в каждом из них. Его имеет смысл использовать в случае, когда третий аргумент `if` – другое условное выражение. Например, следующее определение `our-member`:

```
(defun our-member (obj lst)
  (if (atom lst)
      nil
      (if (eql (car lst) obj)
          lst
          (our-member obj (cdr lst)))))
```

может быть определено с помощью `cond`:

```
(defun our-member (obj lst)
  (cond ((atom lst) nil)
        ((eql (car lst) obj) lst)
        (t (our-member obj (cdr lst)))))
```

В процессе исполнения второй пример будет оттранслирован в первый.

В общем случае `cond` может принимать любое число аргументов, в том числе ноль. Каждый аргумент должен быть представлен списком, состоящим из условия и следующих за ним выражений, которые будут вычисляться в случае истинности этого условия. При вычислении вызова `cond` условия проверяются по очереди до тех пор, пока одно из них не окажется истинным. Далее вычисляются выражения, следующие за этим условием и возвращается значение последнего из них. Если после выражения, оказавшегося истинным, нет других выражений, то возвращается его значение:

```
> (cond (99))
99
```

Если условие имеет тестовое выражение `t`, оно будет истинным всегда. Этот факт удобно использовать для указания условия, которое будет выполняться в случае, когда ни одно другое не оказалось истинным. По умолчанию в таком случае возвращается `nil`, однако возвращать `nil` в таком случае может быть опасно и необдуманное использование этой особенности является признаком плохого стиля. (Подобная проблема показана на стр. Ё018out.)

Если вы сравниваете выражение с набором постоянных, уже вычисленных объектов, более правильно пользоваться конструкцией `case`. Например, мы можем воспользоваться `case` для преобразования имени месяца в его продолжительность в днях:

```
(defun month-length (mon)
  (case mon
    ((jan mar may jul aug oct dec) 31)
    ((apr jun sept nov) 30)
    (feb (if (leap-year) 29 28))
    (otherwize "unknown month")))
```

Конструкция `case` начинается с выражения, значение которого будет сравниваться с предложенными вариантами. За ним следуют выражения, начинающиеся либо с ключа, либо со списка ключей, за которым следует произвольное количество выражения. Сами ключи рассматриваются как константы и не вычисляются. Значение ключа (ключей) сравнивается с оригинальным объектом с помощью `eql`. В случае совпадения вычисляются следующие за ключом выражения. Значением вызова `case` считается значение последнего вычисленного выражения. Умолчальное выражение может быть обозначено через `t` или `otherwize`. Если ни

один из ключей не подошел, или же вариант, содержащий подходящий ключ, не содержит выражений, возвращается `nil`:

```
> (case 99 (99))
NIL
```

Макрос `typecase` похож на `case`, но вместо ключей использует спецификаторы типов, а искомое выражение сверяется с ними через `typep` вместо `eql`. (Пример с использованием `typecase` вы найдете на стр. Ё054out.)

5.4. Итерации

Основная итерационная конструкция, `do`, была представлена в разделе 2.13. Мы знаем, что раз `do` неявным образом использует `block` и `tagbody`, внутри него возможно использовать `return`, `return-from` и `go`.

В разделе 2.13 было указано, что первый аргумент `do` должен быть списком, содержащим описания переменных вида

```
(variable initial update)
```

Выражения `initial` и `update` не обязательны. Если выражение `update` пропущено, значение соответствующей переменной не будет меняться. Если пропущено `initial`, то переменной присваивается первоначальное значение `nil`.

В примере на стр. Ё017out:

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))
```

для переменной, создаваемой `do`, задано выражение `update`. Практически всегда в процессе выполнения выражения `do` изменяется хотя бы одна переменная.

В случае модификации сразу нескольких переменных встает вопрос: что произойдет, если `update`-выражение одной из переменных зависит от другой переменной, которая имеет свое `update`-выражение? Какое значение последней переменной будет использоваться: зафиксированное на предыдущей итерации или уже измененное? Макрос `do` использует значение предыдущей итерации:

```
> (let ((x 'a))
    (do ((x 1 (+ x 1))
         (y x x))
        ((> x 5))
      (format t "(~A ~A) " x y)))
(1 A) (2 1) (3 2) (4 3) (5 4)
NIL
```

На каждой итерации значение `x` увеличивается на `1`, а `y` получает значение `x` на предыдущей итерации.

Существует также оператор `do*`, имеющий такое же отношение к `do`, как и `let*` к `let`. Выражения `initial` и `update` могут ссылаться на текущие, а не на предыдущие значения переменных:

```
> (do* ((x 1 (+ x 1))
        (y x x))
       ((> x 5))
    (format t "(~A ~A) " x y))
(1 1) (2 2) (3 3) (4 4) (5 5)
NIL
```

Помимо `do` и `do*` существуют и более специализированные операторы. Итерации по списку выполняются с помощью `dolist`:

```
> (dolist (x '(a b c d) 'done)
    (format t "~A " x))
A B C D
DONE
```

Третье выражение в первом аргументе будет вычислено на выходе из цикла. По умолчанию это `nil`.

Похожим образом действует `dotimes`, пробегающий для заданного числа n значения от `0` до `n-1`:

```
> (dotimes (x 5 x)
    (format t "~A " x))
0 1 2 3 4
5
```

Как и в `dolist`, выходное выражение не обязательно и по умолчанию установлено как `nil`. Обратите внимание, что это выражение может содержать саму итерируемую переменную.

Суть оператора `do`

В книге «The Evolution of Lisp» Стил и Гэбриэл выражают суть `do` настолько точно, что соответствующий отрывок из нее достоин цитирования:

Отложим обсуждение синтаксиса в сторону. Нужно признать, что цикл, выполняющий лишь итерацию по одной переменной, довольно бесполезен, в любом языке программирования. Практически всегда в таких случаях одна переменная используется для получения последовательных значений, в то время как еще одна собирает их вместе. Если цикл лишь увеличивает значение переменной на заданную величину, то каждое новое значение должно устанавливаться благодаря присвоению … или иным побочным эффектам. Функция `do`, умеющая работать с несколькими переменными, находится посередине между генерацией значений и их сбором, позволяя выражать итерации без использования явных побочных эффектов:

```
(defun factorial (n)
  (do ((j n (- j 1))
       (f 1 (* j f)))
      ((= j 0) f)))
```

Разумеется, нет ничего необычного в том, что `do` не выполняет в цикле каких-либо операций, и вся его работа сводится к выполнению шагов итерации.°Ё079in

Ё004inФункция `mapc` похожа на `mapcar`, однако не строит список из вычисленных значений, поэтому может использоваться лишь ради побочных эффектов. Она более гибка, чем `dolist`, потому что может итерировать параллельно сразу по нескольким спискам:

```
> (mapc #'(lambda (x y)
            (format t "~A ~A " x y))
        '(hip flip slip)
        '(hop flop slop))
HIP HOP FLIP FLOP SLIP SLOP
(HIP FLIP SLIP)
```

Функция `mapc` всегда возвращает свой второй аргумент.

5.5. Множественные значения

Чтобы подчеркнуть важность функционального программирования, часто говорят, что в Лиспе каждое выражение возвращает значение. На самом деле, все несколько сложнее. В Common Lisp выражение может возвращать несколько значений или же не возвращать ни одного. Максимальное их количество может различаться в разных реализациях, но должно быть не менее 19.

Для функции, которая вычисляет несколько значений, эта возможность позволяет возвращать их без заключения в какую-либо структуру. Например, встроенная функция `get-decoded-time` возвращает текущее время в виде девяти значений: секунды, минуты, час, день, месяц, год и еще два.

Множественные значения позволяют также функциям поиска разделять случаи, когда элемент не найден и когда найден элемент `nil`. Именно поэтому `gethash` возвращает два значения. Второе значение информирует об успешности поиска,

поэтому мы можем хранить `nil` в хеш-таблице точно так же, как остальные значения.

Возвратить множественные значения можно с помощью функции `values`. Она возвращает в точности те значения, которые были переданы ей в виде аргументов:

```
> (values 'a nil (+ 2 4))
A
NIL
6
```

Если вызов `values` является последним в теле функции, то эта функция возвращает те же множественные значения, что и `values`. Множественные значения могут передаваться через несколько вызовов:

```
> ((lambda () ((lambda () (values 1 2)))))
1
2
```

Тем не менее, если в цепочке передачи значений что-то принимает лишь один аргумент, то все остальные будут потеряны:

```
> (let ((x (values 1 2)))
    x)
1
```

Используя `values` без аргументов, мы имеем возможность не возвращать никаких значений. Если же что-либо будет ожидать значение, будет возвращен `nil`.

```
> (values)
> (let ((x (values)))
    x)
NIL
```

Чтобы получить несколько значения, можно использовать `multiple-value-bind`:

```
> (multiple-value-bind (x y z) (values 1 2 3)
     (list x y z))
(1 2 3)
> (multiple-value-bind (x y z) (values 1 2)
    (list x y z))
(1 2 NIL)
```

Если переменных больше, чем возвращаемых значений, лишним будет присвоен `nil`. Если же значений возвращается больше, чем выделено переменных, то лишние значения будут отброшены. Таким образом, вы можете написать функцию, которая просто печатает текущее время:°Ё080in

```
> (multiple-value-bind (s m h) (get-decoded-time)
    (format nil "~A:~A:~A" h m s))
"4:32:13"
```

Передать какой-либо функции множественные значения в качестве аргументов вы можете с помощью `multiple-value-call`:

```
> (multiple-value-call #'+ (values 1 2 3))
6
```

Кроме того, есть функция `multiple-value list`:

```
> (multiple-value-list (values 'a 'b 'c))
(A B C)
```

которая действует так же, как `multiple-value-call` с функцией `#'list`.

5.6. Прерывания

Чтобы выйти из блока, достаточно вызвать `return`. Однако, иногда вам может потребоваться нечто более радикальное, например, передача управления назад через несколько вызовов функций. Для этого существуют `catch` и `throw`. Конструкция

case использует тег, которым может быть любой объект. За тегом следует набор выражений.

```
(defun super ()
  (catch 'abort
    (sub)
    (format t "We'll never see this.")))

(defun sub ()
  (throw 'abort 99))
```

Выражения после тега вычисляются так же, как в progn. Вызов throw внутри любого из этих выражений, приведет к немедленному выходу из catch:

```
> (super)
99
```

Функция throw с заданным тегом передаст управление соответствующей конструкции catch (то есть, завершит ее выполнение) через любое количество вызовов catch. Соответствие определяется с помощью тега. Если catch с искомым тегом не найден, throw возвращает ошибку.

Вызов error также прерывает выполнение, но вместо передачи управления вверх по дереву вызовов, он передает его обработчику ошибок Лиспа. В результате вы, скорее всего, попадете в отладчик (break loop). Вот что произойдет в некой абстрактной реализации Common Lisp:

```
> (progn
    (error "Oops! ")
    (format t "After the error. "))
Error: Oops!
       Options: :abort, :backtrace
>>
```

За более подробной информацией об ошибках и условиях вы можете обратиться в приложение A.

Иногда вам может понадобиться некая гарантия защиты от ошибок и прерываний типа throw. Используя unwind-protect, вы можете быть уверены, что в результате подобных явлений программа не окажется в противоречивом состоянии. Конструкция unwind-protect принимает любое количество аргументов и возвращает значение первого. Остальные выражения будут вычислены, даже если вычисление первого будет прервано.

```
> (setf x 1)
1
> (catch 'abort
    (unwind-protect
      (throw 'abort 99)
      (setf x 2)))
99
> x
2
```

Даже несмотря на то, что throw передал управление catch, второе выражение было вычислено прежде, чем был осуществлен выход из catch. В случае, когда перед преждевременным завершением необходимы какие-то завершающие операции, unwind-protect может быть очень полезен. Один из таких примеров представлен на стр. Ё016out.

5.7. Пример: арифметические операции с датами

В некоторых приложения полезно иметь возможность складывать и вычитать даты, например, чтобы сказать, что через 60 дней после 17 декабря 1997 года настанет 15 февраля 1998. В этом разделе мы создадим необходимые для этого инструменты. Нам потребуется конвертировать даты в целые числа, такие, что ноль – это 1 января 2000 года. Далее мы сможем работать с датами как с целыми числами, используя

функции + и −. Кроме того, необходимо уметь конвертировать целое число обратно в дату.

Чтобы преобразовать дату в целое число, будем складывать количества дней, соответствующие различным компонентам даты. Например, число, соответствующее 13 ноября 2004 года, получим, складывая количество дней до 2004 года, количество дней в году до ноября и число 13.

Нам потребуется таблица, устанавливающая соответствие между месяцем и количеством дней в нем для невисокосного года. Начнем со списка, содержащего длины месяцев:

```
> (setf mon '(31 28 31 30 31 30 31 31 30 31 30 31))
(31 28 31 30 31 30 31 31 30 31 30 31)
```

Проведем несложную проверку, сложив все эти числа:

```
> (apply #'+ mon)
365
```

Теперь перевернем этот список и применим с помощью `maplist` функцию + к последовательности хвостов (`cdr`) списка. Мы получим количества дней, прошедшие до начала каждого последующего месяца:

```
> (setf nom (reverse mon))
(31 30 31 30 31 31 30 31 30 31 28 31)
> (setf sums (maplist #'(lambda (x)
                          (apply #'+ x))
                      nom))
(365 334 304 273 243 212 181 151 90 59 31)
> (reverse sums)
(31 59 90 120 151 181 212 243 273 304 334 365)
```

Эти цифры означают, что до начала февраля пройдет 31 день, до начала марта 59, и так далее.

На рис. 5.1. приведен код, выполняющий преобразования дат. В нем полученный нами список преобразуется в вектор.

```
(defconstant month
  #(0 31 59 90 120 151 181 212 243 273 304 334 365))

(defconstant yzero 2000)

(defun leap? (y)
  (and (zerop (mod y 4))
       (or (zerop (mod y 400))
           (not (zerop (mod y 100))))))

(defun date->num (d m y)
  (+ (- d 1) (month-num m y) (year-num y)))

(defun month-num (m y)
  (+ (svref month ( - m 1))
     (if (and (> m 2) (leap? y)) 1 0)))

(defun year-num (y)
  (let ((d 0))
    (if (>= y yzero)
        (dotimes (i (- y yzero) d)
          (incf d (year-days (+ yzero i))))
        (dotimes (i (- yzero y) (- d))
          (incf d (year-days (+ y i)))))))

(defun year-days (y) (if (leap? y) 366 365))
```

Рис. 5.1. Операции с датами: преобразование дат в целые числа

Жизненный цикл обычной Лисп-программы выглядит так: она сначала пишется, потом читается, компилируется и затем исполняется. Отличительной особенностью Лиспа является тот факт, что Лисп-система принимает участие в каждом шаге этой последовательности. Вы можете взаимодействовать с Лиспом не только во время работы программы, но также во время компиляции (раздел 10.2) и даже ее чтения (раздел 14.3). Способ, каким образом мы создали вектор `month`, свидетельствует о том, что мы используем Лисп даже во время написания программы.

Производительность программы имеет значение лишь на четвертом этапе – выполнении программы. На первых трех этапах вы можете в полной мере пользоваться гибкостью и мощью списков, не задумываясь об их стоимости.

Если вы вздумаете использовать этот код для управления машиной времени, то люди могут не согласиться с вами по поводу текущей даты, если вы попадете в прошлое. По мере накопления информации о продолжительности года люди вносили изменения в календарь. В англоговорящих странах подобная нестыковка последний раз устранялась в 1752 году, когда сразу после 2 сентября последовало 14 сентября.°Ё081in

Количество дней в году зависит от того, является ли он високосным. Год не является високосным, если он не кратен 4, либо кратен 100 и не кратен 400. 1904 год был високосным, 1900 не был, а 1600 был им.

Для определения делимости существует функцияЁ031in `mod`, возвращающая остаток от деления первого аргумента на второй:

```
> (mod 23 5)
3
> (mod 25 5)
0
```

Одно число считается делимым на другое, если остаток от деления на него равен нулю. Функция `leap?` использует этот подход для определения високосности года:

```
> (mapcar #'leap? '(1904 1900 1600))
(T NIL T)
```

Дату в целое число преобразует функция `date->num`. Она возвращает сумму численных представлений каждого компонента даты. Чтобы выяснить, сколько дней прошло до начала заданного месяца, она использует функцию `month-num`, которая

обращается к соответствующему компоненту вектора month, добавляя к нему 1, если год високосный, и заданный месяц находится после февраля.

Чтобы найти количество дней до наступления заданного года, date->num вызывает year-num, которая возвращает численное представление 1 января этого года. Эта функция отсчитывает годы до нулевого (то есть, 2000) года.

На рис. 5.2. показан код второй части программы. Функция num->date преобразует целые числа обратно в даты. Вызывая num-year, она получает год и количество дней в остатке, по которому num-month вычисляет месяц и день.

```
(defun num->date (n)
  (multiple-value-bind (y left) (num-year n)
    (multiple-value-bind (m d) (num-month left y)
      (values d m y))))

(defun num-year (n)
  (if (< n 0)
      (do* ((y (- yzero 1) (- y 1))
            (d (- (year-days y)) (- d (year-days y))))
           ((<= d n) (values y (- n d))))
      (do* ((y yzero (+ y 1))
            (prev 0 d)
            (d (year-days y) (+ d (year-days y))))
           ((> d n) (values y (- n prev))))))

(defun num-month (n y)
  (if (leap? y)
      (cond ((= n 59) (values 2 29))
            ((> n 59) (nmon (- n 1)))
            (t        (nmon n)))
      (nmon n)))

(defun nmon (n)
  (let ((m (position n month :test #'<)))
    (values m (+ 1 (- n (svref month (- m 1)))))))

(defun date+ (d m y n)
  (num->date (+ (date->num d m y) n)))
```

Рис. 5.2. Операции с датами: преобразование целых чисел в даты

Как и year-num, num-year ведет отсчет от 2000 года, складывая продолжительности каждого года до тех пор, пока эта сумма не превысит заданное число (или не будет равна ему). Если сумма, полученная на какой-то итерации, превысит его, то num-year возвращает год, полученный на предыдущей итерации. Для этого введена переменная prev, которая хранит число дней, полученное на предыдущей итерации.

Функция num-month и ее подпроцедура nmon ведут себя обратно month-num. Они вычисляют позицию в векторе month по численному значению, тогда как month-num вычисляет значение, исходя из заданного элемента вектора.

Первые две функции на рис. 5.2. могут быть собраны в одну. Вместо вызова отдельной функции num-year могла бы вызывать непосредственно num-month. Однако, на этапе тестирования и отладки текущий вариант более удобен, и объединение функций может стать следующим шагом после ее тестирования.

Используя date->num и num->date, мы имеем простую арифметику дат.°Ё082in С их помощью работает функция date+, позволяющая складывать и вычитать дни из заданной даты. Теперь мы сможем вычислить дату по прошествии 60 дней с 17 декабря 1997 года:

```
> (multiple-value-list (date+ 17 12 1997 60))
(15 2 1998)
```

Мы получили 15 февраля 1998 года.

Итоги главы

1. В Common Lisp есть три основных конструктора блоков: `progn`; `block`, позволяющий выход из блока; `tagbody`, позволяющий переходы по меткам. Многие встроенные операторы неявно используют их.
2. Создание нового лексического контекста эквивалентно вызову функции.
3. Common Lisp предоставляет условные операторы, приспособленные для различных ситуаций. Все они могут быть определены с помощью `if`.
4. В Common Lisp есть также разнообразные итерационные операторы.
5. Выражения могут возвращать несколько значений.
6. Вычисления могут быть прерваны, а так же могут быть защищены от неблагоприятных последствий таких прерываний.

Упражнения

1. Запишите следующие выражения без использования `let` или `let*`, а также не вычисляющие одно и то же выражение дважды:

```
(a) (let ((x (car y)))
      (cons x x))
(b) (let* ((w (car x))
           (y (+ w z)))
      (cons w y))
```

2. Перепишите функцию `mystery` (см. стр. Ё019out) с использованием `cons`.
3. Определите функцию, возвращающую квадрат своего аргумента лишь когда аргумент – положительное число, меньше или равное пяти.
4. Перепишите `month-num` (см. рис. 5.1), используя `case` вместо `svref`.
5. Определите функцию (рекурсивную и итеративную версии) от объекта `x` и вектора `v`, возвращающую список объектов, следующих непосредственно перед `x` в `v`:

```
> (precedes #\a "abracadabra")
(#\c #\d #\r)
```

6. Определите функцию (итеративно и рекурсивно), принимающую объект и список, и возвращающую новый список, в котором заданный элемент находится между каждой парой элементов исходного списка:

```
> (intersperse '- '(a b c d))
(A - B - C - D)
```

7. Определите функцию, принимающую список чисел м возвращающую истину, если разница между каждой последующей их парой равна `1`. Используйте:

   (a) рекурсию

   (b) `do`

   (c) `mapc` и `return`

8. Определите одиночную рекурсивную функцию, которая возвращает два значения – максимальный и минимальный элементы вектора.
9. Программа на рис. 3.12. продолжает поиск после нахождения первого подходящего пути в очереди. Для больших сетей это может стать проблемой.

   (a) Используя `catch` и `throw`, измените программу таким образом, чтобы она возвращала первый найденный путь сразу же после его отыскания.

   (b) Перепишите программу, чтобы она делала то же самое без использования `catch` и `throw`.

6. Функции

Понимание функций – один из ключей к пониманию всего Лиспа. Функции – основная концепция, которая легла в основу Лиспа. На практике это наиболее полезный инструмент, находящийся в вашем распоряжении.

6.1. Глобальные функции

Предикат `fboundp` сообщает, существует ли функция с именем, заданным в виде символа. Если какой-либо символ является именем функции, то ее можно получить с помощью `symbol-function`:

```
> (fboundp '+)
T
> (symbol-function '+)
#<Compiled-Function + 17BA4E>
```

Используя `setf` над `symbol-function`:

```
> (setf (symbol-function 'add2)
        #'(lambda (x) (+ x 2)))
```

мы можем определить глобальную функцию, которой сможем пользоваться точно так же, как если бы определили ее с помощью `defun`:

```
> (add2 1)
3
```

Фактически, `defun` делает немногим более, чем просто трансляцию выражения типа:

```
(defun add2 (x) (+ x 2))
```

в выражение с `setf`. Однако, использование `defun` делает программы более легкими для понимания, а также сообщает компилятору некоторую информацию о функции, хотя, строго говоря, использовать `defun` при написании программ вовсе не обязательно.

Задать поведение `setf` для какой-либо функции, то есть, для выражения вида `(setf f)`, вы можете, сообщив такое выражение первым аргументом для `defun` (там, где обычно стоит имя функции). Первый ее параметр будет соответствовать устанавливаемому значению, а остальные – аргументам `f`.°Ё083in Ниже приведен пример функции `primo`, являющейся аналогом `car`:

```
(defun primo (lst) (car lst))

(defun (setf primo) (val lst)
  (setf (car lst) val))
```

В последнем определении на месте функции находится выражение `(setf primo )`, первым параметром является устанавливаемое значение, а последующий параметр – аргумент `primo`.°Ё084in
Теперь любой `setf` для `primo` будет являться вызовом функции, определенной выше:

```
> (let ((x (list 'a 'b 'c)))
     (setf (primo x) 480)
     x)
(480 B C)
```

Совсем не обязательно определять саму функцию `primo`, чтобы определить поведение `(setf primo)`, однако, такие определения обычно даются парами.

Так как строки в Лиспе – полноценные выражения, то ничто не мешает им находиться внутри тела функции. Сама по себе строка не производит побочных эффектов, и нахождение ее в середине тела функции является нецелесообразным.

Однако, если строка будет первым выражением, то она будет считаться строкой документации к функции.

```
(defun foo (x)
  "Implements an enhanced paradigm of diversity."
  x)
```

Документация к функции, определенной глобально, может быть получена с помощью `documentation`:

```
> (documentation 'foo 'function)
"Implements an enhanced paradigm of diversity."
```

6.2. Локальные функции

Функции, определенные через `defun` или `setf` вместе с `symbol-function`, являются *глобальными*. Как и глобальные переменные, они могут быть использованы везде. Тем не менее, есть возможность определять *локальные* функции, доступные, как и локальные переменные, лишь внутри своего контекста.

Локальные функции могут быть определены через конструкцию `labels`[1], своего рода `let` для функций. Ее первым аргументом является не список, задающий переменные, а список определений локальных функций. Каждый элемент этого списка является списком следующего вида:

```
(name parameters . body)
```

Внутри `labels` любое выражение с `name` эквивалентно лямбда-вызову (`lambda parameters . body`).

```
> (labels ((add10 (x) + x 10))
          (consa (x) (cons 'a x)))
    (consa (add10 3)))
(A . 13)
```

Аналогия с `let`, однако, не является полной. Локальная функция в `labels` может ссылаться на любые другие функции, определенные в той же конструкции `labels`, в том числе и на саму себя. Это дает возможность создания с помощью `labels` локальных рекурсивных функций:

```
> (labels ((len (lst)
             (if (null lst)
                 0
                 (+ (len (cdr lst)) 1))))
    (len '(a b c)))
3
```

В разделе 5.2 упоминалось, что конструкция `let` может рассматриваться как вызов функции. Аналогично, выражение с `do` может считаться вызовом рекурсивной функции. Выражение:

```
(do ((x a (b x))
    (y c (d y)))
   ((test x y) (z x y))
  (f x y))
```

эквивалентно

```
(labels ((rec (x y)
          (cond ((test x y)
                 (z x y))
                (t
                 (f x y)
                 (rec (b x) (d y))))))
  (rec a c))
```

Такая аналогия позволяет снять многие вопросы об особенностях поведения `do`.

---

[1] В большинстве случаев (если локальные функции не вызывают друг друга и самих себя) лучше использовать `flet` вместо `labels`. Это не только более эффективно, но и считается более хорошим тоном. Описание `flet` вы найдете ниже. – *Прим. перев.*

6.3. Списки параметров

В разделе 2.1 было показано, что благодаря префиксной нотации + может принимать любое количество аргументов. Далее мы познакомились с многими подобными функциями. Чтобы написать такую функцию самостоятельно, нам придется воспользоваться параметром *остаток* (*rest*).

Ё008inПоместив элемент `&rest` перед последним элементом в списке параметров функции, то при вызове этой функции последний параметр получит список всех оставшихся аргументов. Теперь мы сможем написать `funcall` с помощью `apply`:

```
(defun our-funcall (fn &rest args)
  (apply fn args))
```

Кроме того, мы уже знакомились с параметрами, которые могут быть пропущены и в таком случае получат умолчальное значение. Такие параметры называются *необязательными* (*optional*) (Для сравнения, обычные параметры иногда называются *обязательными* (*required*)) Если символ `&optional` встречается в списке параметров функции:

```
(defun philosoph (thing &optional property)
  (list thing 'is property))
```

то все последующие аргументы будут необязательными, а их значением по умолчанию будет `nil`:

```
> (philosoph 'death)
(DEATH IS NIL)
```

Задать исходное значение явным образом можно, заключив это значение в скобки вместе с параметром:

```
(defun philosoph (thing &optional (property 'fun))
  (list thing 'is property))
```

Значением по умолчанию может быть не только константа, но и любое Лисп-выражение. Оно будет вычисляться заново каждый раз, когда потребуется.

Ключевые (*keyword*) параметры предоставляют большую гибкость, чем необязательные. Поместив символ `&key` в список параметров, вы помечаете все последующие параметры как необязательные. Кроме того, при вызове функции эти параметры будут передаваться не в соответствии с их ожидаемым положением, а по соответствующему тегу – ключевому слову:

```
> (defun keylist (a &key x y z)
    (list a x y z))
KEYLIST
> (keylist 1 :y 2)
(1 NIL 2 NIL)
> (keylist 1 :y 3 :x 2)
(1 2 3 NIL)
```

Как и обычные необязательные параметры, умолчальным значением ключевых также является `nil`, однако исходное значение может быть задано таким же образом, как и для необязательного параметра.

Ключевые слова и связанные с ними значения могут собираться с помощью `&rest` и в таком виде передаваться другой функции, которая способна их распознать. Например, мы могли бы определить `adjoin` следующим образом:

```
(defun our-adjoin (obj lst &rest args)
  (if (apply #'member obj lst args)
    lst
    (cons obj lst)))
```

Так как `adjoin` принимает те же ключевые параметры, что и `member`, достаточно их просто собрать в список и передать `member`.

В разделе 5.2 был представлен макрос `destructing-bind`. В общем случае, каждое поддерево шаблона, заданного первым аргументом, может быть устроено так же сложно, как и список параметров функции:

```
> (destructing-bind ((&key w x) &rest y) '((:w 3) a)
    (list w x y))
(3 NIL (A))
```

6.4. Пример: утилиты

В разделе 2.6. упоминалось, что Лисп состоит по большей части из функций, точно таких же, какие вы можете определять самостоятельно. Эта особенность крайне полезна, потому что такой язык программирования не только не требует подгонки задачи под себя, но и сам может быть подстроен под каждую задачу. Если вы захотите видеть в Common Lisp какую-либо конкретную функцию, вы можете написать ее самостоятельно, и она станет такой же частью языка, как `+` или `eql`.

Опытные Лисп-разработчики создают программы как снизу-вверх, так и сверху-вниз, подстраивая одновременно язык под задачу и задачу под язык, пока, рано или поздно, они не встретятся, или не окажутся очень близки. Операторы, создаваемые с этой целью, называют *утилитами*. Создавая программы, вы обнаружите, что некоторые функции, созданные для одной программы, могут пригодиться и для другой.

Профессиональные разработчики используют эту довольно привлекательную идею для создания повторно используемых программ. Эта идея некоторым образом связана с объектно-ориентированным программированием, однако программа вовсе не обязана быть объектно-ориентированной, чтобы быть пригодной к повторному использованию. Подтверждением тому являются сами языки программирования (точнее, их компиляторы), которые являются, вероятно, наиболее приспособленными к повторному использованию.

Основной секрет в написании таких программ – подход снизу-вверх, и такие программы вовсе не обязаны быть объектно-ориентированными. Функциональный стиль соответствует ему даже в большей степени, нежели объектно-ориентированный. Посмотрим, например, на `sort`. Имея в распоряжении эту функцию, вам вряд ли потребуется писать свои сортирующие процедуры, потому что `sort` эффективна и может быть приспособлена к самым разным задачам. Именно *это* и называется повторным использованием.

Вы можете создавать самостоятельно подобные вещи, называемые утилитами. На рис. 6.1 представлена небольшая выборка полезных утилит. Первые две, `single?` и `append1`, приведены, чтобы показать, что даже очень короткие утилиты могут быть полезными. Первая из них возвращает истину, когда ее аргумент – список из одного элемента:

```
(defun single? (lst)
  (and (consp lst) (null (cdr lst))))

(defun append1 (lst obj)
  (append lst (list obj)))

(defun map-int (fn n)
  (let ((acc nil))
    (dotimes (i n)
      (push (funcall fn i) acc))
    (nreverse acc)))

(defun filter (fn lst)
  (let ((acc nil))
    (dolist (x lst)
      (let ((val (funcall fn x)))
        (if val (push val acc))))
    (nreverse acc)))

(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcall fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setf wins obj
                    max  score))))
        (values wins max))))
```

Рис. 6.1. Утилиты

```
> (single? '(a))
T
```

Вторая напоминает `cons`, однако добавляет элемент в конец списка, а не в начало, в отличие от `cons`:

```
> (append1 '(a b c) 'd)
(A B C D)
```

Следующая утилита, `map-int`Ё040in, принимает функцию и целое число `n`, возвращая список, состоящий из значений этой функции для всех целых чисел от `0` до `n-1`.

Такая утилита может пригодиться при тестировании кода. (Одно из преимуществ Лиспа заключается в интерактивности разработки, благодаря которой вы можете создавать одни функции для тестирования других.) Если нам вдруг понадобится список чисел от 0 до 9, мы сможем сказать:

```
> (map-int #'identity 10)
(0 1 2 3 4 5 6 7 8 9)
```

Получить список из десяти случайных чисел между 0 и 99 можно, проигнорировав параметр лямбда-выражения:

```
> (map-int #'(lambda (x) (random 100))
           10)
(85 40 73 64 28 21 67 5 32)
```

На примере `map-int` показана распространенная Лисп-идиома для построения списков. В ней создается аккумулятор `acc`, который исходно содержит `nil`, и последовательно добавляем в него элементы с помощью `push`. По завершении возвращается перевернутый список `acc`.[1]

[1] Более правильным здесь будет использование `nreverse` (см. стр. Ё062out), которая в данном случае ничем не отличается от `reverse`, однако более эффективна.

Эту же идиому мы видим и в `filter`Ё026in. Функция `filter` принимает функцию и список, применяя эту функцию к каждому его элементу и возвращая список, состоящий только из элементов, не являющихся `nil`:

```
> (filter #'(lambda (x)
              (and (evenp x) (+ x 10)))
          '(1 2 3 4 5 6 7))
(12 14 16)
```

Функцию `filter` можно также рассматривать как обобщение `remove-if`.

Последняя функция на рис. 6.1, `most`, возвращает элемент, имеющий наивысший рейтинг с точки зрения заданной рейтинговой функции. Помимо этого элемента, она также возвращает соответствующие ему значение.

```
> (most #'length '((a b) (a b c) (a)))
(A B C)
3
```

Если таких элементов несколько, `most`Ё025in возвращает первый из них.

Обратите внимание, последние три функции на рис. 6.1 принимают другие функции в качестве аргументов. Для Лиспа это явление совершенно естественно и является одной из причин его приспособленности к программированию снизу-вверх.°Ё085in Хорошая утилита должна быть как можно более обобщенной, а особенности ее поведения должны следовать из задаваемых аргументов.

Функции, приведенные в этом разделе, действительно являются утилитами общего назначения и могут быть использованы в самых разных программах. Однако утилиты могут быть и более специализированными, чтобы использоваться в программах одного рода. Более того, далее будет показано, как создавать поверх Лиспа специализированные языки, полностью удовлетворяющие задаче, что, в свою очередь, может отлично подходить для создания повторно используемых программ.

6.5. Замыкания

Функция, как и любой другой объект, может возвращаться как значение выражения. Ниже приведен пример функции, которая возвращает функцию, применимую для сочетания объектов того же типа, что и ее аргумент:Ё054in

```
(defun combiner (x)
  (typecase x
    (number #'x)
    (list   #'append)
    (t      #'list)))
```

С ее помощью мы сможем создать комбинирующую функцию общего вида:

```
(defun combine (&rest args)
  (apply (combiner (car args))
         args))
```

Она принимает аргументы любого типа и объединяет их в соответствии с их типом. (Для упрощения рассматривается лишь тот случай, когда все аргументы одного типа.)

```
> (combine 2 3)
5
> (combine '(a b) '(c d))
(A B C D)
```

В разделе 2.10 было объяснено, что лексические переменные действительны только внутри контекста, в котором были определены. Вместе с этим ограничением мы получаем обещание, что они будут *по-прежнему* действительны, если этот контекст все еще используется где-нибудь.

Если функция была определена в зоне действия лексической переменной, то она сможет продолжать ссылаться на эту переменную даже будучи возвращенной как значение, а затем использованной вне контекста этой переменной. Ниже мы приводим функцию, добавляющую 3 к своему аргументу:

```
> (setf fn (let ((i 3))
             #'(lambda (x) (+ x i))))
#<Interpreted-Function C0A51E>
> (funcall fn 2)
5
```

Если функция использует определенную вне ее переменную, то такая переменная называется *свободной*. Функцию, ссылающуюся на свободную лексическую переменную, принято называть *замыканием* (*closure*)[1]. Свободная переменная будет существовать до тех пор, пока доступна использующая ее функция.

Замыкание – это сочетание функции и окружения. Замыкания создаются неявно в случае, когда функция ссылается на что-либо из лексического окружения, в котором она была определена. Это происходит без каких-либо внешних проявлений в функции, приведенной ниже:Ё020in

```
(defun add-to-list (num lst)
  (mapcar #'(lambda (x)
              (+ x num))
          lst))
```

Функция принимает число и список, возвращая новый список, в котором, по сравнению с исходным, к каждому элементу добавлении заданное число. Переменная `num` является свободной, значит, функции `mapcar` передается замыкание.

Более очевидным примером является функция, возвращающая само замыкание. Следующая функция возвращает замыкание, выполняющее сложение с заданным числом:

```
(defun make-adder (n)
  #'(lambda (x)
      (+ x n)))
```

Возвращаемая функция складывает число `n` с *ее* аргументом:

```
> (setf add3 (make-adder 3))
#<Interpreted-Function C0EBF6>
> (funcall add3 2)
5
> (setf add27 (make-adder 27))
#<Interpreted-Function C0EE4E>
> (funcall add27 2)
29
```

Более того, несколько замыканий могут использовать одну и ту же переменную. Ниже определены две функции, использующие переменную `counter`Ё038in:

```
(let ((counter 0))
  (defun reset ()
    (setf counter 0))
  (defun stamp ()
    (setf counter (+ counter 1))))
```

Такая пара функций может использоваться для создания временных меток. Каждый раз, вызывая `stamp`, мы получаем число, на единицу большее, чем предыдущее. Вызывая `reset`, мы сбрасываем счетчик в ноль:

```
> (list (stamp) (stamp) (reset) (stamp))
(1 2 0 1)
```

Несомненно, можно сделать то же самое с использованием глобальной переменной, однако последняя не защищена от воздействий извне.

В Common Lisp существует функция `complement`, принимающая предикат и возвращающая противоположный ему. Например:

```
> (mapcar (complement #'oddp)
          '(1 2 3 4 5 6))
```

[1] Название «замыкание» взято из ранних диалектов Лиспа. Оно происходит из метода, с помощью которого замыкания реализовывались в динамическом окружении.

```
(NIL T NIL T NIL T)
```

Благодаря замыканиям такую написать функцию очень просто:

```
(defun our-complement (f)
  #'(lambda (&rest args)
      (not (apply f args))))
```

Если вы отвлечетесь от конкретного примера, то, вероятно, обнаружите, что этот маленький, но замечательный пример – лишь вершина айсберга. Замыкания являются одной из уникальных черт Лиспа. Они открывают путь к новым возможностям в программировании, недостижимым в других языках.°Ё086in

6.6. Пример: компоновщики функций

Язык Dylan известен, как гибрид Scheme и Common Lisp, использующий синтаксис Pascal.°Ё087in Он имеет большой набор функций, которые возвращают функции. Помимо `complement`, рассмотренной нами в предыдущем разделе, Dylan включает `compose`, `disjoin`, `conjoin`, `curry`, `rcurry` и `always`. На рис. 6.2. приведены Common Lisp реализации этих функций, а на рис. 6.3. показаны эквиваленты, следующие из этих определений.

```
(defun compose (&rest fns)
  (destructuring-bind (fn1 . rest) (reverse fns)
    #'(lambda (&rest args)
        (reduce #'(lambda (v f) (funcall f v))
                rest
                :initial-value (apply fn1 args)))))

(defun disjoin (fn &rest fns)
  (if (null fns)
      fn
      (let ((disj (apply #'disjoin fns)))
        #'(lambda (&rest args)
            (or (apply fn args) (apply disj args))))))

(defun conjoin (fn &rest fns)
  (if (null fns)
      fn
      (let ((conj (apply #'conjoin fns)))
        #'(lambda (&rest args)
            (and (apply fn args) (apply conj args))))))

(defun curry (fn &rest args)
  #'(lambda (&rest args2)
      (apply fn (append args args2))))

(defun rcurry (fn &rest args)
  #'(lambda (&rest args2)
      (apply fn (append args2 args))))

(defun always (x) #'(lambda (&rest args) x))
```

Рис. 6.2. Компоновщики функций из Dylan

```
(cddr = (compose #'cdr # 'cdr)
 nth = (compose #'car #'nthcdr)
atom = (compose #'not # 'consp)
     = (rcurry #'typep 'atom)
  <= = (disjoin #'< #'=)
listp = (disjoin #'null #'consp)
     = (rcurry #'typep 'list)
  1+ = (curry #' + 1)
     = (rcurry #' + 1)
  1- = (rcurry #'- 1)
mapcan = (compose (curry #'apply #'nconc) #'mapcar)
complement = (curry #'compose #'not)
```

Рис. 6.3. Некоторые эквиваленты

Первая из них, `compose`, принимает одну или несколько функций, возвращая новую функцию, которая применяет их по очереди. Таким образом,

```
(compose #'a #'b #'c)
```

возвращает функцию, эквивалентную вызову:

```
#'(lambda (&rest args) (a (b (apply #'c args))))
```

Из приведенного примера следует, что последняя функция может принимать нефиксированное количество аргументов, в то время как остальные требуют лишь один.

Давайте создадим функцию, вычисляющую квадратный корень числа, округляющую результат и создающую содержащий его список:

```
> (mapcar (compose #'list #'round #'sqrt)
          '(4 9 16 25))
((2) (3) (4) (5))
```

Следующие две функции, `disjoin` и `conjoin`, обе принимают некоторый набор предикатов. Вызов `disjoin` возвращает предикат, возвращающий истину, если хотя бы один из заданных предикатов является истинным. Функция `conjoin` возвращает предикат, который будет истинным лишь в случае соответствия проверяемого аргумента всем заданным предикатам.

```
> (mapcar (disjoin #'integerp #'symbols)
          '(a "a" 2 3))
(T NIL T  T)
> (mapcar (conjoin #'integerp #'oddp)
          '(a "a" 2 3))
(NIL NIL NIL T)
```

Рассматривая предикаты как множества, `disjoin` возвращает объединение множеств, а `conjoin` – их пересечение.

Функции `curry` и `rcurry` («right curry») имеют общую идею с определенной в предыдущем разделе `make-adder`. Они обе принимают функцию и некоторые ее аргументы, возвращая функцию, которая ожидает получить лишь те аргументы, которые не были заданы в момент ее создания. Каждое из следующих выражений равносильно `(make-adder 3)`:

```
(curry #'+ 3)
(rcurry #'+ 3)
```

Разница между `curry` и `rcurry` станет заметна на примере функции, различающей свои аргументы. Например, для функции - функция, полученная с помощью `curry`, будет вычитать свой аргумент из заданного числа:

```
> (funcall (curry #'- 3) 2)
1
```

в то время как для `rcurry` полученная функция будет вычитать заданное число из своего аргумента:

```
> (funcall (rcurry #'- 3) 2)
```

```
-1
```

Наконец, `always` имеет свой аналог в Common Lisp – `constanly`. Она принимает любой аргумент и возвращает функцию, всегда возвращающую его. Эта функция напоминает `identity` и употребляется, как правило, когда требуется функциональный аргумент.

6.7. Динамическое окружение

В разделе 2.11 было показано различие между локальными и глобальными переменными. В действительности, различают лексические переменные, которые имеют лексическое окружение, и специальные переменные, имеющие динамическое окружение. На практике эти понятия можно считать эквивалентными, потому что локальные переменные практически всегда являются лексическими, а специальные переменные всегда являются глобальными.

Внутри лексического окружения символ будет ссылаться на переменную, которая имеет такое же имя в контексте, где появляется этот символ. Локальные переменные по умолчанию имеют лексическое окружение. Итак, если мы определим функцию в окружении, содержащем переменную `x`:

```
(let ((x 10))
  (defun foo ()
    x))
```

то символ `x`, используемый в этой функции, будет ссылаться на эту переменную, независимо от того, будет ли в окружении вызова этой функции переменная `x`, имеющая другое значение:

```
> (let ((x 20)) (foo))
10
```

При использовании динамического окружения используется то значение переменной, которое имеется в том окружении, в котором функция вызывается, а не в том, где она была определена.°Ё088in Вынудить переменную использовать динамическое окружение можно с помощью декларации `special` в любом месте, где она встречается. Если мы определим `foo` следующим образом:

```
(let ((x 10))
  (defun foo ()
    (declare (special x))
    x))
```

то переменная `x` более не будет принадлежать лексическому окружению, в котором функция была определена, а при вызове функции `x` будет получать значение из текущего окружения:

```
> (let ((x 20))
    (declare (special x))
    (foo))
20
```

Декларация `declare` может находиться в любом месте кода, создающего новую переменную. Декларация `special` уникальна тем, что может изменить поведение программы. В главе 13 будут рассмотрены другие декларации. Прочие из них являются всего лишь советами компилятору, которые помогают сделать программу быстрее, но не изменяют ее поведение.

Глобальные переменные, установленные с помощью `setf` в toplevel, подразумеваются специальными:

```
> (setf x 30)
30
> (foo)
30
```

Тем не менее, правильнее избегать неявных деклараций такого рода, используя вместо этого `defparameter`. Такой подход позволит сделать программу более легкочитаемой.

Где может быть полезным динамическое окружение? Обычно оно используется, чтобы присвоить некоторой глобальной переменной новое временное значение. Например, для управления параметрами печати объектов используются 11 глобальных переменных, включая `*print-base*`, по умолчанию установленную как `10`. Чтобы печатать числа в шестнадцатеричном виде (с основанием 16), вы нужно всего лишь переопределить `*print-base*`:Ё043in

```
> (let ((*print-base* 16))
    (princ 32))
20
32
```

В этом примере печатаются два числа: вывод `princ` и значение, которое она возвращает. Они представляют одно и то же число, напечатанное сначала в шестнадцатеричном формате, так как `*print-base*` имела значение `16`, а затем в десятичном, вне выражения `let`, где `*print-base*` снова имела значение `10`.

6.8. Компиляция

В Common Lisp имеется возможность компилировать файл целиком или функции по отдельности. Если вы просто наберете определение функции в toplevel:

```
> (defun foo (x) (+ x 1))
FOO
```

то многие реализации создадут интерпретируемый код. Проверить, является ли функция скомпилированной, вы можете с помощью `compiled-function-p`:

```
> (compiled-function-p #'foo)
NIL
```

Скомпилировать функцию можно, сообщив `compile` имя нужной функции:

```
> (compile 'foo)
FOO
```

Скомпилированные и интерпретированные функции ведут себя абсолютно одинаково, за исключением отношения к `compiled-function-p`.

Функция `compile` понимает так же списки. Такое использование `compile` обсуждается на стр. Ё053out.

К некоторым функциям `compile` неприменима – это функции типа `stamp` или `reset`, определенные через toplevel в собственном (созданном `let`) лексическом окружении[1]. Вам придется набрать эти функции в файле и затем его скомпилировать и загрузить. Этот запрет установлен по техническим причинам, а не потому, что что-то не так определением функций в иных лексических окружениях.

Чаще всего функции компилируются не по отдельности, а в составе файла с помощью `compile-file`. Эта функция создает скомпилированную версию заданного файла, обычно – с тем же именем, но другим расширением. После загрузки скомпилированного файла `compiled-function-p` вернет истину для любой функции из этого файла.

Если одна функция используется внутри другой, то она также должна быть скомпилирована. Таким образом, `make-adder` (см. стр. Ё020out), будучи скомпилированной, возвращает скомпилированную функцию:

```
> (compile 'make-adder)
MAKE-ADDER
> (compiled-funcion-p (make-adder 2))
T
```

6.9. Использование рекурсии

Рекурсия имеет большее значение в Лиспе, чем в других языках. Этому есть три основные причины:

[1] В Лиспах, существовавших до ANSI Common Lisp, первый аргумент `compile` не мог быть уже скомпилированной функцией.

1. Функциональное программирование. Рекурсивные алгоритмы пользуются побочными эффектами в существенно меньшей степени.
2. Рекурсивные структуры данных. Неявное использование указателей в Лиспе облегчает рекурсивное создание структур данных. Наиболее общей структурой такого типа является список: либо `nil`, либо `cons`, чей `cdr` – также список.
3. Элегантность. Лисп-программисты придают огромное значение красоте их программ, а рекурсивные алгоритмы часто выглядят намного элегантнее их итеративных аналогов.

Новички часто находят рекурсию поначалу сложной для понимания. Но в разделе 3.9 было показано, что для проверки корректности рекурсивной функции вам вовсе необязательно представлять всю последовательность вызовов.

Это же утверждение верно и в случае, когда вам необходимо написать свою рекурсивную функцию. Если вы сможете сформулировать рекурсивное решение проблемы, то вам не составит труда перевести это решение в код. Чтобы решить задачу с помощью рекурсии, вам необходимо сделать следующее:

1. Понять, как можно решить ее с помощью разделения на конечное число похожих, но более мелких подзадач.
2. Понять, как решить такую подзадачу с помощью конечного набора операторов.

Если вы в состоянии сделать это, значит, вы готовы к написанию рекурсивной функции, потому что вы знаете, как решить конечное число сходных задач за конечное количество шагов.

Например, в предложенном ниже рекурсивном алгоритме нахождения длины списка мы на каждом шаге рекурсии находим длину уменьшенного списка:

1. В общем случае, длина списка равна длине его `cdr`, увеличенной на `1`.
2. Длину пустого списка принимаем равной `0`.

Когда это определение переводится в код, сначала необходимо определить базовый случай. Однако, на этапе формализации задачи сначала рассматривается наиболее общий случай.

Рассмотренный выше алгоритм описывает нахождение длины нормального списка. Определяя рекурсивную функцию, вы должны быть уверены, что разделение задачи действительно приводит к подзадачам меньшего размера. Переход к `cdr` списка приводит к меньшей задаче, однако лишь для списка, не являющегося циклическим.

Сейчас мы приведем еще два примера рекурсивных функций. Опять же, они подразумевают конечный размер аргументов. Обратите внимание, что вторая на каждом шаге рекурсии разбивает задачу на *две* подзадачи.

`member` Объект содержится в списке если он является его первым элементом или содержится в `cdr` этого списка. В пустом списке не содержится ничего.

`copy-tree` Копия дерева, представленного как cons-ячейка, - это ячейка, построенная из `copy-tree` для `car` исходной ячейки и `copy-tree` для ее `cdr`. Для атома `copy-tree` является самим атомом.

Сумев формализовать рекурсивный алгоритм таким образом, вы легко сможете написать соответствующее рекурсивное определение вашей функции.

Некоторые алгоритмы естественным образом ложатся на такие определения, но не все. Вам придется согнуться в три погибели, чтобы определить `our-copy-tree` (см. стр. Ё021out) без использования рекурсии. С другой стороны, итеративный вариант `show-squares` (см. стр. Ё017out) более доступен для понимания, нежели его рекурсивный аналог (см. стр. Ё022out). Часто оптимальный выбор остается неясен до тех пор, пока вы не приступите к написанию кода.

Если производительность функции имеет для вас существенное значение, вам следует учитывать два момента. Один из них – хвостовая рекурсия, которая будет обсуждаться в разделе 13.2. Хороший компилятор выполняет хвостовую рекурсию так же быстро, как аналогичный цикл.[1] Правда, если вам приходится модифицировать саму функцию, чтобы она удовлетворяла условию хвостовой

[1] В действительности, хвостовая рекурсия просто преобразуется в соответствующий цикл. Такая оптимизация хвостовой рекурсии входит в стандарт языка Scheme, но отсутствует в Common Lisp. Тем не менее, многие компиляторы Common Lisp поддерживают такую оптимизацию. – *Прим. перев.*

рекурсивности, то иногда может оказаться проще сразу переделать ее в итеративную

Кроме того, вам необходимо помнить, что рекурсивный по сути алгоритм не всегда эффективен сам по себе. Классический пример – функция Фибоначчи. Эта функция рекурсивна по определению:

1. `Fib(0) = Fib(1) = 1.`
2. `Fib(n) = Fib(n-1) + Fib(n-2).`

При этом дословная трансляция данного определения в код:

```
(defun fib (n)
  (if (<= n 1)
      1
      (+ (fib (- n 1))
        (fib (- n 2)))))
```

дает совершенно неэффективную функцию. Дело в том, что такая функция вычисляет одни и те же вещи по несколько раз. Например, вычисляя `(fib 10)`, она вызовет `(fib 9)` и `(fib 8)`. Однако, вычисление `(fib 9)` уже включает в себя `(fib 8)`, и она выполняет это вычисление заново.

Ниже приведена аналогичная итеративная функция:

```
(defun fib (n)
  (do ((i  n (- i 1))
       (f1 1 (+ f1 f2))
       (f2 1 f1))
      ((<= i 1) f1)))
```

Итеративная версия куда менее понятна, зато намного более эффективна. Насколько часто такие различия имеют место? Очень редко. Именно поэтому в различных книгах приводится один и тот же пример. Тем не менее, таких ситуаций следует остерегаться.

Итоги главы

1. Именованная функция хранится как `symbol-function` соответствующего символа. Макрос `defun` скрывает подобные детали. Кроме того, он позволяет сопровождать функции документацией, а также настраивать поведение `setf`.
2. Имеется возможность определять локальные функции, этот процесс напоминает определение локальных переменных.
3. Функции могут иметь необязательные, остаточные и ключевые параметры.
4. Утилиты дополняют язык, являясь уменьшенным примером программирования снизу-вверх.
5. Лексические переменные существуют до тех пор, пока на них ссылается какой-либо объект. Замыкания – это функции, ссылающиеся на свободные переменные. Вы можете самостоятельно определять функции, возвращающие замыкания.
6. Dylan предоставляет функции для построения других функций. С использованием замыканий мы можем реализовать их в Common Lisp.
7. Специальные переменные имеют динамическое окружение.
8. Функции могут компилироваться индивидуально или (обычно) в составе файла.
9. Рекурсивный алгоритм решает задачу разделением ее на конечное число похожих подзадач меньшего размера.

Упражнения

1. Определите версию `tokens` (см. стр. Ё023out), которая использует ключи `:test` и `:start`, по умолчанию равные `#'constituent` и `0`.
2. Определите версию `bin-search` (см. стр. Ё024out), использующую ключи `:key`, `:test`, `:start` и `:end`, имеющие обычные для них значения по умолчанию.

3. Определите функцию, принимающую любое количество аргументов и возвращающую их количество.
4. Измените функцию `most` (см. стр. Ё025out), чтобы она возвращала два значение – два элемента, имеющих наибольший рейтинг.
5. Определите `remove-if` (без ключевых параметров) с помощью `filter` (см. стр. Ё026out).
6. Определите функцию, принимающую одно число и возвращающую наибольшее число из всех ранее полученных ею.
7. Определите функцию, принимающую одно число и возвращающую его, если оно больше числа, переданного этой функции на предыдущем вызове. Во время первого вызова она должна возвращать `nil`.
8. Функция `expensive` имеет один аргумент, целое число между 0 и 100 включительно. Она производит сложные и дорогостоящие вычисления. Опередите функцию `frugal`, возвращающую тот же ответ, что и `expensive`, но вызывающую `expensive` лишь когда значение для заданного числа выполняется впервые.
9. Определите функцию наподобие `apply`, которая печатает любой численный результат в восьмеричном формате `(base 8)`.

7. Ввод и вывод

Common Lisp имеет богатые возможности для осуществления ввода/вывода. Для ввода, наряду с обычными возможностями для чтения символов, у нас есть `read`, который включает полноценный Лисп-парсер. Для вывода, наряду с привычными средствами для записи символов, мы получаем `format`, который сам по себе небольшой язык. В этой главе вводятся все основные концепции ввода/вывода.

Существует два вида потоков: символьные и бинарные. В этой главе рассмотрены лишь символьные потоки, бинарные будут описаны в разделе 14.2.

7.1. Потоки

Потоки – это лисповые объекты, представляющие источники и/или направления передачи символов. Чтобы прочитать или записать файл, необходимо открыть соответствующий поток. Однако, поток – не то же самое, что и файл. Когда вы читаете или печатаете что-либо в toplevel, вы также работаете с потоками. Создавая потоки, вы можете читать из строк или отправлять что-либо в них.

Для ввода по умолчанию используется поток `*standard-input*`, для вывода - `*standard-output*`. Первоначально они ссылаются на один и тот же поток, соответствующий toplevel.

С функциями `read` и `format` мы уже знакомы. Ранее мы пользовались ими для чтения и печати чего-либо в toplevel. Функция `read` имеет необязательный аргумент, который определяет входной поток и по умолчанию установлен в `*standard-input*`. Первый аргумент `format` также может быть потоком. Ранее мы вызывали `format` с первым аргументом `t`, где `t` означает умолчальный поток, то есть `*standard-output*`. Таким образом, до этого момента мы наблюдали лишь поведение этих функций по умолчанию. Однако те же операции мы можем совершать и с любыми другими потоками.

Путь (*pathname*) – это переносимый способ определения пути к файлу. Путь имеет шесть компонентов: хост, устройство, директорию, тип и версию. Собственные пути можно создавать с помощью `make-pathname` с одним и более ключевыми параметрами. В простейшем случае мы можем обозначить лишь имя, сохранив умолчальные значения остальных параметров:

```
> (setf path (make-pathname :name "myfile"))
#P"myfile"
```

Основная функция для открытия файлов – `open`. Ей необходимо сообщить путь[1], а ее поведением можно управлять с помощью многочисленных ключевых параметров. В случае успеха она возвращает поток, связанный с файлом.

Вам нужно сообщить, как вы собираетесь использовать создаваемый поток. Параметр `:direction` определяет, будет ли производиться чтение (`:input`), запись (`:output`) или то и другое одновременно (`:io`). Параметр `:if-exists` для потока, используемого для чтения, определяет поведение в случае, если файл уже существует. Обычно используется `:supersede`[2]. Итак, чтобы создать поток, через который мы сможем записать что-либо в файл `"myfile"`, напечатаем:

```
> (setf str (open path :direction :output
                       :if-exists :supersede))
#<Stream C017E6>
```

Способ отображения потоков при печати зависит от используемой реализации Common Lisp.

Теперь мы можем сообщить поток `str` первым аргументом функции `format`, и она будет печатать свой вывод в поток `str`, а не в toplevel:

---

[1] Вместо полноценного пути вы можете использовать обычные строки, однако это решение не переносимо.

[2] Запись поверх существующего файла. – *Прим. перев.*

```
> (format str "Something~%")
NIL
```

Если мы теперь посмотрим на содержимое файла, то либо его обнаружим, либо нет. Некоторые реализации осуществляют вывод порциями, и ожидаемое содержимое файла может появиться лишь после закрытия потока:

```
> (close str)
NIL
```

Всегда заботьтесь о закрытии файлов по окончании их использования. Пока вы не сделаете этого, вы не можете быть уверены относительно его содержимого. Если мы теперь взглянем на содержимое файла `"myfile"`, мы обнаружим строчку:

```
Something
```

Если мы хотим всего лишь прочитать содержимое файла, мы сообщим `:input` вместе с параметром `:direction`:

```
> (setf str (open path :direction :input))
#<Stream C01C86>
```

Мы можем читать содержимое файла с помощью любой функции чтения. В разделе 7.2 ввод описан более подробно. Здесь, в качестве примера, мы воспользуемся функцией `read-line` для чтения строки из файла:

```
> (read-line str)
"Something"
NIL
> (close str)
NIL
```

Не забудьте закрыть файл после завершения чтения.

Функции `open` и `close` практически никогда не используются явно. Гораздо удобнее использовать макрос `with-open-file`. Первый его аргумент – список, содержащий некоторое имя переменной и все те же аргументы, которые вы могли бы передать функции `open`. Далее следуют выражения, которые могут использовать заданную выше переменную, которая внутри тела макроса связана с именем переменной. После выполнения всех выражений тела макроса поток закрывается автоматически. Таким образом, операция записи в файл может быть целиком выражена так:

```
(with-open-file (str path :direction :output
                          :if-exists :supersede)
  (format str "Something~%"))
```

Макрос `with-open-file` использует `unwind-protect`Ё018in, чтобы гарантировать, что файл будет действительно закрыт, даже если выполнение выражений внутри тела макроса будет аварийно прервано.

7.2. Ввод

Двумя наиболее популярными функциями чтения считаются `read-line` и `read`. Первая читает все символы до начала новой строки, возвращая их в виде строки. Один ее необязательный параметр, как и для `read`, определяет входной поток. Если он не задан явно, то используется `*standard-input*`:

```
> (progn
    (format t "Please enter your name: ")
    (read-line))
Please enter your name: Rodrigo de Bivar
"Rodrigo de Bivar"
NIL
```

Второй аргумент будет `t` лишь тогда, когда `read-line` закончит чтение файла прежде, чем встретит конец строки.

В общем случае `read-line` может получать четыре аргумента: поток; информацию о том, сигнализировать ли ошибку по достижении конца файла (`eof`);

что возвращать, если предыдущий аргумент `nil`; четвертый аргумент (см. стр. Ё027out) обычно можно игнорировать.

Итак, отобразить содержимое файла в toplevel можно с помощью следующей функции:

```
(defun pseudo-cat (file)
  (with-open-file (str file :direction :input)
    (do ((line (read-line str nil 'eof)
               (read-line str nil 'eof)))
        ((eql line 'eof))
      (format t "~A~%" line))))
```

Если вы хотите, чтобы ввод обрабатывался так же, как лисповые выражения, используйте `read`. Эта функция считывает ровно одно выражение. Это значит, что она может считывать более или менее, чем строку. Разумеется, считываемый объект должен быть валидным с точки зрения синтаксиса Лиспа.

Воспользовавшись `read` в toplevel, мы обнаружим, что считываемый объект действительно может состоять из нескольких строк:

```
> (read)
(a
b
c)
(A B C)
```

С другой стороны, если на одной строке будет находиться несколько лисповых объектов, `read` прекратит считывание знаков, закончив чтение первого объекта, оставляя другие для следующий вызовов `read`. Проверим это предположение с помощью `ask-number` (см. стр. Ё028out). Введем одновременно несколько выражений и посмотрим, что происходит:

```
> (ask-number)
Please enter a number. a b
Please enter a number. Please enter a number. 43
43
```

Два приглашения напечатаны одно за другим на одной строке. Первый вызов `read` возвращает `a`, и это не число, поэтому функция заново предлагает ввести число. Однако следующий вызов `read` сразу же получает объект `b`, который также не число. Поэтому приглашение выводится еще раз.

Чтобы избегать подобных случаев, может быть полезно не использовать `read` напрямую. Предыдущая функция будет более безопасной, если пользовательский ввод будет читаться с помощью `read-line` и далее обрабатываться с помощью `read-from-string`.°Ё089in Эта функция принимает строку и возвращает первый объект, прочитанный из нее.

```
> (read-from-string "a b c")
A
2
```

Помимо прочитанного объекта она возвращает позицию в строке, на которой завершилось чтение.

В общем случае `read-from-string` может принимать два необязательных и три ключевых параметра. Два необязательных – те же, что и третий и четвертый в `read-line`. Ключевые параметры `:start` и `:end` ограничивают диапазон строки, в котором будет выполняться чтение.

Рассмотренные в этом разделе функции определены с помощью более примитивной `read-char`, считывающей одиночный символ. Она принимает те же четыре необязательных аргумента, что и `read` и `read-line`. Common Lisp также определяет функцию `peek-char`, которая похожа на `read-char`, но не удаляет прочитанный символ из потока.

7.3. Вывод

Три простейшие функции вывода – `prin1`, `princ` и `terpri`. Всем им можно сообщить выходной поток, по умолчанию это `*standard-output*`.

Разница между `prin1` и `princ`, грубо говоря, в том, что `prin1` генерирует вывод для программ, а `princ` – для людей. Так, например, `prin1` печатает двойные кавычки вокруг строк, а `princ` – нет:

```
> (prin1 "Hello")
"Hello"
"Hello"
> (princ "Hello")
Hello
"Hello"
```

Все они возвращают напечатанный аргумент, который, кстати, совпадает с печатаемым `prin1`. Функция `terpri` печатает новую строку.

Устройство этих функций полезно понимать, чтобы объяснять поведение `format`. С помощью функции `format` может быть осуществлен практически любой вывод. Она принимает поток (а также `t` или `nil`), управляющую строку и ноль или более аргументов. Управляющая строка может содержать директивы форматирования, которые начинаются со знака `~` (тильда). Некоторые директивы указывают на местоположение последующих аргументов, переданных `format`.

Передавая `t`, мы направляем вывод в `*standard-output*`. Если первый аргумент `nil`, то `format` возвратит строку вместо того, чтобы ее напечатать. Ради краткости будем приводить только такие примеры. В зависимости от угла рассмотрения, `format` или невероятно мощный, либо жутко сложный. Он понимает огромное количество директив, лишь немногие из которых используются на практике. Две наиболее распространенные директивы - `~A` и `~%`. (Между `~a` и `~A` нет различия, однако принято использовать вторую директиву, потому что в управляющей строке она более заметна.) `~A` указывает на положение вставки значения, которое будет напечатано с помощью `princ`. `~%` соответствует новой строке.

```
> (format nil "Dear ~A,~% Our records indicate…"
          "Mr. Malatesta")
"Dear Mr. Malatesta,
  Our records indicate…"
```

В этом примере `format` возвращает один аргумент – строку, содержащую символ переноса строки.

Директива `~S` похожа на `~A`, однако печатает свой объект так же, как `prin1`, а не как `princ`:

```
> (format t "~S ~A" "z" "z")
"z" z
NIL
```

Директивы форматирования сами могут принимать аргументы. `~F`, используемая для печати выровненных справа чисел с плавающей запятой[1], может принимать пять аргументов:

1. Суммарное количество выводимых символов. По умолчанию это длина числа.
2. Количество выводимых символов после точки. По умолчанию выводятся все.
3. Смещение точки влево (смещение на один знак соответствует умножению на 10). По умолчанию отсутствует.
4. Символ, который будет выведен вместо числа, если оно не умещается в количество символов, разрешенное в первом аргументе. Если ничего не задано, то число, превышающее допустимый лимит, будет напечатано как есть.
5. Символ, печатаемый перед левой цифрой. По умолчанию пробел.

[1] В англоязычной литературе в принято использовать точку в качестве разделителя в десятичных дробях. В русском языке принято использовать запятую, а такие числа называть «числами с плавающей запятой» вместо «floating point numbers». В Common Lisp для записи десятичных дробей всегда используется точка. – *Прим. перев.*

Вот пример, в котором используются все пять аргументов:

```
> (format nil "~10,2,0,'*,' F" 26.21875)
"     26.22"
```

Исходное число округляетсяЁ033in до двух знаков после точки, (сама точка смещается на 0 положений влево, то есть не смещается), для печати числа выделяется пространство 10 символов, на месте незаполненных слева полей печатаются пробелы. Обратите внимание, что символ звездочки передается как `'*`, а не `#\*`. Так как заданное число вписывается в предложенное пространство 10 символов, четвертый аргумент не используется.

Все эти аргументы не обязательны для использования. Чтобы использовать умолчание, достаточно просто пропустить соответствующий аргумент. Если вы желаете напечатать число, округленное до двух знаков после точки, достаточно сказать:

```
> (format nil "~,2,,,F" 26.21875)
"26.22"
```

Такая последовательность запятых может быть опущена, и более распространена такая запись:

```
> (format nil "~,2F" 26.21875)
"26.22"
```

Предупреждение: Округляя числа, `format` не гарантирует округление в большую или меньшую сторону. Поэтому `(format nil "~,1F" 1.25)` может привести либо к `"1.2"`, либо к `"1.3"`. Таким образом, если вам нужна гарантия округления лишь в одну сторону (например, при конвертации валют), округляйте выводимое число явным образом.

7.4. Пример: замена строк

В этом разделе приведен пример использования ввода/вывода – простая программа для замены строк в текстовых файлах. Мы создадим функцию, которая сможет заменить каждую строку `old` в файле на строку `new`. Простейший способ сделать это – сверять каждый символ с первым символом `old`. Если они не совпадают, то мы можем просто напечатать символ из файла. Если они совпадают, то мы сверяем следующие символы из файла и искомой строки, соответственно. И так далее. Если найдено совпадение, то печатаем на выход строку `new`.°Ё090in

Что происходит, когда мы натыкаемся на несовпадение в процессе сверки символов? Например, предположим, что ищем шаблон `"abac"`, а входной файл содержит `"ababac"`. Совпадение будет наблюдаться вплоть до четвертого символа: в файле это `b`, а в шаблоне `c`. Дойдя до четвертого символа, мы понимаем, что можем напечатать первый символ `a`. Однако, некоторые пройденные символы нам все еще нужны, потому что третий прочтенный символ `a` совпадает с первым символом шаблона. Итак, нам понадобится место, где мы будем хранить все прочитанные символы, которые нам пока еще нужны.

Очередь для временного хранения входной информации называется *буфером*. В данном случае необходимый размер буфера нам заранее неизвестен, и мы воспользуемся такой структурой данных какЁ032in *кольцевой буфер*. Кольцевой буфер строится на основе вектора. Этот вектор последовательно заполняется вновь поступающими символами. Когда вектор заполняется полностью, процесс начинается с начала поверх уже существующих элементов. Если нам заранее известно, что не понадобится хранить более n элементов, вектора длиной n будет достаточно, то перезапись элементов с начала не приведет к потере данных.

На рис. 7.1 показан код, реализующий операции с кольцевым буфером. Структура `buf` имеет пять полей: вектор для хранения объектов и четыре индекса. Два из них, `start` и `end`, необходимы для любых операций с кольцевым буфером: `start` указывает на первое значение буфера и будет увеличиваться при взятии элемента из буфера; `end` указывает на последнее значение в буфере и будет увеличиваться при добавлении нового элемента.

```
(defstruct buf
  vec (start -1) (used -1) (new -1) (end -1))

(defun bref (buf n)
  (svref (buf-vec buf)
         (mod n (length (buf-vec buf)))))

(defun (setf bref) (val buf n)
  (setf (svref (buf-vec buf)
               (mod n (length (buf-vec buf))))
        val))

(defun new-buf (len)
  (make-buf :vec (make-array len)))

(defun buf-insert (x b)
  (setf (bref b (incf (buf-end b))) x))

(defun buf-pop (b)
  (prog1
    (bref b (incf (buf-start b)))
    (setf (buf-used b) (buf-start b)
          (buf-new  b) (buf-end   b))))

(defun buf-next (b)
  (when (< (buf-used b) (buf-new b))
    (bref b (incf (buf-used b)))))

(defun buf-reset (b)
  (setf (buf-used b) (buf-start b)
        (buf-new  b) (buf-end   b)))

(defun buf-clear (b)
  (setf (buf-start b) -1 (buf-used  b) -1
        (buf-new   b) -1 (buf-end   b) -1))

(defun buf-flush (b str)
  (do ((i (1+ (buf-used b)) (1+ i)))
      ((> i (buf-end b)))
    (princ (bref b i) str)))
```

Рис. 7.1. Операции с кольцевым буфером.

Два других индекса, `used` и `new`, потребуются для использования буфера в нашем приложении. Они могут принимать значение между `start` и `end`, причем будет всегда соблюдаться следующее соотношение:

```
start ≤ used ≤ new ≤ end
```

Пару `used` и `new` можно считать аналогом `start` и `end` для текущего совпадения. Когда мы начинаем отслеживать совпадение, `used` будет равен `start`, а `new` - `end`. Для каждого совпадения пары символов `used` будет увеличиваться. Когда `used` достигнет `new`, из буфера считываются все элементы, занесенные туда с начала проверки данного совпадения. Нам не нужны знаки, находившиеся в буфере до нахождения текущего совпадения. Поэтому требуется другой индекс, `new`, который исходно равен `end` и не увеличивается при добавлении новых знаков во время проверки совпадения.

Функция `bref` возвращает элемент, хранящийся в заданном положении буфера. Используя остаток от деления заданного индекса на длину вектора, мы можем предполагать, что наш буфер имеет достаточно большой произвольный размер. Вызов `(new-buf n)` создает новый буфер, способный удерживать до `n` элементов.

Чтобы поместить в буфер новое значение, будем пользоваться `buf-insert`. Эта функция увеличивает индекс `end` и кладет на это место заданный элемент. Обратной функцией является `buf-pop`, которая забирает первый элемент буфера и увеличивает индекс `start`. Эти две функции совершенно необходимы для использования кольцевого буфера.

Следующие две функции написаны специально для нашего приложения: `buf-next` читает значение из буфера без его извлечения, `buf-reset` сбрасывает `used` и `new` до исходных значений, `start` и `end`. Если все значения до `new` прочитаны, `buf-next` возвратит `nil`. Хотя это и не соответствует реальному значение, это не является проблемой, потому что буфер нужен нам лишь для хранения символов.

Наконец, `buf-flush` выводит содержимое буфера, записывая все доступные элементы в заданный поток, а `buf-clear` очищает буфер, устанавливая все индексы в `-1`.

Эти функции используются в коде на рис. 7.2, предоставляющем средства для замены строк. Функция `file-subst` использует четыре аргумента: искомую строку, ее замену, входной и выходной файлы. Она создает потоки, связанные с заданными файлами и вызывает функцию `stream-subst`, которая и выполняет основную работу.

```
(defun file-subst (old new file1 file2)
  (with-open-file (in file1 :direction :input)
     (with-open-file (out file2 :direction :output
                                :if-exists :supersede)
       (stream-subst old new in out))))

(defun stream-subst (old new in out)
  (let* ((pos 0)
         (len (length old))
         (buf (new-buf len))
         (from-buf nil))
    (do ((c (read-char in nil :eof)
            (or (setf from-buf (buf-next buf))
                (read-char in nil :eof))))
        ((eql c :eof))
      (cond ((char= c (char old pos))
             (incf pos)
             (cond ((= pos len)            ; 3
                    (princ new out)
                    (setf pos 0)
                    (buf-clear buf))
                   ((not from-buf)         ; 2
                    (buf-insert c buf))))
            ((zerop pos)                   ; 1
             (princ c out)
             (when from-buf
               (buf-pop buf)
               (buf-reset buf)))
            (t                             ; 4
             (unless from-buf
               (buf-insert c buf))
             (princ (buf-pop buf) out)
             (buf-reset buf)
             (setf pos 0))))
    (buf-flush buf out)))
```

Рис. 7.2. Замена строк.

Функция `stream-subst` использует алгоритм, набросок которого был описан в начале раздела. Каждый раз она читает из входного потока один символ. Если он не совпадает с первым элементом искомой строки, он тут же записывается в выходной поток (1). Когда начинается совпадение, символы пересылаются в буфер `buf` (2).

Переменная `pos` указывает на положение символа в искомой строке. Если оно равно длине самой строки, это означает, что совпадение найдено. В таком случае в выходной поток записывается ее замена, а буфер очищается (3). Если в какой-либо момент последовательность символов перестает соответствовать искомой строке, то мы вправе забрать символ из буфера и записать в выходной поток, установить `pos` равным нулю (4), а сам буфер очистить.

Следующая таблица наглядно демонстрирует, что происходит при замене `"baro"` на `"baric"` в файле, содержащем только одно слово `barbarous`:

Символ	Источник	Совпадение	Действие	Вывод	Буфер
b	файл	b	2		b
a	файл	a	2		b a
r	файл	r	2		b a r
b	файл	o	4	b	b.a r b.
a	буфер	b	1	a	a.r b.
r	буфер	b	1	r	r.b.
b	буфер	b	1		r b:
a	файл	a	2		r b:a
r	файл	r	2		r b:a r
o	файл	o	3	baric	
u	файл	b	1	u	
s	файл	b	1	s	

Первая колонка содержит текущий знак – значение `c`; вторая показывает, откуда он был считан – из буфера или прямиком из файла; третья показывает знак, обеспечивающий совпадение – элемент `pos`; четвертая указывает на действие, совершаемое в данном случае; пятая колонка соответствует текущему содержимому буфера после выполнения операции. В последней колонке точками показаны также позиции `used` и `new`. Если они совпадают, то используется двоеточие.

Предположим, что у нас есть файл `"test1"`, содержащий следующий текст:

```
The struggle between Liberty and Authority is the most
conspicuous feature in the portions of history with which we are
earliest familiar, particularly in that of Greece, Rome, and
England.
```

После выполнения `(file-subst " th" " z" "test1" "test2")` файл `"test2"` будет иметь следующий вид:

```
The struggle between Liberty and Authority is ze most
conspicuous feature in ze portions of history with which we are
earliest familiar, particularly in zat of Greece, Rome, and
England.
```

С целью максимального упрощения примера, код на рис. 7.2. просто заменяет одну строку на другую. Однако несложно соорудить поиск по полноценным шаблонам вместо последовательностей букв. Все, что вам потребуется – заменить вызов `char=` на функцию проверки на соответствие шаблону.

7.5. Макросимволы

Макросимволы (*macro character*) – это символы, имеющие особое значение для `read`. Символы `a` и `b` обычно обрабатываются как есть, однако, символ открывающей круглой скобки распознается как начало считывания списка.

Макросимвол или их комбинация известны также как *макросы чтения* (*read-macro*). Многие макросы чтения в Common Lisp на деле являются сокращениями. Например, кавычка. Цитируемое выражение, например, `'a`, при обработке с помощью `read` раскрывается в список `(quote a)`. Набирая подобное выражение в toplevel, оно вычисляется сразу же после прочтения, и вы никогда не увидите этого преобразования. Его можно обнаружить, вызывая `read` явно:

```
> (car (read-from-string "'a"))
QUOTE
```

Макрос чтения, соответствующей `quote`, состоит лишь из одного знака, что является редкостью. Имея ограниченный набор символов, сложно иметь много односимвольных макросов чтения. Большинство макросов чтения состоит из двух или более знаков.

Такие макросы чтения называются *управляемыми* (*dispatching*), и управление их поведением осуществляется с помощью управляющего символа. У всех стандартных макросов чтения управляющим символом является знак решетки, `#`. С некоторыми из них мы уже знакомы. Например, сокращение `#'` соответствует `(function ...)`, так же как `(quote ...)` для `'`.

Некоторые другие – `#(...)` для векторов, `#na(...)` для массивов, `#\` для знаков, `#S` для структур. При выводе на печать через `prin1` такие объекты отобразятся в виде соответствующих макросов чтения.[1] Это означает, что такие объекты могут быть записаны, а затем считаны обратно в таком же виде:

```
> (let ((*print-array* t))
    (vectorp (read-from-string (format nil "~S"
                                       (vector 1 2)))))
T
```

Разумеется, на выходе получается не тот же самый вектор, а другой, состоящий из тех же элементов.

Не все объекты отображаются в соответствии с таблицей чтения `readtable`. Функции и хеш-таблицы, например, отображаются через `#<...>`. Фактически, `#<` - тоже макрос чтения, однако при передаче в `read` он всегда вызывает ошибку. Функции и хеш-таблицы не могут быть напечатаны а затем прочитаны обратно, и этот макрос чтения гарантирует, что пользователи не будут питать иллюзий на этот счет[2].

При определении собственного представления для какого-либо объекта (например, для отображения структур) важно помнить этот принцип. Или объект в вашем представлении может быть прочитан обратно, либо используйте `#<...>`.

Итоги главы

1. Потоки – источники ввода и направления вывода. В символьных потоках ввод и вывод состоят из символов. Поток, используемый по умолчанию, соответствует toplevel. При открытии файлов создаются новые потоки.
2. Ввод может обрабатываться как набор объектов, строка знаков или отдельные знаки.
3. Функция `format` предоставляет полное управление выводом.
4. Чтобы осуществить замену строк в текстовом файле, вам придется считывать символы в буфер.
5. Когда `read` встречает макросимвол типа `'`, он вызывает связанную с ним функцию.

Упражнения

1. Определите функцию, возвращающую список из строк, прочитанных из заданного файла.
2. Определите функцию, возвращающую список выражений, содержащихся в заданном файле.
3. Пусть в файле некоторого формата комментарии помечаются символом `%`. Содержимое строки от начала символа комментария до ее конца игнорируется. Определите функцию, принимающую два имени файла и записывающую во второй содержимое первого с вырезанными комментариями.
4. Определите функцию, принимающую двумерный массив чисел с плавающей точкой и отображающую его в виде аккуратных колонок. Каждый элемент должен печататься с двумя знаками после запятой на пространстве 10 знаков. (Исходите из предположения, что все они уместятся в отведенном пространстве.) Вам потребуется функция `array-dimensions` (см. стр. Ё029out).
5. Измените функцию `stream-subst`, чтобы она понимала шаблоны с групповым символом `+`. Если знак `+` появляется в строке `old`, он может соответствовать любому

[1] Чтобы векторы и массивы печатались таким образом, необходимо установить значение `*print-array*`, равное `t`.

[2] Решетка с кавычкой не могут использоваться для описания функций, так как этот макросимвол не в состоянии представлять замыкание.

знаку.

6. Измените функцию `stream-subst` так, чтобы шаблон мог содержать элемент, соответствующий: любой цифре, любой цифре и букве, любому символу. Шаблон должен уметь распознавать любые читаемые символы. (Подсказка: `old` теперь не обязательно должен быть строкой.)

8. Символы

С Лисп-символами вы уже немного познакомились. Они не бросаются в глаза, но о них есть что рассказать. Поначалу разумно было не трогать механизм их реализации. Вы можете по-прежнему использовать как объекты или имена, не задумываясь о том, каким образом они связаны. Но именно здесь полезно остановиться и разобраться с тем, что же происходит. В этой главе символы обсуждаются подробно.

8.1. Имена символов

В главе 2 символы описывались как имена переменных, существующие сами по себе. Однако, область применения символов в Лиспе шире, чем область применения переменных в большинстве других языков. Имя символа может быть представлено в виде строки. Получить имя символа возможно с помощью `symbol-name`:

```
> (symbol-name 'abc)
"ABC"
```

Обратите внимание, что имена символов записываются в верхнем регистре. По умолчанию Common Lisp преобразует все буквенные имена при чтении к верхнему регистру. Это означает, что по умолчанию Common Lisp не чувствителен к регистру:

```
> (eql 'aBc 'Abc)
T
> (CaR '(a b c))
A
```

Для работы с символами, названия которых имеют пробелы или другие знаки, не являющиеся буквами, используется особый синтаксис. Любая последовательность знаков между вертикальными черточками, считается символом. Это означает, что вы можете использовать любые знаки в именах символов:

```
> (list '|Lisp 1.5| '|| '|abc| '|ABC|)
(|Lisp 1.5| || |abc| ABC)
```

При считывании такого символа не производится преобразование к верхнему регистру, а макросимволы считаются обычными знаками.

Итак, какие символы могут быть созданы без знаков вертикальной черты? По сути, такие, которые не содержат в названии знаков, имеющих для `read` специального значения. Чтобы узнать, может ли символ существовать без такого обрамления, достаточно просто посмотреть, как Лисп его напечатает. Если так, то при печати символ не будет обрамлен вертикальными черточками.

Следует помнить, что вертикальные черточки – специальный синтаксис, а не часть имени символа:

```
> (symbol-name '|a b c|)
"a b c"
```

(Если вы желаете включить черточку в имя символа, расположите перед ней знак `"\"`.)

8.2. Списки свойств.

В Common Lisp каждый список имеет собственный *список свойств* (*property-list*, *plist*). Функция `get` принимает символ и ключ, возвращая связанное с этим ключом значение из списка свойств:

```
> (get 'alizarin 'color)
NIL
```

Для сопоставления ключей используется `eql`. Если указанное свойство не задано, `get` возвращает `nil`.

Связать значение с ключом возможно с помощью `setf`:

```
> (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 являются ассоциативными, хотя и работают сходным образом.

В Common Lisp списки свойств используются довольно редко. Они в значительной мере вытеснены хеш-таблицами.

8.3. А символы-то не маленькие

Символы создаются неявным образом, когда мы печатаем их имена, а при печати самих символов мы опять же видим лишь имена. При таких обстоятельствах легко подумать, что символ – лишь имя и ничего более. Однако, многое скрыто от наших глаз.

Из того, как мы видим и используем символы, может показаться, что это маленькие объекты, такие же, как, скажем, целые числа. На деле символы обладают внушительными размерами и более походят на структуры, определяемые через `defstruct`. Для символа могут быть определены имя, пакет, значение связанной с ним переменной, значение связанной функции и список свойств. Устройство символа и взаимосвязь между его компонентами показана на рис. 8.1.

Рис. 8.1. Структура символа.

Мало кто использует настолько большое количество символов, что вставал бы вопрос об экономии памяти. Однако, стоит держать в уме, что символы – полноценные объекты, а не просто имена. Две переменные могут ссылаться на один символ, как и на один список: в таком случае они имеют указатель на общий объект.

8.4. Создание символов

В разделе 8.1 было показано, как получать имена символов. Есть возможность и обратного действия, преобразования строки в символ. Это более сложная задача, потому что для ее выполнения необходимо иметь представление о пакетах.

Логически, пакеты – это таблицы, отображающие имена в символы. Любой символ принадлежит конкретному пакету. Символ, принадлежащий пакету, называют *интернированным* в него. Пакеты делают возможной модульность, ограничивая область видимости символов. Имена функций и переменных считаются символами, поэтому они могут быть доступны в соответствующих пакетах.

Большинство символов интернируются во время считывания. Когда вы впервые вводите символ, Лисп создает новый символьный объект и интернирует его в текущий пакет (по умолчанию это `common-lisp-user`). Однако вы можете сделать то же самое вручную с помощью `intern`:

```
> (intern "RANDOM-SYMBOL")
RANDOM-SYMBOL
NIL
```

Функция `intern` принимает также дополнительный аргумент – пакет, по умолчанию используется текущий. Приведенный выше вызов создает в текущем пакете символ с именем `"RANDOM-SYMBOL"`, если этот символ пока еще не имеется

в данном пакете. Второй аргумент возвращает истину, когда такой символ уже существует.

Не все символы являются интернированными. Иногда может оказаться полезным использование неинтернированных символов, также как иногда бывает полезным записывать номера телефонов не в записную книжку, а на отдельный клочок бумаги. Такие символы создаются через `gensym`. Мы познакомимся с ними при разборе макросов в главе 10.

8.5. Использование нескольких пакетов

Большие программы часто разделяют на несколько пакетов. Каждая часть программы расположена в собственном пакете, и разработчик другой ее части может смело использовать имеющиеся в первой имена функций и переменных.

При использовании языков, не поддерживающих разделение пространства имен, программистам, работающим над серьезными проектами, приходится вводить договоренности по поводу используемых имен. Например, разработчик подсистемы отображения, вероятно, будет использовать имена, начинающиеся с `disp`, программист, разрабатывающий математические процедуры, будет использовать имена, начинающиеся с `math_`. Так, например, функция, выполняющая быстрое преобразование Фурье, вероятно, будет носить имя `math_fft`.

Пакеты выполняют эту работу самостоятельно. Внутри отдельного пакета вы вольны использовать любые имена. Только те символы, которые будут явным образом *экспортированы*, будут видимы в других пакетах, где будут доступны с помощью префикса, соответствующего имени содержащего их пакета.

Пусть, например, имеется программа, разделенная на два пакета, `math` и `disp`. Если символ `fft` экспортируется пакетом `math`, то в пакете `disp` он будет доступен как `math:fft`. Внутри пакета `math` к нему можно обращаться без префикса: `fft`.

Ниже приводится пример определения пакета, которое можно поместить в начало файла:Ё030in

```
(defpackage "MY-APPLICATION"
            (:use "COMMON-LISP" "MY-UTILITIES")
            (:nicknames "APP")
            (:export "WIN" "LOSE" "DRAW"))

(in-package my-application)
```

Для создания нового пакета используется `defpackage`. Пакет `my-application`[1] использует два других пакета, `common-lisp` и `my-utilities`. Это означает, что символы, экспортируемые ими, доступны в новом пакете *без* использования префиксов. Практически всегда в создаваемых пакетах используется `common-lisp`, и вам не обязательно использовать тот префикс для обращения к встроенным в Лисп операторам и переменным.

Сам `my-applications` экспортирует лишь три символа: `win`, `lose` и `draw`. В `defpackage` был задано также сокращенное имя пакета (`nickname`) – `app`, и к экспортируемым символам можно будет обращаться также и через него: `app:win`.

За `defpackage` следует вызов `in-package`, выставляющий текущим пакетом `my-application`. Теперь все новые символы, для которых не указан пакет, будут интернироваться в `my-application`, и так до следующего вызова `in-package`. После загрузки файла, содержащего вызов `in-package`, текущим пакетом становится тот, который был до его загрузки.

8.6. Ключевые слова

Символы пакета `keyword` (известные как *ключевые слова*) имеют две особенности: они всегда самовычисляемы, и вы можете всегда ссылаться на них без использования префикса: `:x` вместо `keyword:x`. Когда ключевые параметры были использованы нами впервые (см. стр. Ё055out), вызов `(member '(a) '((a) (z)`

[1] В этом примере используются имена, состоящие лишь из заглавных букв, потому что, как упомянуто в разделе 8.1, имена символов конвертируются в верхний регистр.

`) test: #'equal)` показаться более естественным, чем `(member '(a) '((a ) (z)) :test #'equal)`. Теперь мы понимаем, почему второе, слегка неестественное выражение является корректным. Двоеточие перед именем символа позволяет отнести его к ключевым словам.

Зачем использовать ключевые слова вместо обычных символов? Потому что они доступны отовсюду. Функция, принимающая символы в качестве аргументов, как правило должна ожидать ввода ключевых слов. Например, следующий код может быть вызван из любого пакета без изменений:

```
(defun noise (animal)
  (case animal
    (:dog :woof)
    (:cat :meow)
    (:pig :oink)))
```

Если бы эта функция использовала обычные символы, то она могла бы вызываться лишь в исходном пакете, пока не будут экспортированы все использованные в ней символы.

8.7. Символы и переменные

С переменными в Лиспе связана особенность, которая может смущать новичков: символы могут быть связаны с переменными двумя различными способами. Если символ является именем специальной переменной, ее значение хранится в одном из полей структуры символа (см. рис. 8.1). Получить доступ к этому полу можно с помощью `symbol-value`. Таким образом, существует прямая связь между символом и представляемой им специальной переменной.

С лексическими переменными дело обстоит иначе. Символ, используемый в качестве лексической переменной, всего лишь заменяет соответствующее ему значение. Компилятор будет заменять ссылку на лексическую переменную регистром или областью памяти. В полностью скомпилированном коде не будет каких-либо упоминаний этого символа (разумеется, если не включена соответствующая опция отладки). Ни о какой связи между символом и значением лексической переменной говорить не приходится: во время исполнения кода в нем остаются лишь значения, но не символы.

8.8. Пример: бредогенератор.

Ё001inЕсли вам предстоит написать программу, каким-либо образом работающую со словами, отличной идеей часто является использование символов вместо строк, потому что символы по своей природе единичны и могут соответствовать лишь одному слову. Символы могут сравниваться за одну итерацию с помощью `eql`, в то время как строки сравниваются побуквенно с помощью `string-equal` или `string=`. В качестве примера такой программы в этом разделе рассматривается генерация случайного текста. Первая часть программы будет считывать образец текста (чем больше – тем лучше), собирая информацию о связях между соседними словами. Вторая ее часть будет случайным образом выполнять проходы по сети, построенной ранее из слов исходного текста. После прохождения каждого слова программа будет делать взвешенный случайный шаг к следующему слову, встретившемуся после данного в оригинальном тексте. Полученный таким образом текст будет местами казаться довольно связным, потому что каждая пара слов в нем соотносится друг с другом. Поразительно, но целые предложения, а иногда даже и абзацы будут казаться вполне осмысленными.

На рис. 8.2 приводится первая часть программы – код для сбора информации из исходного текста. На его основе строится хеш-таблица `*words*`. Ключами ней являются символы, соответствующие словам, а значениям будут ассоциативные списки типа такого:

```
(defparameter *words* (make-hash-table :size 10000))

(defconstant maxword 100)

(defun read-text (pathname)
  (with-open-file (s pathname :direction :input)
    (let ((buffer (make-string maxword))
          (pos 0))
      (do ((c (read-char s nil :eof)
              (read-char s nil :eof)))
          ((eql c :eof))
        (if (or (alpha-char-p c) (char= c #\'))
            (progn
              (setf (aref buffer pos) c)
              (incf pos))
            (progn
              (unless (zerop pos)
                (see (intern (string-downcase
                              (subseq buffer 0 pos))))
                (setf pos 0))
              (let ((p (punc c)))
                (if p (see p)))))))))

(defun punc (c)
  (case c
    (#\. '|.|) (#\, '|,|) (#\; '|;|)
    (#\! '|!|) (#\? '|?|) ))

(let ((prev `|.|))
  (defun see (symb)
    (let ((pair (assoc symb (gethash prev *words*))))
      (if (null pair)
          (push (cons symb 1) (gethash prev *words*))
          (incf (cdr pair))))
    (setf prev symb)))
```

Рис. 8.2. Чтение образца текста.

```
((|sin| . 1) (|wide| . 2) (|sights| . 1))
```

Выше приведено значение ключа `|discover|` для отрывка из «Paradise Lost» Мильтона[1]. Мы видим, что слово «discover» было использовано в поэме четыре раза, по разу перед словами «sin» и «sights», дважды перед словом «wide». Подобная информация собирается функцией `read-text`, которая строит такой ассоциативный список для каждого слова в заданном тексте. Файл читается побуквенно, слова собираются в строку `buffer`. С параметром `maxword=100` программа сможет читать слова, состоящие не более, чем из ста букв. Для английских слов этого более чем достаточно.

Если считанный знак является буквой (определяется с помощью `alpha-char-p`) или апострофом, то он записывается в буфер. Любой другой символ сигнализирует окончание слова, которое тут же отправляется в функцию `see`. Некоторые знаки пунктуации также расцениваются как слова. Функция `punc` выводит словесный эквивалент заданного знака пунктуации.

Функция `see` регистрирует каждое встреченное слово. Ей также необходимо иметь информацию о предыдущем слове, для этого используется переменная `prev`. Первоначально эта переменная содержит псевдослово, соответствующее точке. Если `see` уже вызывалась хотя бы раз, то `prev` содержит слово, переданное этой функции в прошлый раз.

После отработки `read-text` таблица `*words*` будет содержать вхождения всех слов в заданном тексте. Подсчитать их количество можно с помощью `hash-table-count`. Английские тексты, не отличающиеся особым словесным разнообразием, редко содержат более 10000 различных слов.

[1] Речь об эпической поэме английского мыслителя Джона Мильтона «Потерянный рай», изданной в 1667 году. Поэма написана белым стихом. – *Прим. перев.*

А вот теперь начинается развлечение. На рис. 8.3 представлен код, генерирующий текст с помощью кода на рис. 8.2. Правит бал рекурсивная функция `generate-text`. Ей необходимо сообщить желаемое количество слов в создаваемом тексте и (не обязательно) предыдущее слово. По умолчанию это точка, и текст начинается с нового предложения.

```
(defun generate-text (n &optional (prev '|.|))
  (if (zerop n)
      (terpri)
      (let ((next (random-next prev)))
        (format t "~A " next)
        (generate-text (1- n) next))))

(defun random-next (prev)
  (let* ((choices (gethash prev *words*))
         (i (random (reduce #'+ choices
                            :key #'cdr))))
    (dolist (pair choices)
      (if (minusp (decf i (cdr pair)))
          (return (car pair))))))
```

Рис. 8.3. Генерация текста.

Для получения каждого последующего слова `generate-text` вызывает `random-text` с предыдущим словом. Функция `random-text` случайным образом выбирает одно из слов, следующих после `prev` в исходном тексте. Вероятность выбора одного из перечисленных слов определяется в соответствии с частотой его появления в тексте°Ё091in.

Теперь самое время осуществить тестовый запуск нашей программы. Однако, в действительности, вы уже знакомы с примером того, что она может сделать: речь о той самой строфе в начале книги, для приготовления которой был использован отрывок из «Paradize Lost» Мильтона°Ё092in.

Итоги главы

7. Именем символа может быть любая строка, но символы, создаваемые через `read`, по умолчанию преобразуются к верхнему регистру.
8. С символами связаны списки свойств, которые функционируют подобно ассоциативным спискам, хотя внешне на них не похожи.
9. Символы имеют внушительные размеры и более напоминают структуры, нежели просто имена.
10. Пакеты отображают строки в символы. Чтобы создать новый символ, принадлежащий пакету, необходимо этот символ интернировать в него. Символы не обязательно должны быть интернированы.
11. С помощью пакетов реализуется модульность, ограничивающая область действия имен. По умолчанию программы будут находиться в пользовательском пакете, однако большие программы часто разделяются на несколько пакетов, имеющих собственное назначение.
12. Символы одного пакета могут быть доступны в другом. Ключевые слова самовычисляемы и доступны из любого пакета.
13. В программах, работающих с отдельными словами, удобно представлять слова с помощью символов.

Упражнения

14. Могут ли два символа иметь одно имя, но не быть эквивалентными с точки зрения `eql`?
15. Оцените различие между количеством памяти для представления строки `"FOO"` и символа `foo`.
16. В вызове `defpackage`, приведенном на стр. Ё030out, использовались лишь строки. Тем не менее, мы могли бы воспользоваться символами вместо строк. Чем это может

быть чревато?

17. Добавьте в программу на рис. 7.1 код, который помещает ее содержимое в пакет `"RING"`. Аналогично, для кода на рис. 7.2 создайте пакет `"FILE"`. Уже имеющийся код должен остаться без изменений.
18. Напишите программу, проверяющую, была ли заданная цитата произведена с помощью Хенли (см. раздел 8.8).
19. Напишите версию Хенли, которая принимает слово и производит предложение, в середине которого находится заданное слово.

9. Числа

Числодробление – одна из сильных сторон Лиспа. В нем имеется богатый набор числовых типов, а по возможностям он даст фору многим другим языкам.

9.1. Типы

Common Lisp предоставляет множество различных числовых типов: целых, с плавающей запятой, рациональные, комплексные. Большинство функций, рассмотренных в этой главе, работают одинаково со всем типами чисел. За исключением, может быть, комплексных чисел, к которым некоторые из них неприменимы.

Целое число записывается строкой цифр: `2001`. Число с плавающей запятой, помимо цифр, содержит десятичный разделитель – точку: `253.72`, или `2.5372e2` в экспоненциальном представлении. Рациональная дробь представляется в виде отношения двух целых чисел: `2/3`. Комплексное число вида `a+bi` можно записать как `#c(a b)`, где `a` и `b` – два действительных числа одного типа.

Проверку на принадлежность к соответствующему типы осуществляют предикаты `integerp`, `floatp` и `complexp`. Иерархия численных типов представлена на рис. 9.1.

Рис. 9.1. Численные типы.

Ниже приведены основные правила, согласно которым можно определить, число какого типа будет получено в результате вычисления:

1. Если функция принимает хотя бы одно число с плавающей запятой, она также вернет десятичную дробь (или комплексное число, компоненты которого - десятичные дроби). `(+ 1.0 2)` вернет `3.0`, а `(+ #c(0 1.0) 2)` вернет `#c(2.0 1.0)`.
2. Рациональная дробь при возможности будет сокращена до целого числа. Вызов `(/ 10 2)` вернет `5`.
3. Комплексные числа, мнимая часть которых равна нулю, будут сконвертированы в действительные. Вызов `(+ #c(1 -1) #c(2 1))` вернет `3`.

Преобразования 2 и 3 вступают в силу в момент прочтения выражения, поэтому:

```
> (list (ratiop 2/2) (complexp #c(1 0)))
(NIL NIL)
```

9.2. Преобразование и извлечение

В Лиспе имеются средства для преобразования, а также извлечения компонентов, чисел, принадлежащих любым типам. Функция `float` преобразует любое действительное число в эквивалентную ему десятичную дробь:

```
> (mapcar #'float '(1 2/3 .5))
(1.0 0.66666667 0.5)
```

Не всегда следует прибегать к преобразованию в числа с плавающей запятой, потому что это может повлечь некоторую потерю информации. Целочисленную компоненту любого действительного числа можно получить с помощью функции `truncate`:

```
> (truncate 1.3)
 1
0.2999995
```

Второе значение соответствует результату вычитания первого значения из аргумента. (Разница в `.00000005` возникает вследствие неизбежной неоднозначности операций с плавающей запятой.)

Функции `floor`, `ceiling` и `round` также приводят свои аргументы к целочисленным значениям. С помощью `floor`, возвращающей наибольшее целое

число, меньшее или равное заданному аргументу, а также `ceiling`, возвращающей наименьшее целое, большее или равное аргументу, мы можем обобщить функцию `mirror?` (см. стр. Ё012out) для распознавания палиндромов:

```
(defun palindrome? (x)
  (let ((mid (/ (length x) 2)))
    (equal (subseq x 0 (floor mid))
           (reverse (subseq x (ceiling mid))))))
```

Как и `truncate`, функции `floor` и `ceiling` возвращают вторым значением разницу между аргументом и своим первым значением:

```
> (floor 1.5)
1
0.5
```

Таким образом, мы могли бы определить `truncate` таким образом:

```
(defun our-truncate (n)
  (if (> n 0)
     (floor n)
     (ceiling n)))
```

Для получения ближайшего целого числа существует функция `round`. В случае, когда ее аргумент равноудален от обоих целых чисел, Common Lisp, как и многие другие языки, *не* округляет его, а возвращает ближайшее четное число:

```
> (mapcar #'round '(-2.5 -1.5 1.5 2.5))
(-2 -2 2 2)
```

Такой подход позволяет сгладить накопление ошибок округления в ряде приложений. В любом случае, если вам необходимо округление вверх, вы можете написать такую функцию самостоятельно.[1] Как и остальные функции такого рода, `round` возвращает второе значение – разницу между исходным и округленным числами.

Функция `mod` возвращает второе значение аналогичного вызова `truncate`, функция `rem` – первое. Мы уже имели дело с `mod` (см. стр. Ё031out) для определения делимости двух чисел, а также когда определяли реальное местонахождение в кольцевом буфере (см. стр. Ё032out).

Для действительных чисел существует функция `signum`, которая возвратит `1`, `0` или `-1`, в зависимости от того, каков знак аргумента: минус, ноль, плюс. Абсолютное значение числа можно получить с помощью функции `abs`. Так, `(* (abs x) (signum x)) = x`.

```
> (mapcar #'signum '(-2 -0.0 0.0 0 .5 3))
(-1 -0.0 0.0 0 1.0 1)
```

В некоторых реализациях `-0.0` может существовать сам по себе, как в примере, приведенном выше. В любом случае, между ними нет никакой разницы, и в вычислениях `-0.0` ведет себя в точности как `0.0`.

Рациональные дроби и комплексные числа являются структурами, состоящими из двух частей. Получить соответствующие целочисленные компоненты рациональной дроби можно с помощью функций `numerator` и `denominator`. (Если аргумент уже является целым числом, то первая функция вернет сам аргумент, а последняя – единицу.) Аналогично, функции `realpart` и `imagpart` извлекают действительную и мнимую части комплексного числа. (Если аргумент не комплексный, то первая вернет это число, а вторая вернет ноль.)

Функция `random` принимает целые числа или десятичные дроби. Выражение вида `(random n)` вернет число, большее или равное нулю и меньшее `n`, и это значение будет того же типа, что и аргумент.

---

[1] Функция `format` при округлении не гарантирует даже того, будет ли получено четное или нечетное число. См. стр. Ё033out.

9.3. Сравнение

Ё010inДля сравнения двух чисел пользуйтесь предикатом `=`:

```
> (= 1 1.0)
T
> (eql 1 1.0)
NIL
```

Он менее строг, чем `eql`, так как последний также требует равенство типов сравниваемых чисел.

Предикатами для сравнения являются `<` (менее), `<=` (менее или равно), `=` (равно), `>=` (более или равно), `>` (более) и `/=` (не равно). Все они принимают один или более аргументов. Вызванные с одним аргументом, они всегда возвращают истину. Для всех функций, кроме `/=`, вызов с тремя и более аргументами:

```
(<= w x y z)
```

равноценен объединению попарных сравнений:

```
(and (<= w x) (<= x y) (<= y z))
```

Так как `/=` возвращает истину лишь когда *ни один* из аргументов не равен другому, выражение:

```
(/ w x y z)
```

эквивалентно

```
(and (/= w x) (/= w y) (/= w z)
     (/= x y) (/= x z) (/= y z))
```

Существуют также специализированные предикаты `zerop`, `plusp` и `minusp`, принимающие только один аргумент и возвращающие истину, если он `=`, `>` или `<` нуля соответственно. Эти предикаты взаимоисключающие. Что касается `-0.0` (если используемая реализация его использует), то, несмотря на знак минуса, это число `=` `0`, и соответствует `zerop`, а не `minusp`:

```
> (list (minusp -0.0) (zerop -0.0))
(NIL T)
```

Предикаты `oddp` и `evenp` действительны лишь для целочисленных аргументов. Первый истинен для нечетных чисел, второй – для четных.

Из всех предикатов, упомянутых в этом разделе, лишь `=`, `/=` и `zerop` применимы к комплексным числам.

Наибольший и наименьший элементы могут быть получены с помощью, соответственно, `max` и `min`. Должен быть задан хотя бы один аргумент:

```
> (list (max 1 2 3 4 5) (min 1 2 3 4 5))
(5 1)
```

Если среди аргументов есть хотя бы одно число с плавающей запятой, то тип возвращаемого значения зависит от используемой реализации.

9.4. Арифметика

Сложение и вычитание выполняются с помощью `+` и `-`. Обе функции могут работать с любым количеством аргументов, в случае их отсутствия возвращают `0`. Выражение вида `(- n)` вычисляется в `-n`. Выражение вида

```
(- x y z)
```

равноценно

```
(- (- x y) z)
```

Кроме того, имеются функции `1+` и `1-`, которые возвращают свой аргумент, увеличенный или уменьшенный на `1`. Имя функции `1-` может сбивать с толку, потому что `(1- x)` возвращает `x-1`, но не `1-x`.

Макросы `incf` (`decf`) увеличивают (уменьшают) свой аргумент. Выражение вида `(incf x n)` имеет такой же эффект, как и `(setf x (+ x n))`, а `(decf x n)` – как `(setf x (- x n))`. В обоих случаях второй аргумент не обязателен и по умолчанию равен `1`.

Умножение выполняется с помощью функции `*`, которая принимает любое количество аргументов и возвращает их общее произведение. Будучи вызванной без аргументов она возвращает `1`.

Функция деления, `/`, требует задания хотя бы одного аргумента. Вызов `(/ n)` равноценен вызову `(/ 1 n)`:

```
> (/ 3)
1/3
```

Выражение:

```
(/ x y z)
```

эквивалентно

```
(/ (/ x y) z)
```

Обратите внимание на сходное поведение `-` и `/`.

Вызванная с двумя аргументами, `/` возвращает рациональную дробь, если первый аргумент не делится на второй:

```
> (/ 365 12)
365/12
```

Если вы будете пытаться вычислять усредненную длину месяца, то вам может показаться, что toplevel над вами издевается. В подобных случаях, когда вам действительно нужна десятичная дробь, пользуйтесь функцией `float`:

```
> (float 365/12)
30.416666
```

9.5. Возведение в степень

Нахождение x^n выполняется как `(expt x n)`:

```
> (expt 2 5)
32
```

Нахождение логарифма $\log_n x$ – как `(log x n)`:

```
> (log 32 2)
5.0
```

Обычно при вычислении логарифма возвращается число с плавающей запятой.

Для нахождения степени числа `e` существует специальная функция `exp`:

```
> (exp 2)
7.389056
```

Вычисление натурального логарифма выполняется функцией `log`, которой достаточно сообщить лишь само число:

```
> (log 7.389056)
2.0
```

Вычисление корней также может выполняться с помощью `expt`, для чего второй ее аргумент должен быть рациональной дробью:

```
> (sqrt 4)
2.0
```

9.6. Тригонометрические функции

Численным представлением числа π является число с плавающей запятой – константа `pi`. Ее точность зависит от используемой реализации. Функции `sin`, `cos` и `tan` вычисляют, соответственно, синус, косинус и тангенс заданного в радианах угла:

```
> (let ((x (/ pi 4)))
    (list (sin x) (cos x) (tan x)))
(0.7071067811865475d0 0.7071067811865476d0 1.0d0)
```

Все вышеперечисленные функции умеют работать с комплексными аргументами.

Обратные преобразования выполняются функциями `asin`, `acos`, `atan`. Для аргументов, находящихся на отрезке от `-1` до `1`, `asin` и `acos` возвращают действительные значения.

Гиперболические синус, косинус и тангенс вычисляются через `sinh`, `cosh` и `tanh`, соответственно. Для них также определены обратные преобразования: `asinh`, `acosh`, `atanh`.

9.7. Представление

Common Lisp не накладывает каких-либо ограничений на размер целых чисел. Целые числа небольшого размера, которые помещаются в машинном слове, относятся к типу *fixnum*. Если для хранения числа требуется памяти более, нежели слово, Лисп переключается на представление *bignum*, которое выделяет несколько слов для хранения целого числа. Это означает, что сам язык не накладывает каких-либо ограничений на размер чисел, и он зависит лишь от доступного объема памяти.

Константы `most-positive-fixnum` и `most-negative-fixnum` задают максимальные величины, которые могут обрабатываться реализацией без использования типа `bignum`. Во многих реализациях они имеют следующие значения:

```
> (values most-positive-fixnum most-negative-fixnum)
536870911
-536870912
```

Принадлежность к определенному типу проверяется с помощью предиката `typep`.

```
> (typep 1 'fixnum)
T
> (typep (1+ most-positive-fixnum) 'bignum)
T
```

Ограничения на размер чисел с плавающей запятой свои в каждой реализации. Common Lisp предоставляет четыре типа чисел с плавающей запятой: `short-float`, `single-float`, `double-float` и `long-float`. Реализации стандарта не обязаны использовать различные представления для разных типов (и не каждая разделяет их все).

Суть типа `short-float` в том, что он занимает одно машинное слово. Типы `single-float` и `double-float` занимают столько места, чтобы удовлетворять установленным требованиям к числам одинарной и двойной точности соответственно. Числа типа `long-float` могут быть действительно большими, настолько, насколько это требуется. Однако, реализациям стандарта здесь предоставлена некоторая свобода.

Задать тип числа можно принудительно, используя соответствующие буквы: `s`, `f`, `d` или `l`, а также `e` для экспоненциальной формы (Заглавные буквы также допускаются, и это отличная идея для представления типа `long-float`, потому что малую `l` легко спутать с цифрой `1`.) Наибольшее представление числа `1` можно задать как `1L0`.

Глобальные ограничения на размер чисел разных типов задаются в сумме шестнадцатью константами. Их имена выглядят как `m-s-f`, где `m` может быть `most` или `least`, `s` соответствует `positive` или `negative`, а `f` – один из четырех типов.°Ё093in

Превышение заданных ограничений приводит к возникновению ошибки:

```
> (* most-positive-long-float 10)
Error: floating-point-overflow.
```

9.8. Пример: трассировка лучей

В качестве примера программы, построенной на численных расчетах, в этом разделе приводится решение задачи трассировки лучей. Трассировка лучей – это превосходный алгоритм рендеринга изображений, с помощью которого можно получать реалистичные картины. Метод, однако, довольно дорогостоящий.

Для моделирования трехмерного изображения нам необходимо определить как минимум четыре предмета: наблюдателя, один или более источников света, набор объектов моделируемого мира и плоскость рисунка (*image plane*), которая служит окном в этот мир. Наша задача сгенерировать изображение, соответствующее проекции мира на область плоскости рисунка.

Что же делает необычным метод трассировки лучей? Попиксельная отрисовка всей картины и симуляция прохождения луча через виртуальный мир. Такой подход позволяет достичь реалистичных оптических эффектов: прозрачности, отражений, затенений; он позволяет задавать отрисовываемый мир как набор геометрических тел, вместо того, чтобы строить их из полигонов. Отсюда вытекает относительная прямолинейность в реализации метода.

На рис. 9.2 показаны основные математические утилиты, которыми мы будем пользоваться. Первая, `sq`, вычисляет квадратный корень. Вторая, `mag`, возвращает длину вектора по трем его компонентам. Эта функция используется в следующих двух: `unit-vector` возвращает три значения, соответствующие координатам единичного вектора, имеющего то же направление, что и заданный.

```
(defun sq (x) (* x x))

(defun mag (x y z)
  (sqrt (+ (sq x) (sq y) (sq z))))

(defun unit-vector (x y z)
  (let ((d (mag x y z)))
    (values (/ x d) (/ y d) (/ z d))))

(defstruct (point (:conc-name nil))
  x y z)

(defun distance (p1 p2)
  (mag (- (x p1) (x p2))
       (- (y p1) (y p2))
       (- (z p1) (z p2))))

(defun minroot (a b c)
  (if (zerop a)
      (/ (- c) b)
      (let ((disc (- (sq b) (* 4 a c))))
        (unless (minusp disc)
          (let ((discrt (sqrt disc)))
            (min (/ (+ (- b) discrt) (* 2 a))
                 (/ (- (- b) discrt) (* 2 a))))))))
```

Рис. 9.2. Математические утилиты

```
> (multiple-value-call #'mag (unit-vector 23 12 47))
1.0
```

Кроме того, `mag` используется в функции `distance`, вычисляющей расстояние между двумя точками в трехмерном пространстве. (Определение структуры `point` содержит параметр `:conc-name`, равный `nil`. Это значит, что функции доступа к

соответствующим полям структуры будут иметь такое же имя, как само поля: `x` вместо `point-x`, например.)

Наконец, функция `minroot` принимает три действительных числа: `a`, `b` и `c`, возвращая наименьшее `x`, для которого $ax^2+bx+c=0$. В случае, когда `a` не равно нулю, корни такого выражения могут быть получены по хорошо известной формуле:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Код, реализующий ограниченно функциональный трассировщик лучей, представлен на рис. 9.3. Он генерирует черно-белые изображения, освещаемые одним источником света, расположенным в том же месте, где и глаз наблюдателя. (Таким образом получается эффект фотографии со вспышкой.)

```
(defstruct surface  color)

(defparameter *world* nil)
(defconstant eye (make-point :x 0 :y 0 :z 200))

(defun tracer (pathname &optional (res 1))
  (with-open-file (p pathname :direction :output)
    (format p "P2 ~A ~A 255" (* res 100) (* res 100))
    (let ((inc (/ res)))
      (do ((y -50 (+ y inc)))
          ((< (- 50 y) inc))
        (do ((x -50 (+ x inc)))
            ((< (- 50 x) inc))
          (print (color-at x y) p))))))

(defun color-at (x y)
  (multiple-value-bind (xr yr zr)
                       (unit-vector (- x (x eye))
                                    (- y (y eye))
                                    (- 0 (z eye)))
    (round (* (sendray eye xr yr zr) 255))))

(defun sendray (pt xr yr zr)
  (multiple-value-bind (s int) (first-hit pt xr yr zr)
    (if s
        (* (lambert s int xr yr zr) (surface-color s))
        0)))

(defun first-hit (pt xr yr zr)
  (let (surface hit dist)
    (dolist (s *world*)
      (let ((h (intersect s pt xr yr zr)))
        (when h
          (let ((d (distance h pt)))
            (when (or (null dist) (< d dist))
              (setf surface s hit h dist d))))))
    (values surface hit)))

(defun lambert (s int xr yr zr)
  (multiple-value-bind (xn yn zn) (normal s int)
    (max 0 (+ (* xr xn) (* yr yn) (* zr zn)))))
```

Рис. 9.3. Трассировка лучей.

Для представления объектов в виртуальном мире используется структура `surface`. Говоря точнее, она будет включена в структуры, представляющие конкретные разновидности объектов, например, сферы. Сама структура `surface` содержит лишь одно поле: цвет в интервале от `0` (черный) до `1` (белый).

Плоскость рисунка располагается вдоль осей x и y. Глаз наблюдателя смотрит вдоль оси z и находится на расстоянии 200 единиц от рисунка. Чтобы добавить объект, необходимо поместить его в список `*world*` (исходно `nil`). Чтобы они были видимыми, их z-координата должна быть отрицательной. На рис. 9.4

демонстрируется прохождение лучей через поверхность рисунка и их падение на сферу.

Рис. 9.4. Трассировка лучей.

Функция `tracer` записывает изображение в файл по заданному пути. Запись производится в обычном ASCII-формате, называемом PGM. По умолчанию границы изображения `100x100`. Заголовок PGM-файла начинается с тега `P2`, за которым следуют числа, соответствующие ширине (100) и высоте (100) в пикселах, причем максимально возможное значение равно 255. Оставшаяся часть файла содержит 10000 целых чисел от 0 (черный) до 255 (белый), которые вместе дают 100 горизонтальных полос по 100 пикселов в каждой.

Разрешение изображения может быть изменено с помощью `res`. Если `res` равно, например, `2`, то изображение будет содержать `200x200` пикселов.

По умолчанию изображение представляет собой квадрат `100x100`. Каждый пиксел представляет количество света, проходящего через данную точку к глазу наблюдателя. Для нахождения этой величины `tracer` вызывает `color-at`. Эта функция ищет вектор от глаза наблюдателя к заданной точке, затем вызывает `sendray`, направляя луч вдоль этого вектора через моделируемый мир. В результате `sendray` получает некоторое число от 0 до 1, которое затем масштабируется на отрезок от 0 до 255.

Для получения интенсивности света `sendray` ищет объект, от которого он был отражен. Для этого вызывается `first-hit`, которая находит среди всех объектов в `*world*` тот, на который луч падает первым, или же убеждается в том, что луч не падает ни на один объект. В последнем случае возвращается цвет фона, которым мы условились считать `0` (черный). Если луч падает на один из объектов, то нам необходимо найти долю света, отраженного от него. Согласно закону Ламберта, интенсивность излучения, отраженного точкой поверхности, пропорциональна скалярному произведению единичного вектора `N`, направленного из этой точки вдоль нормали к поверхности (вектор длины `1`, направленный перпендикулярно поверхности в заданной точке), и единичного вектора `L`, направленного вдоль луча к источнику света:

```
i = N·L
```

Если источник света находится в этой точке, `N` и `L` будут сонаправлены, и их скалярное произведение будет иметь максимальное значение - `1`. Если поверхность находится под углом 90° к источнику света, то `N` и `L` будут перпендикулярны и их произведение будет равно `0`. Если источник света находится за поверхностью, то произведение будет отрицательным.

В нашей программе используется предположение, что источник света находится там же, где и глаз наблюдателя, и функция `lambert`, использующая вышеописанное правило для нахождения освещенности некоторой точки поверхности, возвращает скалярное произведение нормали и трассируемого луча.

В функции `sendray` это число умножается на цвет поверхности (темная поверхность отражает меньше света), чтобы определить интенсивность в заданной точке. Упростим задачу, ограничившись лишь объектами типа сферы. Код, реализующий поддержку сфер, приведен на рис. 9.5. Структура `sphere` включает `surface`, и `sphere` будет иметь параметр `color` так же, как и параметры `center` и `radius`. Вызов `defsphere` добавляет в наш мир новую сферу.

```
(defstruct (sphere (:include surface))
  radius center)

(defun defsphere (x y z r c)
  (let ((s (make-sphere
             :radius r
             :center (make-point :x x :y y :z z)
             :color  c)))
    (push s *world*)
    s))

(defun intersect (s pt xr yr zr)
  (funcall (typecase s (sphere #'sphere-intersect))
           s pt xr yr zr))

(defun sphere-intersect (s pt xr yr zr)
  (let* ((c (sphere-center s))
         (n (minroot (+ (sq xr) (sq yr) (sq zr))
                     (* 2 (+ (* (- (x pt) (x c)) xr)
                             (* (- (y pt) (y c)) yr)
                             (* (- (z pt) (z c)) zr)))
                     (+ (sq (- (x pt) (x c)))
                        (sq (- (y pt) (y c)))
                        (sq (- (z pt) (z c)))
                        (- (sq (sphere-radius s)))))))
    (if n
        (make-point :x  (+ (x pt) (* n xr))
                    :y  (+ (y pt) (* n yr))
                    :z  (+ (z pt) (* n zr))))))

(defun normal (s pt)
  (funcall (typecase s (sphere #'sphere-normal))
           s pt))

(defun sphere-normal (s pt)
  (let ((c (sphere-center s)))
    (unit-vector (- (x c) (x pt))
                 (- (y c) (y pt))
                 (- (z c) (z pt)))))
```

Рис. 9.5. Сферы.

Функция `intersect` (пересечение) распознает тип объекта и вызывает соответствующую ему функцию. На данный момент мы умеем работать лишь со сферами. Для сферы из `intersect` будет вызываться `sphere-intersect`, но наш код может быть с легкостью расширен для поддержки других объектов.

Как найти пересечение луча со сферой? Луч представляется точкой $p = \langle x_0, y_0, z_0 \rangle$ и единичным вектором $v = \langle x_r, y_r, z_r \rangle$. Любая точка, принадлежащая лучу, может быть выражена как $p+nv$ для некоторого n, или, что то же самое, $\langle x_0+nx_r, y_0+ny_r, z_0+nz_r \rangle$. Когда луч падает на сферу, расстояние до центра $\langle x_c, y_c, z_c \rangle$ равно радиусу r. Таким образом, условие пересечения луча и сферы можно записать следующим образом:

$$r = \mathrm{sqrt}((x_0-nx_r-x_c)^2+(y_0+ny_r-y_c)^2+(z_0+nz_r-z_c)^2)$$

Это выражение можно записать следующим образом:

$$an^2+bn+c = 0$$

где

$$\begin{aligned} a &= xr^2+yr^2+zr^2 \\ b &= 2((x_0-x_c)x_r+(y_0-y_c)y_r+(z_0-z_c)z_r) \\ c &= (x_0-xc)^2+(y_0-yc)^2+(z_0-zc)^2-r^2 \end{aligned}$$

Для нахождения точки пересечения нам необходимо решить это квадратное уравнение. Оно может иметь два, одно или ни одного решения. Отсутствие решений означает, что луч проходит мимо сферы; одно означает, что луч проходит через

сферу лишь в одной точке (то есть, касается ее); два решения сигнализируют о том, что луч проходит через сферу, пересекая ее поверхность два раза. В последнем случае нам интересен лишь наименьший корень. При удалении луча `n` увеличивается, поэтому наименьшее решение будет соответствовать меньшему `n`. Для нахождения корня используется `minroot`. Если корень существует, `sphere-intersect` возвратит соответствующую ему точку $\langle x_0+nx_r,\ y_0+ny_r,\ z_0+nz_r \rangle$.

Две другие функции на рис. 9.5, `normal` и `sphere-normal`, связаны аналогично `intersect` и `sphere-intersect`. Поиск нормали для сферы крайне прост – это всего лишь вектор, направленный из точки на поверхности к ее центру.

На рис. 9.6. показано, как будет выполняться генерация изображения; `ray-trace` создает 38 сфер (не все они будут наблюдаемы), а затем генерирует изображение, которое записывается в файл `"spheres.pgm"`. Итог работы программы с параметром `res = 10` представлен на рис. 9.7.

```
(defun ray-test (&optional (res 1))
  (setf *world* nil)
  (defsphere 0 -300 -1200 200 .8)
  (defsphere -80 -150 -1200 200 .7)
  (defsphere 70 -100 -1200 200 .9)
  (do ((x -2 (1+ x)))
      ((> x 2))
    (do ((z 2 (1+ z)))
        ((> z 7))
      (defsphere (* x 200) 300 (* z -400) 40 .75)))
  (tracer (make-pathname :name "spheres.pgm") res))
```

Рис. 9.6. Использование трассировщика.

Рис. 9.7. Изображение, полученное методом трассировки лучей.

Полноценный трассировщик лучей в состоянии производить куда более сложные изображения с учетом нескольких источников света разной интенсивности, расположенные в произвольных точках пространства. Для этого программа должна учитывать эффекты отбрасывания теней. Расположения источника света рядом с глазом наблюдателя позволило нам существенно упростить задачу, так как наблюдатель не сможет видеть ни одну из отбрасываемых теней.

Кроме того, полноценный трассировщик должен учитывать вторичные отражения. Разумеется, объекты различных цветов также будут отражать свет по-разному. Тем не менее, наша реализация (см. рис. 9.3) предоставляет основной механизм трассировки, который в дальнейшем может быть улучшен и дополнен, например, с помощью рекурсивного использования уже имеющихся методов.

Кроме того, нормальная программа-трассировщик должна быть тщательно оптимизирована. Наша программа имеет сжатые размеры и не оптимизирована ни как Лисп-программа, ни как трассировщик лучей. Одни только инлайн-декларации и декларации типов (см. раздел 13.3) могут обеспечить более чем двукратный прирост производительности.

Итоги главы

1. Common Lisp предоставляет в ваше распоряжение целые числа, рациональные дроби, числа с плавающей запятой и комплексные числа.
2. Числовые значения могут быть преобразованы или упрощены, а их компоненты могут быть извлечены.
3. Предикаты сравнения чисел принимают любое количество аргументов, и последовательно сравнивают их пары. `/=` сравнивает все возможные пары аргументов.
4. В Common Lisp имеются практически все функции, необходимые в низкоуровневом инженерном калькуляторе. Одни и те же функции могут применяться к аргументам любых типов.
5. Числа типа `fixnum` – небольшие целые числа, умещающиеся в одном машинном слове. При необходимости могут быть использованы числа типа `bignum`. Их использование существенно дороже и производится без каких-либо предупреждений.

Common Lisp также имеет вплоть до четырех типов чисел с плавающей запятой. Ограничения на их размер свои для каждой реализации и могут быть получены из специальных констант.

6. Трассировщик лучей генерирует изображение, отслеживая прохождение света через смоделированный мир и вычисляя интенсивность излучения для каждого пиксела изображения.

Упражнения

1. Определите функцию, принимающую список действительных чисел и возвращающую истину, когда числа следуют в порядке неубывания.
2. Определите функцию, принимающую целочисленную величину – количество центов, и возвращающую четыре значения, показывающие, как собрать заданную сумму с помощью 25-, 10-, 5- и одноцентовых монет, используя наименьшее их количество.
3. На очень далекой планете живут два вида существ: вигглы и вобблы. И первые, и вторые неплохо поют. Каждый год на той планете проводится грандиозное соревнование, по результатам которого выбирают десять лучших певцов. Вот результаты за прошедшие десять лет:

Год	1	2	3	4	5	6	7	8	9	10
Вигглы	6	5	6	4	5	5	4	5	6	5
Воблы	4	5	4	6	5	5	6	5	4	5

   Напишите программу, симулирующую подобные соревнования. Как по-вашему, действительно ли жюри каждый год выбирает десять самых лучших певцов?

4. Определите функцию, принимающую 8 действительных чисел, представляющих два отрезка в двумерном пространстве. Функция возвращает `nil`, если отрезки не пересекаются, в противном случае возвращаются два значения – x- и y-координаты точки их пересечения.
5. Пусть функция `f` имеет один действительный аргумент, а `min` и `max` – ненулевые действительные числа разных знаков; `f` имеет корень (то есть, равна нулю) для некоторого числа `i`, такого, что `min < i < max`. Определите функцию четырех аргументов: `f`, `min`, `max` и `epsilon`, возвращающую приближенное значение `i` с точностью до `epsilon`.
6. *Метод Горнера* позволяет эффективно вычислять полиномы. Чтобы найти $ax^3+bx^2+c^x+d$, нужно вычислить `x(x(ax+b)+c)+d`. Определите функцию, принимающую один или несколько аргументов – значение `x` и следующие за ним `n` действительных чисел, представляющих полином степени `(n-1)`. Функция должна вычислять значение полинома для заданного `x` методом Горнера.
7. Сколько бит использовала бы ваша реализация для представления `fixnum`?
8. Сколько различных типов чисел с плавающей запятой в ней было бы представлено?

10. Макросы

Лисп-код записывается в виде списков, которые сами являются Лисп-объектами. В разделе 2.3 было обещано показать, как, благодаря этой особенности, становится возможной генерация Лисп-кода. В этой главе показано, как объединить списки и код.

10.1. Eval

Вам уже должно быть совершенно очевидно, как создавать выражения: с помощью `list`. Однако возникает другой вопрос – как объяснить Лиспу, что созданный список – код? Для этого существует функция `eval`, вычисляющая заданное выражение и возвращающая его значение:

```
> (eval '(+ 1 2 3))
6
> (eval '(format t "Hello"))
Hello
NIL
```

Да, именно об `eval` мы говорили все это время. Следующая функция реализует примитивное подобие toplevel:

```
(defun out-toplevel ()
  (do ()
     (nil)
    (format t "~%> ")
    (format (eval (read)))))
```

По этой причине toplevel также называют read-eval-print loop (цикл чтение-вычисление-вывод).

Вызов `eval` – один из способов соединить списки и код. В любом случае, это не лучший способ:

1. Он неэффективен: `eval` получает в руки простой список, и он вынужден его либо скомпилировать, либо вычислить его в интерпретаторе. Несомненно, ни один из этих способов не сможет угнаться за исполнением уже скомпилированного кода.
2. Выражение вычисляется вне какого-либо лексического контекста. Конструкция `eval`, вызываемая внутри `let`, не сможет работать с определенными в ней переменными.

Существуют гораздо лучшие способы (описанные в следующем разделе) для кодогенерации. А сам `eval` уместен лишь для создания чего-то вроде циклов toplevel.

Тем не менее, `eval` может быть полезен для понимания сути Лиспа. Можно считать, что `eval` определен как большое выражение `cond`:

```
(defun eval (expr env)
  (cond ...
        ((eql (car expr) 'quote) (cadr expr))
        ...
        (t (apply (symbol-function (car wxpr))
                  (mapcar #'(lambda (x)
                              (eval x env))
                          (cdr expr))))))
```

Большинство выражений обрабатываются последним условием, которое рассматривает выражение как вызов функции, именем которой является `car`, а аргументами – `cdr`. Перед тем, как передать аргументы в функцию, они вычисляются в порядке от первого к последнему[1].

[1] Как и реальный `eval` наша функция имеет еще один аргумент (`env`), представляющий лексическое окружение. Неточность нашей модели `eval` состоит в том, что она получает функцию перед тем, как приступает к обработке аргументов, в то время как в Common Lisp порядок выполнения этих операций является произвольным.

Тем не менее, для выражения `(quote x)` такой подход неприменим, так как `quote` защищает свой аргумент от вычисления. Это значит, что нам необходимо специальное условие для `quote`. Так мы приходим к сути понятия «специальный оператор» - это оператор, для которого в `eval` имеется собственное условие.

Функции `coerce` и `compile`Ё053in также предоставляют переход от списков к коду. Лямбда-выражение может быть превращено в код:

```
> (coerce '(lambda (x) x) 'function)
#<Interpreted-Function BF9D96>
```

С помощью функции `compile` с первым аргументом `nil` вы также можете скомпилировать лямбда-выражение, переданное вторым аргументом:

```
> (compile nil '(lambda (x) (+ x 2)))
#<Compiled-Function DF55BE>
NIL
NIL
```

Так как `coerce` и `compile` могут принимать списки как аргументы, новые функции могут создаваться на лету. Однако, эти методы так же не лучший выбор, потому что обладают теми ж недостатками, что и `eval`.

Причина неэффективности `eval`, `coerce` и `compile` в том, что они создают функции в момент выполнения (run-time). Избежать этих проблем позволяет выполнение тех же операций во время компиляции. В следующем разделе будет рассмотрена эта возможность.

10.2. Макросы

Наиболее естественным способом генерации кода является создание макросов. *Макросы* – это операторы, осуществляющие преобразование кода. Задача определения макроса состоит в указании пути преобразования кода. Само преобразование выполняется компилятором и называется *раскрытием макроса* (*macro-expansion*). Код, полученный в результате раскрытия макроса, естественным образом становится частью программы, как если бы он был набран собственноручно.

Обычно макросы создаются с помощью `defmacro`. Вызов `defmacro` напоминает вызов `defun`, но вместо вычисления заданного выражения, он указывает, как должно быть преобразовано такое выражение. Так, например, будет выглядеть макрос, устанавливающий значение своего аргумента в `nil`:

```
(defmacro nil! (x)
  (list 'setf x nil))
```

Данный макрос создает новый оператор, называемый `nil!`, который принимает один аргумент. Вызов вида `(nil! a)` будет преобразован в `(setf a nil)` и лишь затем скомпилирован и вычислен. Таким образом, вычисляя `(nil! x)` в toplevel,

```
> (nil! x)
NIL
> x
NIL
```

мы получим результат, аналогичный вызову `(setf x nil)`.

Чтобы протестировать функции, ее вызывают. Чтобы протестировать макрос, его раскрывают. Функция `macroexpand-1` принимает вызов макроса и возвращает то, во что он должен будет раскрыться:

```
> (macroexpand-1 '(nil! x))
(SETF X NIL)
T
```

Макрос может раскрываться в другой макровызов. Когда компилятор (а также toplevel) встречает такой макровызов, он раскрывает этот макрос до тех пор, пока в коде не останется ни одного другого макровызова.

Секрет понимания макросов в осознании их реализации. Они – не более чем функции, преобразующие выражения. Для примера, выражение вида `(nil! a)` может быть обработано следующей функцией:

```
(lambda (expr)
  (apply #'(lambda (x) (list 'setf x nil))
         (cdr expr)))
```

которая вернет `(setf a nil)`. Используя `defmacro`, вы выполняете похожую работу. Все, что делает `macroexpand-1`, когда встречает выражение, `car` которого соответствует имени одного из макросов, - перенаправляет выражения в соответствующую функцию.

10.3. Обратная кавычка

Макрос *обратная кавычка* (*backquote*) позволяет строить списки на основе специальных шаблонов. Она широко используется при определении других макросов. Обычной кавычке на клавиатуре соответствует закрывающая прямая кавычка (апостроф), а обратной – открывающая. Ее называют обратной из-за ее наклона влево.

Обратная кавычка может служить заменой обычной кавычке:

```
> `(a b c)
(A B C)
```

Обратная кавычка, как и обычная, предотвращает вычисление выражения.

Преимуществом обратной кавычки является возможность выборочного вычисления частей выражения с помощью `,` (запятой) и `,@` (запятая и «собака»).Любое подвыражение, предваряемое запятой, будет вычислено. Сочетая обратную кавычку и запятую, можно строить шаблоны списков:

```
> (setf a 1 b 2)
2
> '(a is ,a and b is ,b)
(A IS 1 AND B IS 2)
```

Использование обратной кавычки вместо `list` позволяет записывать определения макросов в таком же виде, каким будет результат его раскрытия. К примеру, `nil!` может быть определен следующим образом:

```
(defmacro nil! (x)
  `(setf ,x nil))
```

Запятая с собакой действует похожим образом, но вклеивает свой аргумент (который должен быть списком) в выражение. Элементом шаблона, содержащего запятую с собакой, становится не сам ее аргумент, а все его элементы:

```
> (setf lst '(a b c))
(A B C)
> `(lst is ,lst)
(LST IS (A B C))
> `(its elements are ,@lst)
(ITS ELEMENTS ARE A B C)
```

Такая возможность полезна для представления параметра `rest` в определении макросов. Представим макрос `while`Ë034in, который вычисляет свои аргументы до тех пор, пока проверочное выражение остается истинным:

```
> (let ((x 0))
    (while (< x 10)
      (princ x)
      (incf x)))
0123456789
NIL
```

Определить такой макрос можно с помощью параметра `rest`, который соберет все выражения тела макроса в список, который затем будет встроен в результат его раскрытия:

```
(defmacro while (test &rest body)
  `(do ()
       ()
       ((not ,test))
       ,@body))
```

10.4. Пример: быстрая сортировка

На рис. 10.1 приведен пример[1] функции, полностью построенной на макросах, - быстрая сортировка (quicksort) вектора.°Ё094in Алгоритм выполняется следующим образом:

```
(defun quicksort (vec l r)
  (let ((i l)
        (j r)
        (p (svref vec (round (+ l r) 2))))    ; 1
    (while (<= i j)                           ; 2
      (while (< (svref vec i) p) (incf i))
      (while (> (svref vec j) p) (decf j))
      (when (<= i j)
        (rotatef (svref vec i) (svref vec j))
        (incf i)
        (decf j)))
    (if (>= (- j l) 1) (quicksort vec l j))    ; 3
    (if (>= (- r i) 1) (quicksort vec i r)))
  vec)
```

Рис. 10.1. Быстрая сортировка.

1. Один из элементов выбирается в качестве базисного. Часто берется элемент из середины вектора.
2. Затем, переставляя местами элементы, производится разбиение вектора, так, чтобы все элементы, меньшие базисного, находились по левую сторону от больших или равных ему.
3. Наконец, хотя бы одна из частей состоит из двух и более элементов, алгоритм применяется рекурсивно к каждому из таких сегментов.

С каждым последующим вызовом сортируемый список становится короче, и так до тех пор, пока весь список не будет отсортирован.

Реализация этого алгоритма (см. рис. 10.4) требует указания двух чисел, задающих диапазон сортировки. Элемент, находящийся в его середине, выбирается с помощью `pivot` (`p`). Затем, по мере продвижения вдоль вектора к окончанию диапазона, производятся перестановки его элементов, которые либо слишком большие, чтобы находиться слева, либо слишком малые, чтобы находиться справа. (Перестановка двух элементов осуществляется с помощью `rotatef`.) Далее эта процедура повторяется рекурсивно для каждой полученной части, содержащей более одного элемента.

Помимо макроса `while`, определенного выше, в `quicksort` используются встроенные макросы `when`, `incf`, `decf`, `rotatef`. Использование данных макросов существенно упрощает код, делая его аккуратным и понятным.

10.5. Разработка макросов

Написание макросов отличается от обычных программистских задач и требует особого подхода. Умение изменять поведение компилятора отчасти похоже на возможность изменять сам компилятор. Получается, что разработка макросов дает вам возможность встать на место разработчика языков.

В данной главе будут рассмотрены основные трудности, связанные с написанием макросов, и методики их разрешения. В качестве примера мы создадим макрос `ntimes`Ё036in, вычисляющий свое тело n раз:

[1] Листинг 10.1 содержит слегка исправленный код. Ошибка найдена Джедом Кросби. – *Прим. перев.*

```
> (ntimes 10
    (princ "."))
..........
NIL
```

Ниже приведено некорректное определение такого макроса, иллюстрирующее некоторые вопросы, связанные с разработкой макросов:

```
(defmacro ntimes (n &rest body)              ; wrong
  `(do ((x 0 (+ x 1)))
       ((>= x ,n))
     ,@body))
```

На первый взгляд такое определение кажется вполне корректным. Для приведенного выше примера оно будет работать нормально, однако, будет ломаться аж в двух случаях.

Первой проблемой, с которой могут столкнуться разработчики макросов, является неаккуратная работа с переменными (точнее, захват переменных Ё059in). Переменная, используемая внутри макроса, может уже использоваться в том окружении, куда встраивается раскрытие макроса. Когда макрос вызывается там, где уже существует переменная с таким же именем, результат его выполнения может не соответствовать нашим ожиданиям:

```
> (let ((x 10))
    (ntimes 5
      (setf x (+ x 1)))
    x)
10
```

Мы ожидали, что значение `x` будет увеличено пять раз и в итоге будет равно `15`. Но из-за того, что переменная `x` используется внутри макроса как итерируемая переменная в `do`, `setf` будет увеличивать значение *той* переменной, а не той, которую предполагалось. Данный макровызов будет раскрываться в следующее выражение:

```
(let ((x 10))
  (do ((x 0 (+ x 1)))
      ((> x 5))
    (setf x (+ x 1)))
  x)
```

Обычной практикой является не использовать обычные символы так, где они могут быть доступны извне. Вместо них принято использовать `gensym` (см. раздел 8.4). В связи с тем, что `read` интернирует каждый встреченный символ, `gensym` никаким образом не будет равен (с точки зрения `eql`) любому другому символу, встречающемуся в тексте программы. Переписав наше определение `ntimes` с использованием `gensym` вместо `x`, мы сможем избавиться от вышеописанной проблемы:

```
(defmacro ntimes (n &rest body)              ; wrong
  (let ((g (gensym)))
    `(do ((,g 0 (+ ,g 1)))
         ((>= ,g ,n))
       ,@body)))
```

Тем не менее, данное определение по-прежнему подвержено другой проблеме: повторным вычислениям. Первый аргумент встраивается непосредственно в вызов `do`, поэтому он будет вычисляться на каждой итерации. Эта ошибка ясно объясняется примером, в котором первое выражение содержит побочные эффекты:

```
> (let ((v 10))
    (ntimes (setf v (- v 1))
      (princ ".")))
….
NIL
```

Исходя из начального значения `v` равного десяти, мы ожидали девять точек, однако получили пять.

Чтобы разобраться с причиной такого эффекта, посмотрим на результат раскрытия макроса:

```
(let ((v 10))
  (do ((#:g1 0 (+ #:g1 1)))
      ((>= #:g1 (setf v (- v 1))))
    (princ ".")))
```

На каждой итерации мы сравниваем значение переменной (`gensym` при печати предваряется `#:`) не с числом `9`, а с выражением, уменьшающимся на единицу при каждом вызове.

Избежать нежелательных повторных вычислений можно, выделив переменную для значения выражения. Здесь нам снова помогает `gensym`:

```
(defmacro ntimes (n &rest body)
  (let ((g (gensym))
        (h (gensym)))
    `(let ((,h ,n))
       (do ((,g 0 (+ ,g 1)))
           ((>= ,g ,h))
         ,@body))))
```

Эта версия `ntimes` является полностью корректной.

Непреднамеренные повторные вычисления и захват переменных – основные проблемы, возникающие при разработке макросов. Есть и другие. С опытом видеть ошибки в макросах станет так же легко и естественно, как предупреждать деление на ноль. Однако, макросы наделяют нас новыми возможностями, и сложности, с которыми придется столкнуться, тоже новы.

Ваша реализация Common Lisp содержит множество примеров, на которых вы можете поучиться разработке макросов. Раскрывая вызовы встроенных макросов, вы сможете понять их устройство. Ниже приведен результат раскрытия `cond` для большинства реализаций:

```
> (pprint (macroexpand-1 '(cond (a b)
                                (c d e)
                                (t f))))

(IF A
    B
    (IF C
        (PROGN D E)
        F))
```

Функция `pprint`, печатающая выражения с необходимыми отступами, сделает выводимый код легкочитаемым.

10.6. Обобщенный доступ

Так как макровызов раскрывается непосредственно в том месте кода, где вызван, любой макровызов, раскрывающийся в выражение, которое может является первым аргументом `setf`, может сам быть первым аргументом `setf`. Определим, например, синоним `car`:

```
(defmacro cah (lst) `(car ,lst))
```

Так как выражение с `car` может быть первым аргументом `setf`, то и аналогичный вызов с `cah` также может иметь место:

```
> (let ((x (list 'a 'b 'c)))
    (setf (cah x) 44)
    x)
(44 B C)
```

Написание макроса, который самостоятельно раскрывается в вызов `setf`, - менее тривиальная задача, несмотря на кажущуюся простоту. На первый взгляд можно определить `incf`, ну, например, так:

```
(defmacro incf (x &optional (y 1))
```

```
    `(setf ,x (+ ,x ,y)))
   x)
(44 B C)
```

Более тонким вопросом является написание макроса, раскрывающегося в вызов `setf`, решение этой задачи не лежит на поверхности. Может показаться, например, что `incf` можно определить следующим образом:

```
(defmacro incf (x &optional (y 1))          ; wrong
  `(setf ,x (+ ,x ,y)))
```

Такое определение не работает, а следующие выражения не эквивалентны:

```
(setf (car (push 1 lst)) (1+ (car (push 1 lst))))

(incf (car (push 1 lst)))
```

Если `lst` исходно `nil`, то второе выражение вернет список `(2)`, тогда как первое вернет `(1 2)`.

Common Lisp предоставляет `define-modify-macro` – способ решение подобной проблемы для ограниченного набора макросов с `setf`. Вам необходимо переть ему три аргумента: имя макроса, его собственные параметры (место, подлежащее изменению, должно быть первым в списке параметров) и имя функции, приводящей к изменению значение заданного места. Теперь мы можем определить `incf`:

```
(define-modify-macro our-incf (&optional (y 1)) +)
```

А вот версия `push` для добавления элемента в конец списка:

```
(define-modify-macro append1f (val)
  (lambda (lst val) (append lst (list val))))
```

Последний макрос работает следующим образом:

```
> (let ((lst '(a b c)))
    (append1f lst 'd)
    lst)
(A B C D)
```

Макросы `push` и `pop` не могут быть определены через `define-modify-macro`, так как в `push` изменяемое место находится не первым аргументом, а значение, возвращаемое `pop`, не является модифицируемым объектом.

10.7. Пример: утилиты на макросах

Концепция утилит, операторов общего назначения, была объяснена в разделе 6.4. С помощью макросов мы в состоянии создать такие утилиты, которые не смогли бы создать как функций. Несколько подобных примеров мы уже видели: `nil!`, `ntimes`, `while`. Все эти макросы позволяют управлять процессом вычисления аргументов. В этом разделе вы найдете больше примеров утилит-макросов. На рис. 10.2 приведена выборка макросов, которые на практике доказали свое право на существование.

```
(defmacro for (var start stop &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))

(defmacro in (obj &rest choices)
  (let ((insym (gensym)))
    `(let ((,insym ,obj))
       (or ,@(mapcar #'(lambda (c) `(eql ,insym ,c))
                     choices)))))

(defmacro random-choice (&rest exprs)
  `(case (random ,(length exprs))
     ,@(let ((key -1))
         (mapcar #'(lambda (expr)
                     `(,(incf key) ,expr))
                 exprs))))

(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))

(defmacro with-gensyms (syms &body body)
  `(let ,(mapcar #'(lambda (s)
                     `(,s (gensym)))
                 syms)
     ,@body))

(defmacro aif (test then &optional else)
  `(let ((it ,test))
     (if it ,then ,else)))
```

Рис. 10.2. Утилиты на макросах.

Первый из них, `for`, по устройству напоминает `while` (см. стр. Ё034out). Его тело вычисляется в цикле, каждый раз для нового значения переменной:

```
> (for x 1 8
    (princ x))
12345678
NIL
```

Выглядит несомненно проще, чем аналогичное выражение с `do`:

```
(do ((x 1 (1+ x)))
    ((> x 8))
  (princ x))
```

Результат раскрытия выражения с `for` будет очень похож на выражение с `do`:

```
(do ((x 1 (1+ x))
     (#:g1 8))
    ((> x #:g1))
  (princ x))
```

Макрос использует дополнительную переменную для хранения верхней границы цикла. В противном случае, выражение 8 вычислялось бы каждый раз, а для более сложного случая это нежелательно. Дополнительная переменная создается с помощью `gensym`, чтобы к ней невозможно было предотвратить непреднамеренный захват переменной.

Второй макрос (см. рис. 10.2), `in`, возвращает истину, если его первый аргумент равен (`eql`) хотя бы одному из остальных своих аргументов. Без этого макроса вместо выражения:

```
(in (car expr) '+ '- '*)
```

нам пришлось бы писать:

```
(let ((op (car expr)))
  (or (eql op '+)
      (eql op '-)
      (eql op '*)))
```

Разумеется, первое выражение раскрывается в подобное этому, за исключением использования `gensym` вместо переменной `op`.

Следующий пример, `random-choice`, случайным образом выбирает один из своих аргументов и вычисляет его. С задачей случайного выбора мы уже сталкивались (см. стр. Ё035out). Макрос `random-choice` предлагает обобщенный механизм. Вызов типа:

```
(random-choice (turn-left) (turn-right))
```

раскрывается в:

```
(case (random 2)
  (0 (turn-left))
  (1 (turn-right)))
```

Следующий макрос, `with-gensyms`, предназначен в первую очередь для использования внутри других макросов. Его применение вполне ожидаемо в макросах, которые вызывают `gensym` для нескольких переменных. С его помощью вместо:

```
(let ((x (gensym)) (y (gensym)) (z (gensym)))
  ...)
```

мы можем написать:

```
(with-gensyms (x y z)
  ...)
```

Таким образом, ни один из приведенных выше макросов не мог быть написан в виде функции. Из этого вытекает правило: макрос должен создаваться лишь при невозможности создания аналогичной функции. Однако, из этого правила бывают исключения. Например, оператор может быть определен как макрос, чтобы имел возможность выполнять какую-либо работу на этапе компиляции. Примером может служить макрос `avg`Ё042in, вычисляющий среднее значение своих аргументов:

```
> (avg 2 4 8)
14/3
```

Аналогичная функция будет иметь вид:

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))
```

Эта функция вынуждена искать количество аргументов в процессе исполнения. Если нам вдруг захочется избежать этого, почему бы не вычислять длину списка аргументов на этапе компиляции?

На примере последнего макроса (см. рис. 10.2), `aif`, продемонстрирован преднамеренный захват переменной извне. Он позволяет ссылаться через переменную `it` на значение, возвращаемое тестовым аргументом `if`. Таким образом, вместо:

```
(let ((val (calculate-something)))
  (if val
      (1+ val)
      0))
```

мы можем написать:

```
(aif (calculate-something)
     (1+ it)
     0)
```

Разумное использование захвата переменных в макросах может оказать полезную услугу. Сам Common Lisp прибегает к такой возможности в нескольких местах, например, в `next-method-p`Ё045in и `call-next-method`.

Разобранные в этом разделе макросы демонстрируют смысл фразы «программы, которые пишут программы». Однажды определенный `for` позволяет избежать написания множества однотипных выражений с `do`. Есть ли смысл в подобном сокращении кода, подлежащего набору? Однозначно. Ровно по этой же причине мы пользуемся языками программирования, заставляя компиляторы самостоятельно генерировать машинный код, вместо того чтобы набирать его вручную. Макросы привносят это идею в высокоуровневый язык, давая преимущество, подобное выгоде использования высокоуровневого языка по сравнению с машинным кодом. Разумное применение макросов позволит существенно сократить размер вашей программы, облегчая тем самым ее написание, понимание и поддержку.

Если вы еще сомневаетесь, представьте ситуацию, когда недоступны встроенные макросы, и код в который они раскрывались бы, необходимо набирать вручную. Теперь подойдите к вопросу с другой стороны. Представьте, что вы пишете программу, содержащую множество сходных частей. Задайте себе вопрос: «А не пишу ли я результаты раскрытия макросов?» Если так, то макросы, генерирующие этот код, окажут вам большую пользу.

10.8. On Lisp

Теперь, когда мы узнали, что такое макросы, мы видим, что большая часть самого Лиспа написана на самом Лиспе с помощью макросов. Большинство операторов в Common Lisp являются макросами, и лишь 25 из них являются специальными операторами.

Джон Фодераро назвал Лисп «программируемым языком программирования».°Ё095in С помощью макросов Лисп может быть изменен до неузнаваемости и даже превращен в любой другой язык. (Подобный пример вы найдете в главе 17.) Каким бы ни был наилучший путь написания программы, можете быть уверены, что сможете подогнать Лисп под него.

Макросы являются основой всей гибкости Лиспа, позволяя не просто изменять язык до неузнаваемости, но и делать это эффективно. Интерес к макросам внутри Лисп-сообщества не падает со временем. Очевидно, что с их помощью можно сделать еще больше, чем делается сейчас. Вами, например, если захотите. В руках программиста Лисп непрерывно эволюционирует. Именно поэтому он до сих пор жив.

Итоги главы

1. Вызов `eval` – один из способов заставить Лисп считать списки кодом. Однако, его использование неэффективно и нежелательно.
2. Чтобы определить макрос, вам необходимо описать то, во что он будет раскрываться. В некотором смысле, макросы – это функции, возвращающие выражения.
3. Тело макроса, определенное с помощью обратной кавычки, соответствует результату его раскрытия.
4. Разработчик макроса должен осознавать суть захвата переменных и повторных вычислений. Чтобы протестировать макрос, полезно изучить результат его раскрытия.
5. Проблема повторных вычислений возникает в большинстве макросов, раскрывающихся в вызовы `setf`.
6. Макросы более гибки по сравнению с функциями и позволяют создавать более широкий круг утилит. Вы даже можете извлекать пользу из возможности захвата переменных.
7. Лисп дожил до настоящих дней лишь потому, что эволюционирует в руках программиста. Это возможно благодаря в первую очередь макросам.

Упражнения

1. Пусть `x = a`, `y = b`, `z = (c d)`. Запишите выражения с обратной кавычкой, содержащее только заданные переменные (`x`, `y` и `z`) и приводящее к следующим результатам:

```
(a) ((C D) A Z)
(b) (X B C D)
(c) ((C D A) Z)
```

2. Определите `if` через `cond`.

3. Определите макрос, аргументами которого являются число `n` и следующие за ним произвольные выражения. Макрос должен возвращать значение `n`-ного выражения:

```
> (let ((n 2))
    (nth-expr n (/ 1 0) (+ 1 2) (/ 1 0)))
3
```

4. Определите `ntimes` (см. стр. Ё036out), раскрывающийся в (локальную) рекурсивную функцию вместо вызова `do`.

5. Определите макрос `n-of`, принимающий на вход число и выражение. Макрос должен возвращать список значений, полученных последовательным вычислением этого выражения для возрастающего `n` раз значения переменной:

```
> (let ((i 0) (n 4))
    (n-of n (incf i)))
(1 2 3 4)
```

6. Определите макрос, принимающий список переменных и тело, содержащее код. Макрос должен убеждаться, что переменные принимают исходные значения по выполнении кода.

7. Что не так со следующим определением `push`?

```
(defmacro push (obj lst)
  `(setf ,lst (cons ,obj ,lst)))
```

   Приведите пример ситуации, в которой данный код будет работать не так, как настоящий `push`.

8. Определите макрос, удваивающий свой аргумент:

```
> (let ((x 1))
    (double x)
    x)
2
```

11. CLOS

Объектная система Common Lisp (Common Lisp Object System, CLOS) представляет собой набор операторов объектно-ориентированного программирования. Всех их объединяет вместе историческое прошлое.°Ё096in С технической точки зрения они ничем не отличаются от остального Common Lisp: `defmethod` – такая же составная его часть (и совсем небольшая), как и `defun`.

11.1. Объектно-ориентированное программирование

Объектно-ориентированное программирование предлагает изменить само устройство программ, так же, как меняется их устройство при использовании распределенных вычислений. В 1970 году под многопользовательской системой понимался один (или несколько) мейнфрейм, с которым соединялись множество терминалов. Теперь под этим термином принято понимать большой набор рабочих станций, объединенных в сеть, что позволяет распределить вычислительные мощности.

Объектно-ориентированное программирование ломает традиционное устройство программ похожим образом. Вместо реализации одной программы, управляющей всеми данными, данные сами определяют поведение программы.

Представим, например, программу, вычисляющую площадь разнообразных двумерных фигур. Традиционный подход предполагает наличие функции, которая выясняет тип объекта и определяет соответствующее поведение. Пример такой функции приведен на рис. 11.1.

```
(defstruct rectangle
  height width)

(defstruct circle
  radius)

(defun area (x)
  (cond ((rectangle-p x)
         (* (rectangle-height x) (rectangle-width x)))
        ((circle-p x)
         (* pi (expt (circle-radius x) 2)))))

> (let ((r (make-rectangle)))
    (setf (rectangle-height r) 2
          (rectangle-width  r) 3)
    (area r))
6
```

Рис. 11.1. Представление площадей через структуры и функции.

С помощью CLOS[1] мы можем переписать код в другом ключе, как показано на рис. 11.2. В объектно-ориентированном подходе программа разбивается на несколько *методов*, каждый для соответствующего типа аргумента. Два метода, показанные на рис. 11.2, неявно определяют функцию `area`, которая будет вести себя так же, как аналогичная функция на рис. 11.1. При вызове `area` Лисп вызывает определенный метод в соответствии с типом аргумента.

[1] Данная глава построена следующим образом: сначала приводится несложный пример без подробных пояснений, а ряд терминов употребляется без объяснения, после чего последовательно рассматривается каждый аспект. – *Прим. перев.*

```
(defclass rectangle ()
  (height width))

(defclass circle ()
  (radius))

(defmethod area ((x rectangle))
  (* (slot-value x 'height) (slot-value x 'width)))

(defmethod area ((x circle))
  (* pi (expt (slot-value x 'radius) 2)))

> (let ((r (make-instance 'rectangle)))
    (setf (slot-value r 'height) 2
          (slot-value r 'width)  3)
    (area r))
6
```

Рис. 11.2. Представление площадей через классы и методы.

Помимо разбиения функций на методы, объектная ориентированность предполагает *наследование*, как для слотов, так и для методов. Пустой список в `defclass` – список суперклассов (родительских классов). Определим теперь новый класс окрашенных объектов, а затем класс окрашенных кругов, имеющий два суперкласса: `circle` и `colored`:

```
(defclass colored ()
  (color))

(defclass colored-circle (circle colored)
  ())
```

Создавая экземпляры `colored-circle`, мы используем два вида наследования:

1. Экземпляры `colored-circle` будут иметь два слота: `radius`, наследуемый из класса `circle`, и `color` из класса `colored`.
2. Так как для класса `colored-circle` не определено своего метода, вызов `area` для данного класса будет использовать метод, определенный для класса `circle`.

С практической точки зрения под объектно-ориентированным программированием подразумевается способ организации программы в терминах методов, классов, экземпляров и наследования. Почему может понадобиться подобная организация? В числе прочего, объектная ориентированность претендует на упрощение модификации программы. Если нам потребуется изменить какое-либо свойство класса, нам достаточно внести соответствующие изменения лишь в один метод. Чтобы уточнить свойство класса для частного случая, мы создадим подкласс, для которого будут заданы некоторые особенности, а все остальное унаследуется от родительского класса. Для внесения изменений в грамотно написанную программу, нам не обязательно даже смотреть на остальной код, не подлежащий модификации.°Ё097in

11.2. Классы и экземпляры

При создании структур (см. раздел 4.6) мы проходили два этапа: с помощью `defstruct` создавали определение структуры, затем с помощью специально определенной функции `make-point` создавали саму структуру. Создание экземпляров требует двух аналогичных шагов. Для начала определяем *класс* с помощью `defclass`:

```
(defclass circle ()
  (radius center))
```

Только что мы определили класс `circle`, который имеет два *слота* (по аналогии с полями структуры), названные `radius` и `center`.

Чтобы создать экземпляр этого класса, вместо вызова специфичной функции мы воспользуемся функцией `make-instance`, общей для всех классов:

```
> (setf c (make-instance 'circle))
#<Circle #XC27496>
```

Доступ к значению слота можно получить с помощью `slot-value`. Новое значение можно установить с помощью `setf`:

```
> (setf (slot-value c 'radius) 1)
1
```

Как и для структур, неинициализированные значения не определенны.

11.3. Свойства слота

Третий аргумент `defclass` должен содержать список определений слотов. Простейшим определением слота, как в нашем примере, служит символ, представляющий имя слота. В общем случае определение может быть списком, содержащим имя и набор свойств, сообщаемых как ключевые параметры.

Параметр `:accessor` неявно создает функцию, обеспечивающую доступ к слоту, отменяя необходимость использовать `slot-value`. Обновленное определение:

```
(defclass circle ()
  ((radius :accessor circle-radius)
   (center :accessor corcle-center)))
```

класса позволит ссылаться на слоты с помощью функций `circle-radius` и `circle-center`:

```
> (setf c (make-instance 'circle))
#<Circle #XC5C726>
> (setf (circle-radius c) 1)
1
> (circle-radius c)
1
```

С помощью параметров `:writer` и `:reader` вместо `:accessor` можно задать по отдельности функцию установки значения слота и доступа к нему, или же только функцию доступа.

Исходное значение слота задается параметром `:initform`. Чтобы его можно было задать при вызове `make-instance`, необходим параметр `:initarg`.[1] Определение класса, обладающего перечисленными свойствами, будет выглядеть так:

```
(defclass circle ()
  ((radius :accessor circle-radius
           :initarg :radius
           :initform 1)
   (center :accessor circle-center
           :initarg :center
           :initform (cons 0 0))))
```

Теперь вновь созданный экземпляр будет иметь установленные через `:initform` значения слотов, если с помощью `:initarg` не установлено другое значение:

```
> (setf c (make-instance 'circle :radius 3))
#<Circle #XC2DE0E>
> (circle-radius c)
3
> (circle-center c)
(0 . 0)
```

Обратите внимание, что `:initarg` имеет превосходство перед `:initform`.

[1] В качестве имен используются, как правило, ключевые слова, хотя это не является обязательным требованием.

Слоты могут быть *разделяемыми* (*shared*), то есть, имеющими одинаковое значение для каждого экземпляра. Сделать слот разделяемым можно декларацией `:allocation :class`. (Другой возможный вариант - `:allocation :instance`, однако, это значение используется по умолчанию и задавать его нет смысла.) Изменение значения такого слота в одном экземпляре приведет к изменению его во всех остальных экземплярах данного класса. Использование разделяемых параметров обосновано в том случае, когда класс имеет свойство, одинаковое для всех его экземпляров.

Представьте задачу наполнения содержимым желтой прессы. Появление новой темы в одном таблоиде тут же приведет к появлению ее и в остальных. Сравнивая желтую прессу с классом, соответствующий теме слот является разделяемым. Определим класс `tabloid`:

```
(defclass tabloid ()
  ((top-story :accessor tabloid-story
              :allocation :class)))
```

Если какая-та тема попадает на первую страницу одного «экземпляра желтой прессы», она тут же попадает на первую страницу другого:

```
> (setf daily-blab       (make-instance 'tabloid)
        unsolicited-mail (make-instance 'tabloid))
#<Tabloid #XC2AB16>
> (setf (tabloid-story daily-blab) 'adultery-of-senator)
ADULTERY-OF-SENATOR
> (tabloid-story unsolicited-mail)
ADULTERY-OF-SENATOR
```

Необязательный параметр `:documentation` должен быть строкой, которая служит пояснением к слоту. Задавая `:type`, вы обязуетесь поставлять слоту лишь значения заданного типа. Декларации типов будут рассмотрены в разделе 13.3.

11.4. Суперклассы

Второй аргумент `defclass` является списком *суперклассов*. От них класс наследует набор слотов. Так, если мы определим класс `screen-circle`, являющийся подклассом `circle` и `graphic`:

```
(defclass graphic ()
  ((color   :accessor graphic-color   :initarg :color)
   (visible :accessor graphic-visible :initarg :visible
            :initform t)))

(defclass screen-circle (circle graphic)
  ())
```

Экземпляры `screen-circle` будут иметь четыре слота, по два от каждого класса-родителя. Вам не требуется указывать их вручную в определении класса. Сам класс `screen-circle` создается лишь для объединения возможностей двух классов.

Параметры слотов (`:accessor`, `:initarg` и другие) наследуются вместе с самими слотами и работают так, как если бы использовались для экземпляров `circle` или `graphic`:

```
> (graphic-color (make-instance 'screen-circle
                                :color 'red :radius 3))
RED
```

Тем не менее, в определении класса можно указывать некоторые параметры для наследуемых слотов, например `:initform`:

```
(defclass screen-circle (circle graphic)
  ((color :initform 'purple)))
```

Теперь экземпляры `screen-circle` будут пурпурными по умолчанию:

```
> (graphic-color (make-instance 'screen-color))
PURPLE
```

11.5. Предшествование

Мы познакомились с ситуацией, когда классы могут иметь несколько суперклассов. Если метод с одинаковым именем определен для нескольких суперклассов, Лисп должен выбрать один из них, руководствуясь правилом *предшествования*.

Для каждого класса существует *список предшествования*: порядок следования классов, включая сам класс, от наиболее специфичного до наименее специфичного. До сих пор определения порядка предшествования было тривиальным, однако, эта задача может усложниться в больших программах. Вот пример более сложной иерархии классов:

```
(defclass sculpture () (height width depth))
(defclass statue (sculpture) (sybject))
(defclas metalwork () (metal-type))
(defclass casting (metalwork) ())
(defclass cast-statue (statue casting) ())
```

Граф, представляющий класс `case-statue` и его суперклассы, изображен на рис. 11.3.

Рис. 11.3. Иерархия классов.

Чтобы построить такой граф, необходимо начать с узла, соответствующего классу, и двигаться снизу-вверх. Ребра, направленные вверх, связывают класс с прямыми суперклассами, которые располагаются слева-направо в соответствии с порядком соответствующих вызовов `defclass`. Эта процедура повторяется до тех пор, пока для каждого класса будет существовать лишь один суперкласс – `standard-object`. Это справедливо для классов, второй аргумент в определении которых был `()`. Узел, соответствующий каждому такому классу, связан с узлом `standard-object`, а он, в свою очередь – с классом `t`. Проделав подобные операции, мы получим граф, изображенный на рис. 11.3.

Теперь мы можем составить список предшествования:

1. Начинаем снизу.
2. Движемся вверх, выбирая левое из ранее непройденных ребер.
3. Ё044inЕсли обнаруживается, что мы только что пришли к узлу, к которому ведет ребро справа, то отматываем все перемещения до тех пор, пока не вернемся в узел с другими непройденными ребрами. Вернувшись, переходим к шагу 2.
4. Завершаем по достижении `t`. Порядок, с которым каждый узел был посещен впервые, определяет список предшествования.

В пункте 3 упоминается важный момент: ни один класс не может находиться в списке предшествования до любого из его подклассов.

Стрелки на рис. 11.3 показывают направление обхода. Список предшествования нашей иерархии будет следующим: `cast-statue`, `statue`, `sculpture`, `casting`, `metalwork`, `standard-object`, `t`. Иногда положение класса в таком списке называют его специфичностью. Классы расположены в нем по убыванию специфичности.

По списку предшествования можно определять, какой метод необходимо применить. Этот процесс рассматривается в следующем разделе. Порядок следования, связанный с выбором наследуемого слота, нами пока не рассматривался. Найти соответствующие правила его определения вы сможете на стр. Ё037out.°Ё098in

11.6. Обобщенные функции

Обобщенной функцией называют функцию, построенную из одного или нескольких методов. Методы определяются через оператор `defmethod`, работающий похожим на `defun` образом:

```
(defmethod combine (x y)
  (list x y))
```

На настоящий момент `combine` имеет один метод, и ее вызов построит список двух аргументов:

```
> (combine 'a 'b)
(A B)
```

Пока мы не сделали чего-либо, с чем не справилась бы обычная функция. Однако, отличия станут заметны при добавлении еще одного метода.

Для начала создадим несколько классов, с которыми будут работать наши методы:

```
(defclass stuff () ((name :accessor name :initarg :name)))
(defclass ice-cream (stuff) ())
(defclass topping (stuff) ())
```

Мы определили три класса: `stuff`, имеющий слот `name`, а также `ice-cream` и `topping`, являющиеся подклассами `stuff`.

Теперь создадим второй метод:

```
(defmethod combine ((ic ice-cream) (top topping))
  (format nil "~A ice-cream with ~A topping."
              (name ic)
              (name top)))
```

Приведенный метод может применяться лишь к определенным аргументам. Первым аргументом должен быть экземпляр `ice-cream`, а вторым – экземпляр `topping`.

Каким образом Лисп определяет, какой метод вызвать? Из тех, что удовлетворяют ограничениям на классы, будет выбран наиболее специфичный. Это значит, что для аргументов, представляющих классы `ice-cream` и `topping` соответственно, будет применен последний определенный нами метод:

```
> (combine (make-instance 'ice-cream :name 'fig)
           (make-instance 'topping :name 'treacle))
"FIG ice-cream with TREACLE topping."
```

Для любых других аргументов будет применен наш первый метод:

```
> (combine 23 'skiddoo)
(23 SKIDDOO)
```

Ни один из аргументов этого метода не был конкретизирован, поэтому он имеет наименьший приоритет и будет вызван лишь, когда ни один другой метод не подойдет. Подобные методы играют роль ключа `otherwize` в case-выражении.

Может быть конкретизирована любая комбинация параметров. В следующем методе ограничивается лишь первый аргумент:

```
(defmethod combine ((ic ice-cream) x)
  (format nil "~A ice-cream with ~A."
          (name ic)
          x))
```

Если мы теперь вызовем `combine` с экземплярами `ice-cream` и `topping`, из двух последних методов будет выбран наиболее специфичный:

```
> (combine (make-instance 'ice-cream :name 'grape)
           (make-instance 'topping :name 'marshmallow))
"GRAPE ice-cream with MARSHMALLOW topping."
```

Если второй аргумент не принадлежит классу `topping`, будет вызван только что определенный метод:

```
> (combine (make-instance 'ice-cream :name 'clam)
           'reluctance)
"CLAM ice-cream with RELUCTANCE."
```

Аргументы обобщенной функции определяют набор *применимых* (*applicable*) методов. Метод считается применимым, если аргументы соответствуют спецификациям, налагаемым данным методом.

Отсутствии применимых методов приводит к ошибке. Если применим лишь один метод, он вызывается. Если применимо несколько методов, то наиболее специфичный выбирается в соответствии с порядком предшествования. Параметры

сверяются слева направо. Если первый параметр одного из применимых методов ограничен более специфичным классом, чем первый аргумент других методов, то он и считается наиболее специфичным. Эти ограничения ломаются при переходе к следующему параметру, и так далее.[1]

В предыдущих примерах легко найти наиболее специфичный применимый метод, так как все объекты модно расположить по убыванию: экземпляр `ice-cream` принадлежит классам `ice-cream`, затем `stuff`, `standard-object` и, наконец, `t`.

Методы не обязаны использовать спецификаторы, являющиеся классами, определенными через `defclass`. Они могут использовать также обычные типы (точнее, классы, соответствующие встроенным типам). Вот пример с числовыми спецификаторами:

```
(defmethod combine ((x number) (y number))
  (+ x y))
```

Кроме того, методы могут использовать отдельные объекты, сверяемые с помощью `eql`:

```
(defmethod combine ((x (eql 'powder)) (y (eql 'spark)))
  'boom)
```

Спецификаторы индивидуальных объектов имеют больший приоритет, нежели спецификаторы классов.

Списки аргументов методов могут быть довольно сложны, как и списки аргументов обычных функций, однако должны быть *конгруэнтны* аргументам соответствующей обобщенной функции. Метод должен иметь столько же обязательных и необязательных параметров, сколько и обобщенная функция, использовать либо `&rest`, либо `&key`, но не одновременно. Все приведенные ниже пары конгруэнтны:

```
(x) (a)
(x &optional y) (a &optional b)
(x y &rest z) (a b &key c)
(x y &key z) (a b &key c d)
```

а эти пары – нет:

```
(x) (a b)
(x &optional y) (a &optional b c)
(x &optional y) (a &rest b)
(x &key x y) (a)
```

Лишь обязательные параметры могут быть специализированы. Таким образом, метод полностью определяется именем и спецификациями обязательных параметров. Если мы создадим еще один метод с тем же именем и спецификациями, он заменит предыдущий. То есть, установив:

```
(defmethod combine ((x (eql 'powder)) (y (eql 'spark)))
  'kaboom)
```

мы тем самым переопределим поведение `combine` для аргументов `powder` и `spark`.

11.7. Вспомогательные методы

Методы могут дополняться *вспомогательными методами*, включая before- (перед), after- (после) и around-методы (вокруг). Before-методы позволяют нам сказать: «Прежде чем приступить к выполнению, сделайте это». Они вызываются в порядке убывания специфичности, предваряя вызов основного метода. After-методы позволяют сказать: «P.S. А сделайте заодно и это». Они вызываются после выполнения основного метода в порядке увеличения специфичности. То, что выполняется между ними, ранее называлось нами просто методом, но, более точно, является *первичным методом*. В результате вызова возвращается именно его

[1] Мы не сможем дойти до конца аргументов и по-прежнему иметь неотмененные ограничения, потому что тогда два метода будут подходить точно под одно же описание. Это невозможно, так как второе будет определено поверх первого и заменит его.

значение, даже если после него были вызваны after-методы. Before- и after-методы позволяют оборачивать вызов первичного метода. Around-методы позволяют выполнять то же самое, но более радикальным образом. Если задан around-метод, он будет вызван *вместо* первичного. Он вызовет первичный метод самостоятельно и по своему усмотрению (с помощью функции `call-next-method`, которая предназначена именно для этого).

Этот механизм называется *стандартной комбинацией методов*. Вызов обобщенной функции привлекает:

1. Наиболее специфичный around-метод, если задан хотя бы один.
2. В противном случае, по порядку:

   (a) Все before-методы в порядке убывания специфичности.

   (b) Наиболее специфичный первичный метод.

   (c) Все after-методы в порядке возрастания специфичности.

   Возвращаемым значением считается значение around-метода (в случае 1) или значение наиболее специфичного первичного метода (в случае 2).

Вспомогательные методы определяются размещением *квалификатора* (*qualifier*) после имени метода в вызове `defmethod`. Определим первичный метод `speak` для класса `speaker`:

```
(defclass speaker () ())

(defmethod speak ((s speaker) string)
  (format t "~A" string))
```

Вызовем `speak` для экземпляра `speaker` просто напечатает второй аргумент:

```
> (speak (make-instance 'speaker)
         "I'm hungry")
I'm hungry
NIL
```

Определив класс `intellectual`, оборачивающий before- и after-методы вокруг первичного метода `speak`:

```
(defclass intellectual (speaker) ())

(defmethod speak :before ((i intellectual) string)
  (princ "Perhaps "))

(defmethod speak :after ((i intellectual) string)
  (princ " in some sense"))
```

мы можем создать подкласс говорунов, всегда добавляющих в конце (и в начале) несколько слов:

```
> (speak (make-instance 'intellectual)
          "I'm hungry")
Perhaps I'm hungry in some sense
NIL
```

Как было упомянуто выше, вызываются все before- и after-методы. Если мы теперь определим before- и after-методы для суперкласса `speaker`:

```
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

то они будут вызываться из середины нашего сэндвича:

```
> (speak (make-instance 'intellectual)
           "I'm hungry")
Perhaps I think I'm hungry in some sense
NIL
```

Независимо от того, вызываются ли before- и after-методы, при вызове обобщенной функции всегда возвращается значение первичного метода. В нашем случае `format` возвращает `nil`.

Это утверждение не распространяется на around-методы. Они вызываются сами по себе, а все остальные методы вызываются лишь если так решит around-метод. Around- или первичный методы могут вызывать следующий метод с помощью `call-next-method`. Перед этим уместно проверить наличие такого метода с помощью `next-method-p`.

С помощью around-методов вы можете определить другой, более предусмотрительный подкласс `speaker`:

```
(defclass courtier (speaker) ())

(defmethod speak :around ((c courtier) string)
  (format t "Does the King believe that ~A? " string)
  (if (eql (read) 'yes)
           (if (next-method-p) (call-next-method))
           (format t "Indeed, it is a preposterous idea.~%"))
  'bow)
```

Когда первым аргументом `speak` является экземпляр класса `courtier`, языком придворного управляет around-метод:

```
> (speak (make-instance 'courtier) "kings will last")
Does the King believe that kings will last? yes
I think kings will last
BOW
> (speak (make-instance 'courtier) "the world is round")
Does the King believe that the world is round? no
Indeed, it is a preposterous idea.
BOW
```

Еще раз обратите внимание, что, в отличие от before- и after-методов, значением вызова обобщенной функции является значение вызова around-метода.

11.8. Комбинация методов

В стандартной комбинации методов позволяется вызывать лишь один первичный метод, являющийся наиболее специфичным (хотя другие методы могут быть вызваны через `call-next-method`). Несмотря на это, нам хотелось бы иметь возможность комбинировать результаты всех применимых первичных методов.

Помимо стандартной комбинации методов могут быть и другие механизмы, например, возврат суммы всех применимых методов. Под *операторной* комбинацией методов может пониматься случай, когда вызов завершается вычислением некоторого Лисп-выражения, являющегося применением оператора к результатам вызовов применимых первичных методов в порядке убывания их специфичности. Если мы зададимся целью определить обобщенную функцию `price`, комбинирующую значения в функцией `+`, причем для `price` не существует применимых around-методов, то она будет вести себя, как будто определена следующим образом:

```
(defun price (&rest args)
  (+ (apply <most specific primary method> args)
     .
     .
     .
     (apply <least specific primary method> args)))
```

Если существуют применимые around-методы, то они вызываются в порядке очередности как в стандартной комбинации. В операторной комбинации around-метод по-прежнему может использовать `call-next-method`, но уже не может вызывать первичные методы.

Используемый способ комбинации методов может быть задан в момент явного создания обобщенной функции через `defgeneric`:

```
(defgeneric price (x)
  (:method-combination +))
```

Теперь метод price будет использовать + для комбинации методов. Теперь второй аргумент defmethod-определения должен содержать +. Определим некоторые классы с ценами:

```
(defclass jacket () ())
(defclass trousers () ())
(defclass suit (jacket trousers) ())

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)
```

Теперь, если мы запросим цену костюма (suit), мы получим сумму применимых методов price:

```
> (price (make-instance 'suit))
550
```

Параметр :method-combination, передаваемый вторым аргументом defgeneric (а также вторым аргументом defmethod) может быть одним из следующих символов:

```
+ and append list max min nconc or progn
```

Символ standard явно задает использование стандартной комбинации методов.

Определив способ комбинации для обобщенной функции, вы вынуждены использовать этот же способ для всех методов, имеющих то же имя. Попытка использовать другие символы (а также :before и :after) приведет к возникновению ошибки. Чтобы изменить способ комбинации price, вам придется удалить саму обобщенную функцию с помощью fmakunbound.

11.9. Инкапсуляция

Объектно-ориентировнные языки часто предоставляют способ абстрагирования внешнего представления объекта от его реализации. Сокрытие деталей реализации несет двойную выгоду: вы можете модифицировать реализацию объекта, не затрагивая интерфейс доступа к нему извне, а также получаете защищенность объекта от воздействий извне, которые могут привести к нежелательным последствиям. Сокрытие деталей иногда называют *инкапсуляцией*.

Несмотря на то, что инкапсуляцию часто считают свойством объектно-ориентированного программирования, это разные вещи, и одно может существовать без другого. Мы уже знакомились с инкапсуляцией (см. стр. Ё038out) в малом масштабе. Функции stamp и reset используют одну переменную counter, однако для их использования не нужно знать ничего об этой переменной и нет необходимости использовать ее явно.

В Common Lisp основным методом разделения информации на общедоступную и частную являются пакеты. Чтобы ограничить доступ к чему-либо, мы кладем это в пакет, экспортируя из него лишь необходимое.

Для инкапсуляции слота достаточно экспортировать лишь функции доступа к нему, но не само самого слота. Например, мы можем определить класс counter и связать с ним методы increment и clear:

```
(defpackage "CTR"
            (:use "COMMON-LISP")
            (:export "COUNTER" "INCREMENT" "CLEAR"))


(in-package ctr)

(defclass counter () ((state :initfprm 0)))

(defmethod increment ((c counter))
  (incf (slot-value c 'state)))

(defmethod clear ((c counter))
  (setf (slot-value c 'state) 0))
```

Функции вне пакета `ctr` будут иметь возможность создавать экземпляры `counter` и вызывать `increment` и `clear`, однако не будут иметь какой-либо доступ к самому слоту и имени `state`.

Для большего отдаления от внешним и внутренним представлениями вы можете сделать *невозможным* сам доступ к значению слота. Для этого достаточно воспользоваться `unintern` после выполнения кода, который ссылается на удаляемый символ:

```
(unintern 'state)
```

Теперь сослаться на слот невозможно ни из какого пакета.°Ё099in

11.10. Две модели

С объектной ориентированностью связано одна тонкость. Дело в том, что этому понятию могут соответствовать две модели: модель передачи сообщений и модель обобщенных функций. Сначала появилась передача сообщений. Второй метод является обобщением первого.

В модели передачи сообщений методы принадлежат объектам и наследуются так же, как слоты. Чтобы найти площадь объекта, придется послать сообщение для `area`:

```
tell obj area
```

Это сообщение вызывает соответствующий метод, который имеет или наследует `obj`.

Иногда нам приходится сообщать дополнительные аргументы. Например, метод `move` может принимать аргумент, определяющий дальность перемещения. Чтобы переместить `obj` на `10`, пошлем ему следующее сообщение:

```
tell obj move 10
```

В перефразированном виде запись:

```
(move obj 10)
```

проясняет ограничения модели передачи сообщений: мы можем специфицировать лишь первый параметр. В такой модели не только не получится использовать методы для нескольких объектов, но нелегко даже помыслить такую возможность.

В данной модели методы *принадлежат* объектам, в то время как в модели обобщенных функций методы специализируются *для* объектов. В модели передачи сообщений мы можем специализировать лишь один объект, а в модели передачи сообщений – произвольное количество. Таким образом, первая модель – лишь частный случай второй, и с помощью обобщенных функций можно изобразить передачу сообщений.

Итоги главы

1. В объектно-ориентированном программировании функция `f` определяется неявно при определении методов `f` для разных объектов. Объекты наследуют методы от своих родителей.
2. Определение класса напоминает определение структуры, но выглядит более обширно. Разделяемый слот принадлежит всему классу.
3. Класс наследует слоты своих суперклассов.
4. Предки класса упорядочены в соответствии со списком предшествования. Алгоритм предшествования легко понять графически.
5. Обобщенная функция состоит из всех методов с тем же именем. Метод определяется именем и специализацией своих параметров. Метод, используемый при вызове обобщенной функции, определяется на основе порядка предшествования.
6. Первичные методы могут быть дополнены вспомогательными. Стандартная комбинация методов означает использование around-методов, если таковые имеются; в противном случае вызываются сначала before-, потом первичный, затем after-методы.

7. В операторной комбинации методов все первичные методы рассматриваются как аргументы заданного оператора.
8. Инкапсуляция может быть осуществлена с помощью пакетов.
9. Существуют две модели объектно-ориентированного программирования. Модель обобщенных функций является обобщением модели передачи сообщений.

Упражнения

1. Определите параметры `accessor`, `initform`, `initarg` для классов, определенных на рис. 11.2. Внесите необходимые правки, чтобы код более не использовал `slot-value`.
2. Перепишите код на рис. 9.5 так, чтобы сферы и точки были классами, а `introspect` и `normal` – обобщенными функциями.
3. Имеется набор классов:

```
(defclass a (c d) ...)        (defclass e () ...)
(defclass b (d c) ...)        (defclass f (h) ...)
(defclass c () ...)           (defclass g (h) ...)
(defclass d (e f g) ...)      (defclass h () ...)
```

   (a) Нарисуйте граф, представляющий предков `a`, и составьте список классов, которым принадлежит экземпляр `a` в порядке убывания их специфичности.

   (b) Повторите аналогичную процедуру для `b`.
4. Имеются следующие функции:

   - `precedence`: принимает объект и возвращает для него список предшествования классов по убыванию специфичности.
   - `methods`: принимает обобщенную функцию и возвращает список всех ее методов.
   - `specializations`: принимает метод и возвращает список специализаций его параметров. Каждый элемент возвращаемого списка будет либо классом, либо выражением вида `(eql x)`, либо `t` (для неспециализированного параметра).

   С помощью перечисленных функций (не пользуясь `compute-applicable-methods` или `find-method`) определите функцию `most-spec-app-meth`, принимающую обобщенную функцию и список аргументов, с которыми она будет вызвана, и возвращающую наиболее специфичный применимый метод, если таковые имеются.
5. Измените код на рис. 11.2 так, чтобы при каждом вызове обобщенной функции `area` увеличивался некий глобальный счетчик. Функция не должна менять свое поведение никаким иным образом.
6. Приведите пример задачи, трудноразрешимой в том случае, когда лишь первый аргумент обобщенной функции может быть специализирован.

12. Структура

В разделе 3.3 было объяснено, каким образом указатели позволяют нам устанавливать произвольное значение куда угодно. Это допускает ряд вольностей, и не все они могут принести пользу. Например, объект может быть элементом самого себя. Хорошо это или плохо – зависит от конкретной ситуации.

12.1 Разделяемая структура

В общем случае ячейки в Лиспе могут быть разделяемыми.[1] В простейшем случае один список может быть частью другого. После:

```
> (setf part (list 'b 'c))
(B C)
> (setf whole (cons 'a part))
(A B C)
```

первая ячейка является частью (точнее, `cdr`) второй. В подобных случаях принято говорить, что два списка разделяют одну структуру. Структура двух таких списков представлена на рис. 12.1.

Рис. 12.1. Разделяемая структура.

Подобные ситуации ищет предикат `tailp`. Он принимает два списка и возвращает истину, если встретит первый список при обходе второго.

```
> (tailp part whole)
T
```

Мы можем реализовать его самостоятельно:

```
(defun our-tailp (x y)
  (or (eql x y)
      (and (consp y)
           (our-tailp x (cdr y)))))
```

В нашем определении предполагается, что любой список является хвостом самого себя, а `nil` является хвостом любого нормального списка.

В более сложном случае два списка могут разделять общую структуру даже когда один не является хвостом другого. Это происходит, когда они разделяют хвост общего списка, как показано на рис. 12.2. Создадим подобную ситуацию:

Рис. 12.2. Разделяемый хвост.

```
(setf part (list 'b 'c)
      whole1 (cons 1 part)
      whole2 (cons 2 part))
```

Теперь `whole1` и `whole2` разделяют структуру, но один не является хвостом другого.

Имея в наличии вложенные списки, важно различать списки с разделяемой структурой и их элементы с разделяемой структурой. *Структура верхнего уровня списка* (*top-level list structure*) включает ячейки, из которых состоит сам список, и не ссылается на ячейки, образующие их самих. Пример структуры верхнего уровня вложенного списка приведен на рис. 12.3.

Рис. 12.3. Структура списка верхнего уровня.

Имеют ли две ячейки разделяемую структуру, зависит от того, считаем мы их списками или деревьями. Два вложенных списка могут разделять одну структуру как деревья, но не разделять как списки. Следующий код создает ситуацию, изображенную на рис. 12.4, где два списка содержат один и тот же элемент-список:

---

[1] Разделяемый (shared) – принятый термин для обозначения наличия общей части нескольких объектов, а вовсе не разделение одного объекта на части. – *Прим. перев.*

Рис. 12.4. Разделяемое поддерево.

```
(setf element (list 'a 'b)
      holds1  (list 1 element 2)
      holds2  (list element 3))
```

Второй элемент `holds1` разделяет структуру (имеет ту же структуру) с первым элементом `holds2`, но `holds1` и `holds2` как списки не обладают разделяемой структурой. Два списка разделяют структуру *как списки*, только если их общий элемент находится в структуре верхнего уровня.

Чтобы избежать разделения структуры, необходимо подключить копирование. Функция `copy-list`, определяемая как:

```
(defun our-copy-list (lst)
  (if (null lst)
      nil
      (cons (car lst) (our-copy-list (cdr lst)))))
```

вернет список, не разделяющий структуру верхнего уровня с исходным списком. Функция `copy-tree`, которая может быть определена следующим образом:Ё021in

```
(defun our-copy-tree (tr)
  (if (atom tr)
      tr
      (cons (our-copy-tree (car tr))
            (our-copy-tree (cdr tr)))))
```

возвратит список, который не разделяет структуру всего дерева с исходным списком. Рис. 12.5 демонстрирует разницу между вызовом `copy-list` и `copy-tree` для вложенного списка.

Рис. 12.5. Два способа копирования.

12.2. Модификация

Ё048inЧем может мешать разделяемость структуры? До сих пор мы рассматривали это явление всего лишь как забавную головоломку, а в программах, которые мы написали ранее, мы нормально обходились без этого понятия. Дело в том, что изменение одного из двух списков с разделяемой структурой повлечет непреднамеренное изменение другого.

В предыдущем разделе мы видели, как сделать один список хвостом другого:

```
(setf whole (list 'a 'b 'c)
      tail (cdr whole))
```

Изменение списка `tail` повлечет симметричное изменение хвоста `whole`, и наоборот, так как по сути это одна и та же ячейка:

```
> (setf (second tail) 'e)
E
> tail
(B E)
> whole
(A B E)
```

Разумеется, то же самое будет происходить и для двух списков, имеющих общий хвост.

Не всегда может быть ошибочно изменять два объекта одновременно. Иногда это именно то, что нужно. Однако, если это происходит непреднамеренно, то может привести к некорректной работе. Опытные программисты умеют избегать подобных ошибок и распознавать такие ситуации. Если список без видимой причины меняет свое содержимое, вероятно, он имеет разделяемую структуру.

Опасна не разделяемость структуры сама по себе, а возможность ее изменения. Не используя `setf` (а также аналогичные операторы типа `pop`, `rplaca` и другие) для списков, вы гарантируете отсутствие подобных ошибок. Если изменяемость списков все же требуется, то необходимо знать, откуда взялся изменяемый список. Если список было получен в результате разделения структуры, то для сохранения

структуры связанного с ним списка модифицировать необходимо не сам список, а его копию.

Вам необходимо проявлять удвоенную осторожность при использовании функций, написанных не вами. Если вы не знаете, как устроена функция, то должны исходить из предположения, что все, что передается функции:

1. Может быть передано деструктивным операторам.
2. Может быть сохранено где-либо, и изменение этого объекта приведет к изменению другого объекта, использующего данный.[1]

В обоих случаях правильным решением является копирование аргументов.

В Common Lisp вызов любой функции, выполняющей проход по структуре списка (например, `mapcar` или `remove-if`) не изменит его структуру.

12.3. Пример: очереди

Разделяемые структуры – не то явление, с которым нужно бороться. Более того, оно может приносить пользу. В этом разделе показано, как с помощью разделяемых структур представить очереди. Очередь – это хранилище объектов, которые могут туда помещаться по одному и извлекаться в том же порядке. Такую модель принято именовать FIFO – сокращение от "первым пришел, первым ушел" ("first in, first out").

С помощью списков легко представить стек, так как добавление и получение элементов которого происходит с одного конца.[2] Задача представления очередей более сложна. Для эффективной их реализации необходимо обеспечить управление обоими концами.

Одна из возможных стратегий приводится на рис. 12.6. На нем показана очередь из трех элементов: `a`, `b` и `c`. Очередью считаем точечную пару, состоящую из списка и последней ячейки этого списка. Будем называть их "начало" и "конец". Чтобы получить элемент из очереди, необходимо вытолкнуть начало. Чтобы добавить новый элемент, необходимо создать новую ячейку, сделать ее хвостом конца очереди и присвоить ей сам конец.

Рис. 12.6. Структура очереди.

Такую стратегию реализует код на рис. 12.7. Вот пример использования:

```
(defun make-queue () (cons nil nil))

(defun enqueue (obj q)
  (if (null (car q))
      (setf (cdr q) (setf (car q) (list obj)))
      (setf (cdr (cdr q)) (list obj)
            (cdr q) (cdr (cdr q))))
  (car q))

(defun dequeue (q)
  (pop (car q)))
```

Рис. 12.7. Реализация очередей.

```
> (setf q1 (make-queue))
(NIL)
> (progn (enqueue 'a q1)
         (enqueue 'b q1)
         (enqueue 'c q1))
(A B C)
```

[1] Например, в Common Lisp попытка изменения строки, используемой как имя символа, считается ошибкой, и нам известно, что intern не копирует строку перед созданием символа. Исходя их этого мы должны предположить, что модификация любой строки, которая передавалась в `intern`, приведет к ошибке.

[2] Модель, реализуемую стеком, называют так же LIFO – "последним пришел, первым ушел" ("last in, first out"). – *Прим. перев.*

Теперь q1 представляет очередь, изображенную на рис. 12.6:

```
> q1
((A B C) C)
```

Теперь попробуем забрать что-нибудь из очереди:

```
> (dequeue q1)
A
> (dequeue q1)
B
> (enqueue 'd q1)
(C D)
```

12.4. Деструктивные функции

Некоторые функции в Common Lisp все же могут изменять содержимое списков, делая это из соображений эффективности. Им позволено изменять ячейки, поставляемые им в качестве аргументов, однако, их имеет смысл использовать не ради побочных эффектов.

Например, delete является деструктивным аналогом remove. Ей позволено очищать переданный ей список, однако, она не может гарантировать ничего касательно получившейся структуры этого списка. Посмотрим, что происходит в большинстве реализаций:

```
> (setf lst '(a r a b i a))
(A R A B I A)
> (delete 'a lst)
(R B I)
> lst
(A R B I)
```

Как и для remove, чтобы зафиксировать побочный эффект, необходимо использовать setf:

```
(setf lst (delete 'a lst))
```

Примером того, как деструктивные функции модифицируют списки, является nconc, деструктивная версия append.[1] Приведем ее версию для двух аргументов, демонстрирующую, каким образом сшиваются списки:

```
(defun nconc2 (x y)
  (if (consp x)
      (progn
        (setf (cdr (last x)) y)
        x)
      y))
```

cdr последней ячейки списка становится указателем на второй список. Полноценная версия для произвольного числа аргументов приводится в приложении B.

Ё005inФункция mapcan похожа на mapcar, но дополнительно сращивает (с помощью nconc) в один список возвращаемые значения, которые должны быть списками:

```
> (mapcan #'list
          '(a b c)
          '(1 2 3 4))
(A 1 B 2 C 3)
```

Определим нашу собственную функцию:

```
(defun our-mapcan (fn &rest lsts)
  (apply #'nconc (apply #'mapcar fn lsts)))
```

[1] Буква n в имени nconc соответствует "non-consing". Имена многих деструктивных функций начинаются с n.

Используйте mapcar с осторожностью, учитывая ее деструктивный характер. Она сращивает возвращаемые списки, и вы уже не сможете воспользоваться ими по отдельности.

Функция mapcan полезна, в частности, на задачах, интерпретируемых как сбор всех узлов одного уровня в дерево. Например, если children возвращает список чьих-то детей, мы сможем получать и список внуков:

```
(defun grandchildren (x)
  (mapcan #'(lambda (c)
              (copy-list (children c)))
          (children x)))
```

Данная функция применяет copy-list к результату вызова children, так как он может возвращать уже существующий объект, а не производить новый.

Ничего не мешает нам определить и недеструктивный вариант mapcan:

```
(defun mappend (fn &rest lsts)
  (apply #'append (apply #'mapcar fn lsts)))
```

Используя mappend, мы можем избежать вызовов copy-list в определении grandchildren:

```
(defun grandchildren (x)
  (mappend #'children (children x)))
```

12.5. Пример: двоичные деревья поиска

В некоторых ситуациях уместнее использовать деструктивные операции, нежели недеструктивные. В разделе 4.7 было показано, как управлять двоичными деревьями поиска, или BST. Все использованные нами функции были недеструктивными, но в данной задаче это предостережение излишне.

На рис. 12.8 приведена деструктивная версия bst-insert (см. стр. Ё039out). Она принимает точно такие же аргументы и возвращает точно такое же значение. Единственным отличием является то, что она может изменять само дерево, передаваемое вторым аргументом.

```
(defun bst-insert! (obj bst <)
  (if (null bst)
      (make-node :elt obj)
      (progn (bsti obj bst <)
             bst)))

(defun bsti (obj bst <)
  (let ((elt (node-elt bst)))
    (if (eql obj elt)
        bst
        (if (funcall < obj elt)
            (let ((l (node-l bst)))
              (if l
                  (bsti obj l <)
                  (setf (node-l bst)
                        (make-node :elt obj))))
            (let ((r (node-r bst)))
              (if r
                  (bsti obj r <)
                  (setf (node-r bst)
                        (make-node :elt obj))))))))
```

Рис. 12.8. Двоичные деревья поиска: деструктивная вставка.

В разделе 2.12 вы были предупреждены, что деструктивные функции вызываются не ради побочных эффектов. И, разумеется, неплохо было бы иметь возможность вызывать bst-insert! без деструктивных последствий для дерева, как если бы мы вызвали bst-insert:

```
> (setf *bst* nil)
NIL
```

```
> (dolist (x '(7 2 9 8 4 1 5 12))
    (setf *bst* (bst-insert! x *bst* #'<)))
NIL
```

Можно также определить и аналог `push` для BST, но это довольно сложно и выходит за рамки данной книги. (Для любопытных, такой макрос определяется на стр. Ё056out.°Ё100in)

На рис. 12.9[1] представлен деструктивный вариант `bst-delete`, аналогичный `bst-remove` (см. стр. Ё057out) в той же мере, как `delete` по отношению к `remove`. Как и `delete`, она не подразумевает использование ради побочных эффектов. Использовать `bst-delete` необходимо так же, как и `bst-remove`:

```
(defun bst-delete (obj bst <)
  (if (null bst)
      nil
      (if (eql obj (node-elt bst))
          (del-root bst)
          (progn
            (if (funcall < obj (node-elt bst))
                (setf (node-l bst) (bst-delete obj
(node-l bst) <))
                (setf (node-r bst) (bst-delete obj
(node-r bst) <)))
            bst))))

(defun del-root (bst)
  (let ((l (node-l bst)) (r (node-r bst)))
    (cond ((null l) r)
          ((null r) l)
          (t        (if (zerop (random 2))
                        (cutnext r bst nil)
                        (cutprev l bst nil))))))

(defun cutnext (bst root prev)
  (if (node-l bst)
      (cutnext (node-l bst) root bst)
      (if prev
          (progn
            (setf (node-elt root) (node-elt bst)
                  (node-l prev)   (node-r bst))
            root)
          (progn
            (setf (node-l bst)    (node-l root))
            bst))))

(defun cutprev (bst root prev)
  (if (node-r bst)
      (cutprev (node-r bst) root bst)
      (if prev
          (progn
            (setf (node-elt root) (node-elt bst)
                  (node-r prev)   (node-l bst))
            root)
          (progn
            (setf (node-r bst)    (node-r root))
            bst))))

(defun replace-node (old new)
  (setf (node-elt old) (node-elt new)
        (node-l   old) (node-l   new)
        (node-r   old) (node-r   new)))

(defun cutmin (bst par dir)
  (if (node-l bst)
```

[1] Данный листинг содержит исправленную версию `bst-delete`. За подробностями обращайтесь к сноске на стр. Ё019out. – *Прим. перев.*

```
            (cutmin (node-l bst) bst :l)
            (progn
              (set-par par dir (node-r bst))
              (node-elt bst))))

      (defun cutmax (bst par dir)
        (if (node-r bst)
            (cutmax (node-r bst) bst :r)
            (progn
              (set-par par dir (node-l bst))
              (node-elt bst))))

      (defun set-par (par dir val)
        (case dir
          (:l (setf (node-l par) val))
          (:r (setf (node-r par) val))))
```

Рис. 12.9. Двоичные деревья поиска: деструктивное удаление.

```
> (setf *bst* (bst-delete 2 *bst* #'<))
#<7>
> (bst-find 2 *bst* #'<)
NIL
```

12.6. Пример: двусвязные списки

Обычные списки в Лиспе являются односвязными, то есть имеют лишь один указатель на другой объект. *Двусвязные списки* имеют также и обратный указатель, и вы можете перемещаться в обе стороны. В этом разделе показано, как создавать и использовать двусвязные списки.

На рис. 12.10 показана их возможная реализация. `Cons`-ячейки имеют два поля: `car`, указывающий на данные, и `cdr`, указывающий на следующий элемент. Элемент двусвязного списка должен иметь еще одно поле, указывающее на предыдущий элемент. Вызов `defstruct` (см. рис. 12.10) создает трехчастичный объект, называемый `dl` (от «double-linked»). Мы будем использовать его для создания двусвязных списков. Поле `data` в `dl` соответствует `car` в `cons`-ячейке, `next`[1] соответствует `cdr`. Поле `prev` похоже на `cdr`, но соответствует перемещению в обратном направлении. (Пример такой структуры приведен на рис. 12.11). Пустому двусвязному списку, как и обычному, соответствует `nil`.

[1] Исправлена авторская опечатка, обнаруженная Адамом Лэнгли. – *Прим. перев.*

```
(defstruct (dl (:print-function print-dl))
  prev data next)

(defun print-dl (dl stream depth)
  (declare (ignore depth))
  (format stream "#<DL ~A>" (dl->list dl)))

(defun dl->list (lst)
  (if (dl-p lst)
      (cons (dl-data lst) (dl->list (dl-next lst)))
      lst))

(defun dl-insert (x lst)
  (let ((elt (make-dl :data x :next lst)))
    (when (dl-p lst)
      (if (dl-prev lst)
          (setf (dl-next (dl-prev lst)) elt
                (dl-prev elt) (dl-prev lst)))
      (setf (dl-prev lst) elt))
    elt))

(defun dl-list (&rest args)
  (reduce #'dl-insert args
          :from-end t :initial-value nil))

(defun dl-remove (lst)
  (if (dl-prev lst)
      (setf (dl-next (dl-prev lst)) (dl-next lst)))
  (if (dl-next lst)
      (setf (dl-prev (dl-next lst)) (dl-prev lst)))
  (dl-next lst))
```

Рис. 12.10. Построение двусвязных списков.

Рис. 12.11. Двусвязный список.

Вызов `defstruct` также определяет функции, аналогичные `car`, `cdr` и `consp`: `dl-data`, `dl-next` и `dl-p`. Функция печати `dl->list` возвращает обычный список с теми же значениями, что и двусвязный.

Функция `dl-insert` похожа на `cons`. По крайней мере, она, как и `cons`, является основным конструктором. В отличие от `cons`, она изменяет двусвязный список, переданный вторым аргументом. В данной ситуации это совершенно нормально. Чтобы поместить новый объект в начало обычного списка, вам не требуется его изменять, однако, чтобы поместить объект в начало двусвязного списка, необходимо присвоить полю `prev` указатель вперед на новый объект.

Другими словами, несколько обычных списков могут иметь общий хвост. Для пары двусвязных списков это невозможно, так как хвост каждого из них имеет разные указатели на голову. Если бы функция `dl-insert` не была деструктивна, ей бы приходилось всегда копировать свой второй аргумент.

Другое интересное различие между одно- и двусвязными списками заключается в способе доступа к их элементам. Имея односвязный список, вы получаете доступ к элементам с его первого элемента. В двусвязном списке вы можете перемещаться от любого элемента в обе стороны, перейти к произвольному элементу вы можете из любого места. Используя это свойство, `dl-insert` умеет добавлять новый элемент не только в начало, но и в середину двусвязного списка.

Функция `dl-list` аналогична `list`. Она получает произвольное количество аргументов и возвращает состоящий из них `dl`:

```
> (dl-list 'a 'b 'c)
#<DL (A B C)>
```

В ней используется `reduce` с параметрами `:from-end t` и `:initial-element nil`, заменяя последовательность вызовов:

```
(dl-insert 'a (dl-insert 'b (dl-insert 'c nil)))
```

Заменив `#'dl-insert` на `#'cons` в определении `dl-list`, она будет вести себя аналогично спискам.

```
> (setf dl (dl-list 'a 'b))
#<DL (A B)>
> (setf dl (dl-insert 'c dl))
#<DL (C A B)>
> (dl-insert dl (dl-next dl))
#<DL (R A B)>
> dl
#<DL (C R A B)>
```

Наконец, для удаления элемента двусвязного списка определена `dl-remove`. Как и с `dl-insert`, ее имеет смысл сделать деструктивной.

12.7. Циклические списки

Да, существует возможность создания циклических списковЁ047in. Они бывают двух видов. Наиболее полезными являются те, которые имеют замкнутую структуру верхнего уровня. Такие списки называются *циклическими по хвосту* (*cdr-circular*), так как цикл замыкается на хвосте ячейки.

Чтобы создать такой список, содержащий один элемент, необходимо установить указатель `cdr` на самого себя:

```
> (setf x (list 'a))
(A)
> (progn (setf (cdr x) x) nil)
NIL
```

Теперь `x` – циклический список. Его структура изображена на рис. 12.12.

Рис. 12.12. Циклические списки.

При попытке напечатать такой список символ `a` будет выводиться до бесконечности. Этого можно избежать, установив значение `*print-circle*` в `t`:

```
> (setf *print-circle* t)
T
> x
#1=(A . #1#)
```

При необходимости вы можете использовать макросы чтения `#n=` и `#n#` для представления такой структуры.

Списки с циклическим хвостом могут быть полезны для представления, например, буферов или пулов. Следующая функция превратит произвольный нециклический непустой список в циклический:

```
(defun circular (lst)
  (setf (cdr (last lst)) lst))
```

Списки могут быть также *циклическими по голове* (*car-circular*). Такой список можно понимать как дерево, являющееся поддеревом самого себя. По аналогии. они содержат цикл, замкнутый на `car` ячейки. В приведенном ниже примере такого списка второй его элемент является им самим:

```
> (let ((y (list 'a)))
    (setf (car y) y)
    y)
#1=(#1#)
```

Полученный список изображен на рис. 12.12. Несмотря на цикличность, он по-прежнему является нормальным, в отличие от циклических по хвосту.

Список может быть циклическим по голове и хвосту одновременно. `Car` и `cdr` такой ячейки будут указывать на нее саму:

```
> (let ((c (cons 1 1)))
    (setf (car c) c
          (cdr c) c)
```

```
    c)
#1=(#1# . #1#)
```

Сложно представить, для чего могут использоваться подобные объекты. Из всей истории вам нужно запомнить одно: с помощью циклических списков можно избегать их непреднамеренного создания, так как большинство функций, работающих со списками, будут уходить в бесконечный цикл.

Циклическая структура свойственна не только спискам, но и, например, массивам:

```
> (setf *print-array* t)
T
> (let ((a (make-array 1)))
    (setf (aref a 0) a)
    a)
#1=#(#1#)
```

Любой объект, состоящий из элементов, может включать себя в качестве одного из них.

Разумеется, структуры, создаваемые `defstruct`, также могут быть циклическими. Например, структура `c`, представляющая элемент дерева, может иметь поле `parent`, содержащее другую структуру `p`, чье поле `child` ссылается обратно на `c`:

```
> (progn (defstruct elt
           (parent nil) (child nil))
         (let ((c (make-elt))
               (p (make-elt)))
           (setf (elt-parent c) p
                 (elt-child  p) c)
           c))
#1=#S(ELT PARENT #S(ELT PARENT NIL CHILD #1#) CHILD NIL)
```

Для печати подобных структур вам необходимо установить `*print-circle*` в `t` или избегать вывода подобных объектов.

12.8. Неизменные структуры

Неизменные объекты также являются частью кода, и наша задача не допускать их модификации. Цитируемый список является неизменной структурой, и необходимо проявлять аккуратность при работе с отдельными его ячейками. Например, имеется предикат для проверки на принадлежность к арифметическим операторам:

```
(defun arith-op (x)
  (member x '(+ - * /)))
```

В случае истинного значения он будет возвращать часть цитируемого списка. Изменяя возвращаемое значение:

```
> (nconc (arith-op '*) '(as it were))
(* / AS IT WERE)
```

мы изменяем и сам исходный список:

```
> (arith-op 'as)
(AS IT WERE)
```

Такое поведение не обязательно является ошибочным. Однако, если вам важна неизменность объектов при работе с деструктивными операторами, нужно помнить о подобных случаях.

Избежать возврата части неизменной структуры в случае `arith-op` можно несколькими способами. В общем случае, замена цитирования на явный вызов функции `list` приведет к созданию нового списка при каждом вызове:

```
(defun arith-op (x)
  (member x (list '+ '- '* '/)))
```

Однако, вызов `list` ударит по производительности, и вместо `member` лучше использовать `find`:

```
(defun arith-op (x)
  (find x '(+ - * /)))
```

Проблема, рассмотренная в этой главе, актуальна и для других типов данных: массивов, строк, структур, экземпляров и так далее. Не следует модифицировать что-либо, заданное в тексте дословно.

Даже если вы собираетесь написать самомодифицируемую программу, изменение структур-констант не является правильным выбором. Компилятор *может* связывать константы с кодом, а деструктивные операторы *могут* изменять свои аргументы, но это не гарантируется. Составлять самомодифицируемые программы вы можете, например, с помощью замыканий (см. раздел 6.5).

Итоги главы

1. Два списка могут разделять один хвост. Списки могут разделять структуры как деревья, без разделения структуры верхнего уровня. Разделения структур можно избежать с помощью копирования.
2. Разделяемость структуры не влияет на поведение функций, но о ней нельзя забывать, если вы собираетесь модифицировать списки. Изменение одного списка может привести к изменению другого, если они разделяют общую структуру.
3. Очереди могут быть представлены как `cons`-ячейки, `car` которых указывает на первую ячейку списка, а `cdr` – на последнюю.
4. По соображениям производительности деструктивным операторам позволено модифицировать свои аргументы.
5. В некоторых приложениях использование деструктивных операторов более естественно.
6. Списки могут быть циклическими по голове и хвосту. Лисп умеет работать с циклическими и разделяемыми структурами.
7. Не следует изменять константы, встречающиеся в тексте программы.

Упражнения

1. Нарисуйте три различных дерева, которые будут выводиться как `((A) (A) (A))`. Напишите выражения, производящие каждое из них.
2. Считая уже определенными `make-queue`, `enqueue` и `dequeue` (см. рис. 12.7), нарисуйте рамочное представление очереди после каждого следующего шага:
```
> (setf q (make-queue))
(NIL)
> (enqueue 'a q)
(A)
> (enqueu 'b q)
(A B)
> (dequeue q)
A
```
3. Определите функцию `copy-queue`, возвращающую копию очереди.
4. Определите функцию, помещающую заданный объект в начало очереди.
5. Определите функцию, (деструктивно) перемещающую первый найденный (`eql`) экземпляр заданного объекта в начало очереди.
6. Определите функцию, проверяющую наличие заданного объекта в списке. Список может быть циклическим по хвосту.
7. Определите функцию, проверяющую, является ли ее аргумент циклическим по хвосту.
8. Определите функцию, проверяющую, является ли ее аргумент циклическим по голове.

13. Скорость

На самом деле, Лисп сочетает в себе два языка: язык для быстрого написания программ и язык для написания быстрых программ. На первых этапах создания программы вы пользуетесь первым, а когда ее структура закристаллизуется, вы берете в руки второй.

Давать общие рекомендации по поводу оптимизации довольно сложно из-за внушительного разнообразия реализаций Common Lisp. Действие, ускоряющее выполнение кода на одной из них, может повлечь падение его производительности в другой. Чем мощнее язык, тем дальше он отстоит от реального железа, а чем дальше от железа находится реализации, тем большими являются различия между ними. Несомненно, ряд действий ускорит выполнение кода в любой реализации, но, все же, не стоит забывать о рекомендательном характере содержимого этой главы.

13.1. Правило бутылочного горла

Содержимое этой главы описывает общие идеи оптимизации и не зависит от реализации. Мы поговорим о том, как алгоритмы могут быть связаны с бутылками.

Для понимания сути оптимизации важно понимать, что в любой программе есть, как правило, несколько участков, выполнение которых занимает большую часть времени. По мнению Кнута, «подавляющая часть времени выполнения программы, не связанной с вводом/выводом, сосредоточена в примерно 3% исходного кода».°Ё101in Оптимизация таких участков принесет заметное ускорение, оптимизация остальных будет пустой тратой времени.

Из этого следует исключительная важность первого шага оптимизации любой программы – поиска подобных узких мест, бутылочных горлышек. Многие реализации Лиспа предоставляют собственные *профилировщики*, назначением которых является поиск узких мест и подсчет временных затрат на них. Профилировщик – ценный инструмент, совершенно необходимый для создания действительно эффективного кода. Если ваша реализация Лиспа предоставляет профилировщик, то наверняка предоставляет и документацию на него. Изучите ее. Если у вас нет профилировщика, то вам придется угадывать узкие места, но это далеко не очевидная задача.

Из всей истории с бутылочным горлышком вам надо понять одну важную вещь: усилия для оптимизации программы на ранних этапах ее написания окажутся потраченными впустую. Кнут предлагает еще более жесткую формулировку: «Преждевременная оптимизация является корнем всех зол (или, по крайней мере, большей части) в программировании».°Ё102in Пока программа не написана, сложно сказать, где возникнет узкое место. Кроме того, оптимизация затрудняет модификацию программы, и попытка оптимизировать программу в момент ее написания напоминает попытку писать картину быстросохнущими красками.

Программа получится хорошей с большей вероятностью, если на каждую подзадачу будет выделено оптимальное время. Одним из достоинств Лиспа является возможность писать код за разное время: быстро писать медленный код и писать быстрый код медленно. На первых этапах вы используете первый метод, и лишь затем переключаетесь на оптимизации. Эта модель соответствует правилу бутылочного горла. С помощью низкоуровневых средств, например, ассемблера, вы фактически оптимизируете каждую строку кода. Большая часть этих усилий идет впустую, так как бутылочное горло образуется из очень небольшого участка кода. Чем более высокоуровневым является язык, тем эффективнее разработка с его помощью.

Приступая к оптимизации, начинайте с самого верху. Для начала убедитесь, что используете наиболее оптимальный алгоритм и лишь потом переходите к низкоуровневым трюкам. Правильный выбор алгоритма играет существенно более важную роль, чем любые последующие оптимизации кода. К сожалению, правильный выбор алгоритма необходимо сделать как можно раньше, и вам придется балансировать между этим правилом и описанным в предыдущем абзаце.

13.2. Компиляция

Для управления процессом компиляции вам доступно пять параметров: `speed` (скорость производимого кода), `compilation-speed` (скорость самого процесса компиляции), `safety` (количество проверок на ошибки), `space` (размер памяти, занимаемой кодом) и `debug` (количество отладочной информации).

Интерактивность вместо интерпретации

Лисп – это интерактивный язык, но это не обязывает его быть интерпретируемым. Ранние версии Лиспа реализовывались как интерпретаторы, в результате чего, сложилось мнение, что все преимущества Лиспа вытекают из его интерпретируемости. Эта идея ошибочна: Common Lisp является интерпретируемым и компилируемым языком одновременно.

Некоторые реализации Common Lisp и вовсе не используют интерпретацию. В них все, что вводится в toplevel, компилируется перед вычислением. Таким образом, называть toplevel интерпретатором не только старомодно, но и, строго говоря, ошибочно.

Параметры компиляции не являются переменными в привычном понимании. Это своего рода веса, определяющие важность каждого параметра: от 0 (не важно) до 3 (очень важно). Представим, что узкое место находится во внутреннем цикле некоторой функции. Добавим соответствующую декларацию:

```
(defun bottleneck (...)
  (do (...)
      (...)
    (do (...)
        (...)
      (declare (optimize (speed 3) (safety 0)))
       ...)))
```

Как правило, подобные декларации внедряются лишь после завершения основной работы по созданию программы.

Чтобы запросить оптимизацию скорости всех последующих функций, вы можете сказать:

```
(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))
```

Это чересчур радикальный и часто ненужный шаг. Не забывайте про бутылочное горлышко.[1]

Важным классом оптимизаций, производимых компиляторами Лиспа, является оптимизация хвостовых вызовов. Задавая максимальный вес параметра `speed`, вы включаете эту опцию, если она поддерживается компилятором.

Вызов считается *хвостовым*, если после него не нужно ничего вычислять. Следующая функция возвращает длину списка:

```
(defun length/r (lst)
  (if (null lst)
      0
      (1+ (length/r (cdr lst)))))
```

Данный рекурсивный вызов не является хвостовым, так как его значение передается функции `1+`. Тем не менее, следующая функция обладает хвостовой рекурсией:

```
(defun length/tr (lst)
  (labels ((len (lst acc)
             (if (null lst)
               acc
               (len (cdr lst) (1+ acc)))))
    (len lst 0)))
```

[1] Старые реализации могут не предоставлять *declaim*. В таком случае используйте *proclaim* с цитируемым аргументом.

Вообще говоря, рекурсивной здесь является локальная функция `len`, и она обладает хвостовой рекурсией, так как сам рекурсивный вызов является последним действием в функции. Вместо того, чтобы вернуть значение вызова назад, оно передается вместе со следующим вызовом, и нам достаточно в конце просто вернуть значение параметра `acc`.

Хороший компилятор может преобразовывать хвостовой вызов в переход `goto`, и хвостовая рекурсия считается обычным циклом.°Ё103in В машинном коде, когда управление впервые передается в сегмент с инструкциями, представляющими `len`, на стек кладется информация, указывающая адрес возврата, чтобы выполнить оставшуюся часть операций. Однако, если таковых нет, достаточно просто выполнить возврат в точку вызова. После установки новых параметров мы можем прыгнуть назад к началу функции и действовать так, как будто это вторая точка вызова. Это означает, что можно обойтись без реального вызова функции.

Другой способ уйти от затрат на полноценный вызов функции – заставить компилятор встраивать функции. Это особенно ценно для небольших функций, затраты на выполнение которых сопоставимы с затратами на сам вызов. Например, следующая функция выясняет, является ли ее аргумент списком или одиночным элементом:

```
(declaim (inline single?))

(defun single? (lst)
  (and (consp lst) (null (cdr lst))))
```

Так как до ее определения была сделана глобальная `inline`-декларация, вызов `single?` не будет приводить к реальному вызовы функции. Если мы определим функцию, вызывающую ее:

```
(defun foo (x)
  (single? (car x)))
```

тогда при компиляции `foo` код `single?` будет встроен в код `foo` так, как если бы мы написали:

```
(defun foo (x)
  (let ((lst (bar x)))
    (and (consp lst) (null (cdr lst)))))
```

Существует два ограничения на встраиваемость функции. Не могут встраиваться рекурсивные функции. Если встраиваемость декларируется для уже определенной функции, то должны быть перекомпилированы все другие функции, вызывающие ее, иначе при их вызове оптимизации не произойдет.

В некоторых более ранних диалектах Лиспа аналогичную работу можно было проделать с помощью вызова макроса (см. раздел 10.2). В Common Lisp это не обязательно.

Различные компиляторы Лиспа различаются по возможностям оптимизации. Чтобы узнать, какая работа реально проделана компилятором, полезно изучить скомпилированный код, который можно получить с помощью `disassemble` дл отдельно выбранной функции. Даже если дизассемблерный листинг является для вас китайской грамотой, вы можете визуально оценить количество сделанных оптимизаций: скомпилируйте две версии, одна из которых содержит необходимые декларации, и просто оцените разницу. С помощью аналогичной методики можно выяснять, была ли применена inline-декларация. В любом случае, перед подобными экспериментами убедитесь, что установлены необходимые параметры компиляции для получения максимально быстрого кода°Ё104in

13.3. Декларации типов

Если Лисп – не первый язык, с которым вы сталкиваетесь, то вас может удивить, что мы добрались до сюда без единой декларации типов, того, что совершенно необходимо во многих других языках.

В большинстве языков для каждой переменной необходимо определить свой тип, и в дальнейшем переменная могут содержать лишь значения этого типа. Такие языки называют языками со *строгой типизацией*. Помимо лишней работы программисту,

такой подход ограничивает круг использования функций, делая невозможным применение их к данным разных типов, а также хранение в структурах данных разных типов.°Ё105in Преимуществом данного подхода является упрощение задачи компилятору: если он видит функцию, то знает заранее, какие действия он должен совершить.

В разделе 2.15 было упомянуто, что Common Lisp использует более гибкий подход к декларации типов.[1] Типы имеют значения, но не переменные. Последние могут содержать объекты любых типов.

Оставляя все как есть, мы вынуждены платить скоростью за гибкость. Раз функция может принимать аргументы разных типов, ей необходимо каждый раз затрачивать дополнительное время на выяснение того, с данными каких типов она вызывается.

Если от функции вам нужно, например, лишь сложение целых чисел, отказ от ее оптимизации приведет к низкой производительности. Подход к решению данной проблемы в Common Lisp таков: сообщите мне все, что вам известно. Если вам заранее известно, что вы складываете два числа типа `fixnum`, соответствующую функцию можно жестко связать с данным типом, как в Си.

Таким образом, само различие в подходе к оптимизации не приводит к разнице в плане скорости. Первый подход требует всех деклараций типов, второй – нет. В Common Lisp все декларации полностью опционны. Они способны ускорить работу программы, но (если программа корректна) неспособны изменить ее поведение. Для декларации типа глобальной переменной достаточно сказать:

```
(declaim (type fixnum *count*))
```

ANSI Common Lisp допускает декларации без использования слова `type`:

```
(declaim (fixnum *count*))
```

Локальные декларации выполняются с помощью `declare`, использование которого аналогично `declaim`. Декларации могут встречаться в любом месте кода там, где появляются новые переменные: в `defun`, `lambda`, `let`, `do` и так далее. Чтобы сообщить компилятору, что параметры функции принадлежат типу `fixnum`, нужно сказать:

```
(defun poly (a b x)
  (declare (fixnum a b x))
  (+ (* a (expt x 2)) (* b x)))
```

Имя переменной в декларации соответствует переменной, действительной в том же контексте, где встречается сама декларация.

Вы можете также задать тип любого выражения в коде с помощью `the`. Например, если нам известно, что значения `a`, `b` и `x` не только принадлежат типу `fixnum`, но и достаточно малы, чтобы промежуточные выражения также принадлежали типу `fixnum`, вы можете указать это явно:Ё060in

```
(defun poly (a b x)
  (declare (fixnum a b x))
  (the fixnum (+ (the fixnum (* a (the fixnum (expt x 2))))
                 (the fixnum (* b x)))))
```

Выглядит довольно неуклюже, так? К счастью, есть несколько причин, по которым вам не нужно шпиговать код словами `the`. Во-первых, это лучше поручить макросам.°Ё106in Во-вторых, многие реализации умеют самостоятельно выводить типы промежуточных объектов.

В Common Lisp существует невероятное многообразие типов, их набор практически не ограничен, ведь вы можете самостоятельно создавать собственные типы. Декларации же имеют значение лишь для некоторых из них. Когда же следует прибегать к декларациям типов? Есть два основных правила:

---

[1] В Лиспе используются два подхода к описанию типов: по месту хранения информации о типах и по месту ее применения. Декларативная типизация подразумевает связывание данных с конкретными типами, а *типизация времени выполнения* (*run-time typing*) подразумевает присвоение типов любым объектам в процессе выполнения программы. По сути, это одно и то же.

1. Имеет смысл декларировать типы в тех функциях, которые могут работать с аргументами разных типов. Если вам известно, что аргументы вызова + всегда будут `fixnum`, или первый аргумент `aref` всегда будет массивом одного типа, декларация будет полезной.
2. Обычно имеет смысл декларировать лишь те типы, которые находятся на дне иерархии типов: обращение с `fixnum` и `simple-array` будет полезным, а вот декларации `integer` и `sequence` не принесут ощутимого результата.

Декларации типов особенно важны при работе со сложными объектами, включая массивы, структуры и экземпляры. Такие декларации не только приводят к более быстрому коду, но и позволяют более эффективно организовать объекты в памяти.

Если о типе элементов массива неизвестно ничего, то он представляется в виде набора указателей. Однако, если тип известен, и все элементы принадлежат одному типу, скажем, `double-float`, тогда массив может быть представлен как набор чисел в формате `double-float`. Во-первых, такой массив буде более экономно использовать память. Во-вторых, отсутствие необходимости переходить по указателям приведет к более быстрому чтению и записи элементов.

Задать тип массива можно с помощью ключа `:element-type` в `make-array`. Такой массив называется *специализированным*. На рис. 13.1 показано, что будет происходить в большинстве реализаций при выполнении следующего кода:

Рис. 13.1. Результат задания типа элементов массива.

```
(setf x (vector 1.234d0 2.345d0 3.456d0)
      y (make-array 3 :element-type 'double-float)
      (aref y 0) 1.234d0
      (aref y 1) 2.345d0
      (aref y 2) 3.456d0)
```

Каждый прямоугольник соответствует машинному слову в памяти. Каждый массив на рис. 13.1 содержит заголовок неопределенной длины, за которым следует представление трех элементов. В массиве `x` каждый элемент – указатель. В нашем случае все три указателя одновременно ссылаются на элементы `double-float`, но могут ссылаться на произвольные объекты, однако, мы не сможем хранить их в виде вектора. В массиве `y` элементы действительно являются числами `double-float`. Второй вариант работает быстрее и занимает меньше места, но мы вынуждены платить за это ограничением на однородность массива.

Заметьте, что для доступа к `y` мы пользовались `aref`. специализированный вектор более не принадлежит типу `simple-vector`, и мы не можем ссылаться на его элементы с помощью `svref`.

Помимо специализации массива полезно включить соответствующую декларацию. Вы можете задать тип элементов и размерность следующим образом:

```
(declare (type (vector fixnum 20) v))
```

Приведенная декларация сообщает, что вектор `v` имеет размерность 20 и специализирован для целых чисел типа `fixnum`.

Наиболее общая декларация включает тип массива, тип элементов и список размерностей:

```
(declare (type (simple-array fixnum (4 4)) ar))
```

Массив `ar` теперь считается простым массивом `4x4`, специализированном для `fixnum`.

На рис. 13.2 показано, как создать массив `1000x1000` элементов `single-float` и как написать функцию, суммирующую все его элементы. Массивы располагаются в памяти в построчном порядке (row-major order). В таком же порядке рекомендуется и обращаться к его элементам (по возможности).

```
(setf a (make-array '(1000 1000)
                    :element-type 'single-float
                    :initial-element 1.0s0))

(defun sum-elts (a)
  (declare (type (simple-array single-float (1000 1000))
                 a))
    (let ((sum 0.0s0))
      (declare (type single-float sum))
      (dotimes (r 1000)
        (dotimes (c 1000)
          (incf sum (aref a r c))))
   sum))
```

Рис. 13.2. Суммирование по массиву.

Для сравнения производительности `sum-elts` с декларациями и без них мы воспользуемся макросом `time`, который помимо прочего выводит время выполнения его тела. В разных реализациях результаты его работы отличаются. Его применение имеет смысл только для скомпилированных функций. Скомпилировав `sum-elts` с параметрами, обеспечивающими максимально быстрый код, он отрабатывает за долю секунды:

```
> (time (sum-elts a))
User Run Time = 0.43 seconds
1000000.0
```

Теперь уберем все декларации и перекомпилируем `sum-elts`:

```
> (time (sun-elts a))
User Run Time = 5.17 seconds
1000000.0
```

Важность деклараций типов, особенно при работе с массивами и отдельными числами, сложно переоценить. Декларации типов обеспечили нам двенадцатикратный прирост производительности.

13.4. Обходимся без мусора

Лисп позволяет относиться беззаботно не только к типам объектов, но и к их расположению в памяти. На ранних стадиях создания программы думать о таких вещах противопоказано, так как они склонны ограничивать воображение. Когда программа становится зрелой, она может в меньшей мере полагаться на динамическое выделение памяти и поэтому стать быстрее.

В любом случае, выделение меньшего количества памяти вовсе не обязано приводить к временным издержкам. К сожалению, ряд реализаций имеют не лучшие сборщики мусора, и этот процесс может отнимать ощутимое время. До недавнего времени большинство реализаций имели не самые удачные сборщики мусора, оттуда и пришло мнение, что ради производительности программы должны избегать выделения памяти. Недавние исследования в области сборки мусора перевернули эту идею с ног на голову. Некоторые реализации имеют настолько изощренных сборщиков мусора, что выделить память под объект и выбросить его при ненадобности оказывается быстрее, чем использовать их повторно.

В этом разделе показаны основные методы экономии памяти. Скажется ли это на производительности, зависит от используемой реализации. Как всегда, лучший совет – попробуйте сами.

Существует множество методов избегания выделения памяти. Некоторые из них практически не изменяют вид программы. Например, одним из наиболее простых методов является использование деструктивных функций. В следующей таблице приводятся некоторые часто употребляемые функции и их деструктивные аналоги:Ё062in

Безопасные	Деструктивные
append	nconc
reverse	nreverse
remove	delete
remove-if	delete-if
remove-duplicates	delete-duplicates
subst	nsubst
subst-if	nsubst-if
union	nunion
intersection	nintersection
set-difference	nset-difference

Если вам известно, что модификация списка безопасна, используйте `delete` вместо `remove`, `nreverse` вместо `reverse`, и так далее.

Если вы желаете полностью исключить выделение, не следует забывать и о возможности создания объектов на лету. В действительности вам необходимо ограничить выделение пространства при их создании на лету, а также удаление их сборщиком. Общим решением является заблаговременное выделение блоков памяти и явная их переработка. *Заблаговременно* означает в момент компиляции, или некоторой процедуры инициализации. В таком случае скорость будет зависеть лишь от самого приложения.

Например, если обстоятельства позволяют нам ограничить размер стека, наш стек будет расти и урезаться вместе с заранее размещенным вектором, вместо того, чтобы стоиться из `cons`-ячеек. Common Lisp имеет встроенную поддержку использования векторов в качестве стека. Для этого есть необязательный параметр `fill-pointer` в `make-array`. Первый аргумент `make-array` задает количество памяти, выделяемой под вектор, а `fill-pointer`, если задан, определяет его фактическую исходную длину:

```
> (setf *print-array* t)
T
> (setf vec (make-array 10 :fill-pointer 2
                           :initial-element nil))
#(NIL NIL)
```

Функции будут считать его последовательностью двух элементов:

```
> (length vec)
2
```

Однако его размер может вырасти до 10. Для вектора, имеющего указатель заполнения, применимы функции `vector-push` и `vector-pop`, аналогичные `push` и `pop` для списков:

```
> (vector-push 'a vec)
2
> vec
#(NIL NIL A)
> (vector-pop vec)
A
> vec
#(NIL NIL)
```

Вызов `vector-push` увеличивает указатель заполнения и возвращает его старое значение. Мы можем помещать в такой вектор новые значения до тех пор, пока не подойдем к размеру вектора, заданному первым аргументом `make-array`. Если свободное место закончилось, `vector-push` вернет `nil`. В наш вектор мы можем поместить до 8 новых элементов.

Векторы с указателем заполнения имеют один недостаток – они более не принадлежат типу `simple-vector`, и вместо `svref` нам приходится использовать `aref`. Преимущества векторов с указателем заполнения сбалансированы их стоимостью.

Некоторые приложения могут производить очень длинные последовательности. В таком случае вам может помочь `map-into` вместо `map`. Вместо типа новой последовательности она получает первым аргументом саму последовательность, в которую будут записаны результаты. Для примера увеличим на единицу все элементы вектора:

```
(setf v (map-into #'1+ v))
```

На рис. 13.3 показан пример приложения, работающего с длинными векторами. Это программа, генерирующая несложный словарь рифм (точнее, словарь найденных рифм). Функция `read-words` читает слова из файла, содержащего по слову на строке,°Ё107in а `write-words` печатает их в обратном алфавитном порядке. Ее вывод может начинаться так:

```
(defconstant dict (make-array 25000 :fill-pointer 0))

(defun read-words (from)
  (setf (fill-pointer dict) 0)
  (with-open-file (in from :direction :input)
    (do ((w (read-line in nil :eof)
            (read-line in nil :eof)))
        ((eql w :eof))
      (vector-push w dict))))

(defun xform (fn seq) (map-into seq fn seq))

(defun write-words (to)
  (with-open-file (out to :direction :output
                          :if-exists :supersede)
    (map nil #'(lambda (x)
                 (fresh-line out)
                 (princ x out))
             (xform #'nreverse
                    (sort (xform #'nreverse dict)
                          #'string<)))))
```

Рис. 13.3. Генерация словаря рифм.

```
a amoeba alba samba marimba ...
```

и завершаться так:

```
... megahertz gigahertz jazz buzz fuzz
```

Используя преимущества векторов с указателем заполнения, мы сможем легко и эффективно расправиться с этой задачей.

В расчетных приложениях будьте аккуратны с типами `bignum`. Операции с `bignum` требуют выделения памяти и работают существенно медленнее. Даже если ваша программа в результате возвращает значения `bignum`, старайтесь организовывать промежуточные результаты с помощью `fixnum`.

Другой способ избежать выделения памяти – размещать объекты не в куче, а на стеке. Если вам известно, что данный объект будет использоваться временно, декларация *dynamic extent* потребует его размещения на стеке.

Декларируя `dynamic-extent` для переменной, вы утверждаете, что ее значение будет использоваться лишь с этой переменной до тех пор, пока она существует. Как может значение существовать дольше переменной? Вот пример:

```
(defun our-reverse (lst)
  (let ((rev nil))
    (dolist (x lst)
      (push x rev))
    rev))
```

Функция `our-reverse` использует для промежуточных вычислений переменную `rev`, по завершении возвращает ее значение. Сама переменная исчезает, а перевернутый список продолжает существовать, и никто не знает, какая его ожидает судьба.

Для контраста рассмотрим реализацию `adjoin`:

```
(defun our-adjoin (obj lst &rest args)
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

Из этого определения видно, что список `args` исчезает тогда же, когда перестают быть нужными его элементы. Значит, имеет смысл сделать декларацию `dynamic-extent`:

```
(defun our-adjoin (obj lst &rest args)
  (declare (dynamic-extent args))
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

Теперь компилятор может (но не обязан) размещать `args` на стеке, который будет упразднен по выходу из `our-adjoin`.

13.5. Пример: пулы

В приложениях, использующих структуры данных, вы можете избегать динамического выделения памяти, заранее размещая конкретное их количество в *пул* (*pool*). Когда вам необходима структура, вы забираете ее из пула, когда она перестает быть нужна, вы возвращаете ее назад.°Ё108in Проиллюстрируем эту идею на прототипе программы, контролирующей перемещение кораблей в порту, переписав этот прототип с использованием пула.

На рис. 13.4 показана первая версия. Переменная `*harbour*` содержит список кораблей, каждый из которых представлен в виде структуры `ship`. Функция `enter` вызывается при заходе корабля в порт; `find-ship` находит корабль с заданным именем (если он существует); `leave` вызывается, когда корабль покидает порт.

```
(defparameter *harbor* nil)

(defstruct ship
  name flag tons)

(defun enter (n f d)
  (push (make-ship :name n :flag f :tons d)
        *harbor*))

(defun find-ship (n)
  (find n *harbor* :key #'ship-name))

(defun leave (n)
  (setf *harbor*
        (delete (find-ship n) *harbor*)))
```

Рис. 13.4. Порт.

Отличный подход для первой версии программы. Однако такая реализация производит достаточно много мусора: новые ячейки производятся при создании новых структур и росте списка `*harbour*`.

От обоих источников мусора мы в состоянии избавиться. Например, так, как показано на рис. 13.5. Эта версия программы не производит никакого мусора.

```
(defconstant pool (make-array 1000 :fill-pointer t))

(dotimes (i 1000)
  (setf (aref pool i) (make-ship)))

(defconstant harbor (make-hash-table :size 1100
                                     :test #'eq))

(defun enter (n f d)
  (let ((s (if (plusp (length pool))
               (vector-pop pool)
               (make-ship))))
    (setf (ship-name s)        n
          (ship-flag s)        f
          (ship-tons s)        d
          (gethash n harbor) s)))

(defun find-ship (n) (gethash n harbor))

(defun leave (n)
  (let ((s (gethash n harbor)))
    (remhash n harbor)
    (vector-push s pool)))
```

Рис. 13.5. Порт, другая версия.

Строго говоря, выделение памяти все же происходит, но не в момент выполнения. Во второй версии `*harbour*` является хеш-таблицей, а не списком, и пространство под нее выделяется при компиляции. Сотня структур `ship` также создается при компиляции и сохраняется в векторе `pool`. (Если параметр `:fill-pointer` имеет значение `t`, указатель заполнения размещается в конце вектора.) Теперь при вызове `enter` нам нет необходимости создавать новую структуру, достаточно получить одну из уже существующих в пуле с помощью `make-ship`. Когда `leave` удаляет корабль из `harbour`, соответствующая структура не становится мусором, а возвращается на место в пул.

В данном примере мы собственноручно выполняли часть работы по управлению памятью. Это позволило нам избежать ее выделения в процессе работы, однако, создало дополнительные вычислительные затраты. Будет ли вторая версия программы работать быстрее первой, зависит от используемой реализации. Грубо говоря, имеет смысл использовать подобные трюки лишь в реализациях с примитивными сборщиками мусора, а также в приложениях реального времени, где неконтролируемый вызов сборщика мусора принесет проблемы.

13.6. Быстрые операторы

Ё003inВ начале главы было сказано, что Лисп – это по сути два разных языка. Если вы приглядитесь к дизайну Common Lisp, то убедитесь, что часть его операторов предназначена для ускорения выполнения, а другая часть – для удобства разработки.

Например, для доступа к элементу вектора существуют три оператора: `elt`, `aref`, `svref`. Такое разнообразие позволяет выжать из программы максимум производительности. На критических участках вы можете использовать `svref` вместо `elt` и `aref`, которые работают также и со списками и массивами.

Для списков использование специализированной функции `nth` эффективнее, чем `elt`. Лишь одна функция не имеет аналогов для разных типов. Почему в Common Lisp нет отдельной функции для списков? Потому что программа, выполняющая поиск длины списка, уже безнадежна в плане производительности. Здесь, как и во множестве других случаев, сам дизайн языка объясняет, что является эффективным, а что – нет.

Другая пара похожих функций – `eql` и `eq`. Первый предикат проверяет на идентичность, второй – на одинаковое размещение в памяти. Второй предикат обеспечивает большую эффективность. Применяйте его, когда известно заранее, что аргументы не являются числами или знаками. Два объекта равны с точки зрения `eq`,

если они имеют одинаковое размещение в памяти. Числа и знаки не обязаны располагаться в каком-либо определенном месте в памяти. (В некоторых реализациях `eq` все же применим к типам `fixnum`.) Для любых других аргументов `eq` будет работать аналогично `eql`.

Разумеется, быстрее всего выполняется сравнение `eq`, так как Лиспу достаточно сравнить лишь два указателя. Это значит, что хеш-таблицы с тестовой функцией `eq` (см. рис. 13.5) обеспечивают более быстрый доступ. В таких таблицах `gethash` сравнивает лишь указатели и даже не смотрит, на что они указывают. Помимо скорости доступа с хеш-таблицами связан еще один момент. Использование `eq`- и `eql`-таблиц приводит к издержкам, связанным с алгоритмами сборки мусора: после прохода GC хеши таких таблиц должны быть пересчитаны. Если это является проблемой, используйте `eql`-таблицы и числа типа `fixnum` в качестве ключей.

Вызов `reduce` может быть более эффективным, чем `apply`,[1] когда функция принимает параметр `rest`. Например, вместо выражения:

```
(apply #'+ '(1 2 3))
```

будет эффективнее использовать следующее выражение:

```
(reduce #'+ '(1 2 3))
```

Кроме того, необходимо помнить, что необязательные, ключевые и остаточные параметры имеют свою цену, так как их использование требует выполнения некоторой работы на этапе выполнения. Ключевые параметры являются наихудшими в этом отношении. Хорошие компиляторы умеют нивелировать эту разницу, но лишь для встроенных функций, а не ваших собственных. Поэтому вам следует избегать использования таких параметров на критических участках. Также разумно не передавать много аргументов параметрам `rest` там, где этого можно избежать.

Каждый компилятор имеет свой набор оптимизируемых условий. Например, некоторые могут оптимизировать вызов `case` с целочисленными ключами в небольшом диапазоне их значений. Как правило, узнать подробнее и специфичных оптимизациях в вашей реализации вы можете из документации к ней.

13.7. Двухстадийная разработка

Если в вашем приложении производительность имеет первостепенное значение, в качестве одного из решений попробуйте переписать критические участки на низкоуровневом языке типа Си или языка ассемблера. Такой подход применим не только к Лиспу, но и к любому высокоуровневому языку – критические участки пишутся на Си или ассемблере, а остальные пользуются преимуществами высокоуровневой разработки.

Стандарт Common Lisp не описывает механизм интеграции кода на других языках. Эта задача полностью ложится на плечи разработчиков конкретной реализации. Разумеется, реализации вовсе не обязаны поддерживать подобную возможность, но большинство из них предоставляют ее.

Вам может показаться напрасной тратой времени написание части кода на одном, а затем на другом языке. Тем не менее, практика показала, что это отличный способ написания приложений. На первой стадии вы разрабатываете функционал программы, на второй доводите критические участки до приемлемой производительности.

Если бы программирование было всего лишь механическим процессом, трансляцией спецификаций в код, было бы разумно выполнять всю работу за один шаг. Но настоящее программирование не имеет ничего общего с этой идеей. Независимо от проработанности спецификаций, исследовательский компонент в написании программ очень важен, и часто в намного большей степени, чем этого ожидает разработчик.

---

[1] Это утверждение автора является довольно сомнительным. По крайней мере, для ряда современных реализаций на приведенном примере наблюдается выраженный противоположный эффект. – *Прим. перев.*

Как бы ни хотелось рассматривать программирование как преобразование спецификаций в код, это порочная идея, хотя и распространенная. Исследовательская компонента важна уже потому, что любая спецификация по определению содержит недостатки.

В ряде случаев точность спецификации имеет первостепенное значение. Вытачивая деталь определенной формы из куска металла, важно получить именно то, что было задумано. Но это правило не распространяется на программы, так как и спецификации, и коды являются текстом. Это значит, что вы попросту *не можете* разработать абсолютно точную спецификацию, иначе это была бы уже готовая программа.°Ё109in

В приложениях, подразумевающих значительное количество исследования (объем которого, опять же, превзойдет ожидания) имеет смысл разграничить две эти стадии. Промежуточный результат после первой стадии не будет конечным. Например, при ваянии бронзовой скульптуры принято делать первый набросок из глины, который затем используется при отливании скульптуры из бронзы.°Ё110in В конечной скульптуре не остается глины, но эффект от ее использования все равно заметен. Теперь вообразите, насколько сложной была бы та же задача при наличии лишь слитка бронзы и зубила. Точно по такой же причине удобнее написать программу на Лиспе, а затем переписать на Си, чем писать ее на Си с самого начала.

Итоги главы

1. К оптимизации не следует приступать раньше времени. Следует уделять основное внимание узким местам, но начинать следует с выбора оптимального алгоритма.
2. Доступно пять параметров компиляции. Они могут устанавливаться как глобальными, так и локальными декларациями.
3. Хорошие компиляторы могут оптимизировать хвостовую рекурсию, превращая ее в циклы. Встраивание кода функций позволяет избежать их вызова.
4. Декларации типов не обязательны, но могут сделать программу более производительной. Особенно важны декларации в расчетном коде и операциях с массивами.
5. Выделение меньшего количества памяти позволяет ускорить программу, особенно на реализациях с примитивным сборщиком мусора. Используйте деструктивные функции, предварительное выделение памяти и размещение объектов на стеке.
6. В ряде ситуаций полезно не создавать объекты, а забирать их из предварительно созданного пула.
7. Некоторые части Лиспа предназначены для работы над производительностью, некоторые – для облегчения процесса разработки.
8. Процесс программирования обязательно включает исследовательский момент, который следует отделять от оптимизации, иногда даже с использованием двух разных языков.

Упражнения

1. Проверьте, понимает ли (то есть, выполняет ли соответствующие оптимизации) ваш компилятор `inline`-декларации.
2. Перепишите следующую функцию с использованием хвостовой рекурсии. Скомпилируйте и сравните их производительность.

```
(defun foo (x)
  (if (zerop x)
      0
      (+ 1 (foo (1- x)))))
```

   Подсказка: вам потребуется еще один параметр.

3. Добавьте декларации в следующие программы. Насколько ускорится их выполнение?

   (a) Арифметика дат в разделе 5.7.

   (b) Трассировщик лучей в разделе 9.8.

4. Перепишите код поиска в ширину в разделе 3.15 так, чтобы он выделял как можно меньше памяти.

5. Измените код двоичных деревьев поиска в разделе 4.7 так, чтобы он использовал пулы.

14. Более сложные вопросы

Эта глава необязательна. В ней описан ряд эзотерических особенностей языка. Common Lisp похож на айсберг: огромная часть его возможностей не видна для большинства пользователей. Быть может, вам никогда не придется определять пакеты или макросы чтения самостоятельно, но знание подобных моментов может сильно облегчить жизнь.

14.1. Спецификаторы типов

Типы не являются объектами. Так, нет объекта, соответствующего типу `integer`. То, что мы получаем при вызове `type-of` и передаем аргументом `typep`, является не типом, а спецификатором типа.

Спецификатор типа – это его имя. Спецификаторами простейших типов являются символы типа `integer`. Они формируют иерархию типов во главе с `t`, типу `t` принадлежат все объекты. Иерархия типов – не дерево, и вы можете добраться снизу доверху несколькими путями.

Типы соответствуют множествам объектов. Это означает, что, как и множеств объектов, количество типов может быть бесконечным. Некоторые типы обозначаются атомарно: `integer` соответствует множеству целых чисел. Мы можем определять типы и для более сложных множеств объектов.

Например, пусть `a` и `b` – спецификаторы некоторых типов, тогда `(or a b)` обозначает объединение множеств объектов, соответствующих типам `a` и `b`. Объект типа `(or a b)` принадлежит как типу `a`, так и типу `b`.

Если бы `circular?` была функцией, истинной для циклических по хвосту списков, то набор нормальных списков можно было охарактеризовать следующим спецификатором:[1]

```
(or vector (and list (not (satisfies circular?))))
```

Некоторые атомарные спецификаторы типов могут встречаться и в составных типах. Например, следующий спецификатор соответствует набору целых чисел от `1` до `100` включительно:

```
(integer 1 100)
```

Такие типы называют *конечными*.

Вы составном спецификаторе типа некоторые поля могут оставаться неопределенными. Такое поле помечается знаком `*`. Так,

```
(simple-array fixnum (* *))
```

описывает набор двухмерных простых массивов, специализированных для `fixnum`, а

```
(simple-array fixnum *)
```

описывает набор (надтип предыдущего) простых массивов, специализированных для `fixnum`. Завершающие звездочки могут быть опущены, и последний спецификатор может быть записан так:

```
(simple-array fixnum)
```

Если составной тип не содержит аргументов, то он эквивалентен атомарному. Так, тип `simple-array` соответствует всем простым массивам.

Чтобы не повторять постоянно громоздкие спецификаторы составных типов, пользуйтесь `deftype` для определения аббревиатур типов. Его использование

[1] Несмотря на то, что этот факт не упоминается в стандарте, вы можете использовать в спецификаторах типов выражения `and` и `or` с любым количеством аргументов, подобно макросам `and` и `or`.

напоминает `defmacro`, но он раскрывается не в выражение, а в спецификатор типа. С помощью определения:

```
(deftype proseq ()
  '(or vector (and list (not (satisfies circular?)))))
```

мы создаем новый атомарный спецификатор `proseq`:

```
> (typep #(1 2) 'proseq)
T
```

Вы можете определить составной спецификатор. Его аргументы не вычисляются, а рассматриваются как формы, как и в `defmacro`. Так,

```
(deftype multiple-of (n)
  `(and integer (satisfies (lambda (x)
                             (zerop (mod x ,n))))))
```

определяет `(multiple-of n)` как спецификатор для всех множителей `n`:

```
> (typep 12 '(multiple-of 4))
T
```

Спецификаторы типов интерпретируются, и это сказывается на производительности. Поэтому в данном случае лучше обойтись аналогичной функцией.

14.2. Бинарные потоки

В главе 7 помимо обсуждения символьных потоков упоминались также и бинарные. Бинарный поток – это источник и/или получатель не знаков, но *целых чисел*. Для создания бинарного потока необходимо задать необходимой подтип `integer`, чаще это `unsigned-byte`, как значение параметра `element-type` при открытии потока.

Для работы с бинарными потоками имеются лишь две функции: `read-byte` и `write-byte`. Так, для копирования содержимого файла вам потребуется написать нечто подобное:

```
(defun copy-file (from to)
  (with-open-file (in from :direction :input
                           :element-type 'unsigned-byte)
    (with-open-file (out to :direction :output
                            :element-type 'unsigned-byte)
      (do ((i (read-byte in nil -1)
              (read-byte in nil -1)))
          ((minusp i))
        (declare (fixnum i))
        (write-byte i out)))))
```

Определяя тип `unsigned-byte` в `:element-type`, вы сообщаете операционной системе размер единицы информации. Если вы желаете работать, например, с 7-битными числами, вы можете использовать:

```
(unsigned-byte 7)
```

14.3. Макросы чтения

В разделе 7.5 была представлена концепция макросимволов как знаков, имеющих особое значение для считывателя `read`. С каждым подобным символом связана функция, вызываемая всегда, когда `read` встретит его. Вы можете изменять определения функций, связанных с уже существующими макросимволами, а также определять собственные макросы чтения.

Функция `set-macro-character` предоставляет один из способов. Она принимает литеру и функцию, вызываемую `read` при нахождении заданной литеры.

Одним из старейших макросов чтения в Лиспе является `'`, `quote`. Его возможное определение выглядит следующим образом:

```
(set-macro-character #\'
  #'(lambda (stream char)
      (list (quote quote) (read stream t nil t))))
```

Когда `read` встречает `'`, она вызывает данную функцию для используемого в тот момент потока и знака. (В данном случае второй аргумент, знак, всегда игнорируется, так как это всегда кавычка.) Так, при чтении `'a` будет вызвано `(quote a)`.

Теперь вы видите назначение последнего аргумента `read`. Он сообщает, встречается ли вызов `read` внутри вызова `read`. Этот аргумент будет одинаковым практически для всех макросов чтения. Второй аргумент, `t`, сообщает, что при достижении конца файла `read` будет сигнализировать об ошибке. Третий аргумент задает значение, возвращаемое вместо данной ошибки в случае истинного второго аргумента, следовательно, в нашем примере оно может быть любое. Четвертый аргумент Ё027in, как было сказано выше, сообщает, что вызов `read` является рекурсивным.

Вы можете определять (через `make-dispatch-macro-character`) свои собственные управляющие макросимволы, но раз `#` уже определен, то вы можете пользоваться им. Шесть комбинаций, начинающихся с `#` уже зарезервированы для вашего использования: `#!`, `#?`, `#[`, `#]`, `#{` и `#}`.

Новая комбинация с управляющим макросимволом создается при вызове `set-dispatch-macro-character` по аналогии с `set-macro-character` (но принимает два символа). Следующий код создает макрос чтения, возвращающий список целых чисел:

```
(setf-dispatch-macro-character #\# #\?
  #'(lambda (stream char1 char2)
      (list 'quote
            (let ((lst nil))
              (dotimes (i (+ (read stream t nil t) 1))
                (push i lst))
              (nreverse lst)))))
```

Теперь `#?n` будет прочитан как список всех целых чисел от `0` до `n`:

```
> #?7
(0 1 2 3 4 5 6 7)
```

Не считая таких простых случаев, часто определяются макросимволы – ограничители списков. `#{` является еще одной комбинацией, зарезервированной для пользователя. Вот пример более хитрых скобок:

```
(set-macro-character #\} (get-macro-character #\)))

(set-dispatch-macro-character #\# #\{
  #'(lambda (stream char1 char2)
      (let ((accum nil)
            (pair (read-delimited-list #\} stream t)))
        (do ((i (car pair) (+ i 1)))
            ((> i (cadr pair))
             (list 'quote (nreverse accum)))
          (push i accum)))))
```

Только что мы определили поведение при встрече выражений вида `#{x y}`, которые преобразуются в список целых чисел от `x` до `y` включительно:

```
> #{2 7}
(2 3 4 5 6 7)
```

Функция `read-delimited-list` предоставляется специально для макросов чтения, подобных этому. Чтобы знак `}` был расценен как разграничитель, необходимо предварительно сообщить об этом с помощью `set-macro-character`.

Чтобы иметь возможность пользоваться макросом чтения в файле, в котором определен сам макрос, его определение должно быть завернуто в выражение `eval-then`, чтобы убедиться, что оно вычисляется в момент компиляции. В

противном случае определение будет скомпилировано, но не вычислено до тех пор, пока скомпилированный файл не будет загружен.

14.4. Пакеты

Пакет – это лисповый объект, отображающий имена в символы. Текущий пакет всегда хранится в глобальной переменной `*package*`. При запуске Common Lisp стартовым пакетом является `common-lisp-user`, известным также как пользовательский пакет. Функция `package-name` возвращает имя текущего пакета, а `find-package` возвращает пакет с заданным именем:

```
> (package-name *package*)
"COMMON-LISP-USER"
> (find-package "COMMON-LISP-USER")
#<Package "COMMON-LISP-USER" 4CD15E>
```

Обычно символ интернируется в пакет, являющийся на тот момент текущим. Функция `symbol-package` принимает символ и возвращает пакет, в который он был интернирован:

```
> (symbol-package 'sym)
#<Package "COMMON-LISP-USER" 4CD15E>
```

Забавно, что, раз это выражение считывается перед выполнением, и при его считывании сам символ `sym` интернируется, такое выражение будет всегда возвращать текущий пакет. Для последующего применения присвоим `sym` некоторые значение:

```
> (setf sym 99)
99
```

Теперь создадим пакет и переключимся в него:

```
> (setf *package* (make-package 'mine
                                :use '(common-lisp)))
#<Package "MINE" 63390E>
```

А вот теперь мы услышим зловещий звук, ведь мы теперь в ином мире, где `sym` не имеет значения:

```
MINE> sym
Error: SYM has no value.
```

Что случилось? Чуть раньше мы дали `sym` значение `99`, но сделали это в другом пакете, а значит, для другого символа, нежели `sym` в пакете `mine`.[1] Чтобы сослаться на исходный `sym` из другого пакета, необходимо предварить его именем его родного пакета и парой двоеточий:

```
MINE> common-lisp-user::sym
99
```

Итак, несколько символов с одинаковыми именами могут сосуществовать вместе. Один символ находится в пакете `common-lisp-user`, другой – в `mine`, и это разные символы. Помещая свою программу в отдельный пакет, вы можете не переживать по поводу используемых в ней имен. Даже если символы в разных пакетах имеют одинаковые имена, они не перестают быть разными символами.

Пакеты предоставляют возможность сокрытия информации. Программы должны ссылаться на функции и переменные по их именам. Если вы не делаете какое-то имя было доступным вне своего пакета, то функции из других пакетов не смогут на их ссылаться.

Использование пары двоеточий для обозначения символа не из текущего пакета обычно является дурным тоном. Фактически, вы нарушаете установленные ограничения, предоставляемые пакетами. Если символ не доступен из какого-то пакета естественным образом, вероятно, автор сделал так по какой-то причине.

---

[1] Некоторые реализации Common Lisp печатают имя пакета перед приглашением toplevel, когда вы находитесь не в пользовательском пакете.

Обычно следует ссылаться лишь на *экспортируемые* символы. Если мы вернемся назад в пользовательский пакет (`in-package` изменяет значение `*package*`) и экспортируем символ из него:

```
MINE> (in-package common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
T
> (setf bar 5)
5
```

то мы сделаем его видимым и в других пакетах. Теперь, вернувшись в пакет `mine`, мы сможем ссылаться на `bar` через одинарное двоеточие:

```
> (in-package mine)
#<Package "MINE" 63390E>
MINE> common-lisp-user:bar
5
```

Кроме того, символ `bar` может быть *импортирован* в текущий пакет, тогда он будет разделен между двумя пакетами:

```
MINE> (import 'common-lisp-user:bar)
T
MINE> bar
5
```

На импортированный символ можно ссылаться без какого-либо префикса. Теперь этот символ принадлежит и пакету `mine`, в котором более не может быть еще одного символа `mine:bar`.

А что произойдет, если такой символ уже есть? В таком случае при импортировании возникнет ошибка. Такую же ошибку мы увидим, попытавшись импортировать `sym`:

```
MINE> (import 'common-lisp-user::sym)
Error: SYM is already present in MINE.
```

Ранее мы предпринимали неудачную попытку получить значение `sym`, но он не имел какого-либо значения, и наша попытка привела к возникновению ошибки. Тем не менее, вследствие этого символ был интернирован в пакет `mine`, и теперь такой символ уже существует.

Другой способ получить доступ к символам другого пакета – *использовать* (*use*) его:

```
MINE> (use-package 'common-lisp-user)
T
```

Теперь *все* символы, экспортируемые пользовательским пакетом, принадлежат `mine`. (Если `sym` принадлежит также используемому пакету, мы получим ту же ошибку.)

Пакет, содержащий имена всех встроенных операторов и функций, имеет имя `common-lisp`. Так как мы передали это имя ключу `:use` в определении пакета `mine`, в нем будут видимы все имена Common Lisp:

```
MINE> #'cons
#<Compiled-Function CONS 462A3E>
```

Как и компиляция, операции с пакетами редко выполняются непосредственно в toplevel. Гораздо чаще они встречаются в файлах с исходными кодами. Обычно обходятся одним выражением (`defpackage` или `in-package`) в самом начале файла (см. стр. Ё030out).

Такая модульность может показаться насколько странной. Модули содержат не объекты, а имена. Каждый пакет, использующий `common-lisp`, содержит символ `cons`, так как этот символ содержится в используемом пакете. Если некоторые аспекты использования пакетов сбивают вас с толку, основная причина тому – они основываются не на объектах, а на именах.°Ё111in

14.5. Loop

Макрос `loop` первоначально был разработан в помощь начинающим Лисп-программистам для написания итеративного кода, позволяя использовать для этого выражения, напоминающие обычные английские фразы, которые затем транслируются в Лисп-код. К несчастью, `loop` оказался похож на английский язык в большей степени, чем предполагали его создатели: в несложных случаях вы можете использовать его, совершенно не задумываясь о том, как это работает, но понять общий механизм его работы практически невозможно.

Если вы желаете ознакомиться с `loop` за один день, то для вас есть две новости: хорошая и плохая. Хорошая заключается в том, что вы не одиноки в своем желании, и мало кто в действительности понимает, как он работает. Плохая же состоит в том, что у вас это не получится хотя бы потому, что стандарт ANSI не предоставляет формального описания его поведения.

Единственным источником информации об устройстве `loop` является используемая вами реализация, а единственным способом изучить его являются примеры. В описывающей `loop` главе стандарта ANSI имеется большой набор примеров, из которых вы можете познакомиться с основными концепциями.

Первой особенностью макроса `loop` является его *синтаксис*. Его тело состоит не из подвыражений, а из предложений, которые не разделяются скобками. Каждое предложение имеет свой синтаксис. В целом, `loop`-выражения напоминают Алгол-подобные языки, хотя имеют и ряд важных отличий. В частности, порядок происхождения событий полностью определяется порядком расположения его предложений.

Вычисление `loop`-выражения осуществляется в три стадии, причем каждое предложение может давать свой вклад более чем в одну фазу:

1. Пролог: выполняется один раз за весь вызов `loop`; устанавливает исходные значения переменных.
2. Тело: вычисляется на каждой итерации; начинается с проверок на завершение, за которым следуют вычисляемые выражения; по завершении значения переменных обновляются.
3. Эпилог: вычисляется по завершении всех итераций; определяет возвращаемое значение.

Рассмотрим некоторые примеры `loop`-выражений и прикинем, к каким стадиям относятся их части:

```
> (loop for x from 0 to 9
        do (princ x))
0123456789
NIL
```

Данное выражение печатает целые числа от `0` до `9` и возвращает `nil`. Первое предложение:

```
for x from 0 to 9
```

дает вклад в две первые стадии, устанавливая значения переменной (`x` = `0`), а также определяющее проверки на завершение (пока `x` не будет равен `9`) и правила обновления переменных. Второе предложение:

```
do (princ x)
```

определяет вычисляемое тело.

В более общем виде предложение с `for` определяет исходное состояние и условия завершения. Завершение может управляться чем-нибудь типа `while` или `until`:

```
> (loop for x = 8 then (/ x 2)
        until (< x 1)
        do (princ x))
8421
NIL
```

Вы можете использовать составные конструкции с `for`, содержащие несколько переменных и обновляющие их параллельно:

```
> (loop for x from 1 to 4
        and y from 1 to 4
        do (princ (list x y)))
(1 1) (2 2) (3 3) (4 4)
NIL
```

Наличие нескольких `for` приведет к последовательному обновлению переменных.

Другой задачей, для которой обычно используют итерацию, является накопление каких-либо значений. Например:

```
> (loop for x in '(1 2 3 4)
        collect (1+ x))
(2 3 4 5)
```

При использовании `in` вместо `from` переменная по очереди принимает значения каждого из элементов списка.

В приведенном примере `collect` оказывает влияние на все три стадии. В первой создается безымянный аккумулятор, имеющий значение `nil`. Вторая стадия получает код `(1+ x)`, значение которого присоединяется к аккумулятору. В итоге возвращается значение безымянного аккумулятора.

Это наш первый пример, возвращающий какое-то конкретное значение. Существуют возможность явного задания возвращаемого значения, но в его отсутствие этим занимается `collect`. Фактически, эту же работу мы могли бы выполнить через `mapcar`.

Наиболее частым использованием `loop` является сбор результатов применения некоторой функции к заданному диапазону аргументов:

```
> (loop for x from 1 to 5
        collect (random 10))
(3 8 6 5 0)
```

В данном примере мы получили список из пяти случайных чисел. Но для этой же цели мы уже определяли функцию `map-int` (см. стр. Ё040out). Тогда зачем же мы это делали, если у нас есть `loop`? Разумно задать и еще один вопрос. Зачем нам нужен `loop`, когда у нас есть `map-int`?°Ё112in

С помощью `collect` можно собирать значения в именованный список. Следующая функция принимает список чисел и возвращает списки его четных и нечетных элементов:

```
(defun even/odd (ns)
  (loop for n in ns
        if (evenp n)
           collect n into evens
           else collect n into odds
        finally (return (values evens odds))))
```

Предложение `finally` задает последнюю стадию преобразования `loop`-выражения. В данном случае оно определяет возвращаемое выражение.

Предложение `sum` похоже на `collect`, но накапливает значения в виде числа, а не списка. Получить сумму всех чисел от `1` до `n` мы можем следующим образом:

```
(defun sum (n)
  (loop for x from 1 to n
        sum x))
```

Более детальное рассмотрение `loop` содержится в приложении D начиная со стр. Ё058out. Приведем лишь несколько примеров. Рис. 14.1 содержит две итеративные функции из предыдущих глав, а рис. 14.2 показывает аналогичную их реализацию с помощью `loop`.

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcall fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setf wins obj
                    max  score))))
        (values wins max))))
(defun num-year (n)
  (if (< n 0)
      (do* ((y (- yzero 1) (- y 1))
            (d (- (year-days y)) (- d (year-days y))))
           ((<= d n) (values y (- n d))))
      (do* ((y yzero (+ y 1))
            (prev 0 d)
            (d (year-days y) (+ d (year-days y))))
           ((> d n) (values y (- n prev))))))
```

Рис. 14.1. Итерация без loop.

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (loop with wins = (car lst)
         with max = (funcall fn wins)
         for obj in (cdr lst)
         for score = (funcall fn obj)
         when (> score max)
           do (setf wins obj
                    max score)
         finally (return (values wins max)))))
(defun num-year (n)
  (if (< n 0)
      (loop for y downfrom (- yzero 1)
         until (<= d n)
         sum (- (year-days y)) into d
         finally (return (values (+ y 1) (- n d))))
      (loop with prev = 0
         for y from yzero
         until (> d n)
         do (setf prev d)
         sum (year-days y) into d
         finally (return (values (- y 1) (- n prev))))))
```

Рис. 14.2. Итерация с помощью loop.

Одно `loop`-выражение может иметь доступ к переменным в другом. Например, в определении `even/odd` предложение `finally` ссылается на переменные, установленные в двух предыдущих предложениях. Отношения между переменными являются одним из самых больших затруднений при использовании `loop`. Сравните два выражения:

```
(loop for y = 0 then z
        for x from 1 to 5
        sum 1 into z
        finally (return (values y z)))

(loop for x from 1 to 5
        for y = 0 then z
        sum 1 into z
        finally (return (values y z)))
```

Они содержат всего четыре предложения и потому кажутся довольно простыми. Действительно ли они возвращают одинаковые значения? Какие значения они вернут? Попытки найти ответ в стандарте окажутся тщетны. Каждое предложение внутри `loop` само по себе незатейливо, но *сочетаться* они могут довольно причудливым образом.

По этой причине необдуманное использование `loop` не рекомендуется. Тем не менее, на несложных примерах типа рис. 14.2 его использование может упрощать код.

14.6. Исключения

В Common Lisp *исключениями* (*conditions*) принято называть ошибки и другие подобные ситуации, которые могут возникать в процессе выполнения. Когда сигнализируется какое-либо условие, вызывается соответствующий обработчик. По умолчанию в большинстве случаев вы попадаете в отладчик. Но Common Lisp предоставляет большие возможности по обработке исключений. Вы можете переопределять уже существующие обработчики и даже определять свои собственные.

Мало кому приходится работать с исключениями непосредственно. Они огорожены несколькими уровнями абстракции, но для их понимания все же полезно иметь представление о нижележащем механизме.

Common Lisp предоставляет несколько операторов, сигнализирующих об ошибках. Основным является `error`. При вызове вы должны сообщить ему те же аргументы, что и `format`:

```
> (error "Your report uses ~A as a verb." 'status)
Error: Your report uses STATUS as verb.
       Options: :abort, :backtrace
>>
```

Если такое исключение не обрабатывается, дальнейшее вычисление будет прервано.

Более абстрактными операторами являются `ecase`, `check-type` и `assert`. Первый напоминает `case`, но сигнализирует об ошибке, когда не найдено совпадение ни с одним ключом:

```
> (ecase 1 (2 3) (4 5))
Error: No applecable clause.
       Options: :abort, :backtrace
>>
```

Обычное `case`-выражение вернет `nil`, но без надобности замалчивать отсутствие совпадений считается плохим стилем. Используйте `ecase`, если не предоставляете умолчальное условие через `otherwize`.

Макрос `check-type` принимает адрес, имя типа и необязательную строку. Он сигнализирует *корректируемую ошибку*, если значение по данному адресу не соответствует заданному типу. Обработчик корректируемой ошибки предложит один из вариантов ее исправления:

```
> (let ((x '(a b c)))
    (check-type (car x) integer "an integer")
    x)
Error: The value of (CAR X), A, should be an integer.
       Options: :abort, :backtrace, :continue
>> :continue
New value of (CAR X)? 99
(99 B C)
>
```

Только что мы привели пример коррекции ошибки, исправив значение `(car x)`, после чего выполнение продолжилось с исправленным значением.

Этот макрос был определен с помощью более общего `assert`, которые принимает тестовое выражение и список одного или более адресов, за которыми следуют те же аргументы, которые вы бы передали в `error`:

```
> (let ((sandwitch '(ham on rye)))
    (assert (eql (car sandwich) 'chicken)
             ((car sandwitch))
             "I wanted a ~A sandwich." 'chicken)
    sandwitch)
Error: I wanted a CHICHEN sandwitch.
       Options: :abort, :backtrace, :continue
>> :continue
New value of (CAR SANDWICH)? 'chicken
(CHICKEN ON RYE)
>
```

Кроме того, имеется возможность создавать собственные обработчики, но ей пользуются нечасто. Обычно обходятся уже имеющимися средствами, например, `ignore-errors`. Этот макрос ведет себя аналогично `progn`, если ни один из его аргументов не приводит к ошибке. Если же всплывает ошибка, то работа не прерывается, а выражение `ignore-errors` немедленно прекращается с возвратом двух значений: `nil` и исключения, ставшего источником ошибки.

Например, если вы даете пользователю возможность вводить собственные выражения, то вам понадобится защита от некорректно сформированных выражений:

```
(defun user-input (prompt)
  (format t prompt)
  (let ((str (read-line)))
    (or (ignore-errors (read-from-string str))
        nil)))
```

Функция вернет `nil`, если считанное выражение содержит синтаксическую ошибку:

```
> (user-input "Please type an expression> ")
Please type an expression> #%@#+!!
NIL
```

15. Пример: умозаключения

В следующих трех главах будут описаны три самостоятельные программы. Они призваны показать, как должны выглядеть полноценные программы на Лиспе. Кроме того, они решают задачи, для которых Лисп подходит особенно хорошо.

В этой главе мы рассмотрим программу, совершающую логические умозаключения, основанные на наборе правил «если-то». Это классический пример, не только часто встречающийся в учебниках, но и отражающий суть Лиспа как языка «символьных вычислений». Многие программы, написанные на заре Лиспа, имеют похожую изюминку.

15.1. Цель

В данной программе мы собираемся представлять информацию знакомым нам способом: списком из предиката и его аргументов. Чтобы представить факт отцовства Дональда по отношению к Ненси, мы скажем:

```
(parent donald nancy)
```

Из подобных фактов складываются правила, которым они подчиняются. Мы будем представлять их так:

```
(<- заголовок тело)
```

где заголовок соответствует условию, а тело – следствию. Переменные внутри них мы будем представлять символами со знаком вопроса. Так, следующее правило:

```
(<- (child ?x ?y) (parent ?y ?x))
```

сообщает, что если `y` является родителем `x`, то `x` является ребенком `y`, или, более точно, мы можем доказать любой факт вида `(child x y)` через доказательство `(parent y x)`.

Выражение `body` может иметь более сложный вид: включать логические операторы `and`, `or`, `not`. Так, правило, согласно которому, если `x` является мужчиной и родителем `y`, то `x` – отец `y`, выглядит следующим образом:

```
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
```

Одни правила могут опираться на другие. Например, правило, определяющее, является ли `x` дочерью `y`, опирается на уже определенное правило родительства:

```
(<- (daughter ?x ?y) (and (child ?x ?y) (female ?x)))
```

При доказательстве выражений может использоваться любое количество правил, необходимых для достижения твердой почвы фактов. Этот процесс иногда называют *обратным логическим выводом* (*backward chaining*). Он заключается в попытке последовательного подгона гипотезы под имеющиеся факты. В результате его применения получается цепочка (скорее, даже дерево) правил, которые могут привести нас от уже известных фактов к доказательству нашей гипотезы.°Ё113in

15.2. Соответствие

Ё002inЧтобы наша программа умела выполнять обратный логический вывод, необходимо научить ее выполнять сопоставление с образцом (pattern-matching): два списка, которые могут содержать переменные, сверяются друг с другом, при этом переменные получают такие значения, чтобы списки стали одинаковыми. Например, если `?x` и `?y` переменные, то два списка:

```
(p ?x ?y c ?x)
(p  a  b c  a)
```

совпадают, если `?x=a` и `?y=b`, а списки:

```
(p ?x b ?y a)
(p ?y b  c a)
```

совпадают при `?x=?y=c`.

На рис. 15.1 показана функция `match`, сопоставляющая два дерева. Если они могут совпадать, то она возвращает ассоциативный список, показывающий, как именно они совпадают:

```
(defun match (x y &optional binds)
  (cond
    ((eql x y) (values binds t))
    ((assoc x binds) (match (binding x binds) y binds))
    ((assoc y binds) (match x (binding y binds) binds))
    ((var? x) (values (cons (cons x y) binds) t))
    ((var? y) (values (cons (cons y x) binds) t))
    (t
     (when (and (consp x) (consp y))
       (multiple-value-bind (b2 yes)
           (match (car x) (car y) binds)
         (and yes (match (cdr x) (cdr y) b2)))))))

(defun var? (x)
  (and (symbolp x)
       (eql (char (symbol-name x) 0) #\?)))

(defun binding (x binds)
  (let ((b (assoc x binds)))
    (if b
        (or (binding (cdr b) binds)
        (cdr b)))))
```

Рис. 15.1. Функции для сопоставления.

```
> (match '(p a b c a) '(p ?x ?y c ?x))
((?Y . B) (?X . A))
T
> (match '(p ?x b ?y a) (p ?y b c a))
((?X . C) (?X . ?Y))
NIL
```

Путем поэлементного сравнения `match` ищет *связи* (*bindings*) переменных со значениями, которые помещает в параметр `binds`. Если совпадение успешно, то `match` возвращает найденные связи, в другом случае возвращает `nil`. Так как не из всякого успешного сопоставления можно извлечь подобные связи, необходимо отдельно информировать об успешности сопоставления. Для этого `match`, подобно `gethash`, возвращает еще один аргумент.

```
> (match '(p ?x) '(p ?x))
NIL
T
```

Словесно действия `match` можно выразить так:

1. Если `x = y` (`eql`), то они совпадают.
2. Если `x` уже ассоциирована со значением, то она совпадает с `y`, если они имеют одинаковое значение.
3. Если `y` уже ассоциирована со значением, то она совпадает с `x`, если они имеют одинаковое значение.
4. Если `x` не ассоциирована, то с ней связывается текущее значение.
5. Если `y` не ассоциирована, то с ней связывается текущее значение.
6. Два значения совпадают, если они оба cons-ячейки, их `car` совпадают, а `cdr` совпадают с учетом полученных связей.

Вот два примера, демонстрирующие все шесть правил:

```
> (match '(p ?v b ?x d (?z ?z))
```

```
          '(p a ?w c ?y ( e  e))
          '((?v . a) (?w . b)))
((?Z . E) (?Y . D) (?X . C) (?V . A) (?W . B))
T
```

Чтобы найти ассоциированное значение, `match` вызывает `binding`. Эта функция рекурсивна, так как в результате сопоставления могут получиться пары, в которых переменная связана со значением косвенно, например, так:

```
> (match '(?x a) '(?y ?y))
((?Y . A) (?X . ?Y))
T
```

Теперь мы видим косвенную связь переменной `?x` со значением, ведь `?x` совпадает с `?y`, а `?y` с `a`.

15.3. Ответы на вопросы

Теперь, когда введены основные концепции сопоставления, можем перейти непосредственно к назначению нашей программы: если мы имеем выражение, вероятно, содержащее переменные, то, исходя из имеющихся фактов и правил, мы можем найти все ассоциации, делающие это выражение истинным. Например, есть факт:

```
(parent donald nancy)
```

и мы спрашиваем программу:

```
(parent ?x ?y)
```

то она вернет нечто типа:

```
(((?x . donald) (?y . nancy)))
```

Это означает, что есть лишь один способ привести наше выражение к истине: `?x` – это `donald`, а `?y` – `nancy`.

Если получены подобные ассоциации, можно утверждать, что мы на правильном пути. Теперь нам нужно научиться определять правила. Соответствующий код приведен на рис. 15.2. Правила содержатся в хеш-таблице `*rules*` в соответствии с предикатами в заголовках. Таким образом мы устанавливаем, что переменные не могут находиться на месте предикатов. От этого ограничения можно избавиться, если хранить все правила в отдельном списке, но тогда для доказательства чего-либо нам пришлось бы сверяться с каждым правилом в этом списке.

```
(defvar *rules* (make-hash-table))

(defmacro <- (con &optional ant)
  `(length (push (cons (cdr ',con) ',ant)
                 (gethash (car ',con) *rules*))))
```

Рис. 15.2. Определение правил.

Мы будем пользоваться макросом `<-` для определения как правил, так и фактов. Факт представляется в виде правила, содержащего только заголовок. Это соответствует нашему представлению о правилах: так как для доказательства заголовка необходимо доказать тело, а раз тела нет, то и оказывать нечего. Вот два уже знакомых примера:

```
> (<- (parent donald nancy))
1
> (<- (child ?x ?y) (parent ?y ?x))
1
```

Вызов `<-` возвращает номер нового правила для заданного предиката. Оборачивание `length` вокруг `push` позволяет избежать большого объема информации, выводимой в toplevel.

На рис. 15.3 содержится большая часть кода, с помощью которого можно будет делать умозаключения. Функция `prove` является осью, вокруг которой вращается

весь логический вывод. Она принимает выражение и (необязательно) набор связей. Если выражение не содержит логических операторов, то вызывается `prove-simple`. Именно здесь происходит сам обратный вывод. Эта функция ищет все правила с истинным предикатом и пытается сопоставить заголовок каждого из них с фактом, который мы желаем доказать. Затем для каждого совпавшего заголовка доказывается его тело с учетом новых связей, созданных `match`. Вызовы `prove` возвращает списки связей, которые затем собираются `mapcan`:

```
(defun prove (expr &optional binds)
  (case (car expr)
    (and (prove-and (reverse (cdr expr)) binds))
    (or  (prove-or (cdr expr) binds))
    (not (prove-not (cadr expr) binds))
    (t   (prove-simple (car expr) (cdr expr) binds))))

(defun prove-simple (pred args binds)
  (mapcan #'(lambda (r)
              (multiple-value-bind (b2 yes)
                  (match args (car r) binds)
                (when yes
                  (if (cdr r)
                      (prove (cdr r) b2)
                      (list b2)))))
          (mapcar #'change-vars
                  (gethash pred *rules*))))

(defun change-vars (r)
  (sublis (mapcar #'(lambda (v) (cons v (gensym "?")))
                  (vars-in r))
          r))

(defun vars-in (expr)
  (if (atom expr)
      (if (var? expr) (list expr))
      (union (vars-in (car expr))
             (vars-in (cdr expr)))))
```

Рис. 15.3. Умозаключения.

```
> (prove-simple 'parent '(donald nancy) nil)
(NIL)
> (prove-simple 'child '(?x ?y) nil)
(((#:?6 . NANCY) (#:?5 . DONALD) (?Y . #:?5) (?X . #:?6)))
```

Оба полученных значения подтверждают, что есть лишь один путь доказательства. (Если доказать утверждение не получилось, возвращается `nil`.) Первый пример был выполнен без создания каких-либо связей, тогда как во втором примере переменные `?x` и `?y` были связаны (косвенно) с `nancy` и `donald`.

Мы видим здесь хороший пример высказанной ранее (см. стр. Ё041out) идеи: так как наша программа написана в функциональном стиле, мы можем тестировать каждую функцию отдельно.

Теперь пара слов о том, зачем нужен `gensym`. Раз мы используем переменные в правилах, нам нужно избегать наличия двух правил с одной и той же переменной. Рассмотрим два правила:

```
(<- (child ?x ?y) (parent ?y ?x))

(<- (daughter ?y ?x) (and (child ?y ?x) (female ?y)))
```

Они определяют связь между `x` и `y` внутри одного определения, и любое совпадение их имен в различных определениях не несет никакого смысла.

Если мы воспользуемся этими правилами в том виде, в каком только что их записали, они не будут работать. Если мы попытаемся доказать, что `a` – дочь `b`, то сопоставление с заголовком второго правила даст связи `?y` = `a` и `?x` = `b`. Имея такие связи, мы не сможем воспользоваться первым правилом:

```
> (match '(child ?y ?x)
         '(child ?x ?y)
         '((?y . a) (?x . b)))
NIL
```

Чтобы переменные действовали лишь в пределах одного правила, мы заменяем все переменные внутри правила на `gensym`. С этой целью определена функция `change-vars`. Так мы приобретаем защиту от совпадений с другими переменными в определениях других правил. Но так как правила могут применяться рекурсивно, то нам нужна защита при сопоставлении с этим же правилом. Для этого будем пользоваться `change-vars` не только при определении правил, но и при каждом их использовании.

Теперь остается лишь научиться доказывать составные утверждения. Соответствующий код приведен на рис. 15.4. Обработка `or` и `not` предельно проста. В первом случае мы собираем все связи, полученные из каждого выражения в `or`. Во втором случае мы просто возвращаем имеющиеся связи, если выражение внутри `not` ни к чему не привело.

```
(defun prove-and (clauses binds)
  (if (null clauses)
      (list binds)
      (mapcan #'(lambda (b)
                  (prove (car clauses) b))
              (prove-and (cdr clauses) binds))))

(defun prove-or (clauses binds)
  (mapcan #'(lambda (c) (prove c binds))
          clauses))

(defun prove-not (clause binds)
  (unless (prove clause binds)
    (list binds)))
```

Рис. 15.4. Логические операторы.

Функция `prove-and` чуть сложнее. Она работает как фильтр, доказывая первое выражение для каждого из наборов связей, полученных из остальных выражений. По этой причине выражения внутри `and` рассматриваются в обратном порядке. Переворачивание результата `prove-and` компенсирует это явление.

Теперь у нас есть рабочая программа, но она не удобна для конечного пользователя. Разбирать списки связей, возвращаемых `prove`, довольно сложно, а ведь с усложнением выражений они будут только расти. Эту проблему решает макрос `with-answer` на рис. 15.5. Он принимает запрос (не вычисленный) и вычисляет свое тело для каждого набора связей, полученных при обработке запроса, попутно позволяя использовать переменные из запроса внутри его тела:

```
(defmacro with-answer (query &body body)
  (let ((binds (gensym)))
    `(dolist (,binds (prove ',query))
       (let ,(mapcar #'(lambda (v)
                         `(,v (binding ',v ,binds)))
                     (vars-in query))
         ,@body))))
```

Рис. 15.5. Макрос, предоставляющий интерфейс.

```
> (with-answer (parent ?x ?y)
    (format t "~A is the parent of ~A.~%" ?x ?y))
DONALD is the parent of NANCY.
NIL
```

Этот макрос расшифровывает полученные связи, предоставляя удобный способ использования `prove` в других программах. Пример его раскрытия показан на рис. 15.6, а некоторые примеры использования – на рис. 15.7.

```
(with-answer (p ?x ?y)
  (f ?x ?y))

;;; раскрывается в:

(dolist (#:g1 (prove '(p ?x ?y)))
  (let ((?x (binding '?x #:g1))
        (?y (binding '?y #:g1)))
    (f ?x ?y)))
```

Рис. 15.6. Раскрытие вызова with-answer.

Если мы выполним `(clrhash *rules*)` и затем определим следующие правила и факты:

```
(<- (parent donald nancy))
(<- (parent donald debbie))
(<- (male donald))
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
(<- (= ?x ?y))
(<- (sibling ?x ?y) (and (parent ?z ?x)
                         (parent ?z ?y)
                         (not (= ?x ?y))))
```

то сможем сделать следующие умозаключения:

```
> (with-answer (father ?x ?y)
    (format t "~A is the father of ~A.~%" ?x ?y))
DONALD is the fther of DEBBIE.
DONALD is the father of NANCY.
NIL
> (with-answer (sibling ?x ?y)
    (format t "~A is the sibling of ~A.~%" ?x ?y))
DEBBIE is the sibling of NANCY.
NANCY is the sibling of DEBBIE.
NIL
```

Рис. 15.7. Использование программы.

15.4. Анализ

Может показаться, что написанный нами код является простым и естественным решением поставленной задачи. Но на деле он крайне неэффективен. Что мы в действительности сделали? написали интерпретатор. В то время как могли создать компилятор.

Приведем набросок его реализации. Основная идея заключается в запаковке всей программы в макросы `<-` и `with-answer`. В таком случае основная часть работы будет выполняться на этапе компиляции, тогда как сейчас она выполняется непосредственно во время запуска. (Зародыш этой идеи можно разглядеть в `avg` на стр. Ё042out.) Будем представлять правила как функции, а не как списки. Вместо функций типа `prove` и `prove-and`, интерпретирующих выражения, у нас будут функции для преобразования выражений в код. Выражения становятся доступны в момент определения правила. Зачем ждать, пока выражение будет проанализировано? Это относится и к `with-answer`, которая будет вызывать функции типа <- для генерации раскрытий.

Звучит устрашающе, но на деле реализация предложенной идеи займет всего в два-три раза больше времени. Читатели могут познакомиться с подобными методиками в книгах «On Lisp» и «Paradigms of Artificial Intelligence Programming»[1], которые содержат примеры программ наподобие этой.

[1] Peter Norvig. «Paradigms of Artificial Intelligence Programming», Morgan Kaufman, 1992. Эта книга, также известная как PAIP, рассматривает программирование задач искусственного интеллекта. Автор использует Common Lisp и сопровождает книгу большим количеством кода, который доступен по адресу: *http://norvig.com/paip/README.html*. - *Прим. перев.*

16. Пример: генерация HTML

В этой главе мы напишем небольшой генератор HTML – программу, автоматически производящую набор связанных web-страниц. Она не только демонстрирует различные концепции Лиспа, но и является хорошим примером разработки «снизу-вверх». Мы начнем с разработки утилит общего назначения, которые затем будем использовать как расширение языка, позволяющее написать полноценный генератор.

16.1. HTML

HTML (HyperText Markup Language, язык разметки гипертекста) – это то, из чего состоят web-страницы. Это очень простой язык, который не имеет продвинутых возможностей, зато легок в изучении. Описание основных моментов HTML дается в этом разделе.

Web-страницы просматриваются с помощью web-браузера. Он получает код страницы (как правило, с другого компьютера) и преобразовывает его в читаемый вид. HTML-файл состоит из *тегов*, воспринимаемых браузером как инструкции.

На рис. 16.1 приведен пример простого HTML-файла, а на рис. 16.2 показано, как он будет отображаться в браузере. Обратите внимание, что текст между угловыми скобками не отображается. Это и есть теги. HTML имеет два вида тегов. Одни используются попарно:

```
<center>
<h2>Your Fortune</h2>
</center>
<br><br>
Welcome to the home page of the Fortune Cookie
Institute. FCI is a non-profit institution
dedicated to the development of more realistic
fortunes. Here are some examples of fortunes
that fall within our guidelines:
<ol>
<li>Your nostril hairs will grow longer.
<li>You will never learn how to dress properly.
<li>Your car will be stolen.
<li>You will gain weight.
</ol>
Click <a href="research.html">here</a> to learn
more about our ongoing research projects.
```

Рис. 16.1. HTML-файл.

```
Your Fortune
Welcome to the home page of the Fortune Cookie Institute.
FCI is a non-
profit institution dedicated to the development of more
realistic fortunes.
Here are some examples of fortunes that fall within our
guidelines:
  1. Your nostril hairs will grow longer.
  2. You will never learn how to dress properly.
  3. Your car will be stolen.
  4. You will gain weight.
Click here to learn more about our ongoing research
projects.
```

Рис. 16.2. Внешний вид web-страницы.

```
<tag> ... </tag>
```

Первый тег обозначает начало некоторого окружения, второй помечает его окончание. Одним из тегов такого рода является `<h2>`. Весь текст, находящийся между тегами `<h2>` и `</h2>`, отображается увеличенным шрифтом. (Наибольший шрифт задается `<h1>`.)

Другими парными тегами являются: `<ol>` («ordered list»), создающий нумерованный список; `<center>`, центрирующий текст; `<a...>` («anchor»), создающий ссылку.

Именно ссылки являются сутью гипертекста. Текст, находящийся между `<a...>` и `</a>`, отображается браузерами особым образом, как правило, с подчеркиванием. При нажатии на ссылку осуществляется переход на другую страницу, адрес которой содержится внутри тега:

```
<a href="foo.html">
```

означает ссылку на другую страницу, находящуюся в той же директории. При нажатии на ссылку на рис. 16.2 браузер загрузит и отобразит файл `"research.html"`.

Ссылки не обязаны указывать на файлы в той же директории, но могут указывать на любой адрес в интернете (хотя в нашем примере это не используется).

Другая разновидность тегов не имеет маркера окончания. В нашем примере (см. рис. 16.1) используются теги `<br>` ("break") для перехода на новую строку и `<li>` ("list item") для обозначения элемента внутри окружения списка. Разумеется, HTML содержит и другие теги, но в этой главе мы постараемся не использовать что-либо, не упомянутое на рис. 16.1.

16.2. Утилиты HTML

Определим некоторые утилиты для генерации HTML. На рис. 16.3 показаны базовые утилиты для генерации тегов. По умолчанию вывод направляется в `*standard-output*`, но мы всегда сможем перенаправить его.

```
(defmacro as (tag content)
  `(format t "<~(~A~)>~A</~(~A~)>"
    ',tag ,content ',tag))

(defmacro with (tag &rest body)
  `(progn
    (format t "~&<~(~A~)>~%" ',tag)
    ,@body
    (format t "~&<~(~A~)>~%" ',tag)))

(defun brs (&optional (n 1))
  (fresh-line)
  (dotimes (i n)
    (princ "<br>"))
  (terpri))
```

Рис. 16.3. Утилиты для генерации тегов.

Макросы `as` и `with` предназначены для генерации выражений, окруженных парой тегов. Первый принимает строку и печатает ее между тегами:

```
> (as center "The Missing Lambda")
<center>The Missing Lambda</center>
NIL
```

Второй принимает код и выводит результат его выполнения между тегами:

```
> (with center
    (princ "The Unbalanced Parenthesis"))
<center>
The Unbalanced Parenthesis
</center>
NIL
```

Управляющие директивы `~(` и `~)` используются в них для генерации тегов в нижнем регистре. HTML не чувствителен к регистру, но теги в нижнем регистре легче воспринимаются, особенно когда их много.

Макрос `as` пытается разместить весь вывод в одну строчку, а `with` располагает теги на отдельных строках. (Директива `~&` проверяет, начинается ли вывод с новой строки.) Это делается лишь для того, чтобы сделать генерируемый код более читаемым. При обработке HTML пробелы около тегов не воспринимаются. Последняя утилита на рис. 16.3, `brs`, производит код нескольких разрывов строк. Иногда это используется для выполнения вертикальных отступов.

Рис. 16.4 содержит утилиты для непосредственной генерации HTML. Первая возвращает имя файла по заданному символу, присоединяя к его имени расширение `".html"` В полноценном приложении должна быть возможность задания полных путей.

```
(defun html-file (base)
   (format nil "~(~A~).html" base))

(defmacro page (name title &rest body)
  (let ((ti (gensym)))
    `(with-open-file (*standard-output*
                      (html-file ,name)
                      :direction :output
                      :if-exists :supersede)
       (let ((,ti ,title))
         (as title ,ti)
         (with center
           (as h2 (string-upcase ,ti)))
         (brs 3)
         ,@body))))
```

Рис. 16.4. Утилиты создания файлов.

Макрос `page` служит для генерации всей страницы. Он похож по устройству на используемый в нем `with-open-file`. При открытии файла `name` переменная `*standard-output*` будет связана с полученным потоком.

В разделе 6.7 было показано, как присваивать временные значения специальным переменным. В примере на стр. Ё043out связь `*print-base*` с `16` выполнялось посредством `let`, и была актуально внутри него. Раскрытие `page` похожим образом связывает `*standard-output*` с потоком, указывающим на файл. В результате вызова `as` или `princ` вывод будет перенаправлен в соответствующий файл.

Заголовок `title` будет напечатан в самом верху страницы. За ним будет следовать весь остальной вывод. Так, вызов:

```
(page 'paren "The Unbalanced Parenthesis"
  (princ "Something in his expression told her..."))
```

приведет к тому, что файл `"paren.html"` будет иметь следующее содержание:

```
<title>The Unbalanced Parenthesis</title>
<center>
<h2>THE UNBALANCED PARENTHESIS</h2>
<center>
<br><br><br>
Something in his expression told her...
```

Нам знакомы все теги, кроме `<title>`. Текст, заворачиваемый в тег `<title>` не будет появляться нигде на странице, зато будет отображаться браузером в заголовке окна.

На рис. 16.5 представлены утилиты для генерации ссылок. Макрос `with-link` похож на `with`. Он заключает значение заданного кода в соответствующий тег и устанавливая адрес файла-ссылки, полученный из второго аргумента:

```
(defmacro with-link (dest &rest body)
  `(progn
     (format t "<a href=\"~A\">" (html-file ,dest))
     ,@body
     (princ "</a>")))

(defun link-item (dest text)
  (princ "<li>")
  (with-link dest
    (princ text)))

(defun button (dest text)
  (princ "[ ")
  (with-link dest
    (princ text))
  (format t " ]~%"))
```

Рис. 16.5. Утилиты для генерации ссылок.

```
> (with-link 'capture
    (princ "The Captured Variable"))
<a href="capture.html">The Captured Variable</a>
"</a>"
```

Он используется в функции `link-item`, производящей элемент списка, также являющийся ссылкой:

```
> (link-item 'bq "Backquote!")
<li><a href="bq.html"> Backquote!</a>
"</a>"
```

и в функции `button`, производящей ссылку внутри квадратных скобок, что напоминает кнопку:

```
> (button 'help "Help")
[ <a href="help.html">Help</a> ]
NIL
```

16.3. Итерационная утилита

В данном разделе вы остановимся на утилитах общего назначения. Можем ли мы заранее знать, что в процессе разработки нам понадобится та или иная утилита? Нет, не можем. Обычно необходимость утилит для написания программы проясняется в самом процессе ее написания. Процесс разработки, во время которого программист отвлекается на написание необходимых утилит, представить на страницах книги нелегко. Поэтому ограничимся оговоркой, что написание программ никогда не будет прямолинейным процессом, но будет включать неоднократное полное или частичное переписывание кода.

Наша новая утилита будет похожа на `mapc`. Она показана на рис. 16.6. Она принимает функцию трех аргументов и список. Для каждого элемента функция применяется для самого элемента, а также предшествующего и последующего элементов. (Если предшествующего или последующего элемента нет, то используется `nil`.)

```
(defun map3 (fn 1st)
  (labels ((rec (curr prev next left)
             (funcall fn curr prev next)
             (when left
               (rec (car left)
                    curr
                    (cadr left)
                    (cdr left)))))
    (when 1st
      (rec (car 1st) nil (cadr 1st) (cdr 1st)))))
```

Рис. 16.6. Итерация по деревьям.

```
> (map3 #'(lambda (&rest args) (princ args))
        '(a b c d))
(A NIL B) (B A C) (C B D) (D C NIL)
NIL
```

Как и `mapc`, она всегда возвращает `nil`. Ситуации, которые могут быть разрешены с ее помощью, возникают довольно часто. С одной из них мы познакомимся в следующем разделе, где нам понадобятся ссылки на предыдущую и следующую страницы.

Представьте, что вам нужно сделать что-либо в промежутке между каждой парой элементов списка:

```
> (map3 #'(lambda (c p n)
            (princ c)
            (if  (princ " | ")))
        '(a b c d))
A | B | C | D
NIL
```

Подобные задачи встречаются довольно часто, но языку программирования вовсе не достаточно иметь встроенный оператор для подобных нужд, ведь неизвестно, какой оператор окажется полезным. Гораздо приятнее иметь возможность создавать необходимые инструменты самостоятельно.

16.4. Генерация страниц

Как книги и журналы, коллекции web-страниц обычно имеют древовидную организацию. Книга часто состоит из глав, состоящих из разделов, в свою очередь состоящих из подразделов, и так далее. Web-страницы часто организуются подобным образом, хотя и не имеют подобных названий.

В этом разделе мы построим программу, производящую коллекции web-страниц, имеющие следующую структуру. Первая страница представляет оглавление, содержащее ссылки на разделы, которые, в свою очередь, содержат ссылки на пункты, являющиеся обычным текстом.

Каждая страница должна иметь ссылки назад, вперед и наверх. Ссылки вперед и назад осуществляют переход в пределах одного уровня вложенности. Например, если ссылка вперед на странице, представляющей пункт, будет указывать на следующий пункт данного раздела. По ссылке вверх осуществляется переход от пункта к разделу, а от раздела к оглавлению. Кроме того, имеется предметный индекс, перечисляющий все пункты в алфавитном порядке. Описанная иерархия изображена на рис. 16.7.

Рис. 16.7. Структура сайта.

Необходимые для создания страниц операторы и структуры данных представлены на рис. 16.8. Наша программа будет работать с двумя типами объектов: пунктами, содержащими блоки текста, и разделами, содержащими списки ссылок на пункты.

```
(defparameter *sections* nil)

(defstruct item
  id title text)

(defstruct section
  id title items)

(defmacro defitem (id title text)
  `(setf ,id
         (make-item :id     ',id
                    :title  ,title
                    :text   ,text)))

(defmacro defsection (id title &rest items)
  `(setf ,id
         (make-section :id    ',id
                       :title ,title
                       :items (list ,@items))))

(defun defsite (&rest sections)
  (setf *sections* sections))
```

Рис. 16.8. Создание сайта.

Разделы и пункты содержат поле `id` – символы, используемые с двумя целями. Одно их назначение мы видим в определениях `defitem` и `defsection`: это уникальное имя, по которому мы будем ссылаться на пункт или раздел. Кроме того, мы будем использовать `id` в имени файла, представляющего пункт или раздел. Так, страница, представляющая пункт `foo`, будет записана в файл `"foo.html"`.

Как разделы, так и пункты имеют поле `title`, которое представляет заголовок страницы и должно быть строкой.

Порядок следования пунктов в разделе определяется из аргументов `defsection`, порядок следования разделов – из аргументов `defsite`.

Функции, производящие оглавление и индекс, представлены на рис. 16.9. Константы `contents` и `index` являются строками, служащими заголовками этих страниц и именами соответствующих файлов, похожи в общих чертах. Они открывают HTML-файл, генерируют заголовок и список ссылок. Разница между ними в том, что в `index` этот список отсортирован. Он строится с помощью функции `all-items`, которая использует функцию упорядочения `title<`. Важно, что все заголовки сравниваются функцией `string-lessp`, которая, в отличие от `string<`, игнорирует регистр знаков.

```
(defconstant contents "contents")
(defconstant index    "index")

(defun gen-contents (&optional (sections *sections*))
  (page contents contents
    (with ol
      (dolist (s sections)
        (link-item (section-id s) (section-title s))
        (brs 2))
      (link-item index (string-capitalize index)))))

(defun gen-index (&optional (sections *sections*))
  (page index index
    (with ol
      (dolist (i (all-items sections))
        (link-item (item-id i) (item-title i))
        (brs 2)))))

(defun all-items (sections)
  (let ((is nil))
    (dolist (s sections)
      (dolist (i (section-items s))
        (setf is (merge 'list (list i) is #'title<))))
    is))

(defun title< (x y)
  (string-lessp (item-title x) (item-title y)))
```

Рис. 16.9. Генерация индекса и оглавления.

В полноценном приложении сравнение может быть более сложным, игнорируя, например, артикли «a» и «the».[1]

Оставшийся код показан на рис. 16.10: `gen-site` генерирует всю коллекцию страниц, остальные функции – отдельные ее части.

```
(defun gen-site ()
  (map3 #'gen-section *sections*)
  (gen-contents)
  (gen-index))

(defun gen-section (sect <sect sect>)
  (page (section-id sect) (section-title sect)
    (with ol
      (map3 #'(lambda (item <item item>)
                (link-item (item-id item)
                           (item-title item))
                (brs 2)
                (gen-item sect item <item item>))
            (section-items sect)))
    (brs 3)
    (gen-move-buttons (if <sect (section-id <sect))
                      contents
                      (if sect> (section-id sect>)))))

(defun gen-item (sect item <item item>)
  (page (item-id item) (item-title item)
    (princ (item-text item))
    (brs 3)
    (gen-move-buttons (if <item (item-id <item))
                      (section-id sect)
                      (if item> (item-id item>)))))
```

[1] Или предлоги для русскоязычных сайтов. – *Прим. перев.*

```
(defun gen-move-buttons (back up forward)
  (if back (button back "Back"))
  (if up (button up "Up"))
  (if forward (button forward "Forward")))
```

Рис. 16.10. Генерация сайта, разделов и пунктов.

Под коллекцией понимается оглавление, индекс, страницы разделов и пунктов. Средства генерации оглавления и индекса показаны на рис. 16.9. Разделы и пункты генерируются `gen-section`, которая производит страницу раздела и вызывает `gen-item` для генерации страниц, представляющих каждый пункт в данном разделе.

Обе эти функции начинаются и заканчиваются сходным образом; принимают аргументы, представляющие сам объект, а также предыдущий и последующий объекты того же уровня вложенности; заканчиваются вызовом `gen-move-buttons` для генерации кнопок, ведущих вперед, назад и вверх. Но, в отличие от `gen-section`, которая производит упорядоченный список ссылок на пункты, `gen-section` просто отправляет текст в выходной файл.

Содержимое каждого пункта полностью ложится на пользователя. Полноценные страницы построены на основе HTML-тегов и могут быть сгенерированы с помощью определенных нами утилит или отдельной программы.

Пример ручного создания небольшой коллекции страниц приведен на рис. 16.11. В нем приводится перечень недавних публикаций в Институте Удачи[1].

```
(defitem des "Fortune Cookies: Dessert or Fraud?" "...")
(defitem case "The Case for Pessimism" "...")
(defsection position "Position Papers" des case)
(defitem luck "Distribution of Bad Luck" "...")
(defitem haz "Health Hazards of Optimism" "...")
(defsection abstract "Research Abstracts" luck haz)
(defsite position abstract)
```

Рис. 16.11. Небольшой сайт.

[1] Игра слов. В оригинале – Fortune Cookie Institute. Печеньки судьбы (fortune cookies) – распространенное на Западе развлечение, запекание бумажек с предсказаниями в печенье. – *Прим. перев.*

17. Пример: объекты

В этой главе мы собираемся реализовать поверх Лиспа свой собственный объектно-ориентированный язык. Такие программы еще называют *встроенными языками* (*emmedded language*). Встраивание объектно-ориентированного языка в Лисп – отличный пример такой задачи, не только подтверждающий упомянутые ранее особенности Лиспа, но и демонстрирующий, как легко объектно-ориентированные абстракции ложатся на его фундаментальные абстракции.

17.1. Наследование

В разделе 11.10 объяснялось различие между моделями обобщенных функций и передачи сообщений. В модели передачи сообщений объекты:

1. Имеют свойства
2. Реагируют на сообщения
3. Наследуют свойства и методы от предков

CLOS, разумеется, реализует модель обобщенных функций. Но в этой главе мы будем сочинять минималистичную объектную систему, не претендующую на соперничество с CLOS. По этой причине мы будем реализовывать более старую модель.

Лисп предоставляет несколько способов хранения наборов свойств. Одним из них являются хеш-таблицы, доступ к свойствам в которых осуществляется с помощью `gethash`:

```
(gethash 'color obj)
```

Раз функции – те же данные, мы также можем хранить их как свойства. Точно так же можем хранить и методы. Чтобы воспользоваться определенным методом, принадлежащим объекту, необходимо применить `funcall` к свойству с данным именем:

```
(funcall (gethash 'move obj) obj 10)
```

Передачу сообщения в стиле Smalltalk мы можем осуществить так:

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

Теперь, чтобы сообщить `obj` перемещение (`move`) на `10`, скажем:

```
(tell obj 'move 10)
```

Пока в нашей реализации не хватает поддержки наследования. Реализация простейшего ее варианта представлена на рис. 17.1. (Имя `rget` происходит от recursive get.) Теперь, имея за плечами восемь строк кода, мы получаем все три вышеперечисленных элемента объектной ориентированности.

```
(defun rget (prop obj)
  (multiple-value-bind (val in) (gethash prop obj)
    (if in
        (values val in)
        (let ((par (gethash :parent obj)))
          (and par (rget prop par))))))

(defun tell (obj message &rest args)
  (apply (rget message obj) obj args))
```

Рис. 17.1. Наследование.

Нам не терпится испробовать этот код в деле. Применим его к рассмотренному ранее примеру. Создадим два объекта, один из которых является потомком другого:

```
> (setf circle-class (make-hash-table)
        our-circle   (make-hash-table)
        (gethash :parent our-circle) circle-class
        (gethash 'radius our-circle) 2)
2
```

Объект `circle-class` будет хранить методы `area` для всех кругов. Это будут функции одного аргумента – объекта, которому посылается сообщение:

```
> (setf (gethash 'area circle-class)
        #'(lambda (x)
            (* pi (expt (rget 'radius x) 2))))
#<Interpreted-Function BF1EF6>
```

Теперь запросим площадь `our-circle`. Она будет вычисляться в соответствии с только что определенным методом: `rget` считывает свойство, а `tell` применяет метод.

```
> (rget 'radius our-circle)
2
T
> (tell our-circle 'area)
12.566370614359173
```

Перед тем, как взять в руки напильник, разберемся с тем, что уже сделано. Те самые восемь строчек кода сделали бы древнюю (до CLOS) версию Лиспа объектно-ориентированным языком. Как нам удалось решить столь нелегкую задачу? Должно быть, это какой-то трюк!

Да, трюк здесь определенно присутствует, но к нашему коду он отношения не имеет. Дело в том, что Лисп в некотором смысле уже является объектно-ориентированным. Все, что нам понадобилось – добавить несколько абстракций.

17.2. Множественное наследование

Пока мы умеем осуществлять лишь однократное наследование – объект может иметь лишь одного родителя. Это легко исправить, сделав `parent` списком, а так же переопределив `rget`, как показано на рис. 17.2.

```
(defun rget (prop obj)
  (dolist (c (precedence obj))
    (multiple-value-bind (val in) (gethash prop c)
      (if in (return (values val in))))))

(defun precedence (obj)
  (labels ((traverse (x)
             (cons x
                   (mapcan #'traverse
                           (gethash :parents x)))))
    (delete-duplicates (1traverse obj))))
```

Рис. 17.2. Множественное наследование.

С однократным наследованием, желая получить некоторое свойство объекта, нам достаточно пройтись рекурсивно по всем его предшественникам. Если текущий объект не содержит достаточной информации об интересующем нас свойстве, мы переходим к его родителю, и так далее. Имея множественное наследование, нам приходится выполнять похожую работу на графе, а не простом дереве, и обычный поиск в глубину здесь не поможет. Множественное наследование позволяет реализовать, например, такую иерархию, как на рис. 17.3: к `a` мы можем спуститься от `b` и `c`, а к ним – от `d`. Поиск в глубину (здесь, скорее, в высоту) даст следующий порядок обхода: `a`, `b`, `d`, `c`, `d`. Если желаемое свойство имеется в `d` и `c`, то мы получим значение из `d`, а не `c`, что не будет соответствовать правилу, по которому значение из подкласса приоритетнее, чем из его родителей.

Рис. 17.3. Несколько путей к одному суперклассу

В нашем случае порядок обхода должен быть таким: `a`, `b`, `c`, `d`. Необходимо четко следить, чтобы никакой объект не обрабатывался ранее, чем все его потомки. Как

это гарантировать? Простейший способ – собрать список из объекта и всех его предков, следующих в правильном порядке, затем проверять по очереди элементы списка.

Такой список производит функция `precedence`. Она начинается с вызова `traverse`, выполняющего поиск в глубину. Если несколько объектов имеют общего предка, то он встретится в списке несколько раз. Сохраняя лишь последний из каждого набора повторений, мы получим порядок, соответствующий CLOS. (Удаление всех повторений, кроме последнего, соответствует правилу 3 на стр. Ё044out.) С этой задачей справляется имеющаяся в Common Lisp функция `delete-duplicates`. После того, как список предшествования создан, `rget` ищет в нем первый объект с искомым свойством.

Используя идею предшествования, мы можем сказать, например, что патриотичный негодяй является прежде всего негодяем и лишь потом патриотом:[1]

```
> (setf scoundrel           (make-hash-table)
        patriot             (make-hash-table)
        patriotic-scoundrel (make-hash-table)
        (gethash 'serves scoundrel) 'self
        (gethash 'serves patriot)   'country
        (gethash :parents patriotic-scoundrel) (list scoundrel
patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget 'serves patriotic-scoundrel)
SELF
T
```

Да, сейчас мы имеем достаточно мощную систему, но пользоваться ей совершенно неудобно. Пора приступать ко второй стадии написания программы – приведению ее наброска в вид, пригодный для использования.

17.3. Определение объектов

Во-первых, нам нужен нормальный способ создания объектов, скрывающий их истинное устройство. Мы определим функцию для построения объектов, одним вызовом которой можно создать объект и определить его предков. Тогда мы сможем строить список предшествования для объекта лишь однажды при его создании, а не каждый раз при поиске свойств.

Мы будем управлять существующим списком предшествования, а не создавать его заново. Здесь мы сталкиваемся с задачей обновления устаревшей информации в списке. Да, на перестройку списка будут затрачиваться некоторые ресурсы, но мы все равно останемся в выигрыше, так как обращение к свойствам намного более частая операция, чем добавление новых предков. Кроме того, это нисколько не убавит гибкости нашей программы, мы просто переносим затраты с часто используемой операции на редко используемую.

На рис. 17.4. показан новый код.°Ё114in Глобальная переменная `*objs*` является списком всех объектов. Функция `parent` получает предков заданного объекта; `(setf parents)` не только назначает новых предков, но и вызывает `make-precedence` для перестройки любого списка предшествования, который также мог поменяться. Списки, как и прежде, строятся с помощью `precedence`.

[1] То есть, служит (serves) сначала себе (self), и лишь затем стране (country), что и продемонстрировано в данном примере. – *Прим. перев.*

```
(defvar *objs* nil)

(defun parents (obj) (gethash :parents obj))

(defun (setf parents) (val obj)
  (prog1 (setf (gethash :parents obj) val)
         (make-precedence obj)))

(defun make-precedence (obj)
  (setf (gethash ipreclist obj) (precedence obj))
  (dolist (x *objs*)
    (if (member obj (gethash :preclist x))
        (setf (gethash :preclist x) (precedence x)))))

(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (push obj *objs*)
    (setf (parents obj) parents)
    obj))

(defun rget (prop obj)
  (dolist (c (gethash :preclist obj))
    (multiple-value-bind (val in) (gethash prop c)
      (if in (return (values val in))))))
```

Рис. 17.4. Создание объектов.

Теперь вместо создания хеш-таблиц пользователи могут вызывать `obj`, который создает новый объект и определяет всех его предков. Кроме того, нам потребовалось переопределить `rget`, чтобы пользоваться уже существующими списками свойств.

17.4. Функциональный синтаксис

Еще одному усовершенствованию может подлежать синтаксис передачи сообщений. Использование `tell` не только загромождает запись, но и лишает нас возможности в полной мере пользоваться префиксной нотацией:

```
(tell (tell obj 'find-owner) 'find-owner)
```

Мы можем избавиться от `tell`, если сможем использовать имена свойств как функции. Определим для этого макрос `defprop` (рис. 17.5). Необязательный аргумент `meth?`, будучи истинным, сигнализирует, что свойство должно считаться методом. В противном случае оно будет считаться слотом, и значение, получаемое `rget`, будет просто возвращаться.

```
(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj)))
     (defun (setf ,name) (val obj)
       (setf (gethash ',name obj) val))))

(defun run-methods (obj name args)
  (let ((meth (rget name obj)))
    (if meth
        (apply meth obj args)
        (error "No ~A method for "A." name obj))))
```

Рис. 17.5. Функциональный синтаксис.

Теперь, когда мы определили имя свойства:

```
(defprop find-owner t)
```

мы можем сослаться на него через вызов функции:

```
(find-owner (find-owner obj))
```

Теперь наш предыдущий пример станет более понятным:

```
> (progn
    (setf scoundrel           (obj)
          patriot             (obj)
          patriotic-scoundrel (obj scoundrel patriot))
    (defprop serves)
    (setf (serves scoundrel) 'self
          (serves patriot)   'country)
    (serves patriotic-scoundrel))
SELF
T
```

17.5. Определение методов

До сих пор мы определяли методы следующим образом:

```
(defprop area t)

(setf circle-class (obj))

(setf (aref circle-class)
      #'(lambda (c) (* pi (expt (radius c) 2))))
```

Внутри метода мы можем выполнять нечто типа `call-next-method`, вызывая первый метод, который найдем в `cdr` списка предшествования. Так, например, если мы захотим печатать что-либо во время вычисления площади круга, мы скажем:

```
(setf grumpy-circle (obj circle-class))

(setf (area grumpy-circle)
      #'(lambda (c)
          (format t "How dare you stereotype me!~%")
          (funcall (some #'(lambda (x) (gethash 'area x))
                         (cdr (gethash :preclist c)))
                   c)))
```

Данный `funcall` эквивалентен `call-next-method`, но не прячет свои внутренности.

Макрос `defmeth` на рис. 17.6 предоставляет более удобный способ создания методов, а также упрощает дальнейшее их использование. Вызов `defmeth` раскрывается в `setf`, внутри которого находится `labels`-выражение, задающее

функцию next, которая работает наподобие next-method-p (см. стр. Ё045out), но возвращает объект, который может быть вызван, то есть, служит еще и как call-next-method.°Ё115in Теперь мы можем задать предшествование двух методов:

```
(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (setf (gethash ',name ,gobj)
       (labels ((next () (get-next ,gobj ',name)))
         #'(lambda ,parms ,@body))))))

(defun get-next (obj name)
  (some #'(lambda (x) (gethash name x))
        (cdr (gethash :preclist obj))))
```

Рис. 17.6. Определение методов.

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))

(defmeth area grumpy-circle (c)
  (format t "How dare you stereotype me!~%")
  (funcall (next) c))
```

Обратите внимание, каким образом в определении defmeth используется захват символов. Тело метода вызывается там, где имеется локальное определение next.

17.6. Экземпляры

До настоящего времени мы не делали различий между классами и экземплярами, используя для них общий термин: *объект*. Конечно, такая унификация дает определенную гибкость и удобство, но, вместе с тем, это совершенно неэффективно. В большинстве объектно-ориентированных приложений граф наследования приплюснут книзу. Например, в задаче симуляции трафика типы транспортных средств представляются менее чем десятью классами, а вот самих их тысячи. Если все они будут размещаться в нескольких списках предшествования, затраты на создание и хранение этих списков будут существенными.

На рис. 17.7 представлен макрос inst, производящий экземпляры. Экземпляры подобны объектам (которые теперь можем называть классами), но имеют лишь одного предка и не содержат списка предшествования. Кроме того, они не включаются в список *objs*.

```
(defun inst (parent)
  (let ((obj (make-hash-table)))
    (setf (gethash :parents obj) parent)
  obj))

(defun rget (prop obj)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (dolist (c prec)
          (multiple-value-bind (val in) (gethash prop c)
            (if in (return (values val in)))))
        (multiple-value-bind (val in) (gethash prop obj)
          (if in
              (values val in)
              (rget prop (gethash :parents obj)))))))

(defun get-next (obj name)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (some #'(lambda (x) (gethash name x))
              (cdr prec))
        (get-next (gethash obj :parents) name))))
```

Рис. 17.7. Создание экземпляров.

Теперь наш предыдущий пример может быть несколько изменен:

```
(setf grumpy-circle (inst circle-class))
```

Так как некоторые объекты отныне не содержат списков предшествования, то мы переопределили `rget` и `get-next`Ё046in, чтобы они могли сразу получать доступ к единственному предку. Прирост в производительности никак не сказался на гибкости. Мы можем делать с экземплярами все, что могли делать с любыми объектами, в том числе создание экземпляров и переназначение предков. В последнем случае `(setf parents)` преобразует объект в класс.

17.7. Новая реализация

Ни одно из наших улучшений не сказалось отрицательно на гибкости программы. Обычно на последних этапах разработки Лисп-программа жертвует частью своей гибкости на благо эффективности. И наша программа не исключение. Представив все объекты в виде хеш-таблиц, мы не только получаем избыточную гибкость, но и платим за это излишнюю цену. В этом разделе мы перепишем нашу программу, представляя объекты в виде векторов.

Да, мы отказываемся от возможности добавлять новые свойства на лету, но мы по-прежнему сможем делать это переопределением объектов. При создании класса нам потребуется передать ему список новых свойств, а при создании экземпляров они будут получать свойства благодаря наследованию.

В предыдущей реализации у нас не было полноценного деления на классы и экземпляры. Экземпляром считался класс, имеющий лишь одного предка. Переназначение предков превращало экземпляр в класс. В новой реализации у нас будет полноценное деление между этими двумя типами объектов, но более не будет возможности преобразовывать экземпляры в классы.

Код на рис. 17.8-17.10 представляет полностью новую реализацию. Рис. 17.8 содержит новые операторы для создания классов и экземпляров. Они представляются в виде векторов. В них первые три элемента содержат информацию, необходимую самой программе, и первые три макроса на рис. 17.8. осуществляют доступ к ним:

```
(defmacro parents (v) `(svref ,v 0))
(defmacro layout (v) `(the simple-vector (svref ,v 1)))
(defmacro preclist (v) `(svref ,v 2))

(defmacro class (&optional parents &rest props)
  `(class-fn (list ,@parents) ',props))

(defun class-fn (parents props)
  (let* ((all (union (inherit-props parents) props))
         (obj (make-array (+ (length all) 3)
                          :initial-element :nil)))
    (setf (parents obj)  parents
          (layout obj)   (coerce all 'simple-vector)
          (preclist obj) (precedence obj))
    obj))

(defun inherit-props (classes)
  (delete-duplicates
    (mapcan #'(lambda (c)
                (nconc (coerce (layout c) 'list)
                       (inherit-props (parents c))))
            classes)))

(defun precedence (obj)
  (labels ((traverse (x)
             (cons x
                   (mapcan #'traverse (parents x)))))
    (delete-duplicates (traverse obj))))

(defun inst (parent)
  (let ((obj (copy-seq parent)))
    (setf (parents obj)  parent
          (preclist obj) nil)
    (fill obj :nil :start 3)
    obj))
```

Рис. 17.8. Реализация с помощью векторов: создание объектов.

1. Поле `parents` аналогично ключу `:parents` в ранее использовавшихся хеш-таблицах. Для класса оно будет содержать список родительских классов, для экземпляра – один родительский класс.
2. Поле `layout` будет содержать вектор с именами свойств, принадлежность к классу или экземпляру будет определяться значением четвертого элемента вектора.
3. Поле `preclist` аналогично ключу `:preclist` в ранее использовавшихся хеш-таблицах. Для класса оно будет содержать список предшествования, для экземпляра – `nil`.

Так как эти операторы – макросы, то они могут быть аргументами `setf` (см. раздел 10.6).

Макрос `class` предназначен для создания классов. Он принимает необязательный аргумент – список суперклассов, за которым могут следовать имена свойств. Макрос возвращает объект, представляющий класс. В новом классе будут объединены его собственные свойства и унаследованные от суперклассов.

```
> (setf *print-array* nil
        geom-class   (class nil area)
        circle-class (class (geom-class) radius))
#<Simple-Vector T 5 C6205E>
```

Здесь мы создали два класса: `geom-class` с одним свойством `area` и без суперклассов, а также `circle-class`, являющийся подклассом `geom-class` и

имеющий дополнительное свойство radius.[1] Функция layout позволяет увидеть значащие поля класса (все, не считая первых трех):[2]

```
> (coerce (layout circle-class) 'list)
(AREA RADIUS)
```

Макрос class предоставляет интерфейс к функции class-fn, которая выполняет основную работу. Она вызывает inherit-props для сборки списка свойств всех предков, создает вектор необходимой длины и устанавливает три первых его элемента. (preclist строится с помощью функции precedence, которую мы практически не изменили.) Остальные поля класса устанавливаются в :nil, что сигнализирует об их незанятости. Получить свойство area класса circle-class можно так:

```
> (svref circle-class
         (+ (position 'area (layout circle-class)) 3))
:NIL
```

Позже мы определим функции, автоматизирующие эту процедуру.

Наконец, функция inst используется для создания экземпляров. Она не обязана быть макросом, так как принимает лишь один аргумент:

```
> (setf our-circle (inst circle-class))
#<Simple-Vector T 5 C6464E>
```

Полезно сравнить inst и class-fn, которые выполняют сходные задачи. Так как экземпляры имеют лишь одного родителя, то нет необходимости определять, какие свойства наследуются. Экземпляр – всего лишь копия значащих полей родительского класса. Кроме того, нет необходимости строить список предшествования, так как у экземпляра его попросту нет. Поэтому создание экземпляров намного быстрее создания классов, и это правильно, ведь как правило новые экземпляры создаются куда чаще, чем новые классы.

Теперь мы сможем построить иерархию классов и экземпляров. Для этого нам потребуются функции чтения и записи свойств. Первой функцией на рис. 17.9 является новая версия rget. По форме она напоминает rget с рис. 17.7. Ее поведение различается для классов и экземпляров.

---

[1] При отображении классов *print-array* должен быть nil. Первый элемент в списке preclist любого класса сам является классом, поэтому попытка отображения такой структуры приведет к попаданию в бесконечный цикл.

[2] Вектор приводится к списку для просмотра его содержимого, так как с *print-array* равным nil содержимое вектора не отображается.

```
(declaim (inline lookup (setf lookup)))

(defun rget (prop obj next?)
  (let ((prec (preclist obj)))
    (if prec
        (dolist (c (if next? (cdr prec) prec) :nil)
          (let ((val (lookup prop c)))
            (unless (eq val :nil) (return val))))
        (let ((val (lookup prop obj)))
          (if (eq val :nil)
              (rget prop (parents obj) nil)
              val)))))

(defun lookup (prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off (svref obj (+ off 3)) :nil)))

(defun (setf lookup) (val prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off
        (setf (svref obj (+ off 3)) val)
        (error "Can't set ~A of ~A." val obj))))
```

Рис. 17.9. Реализация с помощью векторов: доступ.

1. Если объект является классом, тогда обходим его список предшествования до тех пор, пока не найдем объект, имеющий значение искомого свойства, не равное `:nil`. Если такой объект не найден, возвращаем `:nil`.
2. Если объект является экземпляром, то ищем лишь среди его свойств, а если не находим, то рекурсивно вызываем `rget`.

У `rget` появился новый, третий аргумент – `next?`, который будет объяснен позже. Сейчас достаточно сказать, что если он `nil`, то `rget` ведет себя как обычно.

Функция `lookup` и обратная ей играют роль `gethash` в старой версии. Они ищут в объекте свойство с заданным именем или устанавливают новое. Этот запрос эквивалентен сделанному ранее:

```
> (lookup 'area circle-class)
:NIL
```

Так как раскрытие `setf` для `lookup` уже определено, мы можем определить метод `area` для `circle-class`:

```
(setf (lookup 'area circle-class)
      #'(lambda (c)
          (* pi (expt (rget 'radius c nil) 2))))
```

Здесь по-прежнему нет четкого разделения между классами и экземплярами, а метод – это просто функция в поле объекта. Но скоро мы запрячем это подальше от людских глаз

Рис. 17.10 завершает нашу новую реализацию. Этот код не добавляет в программу никакого функционала, но делает ее более простой в использовании. Макрос `defprop` по сути остался тем же, просто вызывает `lookup` вместо `gethash`. Как и ранее, он позволяет ссылаться на свойства в функциональном стиле:[1]

[1] Данный листинг содержит исправление авторской опечатки, которая была обнаружена Тимом Мензисом (Tim Menzies) и опубликована в новостной группе comp.lang.lisp (19.12.2010). Данная опечатка не внесена автором в официальный список опечаток книги. – *Прим. перев.*

```
(declaim (inline run-methods))

(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj nil)))
     (defun (setf ,name) (val obj)
       (setf (lookup ',name obj) val))))

(defun run-methods (obj name args)
  (let ((meth (rget name obj nil)))
    (if (not (eq meth :nil))
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))

(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (setf (lookup ',name ,gobj)
             (labels ((next () (rget ',name ,gobj t)))
               #'(lambda ,parms ,@body))))))
```

Рис. 17.10. Реализация с помощью векторов: интерфейс на макросах.

```
> (defprop radius)
(SETF RADIUS)
> (radius our-circle)
:NIL
> (setf (radius our-circle) 2)
2
```

Если необязательный второй аргумент `defprop` истинный, то его вызов раскрывается в `run-methods`, который сохранился практически без изменений.

Наконец, функция `defmeth` предоставляет удобный способ определения методов. В ее новой версии есть три новых момента: она неявно вызывает `defprop`, использует `lookup` вместо `gethash` и вызывает `rget` вместо `get-next` (см. стр. Ё046out) для получения следующего метода. Она похожа на `get-next` настолько, что мы можем объединить их в одну функцию добавлением еще одного аргумента. Если этот аргумент истинный, вместо `get-next` используется `rget`.

Теперь мы можем достичь того же эффекта, что и в предыдущем примере, но теперь это существенно проще:

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))
```

Обратите внимание, что вместо вызова `rget` мы можем сразу использовать `radius`, так как соответствующий вызов `defprop` уже был выполнен. Так как вызов `defprop` для `area` был уже сделан внутри `defmeth`, то нам доступна функция `area`, получающая площадь `our-circle`:

```
> (area our-circle)
12.566370614359173
```

17.8. Анализ

Только что мы встроили новый язык, приспособленный для написания реальных объектно-ориентированных приложений. Он прост, но для своего размера достаточно мощный. На большинстве приложений он будет еще и довольно быстрым, так как операции с экземплярами обычно выполняются намного чаще, чем с классами. Основной задачей переработки кода было именно повышение производительности при работе с экземплярами.

В нашей программе создание новых классов трудоемко и производит большое количество мусора. Но это не страшно, если классы не создаются в критических

участках приложения. В плане скорости доступа к экземплярам наша программа достигла предела, и дальнейшие усовершенствования могут быть связаны с оптимизацией.°Ё116in Кроме того, никакие операции с экземплярами не требуют выделения памяти. Кроме создания экземпляра, разумеется. Сейчас представляющие их векторы размещаются в памяти динамически. Это вполне естественно, но при желании можно избежать динамического выделения памяти, применив стратегию, описанную в разделе 13.4.

Наш встроенный язык является характерным примером программирования на Лиспе. Речь не только о встраивании языков, но и о пути, который мы проделали от ограниченной и неэффективной версии программы через полноценную, но неэффективную реализацию к эффективной, но слегка ограниченной.

Репутация Лиспа как медленного языка происходит не из его природы (еще с 1980-х годов компиляторы Лиспа составляют жесткую конкуренцию Си в плане быстроты кода), а из того, что многие программисты останавливаются здесь, на этом втором шаге. Как писал Ричард Гэбриел:

> «В Лиспе очень легко написать программу с плохой производительностью; в Си это практически невозможно.»°Ё117in

Да, это так, но это высказывание может пониматься как в пользу Лиспа, так и в обратную сторону:

1. Приобретая гибкость ценой скорости, вы облегчаете процесс написания программ в Лиспе; в Си вы не имеете такой возможности.
2. До тех пор, пока вы не оптимизируете свою реализацию, она будет обладать низкой эффективностью.

Как вы будете пользоваться этим преимуществом – решать вам, но у вас, по крайней мере, есть такая возможность.

Наш пример неплох, но он не предназначен для создания модели CLOS (за исключением, пожалуй, прояснения механизма работы `call-next-method`). Да и как можно говорить о сходстве слона CLOS и нашей 70-строчной Моськи. Отличий между ними больше, чем сходств. Мы убедились в широте понятия объектной ориентированности. Наша программа будет помощнее, чем некоторые другие реализации объектно-ориентированного программирования, но тем не менее составляет лишь небольшую часть CLOS.

Наша реализация отличается от CLOS тем, что каждый метод принадлежит определенному объекту. Такое понимание методов приравнивает их к функции, поведение которой управляется ее первым аргументом. Обобщенные функции CLOS могут управляться произвольным числом аргументов, а методами считаются компоненты обобщенной функции. Если они будут специфицированы только по первому аргументу, может показаться, что методы принадлежат определенным классам, но понимание CLOS в таком ключе некорректно, так как CLOS имеет множество других отличий.

Недостатками CLOS являются ее сложность и огромный размер, которые скрывают тот факт, что это всего лишь расширение Лиспа. Пример, приведенный в данной главе, по крайней мере проясняет этот момент. Для реализации модели передачи сообщений потребовалось немногим больше страницы кода, и это значит, что Лисп с легкостью поддерживает объектно-ориентированное программирование. А теперь более интересный вопрос: на что еще он способен?

A. Отладка

В этом приложении объясняется, как производить отладку в Лиспе, а также приводятся часто встречающиеся ошибки.

Отладчик

Если вы предлагаете Лиспу сделать то, чего он не может, вычисление будет прервано сообщением об ошибке, и вы попадете в отладчик. Поведение отладчика зависит от используемой реализации, но везде отображаются, как правило, следующие вещи: сообщение об ошибке, список опций и приглашение иного вида.

В отладчике вы по-прежнему можете вычислять выражения. Это позволяет выяснить причину ошибки и даже исправить ее на лету, после чего программа продолжит выполнение. Тем не менее, чаще всего единственное, что вы будете делать в отладчике – это выходить из него. Попадание в отладчик как правило происходит при несоответствии типов и других подобных оплошностях, когда источник ошибки понятен и пребывание в отладчике не даст ничего нового. В нашей гипотетической реализации необходимо ввести `:abort`, чтобы попасть в toplevel:

```
> (/ 1 0)
Error: Division by zero.
       Options: :abort, :backtrace
>> :abort
>
```

В вашей реализации, скорее всего, придется ввести нечто другое.

Если вы находитесь в отладчике, и происходит еще одна ошибка, то вы попадаете в другой отладчик.[1] В большинстве случаев в приглашении отладчика отображается его номер:

```
>> (/ 2 0)
Error: Division by zero.
       Options: :abort, :backtrace, :previous
>>>
```

Теперь глубина вложенности отладчиков равна двум и мы можем вернуться в предыдущий отладчик или сразу попасть в toplevel.

Трассировка и обратная трассировка

Если ваша программа делает не то, что вы ожидали, прежде всего необходимо выяснить, что же она делает. Напечатав `(trace foo)` вы предлагаете Лиспу выводить сообщение каждый раз, когда `foo` вызывается и возвращает значение. Вы можете проследить таким образом за любой функцией.

Следы функции обычно выравниваются по глубине вложенности вызова. Для функции, выполняющей обход дерева, например, добавляющей единицу к каждому истинному элементу:

```
(defun tree1+ (tr)
  (cond ((null tr) nil)
        ((atom tr) (1+ tr))
        (t (cons (tree1+ (car tr))
                 (tree1+ (cdr tr))))))
```

след будет отображать структуру дерева:Ё007in

```
> (trace tree1+)
(tree1+)
```

[1] Как раз поэтому отладчик в Лиспе называют break loop. – *Прим. перев.*

```
> (tree1+ '((1 . 3) 5 . 7))
1 Enter TREE1+ ((1 . 3) 5 . 7)
  2 Enter TREE1+ ( 1 . 3 )
    3 Enter TREE1+ 1
    3 Exit TREE1+ 2
    3 Enter TREE1+ 3
    3 Exit TREE1+ 4
  2 Exit TREE1+ (2 . 4)
  2 Enter TREE1+ (5 . 7)
    3 Enter TREE1+ 5
    3 Exit TREE1+ 6
    3 Enter TREE1+ 7
    3 Exit TREE1+ 8
  2 Exit TREE1+ (6 . 8)
1 Exit TREE1+ ((2 . 4) 6 . 8)
((2 . 4) 6 . 8)
```

Для отключения трассировки `foo` введите `(untrace foo)`; для отключения трассировки всех функций – просто `(untrace)`.

Более гибкой альтернативой трассирования является встраивание диагностической печати в код. Этот метод, ставший классическим, применяется обычно в десять раз чаще, чем изощренные отладочные утилиты. Это еще одна причина, по которой полезно интерактивное переопределение функций.

Обратный след (*backtrace*) является списком всех вызовов, находящихся на стеке в момент попадания в отладчик. Задача трассировки может быть выражена словами: «Что мы делали?», а обратная трассировка – словами: «Как мы это получили?». Эти операции дополняют друг друга. Трассировка покажет каждый вызов заданной функции в выбранной части кода (на пути от toplevel к месту возникновения ошибки).

В типичной реализации мы получим обратный след, введя `:backtrace` в отладчике:

```
> (tree1+ '((1 . 3) 5 . A))
Error: A is not a valid argument to 1+.
       Options: :abort, :backtrace
>> :backtrace
(1+ A)
(TREE1+ A)
(TREE1+ (5 . A))
(TREE1+ ((1 . 3) 5 . A))
```

С помощью обратной трассировки поймать ошибку иногда довольно легко. Для этого достаточно посмотреть на цепочку вызовов. Другим преимуществом функционального программирования (раздел 2.12) является то, что все баги проявятся при обратной трассировке. В полностью функциональном коде все, что могло повлиять на возникновение ошибки, при ее появлении находится на стеке.

Количество информации в обратном следе зависит от используемой реализации. Одна реализация может выводить историю всех незавершенных вызовов с их аргументами, другая может не выводить практически ничего. Имейте ввиду, что обычно трассировка (и обратная трассировка) более информативна для интерпретированных функций, нежели для скомпилированных.

Традиционно отладку ведут в режиме интерпретации, и лишь затем компилируют отлаженную версию. Однако взгляды на этот вопрос постепенно изменяются, тем более, что некоторые реализации вовсе не содержат интерпретатора.

Когда ничего не происходит

Не все ошибки приводят к прерыванию вычислений. Часто возникает ситуация, когда Лисп полностью игнорирует все ваши действия. Это может означать, что вы попали в бесконечный цикл. Если это происходит в процессе итерации, то Лисп со счастливой улыбкой будет выполнять ее бесконечно. Если же вы столкнулись с бесконечной рекурсией (которая не оптимизирована как хвостовая), то вы рано или поздно получите сообщение, информирующее об отсутствии свободного места на стеке:

```
> (defun blow-stack () (1+ (blow-stack)))
BLOW-STACK
> (blow-stack)
Error: Stack overflow.
```

Если же этого не происходит, то вам остается принудительно вызвать отладчик и выйти их него.

Может случиться, что программа, решающая трудоемкую задачу, исчерпывает место на стеке не по причине бесконечной рекурсии. Но это происходит довольно редко, и исчерпание стека как правило свидетельствует о программной ошибке.

При создании рекурсивных функций часто забывают базовое условие. Это происходит иногда и при словесном описании рекурсивного алгоритма. Например, мы можем сказать, что `obj` принадлежит `lst`, если он является первым элементом либо принадлежит к остатку `lst`. Строго говоря, это не так, ведь есть еще одно условие – `obj` не принадлежит `lst`, если `lst` пуст. Наше первое определение соответствует бесконечному циклу.

В Common Lisp `car` и `cdr` возвращают `nil`, если их аргументом является `nil`:

```
> (car nil)
NIL
> (cdr nil)
NIL
```

Так что, если мы пропустим базовое условие в определении `member`:

```
(defun our-member (obj lst)                    ; wrong
  (if (eql (car lst) obj)
      lst
      (our-member obj (cdr lst))))
```

то мы получим бесконечную рекурсию, когда искомый объект будет отсутствовать в списке. Когда мы достигнем конца списка, рекурсивный вызов будет иметь следующий вид:

```
(our-member obj nil)
```

В корректном определении (см. стр. Ё014out) базовое условие приведет к остановке рекурсии, возвращая `nil`. В нашем случае функция будет покорно искать `car` и `cdr` от `nil`, которые также равны `nil`, что приводит к бесконечному процессу.

Когда причина бесконечного процесса не лежит на поверхности, как в нашем случае, вам поможет трассировка. Бесконечные циклы бывают двух видов. Вам повезло, если источником бесконечного повторения является структура программы. В случае `our-member` трассировка тут же покажет, где проблема.

Более сложным случаем являются некорректно-сформированные структуры данных. Например, попытка обхода циклического списка (см. рис. Ё047out) приводит к бесконечному циклу. Подобные проблемы трудно обнаружить, так как они не проявляют себя раньше времени. Правильным решением является предупреждение подобных проблем (см. стр. Ё048out): избегайте деструктивных операторов до тех пор, пока не убедитесь в корректности работы программы.

Если Лисп игнорирует ваши действия, причиной тому может быть ожидание ввода. Во многих системах нажатие Enter не приводит к каким-либо эффектам, если ввод выражения не завершен. Это дает возможность введения многострочных выражений. Вместе с тем, если вы забудете ввести закрывающую скобку или кавычку, Лисп будет ожидать завершения ввода выражения, в то время как вы будете думать, что ввели его полностью:

```
> (format t "for example ~A~% 'this)
```

Мы пропустили закрывающую кавычку в конце шаблона форматирования. Нажатие на Enter не приведет к чему-либо, так как Лисп продолжает считывать строку.

Хотя в некоторых реализациях есть возможность редактирования предыдущей строки, универсальным решением является принудительный вызов отладчика с последующим выходом из него.

Переменные без значения/несвязанные

Частой причиной жалоб со стороны Лиспа является несвязанные символы и символы без значения. Такая проблема может возникать по ряду причин.

Локальные переменные, устанавливаемые в `let` или `defun`, действительны только внутри их тела. Попытка получить доступ к переменным откуда-либо извне приведет к ошибке:

```
> (progn
    (let ((x 10))
      (format t "Here x = ~A.~%" x))
    (format t "But now it's gone...~%")
    x)
Here x = 10.
But now it's gone...
Error: X has no value.
```

Когда Лисп жалуется подобным образом, это значит, что он пытается получить доступ к несуществующей переменной. Если в текущем окружении не существует локальной переменной `x`, Лисп ищет глобальную переменную или константу с тем же именем. Если он не находит никакого значения, то появляется ошибка. Если возникает подобная проблема, проверьте, нету ли где-то опечатки в имени переменной.

Похожая проблема случается, если мы пытаемся сослаться на функцию как на переменную:

```
> defun foo (x) (+ x 1))
Error: DEFUN has no value.
```

Казалось бы, как это `defun` не имеет значения? Мы забыли открывающую скобку, и Лисп воспринимает символ `defun` как глобальную переменную, которая при считывании должна быть связана со значением.

Иногда программисты забывают инициализировать глобальные переменные. Если вы не сообщаете второй аргумент `defvar`, глобальная переменная будет задекларирована, но не инициализирована. Будьте внимательны с `defvar`.

Неожиданные nil

Ё016inЕсли Лиспу не нравится, что функция получила `nil` в качестве аргумента, то дело, как правило, в одной из предыдущих функций. Ряд встроенных операторов возвращает `nil` при неудачном завершении. Но `nil` – полноценный объект, и проблема может обнаружить себя существенно позже, когда какая-либо функция получает некорректное для нее значение `nil`.

Например, представим, что наша функция, возвращающая количество дней в месяце, содержит баг: мы забыли про октябрь:

```
(defun month-length (mon)
  (case mon
    ((jan mar may jul aug dec) 31)
    ((apr jun sept nov) 30)
    (feb (if (leap-year) 29 28))))
```

Кроме того, мы имеем функцию для вычисления количества недель в месяце:

```
(defun month-weeks (mon) (/ (month-length mon) 7.0))
```

В таком случае при вызове `month-weeks` может произойти следующее:

```
> (month-weeks 'oct)
Error: NIL is not a valid argument to /.
```

Ни одно из заданных условий в `case`-выражении не было выполнено, и `case` вернул `nil`; `month-weeks` ожидает число, которое затем передается функции `/`, но получает `nil`, что и приводит к ошибке.

В нашем примере источник и место проявления бага находятся рядом. Чем дальше они находятся, тем сложнее для понимания. Для предотвращения подобных ситуаций в Common Lisp есть `ecase` (см. раздел 14.6).

Переименование

К неприятным последствиям может привести переименование функций или переменных в коде. Вы легко можете пропустить одно или несколько мест в коде, где встречается подлежащее переименованию имя. Например, представим следующую (неэффективную) реализацию функции поиска глубины вложенного списка:

```
(defun depth (x)
  (if (atom x)
      1
      (1+ (apply #'max (mapcar #'depth x)))))
```

При первом же тестировании мы обнаружим, что она дает значение, завышенное на 1:

```
> (depth '((a)))
3
```

Исходное значение должно быть `0`, а не `1`. Исправим этот недочет, а заодно дадим функции более конкретное имя:

```
(defun nesting-depth (x)
  (if (atom x)
      0
      (1+ (apply #'max (mapcar #'depth x)))))
```

Теперь снова протестируем:

```
> (nesting-depth '((a)))
3
```

Мы получили прежний результат! В чем же дело? Да, ошибку мы исправили, но этот результат получается не из нового кода. Приглядитесь внимательно, ведь мы забыли исправить имя в рекурсивном вызове, который вызывает по-прежнему некорректную функцию `depth`.

Ключевые слова как необязательные параметры

Если функция принимает одновременно ключевые и необязательные параметры, обычной ошибкой является передача ключевого параметра как необязательного по невнимательности. Например, функция `read-from-string` имеет следующий список параметров:

```
(read-from-string string &optional eof-error eof-value
                         &key start end preserve-whitespace)
```

Чтобы передать такой функции ключевые параметры, нужно сначала передать ей все необязательные. Если мы забудем об этом, как в данном примере:

```
> (read-from-string "abcd" :start 2)
ABCD
4
```

то `:start` и `2` будут расценены как необязательные параметры. Корректный вызов будет выглядеть следующим образом:

```
> (read-from-string "abcd" nil nil :start 2)
CD
4
```

Некорректные декларации

В главе 13 описывался механизм декларации переменных и структур данных. Декларируя что-либо, мы даем обещание в дальнейшем следовать этой декларации, и компилятор полностью доверяется нашему обязательству. Например, оба аргумента следующей функции обозначены как `double-float`:

```
(defun df* (a b)
  (declare (double-float a b))
  (* a b))
```

и компилятор сгенерировал код, предназначенный для умножения только чисел `double-float`.

Если `df*` вызывается с аргументами, не соответствующими задекларированному типу, то возвращается ошибка, либо просто производится немного мусора. Кроме того, это может привести также к более серьезной низкоуровневой ошибке.[1]

Предупреждения

Иногда Лисп может жаловаться на что-либо без прерывания вычислений. Подобные предупреждения сигнализируют о каком-либо негативном, но не критическом явлении. Например, предупреждением сопровождается компиляция функции, содержащей незадекларированные или неиспользуемые переменные. Например, во втором вызове `map-int` (см. стр. Ё40out) переменная `x` не используется. Чтобы заткнуть компилятор, пытающийся каждый раз сообщить об этом факте, используйте декларацию `ignore`:

```
(map-int #'(lambda (x)
             (declare (ignore x))
             (random 100))
         10)
```

[1] Если ошибка, вызванная несоответствием деклараций, происходит в коде, скомпилированном без проверок типов, т.е. `(safety 0)`, то это может привести к повреждению объектов в памяти и другим не менее неприятным последствиям. – *Прим. перев.*

B. Лисп на Лиспе

Это приложение содержит определения 58 наиболее используемых операторов Common Lisp. Так как все (или большая часть) операторов Лиспа пишутся на нем самом, и при этом не выглядят сложными, то их разбор очень полезен для понимания языка.

Кроме того, данное упражнение демонстрирует, что Common Lisp не такой уж и большой язык, как может показаться. Большинство операторов Common Lisp являются по сути библиотечными функциями. Набор операторов, необходимых для определения остальных, на деле совсем мал. В этом приложении мы определим следующие операторы:

```
apply aref backquote block car cdr ceiling char= cons defmacro
documentation eq error expt fdefinition function floor gensym
get-setf-expansion if imagpart labels length multiple-value-bind
nth-value quote realpart symbol-function tagbody type-of typep =
+ - / < >
```

Код написан таким образом, чтобы объяснять Common Lisp, но не реализовывать его. В полноценных реализациях много внимания уделяется производительности и проверке на наличие ошибок. Операторы определены в алфавитном порядке для упрощения их поиска. Чтобы эту реализацию было возможно использовать, определения макросов должны находиться раньше, чем вызовы соответствующих функций в коде.

```
(defun -abs (n)
  (if (typep n 'complex)
      (sqrt (+ (expt (realpart n) 2) (expt (imagpart n) 2)))
      (if (< n 0) (- n) n)))

(defun -adjoin (obj lst &rest args)
  (if (apply #'member obj lst args) lst (cons obj lst)))

(defmacro -and (&rest args)
  (cond ((null args) t)
        ((cdr args)  `(if ,(car args) (-and ,@(cdr args))))
        (t           (car args))))

(defun -append (&optional first &rest rest)
  (if (null rest)
      first
      (nconc (copy-list first) (apply #'-append rest))))

(defun -atom (x) (not (consp x)))

(defun -butlast (lst &optional (n 1))
  (nreverse (nthcdr n (reverse lst))))

(defun -cadr (x) (car (cdr x)))

(defmacro -case (arg &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,arg))
       (cond ,@(mapcar #'(lambda (cl)
                           (let ((k (car cl)))
                             `(,(cond ((member k '(t otherwise))
                                       t)
                                      ((consp k)
                                       `(member ,g ',k))
                                      (t `(eql ,g ',k)))
                               (progn ,@(cdr cl)))))
                       clauses)))))

(defun -cddr (x) (cdr (cdr x)))
```

```
(defun -complement (fn)
  #'(lambda (&rest args) (not (apply fn args))))

(defmacro -cond (&rest args)
  (if (null args)
      nil
      (let ((clause (car args)))
        (if (cdr clause)
            `(if ,(car clause)
                 (progn ,@(cdr clause))
                 (-cond ,@(cdr args)))
            `(or ,(car clause)
                 (-cond ,@(cdr args)))))))

(defun -consp (x) (typep x 'cons))

(defun -constantly (x) #'(lambda (&rest args) x))

(defun -copy-list (lst)
  (labels ((cl (x)
             (if (atom x)
                 x
                 (cons (car x)
                       (cl (cdr x))))))
    (cons (car lst)
          (cl (cdr lst)))))

(defun -copy-tree (tr)
  (if (atom tr)
      tr
      (cons (-copy-tree (car tr))
            (-copy-tree (cdr tr)))))

(defmacro -defun (name parms &rest body)
  (multiple-value-bind (dec doc bod) (analyze-body body)
    `(progn
       (setf (fdefinition ',name)
             #'(lambda ,parms
                 ,@dec
                 (block ,(if (atom name) name (second name))
                   ,@bod))
             (documentation ',name 'function)
             ,doc)
       ',name)))

(defun analyze-body (body &optional dec doc)
  (let ((expr (car body)))
    (cond ((and (consp expr) (eq (car expr) 'declare))
           (analyze-body (cdr body) (cons expr dec) doc))
          ((and (stringp expr) (not doc) (cdr body))
           (if dec
               (values dec expr (cdr body))
               (analyze-body (cdr body) dec expr)))
          (t (values dec doc body)))))

;;; Это определение не является полностью корректным; см let.

(defmacro -do (binds (test &rest result) &rest body)
  (let ((fn (gensym)))
    `(block nil
       (labels ((,fn ,(mapcar #'car binds)
                     (cond (,test ,@result)
                           (t (tagbody ,@body)
                              (,fn ,@(mapcar #'third binds))))))
         (,fn ,@(mapcar #'second binds))))))

(defmacro -dolist ((var lst &optional result) &rest body)
  (let ((g (gensym)))
```

```
    `(do ((,g ,lst (cdr ,g)))
         ((atom ,g) (let ((,var nil)) ,result))
       (let ((,var (car ,g)))
         ,@body))))

(defun -eql (x y)
  (typecase x
    (character (and (typep y 'character) (char= x y)))
    (number    (and (eq (type-of x) (type-of y))
                    (= x y)))
    (t         (eq x y))))

(defun -evenp (x)
  (typecase x
    (integer (= 0 (mod x 2)))
    (t       (error "non-integer argument"))))

(defun -funcall (fn &rest args) (apply fn args))

(defun -identity (x) x)

;;; Это определение не является полностью корректным: выражение
;;; (let ((&key 1) (&optional 2))) корректно, а вот его
раскрытие – нет.

(defmacro -let (parms &rest body)
  `((lambda ,(mapcar #'(lambda (x)
                         (if (atom x) x (car x)))
                     parms)
      ,@body)
    ,@(mapcar #'(lambda (x)
                  (if (atom x) nil (cadr x)))
              parms)))
(defun -list (&rest elts) (copy-list elts))

(defun -listp (x) (or (consp x) (null x)))

(defun -mapcan (fn &rest lsts)
  (apply #'nconc (apply #'mapcar fn lsts)))

(defun -mapcar (fn &rest lsts)
  (cond ((member nil lsts) nil)
        ((null (cdr lsts))
         (let ((lst (car lsts)))
           (cons (funcall fn (car lst))
                 (-mapcar fn (cdr lst)))))
        (t
         (cons (apply fn (-mapcar #'car lsts))
               (apply #'-mapcar fn
                      (-mapcar #'cdr lsts))))))

(defun -member (x lst &key test test-not key)
  (let ((fn (or test
                (if test-not
                    (complement test-not))
                #'eql)))
    (member-if #'(lambda (y)
                   (funcall fn x y))
               lst
               :key key)))

(defun -member-if (fn lst &key (key #'identity))
  (cond ((atom lst) nil)
        ((funcall fn (funcall key (car lst))) lst)
        (t (-member-if fn (cdr lst) :key key))))

(defun -mod (n m)
  (nth-value 1 (floor n m)))
```

```
(defun -nconc (&optional lst &rest rest)
  (if rest
      (let ((rest-conc (apply #'-nconc rest)))
        (if (consp lst)
            (progn (setf (cdr (last lst)) rest-conc)
                   lst)
            rest-conc))
      lst))

(defun -not (x) (eq x nil))
(defun -nreverse (seq)
  (labels ((nrl (lst)
             (let ((prev nil))
               (do ()
                   ((null lst) prev)
                 (psetf (cdr lst) prev
                        prev      lst
                        lst       (cdr lst)))))
           (nrv (vec)
             (let* ((len (length vec))
                    (ilimit (truncate (/ len 2))))
               (do ((i 0 (1+ i))
                    (j (1- len) (1- j)))
                   ((>= i ilimit) vec)
                 (rotatef (aref vec i) (aref vec j))))))
    (if (typep seq 'vector)
        (nrv seq)
        (nrl seq))))

(defun -null (x) (eq x nil))

(defmacro -or (&optional first &rest rest)
  (if (null rest)
      first
      (let ((g (gensym)))
        `(let ((,g ,first))
           (if ,g
               ,g
               (-or ,@rest))))))

;;; Не входит в Common Lisp, но требуется для
;;; нескольких других определений здесь

(defun pair (lst)
  (if (null lst)
      nil
      (cons (cons (car lst) (cadr lst))
            (pair (cddr lst)))))

(defun -pairlis (keys vals &optional alist)
  (unless (= (length keys) (length vals))
    (error "mismatched lengths"))
  (nconc (mapcar #'cons keys vals) alist))




(defmacro -pop (place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (let ((g (gensym)))
      `(let* (,@(mapcar #'list vars forms)
              (,g ,access)
              (,(car var) (cdr ,g)))
         (prog1 (car ,g)
                ,set)))))

(defmacro -prog1 (arg1 &rest args)
  (let ((g (gensym)))
```

```
    `(let ((,g ,arg1))
       ,@args
       ,g)))

(defmacro -prog2 (arg1 arg2 &rest args)
  (let ((g (gensym)))
    `(let ((,g (progn ,arg1 ,arg2)))
       ,@args
       ,g)))

(defmacro -progn (&rest args) `(let nil ,@args))

(defmacro -psetf (&rest args)
  (unless (evenp (length args))
    (error "odd number of arguments"))
  (let* ((pairs (pair args))
         (syms (mapcar #'(lambda (x) (gensym))
                       pairs)))
    `(let ,(mapcar #'list
                   syms
                   (mapcar #'cdr pairs))
       (setf ,@(mapcan #'list
                       (mapcar #'car pairs)
                       syms)))))

(defmacro -push (obj place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
              ,@(mapcar #'list vars forms)
              (,(car var) (cons ,g ,access)))
         ,set))))

(defun -rem (n m)
  (nth-value 1 (truncate n m)))

(defmacro -rotatef (&rest args)
  `(psetf ,@(mapcan #'list
                    args
                    (append (cdr args)
                            (list (car args))))))

(defun -second (x) (cadr x))

(defmacro -setf (&rest args)
  (if (null args)
      nil
      `(setf2 ,@args)))

(defmacro setf2 (place val &rest args)
  (multiple-value-bind (vars forms var set)
                       (get-setf-expansion place)
    `(progn
       (let* (,@(mapcar #'list vars forms)
              (,(car var) ,val))
         ,set)
       ,@(if args `((setf2 ,@args)) nil))))

(defun -signum (n)
  (if (zerop n) 0 (/ n (abs n))))

(defun -stringp (x) (typep x 'string))

(defun -tailp (x y)
  (or (eql x y)
      (and (consp y) (-tailp x (cdr y)))))

(defun -third (x) (car (cdr (cdr x))))
```

```
(defun -truncate (n &optional (d 1))
  (if (> n 0) (floor n d) (ceiling n d)))

(defmacro -typecase (arg &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,arg))
       (cond ,@(mapcar #'(lambda (cl)
                           `((typep ,g ',(car cl))
                             (progn ,@(cdr cl))))
                       clauses)))))
(defmacro -unless (arg &rest body)
  `(if (not ,arg)
       (progn ,@body)))

(defmacro -when (arg &rest body)
  `(if ,arg (progn ,@body)))

(defun -1+ (x) (+ x 1))

(defun -1- (x) (- x 1))

(defun ->= (first &rest rest)
  (or (null rest)
      (and (or (> first (car rest)) (= first (car rest)))
           (apply #'->= rest))))
```

C. Изменения в Common Lisp

ANSI Common Lisp отличается от Common Lisp, описанного в 1984 году в первом издании «Common Lisp: the Language» Гая Стила. Он также отличается и от описания языка во втором издании (1990). Это приложение описывает наиболее существенные различия. Изменение, внесенные с 1990 перечислены отдельно в последнем разделе.

Основные дополнения

1. Система объектов Common Lisp (CLOS) стала частью языка.
2. Макрос `loop` теперь реализуется в виде встроенного языка с инфиксным синтаксисом.
3. Common Lisp теперь включает набор новых операторов, имеющих общее название «система исключений», для сигнализации и обработки ошибок и прочих исключений.
4. Common Lisp теперь предоставляет явную поддержку и управление аккуратной печатью (pretty printing).

Частные дополнения

1. Добавлены следующие операторы:

```
complement                  nth-value
declaim                     print-unreadable-object
defpackage                  readtable-case
delete-package              row-major-aref
destructuring-bind          stream-external-format
fdefinition                 with-compilation-unit
file-string-length          with-hash-table-iterator
function-lambda-expression  with-package-iterator
load-time-value             with-standard-io-syntax
map-into
```

2. Добавлены следующие глобальные переменные:

```
*debugger-hook*    *read-eval*    *print-readably*
```

Функции

1. Смысл имени функции был обобщен на выражения вида `(setf f)`. Подобные выражения теперь принимаются любым оператором или декларацией, которые ожидают имени функции. Новая функция `fdefinition` подобна `symbol-function`, но понимает имя функции в более общем смысле.
2. Тип `function` более не включает `fboundp`-символы и лямбда-выражения. Символы (но не лямбда-выражения) по-прежнему могут использоваться в качестве имен функций. Лямбда-выражения могут быть преобразованы в функции с помощью `coerce`.
3. Символы, используются как имена ключевых параметров, более не обязаны быть ключевыми словами. (Символы не из пакета `keyword` должны быть закавычены при таком их использовании).
4. Для остаточных параметров (`&rest`) не гарантируется новое выделение памяти, изменять их деструктивно небезопасно.
5. Локальные функции, определенные через `flet` или `labels`, а также результаты раскрытия `defmacro`, `macrolet` или `defsetf`, неявно заключены в `block` с соответствующим именем.
6. Функция, имеющая интерпретируемое определение в ненулевом лексическом окружении (т.е. определенные из toplevel внутри `let`) не могут быть скомпилированы.

Макросы

1. Были введены макросы компиляции и символ-макросы, а также соответствующие операторы.
2. Теперь раскрытие макроса определяется в том окружении, где был совершен вызов `defmacro`. По этой причине в ANSI Common Lisp код раскрытия макроса может ссылаться на локальные переменные:

```
(let ((op 'car))
  (defmacro pseudocar (lst)
    `(,op ,lst)))
```

   В 1984 году раскрытие макроса определялось в нулевом окружении (т.е. в toplevel).
3. Гарантируется, что макровызовы не будут перераскрываться в скомпилированном коде.
4. Макровызовы теперь могут быть точечными списками.
5. Раскрытия макросов теперь могут не содержать `declare`.

Вычисление и компиляция

1. Переопределен специальный оператор `eval-then`, все его исходные ключевые параметры объявлены устаревшими.
2. Функция `compile-file` теперь принимает аргументы `:print` и `:verbose`.
3. Новые переменные `*compile-name-pathname*` и `*compile-file-truename*` связываются на время вызова `compile-file`, как и `*load-pathname*` и `*load-truename*` в процессе `load`.
4. Добавлена декларация `dynamic-extent`.
5. Добавлен параметр компиляции `debug`.
6. Добавлен специальный оператор `compiler-let`.
7. Удален макрос чтения `#,`.
8. Удалена глобальная переменная `*break-on-warnings*`, на замену ей введена более общая `*break-on-signals*`.

Побочные эффекты

1. Метод `setf` может выполнять несколько присвоений.
2. Изменены способы изменения аргументов многими деструктивными функциями. Например, большинство операторов, которые могут модифицировать списки, не только имеют разрешение на это, но и обязаны это делать. Теперь деструктивные функции могут применяться для получения не только значений, но и побочных эффектов.
3. Теперь явным образом запрещено использование отображающий функций (а также макросов типа `dolist`) для изменения последовательностей, проход по которым они выполняют.

Символы

1. Новая глобальная переменная `*gensym-counter*` содержит целое число, используемое для создания имен символов, создаваемых `gensym`. В 1984 г. счетчик мог быть неявно сброшен передачей целого числа аргументом `gensym`; теперь это действие признано устаревшим.
2. Модификация строки, используемой как имя символа, теперь считается ошибкой.
3. Функция `documentation` стала обобщенной функцией.

Списки

1. Функции `assoc-if`, `assoc-if-not`, `rassoc-if`, `rassoc-if-not` и `reduce` теперь принимают ключевой параметр `:key`.
2. Функция `last` теперь имеет необязательный параметр, задающий длину возвращаемого хвоста.

3. С добавлением `complement` функции `-if-not` и ключевой параметр `:test-not` признаны устаревшими.

Массивы

1. Добавлены новые функции, позволяющие программам запрашивать информацию об обновлении типов (type-upgrading) элементов массивов и комплексных чисел.
2. Индексы массивов теперь определены как `fixnum`.

Строки

1. Удален тип `string-char`. Тип `string` более не идентичен типу `(vector string-char)`. Тип `character` разделен на два новых подтипа: `base-char` и `extended-char`. Тип `string` получил новый подтип `base-string`, который в свою очередь имеет новый подтип `simple-base-string`.
2. Несколько функций для создания строк могут принимать параметр `:element-type`.
3. Упразднены атрибуты знаков `font` и `bits`, а также все связанные с ними функции и константы. Единственным атрибутом знака теперь является `code`.
4. Большинство строковых функций могут преобразовывать аргументы одинаковых типов в строки. `(string= 'x 'x)` теперь вернет `t`.

Структуры

1. Отныне не обязательно задавать какие-либо слоты в теле вызова `defstruct`.
2. Последствия переопределения структуры, т.е. двукратного вызова `defstruct` с одним именем, не определены.

Хеш-таблицы

1. Функция `equalp` применима к хеш-таблицам.
2. Добавлены новые функции доступа, позволяющие программам ссылаться на свойства хеш-таблиц: `hash-table-rehash-size`, `hash-table-rehash-threshold`, `hash-table-size` и `hash-table-test`.

I/O

1. Введена концепция логических путей, а также соответствующие операторы.
2. Введены несколько новых типов потоков, а также соответствующие предикаты и функции доступа.
3. Введены директивы форматирования `~`, `~W` и `~I`. Директивы `~D`, `~B`, `~O`, `~X` и `~R` принимают дополнительный аргумент.
4. Функции `write` и `write-to-string` принимают пять новых ключей.
5. Для ввода путей добавлен новый макрос чтения `#P`.
6. Новая переменная `*print-readably*` может использоваться для принудительного вывода в читаемом виде.

Числа

1. Теперь тип `fixnum` имеет размер не менее 16 бит.
2. Добавлены две новые константы, ограничивающие допустимые значения нормализованных чисел с плавающей запятой.
3. Добавлен тип `real`; он является подтипом для `number` и надтипом для `rational` и `float`.

Пакеты

1. Изменено соглашение о способе определения пакетов в файлах с исходными кодами. Вызовы функций, связанных с пакетами, из toplevel не обрабатываются компилятором. Вместо этого введен макрос `defpackage`. Старая функция `in-package` заменена на макрос с тем же именем. Новый макрос не вычисляет свой аргумент, принимает параметр `:use`, неявно создает пакеты.
2. Пакеты `lisp` и `user` переименованы в `common-lisp` и `common-lisp-user`. Вновь создаваемые пакеты не используют `common-lisp` по умолчанию, как это делалось с пакетом `lisp`.
3. Имена встроенных переменных, функций, макросов и специальных операторов принадлежат `common-lisp`. Попытка переопределения, связывания, трассировки и деклараций применительно к любым встроенным операторам считается ошибкой.

Типы

1. Добавлен спецификатор типов `eql`.
2. Удаляет тип `common` и функция `commonp`.

Изменения с 1990 года

1. Добавлены следующие операторы:

```
allocate-instance          ensure-directories-exist
array-displacement         lambda
constantly                 read-sequence
define-symbol-macro        write-sequence
```

2. Удалены следующие операторы и переменные:

```
applyhook                  function-information
*applyhook*                get-setf-method
augment-environment        generic-flet
declare                    generic-function
enclose                    generic-labels
evalhook                   parse-macro
*evalhook*                 variable-information
declaration-information    with-added-methods
define-declaration
```

   Убраны `get-setf-method` и `get-setf-method-multiple-value`, добавлен `get-setf-expansion`. Символ `declare` продолжает использоваться, но более не является оператором.

3. Следующие четыре оператора были переименованы:

```
define-setf-[method → expander]
get-setf-[method-multiple-value → expansion]
special-[form → operator]-p
simple-condition-format-[string → control]
```

   а также два типа:

```
base-[character → char]
extended-[character → char]
```

4. Использование модулей, убранное в 1990 г., было возобновлено, однако объявлено нежелательным.
5. Первым аргументом `setf` может быть выражение `values`.
6. Стандарт ANSI более конкретно определяет, какие функции могут принимать точечные списки. Например, сейчас определено, что аргументами `nconc` могут быть точечные списки. (Странно, что аргументами `append` должны быть нормальные списки, то есть `nconc` и `append` принимают не одинаковые аргументы.)
7. Теперь невозможно преобразовывать `integer` в `character` с помощью `coerce`. Добавлена возможность преобразования `(setf f)` в функцию с помощью `coerce`.
8. Ослаблено ограничение, согласно которому аргумент `compile` должен быть определен в нулевом лексическом окружении; теперь окружение может содержать определения и декларации локальных макросов или символ-макросов.
9. Функции `gentemp` и `set` признаны устаревшими.

10. Символ `type` в декларациях типов может быть опущен. Таким образом, ошибкой является определение типа с таким же именем, как и декларация, и наоборот.
11. Новая декларация `ignorable` может использоваться, чтобы избавиться от предупреждений, независимо от того, используется переменная или нет.
12. Константа `array-total-size-limit` теперь является `fixnum`. Поэтому аргумент `row-major-aref` всегда декларируется как `fixnum`.
13. Вместо `:print-function` структура, определяемая через `defstruct`, может иметь `:print-object` (с двумя аргументами).
14. Введен новый день `boolean`, которому принадлежат только `nil` и `t`.

Справочник по языку

Это приложение описывает каждый оператор в ANSI Common Lisp. Мы будем следовать ряду соглашений:

Синтаксис

Описания функций представлены списком, начинающимся с имени функции, за которым следуют описания ее параметров. Описания специальных операторов и макросов являются регулярными выражениями, описывающими внешний вид корректного вызова.

Опишем соглашения по формату регулярных выражений. Звездочка[1] позади объекта означает, что объект повторяется произвольное количество раз (в т.ч. ноль): `(a*)` может соответствовать `()`, `(a)`, `(a a)` и так далее. Объект в скобках соответствует его отсутствию или однократному повторению: `(a [b] c)` может быть `(a c)` или `(a b c)`. Фигурные скобки иногда используются для группировки: `({a b}*)` может соответствовать `()`, `(a b)`, `(a b a b)` и так далее. Вертикальная черта соответствует выбору между несколькими альтернативами: `(a {1 | 2} b)` может быть `(a 1 b)` или `(a 2 b)`.

Имена параметров

Имена параметров соответствуют ограничениям на аргументы. Если имя параметра соответствует определенному типу, то данный аргумент может быть только этого типа. Ряд других имен перечислен (с описаниями) в нижеследующей таблице.

Аргументы макросов и специальных операций не вычисляются, если в описании это не указано явно. Если такой аргумент не вычисляется, ограничения на тип задаются его именем применительно к самому аргументу, если вычисляются – то к его значению. Если аргумент макроса вычисляется, он вычисляется в том окружении, где встречается макровызов, если явно не сказано другое.

alist	должен быть ассоциативным списком, т.е. нормальным списком, состоящим из элементов вида *(key . value)*.
body	обозначает аргументы, которые могут следовать после списка параметров в выражении `defun`: либо *declaration** `[`*string*`]` *expression**, либо `[`*string*`]` *declaration** *expression**. Выражение вида `(defun` *fname parameters . body*`)` означает, что синтаксис выражения `defun` может быть `(defun` *fname parameters* `[`*string*`]` *expression**`)` или `(defun` *fname parameters* `[`*string*`]` *declarations** *expressions**`)`. Если после строки *string* есть хотя бы одно выражение, то строка считается документацией.
c	комплексное число.
declaration	список, `car` которого – `declare`.
environment	означает, что объект представляет собой лексическое окружение. (Создавать такие объекты вручную вам не позволено, Лисп использует такие параметры для своих целей.) Символ `nil` всегда представляет глобальное окружение.
f	число с плавающей точкой.
fname	имя функции либо список `(setf` *s*`)`, где *s* – символ.

[1] Надстрочный знак * в регулярных выражениях не следует путать с `*`.

format	строка, которая может быть вторым аргументом `format`, либо функция, которая принимает поток и ряд необязательных аргументов и (предположительно) записывает что-либо в этот поток.
I	целое число.
list	список любого липа. Может ли он быть циклическим, зависит от контекста. Функции, принимающие списки как аргументы, могут работать с cdr-циклическими списками тогда и только тогда, когда их задача никогда не потребует поиска конца списка. Так, `nth` может работать с cdr-циклическими списками, а `find` – нет. Параметр, названный *list*, всегда может быть car-циклическим.
n	неотрицательное целое число.
object	объект любого типа.
package	пакет; или строка, представляющая имя пакета; или символ, представляющий имя пакета.
path	путь к файлу; поток, связанный с файлом (в таком случае из него можно получить имя соответствующего файла); строка.
place	выражение, которое может быть первым аргументом `setf`.
plist	список свойств; или нормальный список с четным количеством элементов.
pprint-dispatch	таблица диспетчеризации аккуратной печати (может быть `nil`).
predicate	функция.
prolist	нормальный список.
proseq	нормальная последовательность, т.е. вектор или нормальный список.
r	действительное число (`real`).
tree	не предполагает ограничений по типу – все, что можно считать деревом; не может быть циклическим списком.
type	спецификатор типа.

Умолчания

Некоторые параметры по умолчанию имеют некоторые значения. Необязательный параметр *stream* всегда имеет умолчальное значение `*standard-input*` или `*standard-output*`, в зависимости от направления потока. Необязательный параметр *package* по умолчанию установлен в `*package*`; *readtable* в `*readtable*`, а *pprint-dispatch* в `*print-pprint-dispatch*`.

Сравнение

Многие функции, сравнивающие элементы последовательностей, имеют ключевые параметры `key`, `test`, `test-not`, `from-end`, `start` или `end`; их использование не зависит от чего-либо (см. стр. Ё049out). Параметры `key`, `test` и `test-not` должны быть функциями, `start` и `end` – неотрицательными целыми числами. Слова «match», «member» и «element» в описании таких функций должны пониматься с учетом подобных аргументов.

В любой функции, сравнивающей элементы последовательности, предикатом сравнения по умолчанию является `eql`.

Структура

Если оператор возвращает структуру (т.е. списки), необходимо помнить, что эта структура разделяется с аргументами этой функции, если явно не указано, что возвращаемый результат является вновь созданной структурой. В любом случае, лишь параметры, заключенные в угловые скобки (<*list*>) могут быть

модифицированы при вызове функции. Если две функции перечисляются одна за другой, это значит, что вторая является деструктивным аналогом первой.

Вычисление и компиляция

- `(compile` *fname* `&optional` *function*`)` функция

Если *function* не предоставляется и *fname* является именем нескомпилированной функции или макроса, `compile` заменяет эту функцию или макрос их скомпилированной версией, возвращая *fname*. (Лексическое окружение функции или макроса не будет отличаться от глобального окружения, за исключением определений локальных макросов и символ-макросов или деклараций.) Если предоставляется *function* (которая может быть функцией или лямбда-выражением), то `compile` преобразует ее к функции, компилирует и присваивает ей имя *fname*, возвращая его. *fname* может быть также `nil`, в таком случае скомпилированная функция не возвращается. Также возвращаются два дополнительных значения: второе является истинным, если при компиляции были получены ошибки или предупреждения; третье является истинным, если при компиляции были получены ошибки и предупреждения, исключая предупреждения о стиле (style warnings).

- `(compiler-macro-function` *fname* `&optional` *environment*`)` функция

Возвращает макрос компиляции, чьим именем в окружении *environment* является *fname* (или `nil`, если ничего не найдено). Макрос компиляции – это функция двух аргументов: сам вызов и окружение, в котором он произошел. Значение устанавливаемо[1].

- `(constantp` *expression* `&optional` *environment*`)` функция

Возвращает истину, если *expression* является именем константы; или списком, чьим `car` является `quote`; или объектом который не является ни символом, ни cons-ячейкой. Может также возвращать истину для других выражений, которые считаются константами в используемой реализации.

- `(declaim` *declaration-spec*`)` макрос

Аналогичен `proclaim`, но вызов в toplevel обрабатывается компилятором, а *declaration-spec* не вычисляется.

- `(declare` *declaration-spec*`*)`

Не оператор, но напоминает выражение, чьим `car` является `declare`. Может находиться в начале тела кода. Определяет декларации в соответствии с *declaration-spec* (не вычисляется) для последующего кода в том окружении, где находится декларация. Допустимы следующие декларации: `dynamic-extent`, `ftype`, `ignorable`, `ignore`, `inline`, `notinline`, `optimize`, `special`, `type`.

- `(define-compiler-macro` *fname parameters* `.` *body*`)` макрос

Похож на `defmacro`, но определяет макрос компиляции. Макросы компиляции похожи на обычные макросы, но раскрываются только компилятором и раньше, чем обычные макросы. Обычно *fname* – это имя уже существующей функции или макроса. Макрос компиляции предназначен для оптимизации конкретных случаев и не изменяет поведение в остальных случая. (Если возвращается нормальный макрос, это приведет к ошибке.) Строка документации доступна через `documentation` с именем *fname* и ключом `compiler-macro`.

- `(define-symbol-macro` *symbol expression*`)` макрос

symbol определяется как макровызов, пока не существует аналогичной специальной переменной с таким же именем. Его раскрытием будет *expression*. Если *symbol* встречается в вызове `setq`, то `setq` будет действовать, как `setf`; аналогично, для `multiple-value-setq` – как `setf` для `values`.

- `(defmacro` *symbol parameters* `.` *body*`)` макрос

[1] В оригинале используется слово settable. Оно означает, что вызов данной функции является «адресом» (или, иначе, «местом»), который может изменяться через `setf`, а также другими подобными макросами, например, `incf`, `decf`, `rotatef`. В данном справочнике фраза «значение устанавливаемо» будет использоваться в этом смысле. В неофициальных русскоязычных материалах используется также жаргонное слово setfable. – *Прим. перев.*

Глобально определяет макрос с именем *symbol*. В большинстве случаев выражение `(symbol` $a_1 \ldots a_n$`)` может пониматься как заменяемое на значение, возвращаемое `((lambda` *parameters* `.` *body*`)` $a_1 \ldots a_n$`)` перед его вычислением. В общем случае *parameters* связаны с аргументами макровызова так же, как с помощью `destructing-bind`; они могут содержать параметр `&whole`, который будет связан со всем макровызовом, и `&environment`, который будет связан с окружением, в котором встречен макровызов. Строка документации может быть получена для имени *symbol* с ключом `function`.

- `(eval` *expression*`)` функция

Вычисляет выражение в глобальном окружении и возвращает его значение(я).

- `(eval-when (`*case*`*) (`*expression*`*)` специальный оператор

Если применимо хотя бы одно условие из *case*, выражения *expressions* вычисляются по-порядку; возвращается значение последнего или `nil`. Условие `compile-toplevel` применимо, когда `eval-when` находится на верхнем уровне файла при его компиляции. Условие `:load-toplevel` применимо, когда выражение `eval-when` попадает в toplevel при загрузке файла. Условие `:execute` применяется, когда `eval-when` вычисляется любым доступным способом (использование только условия `:execute` делает вызов `eval-then` эквивалентным `progn`). Символы `compile`, `load` и `eval` являются устаревшими синонимами `:compile-toplevel`, `:load-toplevel` и `:execute`.

- `(lambda` *parameters* `.` *body*`)` макрос

Эквивалент `(function (lambda` *parameters* `.` *body*`))`

- `(load-time-value` *expression* `&optional` *constant*`)` специальный оператор

Эквивалент `(quote val)`, где `val` – значение, возвращенное выражением при загрузке скомпилированного файла, содержащего выражение `load-time-value`. Если *constant* (не вычисляется) равна `t`, сигнализируется, что это значение не будет изменено.

- `(locally` *declaration*`*` *expression*`*)` специальный оператор

Выражение вида `(locally` $e_1 \ldots e_n$`)` эквивалентно `(let ()` $e_1 \ldots e_n$`)`. Выражения *expression* последовательно вычисляются в локальном окружении, где заданы декларации *declaration*.

- `(macroexpand` *expression* `&optional` *environment*`)` функция

Возвращает раскрытие макровызова *expression*, получаемое последовательным применением `macroexpand-1` до тех пор, пока результат не будет макровызовом. Второе возвращаемое значение истинно, когда возвращаемое значение отлично от *expression*.

- `(macroexpand-1` *expression* `&optional` *environment*`)` функция

Если *expression* является макровызовом, выполняет один этап его раскрытия, в противном случае возвращает *expression*. Вызывает `*macroexpand-hook*`, исходно содержащую вызов функции трех аргументов: соответствующую макрофункцию, *expression* и *environment*. Раскрытие производится передачей аргументов, полученных macroexpand-1, в вызов макрофункции. Второе возвращаемое значение истинно, когда возвращаемое значение отлично от *expression*.

- (`macro-function` *symbol* `&optional` *environment*`)` функция

Возвращает макрофункцию с именем *fname* в окружении *environment* или `nil` в случае отсутствия. Макрофункция – это функция двух аргументов: самого вызова и окружения. Может быть аргументом `setf`.

- `(proclaim` *declaration-spec*`)` функция

Выполняет глобальные декларации согласно *declaration-spec*. Допустимы следующие декларации: `declaration`, `ftype`. `inline`, `notinline`, `optimize`, `special`. `type`.

- `(special-operator-p symbol)` функция

Возвращает истину, когда *symbol* является именем специального оператора.

- `(symbol-macrolet ((`*symbol expression*`)*) `*declaration*`* `*expression*`*)` специальный оператор

Вычисляет свое тело, связав каждый символ локально с символ-макросом таким же образом, как и через `define-symbol-macro`.

- `(the `*type expression*`)` специальный оператор

Возвращает значение выражения *expression* и декларирует его принадлежность к типу *type*. Может работать с выражениями, возвращающими несколько значений. (Спецификаторы типов с `values` описаны на стр. Ё050out.) Количество деклараций и значение могут не совпадать. Лишние декларации должны быть `t` ли `nil`, для лишних значений типы не декларируются.

- `(quote `*object*`)` специальный оператор

Возвращает свой аргумент без его вычисления.

Типы и классы

- `(coerce `*object type*`)` функция

Возвращает эквивалентный объект типа *type*. Если объект уже принадлежит типу *type*, возвращается он сам. В противном случае, если объект является последовательностью, и последовательность типа *type* может содержать те же элементы, что и *object*, то возвращается последовательность типа *type* с теми же объектами. Если объект – это строка из одного знака или символ с именем, строка которого состоит из одного знака, и задан тип `character`, то возвращается данный знак. Если объект является действительным числом (`real`), и тип *type* соответствует числу с плавающей запятой, результатом будет аппроксимация *object* в виде числа с плавающей запятой. Если объект – это имя функции (символ, `(setf `*f*`)` или лямбда-выражение), и задан тип `function`, возвращается функция, на которую указывает объект; в случае лямбда-выражения функция будет определена в глобальном окружении, а не в окружении вызова `coerce`.

- `(deftype `*name parameters* `. `*body*`)` макрос

Похож на `defmacro`, но «вызов» *name* используется в качестве указателя типа, а не выражения. Строка документации становится документацией к типу, доступной по ключу `type` для имени *name*.

- `(subtypep `*type1 type2* `&optional `*environment*`)` функция

Возвращает два значения: первое истинно тогда, когда может быть доказано, что *type1* является подтипом *type2*, второе – когда взаимоотношения между двумя типами известны точно.

- `(type-error-datum `*condition*`)` функция

Возвращает объект, вызвавший исключение *condition* по типу.

- `(type-error-expected-type `*condition*`)` функция

Возвращает тип, к которому мог принадлежать объект, вызвавший исключение по типу.

- `(type-of `*object*`)` функция

Возвращает спецификатор типа, к которому принадлежит объект *object*.

- `(typep `*object type* `&optional `*environment*`)` функция

Возвращает истину, если объект *object* принадлежит типу *type*.

Управление и потоки данных

- `(and `*expression*`*)` макрос

Вычисляет выражения одно за другим, если значение одного из них будет `nil`, то дальнейшее вычисление прерывается и возвращается `nil`. Если все выражения истинны, возвращается значение последнего. Если выражения не заданы, возвращает `t`.

- `(apply `*function* `&rest `*args*`)` функция

Применяет функцию *function* к аргументам *args* (которых должно быть не менее одного). Последний аргумент должен быть списком. Аргументами вызова *function* являются все элементы *args*, кроме последнего, и все элементы последнего элемента *args*, то есть, список аргументов составляется так, как с помощью `list*`. *function* может быть также символом, в таком случае используется связанная с ним глобальная функция.

- `(block` *symbol expression*`*)` специальный оператор

Вычисляет тело, заключенное в блок с именем *symbol* (не вычисляется). Используется вместе с `return-from`.

- `(case` *object* `(`*key expression*`*)* `[({t | otherwize}` *expression*`*)]` `)` специальный оператор

Вычисляет объект, затем сверяет его значение с последующими выражениями следующим образом: если значение равно (`eql`) ключу *key* (не вычисляется) или принадлежит (`member`) ему, то вычисляются соответствующие ключу выражения, и возвращается значение последнего из них. Ключи `t` и `otherwize` соответствуют любому значению. Если совпадение не найдено, возвращается `nil`. Символы `t` и `otherwize` могут не использоваться в качестве ключей, но тот же эффект можно получить с помощью `(t)` и `(otherwize)`.

- `(catch` *tag expression*`*)` специальный оператор

Вычисляет свое тело, ожидая получения тега, чье имя является значением *tag*. Используется совместно с `throw`.

- `(ccase` *object* `(`*key expression*`*)*`)` макрос

Действует аналогично `case`, за исключением случая, когда совпадение не найдено. Возвращает `nil` в том случае, когда с совпавшим ключом не связано ни одного выражения. Если совпадение не найдено, возвращает исправимую ошибку типа.

- `(compiled-function-p` *object*`)` функция

Возвращает истину, когда *object* является скомпилированной функцией.

- `(complement` *predicate*`)` функция

Возвращает функцию одного аргумента, возвращающую истину тогда, когда заданный предикат (также принимающий один аргумент) возвращает ложь, и наоборот.

- `(cond (`*test expression*`*)*`)`[1] макрос

Вычисляет выражения *test* до тех пор, пока одно из них не окажется истинным. Если с ним не связано каких-либо выражений, возвращается значение его самого, в противном случае связанные с ним выражения вычисляются по очереди, и возвращается значение последнего. Если ни одно тестовое выражение не оказалось истинным, возвращается `nil`.

- `(constantly` *object*`)` функция

Возвращает функцию, принимающую произвольное количество аргументов, и возвращает *object*.

- `(ctypecase` *object* `(`*type expression*`*)*`)` макрос

Вычисляет объект, затем по очереди сверяет его значение с типами (не вычисляются). Если найдено совпадение, вычисляются соответствующие выражения и возвращается значение последнего из них. В случае отсутствия связанных выражений возвращается `nil`. В случае отсутствия совпадения вызывается исправимая ошибка типа.

- `(defconstant` *symbol expression* `[`*string*`])` макрос

Определяет символ как глобальную константу со значением выражения *expression*. Никакие глобальные или локальные переменные не могут иметь это же значение. Выражение *expression* может вычисляться на этапе компиляции. Строка *string*, если задана, становится строкой документации, доступной по ключу `variable` для имени *symbol*. Возвращает *symbol*.

- `(define-modify-macro` *name parameters symbol* `[`*string*`])` макрос

[1] Исправлена авторская опечатка, обнаруженная Адамом Якобсом. – *Прим. перев.*

Выражение вида `(define-modify-macro m (p1 ... pn) f)` определяет новый макрос *m*, такой, что вызов вида `(m place a1 ... an)` установит значение по адресу *place* в `(f val a1 ... an)`, где *val* представляет значение *place*. Среди параметров могут быть также необязательные и остаточные. Строка *string*, если задана, становится документацией к новому макросу.

- `(define-setf-expander` *reader parameters* `.` *body*`)` макрос

Определяет порядок раскрытия выражений вида `(setf (reader a1 ... an) val)`. При вызове `get-setf-expander` для такого выражения будут возвращены пять значений. Строка *string*, если задана, становится документацией для *reader* с ключом `setf`.

- `(defparameter` *symbol expression* `[`*string*`])` макрос

Создает глобальную переменную *symbol* со значением выражения *expression*. Строка, если задана, становится документацией для *symbol* с ключом `variable`. Возвращает *symbol*.

- `(defsetf` *reader writer* `[`*string*`])` макрос

Сокращенный вид: преобразует вызовы вида `(setf (reader a1 ... an) val)` в `(writer a1 ... an val)`; *reader* и *writer* должны быть символами и не вычисляются. Строка, если задана, становится документацией к *reader* с ключом `setf`.

- `(defsetf` *reader parameters* `(`*var*`*)` `.` *body*`)` макрос

Расширенный вид: преобразует вызовы вида `(setf (reader a1 ... an) val)` в выражения, получаемые вычислением тела `defsetf` так же, как и для `defmacro`. *reader* должен быть символом (не вычисляется), соответствующим имени функции или макроса, который вычисляет все свои аргументы; *parameters* – это список параметров для *reader*, а *vars* будут представлять значение(я) *val*. Строка *string*, если задана, становится документацией к *reader* с ключом `setf`.

Чтобы соответствовать принципу, по которому `setf` возвращает новое значение его первого аргумента, раскрытие `defsetf` также должно возвращать это значение.

- `(defun fname parameters . body)` макрос

Глобально определяет *fname* как функцию, определенную в лексическом окружении, в котором находится тело `defun`. Тело заключено в блок с именем *fname*, если *fname* – символ, или *f*, если *fname* – список вида `(setf f)`. Строка *string*, если задана, становится документацией для *fname* с ключом `function`.

- `(defvar` *symbol* `[`*expression* `[`*string*`]])` макрос

дает глобальной переменной *symbol* значение *expression*, если выражение *expression* предоставлено, и эта переменная до настоящего момента не имела значения. Строка, если задана, становится документацией к *symbol* с ключом `variable`. Возвращает *symbol*.

- `(destructing-bind` *variables tree declaration*`*` *expression*`*)` макрос

Вычисляет значение выражений для набора переменных *variables* (дерево с внутренними узлами, являющимися списками переменных), связанных со значениями – соответствующими элементами дерева `t`. Форма двух деревьев должна совпадать.

- `(ecase` *object* `(`*key expression*`*)` `*)` макрос

Подобно `ccase`, но вызывает некорректируемую ошибку при отсутствии совпадения.

- `(eq` *object1 object2*`)` функция

Возвращает истину, когда *object1* и *object2* идентичны.

- `(eql` *object1 object2*`)` функция

Возвращает истину, когда *object1* и *object2* равны с точки зрения `eq`, являются одним и тем же знаком или числами, выглядящими одинаково при печати.

- `(equal` *object1 object2*`)` функция

Возвращает истину, когда *object1* и *object2* равны (`eql`); или являются cons-ячейками, чьи `car` и `cdr` эквивалентны с точки зрения `equal`, либо это строки и бит-векторы одной длины (с учетом указателей заполнения), чьи элементы эквивалентны с точки зрения `eql`; либо являются путями, чьи компоненты

являются эквивалентными. Вызов может не завершаться для циклических аргументов.

- `(equalp` *object1 object2*`)` функция

Возвращает истину, когда *object1* и *object2* равны как `equal`, `char-equal`, `=`; или являются cons-ячейками, `car` и `cdr` которых эквивалентны как `equalp`; или являются массивами одинаковой длины, элементы которых эквивалентны как `equalp`; либо являются структурами одного типа, чьи элементы равны как `equalp`; либо являются хеш-таблицами с одинаковыми тестовыми функциями и количеством элементов, ключи которых связаны со значениями, равными для двух таблиц как `equalp`. Логично предполагать, что вызов с циклическими аргументами может не завершиться.

- `(etypecase` *object* `(`*key exression*`*)*`)` макрос

Как `ctypecase`, но вызывает некорректируемую ошибку типа, если не находит совпадения с одним из ключей.

- `(every` *predicate proseq* `&rest` *proseqs*`)` функция

Возвращает истину, когда предикат *predicate*, который должен быть функцией стольких аргументов, сколько последовательностей передано, истинен для всех первых элементов последовательностей, затем для всех вторых элементов, и так до `n`-ного элемента, где `n` – дина кратчайшей последовательности. Проход по последовательностям завершается, когда предикат возвратит `nil`. В этом случае в результате всего вызова возвращается `nil`.

- `(fboundp` *fname*`)` функция

Возвращает истину, когда *fname* является именем глобальной функции или макроса.

- `(fdefinition` *fname*`)` функция

Возвращает глобальную функцию с именем *fname*. Может быть первым аргументом `setf`.

- `(flet ((`*fname parameters . body*`)`*`) declaration*`*` *expression*`*`)` специальный оператор

Вычисляет свое тело, связав локально каждое имя *fname* с соответствующей функцией. Действует подобно `labels`, но локальные функции видны лишь внутри тела и не могут вызывать сами себя и друг друга (то есть, не могут быть рекурсивными).

- `(fmakunbound` *fname*`)` функция

Удаляет определение глобальной переменной или макроса для *fname*. Если определения не найдено, вызывает ошибку. Возвращает *fname*.

- `(funcall` *function* `&rest` *args*`)` функция

Применяет функцию к аргументам. Функция может задана как символ, в таком случае используется определение глобальной функции, связанной с этим символом.

- `(function` *name*`)` специальный оператор

Возвращает функцию с именем *name*, которое может быть символом, списком вида `(setf` *f*`)` или лямбда-выражением. Если *f* – встроенный оператор, наличие или отсутствие функции `(setf` *f*`)` зависит от используемой реализации.

- `(function-lambda-expression` *function*`)` функция

Предназначается для получения лямбда-выражения, соответствующего функции, но может также возвращать `nil`. Возвращает два дополнительных значения: второе, будучи `nil`, говорит, что функция была определена в нулевом лексическом окружении; третье значение может использоваться вашей реализацией, чтобы передать имя функции.

- `(functionp` *object*`)` функция

Возвращает истину, когда объект является функцией.

- `(labels ((`*fname parameters . body*`)`*`) declaration*`*` *expression*`*`)` специальный оператор

Вычисляет свое тело, предварительно связав локально имена *fname* с соответствующими определениями функций. Действует подобно `flet`, но имена

локальных функции видны внутри самих определений. Такие функции могут вызывать друг друга и самих себя (то есть, быть рекурсивными).

- `(get-setf-expansion` *place* `&optional` *environment*`)` функция

Возвращает пять значений (*v1* ... *v5*), определяющих вид раскрытия выражений `(setf` *place val*`)` в окружении *environment*. Эти пять значений могут быть: списком уникальных имен переменных (`gensym`); списком неодинаковых количеств значений, связанных с ними; списком переменных, хранящих значение(я) по адресу *place*; выражением, выполняющим установление нового значение, которое может ссылаться на переменные из *v1* и *v3*; выражением, получающим оригинальное значение *place*, которое может ссылаться на переменные из *v1*.

- `(go` *tag*`)` специальный оператор

Находясь внутри выражения `tagbody`, передает управление по адресу ближайшего тега из лексической области видимости, равному заданному как `eql`.

- `(identity` *object*`)` функция

Возвращает объект *object*.

`(if` *test then* `[`*else*`])` специальный оператор

Вычисляет значение *test*. Если оно истинно, вычисляет и возвращает значение *then*; в противном случае вычисляет и возвращает значение *else*, или возвращает `nil` в случае отсутствия *else*.

- `(let ({`*symbol* `|` `(`*symbol* `[`*value*`])}`*`)` *declaration** *expression**`)` специальный оператор

Вычисляет свое тело, предварительно связав каждый символ с соответствующие значением, или с `nil` в случае отсутствия значения.

- `(let* ({`*symbol* `|` `(`*symbol* `[`*value*`])}`*`)` *declaration** *expression**`)` специальный оператор

Действует подобно `let`, но выражения *value* могут ссылаться на предыдущие символы *symbol*.

- `(macrolet ((`*symbol parameters* `.` *body*`)`*`)` *declaration** *expression**`)` специальный оператор

Вычисляет свое тело, предварительно связав каждый символ локально с соответствующим макросом. Функции раскрытия определяются в том локальном окружении, где находится вызов `macrolet`. Подобно `flet`, локальные макросы не могут вызывать друг друга.

- `(multiple-value-bind (`*symbol**`)` *expression1 declaration** *expression**`)` макрос

Вычисляет expression1, затем его тело, предварительно связав каждый символ с соответствующим значением expression1. Если значений меньше, чем символов, остальные получают значения nil; если символов меньше, чем значений, то лишние значения игнорируются.

- `(multiple-value-call` *function expression**`)` специальный оператор

Вызывает функцию *function* (вычисляется) с аргументами, которыми являются все значения всех выражений *expression*.

- `(multiple-value-list` *expression*`)` макрос

Возвращает список значений, возвращаемых выражением *expression*.

- `(multiple-value-prog1` *expression1 expression**`)` специальный оператор

Вычисляет свои аргументы друг за другом, возвращая значение(я) первого.

- `(multiple-value-setq (`*symbol**`)` *expression*`)` макрос

Связывает символы (не вычисляются) со значениями, возвращаемыми выражением *expression*. Если значений слишком мало, оставшиеся символы получают `nil`; если значений слишком много, то лишние игнорируются.

- `(not` *object*`)` функция

Возвращает истину, когда объект *object* имеет значение `nil`.

- `(notany` *predicate proseq* `&rest` *proseqs*`)` функция

Выражение вида `(notany` *predicate* $s_1 \ldots s_n$`)` эквивалентно `(not (some` *predicate* $s_1 \ldots s_n$`))`.

- `(notevery` *predicate proseq* `&rest` *proseqs*`)` функция

Выражение вида `(notevery` *predicate* $s_1 \ldots s_n$`)` эквивалентно `(not (every` *predicate* $s_1 \ldots s_n$`))`.

- `(nth-value n` *expression*`)` макрос

Возвращает `n`-ное (`n` вычисляется) значение выражения *expression*. Нумерация начинается с `0`. В случае, когда выражение вернуло меньше `n+1` значений, возвращается `nil`.

- `(or` *expression**`)` макрос

Вычисляет выражения друг за другом до тех пор, пока одно из значений не окажется истинным. В таком случае возвращается само значение, в противном случае – `nil`. Может возвращать множественные значения (справедливо только для последнего выражения).

- `(prog ({`*symbol* `| (`*symbol* `[`*value*`])}`*`)` *declaration** `{`*tag* `|` *expression*`}`*`)` макрос

Вычисляет свое тело, предварительно связав каждый символ со значением соответствующего выражения *value* (или с `nil`, если ничего не задано). Тело заключено в `tagbody` и блок с именем `nil`, так что выражения *expression* могут использовать `go`, `return` и `return-from`.

- `(prog* ({`*symbol* `| (`*symbol* `[`*value*`])}`*`)` *declaration** `{`*tag* `|` *expression*`}`*`)` макрос

Действует подобно `prog`, но выражения *value* могут использовать ранее связанные символы.

- `(prog1` *expression expression**`)` макрос

Вычисляет свои аргументы один за другим, возвращая значение первого.

- `(prog2` *expression1 expression2 expression**`)` макрос

Вычисляет свои аргументы один за другим, возвращая значение второго.

- `(progn` *expression**`)` специальный оператор

Вычисляет свои аргументы один за другим, возвращая значение(я) последнего.

- `(progv` *symbols values expression**`)` специальный оператор

Вычисляет свое тело, предварительно связав динамически каждый символ в списке *symbols* с соответствующим элементом в списке *values*. Если переменных слишком много, попытка сослаться на оставшиеся вызовет ошибку; если их слишком мало, то лишние значения будут проигнорированы.

- `(psetf {`*place value*`}`*`)` макрос

Действует подобно `setf`, но если выражение *value* ссылается на один из предшествующих адресов *place*, то получит предыдущее значение. Это значит, что `(psetf` *x y y x*`)` обменяет значения *x* и *y*.

- `(psetq {`*symbol value*`}`*`)` макрос

Действует подобно `psetf`, но оперирует с символами, а не с адресами.

- `(return` *expression*`)` макрос

Эквивалент `return-from nil` *expression*). Многие макросы (в т.ч. `do`) неявно заключают свое тело в блок с именем `nil`.

- `(return-from` *symbol expression*`)` специальный оператор

Возвращает значение(я) выражения *expression* из ближайшего лексически блока с именем *symbol* (не вычисляется). Попытка использования `return-from` не внутри блока с заданным именем приводит к ошибке.

- `(rotatef` *place**`)` макрос

Смещает значения слева на одно, как в циклическом буфере. Все аргументы вычисляются по очереди; затем для вызова вида `(rotatef` $a_1 \ldots a_n$`)` адрес a_2

получает значение по адресу a_1, a_3 - значение a_2, и т.д., a_1 получает значение a_n. Возвращает `nil`.

- `(setf {`*place value*`}*)` макрос

Обобщение `setq`: помещает значение выражения *value* по заданному адресу. Если одно из выражений ссылается на ранее определенное значение по некоторому адресу, оно будет использовать новое значение по этому адресу. Возвращает значение последнего выражения *value*.

Валидным выражением, представляющим адрес, может быть: переменная; вызов `apply` с первым аргументом из числа следующих: `#'aref`, `#'bit`, `#'sbit`; вызов функции доступа, определенной через `defstruct`; выражение *values*, аргументы которого являются валидными адресами; вызов функции, для которой определено setf-раскрытие; макрос, раскрывающийся в один из вышеописанных случаев.

- `(setq {`*symbol value*`}*)` специальный оператор

Дает каждой переменной *symbol* значение соответствующего выражения *value*. Если одно из выражений ссылается на ранее определенную переменную, оно будет использовать новое значение. Возвращает значение последнего выражения *value*.

- `(shiftf` *place1 place* expression*`)` макрос

Смещает значения своих аргументов на одно влево. Аргументы вычисляются один за другим; затем для вызова вида `(shiftf` $a_1 \ldots a_n$ *val*`)` значение a_2 помещается по адресу a_1 и т.д., а по адресу a_n помещается значение *val*. Возвращается значение a_1.

- `(some predicate` *proseq* `&rest` *proseqs*`)` функция

Возвращает истину, если предикат *predicate*, который должен быть функцией стольких аргументов, сколько задано последовательностей, возвращает истину, будучи примененным к первым элементам всех последовательностей, или ко вторым, или к `n`-ным, где `n` – длина кратчайшей последовательности. Проход по последовательностям останавливается тогда, когда предикат вернет истину, при этом возвращается значение предиката на тот момент.

- `(tagbody {`*tag* `|` *expression**`})` специальный оператор

Вычисляет выражения *expression* одно за другим и возвращает `nil`. Может содержать вызовы `go`, изменяющие порядок вычисления выражений. Теги, которые должны быть символами или целыми числами, не вычисляются. Атомы, полученные в результате раскрытия макросов, тегами не считаются. Все макросы, имена которых начинаются с `do`, а также `prog` и `prog*`, неявно заключают свое тело в вызов `tagbody`.

- `(throw` *tag expression*`)` специальный оператор

Возвращает значение(я) выражения *expression* из ближайшего в динамической области видимости выражения `catch` с тегом, равным (`eq`) заданному тегу.

- `(typecase` *object* `(`*type expression**`)*` `[({t | otherwize}` *expression**`)])` макрос

Вычисляет объект *object*, затем по очереди сверяет его значение с перечисленными типами (не вычисляются). Если найдено совпадение, или достигнут вариант `t` или `otherwize`, вычисляются соответствующие выражения и возвращается значение последнего из них. Возвращает `nil`, если совпадений не найдено, или для совпавшего типа не определено никаких выражений.

- `(unless` *test expression**`)` макрос

Выражение вида `(unless` *test e1 ... en*`)` эквивалентно `(when (not` *test*`)` $e_1 \ldots e_n$ `)`.

- `(unwind-protect` *expression1 expression**`)` специальный оператор

Вычисляет свои аргументы друг за другом и возвращает значение первого. Остальные выражения будут вычислены, даже если вычисление первого было прервано.

- `(values &rest` *objects*`)` функция

Возвращает свои аргументы.

- `(values-list` *prolist*`)` функция

Возвращает элементы списка *prolist*.

- `(when` *test expression*`*)` макрос

Вычисляет выражение *test*. Если оно истинно, вычисляет выражения *expression* одно за другим, возвращая значение(я) последнего. В противном случае возвращает `nil`. Возвращает `nil`, если не задано не одного выражения *expression*.

Итерации

- `(do ({`*var* `|` `(`*var* `[`*init* `[`*update*`]])}`*`)` `(`*test result*`*)` *declaration*`*` `{`*tag* `|` *expression*`}`*`)` макрос

Вычисляет свое тело сначала для исходных значений переменных (если исходное значение переменной не задано, используется `nil`), затем для новых значений переменных, полученных из выражений *update*, до тех пор, пока не будет выполнен критерий останова *test*. Если значение *test* ложно, вычисляется тело цикла, если истинно – цикл завершается вычислением выражений *result*, значение последнего из которых возвращается. Тело цикла неявно заключено в `tagbody`, а выражение `do` целиком – в блок с именем `nil`.

Если выражение *init* ссылается на переменную *var*, будет использоваться ее текущее значение в контексте, где находится выражение. Если на переменную *var* ссылается выражение *update*, будет использоваться значение на предыдущей итерации. Значения устанавливаются так же, как через `let`, а изменяются так же, как через `psetq`.

- `(do*` `({`*var* `|` `(`*var* `[`*init* `[`*update*`]])}`*`)` `(`*test result*`*)` *declaration*`*` `{`*tag* `|` *expression*`*}`) макрос

Действует подобно `do`, но значения переменных устанавливаются последовательно, как через `let*`, а изменяются так же, как через `setq`.

- `(dolist (`*var list* `[`*result*`])` *declaration*`*` `{`*tag* `|` *expression*`}`*`)` макрос

Вычисляет свое тело, связывая переменную *var* последовательно с каждым элементом списка `list`. Если предоставлен пустой список, тело не будет вычислено ни разу. Тело неявно заключено в `tagbody`, вызов `dolist` целиком – в блок с именем `nil`. Возвращает значение(я) выражения *result*, или `nil`, если это выражение отсутствует. Если выражение *result* ссылается на переменную *var*, будет использоваться значение `nil`.

- `(dotimes (`*var integer* `[`*result*`])` *declaration*`*` `{`*tag* `|` *expression*`}`*`)` макрос

Вычисляет свое тело, связывая переменную *var* последовательно с целыми числами от `0` до значения `(-` *integer* `1)` включительно. Если значение *integer* не является положительным, тело не будет вычислено ни разу. Тело неявно заключено в `tagbody`, вызов `dotimes` целиком – в блок с именем `nil`. Возвращает значение(я) выражения *result*, или `nil`, если это выражение отсутствует. Если выражение *result* ссылается на переменную *var*, оно будет равно количеству вычислений тела цикла.

- `(loop` *expression*`*)` макрос

Сокращенная форма. Если выражения *expression* не включают ключевых слов макроса `loop`, то они просто вычисляются друг за другом бесконечное число раз. Выражения неявно заключены в блок с именем `nil`.

- `(loop [`*name-clause*`]` *var-clause*`*` *body-clause*`*)` макрос[Ё058in]

Развернутая форма: `loop`-выражение содержит ключевые слова, которые обрабатываются следующим образом:

1. Исходно все создаваемые переменные связаны с неопределенными, вероятно, случайными значениями корректного типа.
2. Вычисляется пролог.
3. Переменные получают исходные значения.
4. Выполняются проверки на выход из цикла.
5. Если проверка не пройдена, вычисляется тело цикла, затем значения переменных обновляются, и управление передается на предыдущий шаг.
6. Если один из тестов на завершение цикла оказался успешен, вычисляется эпилог, и возвращаются заданные значения.

Отдельно взятое предложение может давать вклад в несколько различных шагов. На каждом шаге предложения вычисляются в том порядке, в котором они встречаются в коде.

Выражение `loop` всегда неявно заключено в блок. По умолчанию имя `nil`, но может быть задано с помощью предложения `named`.

Значением всего `loop`-выражения является последнее накопленное значение или `nil`, если такового не имеется. Накопленным значением можно считать значение, полученное последовательно через предложения `collect`, `append`, `nconc`, `count`, `sum`, `maximize`, `minimize`, `always`, `never` или `thereis`.

Предложение *name-clause* задает имя блока с помощью `named`; *var-clause* может содержать предложения `with`, `initially`, `finally` или `for`; *body-clause* может быть предложением любого типа, за исключением `named`, `with` или `for`. Каждое из этих предложений описывается ниже.

Соглашения: если об этом не говорится явно, элементы предложений не вычисляются. Слово `type` означает декларацию типа: это может быть отдельный указатель типа или их дерево, распознаваемое по шаблону.

Синонимы: макрос `loop` щедро делится синонимами одних и тех же ключевых слов. В следующих парах ключей второе является полноценной заменой первому: `upfrom`, `from`; `downfrom`, `from`; `upto`, `to`; `downto`, `to`; `the`, `each`; `of`, `in`; `when`, `if`; `hash-keys`, `hash-key`; `hash-value`, `hash-values`; `symbol`, `symbols`; `present-symbol`, `present-symbols`; `external-symbol`, `external-symbols`; `do`, `doing`; `collect`, `collecting`; `append`, `appending`; `nconc`, `nconcing`; `count`, `counting`; `sum`, `summing`; `maximize`, `maximizing`; `minimize`, `minimizing`.

`named` *symbol*

Делает заданный символ именем блока, в который заключается loop-выражение.

`with` *var1* `[`*type1*`]` `=` *expression1* `{and` *var* `[`*type*`]` `=` *expression*`}`*

Связывает каждую переменную со значением соответствующего выражения, делая это параллельно, как `let`.

`initially` *expression1* *expression*`*`

Вычисляет выражения как часть `loop`-пролога.

`finally` *expression1* *expression*`*`

Вычисляет выражения как часть `loop`-эпилога.

`for` *var* `[`*type1*`]` *for-rest1* `{and` *var* `[`*type*`]` *for-rest*`}`*

Связывает каждую переменную со значением, получаемым из *for-rest* на каждой последующей итерации. Использование нескольких for-предложений, связанных через `and`, приведет к параллельному обновлению значений переменных, как в `do`. Возможные формы для *for-rest* выглядит следующим образом:

`[upfrom` *start*`]` `[{upto | below} end]` `[by` *step*`]`

Как минимум одно из этих подвыражений должно быть задействовано. Если используется несколько, то они могут следовать в произвольном порядке. Значение переменной исходно установлено в `start` или `0`. На каждой последующей итерации оно увеличивается на `step` или `1`. `upto` или `below` позволяют ограничить количество итераций: `upto` остановит итерации, когда значение переменной будет больше `end`, `below` – когда больше или равно. Если синонимы `from` и `to` используются совместно, или `from` используется без `to`, то `from` будет интерпретирован как `upfrom`.

`downfrom` *start* `[{downto | above} end]` `[by` *step*`]`

Аналогично, подпредложения могут следовать в произвольном порядке. Значение переменной исходно установлено в `start`. На каждой последующей итерации оно уменьшается на `step` или `1`. Выражения `downto` или `above` добавляют ограничение итераций: `downto` остановит итерации, когда значение переменной будет меньше `end`, `above` – когда меньше или равно.

`{in | on}` *list* `[by` *function*`]`

Если первое слово `in`, переменная получает по очереди все элементы списка `list`, если `on` – последовательно все хвосты. Если предоставлена функция, она применяется к списку вместо `cdr` на каждой итерации. Также добавляет тест на завершение: итерация прекращаются при достижении конца списка.

`=` *expression1* `[then` *expression2*`]`

Переменная исходно получает значение *expression1*. На каждой последующей итерации оно будет получать значение *expression2*, если задано `then`, иначе *expression1*.

`across` *vector*

Переменная связывается по очереди со всеми элементами вектора. Также добавляет тест на завершение: итерация завершается после достижения последнего элемента.

`being the hash-keys of` *hash-table* `[using (hash-value` *v2*`)]`

`being the hash-values of` *hash-table* `[using (hash-key` *v2*`)]`

В первом случае переменная будет связываться с одним из ключей хеш-таблицы (порядок не определен), а *v2* (необязательный) соответствующее ключу значение. Во втором случае переменная получит значение, а *v2* – соответствующий ему ключ. Добавляет тест на завершение: итерации прекратятся после достижения последнего ключа в хеш-таблице.

`being each {symbol | present-symbol | external-symbol} [of` *package*`]`

Переменная будет получать все достижимые символы, имеющиеся символы или символы внешние символы для пакета. Аргумент *package* используется так же, как в `find-package`. Если пакет не задан, используется текущий. Добавляет тест на завершение: итерация завершится после достижения последнего символа.

`do` *expression1* *expression**

Вычисляет выражения одно за другим.

`return` *expression*

Заставляет `loop`-выражение немедленно прекратить вычисления и возвратить значение *expression*. `loop`-эпилог не вычисляется.

`{collect | append | nconc}` *expression* `[into` *var*`]`

Собирает список, исходно `nil`, в процессе итерации: `collect` собирает список значений выражения *expression*, `append` объединяет получающиеся списки в один, а `nconc` делает то же самое деструктивно. Если предоставляется переменная *var*, то ей будет присвоен полученный список, но тогда он не будет возвращаться как результат `loop`-вызова по умолчанию.

Внутри `loop collect`, `append` и `nconc` могут аккумулировать значения в одну и ту же переменную. Если для нескольких предложений не предоставлены отдельные переменные, это будет расценено как желание собирать все значения в один список.

`{count | sum | maximize | minimize}` *expression* `[into` *var*`]` `[`*type*`]`

Аккумулирует числа. Если используется ключ `count`, возвращаемое значение будет отражать количество истинных результатов вычисления выражения *expression*; если `sum`, будет возвращена сумма всех полученных значений; если `maximize`/`minimize`, то максимальное/минимальное значение из всех полученных. Для `count` и `sum` исходным значением будет `0`, для `maximize` и `minimize` исходное значение не определено. Если предоставлена *var*, она будет связана с полученным значением, и оно не будет возвращено как значение `loop`-выражения по умолчанию. Также может быть задан тип *type* аккумулируемого значения.

Внутри `loop sum` и `count` могут аккумулировать значения в одну и ту же переменную. Если для нескольких предложений не предоставлены отдельные

переменные, это будет расценено как желание аккумулировать все значения в одно. То же для `maximize` и `minimize`.

`when` *test then-clause1* `{and` *then-clause*`}`*

`[else` *else-clause1* `{and` *else-cause*`}`*`] [end]`

Вычисляет выражение *test*. Если оно истинно, вычисляются выражения *test-clause* друг за другом. Выражения *then-clause* и *else-clause* могут использовать `do`, `return`, `when`, `unless`, `collect`, `append`, `nconc`, `count`, `sum`, `maximize` и `minimize`.

Выражение в *then-clause1* и *else-clause1* может использовать `it`, в таком случае оно будет ссылаться на значение выражения *test*.

`unless` *test then-clause1* `{and` *then-clause*`}`*

`[else` *else-clause1* `{and` *else-clause*`}`*`] [end]`

Предложение вида `unless` *test* $e_1 \ldots e_n$ эквивалентно `when (not` *test*`)` $e_1 \ldots e_n$.

`repeat` *integer*

Добавляет тест на завершение: итерации останавливаются после *integer* итераций.

`while` *expression*

Добавляет тест на завершение: итерации останавливаются, когда *expression* будет ложным.

`until` *expression*

Эквивалент `while (not` *expression*`)`.

`always` *expression*

Действует подобно `while`, но также предоставляет значение, возвращаемое `loop`: `nil`, если *expression* ложно, `t` в противном случае.

`never` *expression*

Эквивалент `always (not` *expression*`)`.

`thereis` *expression*

Действует подобно `until`, но также предоставляет значение, возвращаемое `loop`: `t`, если *expression* ложно, `nil` в противном случае.

- `(loop-finish)` макрос

Может использоваться только внутри `loop`-выражения, где он завершает текущую итерацию и передает управление в `loop`-эпилог, после чего `loop`-выражение завершается корректно.

Объекты

- `(add-method` *generic-function method*`)` обобщенная функция

Создает метод *method* для обобщенной функции *generic-function*, возвращая *generic-function*. При совпадении спецификаторов перезаписывает существующий метод. Добавляемый метод может принадлежать другой обобщенной функции.

- `(allocate-instance` *class* `&rest` *initargs* `&key)` обобщенная функция

Возвращает экземпляр класса *class* с неинициализированными слотами. Позволяет использовать любые ключи.

- `(call-method` *method* `&optional` *next-methods*`)` макрос

Если вызывается внутри метода, включает метод *method*, возвращая значения, которые вернет этот вызов. *Next-methods*, если предоставляется, должен быть списком методов; они будут вызваны следующими. Вместо метода может так же предоставляться список вида `(make-method` *expression*`)`, где *expression* – тело метода. В таком случае *expression* вычисляется в глобальном окружении, за исключением случая локального определения макроса `call-method`.

- `(call-next-method &rest` *args*`)` функция

Если вызывается внутри метода, включает следующий метод с аргументами *args*, возвращая значения его вызова. Если не предоставлено никаких аргументов, используются аргументы вызова текущего метода (игнорируя любые присвоения, проведенные с данными параметрами). В стандартной комбинации методов `call-next-method` может быть вызван в первичном или around-методе. Если следующий метод отсутствует, вызов `call-next-method` по умолчанию вызовет ошибку.

- `(change-class instance` *class* `&rest` *initargs* `&key)` обобщенная функция

Изменяет класс экземпляра *instance* на класс *class*, возвращая *instance*. Существующий слот, имеющий то же имя, что и локальный слот класса *class*, останется нетронутым, в противном случае он не используется. Новые локальные слоты, требуемые классом *class*, инициализируются вызовом `update-instance-for-redefined-class`. Допускает использование любых ключевых параметров.

- `(class-name` *class*`)` обобщенная функция

Возвращает имя класса. Может быть первым аргументом `setf`.

- `(class-of` *object*`)` функция

Возвращает класс, экземпляром которого является *object*.

- `(compute-applicable-methods` *generic-function args*`)` обобщенная функция

Возвращает сортированный по убыванию специфичности список методов для обобщенной функции *generic-function*, вызванной с аргументами из списка *args*.

- `(defclass` *name* `(`*superclass*`*)` `(`*slot-spec*`*)` *class-spec*`*)` макрос

Определяет и возвращает новый класс с именем *name*. Если *name* соответствует уже существующему классу, существующие экземпляры будут приведены к новому классу. Суперклассы задаются как имена *superclass*, но не обязательно должны присутствовать. Слотами нового класса является комбинация наследуемых слотов и локальных слотов, определенных в *slot-specs*. За объяснением механизма разрешения конфликтов обращайтесь на стр. Ё037out. Каждое определение слота должно быть символом либо списком `(symbol` k_1 v_1^* ... k_n v_n^*`)`, где каждый ключ *k* не может использоваться дважды. Символ является именем слота. Ключи могут быть:

`:reader` *fname*`*`

Для каждого *fname* определяет метод с именем *fname*, возвращающий значение соответствующего слота.

`:writer` *fname*`*`

Для каждого *fname* определяет метод с именем *fname*, ссылающийся на соответствующий слот. Используется только вместе с `setf`, но не самостоятельно.

`:accessor` *fname*`*`

Для каждого *fname* определяет метод с именем *fname*, ссылающийся на соответствующий слот. Может использоваться вместе с `setf`.

`:allocation` *where*

Если *where* – это `:instance` (по умолчанию), каждый экземпляр имеет свой собственный слот, если `:class`, то слот будет использоваться всеми экземплярами класса.

`:initform` *expression*

Если при создании экземпляра не передаются исходные значения слота, будет установлено значение *expression*.

`:initarg` *symbol*`*`

Каждый символ *symbol* может использоваться как ключевой параметр для определения значения слота при создании экземпляра через `make-instance`. Символы не обязательно должны быть ключевыми словами. Один и тот же символ может быть использован как `initarg` в нескольких слотах класса.

`:type` *type*

Декларирует тип значения, содержащегося в слоте.

`:documentation` *string*

Предоставляет строку документации к слоту.

Определения *class-spec* могут быть любой комбинацией выражений: `(:documentation` *string*`)`, `(:metaclass` *symbol*`)` и `(:default-initargs` k_1 e_1 ... k_n e_n`)`. Последнее используется при создании экземпляров; см. `make-instance`. (В любом вызове `make-instance` при отсутствии явного задания значения ключа *k* будет использоваться значение соответствующего выражения *e*.) Параметр `:documentation` становится документацией к классу, а `:metaclass` может использоваться для задания нового метакласса, отличного от `standard-class`.

- `(defgeneric` *fname parameters entry**`)` макрос

Определяет или дополняет текущее определение обобщенной функции *fname*. Возвращает саму функцию. Вызывает ошибку, если *fname* является обычной функцией или макросом.

Parameters – это специализированный лямбда-список; см. `defmethod`. Все методы для обобщенной функции должны иметь списки параметров, соответствующие друг другу и списку *parameters*.

Entry может быть одним из:

`(:argument-precedence-order` *parameter**`)`

Перезаписывает порядок предшествования, подразумеваемый вторым аргументом `defgeneric`. Должны быть заданы все те же параметры, что и в соответствующей обобщенной функции.

`(declare (optimize` *property**`))`

Декларация учитывается в процессе компиляции самой обобщенной функции, то есть, ее управляющего кода. Не влияет на компиляцию самих методов. См. `declare`.

`(:documentation` *string*`)`

Предоставляет документацию к имени *fname* с ключом `function`.

`(:method-combination` *symbol arguments**`)`

Определяет тип комбинации методов, используемый для имени *symbol*. Встроенные типы комбинации не требуют каких-либо аргументов, собственные типы комбинации могут определяться с помощью развернутой формы `define-method-combination`.

`(:generic-function-class` *symbol*`)`

Определяет принадлежность обобщенной функции к классу *symbol*. Может использоваться для изменения класса существующей обобщенной функции. По умолчанию используется `standard-generic-function`.

`(:method-class` *symbol*`)`

Определяет, что все методы обобщенной функции должны быть одного класса *symbol*. Могут переопределяться классы существующих методов. По умолчанию используется `standard-method`.

`(:method` *qualifier** *parameters . body*`)`

Эквивалент `(defmethod` *fname qualifier** *parameters . body*`)`. *Entry* может включать более одного выражения такого типа.

- `(define-method-combination` *symbol property**`)` макрос

Сокращенная форма: определяет новый тип комбинации методов. Краткая форма используется для прямой операторной комбинации методов. Если c_1 ... c_n – список вызовов применимых методов (от более специфичного к менее специфичному) для комбинации методов с оператором *symbol*, то вызов обобщенной функции будет эквивалентным `(symbol` c_1 ... c_n`)`.*Property* может быть:

`:documentation` *string*

Создает строку документации для *symbol* с ключом `method-combination`.

`:identity-with-one-argument` *bool*

Делает возможной оптимизацию вызовов обобщенной функции, для которой существует только один применимый метод. Если *bool* истинен, значения, возвращаемые методом, возвращаются как значения обобщенной функции. Используется, например, в `and`- и `progn`-комбинации методов.

`:operator` *opname*

Определяет актуальный оператор (может отличаться от *symbol*), используемый обобщенной функцией; *opname* может быть символом или лямбда-выражением.

- `(define-method-combination` *symbol parameters* макрос

  `(`*group-spec*`*`)`
  `[(:arguments . `*parameters2*`)]`
  `[(:generic-function `*var*`)]`
  `.` *body*`)`

Полная форма: определяет новую форму комбинации методов путем определения раскрытия вызова обобщенной функции. Вызов обобщенной функции, использующей symbol-комбинацию, будет эквивалентен выражению, возвращаемому телом *body*; при вычислении этого выражения единственной локальной связью будет макро определение для `call-method`.

Формы, предшествующие *body*, связывают переменные, которые могут использоваться для генерации раскрытия: *parameters* получает список параметров, следующих за *symbol* в аргументе `:method-combination` в `defgeneric`; parameters2 (если задан) получает форму, которая появляется в вызове обобщенной функции; оставшиеся необязательные параметры получают соответствующие initform-значения; кроме того, допустимо использование параметра `&whole`, который получает список всех аргументов формы; *var* (если задан) будет связан с самим объектом обобщенной функции.

Выражения *group-spec* могут использоваться для связывания переменных с непересекающимися списками применимых методов. Каждый *group-spec* может быть формой вида `(var {`*pattern*`* | `*predname*`} `*option*`*`)`. Каждая переменная *var* будет связана со списком методов, чьи квалификаторы совпадают с шаблоном *pattern* или удовлетворяют предикату с именем *predname*. (Если не задан не один предикат, то должен быть представлен по крайней мере один шаблон.) Шаблоном может быть `*`, что соответствует любому списку квалификаторов, или список символов, который совпадает (`equal`) со списком квалификаторов. (Этот список может также содержать `*` как элемент или как хвост списка.) Метод для заданного списка квалификаторов будет аккумулироваться в первой переменной *var*, чей предикат возвратит истину для данного списка, или один из шаблонов обеспечит совпадение. Доступны следующие опции:

`:description` *format*

Некоторые утилиты используют `format` как второй аргумент в вызове `format`, третьим аргументом которого является список квалификаторов метода.

`:order` *order*

Если используется значение `:most-specific-first` (по умолчанию), методы аккумулируются по убыванию специфичности; если используется `:most-specific-last`, то в обратном порядке.

`:required` *bool*

Если значение *bool* истинно, в случае отсутствия аккумулированных методов будет возвращаться ошибка.

- `(defmethod` *fname qualifier*`*` *parameters . body*`)` макрос

Определяет метод обобщенной функции *fname*, которая создается, если не существует. Возвращает новый метод. Вызывает ошибку, если *fname* является именем нормальной функции или макроса.

Квалификаторы *qualifier* – это атомы, используемые при комбинации методов. Стандартная комбинация методов допускает квалификаторы `:before`, `:after` и `:around`.

Параметры *parameter* подобны аналогичным в нормальных функциях, за исключением того, что обязательные параметры являются списками вида `(name specialization)`, где *specialization* – класс, либо имя класса, либо список вида `(eql expression)`. Последний вариант указывает, что параметр в момент раскрытия выражения `defmethod` должен быть равен (`eql`) *expression*. Методы определяются уникальным образом по квалификаторам и специализации, если два метода идентичны, то новый заменит старый. Список *parameters* должен соответствовать параметрам других методов данной обобщенной функции, а также самой обобщенной функции.

Применение метода эквивалентно вызову `(lambda parms . body)`, где *parms* – *parameters* без специализации, с аргументами обобщенной функции. Как и для `defun`, тело неявно заключается в блок с именем *fname*, если *fname* – символ, или *f*, если *fname* – список вида `(setf f)`.

- `(ensure-generic-function` *fname* `&key` *argument-precedence-order* функция
  declare documentation
  environment generic-function-class
  lambda-list method-class
  method-combination`)`

Делает *fname* (который не может быть именем нормальной функции ли макроса) именем обобщенной функции с заданными свойствами. Если обобщенная функция с таким именем уже существует, ее свойства заменяются, вероятно, после преодоления ряда ограничений. Свойства *argument-precedence-order*, *declare*, *documentation* и *method-combination* перезаписываются всегда; *lambda-list* должен соответствовать спискам параметров существующих методов; *generic-function-class* должно быть совместимо со старым значением, в этом случае для замены значения вызывается `change-class`. Когда изменяется `method-class`, существующие методы остаются без изменения.

- `(find-class` *symbol* `&optional` *error environment*`)` функция

Возвращает класс, чьим именем в *environment* является *symbol*. Если такой класс отсутствует, в случае, когда *error* истина (по умолчанию), вызывается ошибка, в противном случае возвращается `nil`. Может быть первым аргументом `setf`. Чтобы отвязать имя от класса, необходимо установить `find-class` в `nil`.

- `(find-method` *generic-function qualifiers specializers* `&optional` *error*`)` обобщенная функция

Возвращает метод обобщенной функции, чьи квалификаторы совпадают с *qualifiers*, а специализация – с *specializers*; *specializers* – это список классов (не имен); класс `t` соответствует отсутствию специализации. Если метод не найден, в случае истинности *error* сигнализируется ошибка, в противном возвращается `nil`.

- `(function-keywords` *method*`)` обобщенная функция

Возвращает два значения: список ключевых параметров, применимых к методу; второе значение истинно, когда метод допускает использование других ключей.

- `(initialize-instance` *instance* `&rest` *initargs* `&key)` обобщенная функция

Встроенный первичный метод, вызывающий `shared-instance` для установки слотов *instance* согласно заданным *initargs*. Используется в `make-instance`. Позволяет использование других ключей.

- `(make-instance` *class* `&rest` *initargs* `&key)` обобщенная функция

Возвращает новый экземпляр класса. Вместо *initargs* должны быть пары символ-значение: k_1 v_1 ... k_n v_n. Каждый слот в новом экземпляре будет инициализироваться следующим образом: если параметр *k* является параметром слота, то значением слота будет соответствующий ему *v*; иначе, если класс или один из его суперклассов имеет умолчальные `initarg`-параметры, в том числе для данного слота, то будет использоваться умолчание для наиболее специфичного класса; иначе, если слот имеет `initform`, слот получает ее значение; иначе слот остается несвязанным. Допускает любые ключи.

- `(make-instance-obsolete` *class*`)` обобщенная функция

Вызывается из `defclass` при попытке изменения определения класса. Обновляет все экземпляры класса (вызывая `update-instance-for-redefined-class`) и возвращает *class*.

- `(make-load-form` *object* `&optional` *environment*`)` обобщенная функция

Если объект является экземпляром, структурой, исключением или классом, возвращает одно или два выражения, которые при вычислении в *environment* приводят к значению, эквивалентному *object* в момент загрузки.

- `(make-load-form-saving-slots` *instance* `&key` *slot-names environment*`)` функция

Возвращает два выражения, которые при вычислении в *environment* приведут к значению, эквивалентному *instance* в момент загрузки. Если даны *slot-names*, будут сохранены лишь перечисленные слоты.

- `(method-qualifiers` *method*`)` обобщенная функция

Возвращает список квалификаторов метода.

- `(next-method-p)` функция

Вызванная внутри метода, возвращает истину в случае наличия доступного следующего метода.

- `(no-applicable-method` *generic-function* `&rest` *args*`)` обобщенная функция

Вызывается, когда для обобщенной функции с заданными аргументами нет ни одного применимого метода. Встроенный первичный метод вызывает ошибку.

- `(no-next-method` *generic-function method* `&rest` *args*`)` обобщенная функция

Вызывается, когда для метода и обобщенной функции не доступно ни одного следующего метода. Встроенный первичный метод вызывает ошибку.

- (`reinitialize-instance` *instance* `&rest` *initargs*`)` обобщенная функция

Устанавливает значения слотов экземпляра в соответствии с *initargs*. Встроенный первичный метод передает аргументы в `shared-instance` (с вторым аргументом `nil`). Допускает другие ключи.

- `(remove-method <`*generic-function*`>` *method*) обобщенная функция

Деструктивно удаляет метод из обобщенной функции, возвращая *generic-function*.

- `(shared-initialize` *instance names* `&rest` *initargs* `&key)` обобщенная функция

Устанавливает слоты экземпляра в соответствии с *initargs*. Все оставшиеся слоты инициализируются согласно их `initform`, если их имена перечислены в *names*, или *names* равен `t`. Допускает другие ключи.

- `(slot-boundp` *instance symbol*`)` функция

Возвращает истину, когда слот с именем *symbol* в заданном экземпляре инициализирован. Если такого слота нет, вызывает *slot-missing*.

- `(slot-exists-p` *object symbol*`)` функция

Возвращает истину, когда объект имеет слот с именем *symbol*.

- `(slot-makunbound <`*instance*`>` *symbol*`)` функция

Отвязывает слот с именем *symbol* в заданном экземпляре.

- `(slot-missing` *class object symbol opname* `&optional` *value*`)` обобщенная функция

Вызывается, когда оператор с именем *opname* не смог найти слот с именем *symbol* в объекте *object* класса *class*. (Если представлено значение *value*, то оно устанавливается на место несуществующего.) Встроенный первичный метод вызывает ошибку.

- `(slot-unbound` *class instance symbol*`)` обобщенная функция

Вызывается, когда `slot-value` запрашивает значение слота *symbol* в экземпляре *instance* (класса *class*), а этот слот является несвязанным. Первичный встроенный метод вызывает ошибку. Если новый метод возвращает значение, это значение будет возвращено `slot-value`.

- `(slot-value` *instance symbol*`)` функция

Возвращает значение слота *symbol* в экземпляре *instance*. Если слот отсутствует, вызывает `slot-missing`, если несвязан – `slot-unbound`. По умолчанию в обоих случаях вызывается ошибка. Может быть первым аргументом `setf`.

- `(with-accessors ((`*var fname*`)*)` *instance declaration** *expression**`)` макрос

Вычисляет свое тело предварительно связав каждую переменную с результатом вызова соответствующей функции для *instance*. Каждое *fname* должно быть именем функции доступа для экземпляра.

- `(with-slots ({`*symbol* `|` `(`*var symbol*`)}*)` *instance declaration** *expression**`)` макрос

Вычисляет свое тело, предварительно связав каждый символ (или *var*, если задана) с локальным символ-макросом, ссылающимся на слот с именем *symbol* для экземпляра *instance*.

- `(unbound-slot-instance` *condition*`)` функция

Возвращает экземпляр, чей слот был несвязан в момент возникновения исключения *condition*.

- `(update-instance-for-different-class` *old new* `&rest` *initargs* `&key)` обобщенная функция

Вызывается из `change-class` для установления значений слотов при изменении класса экземпляра. Экземпляр *old* является копией оригинального экземпляра с использованием dynamic extent; *new* – оригинальный экземпляр, к которому добавляются слоты. Встроенный первичный метод использует `shared-initialize` с: *new*, списком имен новых слотов, *initargs*. Допускает другие ключи.

- `(update-instance-for-redefined-class` *instance added deleted plist* `&rest` *initargs*`)` обобщенная функция

Вызывается из `make-instance-obsolete` для установления значений слотов при изменении класса экземпляра; *added* – это список добавляемых слотов, *deleted* – список удаляемых (включая те, которые перешли из локальных в разделяемые), а *plist* содержит элементы вида `(`*name* `.` *val*`)` для каждого элемента *deleted*, имевшего значение *val*. Встроенный первичный метод вызывает `shared-initialize` с: *instance*, *added* и *initargs*.

Структуры

- `(copy-structure` *structure*`)` функция

Возвращает новую структуру того же типа, что и *structure*, все значения которой равны (`eql`) старым.

- `(defstruct {`*symbol* `|` `(`*symbol property**`)}` `[`*string*`]` *field**`)` макрос

Определяет новый тип структуры с именем *symbol*, возвращает *symbol*. Если *symbol* уже соответствует имени структуры, последствия не определены, но фактически переопределение структуры с теми же параметрами безопасно. Если используется символ `str`, то попутно определяются: функция `make-str`, возвращающая новую структуру; предикат `str-p`, проверяющий на принадлежность к `str`; функцию `copy-str`, копирующую `str`; функции доступа к полям `str`; тип с именем `str`.

Строка *string*, если представлена, становится документацией к *symbol* с ключом `structure`. По умолчанию она также становится документацией с ключом `type`, а также документацией для сопутствующего класса.

Property может быть одним из:

`:conc-name | (:conc-name [`*name*`])`

Функция для доступа к полю f структуры `str` будет называться *namef* вместо `str-`*f*. Если *name* `nil` или не предоставляется, именем функции будет просто *f*.

`:constructor | (:constructor [`*name* `[`*parameters*`]])`

Если *name* не `nil`, функция создания новых структур будет называться *name*, в противном случае такая функция определяться не будет. Если имя не задается совсем, будет использоваться стандартное имя `make-str`. Если задан список полей *parameters*, он становится списком параметров функции-конструктора, и каждое поле вновь создаваемой структуры будет иметь значение, переданное соответствующим аргументом. Для одной структуры могут быть определены несколько конструкторов.

`:copier | (:copier [`*name*`])`

Если *name* не `nil`, функция копирования структуры будет называться *name*, в противном случае такая функция не будет определена. Если имя не задается совсем, используется стандартное имя `copy-str`.

`(:include` *name field*`*)`

Означает, что все структуры `str` также включают все поля существующей структуры типа *name*. Функции доступа включаются вместе с полями. Поля *fields* в `:include` используют обычный синтаксис (см. ниже); они могут использоваться для задания исходного значения поля (или его отсутствия), или для установки флага «только чтение», или для спецификации типа поля (который должен быть подтипом оригинального). Если структура содержит параметр `:type` (см. ниже), включаемая структура также должна быть этого типа, иначе тип включаемой структуры будет подтипом новой.

`(:initial-offset` *i*`)`

Структуры будут размещаться со смещением в *i* неиспользуемых полей. Используется только вместе с `:type`.

`:named`

Структуры будут размещаться так, что первым элементом является имя структуры. Используется только вместе с `:type`.

`:predicate | (:predicate [`*name*`])`

Если *name* не `nil`, предикат идентификации структуры будет называться *name*, в противном случае такая функция определена не будет. Если имя не задается совсем, будет использовано стандартное имя `str-p`. Не может использоваться вместе с `:type`, если также не задано `:named`.

`(:print-function [`*fname*`])`

Когда `str` выводится на печать, функция *fname* (имя или лямбда-выражение) будет вызываться с тремя аргументами: структура, поток, в который выполняется печать, и целое число, представляющее глубину печати. Реализована добавлением метода `print-object`. Не может быть использована вместе с `:type`.

`(:print-object [`*fname*`])`

Подобно `:print-function`, но вызывается лишь с двумя аргументами. Одновременно может использоваться либо `:print-function`, либо `:print-object`.

`(:type {vector | (vector` *type*`) | list})`

Указывает способ реализации структуры в виде объекта заданного типа. Структуры могут быть списками или векторами; не может быть добавлен никакой новый тип, а также предикат для идентификации структур (до тех пор, пока не задано `:named`). Если задан тип `(vector` *type*`)`, `:named` может использоваться лишь, когда *type* является надтипом `symbol`.

Каждое поле *field* может быть одиночным символом *name* или `(name [initform` *property*`*])`

Имя одного поля не должно совпадать с именем другого, включая локальные и наследуемые через `:include`. Имя поля может использоваться при создании функции доступа к этому полю; по умолчанию для структуры `str` будет определяться функция `str-name`, но это поведение можно настроить с помощью

`:conc-name`. Имя также становится ключевым параметром в функции создания структур. Исходное значение, если задано, вычисляется каждый раз при создании новой структуры в том окружении, где находился вызов `defstruct`. Если исходное значение не задано, значение поля на момент создания не определено. *Property* может быть одним из:

`:type` *type*

Декларирует, что данное поле будет содержать объекты только такого типа.

`:read-only`[1] *bool*

Если *bool* не `nil`, поле будет «только для чтения»

Исключения

- `(abort &optional` *condition*`)` функция

Исполняет перезапуск, возвращаемый `(find-restart 'abort` *condition*`)`.

- `(assert test [(`*place*`*) [`*cond arg*`*]])` макрос

Если вычисление *test* возвращает `nil`, вызывается корректируемая ошибка, указываются на значения *cond* и *args*. Значение *test* должно зависеть от адресов *place*; имеется возможность дать им новые значения и продолжить вычисление. При корректном завершении возвращает `nil`.

- `(break &rest` *args*`)` функция

Вызывает `format` с *args*, затем вызывает отладчик. Не сигнализирует исключение.

- `(cell-error-name` *condition*`)` функция

Возвращает имя объекта, вызвавшего исключение при попытке доступа к нему.

- `(cerror` *format cond* `&rest` *args*`)` функция

Действует подобно `error`, но позволяет продолжить вычисление с момента возникновения ошибки, возвращая `nil`. Строка *format* передается `format` при отображении информации об ошибке.

- `(check-type` *place type* `[`*string*`])` макрос

Вызывает корректируемую ошибку, если значение по адресу *place* не принадлежит типу *type*. Если задана строка *string*, то она будет выводиться как описание требуемого типа.

- `(compute-restarts &optional` *condition*`)` функция

Возвращает список незавершенных перезапусков, от последнего к первому. Если предоставлено исключение *condition*, список будет содержать перезапуски, связанные с данным исключением (а также не связанные с каким-либо исключением), иначе будет содержать все перезапуски. Возвращаемый список не должен изменяться.

- `(continue &optional` *condition*`)` функция

Возвращает перезапуск, возвращаемый `(find-restart 'abort` *condition*`)`, если он существует, иначе возвращает `nil`.

- `(define-condition` *name* `(`*parent*`*) (`*slot-spec*`*) *class-spec*`*)` макрос

Определяет новый тип исключений, возвращая его имя. Имеет такой же синтаксис и поведение, что и `defclass`, за исключением того, что *class-specs* не может включать предложение `:metaclass` и может включать `:report`. Предложение `:report` определяет порядок вывода отчета об исключении. Аргумент может быть символом или лямбда-выражением, соответствующим функции двух аргументов (исключение и поток), или строкой.

- `(error` *cond* `&rest` *args*`)` функция

Сигнализирует простую ошибку для *cond* и *args*. Если она не ловится обработчиком, вызывается отладчик.

- `(find-restart` *r* `&optional` *condition*`)` функция

[1] Исправлена авторская опечатка, обнаруженная Лорентом Пероном. – *Прим. перев.*

Возвращает последний незавершенный перезапуск с именем *r*, если *r* символ, или с именем, равным (eq) *r*, если *r* – перезапуск. Если предоставлено исключение *condition*, учитываются лишь связанные с ним перезапуски (а также не связанные с каким-либо исключением). Если заданный перезапуск не найден, возвращает nil.

- (handler-bind ((*type handler*)*) *expression**) макрос

Вычисляет выражения с локальными обработчиками исключений. Если сигнализируется исключение, оно передается в функцию (одним из аргументов), связанную с первым *handler*, чей тип *type* соответствует исключению. Если обработчик отказывается от исключения (возвращая управление), поиск продолжается. Пройдя так все локальный обработчики, система начинает искать обработчики, существовавшие на момент вычисления handler-bind.

- (handler-case *test* (*type* ([*var*]) *declaration** *expression**)* макрос
  [(:no-error *parameters declaration** *expression**)])

Вычисляет *test*. Если сигнализируется исключение, и оно принадлежит одному из типов *type*, оно обрабатывается, и выражение handler-case возвращает результат(ы) вычисления выражений, связанных с первым совпавшим типом. Если исключение не возникло, и не задано поведение :no-error, то выражение handler-case возвращает результат выражения *test*. Если задано :no-error, handler-case возвращает результат(ы) вычисления его выражений, связав *parameters* со значениями, возвращенными *test*. Предложение :no-error не обязательно должно быть последним.

- (ignore-errors *expression**) макрос

Действует подобно progn, но выражения вычисляются с локальным обработчиком всех ошибок. Этот обработчик заставит выражения возвращать два значения: nil и вызванное исключение.

- (invalid-method-error *method format* &rest *args*) функция

Используется для сигнализации ошибки, когда один из применимых методов содержит некорректный квалификатор. Параметры *format* и *arg* используется для отображения сообщения об ошибке.

- (invoke-debugger *condition*) функция

Вызывает отладчик для заданного исключения.

- (invoke-restart *restart* &rest *args*) функция

Если *restart* – перезапуск, вызывает его функцию перезапуска с аргументами *args*; если это символ, вызывает функцию последнего незавершенного перезапуска с этим именем.

- (invoke-restart-interactively *restart*) функция

Действует подобно invoke-restart, но предоставляет интерактивную подсказку для ввода аргументов.

- (make-condition *type* &rest *initargs*) функция

Возвращает новое исключение типа *type*. По сути, это специализированная версия make-instance.

- (method-combination-error *format* &rest *args*) функция

Используется для сигнализации ошибки при комбинации методов. Аргументы передаются format для отображения сообщения об ошибке.

- (muffle-warning &optional *condition*) функция

Вызывает перезапуск, возвращаемый (find-restart 'muffle-warning *condition*).

- (restart-bind ((*symbol function* {*key val*}*)*) *expression**) макрос

Вычисляет выражения с новым ожидающим перезапуском. Каждый символ становится именем перезапуска с соответствующей функцией. (Если символ nil, перезапуск будет безымянным.) Ключи могут быть:

:interactive-function

Соответствующее значение должно вычисляться в функцию без аргументов, строящую список аргументов для `invoke-restart`. по умолчанию не передаются никакие аргументы.

`:report-function`

Соответствующее значение должно вычисляться в функцию одного аргумента (поток), печатающую в поток описание действий перезапуска.

`:test-function`

Соответствующее значение должно вычисляться в функцию одного аргумента (исключение), которая возвращает истину, когда перезапуск применим с данным исключением. По умолчанию перезапуск применим с любым исключением.

- `(restart-case` *test* `(`*symbol parameters* `{`*key val*`}`* *declaration** *expression**`)*` макрос

Вычисляет *test* с новыми ожидающими перезапусками. Каждый символ становится именем перезапуска с функцией `(lambda` *parameters declaration** *expression**`)`. (Если символ `nil`, перезапуск будет безымянным.) Ключи могут быть:

`:interactive`

Соответствующее значение должно быть символом или лямбда-выражением, описывающим функцию без аргументов, строящую список аргументов для `invoke-restart`. По умолчанию не передаются никакие аргументы.

`:report`

Соответствующее значение может быть строкой, описывающей действия перезапуска, а также символом или лямбда-выражением, описывающим функцию одного аргумента (поток), выводящую в поток описание перезапуска.

`:test`

Соответствующее значение должно быть символом или лямбда-выражением, описывающим функцию одного аргумента (исключение), которая истинна, когда перезапуск применим с данным исключением. По умолчанию перезапуск применим с любым исключением.

- `(restart-name` *restart*`)` функция

Возвращает имя перезапуска или `nil`, если он безымянный.

- `(signal` *cond* `&rest` *args*`)` функция

Сигнализирует исключение, связанное с *cond* и *args*. Если оно не ловится обработчиком, возвращается `nil`.

- `(simple-condition-format-arguments` *condition*`)` функция

Возвращает параметры `format` для `simple-condition`.

- `(simple-condition-format-control` *condition*`)` функция

Возвращает строку (или функцию) `format` для `simple-condition`.

- `(store-value` *object* `&optional` *condition*`)` функция

Вызывает перезапуск, возвращаемый `(find-restart 'store-value` *condition* `)` для объекта *object*. Если он не найден, возвращает `nil`.

- `(use-value` *object* `&optional` *condition*`)` функция

Вызывает перезапуск, возвращаемый `(find-restart 'use-value` *condition*`)` для объекта *object*. Если он не найден, возвращает `nil`.

- `(warn` *cond* `&rest` *args*`)` функция

Сигнализирует `simple-warning` для *cond* и *args*. Если оно не ловится обработчиком, печатается сообщение об ошибке в `*error-output*` и возвращается `nil`.

- `(with-condition-restarts` *condition restarts expression**`)` макрос

Вычисление первого исключения *condition* производит исключение, *restarts* – список перезапусков. Затем все перезапуски связываются с полученным условием, и вычисляются остальные выражения.

- (`with-simple-restart (`*symbol format arg**`)` *expression**`)` макрос

Вычисляет выражения с новым перезапуском с именем *symbol*, который, будучи вызванным, заставляет `with-simple-restart` возвращать два значения: `nil` и `t`. Аргументы *format* и *args* передаются в `format` для вывода описания перезапуска.

Символы

- `(boundp` *symbol*`)` функция

Возвращает истину, когда символ является именем специальной переменной.

- `(copy-symbol` *symbol* `&optional` *props-too*`)` функция

Возвращает новый неинтернированный символ с именем, равным `(string=)` данному. Если параметр *props-too* истинен, новый символ будет иметь те же `symbol-value`, `symbol-function`, что и *symbol*, а также копию его `symbol-plist`.

- `(gensym &optional` *prefix*`)` функция

Возвращает новый неинтернированный символ. По умолчанию он начинается с `"G"` и содержит увеличенное представление значения счетчика `*gensym-counter*`. Если предоставлена строка-префикс, она будет использоваться вместо `"G"`.

- `(gentemp &optional (`*prefix* `"T") `*package*`)` [функция]

Возвращает новый символ, интернированный в пакет *package*. Имя символа содержит префикс и значение внутреннего счетчика, которое увеличивается, если символ с таким именем уже существует.

- `(get` *symbol key* `&optional` *default*`)` функция

Возвращает значение для заданного ключа из списка свойств символа. Возвращает *default*, если искомое свойство отсутствует. Может быть первым аргументом `setf`.

- `(keywordp` *object*`)` функция

Возвращает истину, когда символ принадлежит пакету `keyword`.

- `(make-symbol` *string*`)` функция

Возвращает новый неинтернированный символ с именем, равным `(string=)` строке *string*.

- `(makunbound` *symbol*`)` функция

Удаляет специальную переменную с именем *symbol*, если такая имеется. После этого `(boundp` *symbol*`)` не будет возвращать истину. Возвращает *symbol*.

- `(set` *symbol object*`)` [функция]

Эквивалент `(setf (symbol-value` *symbol*`)` *object*`)`.

- `(symbol-value` *symbol*`)` функция

Возвращает глобальную функцию с именем *symbol*. Сигнализирует ошибку, если функция с таким именем отсутствует. Может быть первым аргументом `setf`.

- `(symbol-name` *symbol*`)` функция

Возвращает строку, являющуюся именем символа. Строка не может быть изменена.

- `(symbol` *object*`)` функция

Возвращает истину, когда объект является символом.

- `(symbol-package` *symbol*`)` функция

Возвращает домашний пакет для символа.

- `(symbol-plist` *symbol*`)` функция

Возвращает список свойств символа. Может быть первым аргументом `setf`.

- `(symbol-value` *symbol*`)` функция

Возвращает значение специальной переменной с именем *symbol*. Сигнализирует ошибку, если такая переменная не существует. Может быть первым аргументом `setf`.

- `(remprop <`*symbol*`> key)` функция

Деструктивно удаляет первый найденный в списке свойств символа ключ *key* и связанное с ним значение. Возвращает истину, если заданный ключ был найден.

Пакеты

- `(defpackage` *name property*`*)` макрос

Возвращает пакет с именем *name* (строка или символ) и заданными свойствами. Если пакет с таким именем не существует, он создается, иначе вносятся изменения в уже имеющийся пакет. Свойствами могут быть:

`(:nicknames` *name*`*)`

Делает имена *name* (символы или строки) мнемоническими именами пакета.

`(:documentation` *string*`)`

Делает *string* строкой документации к пакету.

`(:use` *package*`*)`

Определяет пакеты, используемые данным; см. `use-package`.

`(:shadow` *name*`*)`

Заданные имена *name* (символы или строки) определяют затеняемые символы; см. `shadow`.

`(:shadow-import-from` *package name*`*)`

Имена могут быть символами и строками; соответствующие символы из *package* будут импортироваться в пакет как с помощью `shadowing-import`.

`(:import-from` *package name*`*)`

Имена могут быть символами и строками; соответствующие символы из *package* будут импортироваться в пакет как с помощью `import`.

`(:export` *name*`*)`

Имена могут быть символами и строками; соответствующие символы будут экспортироваться из *package*; см. `export`.

`(:intern` *name*`*)`

Имена могут быть символами и строками; создает соответствующие символы, если они еще не существуют; см. `intern`.

`(:size` *integer*`)`

Декларирует ожидаемое количество символов в данном пакете.

Любые свойства, кроме `:documentation` и `:size`, могут встречаться неоднократно. Порядок применения свойств таков: `:shadow` и `:shadowing-import-from`, `:use`, `:import-from` и `:intern`, затем `:export`. Вся работа выполняется на этапе компиляции, если выражение находится в toplevel.

- `(delete-package` *package*`)` функция

Удаляет пакет из списка активных, но представляющий его объект не затрагивается. Возвращает истину, если до удаления пакет находился в списке активных.

- `(do-all-symbols (`*var* `[`*result*`]) declaration*` `{`*tag* `|` *expression*`}*)` макрос

Действует подобно `do-symbols`, но итерирует по всем активным пакетам.

- `(do-external-symbols (`*var* `[`*package* `[`*result*`]]) declaration*` *tag* `|` *expression*`}*)` макрос

Действует подобно `do-symbols`, но итерирует только по внешним символам пакета.

- `(do-symbols (`*var* `[`*package* `[`*result*`]]) declaration*` `{`*tag* `|` *expression*`}*)` макрос

Вычисляет свое тело, связывая *var* по очереди со всеми символами, доступными в пакете *package*. Символы, наследуемые из других пакетов, могут обрабатываться несколько раз. Тело заключается в `tagbody`, в весь вызов `do-symbols` – в блок с

именем `nil`. Возвращает значение(я) выражения *result* или `nil`, если *result* не предоставляется; *result* может ссылаться на *var*, которая будет `nil`.

- `(export` *symbols* `&optional` *package*`)` функция

Делает символ, доступный в данном пакете (если *symbols* – список, то каждый символ в нем) внешним для пакета *package*. Возвращает `t`.

- `(find-all-symbols` *name*`)` функция

Возвращает список, содержащий все символы активного пакета с именем *name* (символ или строка).

- `(find-package` *package*`)` функция

Возвращает пакет, на который указывает *package*, или `nil`, если такого пакета нет.

- `(find-symbol` *string* `&optional` *package*`)` функция

Возвращает символ с именем *string*, доступный в пакете *package*. Второе возвращаемое значение дает характер символа: `:internal`, `:external` или `:inherited`. Если символ не найдет, оба значения `nil`.

- `(import` *symbols* `&optional` *package*`)` функция

Делает символ, доступный в данном пакете (если *symbols* – список, то каждый символ в нем) доступным в пакете *package*. Для символов, не имеющих домашнего пакета, им станет *package*. Возвращает `t`.

- `(in-package` *name*`)` макрос

Делает пакет, соответствующий *name* (строка или символ), текущим. Работа выполняется на этапе компиляции, если выражение расположено в toplevel.

- `(intern` *string* `&optional` *package*`)` функция

Возвращает символ, доступный в *package*, с именем, равным (`string=`) *string*, создавая такой символ при необходимости. Второе возвращаемое значение дает характер символа: `:internal`, `:external`, `:inherited` или `nil` (что означает, что символ был только что создан).

- `(list-all-packages)` функция

Возвращает новый список всех активных пакетов.

- `(make-package` *name* `&key` *nicknames use*`)` функция

Возвращает новый пакет с именем *name* (строка или символ), мнемоническими именами *nicknames* (и строками, и соответствующими символами), и используемыми пакетами из use (список имен и/или символов).

- `(package-error-package` *condition*`)` функция

Возвращает пакет, вызвавший исключение *condition*.

- `(package-name` *package*`)` функция

Возвращает строку, являющуюся именем пакета, или *nil*, если пакет не является активным.

- `(package-nicknames` *package*`)` функция

Возвращает список строк, являющихся мнемоническими именами пакета.

- `(packagep` *object*`)` функция

Возвращает истину, если объект является пакетом.

- `(package-shadowing-symbols` *package*`)` функция

Возвращает список затеняемых символов для пакета.

- `(package-used-by-list` *package*`)` функция

Возвращает список пакетов, в которых используется пакет *package*.

- `(package-use-list` *package*`)` функция

Возвращает список пакетов, используемых пакетом *package*.

- `(rename-package` *<package>* *name* `&optional` *nicknames*`)` функция

Делает *name* (строка или символ) новым именем пакета, а также устанавливает новые мнемонические имена. Возвращает пакет.

- `(shadow` *names* `&optional` *package*`)` функция

Names может быть строкой, символом, списком строк и/или символов. Добавляет означенные символы в список затеняемых для данного пакета. Несуществующие символы при необходимости создаются. Возвращает `t`.

- `(shadowing-import` *symbols* `&optional` *package*`)` функция

Делает символ (если *symbols* – список, то каждый символ этого списка) внутренним для *package* и добавляет его в список затеняемых. Если в пакете уже существует доступный символ с таким именем, он удаляется. Возвращает `t`.

- `(unexport` *symbols* `&optional` *package*`)` функция

Делает символ (если *symbols* – список, то каждый символ этого списка) внутренним для *package*. Возвращает `t`.

- `(unintern` *symbol* `&optional` *package*`)` функция

Удаляет символ из пакета (и из списка затеняемых символов). Если пакет является домашним для символа, символ удаляется насовсем. Возвращает истину, если символ был доступен в *package*.

- `(unuse-package` *packages* `&optional` *package*`)` функция

Отменяет эффект `use-package`, использует те же аргументы. Возвращает `t`.

- `(use-package` *packages* `&optional` *package*`)` функция

Делает все внешние символы пакетов *packages* (это может быть пакет, строка, символ или список из них) доступными в *package*. Пакет `keyword` не может использоваться. Возвращает `t`.

- `(with-package-iterator (`*symbol packages key*`*)` *declaration*`*` *expression*`*)` макрос

Вычисляет выражения, связав *symbol* с локальным макросом, последовательно возвращающим символы из пакетов *packages* (это может быть пакет, символ, строка или список из них). Ключи `:internal`, `:external` и `:inherited` могут ограничивать круг символов. Локальный макрос возвращает четыре значения: одно истинно, когда возвращается символ (`nil` означает холостой запуск); символ; ключ, характеризующий символ (`:internal`, `:external` или `:inherited`); пакет, из которого был получен символ. Локальный макрос может возвращать символы в любом порядке, а также может возвращать один символ несколько раз, если он наследуется из нескольких пакетов.

Числа

- `(abs` *n*`)` функция

Возвращает неотрицательное действительное число той же величины, что и *n*.

- `(acos` *n*`)` функция

Возвращает арккосинус *n* (в радианах).

- `(acosh` *n*`)` функция

Возвращает гиперболический арккосинус *n*.

- `(arithmetic-error-operands` *condition*`)` функция

Возвращает список операндов для исключения, вызванного арифметической ошибкой.

- `(arithmetic-error-operation` *condition*`)` функция

Возвращает оператор (или его имя) для исключения, вызванного арифметической ошибкой.

- `(ash` *i pos*`)` функция

Возвращает целое число, полученное сдвигом числа в дополнительном коде[1] *i* на *pos* позиций влево (или вправо, если *pos* меньше нуля).

[1] Дополнительный код (two's complement) – распространенное машинное представление отрицательных чисел. В такой записи старший разряд определяет знак числа: 0 для положительных, 1 для отрицательных чисел. Далее такое представление для краткости может называться «дополнительным». – *Прим. перев.*

- `(asin n)` функция

Возвращает арксинус *n* (в радианах).

- `(asinh n)` функция

Возвращает гиперболический арксинус *n*.

- `(atan n1 &optional (n2 1))` функция

Возвращает арктангенс *n1*/*n2* (в радианах).

- `(atanh n)` функция

Возвращает гиперболический арктангенс *n*.

- `(boole op i1 i2)` функция

Возвращает целое число, полученное применением логического оператора *op* к числам в дополнительном коде *i1* и *i2*. Common Lisp определяет 16 констант, представляющих побитовые логические операции. В следующей таблице приведены действия, совершаемые `boole` для различных операторов:

Оператор	Результат
`boole-1`	`i1`
`boole-2`	`i2`
`boole-andc1`	`(logandc1 i1 i2)`
`boole-andc2`	`(logandc2 i1 i2)`
`boole-and`	`(logand i1 i2)`
`boole-c1`	`(lognot i1)`
`boole-c2`	`(lognot i2)`
`boole-clr`	всегда `0`
`boole-eqv`	`(logeqv i1 i2)`
`boole-ior`	`(logior i1 i2)`
`boole-nand`	`(lognand i1 i2)`
`boole-nor`	`(lognor i1 i2)`
`boole-orc1`	`(logorc1 i1 i2)`
`boole-orc2`	`(logorc2 i1 i2)`
`boole-set`	всегда `1`
`boole-xor`	`(logxor i1 i2)`

- `(byte length pos)` функция

Возвращает спецификатор байта длиной *length* бит, нижний бит которого представляет 2^{pos}.

- `(byte-position spec)` функция

Возвращает $\log_2$ числа, представляющего значение нижнего бита для байт-спецификатора *spec*.

- `(byte-size spec)` функция

Возвращает количество бит для байт-спецификатора *spec*.

- `(ceiling r &optional (d 1))` функция

Возвращает два значения: наименьшее целое число *i*, большее или равное *r*/*d*, и *r*−*id*; *d* должно быть ненулевым действительным числом.

- `(cis r)` функция

Возвращает комплексное число с действительной частью `(cos r)` и мнимой частью `(sin r)`.

- `(complex r1 &optional r2)` функция

Возвращает комплексное число в действительной частью *r1* и мнимой частью *r2* или ноль, если *r2* не задано.

- `(complexp object)` функция

Возвращает истину, когда объект является комплексным числом.

- `(conjugate n)` функция

Возвращает результат комплексного сопряжения *n*: *n*, если *n* действительное, и `#c(a –b)`, если *n* равен `#c(a b)`.

- `(cos n)` функция

Возвращает косинус *n* (в радианах).

- `(cosh n)` функция

Возвращает гиперболический косинус *n*.

- `(decf place [n])` макрос

Уменьшает значение по адресу *place* на *n*, если *n* не задан, то на `1`.

- `(decode-float f)` функция

Возвращает три значения: мантиссу *f*, его экспоненту и знак (`-1.0` для отрицательного, в противном случае `1.0`). Первое и третье значения – числа с плавающей запятой того же типа, что и *f*, а второе – целое число.

- `(denominator rational)` функция

Если *rational* – правильная рациональная дробь a/b, возвращает b.

- `(deposit-field new spec i)` функция

Возвращает результат замены битов в числе *i* согласно байт-спецификатору *spec* на соответствующие биты *new*.

- `(dpb new spec i)` функция

Возвращает результат замены младших *s* бит на *new* в числе *i* согласно байт-спецификатору *spec* размера *s*.

- `(evenp i)` функция

Возвращает истину, когда *i* четное.

- `(exp n)` функция

Возвращает e^n.

- `(expt n1 n2)` функция

Возвращает $n1^{n2}$.

- `(fceiling r &optional (d 1))` функция

Действует подобно `ceiling`, но первое число возвращается в десятичном формате.

- `(ffloor r &optional (d 1))` функция

Действует подобно `floor`, но первое число возвращается в десятичном формате.

- `(float n &optional f)` функция

Возвращает десятичную аппроксимацию *n* в формате *f*, или `single-float`, если *f* не задан.

- `(float-digits f)` функция

Возвращает количество цифр во внутреннем представлении *f*.

- `(floatp object)` функция

Возвращает истину, когда объект является числом с плавающей запятой.

- `(float-precision f)` функция

Возвращает количество значащих цифр во внутреннем представлении *f*.

- `(float-radix f)` функция

Возвращает основание представления *f*.

- `(float-sign f1 &optional (f2 (float 1 f1)))` функция

Возвращает положительное или отрицательное значение *f2* в зависимости от знака *f1*.

- `(floor r &optional (d 1))` функция

Возвращает два значения: наибольшее целое число, меньшее или равное r/d, и $r-id$; *d* должен быть ненулевым действительным числом.

- `(fround r &optional (d 1))` функция

Действует подобно *round*, но первое значение возвращается в десятичном формате.

- `(ftruncate r &optional (d 1))` функция

Действует подобно *truncate*, но первое значение возвращается в десятичном формате.

- `(gcd &rest is)` функция

Возвращает наибольший общий делитель своих аргументов, или 0, если аргументы не заданы.

- `(imagpart n)` функция

Возвращает мнимую часть *n*.

- `(incf place [n])` макрос

Увеличивает значение по адресу *place* на *n*, если *n* не задан, то на `1`.

- `(integer-decode-float f)` функция

Возвращает три целых числа, связанных друг с другом так же, как и возвращаемые `decode-float`.

- `(integer-length i)` функция

Возвращает количество бит, необходимых для представления i в виде дополнительного кода.

- `(integerp object)` функция

Возвращает истину, когда объект является целым числом.

- `(isqrt i)` функция

Возвращает наибольшее целое число, меньшее или равное квадратному корню *i*, который должен быть положительным числом.

- `(lcm &rest is)` функция

Возвращает наименьшее общее кратное своих аргументов и 1, если аргументов не задано.

- `(ldb spec i)` функция

Возвращает целое число, соответствующее битовому представлению *i* в соответствии с байт-спецификатором *spec*. Может быть первым аргументом `setf`.

- `(ldb-test spec i)` функция

Возвращает истину, если любой из битов *i* в соответствии с байт-спецификатором *spec* равен `1`.

- `(log n1 &optional n2)` функция

Возвращает $\log_{n2} n1$ или $\log_e n1$, если *n2* не задан.

- `(logand &rest is)` функция

Возвращает целочисленный результат логического И дополнительного представления аргументов, или `0`, если аргументы не заданы.

- `(logandc1 i1 i2)` функция

Возвращает целочисленный результат логического И дополнительных представлений *i2* и дополнения для *i1*.

- `(logandc1 i1 i2)` функция

Возвращает целочисленный результат логического И дополнительных представлений *i1* и дополнения для *i2*.

- `(logbitp pos i)` функция

Возвращает истину, когда бит с индексом *pos* дополнительного представления *i* равен `1`. Индекс младшего бита считается равным `0`.

- `(logcount i)` функция

Возвращает количество нулей в дополнительном представлении *i*, если *i* отрицательно, иначе количество единиц.

- `(logeqv &rest is)` функция

Возвращает целочисленный результат обратного исключающего ИЛИ[1] для дополнительных представлений аргументов, или `-1`, если аргументы не заданы.

- `(logior &rest` *is*`)` функция

Возвращает целочисленный результат операции исключающего ИЛИ для дополнительных представлений аргументов, или `0`, если аргументы не заданы.

- `(lognand` *i1 i2*`)` функция

Возвращает целочисленное дополнение результата применения логического И дополнительных представлений аргументов.

- `(lognor` *i1 i2*`)` функция

Возвращает целочисленное дополнение результата применения логического ИЛИ дополнительных представлений аргументов.

- `(lognot` *i*`)` функция

Возвращает целое число, чье дополнительное представление является дополнением до *i*.

- `(logorc1` *i1 i2*`)` функция

Возвращает целочисленный результат применения логического ИЛИ для дополнительных представлений *i2* и дополнения до *i1*.

- `(logorc2` *i1 i2*`)` функция

Возвращает целочисленный результат применения логического ИЛИ для дополнительных представлений *i1* и дополнения до *i2*.

- `(logtest` *i1 i2*`)` функция

Возвращает истину, если хотя бы одна их единиц в дополнительном представлении *i1* имеется в дополнительном представлении *i2*.

- `(logxor &rest` *is*`)` функция

Возвращает целочисленный результат применения логического исключающего ИЛИ к дополнительным представлениям аргументов, или 0, если аргументы не заданы.

- `(make-random-state &optional` *start*`)` функция

Возвращает новый генератор случайных чисел. Если *state* уже является генератором, возвращается его копия; если `nil`, то копия `*random-state*`, если `t`, то генератор, инициализированный случайным образом.

- `(mask-field` *spec i*`)` функция

Возвращает целое число, чье представление имеет такие же биты, как число *i* в области, определенной байт-спецификатором *spec*.

- `(max` *r1* `&rest` *rs*`)` функция

Возвращает наибольший из аргументов.

- `(min` *r1* `&rest` *rs*`)` функция

Возвращает наименьший из аргументов.

- `(minusp` *r*`)` функция

Возвращает истину, если *r* меньше нуля.

- `(mod` *r1 r2*`)` функция

Возвращает второе значение, которое вернул бы *floor* с теми же аргументами.

- `(numberp` *object*`)` функция

Возвращает истину, когда объект является числом.

- `(numerator` *rational*`)` функция

Если *rational* – правильная рациональная дробь *a*/*b*, возвращает *a*.

- `(oddp` *i*`)` функция

Возвращает истину, если *i* нечетное.

---

[1] XNOR, логическая операция, возвращающая истину, когда ее аргументы равны. – *Прим. перев.*

- `(parse-integer` *string* `&key` *start end radix junk-allowed*`)` функция

Возвращает два значение: целое число, прочитанное из строки (по умолчанию используется основание `10`), и позицию первого непрочитанного знака в ней. Параметры *start* и *end* ограничивают область чтения строки. Строка может содержать пробелы, знаки `+` и `-`, и должна содержать последовательность цифр, при необходимости завершаемую пробелами. (Макросы чтения не допускаются.) Если параметр *junk-allowed* ложен (по умолчанию), чтение строки, содержащей иные символы, приведет к возникновению ошибки; если он истинен, `parse-integer` просто вернет `nil`, если встретит недопустимый символ.

- `(phase` *n*`)` функция

Возвращает угловую часть полярного представления числа.

- `(plus` *r*`)` функция

Возвращает истину, когда *r* больше нуля.

- `(random` *limit* `&optional (`*state* `*random-state*))` функция

Возвращает случайное число, меньшее *limit* (который должен быть положительным целым числом или десятичной дробью) того же типа. Используется генератор *state* (который изменяется).

- `(random-state-p` *object*`)` функция

Возвращает истину, когда объект является генератором случайных чисел.

- `(rational` *r*`)` функция

Преобразует *r* в рациональную дробь. Если *r* является десятичной дробью, преобразование считается точным.

- `(rationalize` *r*`)` функция

Преобразует *r* в рациональную дробь. Если *r* является десятичной дробью, преобразование производится с сохранением точности представленного числа.

- `(rationalp` *object*`)` функция

Возвращает истину, когда объект является рациональной дробью.

- `(realp` *object*`)` функция

Возвращает истину, когда объект является действительным числом.

- `(realpart` *n*`)` функция

Возвращает действительную часть *n*.

- `(rem` *r1 r2*`)` функция

Возвращает второе значение вызова *truncate* с теми же аргументами.

- `(round` *r* `&optional (`*d* `1))` функция

Возвращает два значения: целое число, ближайшее к r/d, и $r-id$. Если r/d равноудалено от двух целых чисел, выбирается четное. Число *d* должно быть ненулевым действительным.

- `(scale-float` *f i*`)` функция

Возвращает результат умножения *f* на r^i, где *r* – основание представления числа с плавающей запятой.

- `(signum` *n*`)` функция

Если *n* действительное число, возвращает единицу, ноль или минус единицу для, соответственно, положительных, нулевых и отрицательных чисел. Если *n* комплексное, возвращается комплексное число в величиной один и той же фазой.

- `(sin` *n*`)` функция

Возвращает синус *n* (в радианах).

- `(sinh` *n*`)` функция

Возвращает гиперболический синус *n*.

- `(sqrt` *n*`)` функция

Возвращает квадратный корень *n*.

- `(tan n)` функция

Возвращает тангенс *n* (в радианах).

- `(tanh n)` функция

Возвращает гиперболический тангенс *n*.

- `(truncate r &optional (d 1))` функция

Возвращает два значения: целое число *i*, которое получается удалением всех цифр после запятой в десятичном представлении *r/d*, и *r−id*; *d* должно быть ненулевым действительным числом.

- `(upgraded-complex-part-type type)` функция

Возвращает тип частей наиболее специализированного комплексного числа, которое может хранить части с типами *type*.

- `(zerop n)` функция

Возвращает истину, когда *n* равен нулю.

- `(= n1 &rest ns)` функция

Возвращает истину, когда все числа равны попарно.

- `(/= n1 &rest ns)` функция

Возвращает истину, когда среди всех аргументов нет двух равных.

- `(> r1 &rest rs)` функция

Возвращает истину, когда каждый последующий аргумент меньше предыдущего.

- `(< r1 &rest rs)` функция

Возвращает истину, когда каждый последующий аргумент больше предыдущего.

- `(>= r1 &rest rs)` функция

Возвращает истину, когда каждый последующий аргумент меньше или равен предыдущему.

- `(<= r1 &rest rs)` функция

Возвращает истину, когда каждый последующий аргумент больше или равен предыдущему.

- `(* &rest ns)` функция

Возвращает произведение двух аргументов или `1`, если аргументы не заданы.

- `(+ &rest ns)` функция

Возвращает сумму аргументов или `0`, если аргументы не заданы.

- `(- n1 &rest ns)` функция

Вызванная с одним аргументом, возвращает *–n1*. Вызов вида `(- a1 ... an)` возвращает *a1 − ... − an*.

- `(/ n1 &rest ns)` функция

Вызванная с одним аргументом (который не должен быть нулем) возвращает обратный ему. Вызванная с несколькими аргументами, возвращает значение первого, разделенное на произведение остальных (которые не должны включать ноль.)

- `(1+ n)` функция

Эквивалент `(+ n 1)`.

- `(1- n)` функция

Эквивалент `(- n 1)`.

Знаки

- `(alpha-character-p char)` функция

Возвращает истину, когда *char* является буквенной литерой.

- `(both-case-p char)` функция

Возвращает истину, когда знак может быть верхнего и нижнего регистра.

- `(alphanumericp char)` функция

Возвращает истину, когда *char* является буквенной литерой или цифрой.

- `(character c)` функция

Возвращает знак, соответствующий знаку, строке, содержащей один знак или символу, имеющему такую строку в качестве имени.

- `(characterp object)` функция

Возвращает истину, когда объект является знаком.

- `(char-code char)` функция

Возвращает атрибут code знака *char*. Значение зависит от реализации, но в большинстве реализаций используется кодировка ASCII.

- `(char-downcase char)` функция

Если *char* находится в верхнем регистре, возвращает соответствующий знак из нижнего регистра, иначе возвращает *char*.

- `(char-greaterp char1 &rest chars)` функция

Действует подобно `char>`, но игнорирует регистр.

- `(char-equal char1 &rest chars)` функция

Действует подобно `char=`, но игнорирует регистр.

- `(char-int char)` функция

Возвращает неотрицательное целое число, представляющее *char*. Если знак не имеет атрибутов, определенных самой реализацией, результат такой же, как и для `char-code`.

- `(char-lessp char1 &rest chars)` функция

Действует подобно `char<`, но игнорирует регистр.

- `(char-name char)` функция

Возвращает строковое имя для *char* или `nil`, если имени-строки не задано.

- `(char-not-greaterp char1 &rest chars)` функция

Действует подобно `char<=`, но игнорирует регистр.

- `(char-not-equal char1 &rest chars)` функция

Действует подобно `char/=`, но игнорирует регистр.

- `(char-not-lessp char1 &rest chars)` функция

Действует подобно `char>=`, но игнорирует регистр.

- `(char-upcase char)` функция

Если char находится в нижнем регистре, преобразует его к верхнему, иначе возвращает *char*.

- `(char= char1 &rest chars)` функция

Возвращает истину, когда все аргументы одинаковы.

- `(char/= char1 &rest chars)` функция

Возвращает истину, когда нет ни одной пары равных аргументов.

- `(char< char &rest chars)` функция

Возвращает истину, когда каждый последующий аргумент больше предыдущего.

- `(char> char &rest chars)` функция

Возвращает истину, когда каждый последующий аргумент меньше предыдущего.

- `(char<= char &rest chars)` функция

Возвращает истину, когда каждый последующий аргумент больше или равен предыдущему.

- `(char>= char &rest chars)` функция

Возвращает истину, когда каждый последующий аргумент меньше или равен предыдущему.

- `(code-char code)` функция

Возвращает знак соответствующего ему кода.

- `(digit-char` *i* `&optional (`*r* `10))` функция

Возвращает знак, представляющий *i* с основанием *r*.

- `(digit-char-p` *char* `&optional (`*r* `10))` функция

Возвращает истину, когда *char* является цифрой с основанием *r*.

- `(graphic-char-p` *char*`)` функция

Возвращает истину, когда *char* является графическим знаком.

- `(lower-case-p` *char*`)` функция

Возвращает истину, когда *char* находится в нижнем регистре.

- `(name-char` *name*`)` функция

Возвращает знак с именем *name* (строка или символ). Нечувствительна к регистру.

- `(standard-char-p` *char*`)` функция

Возвращает истину, когда *char* является стандартным знаком.

- `(upper-case-p` *char*`)` функция

Возвращает истину, когда *char* находится в верхнем регистре.

Ячейки

- `(acons` *key value alist*`)` функция

Эквивалент `(cons (cons` *key value*`)` *alist*`)`.

- `(adjoin object` *prolist* `&key` *key test test-not*`)` функция

Если `member` вернет истину с теми же аргументами, возвращает *prolist*, иначе `(cons` *object* `prolist`*)*.

- `(append &rest` *prolists*`)` функция

Возвращает список с элементами из всех списков *prolists*. Последний аргумент, который может быть любого типа, не копируется, и `(cdr (append '(x) x)` будет равен (`eq`) `x`. Возвращает `nil` без аргументов.

- `(assoc` *key alist* `&key` *test test-not*`)` функция

Возвращает первый элемент *alist*, чей `car` совпадает с *key*.

- `(assoc-if` *predicate alist* `&key` *key*`)` функция

Возвращает первый элемент *alist*, для которого будет истинным *predicate*.

- `(assoc-if-not` *predicate alist* `&key` *key*`)` [функция]

Возвращает первый элемент *alist*, для которого будет ложным истинным *predicate*.

- `(atom` *object*`)` функция

Возвращает истину, когда объект не является ячейкой.

- `(butlast` *list* `&optional (`*n* `1))` функция
- `(nbutlast <`*list*`> &optional (`*n* `1))` функция

Возвращает копию списка без последних *n* элементов, или `nil`, если список имеет менее *n* элементов.

- `(car` *list*`)` функция

Если *list* ячейка, возвращает ее `car`. Если *list* `nil`, возвращает `nil`. Может устанавливаться через `setf`.

- `(cdr` *list*`)` функция

Если *list* ячейка, возвращает ее `cdr`. Если *list* `nil`, возвращает `nil`. Может устанавливаться через `setf`.

- `(c`*x*`r` *list*`)` функции

x представляет строку длиной от одного до четырех, состоящую из знаков *a* и *d*. Эквивалент соответствующего сочетания `car` и `cdr`. Например, `(cdaar x)` эквивалентно `(cdr (car (car x)))`. Может устанавливаться с помощью `setf`.

- `(cons` *object1 object2*`)` функция

Возвращает новую ячейку из двух объектов. Если *object2* является списком вида (*e1* ... *en*), то она вернет (*object1 e1* ... *en*).

- (consp *object*) функция

Возвращает истину, когда *object* является cons-ячейкой.

- (copy-alist *alist*) функция

То же, что и (mapcar #'(lambda (x) (cons (car x) (cdr x))) *alist*).

- (copy-list *list*) функция

Возвращает список, равный (equal) *list*, структура верхнего уровня которого состоит из новых ячеек. Если *list* равен nil, возвращается nil.

- (copy-tree *tree*) функция

Возвращает новое дерево той же формы и содержания, состоящее из новых ячеек. Если *tree* атом, возвращается *tree*.

- (endp *list*) функция

Возвращает истину, когда *list* равен nil.

- (first *list*) ... (tenth *list*) функция

Возвращают элементы списка от первого до десятого, или nil, если список не имеет столько элементов. Может устанавливаться через setf.

- (getf *plist key* &optional (*default* nil)) функция

Если *plist* имеет вид (*p1 v1* ... *pn vn*), и *pi* – первый ключ, равный (eq) *key*, возвращает *vi*. Если ключ не найден, возвращает *default*. Может устанавливаться через setf.

- (get-properties *plist prolist*) функция

Если *plist* имеет вид (*p1 v1* ... *pn vn*), и *pi* – первый ключ, равный (eq) какому-либо элементу *prolist*, возвращаются *pi*, *vi* и (*pi vi* ... *pn vn*), иначе возвращаются nil.

- (intersection *prolist1 prolist2* &key *test test-not*) функция
- (nintersection <*prolist1*> *prolist2* &key *test test-not*) функция

Возвращают список элементов *prolist1*, принадлежащих *prolist2*. Правильный порядок следования элементов не гарантируется.

- (last *list* &optional (*n* 1)) функция

Возвращает последние *n* ячеек в *list*, или *list*, если он содержит меньше *n* элементов. Если *n* равен 0, возвращает cdr последней ячейки *list*.

- (ldiff *list object*) функция

Если *object* является хвостом *list*, возвращает новый список из элементов до *object*. В противном случае возвращает копию *list*.

- (list &rest *objects*) функция

Возвращает список объектов.

- (list* *object* &rest *objects*) функция

Если передан только один аргумент, возвращается он сам. В противном случае, (list* *arg1* ... *argn*) эквивалентно (nconc (list *arg1* ... *argn-1*) *argn*).

- (list-length *list*) функция

Возвращает количество ячеек в списке или nil, если список является циклическим (в отличие от length, которая не определена для циклических списков). Если *list* является точечной парой, вызывается ошибка.

- (listp *object*) функция

Возвращает истину, когда *object* является списком, то есть, cons-ячейкой или nil.

- (make-list *n* &key (*initial-element* nil)) функция

Возвращает новый список из n элементов initial-element.

- (mapc *function prolist* &rest *prolists*) функция

Вызывает функцию *function* *n* раз (*n* – длина кратчайшего *prolist*): один раз для первого элемента каждого *prolist*, затем для всех *n*-ных элементов каждого *prolist*. Возвращает *prolist*.

- `(mapcan` *function prolist* `&rest` *prolists*`)` функция

Эквивалент применения `nconc` к результату вызова `mapcar` с теми же аргументами.

- `(mapcar` *function prolist* `&rest` *prolists*`)` функция

Вызывает функцию *function* *n* раз (*n* – длина кратчайшего *prolist*): один раз для первого элемента каждого *prolist*, затем для всех *n*-ных элементов каждого *prolist*. Возвращает список значений, полученных *function*.

- `(mapcon` *function prolist* `&rest` *prolists*`)` функция

Эквивалент применения `nconc` к результату вызова `maplist` с теми же аргументами.

- `(mapl function` *prolist* `&rest` *prolists*) функция

Вызывает функцию *function* *n* раз (*n* – длина кратчайшего *prolist*): один раз для первого элемента каждого *prolist*, затем для всех (*n-1*)-вых `cdr` каждого *prolist*. Возвращает *prolist*.

- `(maplist` *function prolist* `&rest` *prolists*`)` функция

Вызывает функцию *function* *n* раз (*n* – длина кратчайшего *prolist*): один раз для первого элемента каждого *prolist*, затем для всех (*n-1*)-вых `cdr` каждого *prolist*. Возвращает список значений, полученных *function*.

- `(member` *object prolist* `&key` *test test-not*`)` функция

Возвращает хвост *prolist*, начинающийся с первого элемента, совпадающего с *object*, или `nil`, если совпадений не найдено.

- `(member-if` *object prolist* `&key` *test test-not*`)` функция

Возвращает хвост *prolist*, начинающийся с первого элемента, для которого *predicate* вернул истину, или `nil`, если совпадений не найдено.

- `(member-if-not` *object prolist* `&key` *test test-not*`)` [функция]

Возвращает хвост *prolist*, начинающийся с первого элемента, для которого *predicate* вернул ложь, или `nil`, если совпадений не найдено.

- `(nconc &rest <`*lists*`>)` функция

Возвращает список с элементами из всех *lists*. Устанавливает `cdr` последней ячейки каждого списка в начало следующего списка. Последний аргумент может быть объектом любого типа. Без аргументов возвращает `nil`.

- `(nth` *n list*`)` функция

Возвращает (*n+1*)-й элемент *list*. Возвращает `nil`, если список имеет меньше элементов, чем (*n+1*). Может устанавливаться через `setf`.

- `(nthcdr` *n list*`)` функция

Эквивалент n-кратного последовательного применения `cdr` к *list*.

- `(null` *object*`)` функция

Возвращает истину, когда объект `nil`.

- `(pairlis` *keys values* `&optional` *alist*`)` функция

Возвращает то же значение, что и `(nconc (mapcar #'cons` *keys values*`)` *alist*`)` или `(nconc (nreverse (mapcar #'cons` *keys values*`))` *alist*`)`. Требует, чтобы *keys* и *values* были одинаковой длины.

- `(pop <`*place*`>)` макрос

Устанавливает значение по адресу *place*, которое должно вычисляться в *list*, в `(cdr` *list*`)`. Возвращает `(car` *list*`)`.

- `(push` *object* `<`*place*`>)` макрос

Устанавливает значение по адресу *place* в `(cons` *object place*`)`. Возвращает это значение.

- `(pushnew` *object* `<`*place*`>` `&key` *key test test-not*`)` макрос

Устанавливает значение по адресу *place*, которое должно вычисляться в нормальный список, в результат вызова `adjoin` с теми же аргументами. Возвращает новое значение *place*.

- `(rassoc` *key alist* `&key` *key test test-not*`)` функция

Возвращает первый элемент *alist*, `cdr` которого совпал с *key*.

- `(rassoc-if` *key alist* `&key` *key test test-not*`)` функция

Возвращает первый элемент *alist*, для `cdr` которого *predicate* вернул истину.

- `(rassoc-if-not` *key alist* `&key` *key test test-not*`)` [функция]

Возвращает первый элемент *alist*, для `cdr` которого *predicate* вернул ложь.

- `(remf <`*place*`> key)` макрос

Первый аргумент должен вычисляться в список свойств *plist*. Для *plist* вида $(p_1\ v_1\ \ldots\ p_n\ v_n)$ и p_i ровного (`eq`) *key*, деструктивно удаляет p_i и v_i из *plist* и устанавливает результат в *place*. Возвращает истину, если было удалено что-либо.

- `(rest` *list*`)` функция

То же. что и `cdr`. Может устанавливаться через `setf`.

- `(revappend` *list1 list2*`)` функция
- `(nreconc <`*list1*`> list2)` функция

Эквиваленты `(nconc (reverse` *list1*`) list2)` и `(nconc (nreverse` *list1*`) list2)`, соответственно.

- `(rplaca <`*cons*`> object)` функция

Эквивалент `(setf (car` *cons*`) object)`, но возвращает *cons*.

- `(rplacd <`*cons*`> object)` функция

Эквивалент `(setf (cdr` *cons*`) object)`, но возвращает *cons*.

- `(set-difference` *prolist1 prolist2* `&key` *key test test-not*`)` функция
- `(nset-difference <`*prolist1*`> prolist2 &key` *test test-not*`)` функция

Возвращают список элементов *prolist1*, не имеющихся в *prolist2*. Правильный порядок следования элементов не гарантируется.

- `(set-exclusive-or` *prolist1 prolist2* `&key` *key test test-not*`)` функция
- `(nset-exclusive-or <`*prolist1*`> prolist2 &key` *key test test-not*`)` функция

Возвращают список элементов, имеющихся либо в *prolist1*, либо в *prolist2*, но не в обоих списках одновременно. Правильный порядок следования элементов не гарантируется.

- `(sublis` *alist tree* `&key` *key test test-not*`)` функция
- `(nsublis` *alist <tree>* `&key` *key test test-not*`)` функция

Возвращают дерево подобное *tree*, но каждое поддерево, совпадающее с ключом в *alist*, заменяется на соответствующее ему значение. Если не было произведено никаких модификаций, возвращается *tree*.

- `(subsetp` *prolist1 prolist2* `&key` *key test test-not*`)` функция

Возвращает истину, когда каждый элемент *prolist1* имеется в *prolist2*.

- `(subst` *new old tree* `&key` *key test test-not*`)` функция
- `(nsubst` *new old <tree>* `&key` *key test test-not*`)` функция

Возвращает дерево, подобное *tree*, в котором все поддеревья, совпадающие с *old*, заменены на *new*.

- `(subst-if` *new predicate tree* `&key` *key*`)` функция
- `(nsubst-if` *new predicate <tree>* `&key` *key*`)` функция

Возвращает дерево, подобное *tree*, в котором все поддеревья, для которых справедлив предикат *predicate*, заменены на *new*.

- `(subst-if-not` *new predicate tree* `&key` *key*`)` [функция]

- `(nsubst-if-not` *new predicate* `<`*tree*`>` `&key` *key*`)` [функция]

Возвращает дерево, подобное *tree*, в котором все поддеревья, для которых не справедлив предикат *predicate*, заменены на *new*.

- `(tailp` *object list*`)` функция

Возвращает истину, когда *object* является хвостом *list*, то есть, когда *object* либо `nil`, либо одна из ячеек, из которых состоит *list*.

- `(tree-equal` *tree1 tree2* `&key` *test test-not*`)` функция

Возвращает истину, когда *tree1* и *tree2* имеют одну форму и все их элементы совпадают.

- `(union` *prolist1 prolist2* `&key key` *test test-not*`)` функция
- `(nunion` `<`*prolist1*`>` *prolist2* `&key` *key test test-not*`)` функция

Возвращает список элементов, имеющихся в *prolist1* и *prolist2*. Правильный порядок следования элементов в возвращаемом списке не гарантируется. Если либо *prolist1*, либо *prolist2* имеет повторяющиеся элементы, то эти элементы будут дублироваться в возвращаемом списке.

Массивы

- `(adjustable-array-p` *array*`)` функция

Возвращает истину, когда массив *array* расширяемый.

- `(adjust-array` `<`*array*`>` *dimensions* `&key ...)` функция

Возвращает массив, подобный *array* (или сам *array*, если он расширяемый) с некоторыми измененными свойствами. Если одна из размерностей уменьшается, исходный массив обрезается; если увеличивается, то новые элементы заполняются согласно параметру `:initial-element`. Ключи те же, что и для `make-array`, с некоторыми замечаниями:

`:element-type` *type*

Тип *type* должен быть совместимым с исходным.

`:initial-element` *object*

Новые элементы будут равны *object*, старые сохранят исходные значения.

`:initial-contents` *seq*

Как и в `make-array`, но существующие элементы будут перезаписаны.

`:fill-pointer` *object*

Если *object* `nil`, указатель заполнения останется прежним.

`:displaced-to` *array2*

Если исходный массив является предразмещенным, а *array2* `nil`, то соответствующие элементы старого оригинального массива будут скопированы в возвращаемый массив, заполняя все неустановленные значения (если задан) через `:initial-element` (если задан). Если исходный массив не является предразмещенным, и *array2* является массивом, исходное значение будет потеряно, и новый массив будет размещен поверх *array2*. В противном случае, действует так же, как `make-array`.

`:displaced-index-offset` *i*

Если этот аргумент не предоставляется, предразмещенный массив не будет смещен относительно оригинала.

- `(aref` *array* `&rest` *is*`)` функция

Возвращает элемент массива с индексами *is* (если массив нуль-мерный, и индексы не заданы, это один элемент). Игнорирует указатели заполнения. Может устанавливаться через `setf`.

- `(array-dimension` *array i*`)` функция

Возвращает длину *i*-ой размерности массива. Индексирование начинается с нуля.

- `(array-dimensions` *array*`)` функция

Возвращает список целых чисел, представляющих длины каждой размерности массива.

- `(array-displacement` *array*`)` функция

Возвращает два значения: массив, поверх которого предразмещен *array*, и его смещение. Возвращает `nil` и `0`, если массив не предразмещенный.

- `(array-element-type` *array*`)` функция

Возвращает тип элементов массива.

- `(array-has-fill-pointer-p` *array*`)` функция

Возвращает истину, когда массив имеет указатель заполнения.

- `(array-in-bounds-p` *array* `&rest` *is*`)` функция

Возвращает истину. если те же аргументы будут корректными аргументами `aref`.

- `(arrayp` *object*`)` функция

Возвращает истину, когда *object* массив.

- `(array-rank` *array*`)` функция

Возвращает количество размерностей массива.

- `(array-row-major-index` *array* `&rest` *is*`)` функция

Возвращает номер элемента для заданных индексов, считая массив одномерным и расположенным построчно. Индексирование начинается с нуля.

- `(array-total-size` *array*`)` функция

Возвращает количество элементов в массиве.

- `(bit bit-array &rest` *is*`)` функция

Аналог `aref` для бит-массива. Может устанавливаться через `setf`.

- `(bit-and <`*bit-array1*`> bit-array2 &optional <`*arg*`>)` функция

Работает с бит-массивами по аналогии с `logand` для целых чисел: складывает два бит-массива одинаковых размерностей, возвращая итоговый массив. Если *arg* `t`, возвращается новый массив; если `nil`, изменяется существующий *bit-array1*; если *arg* – бит-массив (такой же размерности), используется он.

- `(bit-andc1 <`*bit-array1*`> bit-array2 &optional <`*arg*`>)` функция

Аналог `bit-and` и `logandc1`.

- `(bit-andc2 <`*bit-array1*`> bit-array2 &optional <`*arg*`>)` функция

Аналог `bit-and` и `logandc2`.

- `(bit-eqv <`*bit-array1*`> bit-array2 &optional <`*arg*`>)` функция

Аналог `bit-and` и `logaeqv`.

- `(bit-ior <`*bit-array1*`> bit-array2 &optional <`*arg*`>)` функция

Аналог `bit-and` и `logior`.

- `(bit-nand <`*bit-array1*`> bit-array2 &optional <`*arg*`>)` функция

Аналог `bit-and` и `lognand`.

- `(bit-nor <`*bit-array1*`> bit-array2 &optional <`*arg*`>)` функция

Аналог `bit-and` и `lognor`.

- `(bit-not <`*bit-array*`> &optional <`*arg*`>)` функция

Работает с бит-массивами по аналогии с `lognot` для целых чисел: возвращает логическое дополнение до *bit-array*. Если *arg* `t`, возвращается новый массив; если `nil`, изменяется существующий *bit-array1*; если *arg* – бит-массив (такой же размерности), используется он.

- `(bit-orc1 <`*bit-array1*`> bit-array2 &optional <`*arg*`>)` функция

Аналог `bit-and` и `logorc1`.

- `(bit-orc2 <`*bit-array1*`> bit-array2 &optional <`*arg*`>)` функция

Аналог `bit-and` и `logorc2`.

- `(bit-xor <`*bit-array1*`> bit-array2 &optional <`*arg*`>)` функция

Аналог `bit-and` и `logxor`.

- `(bit-vector-p` *object*`)` функция

Возвращает истину, если объект является бит-вектором.

- `(fill-pointer` *vector*`)` функция

Возвращает указатель заполнения вектора. Может устанавливаться через `setf`, но лишь когда вектор уже имеет указатель заполнения.

- `(make-array` *dimensions* `&key` *element-type initial-element initial-contents adjustable* функция
  fill-pointer displaced-to displaced-index-offset`)`

Возвращает новый массив с размерностями *dimensions* (это может быть одним числом, тогда создается вектор заданной длины). По умолчанию элементы могут иметь любой тип, если не задан конкретный тип. Доступны следующие ключевые параметры:

`:element-type` *type*

Декларирует тип элементов массива.

`:initial-element` *object*

Каждым элементом создаваемого массива будет *object*. Не может использоваться вместе с `:initial-contents`.

`:initial-contents` *seq*

Элементами массива будут соответствующие элементы набора вложенных последовательностей *seq*. Для нуль-размерного массива может использоваться одиночный аргумент.

`:adjustable` *object*

Если *object* истинен, создаваемый массив гарантированно будет расширяемым; впрочем, он может быть таковым и в противном случае.

`:fill-pointer` *object*

Если *object* истинен, массив (должен быть вектором) будет иметь указатель заполнения. Если объект является целым числом от нуля до длины вектора, он будет исходным значением указателя.

`:displaced-to` *array*

Массив будет предразмещен поверх *array*. Ссылка на его элемент будет транслироваться в ссылку на соответствующий элемент *array*.

`:displaced-index-offset` *i*

Смещение отображения на целевой массив. Может задаваться лишь совместно с `:displaced-to`.

- `(row-major-aref` *array i*`)` функция

Возвращает *i*-ый элемент массива *array*, рассматриваемого как одномерный с построчным заполнением. Индексирование начинается с нуля. Может устанавливаться через `setf`.

- `(sbit simple-bit-array &rest` *is*`)` функция

Аналог `aref` для простых бит-массивов. Может устанавливаться через `setf`.

- `(simple-bit-vector-p` *object*`)` функция

Возвращает истину, когда *object* является простым бит-массивом.

- `(simple-vector-p` *object*`)` функция

Возвращает истину, когда *object* является простым вектором.

- `(svref` *simple-vector i*`)` функция

Возвращает *i*-ый элемент *simple-vector*. Индексирование начинается с нуля, значение может устанавливаться через `setf`.

- `(upgraded-array-element-type` *type* `&optional` *env*`)` функция

Возвращает действительный тип элементов, с которым может работать реализация, для массива с `:element-type` *type*.

- `(vector &rest objects)` функция

Возвращает новый простой вектор с элементами *objects*.

- `(vectorp &rest objects)` функция

Возвращает истину, когда *object* является вектором.

- `(vector-pop vector)` функция

Уменьшает индекс заполнения вектора и возвращает элемент, на который он указывал. Попытка применения к вектору без указателя заполнения или с указателем, равным `0`, приводит к ошибке.

- `(vector-push object vector)` функция

Если указатель заполнения равен длине вектора, возвращает `nil`. В противном случае замещает элемент, на который указывает указатель заполнения, на *object* и увеличивает указатель заполнения, возвращая старое значение. Попытка применения к вектору без указателя заполнения приводит к ошибке.

- `(vector-push-extend object vector &optional i)` функция

Аналог `vector-push`, но если указатель заполнения равен длине вектора, вектор предварительно удлиняется на *i* элементов (или на умолчальное значение, зависящее от реализации) через `adjust-array`.

Строки

- Ё011in`(char string i)` функция

Возвращает *i*-ый знак в строке. Индексация начинается с нуля. Игнорирует указатели заполнения Работает с `setf`.

- `(make-string n &key initial-element (element-type 'character))` функция

Возвращает новую строку из *n* элементов *initial-element* (умолчальное значение зависит от реализации).

- `(schar simple-string i)` функция

Действует подобно `char`, но применима только к простым строкам. Работает с `setf`.

- `(simple-string-p object)` функция

Возвращает истину, когда *object* является простой строкой.

- `(string arg)` функция

Если *arg* строка, она возвращается; если это символ, возвращается его имя; если знак, возвращается содержащая его строка.

- `(string-capitalize string &key start end)` функция
- `(nstring-capitalize <string> &key start end)` функция

Возвращает строку, в которой первая буква каждого слова находится в верхнем регистре, а все остальные – в нижнем. Словом считается любая последовательность алфавитных символов. Первым аргументом `string-capitalize` может быть символ, в таком случае будет использоваться его имя.

- `(string-downcase string &key start end)` функция
- `(nstring-downcase <string> &key start end)` функция

Действуют подобно `string-upcase` и `nstring-upcase`, но знаки преобразуются к нижнему регистру.

- `(string-equal string1 string2 &key start1 end1 start2 end2)` функция

Действует подобно `string=`, но игнорирует регистр.

- `(string-greaterp string1 string2 &key start1 end1 start2 end2)` функция

Действует подобно `string>`, но игнорирует регистр.

- `(string-upcase string &key start end)` функция

- `(nstring-upcase <string> &key start end)` функция

Возвращают строку, в которой все знаки в нижнем регистре преобразованы к соответствующим знакам в верхнем. Параметры *start* и *end* действуют так же, как и в других строковых функциях. Первый аргумент `string-upcase` может быть символом, в таком случае используется его имя.

- `(string-left-trim seq string)` функция

Действует подобно `string-trim`, но обрезает только передний край.

- `(string-lessp string1 string2 &key start1 end1 start2 end2)` функция

Действует подобно `string<`, но игнорирует регистр.

- `(string-not-equal string1 string2 &key start1 end1 start2 end2)` функция

Действует подобно `string/=`, но игнорирует регистр.

- `(string-not-greaterp string1 string2 &key start1 end1 start2 end2)` функция

Действует подобно `string<=`, но игнорирует регистр.

- `(string-not-lessp string1 string2 &key start1 end1 start2 end2)` функция

Действует подобно `string>=`, но игнорирует регистр.

- `(stringp object)` функция

Возвращает истину, когда *object* является строкой.

- `(string-right-trim seq string)` функция

Действует подобно `string-trim`, но обрезает только задний край.

- `(string-trim seq string)` функция

Возвращает строку, подобную *string*, в которой все знаки, перечисленные в *seq*, удалены с обоих концов.

- `(string= string1 string2 &key start1 end1 start2 end2)` функция

Возвращает истину, когда подпоследовательности *string1* и *string2* имеют одну длину и содержат одинаковые знаки. Параметры *start1* и *end1*, *start2* и *end2*, работают как обычные *start* и *end* для *string1* и *string2*, соответственно.

- `(string/= string1 string2 &key start1 end1 start2 end2)` функция

Возвращает истину, когда `string=` возвращает ложь.

- `(string< string1 string2 &key start1 end1 start2 end2)` функция

Возвращает истину, когда два подпоследовательности содержат одинаковые знаки до конца первой последовательности, и вторая длиннее первой; или когда подпоследовательности имеют разные знаки, и для первой отличающейся паре знаков справедлив `char<`. Параметры те же, что и в `string=`.

- `(string> string1 string2 &key start1 end1 start2 end2)` функция

Возвращает истину, когда два подпоследовательности содержат одинаковые знаки до конца второй последовательности, и первая длиннее второй; или когда подпоследовательности имеют разные знаки, и для первой отличающейся паре знаков справедлив `char>`. Параметры те же, что и в `string=`.

- `(string<= string1 string2 &key start1 end1 start2 end2)` функция

Возвращает истину, когда истинны `string<` либо `string=` с теми же аргументами.

- `(string>= string1 string2 &key start1 end1 start2 end2)` функция

Возвращает истину, когда истинны `string>` либо `string=` с теми же аргументами.

Последовательности

- `(concatenate type &rest sequences)` функция

Возвращает новую последовательность типа *type* со следующими по порядку элементами последовательностей *sequences*. Копирует каждую последовательность, даже последнюю.

- `(copy-seq` *proseq*`)` функция

Возвращает новую последовательность того же типа и с теми же элементами.

- `(count` *object proseq* `&key` *key test test-not from-end start end*`)` функция

Возвращает количество совпадающих с *object* элементов *proseq*.

- `(count-if` *predicate proseq* `&key` *key test test-not from-end start end*`)` функция

Возвращает количество соответствующих предикату элементов proseq.

- `(count-if-not` *predicate proseq* `&key` *key test test-not from-end start end*`)` [функция]

Возвращает количество не соответствующих предикату элементов *proseq*.

- `(elt` *proseq n*`)` функция

Возвращает (*n*+1)-ый элемент *proseq*. Если длина *proseq* недостаточна, вызывается ошибка. Работает с `setf`.

- `(fill <`*proseq*`> object &key start end`) функция

Деструктивно заполняет *proseq* элементами *object*. Возвращает *proseq*.

- `(find` *object proseq* `&key` *key test test-not from-end start end*`)` функция

Возвращает первый совпадающий с *object* элемент *proseq*.

- `(find-if` *predicate proseq* `&key` *key test test-not from-end start end*`)` функция

Возвращает первый соответствующий предикату элемент proseq.

- `(find-if-not` *predicate proseq* `&key` *key test test-not from-end start end*`)` [функция]

Возвращает первый не соответствующий предикату элемент *proseq*.

- `(length` *proseq*`)` функция

Возвращает количество элементов в последовательности. Если *proseq* имеет указатель заполнения, учитываются лишь элементы до него.

- `(make-sequence` *type n* `&key (`*initial-element* `nil))` функция

Возвращается новая последовательность типа *type* из *n* элементов *initial-element*.

- `(map` *type function proseq* `&rest` *proseqs*`)` функция

Вызывает функцию *function n* раз (*n* – дина кратчайшей *proseq*): один раз для первого элемента каждой *proseq*, затем для всех *n*-ных элементов каждой *proseq*, Возвращает список значений, полученных *function*. (Если *type* `nil`, то ведет себя как `mapc` для последовательностей.)

- `(map-into <`*result*`> function proseq &rest proseqs`) функция

Вызывает функцию *function n* раз (*n* – дина кратчайшей *proseq*, включая *result*): один раз для первого элемента каждой *proseq*, затем для всех *n*-ных элементов каждой *proseq*, Деструктивно заменяет первые *n* элементов *result* на значение, возвращаемые *function*, и возвращает *result*.

- `(merge` *type* `<`*sequence1*`> <`*sequence2*`> predicate &key key`) функция

Эквивалент `(stable-sort (concatenate` *type sequence1 sequence2*`) predicate :key` *key*`)`, но деструктивна и более эффективна.

- `(mismatch` *sequence1 sequence2* `&key key test test-not from-end start1 end1 start2 end2`) функция

Возвращает положение (индексация с нуля) первого элемента *sequence1*, на котором начинается различие между *sequence1* и *sequence2*. Если последовательности полностью совпадают, возвращается `nil`.

- `(position` *object proseq* `&key` *key test test-not from-end start end*`)` функция

Возвращает положение (индексация с нуля) первого элемента *proseq*, совпадающего с *object*.

- `(position-if` *predicate proseq* `&key` *key from-end start end*`)` функция

Возвращает положение (индексация с нуля) первого элемента proseq, для которого будет справедлив предикат.

- `(position-if-not` *predicate proseq* `&key` *key from-end start end*`)` [функция]

Возвращает положение (индексация с нуля) первого элемента *proseq*, для которого будет несправедлив предикат.

- `(reduce` *function proseq* `&key` *key from-end start end initial-value*`)` функцияË052in

Поведение расписано в следующей таблице, где *f* – функция, а элементами *proseq* являются *a*, *b* и *c*:

from-end	*initial-value*	эквивалент
нет	нет	(*f* (*f a b*) *c*)
нет	да	(*f* (*f* (*f initial-value a*) *b*) *c*)
да	нет	(*f a* (*f b c*))
да	да	(*f a* (*f b* (*f c initial-value*)

Если *proseq* состоит из одного элемента, и *initial-value* не предоставляется, возвращается сам элемент. Если *proseq* пуста, и предоставлено *initial-value*, возвращается оно, однако, если оно также не предоставлено, функция вызывается без аргументов. Если предоставлены *key* и *initial-value*, *key* не применяется к *initial-value*.

- `(remove` *object proseq* `&key` *key test test-not from-end start end count*`)` функция
- `(delete` *object <proseq>* `&key` *key test test-not from-end start end count*`)` функция

Возвращают последовательность, похожую на *proseq*, но без всех элементов, совпадающих с *object*. Если предоставлен *count*, удаляется лишь столько экземпляров.

- `(remove-duplicates` *proseq* `&key` *key test test-not from-end start end count*`)` функция
- `(delete-duplicates <proseq>` `&key` *key test test-not from-end start end count*`)` функция

Возвращает последовательность, похожую на *proseq*, из которой удалены все повторяющиеся элементы, кроме последнего.

- `(remove-if` *object proseq* `&key` *key from-end start end count*`)` функция
- `(delete-if` *object <proseq>* `&key` *key from-end start end count*`)` функция

Возвращают последовательность, похожую на *proseq*, но без всех элементов, для которых справедлив *predicate*. Если предоставлен *count*, удаляется лишь столько экземпляров.

- `(remove-if-not` *object proseq* `&key` *key from-end start end count*`)` [функция]
- `(delete-if-not` *object <proseq>* `&key` *key from-end start end count*`)` [функция]

Возвращают последовательность, похожую на *proseq*, но без всех элементов, для которых несправедлив *predicate*. Если предоставлен *count*, удаляется лишь столько экземпляров.

- `(replace <sequence1>` *sequence2 &key start1 end1 start2 end2*`)` функция

Деструктивно заменяет *sequence1* на *sequence2* и возвращает *sequence1*. Количество заменяемых элементов равно длине кратчайшей последовательности. Работает, когда последовательности равны (`eq`), но не разделяют элементы структуры.

- `(nerverse` *proseq*`)` функция
- `(nreverse <`*proseq*`>)` функция

Возвращают последовательность того же типа, что и *proseq*, содержащую те же элементы в обратном порядке. Последовательность, возвращаемая *reverse*, всегда является копией. Если *proseq* вектор, то *reverse* возвращает простой вектор.

- `(search` *sequence1 sequence2* `&key` *key test test-not from-end start1 end1 start2 end2*`)` функция

Возвращает положение (индексация с нуля) начала первой совпадающей с *sequence1* подпоследовательности в *sequence2*. Если не найдено ни одной совпадающей подпоследовательности, возвращается `nil`.

- `(sort <proseq> predicate &key key)` функция

Возвращает последовательность того же типа, что и *proseq*, и с теми же элементами, отсортированными так, что для любых двух последующих элементов *e* и *f* значение `(`*predicate e f*`)` ложно, а `(`*predicate f e*`)` истинно.

- `(stable-sort <`*proseq*`>` *predicate* `&key` *key*`)` функция

Действует подобно `sort`, но старается максимально сохранить порядок следования элементов.

- `(subseq` *proseq start* `&optional` *end*`)` функция

Возвращает новую последовательность, являющуюся подпоследовательностью *proseq*. Параметры *start* и *end* ограничивают положение подпоследовательности: *start* указывает на положение (индексация с нуля) первого элемента подпоследовательности, а *end*, если задан, положение после последнего элемента подпоследовательности. Может устанавливаться через `setf`, а так же `replace`.

- `(substitute` *new old proseq* `&key` *key test test-not from-end start end count*`)` функция
- `(nsubstitute` *new old* `<`*proseq*`>` `&key` *key test test-not from-end start end count*`)` функция

Возвращают подпоследовательность, похожую на *proseq*, в которой все элементы, совпадающие с *old*, заменяются на *new*. Если предоставляется *count*, заменяются только такое количество совпавших элементов.

- `(substitute-if` *new predicate proseq* `&key` *key from-end start end count*`)` функция
- `(nsubstitute-if` *new predicate* `<`*proseq*`>` `&key` *key from-end start end count*`)` функция

Возвращают подпоследовательность, похожую на *proseq*, в которой все элементы, для которых справедлив *predicate*, заменяются на *new*. Если предоставляется *count*, заменяются только такое количество совпавших элементов.

- `(substitute-if-not` *new predicate proseq* `&key` *key from-end start end count*`)` [функция]
- `(nsubstitute-if-not` *new predicate* `<`*proseq*`>` `&key` *key from-end start end count*`)` [функция]

Возвращают подпоследовательность, похожую на *proseq*, в которой все элементы, для которых несправедлив *predicate*, заменяются на *new*. Если предоставляется *count*, заменяются только такое количество совпавших элементов.

Хеш-таблицы

- `(clrhash` *hash-table*`)` функция

Удаляет все элементы, хранящиеся в таблице, и возвращает ее.

- `(gethash` *key hash-table* `&optional` *default*`)` функция

Возвращает объект, соответствующий ключу *key* в *hash-table* или *default*, если заданный ключ не найден. Второе возвращаемое значение истинно, когда искомый элемент был найден. Работает вместе с `setf`.

- `(hash-table-count` *hash-table*`)` функция

Возвращает количество элементов в *hash-table*.

- `(hash-table-p` *object*`)` функция

Возвращает истину, когда элемент является хеш-таблицей.

- `(hash-table-rehash-size` *hash-table*`)` функция

Возвращает число, имеющее такое же значение, что и параметр `:rehash-size` в `make-hash-table`. Он показывает, насколько вырастет размер *hash-table* при ее расширении.

- `(hash-table-size` *hash-table*`)` функция

Возвращает емкость *hash-table*.

- `(hash-table-test` *hash-table*`)` функция

Возвращает функцию, используемую для определения равенства ключей в *hash-table*.

- `(make-hash-table &key` *test size rehash-size rehash-threshold*`)` функция

Возвращает новую хеш-таблицу, использующую *test* (по умолчанию `eql`) для установления равенства ключей. Размер *size* является предполагаемой емкостью создаваемой таблицы; *rehash-size* является предполагаемой емкостью, добавляемой к таблице при ее расширении (если `integer`, то объем складывается, если `float`, то умножается); положительное число *rehash-threshold* определяет степень заполнения, начиная с которой будет разрешено расширение таблицы.

- `(maphash` *function hash-table*`)` функция

Применяет *function*, которая должна быть функцией двух аргументов, к каждому ключу и соответствующему ему значению в *hash-table*.

- `(remhash key <`*hash-table*`>)` функция

Удаляет из *hash-table* объект, связанный с ключом *key*, возвращает истину, если ключ был найден.

- `(sxhash` *object*`)` функция

По сути, хеширующая функция для `equal`-таблиц. Возвращает уникальное неотрицательное число типа `fixnum` для каждого набора равных (`equal`) аргументов.

- `(with-hash-table-iterator (`*symbol hash-table*`)` *declaration** *expression**`)` макрос

Вычисляет выражения *expression*, связывая *symbol* с локальным макросом, последовательно возвращающим информацию об элементах *hash-table*. Локальный макрос обычно возвращает три значения: значение, означающее, что возвращаются другие значения (`nil` соответствует холостому запуску), ключ и соответствующее ему значение.

Пути к файлам

- `(directory-namestring` *path*`)` функция

Возвращает строку (зависящую от реализации), представляющую компоненту `directory` пути *path*.

- `(enough-namestring` *path* `&optional` *path2*`)` функция

Путь по умолчанию задается как *path2* (исходно `*default-pathname-defaults*`). Если путь *path* может быть сокращен с учетом *path2*, возвращается строку (зависящую от реализации), представляющую путь *path* относительно *path2*.

- `(file-namestring` *path*`)` функция

Возвращает строку (зависящую от реализации), представляющую компоненты `name`, `type` и `version` пути *path*.

- `(host-namestring` *path*`)` функция

Возвращает строку (зависящую от реализации), представляющую компоненту `host` пути *path*.

- `(load-logical-pathname-translations` *string*`)` функция

Загружает определение логического хоста с именем *string*, если оно до сих пор не загружено. Возвращает истину, если что-либо было загружено.

- `(logical-pathname` *path*`)` функция

Возвращает логический путь, соответствующий *path*.

- `(logical-pathname-translations` *host*`)` функция

Возвращает список преобразований для *host*, который должен быть логическим хостом или указывающей на него строкой.

- `(make-pathname &key` *host device directory name type version defaults case*`)` функция

Возвращает путь, составленный из своих аргументов. Значения незаданных аргументов берутся из *defaults*; если он не предоставляется, хостом по умолчанию считается `*default-pathname-defaults*`, остальные компоненты по умолчанию имеют значение `nil`.

Хост может быть строкой или списком строк; устройство может быть строкой; директория может быть строкой, списком строк или `:wild`; имя и тип могут быть строками или `:wild`; версия может быть неотрицательным целым числом, `:wild` или `:newest` (в большинстве реализаций также доступны `:oldest`, `:previous` и `:installed`). Все вышеперечисленные параметры могут быть `nil`, в этом случае компонента получает значение по умолчанию, или `:unspecified`, что означает неприменимость данной компоненты и не является переносимым.

Аргумент *defaults* может быть любым валидным аргументом *pathname* и расценивается подобным образом. Компоненты, которые не заданы или заданы как `nil`, получают свои значения из параметра *default*, если он задан.

Если *case* задан как `:local` (по умолчанию), компоненты пути будут в регистре локальной системы; если `:common`, то компоненты, находящиеся полностью в верхнем регистре используются так, как принято в данной системе, а компоненты в смешанном регистре используются без изменений.

- `(merge-pathnames` *path* `&optional` *default-path version*`)` функция

Возвращает путь, полученный заполнением всех недостающих компонент в *path* с помощью *default-path* (по умолчанию `*default-pathname-defaults*`). Если *path* включает компоненту `name`, версия может быть взята из *version* (по умолчанию `:newest`), в противном случае она берется из *default-path*.

- `(namestring` *path*`)` функция

Возвращает строку (зависящую от реализации), представляющую путь *path*.

- `(parse-namestring` *path* `&optional` *host default* `&key` *start end junk-allowed*`)` функция

Если путь не является строкой, возвращает соответствующий путь в нормальном виде. Если задана строка, обрабатывает ее как логический путь, получая хост из параметра *host*, самой строки или пути *default* (в приведенном порядке). Если не найдено валидного пути, и не задан *junk-allowed*, вызывает ошибку, с *junk-allowed* в таком случае возвращает `nil`. Параметры *start* и *end* используются по аналогии со строковыми функциями. Вторым значением возвращает положение в строке, на котором завершилось ее чтение.

- `(pathname` *path*`)` функция

Возвращает путь, соответствующий *path*.

- `(pathname-host` *path* `&key` *case*`)` функция

Возвращает компоненту `host` пути *path*. Параметр `:case` обрабатывается так же, как в `make-pathname`.

- `(pathname-device` *path* `&key` *case*`)` функция

Возвращает компоненту `device` пути *path*. Параметр `:case` обрабатывается так же, как в `make-pathname`.

- `(pathname-directory` *path* `&key` *case*`)` функция

Возвращает компоненту `directory` пути *path*. Параметр `:case` обрабатывается так же, как в `make-pathname`.

- `(pathname-match-p` *path wild-path*`)` функция

Возвращает истину, когда *path* совпадает с *wild-path*; любой отсутствующий компонент *wild-path* рассматривается как `:wild`.

- `(pathname-name` *path* `&key` *case*`)` функция

Возвращает компоненту *name* пути *path*. Параметр `:case` обрабатывается так же, как в `make-pathname`.

- `(pathnamep` *object*`)` функция

Возвращает истину, когда *object* является путем.

- `(pathname-type` *path* `&key` *case*`)` функция

Возвращает компоненту `type` пути *path*. Параметр `:case` обрабатывается так же, как в `make-pathname`.

- `(pathname-device` *version* `&key` *case*`)` функция

Возвращает компоненту *version* пути *path*. Параметр `:case` обрабатывается так же, как в `make-pathname`.

- `(translate-logical-pathname` *path* `&key)` функция

Возвращает физический путь, соответствующий *path*.

- `(translate-pathname` *path1 path2 path3* `&key)` функция

Преобразовывает путь *path1*, совпадающий с `wild`-путем *path2* в соответствующий путь, совпадающий с `wild`-путем *path3*.

- `(wild-pathname-p` *path* `&optional` *component*`)` функция

Возвращает истину, когда компонента пути, на которую указывает *component* (может быть `:host`, `:device`, `:directory`, `:name`, `:type` или `:version`) является `wild` или, если component задан как `nil`, когда *path* содержит какие-либо `wild`-компоненты.

Файлы

- `(delete-file` *path*`)` функция

Удаляет файл, на который указывает *path*. Возвращает `t`.

- `(directory` *path* `&key)` функция

Создает и возвращает список путей, соответствующих реальным файлам, совпадающим с шаблоном *path* (который может содержать `wild`-компоненты).

- `(ensure-directories-exist` *path* `&key` *verbose*`)` функция

Если директории, в которых находится файл *path*, не существуют, пытается создать их (информируя об этом, если задан *verbose*). Возвращает два значения: *path* и еще одно, истинное, когда были созданы какие-либо директории.

- `(file-author` *path*`)` функция

Возвращает строку, представляющую автора (владельца) файла *path* или `nil`, если владелец не может быть установлен.

- `(file-error-pathname` *condition*`)` функция

Возвращает путь, который привел к исключению *condition*.

- `(file-write-date` *path*`)` функция

Возвращает время, в том же формате, что и `get-universal-time`, соответствующее времени последнего изменения файла *path*, или `nil`, если время не может быть установлено.

- `(probe-file` *path*`)` функция

Возвращает актуальное имя файла, на который указывает *path*, или `nil`, если файл не найден.

- `(rename-file` *path1 path2*`)` функция

Переименовывает файл *path1* в *path2* (не может быть потоком). Незаданные компоненты в *file2* по умолчанию получаются из *path1*. Возвращает три значения: полученный путь, актуальное имя старого файла и актуальное имя нового файла.

- `(truename` *path*`)` функция

Возвращает актуальное имя файла, на который указывает *path*. Если файл не найден, вызывает ошибку.

Потоки

- `(broadcast-stream-streams` *broadcast-stream*`)` функция

Возвращает список потоков, составляющих *broadcast-stream*.

- `(clear-input &optional` *stream*`)` функция

Очищает содержимое входного потока *stream* и возвращает `nil`.

- `(clear-output &optional` *stream*`)` функция

Отменяет буферизированный вывод *stream* и возвращает `nil`.

- `(close` *stream* `&key` *abort*`)` функция

Закрывает поток *stream*, возвращая `t`, если поток был открытым. Если задан *abort*, пытается устранить все побочные эффекты от создания потока (например, связанный с ним выходной файл будет удален). Запись в закрытый поток невозможен, но он может передаваться вместо пути в другие функции, например, в `open`.

- `(concatenated-stream-streams` *concatenated-stream*`)` функция

Возвращает упорядоченный список потоков, из которых *concatenated-stream* будет продолжать чтение.

- `(echo-stream-input-stream` *echo-stream*`)` функция

Возвращает входной поток для *echo-stream*.

- `(echo-stream-output-stream` *echo-stream*`)` функция

Возвращает выходной поток для *echo-stream*.

- `(file-length` *stream*`)` функция

Возвращает количество элементов в потоке или `nil`, если значение не может быть определено.

- `(file-position` *stream* `&optional` *pos*`)` функция

Если *pos* не задан, возвращает текущее положение в потоке или `nil`, если положение не может быть определено. Если *pos* задан, он может быть `:start`, `:end` или неотрицательным целым числом, в соответствии с которым будет установлено положение новое в потоке.

- `(file-string-length` *stream object*`)` функция

Возвращает разницу между текущем положением в потоке и положением после записи в него объекта. Если не может быть определено, возвращает `nil`.

- `(finish-output &optional` *stream*`)` функция

Принудительно выводит содержимое буфера выходного потока, затем возвращает `nil`.

- `(force-output &optional` *stream*`)` функция

Действует подобно *finish-output*, но не ожидает завершения операции чтения/записи перед возвращением.

- `(fresh-line &optional` *stream*`)` функция

Записывает новую строку с поток, если текущее положение в потоке не соответствует началу новой строки.

- `(get-output-stream-string` *stream*`)` функция

Возвращает строку, содержащую все знаки, переданные в поток (который должен быть открытым) с момента его открытия или последнего вызова `get-output-stream-string`.

- `(input-stream-p` *stream*`)` функция

Возвращает истину, когда *stream* является входным потоком.

- `(interactive-stream-p` *stream*`)` функция

Возвращает истину, когда *stream* является интерактивным потоком.

- `(listen &optional` *stream*`)` функция

Возвращает истину, когда имеются знаки, ожидающие прочтения из потока, который предполагается интерактивным.

- `(make-broadcast-stream &rest` *streams*`)` функция

Возвращает новый широковещательный поток, составленный из *streams*.

- `(make-concatenated-stream &rest` *input-streams*`)` функция

Возвращает новый связанный поток, составленный из *input-streams*.

- `(make-echo-stream` *input-stream output-stream*`)` функция

Возвращает новый эхо-поток, получающий ввод из *input-stream* и направляющий его в *output-stream*.

- `(make-string-input-stream` *string* `&optional` *start end*`)` функция

Возвращает входной поток, который, при чтении из него, выдаст знаки, содержащиеся в строке, после чего выдаст конец файла. Параметры *start* и *end* используются, как и в других строковых функциях.

- `(make-string-output-stream &key` *element-type*`)` функция

Возвращает выходной поток, принимающий знаки типа *element-type*. Записанные в него знаки никуда не перенаправляются, но могут быть считаны через `get-output-stream-string`.

- `(make-synonym-stream` *symbol*`)` функция

Возвращает поток, который будет синонимичен потоку, соответствующему специальной переменной с именем *symbol*.

- `(make-two-way-stream` *input-stream output-stream*`)` функция

Возвращает новый двунаправленный поток, получающий вход от *input-stream* и направляющий свой вывод в *output-stream*.

- `(open` *path* `&key` *direction element-type if-exists if-does-not-exist external-format*`)` функция

Открывает и возвращает поток, связанный с *path*, или `nil`, если поток не может быть создан. Использует следующие ключевые параметры:

`:direction` *symbol*

Определяет направление потока объектов. Может быть `:input` (по умолчанию) для чтения из потока; `:output` для записи в поток, `:io` для обеих операций и `:probe`, чтобы был возвращен закрытый поток.

`:element-type` *type*

Декларирует тип объектов, с которыми будет работать поток. Символьный поток должен использовать какой-либо подтип `character`; бинарный поток должен использовать подтип либо `integer`, либо `signed-byte`, либо `unsigned-byte` (в этом случае размер элементов будет определяться операционной системой). По умолчанию задан `character`.

`:if-exists` *symbol*

Определяет действия в случае, когда с указанный файл уже существует. Возможны значения: `:new-version`, используется по умолчанию, если компонента *version* файла *path* равна `:newest`; `:error`, по умолчанию в противном случае; `:rename`, переименовывающее существующий файл; `:rename-and-delete`, переименовывающее и удаляющее его без вычеркивания; `:overwrite`, выполняющее запись поверх существующих данных с начала файла; `:append`, выполняющее запись в существующий файл, добавляя новые знаки в конец; `:supersede`, заменяющее существующий файл

новым с тем же именем (но исходный файл вероятно не удаляется до момента закрытия потока); `nil`, не создающий никаких потоков (`open` возвратит `nil`).

`:if-does-not-exist` *symbol*

Определяет действия в случае, когда указанный файл не найден. Возможные значения: `:error`, используемое по умолчанию для направления `:input`, или `:if-exists`, `:override` или `:append`; `:create`, используемое по умолчанию для направления `:output` или `:io`, или `if-exists` – ни `:overwrite`, ни `:append`; `nil`, по умолчанию для направления `:probe` (поток не создается, `open` возвращает `nil`).

`:external-format` *format*

Назначенный формат файла. Единственным предопределенным форматом является `:default`.

- `(open-stream-p` *stream*`)` функция

Возвращает истину, когда поток является открытым.

- `(peek-char &optional` *kind stream eof-error eof-value recursive*`)` функция

Возвращает, но не забирает текущий знак из потока. Если *kind* `nil`, возвращается первый найденный знак; если `t`, пропускает пробелы (а также табуляцию и проч.) и возвращает первый печатный знак; если задан знак, пропускает все знаки, кроме заданного, затем возвращает его. Если чтение дошло до конца файла, либо сигнализируется ошибка, либо возвращается *eof-value*, в зависимости от значения *eof-error* (по умолчанию истина). Параметр *recursive* будет истинным, когда `peek-char` вызывается из другой считывающей функции.

- `(read-byte` *stream* `&optional` *eof-error eof-value*`)` функция

Читает байт из бинарного входного потока. Если поток пуст, вызывается ошибка либо возвращается *eof-value* (в зависимости от *eof-error*).

- `(read-char &optional` *stream eof-error eof-value recursive*`)` функция

Забирает и возвращает первый знак из потока. Если встречен конец файла, сигнализирует об ошибки либо возвращает *eof-value*, в зависимости от значения *eof-error* (по умолчанию истина). Параметр *recursive* будет истинным, когда `read-char` вызывается из другой считывающей функции.

- `(read-char-no-hang &optional` *stream eof-error eof-value recursive*`)` функция

Действует подобно `read-char`, но сразу же возвращает `nil`, если поток не содержит непрочтенных знаков.

- `(read-line &optional` *stream eof-error eof-value recursive*`)` функция

Возвращает строку из знаков в потоке до первого знака переноса строки (он считывается, но не включается) или знака конца файла. Если до знака конца файла нет ни одного другого знака, вызывается ошибка либо *eof-value*, в зависимости от *eof-error* (по умолчанию истина). Параметр *recursive* будет истинным, когда `read-line` вызывается из другой считывающей функции.

- `(read-sequence <`*proseq*`> &optional` *stream* `&key` *start end*`)` функция

Считывает элементы из *stream* в *proseq*, возвращая положение первого неизмененного элемента. Параметры *start* и *end* те же, что и в других строковых функциях.

- `(stream-element-type` *stream*`)` функция

Возвращает тип объектов, которые могут быть прочитаны или записаны в *stream*.

- `(stream-error-stream` *condition*`)` функция

Возвращает поток, вовлеченный в исключение *condition*.

- `(stream-external-format` *stream*`)` функция

Возвращает внешний формат потока *stream*.

- `(streamp` *object*`)` функция

Возвращает истину, когда *object* является потоком.

- `(synonym-stream-symbol` *synonym-stream*`)` функция

Возвращает имя специальной переменной, для которой значение *synonym-stream* является сининомом.

- `(terpri &optional` *stream*`)` функция

Записывает новую строку в *stream*.

- `(two-way-stream-input-stream` *two-way-stream*`)` функция

Возвращает входной поток для *two-way-stream*.

- `(two-way-stream-output-stream` *two-way-stream*`)` функция

Возвращает выходной поток для *two-way-stream*.

- (`unread-char` *character* `&optional` *stream*`)` функция

Отменяет одно прочтение `read-char` для *stream*. Не может применяться дважды подряд без использования `read-char` в промежутке между ними. Не может использоваться после `peak-char`.

- `(with-input-from-string (`*symbol string* `&key` *index start end*`)` *declaration* expression**`)` макрос

Вычисляет выражения, связав *symbol* со строкой, полученной из входного потока с помощью `make-string-input-stream` с аргументами *string*, *start* и *end*. Создаваемый поток существует только внутри выражения `with-input-from-string` и закрывается после возврата значения или прерывания его вычисления. Параметр *index* может быть выражением (не вычисляется), служащим первым аргументом `setf`; если вызов `with-input-from-string` завершается нормально, значение по соответствующему адресу будет установлено в индекс первого непрочитанного из *stream* знака.

- `(with-open-file (`*symbol path arg**`)` *declaration* expression**`)` макрос

Вычисляет выражения *expression*, связав *symbol* с потоком, полученным передачей пути *path* и параметров arg функции `open`. Поток существует только внутри выражения `with-open-file` и закрывается автоматически при возврате из выражения `with-open-file` или прерыванию его вычисления. В последнем случае, если поток открывался на запись, созданный файл удаляется.

- `(with-open-stream (`*symbol stream*`)` *declaration* expression**`)` макрос

Вычисляет выражения, связав *symbol* со значением *stream*. Поток существует только внутри выражения `with-open-stream` и закрывается автоматически при возврате или прерыванию `with-open-stream`.

- `(with-output-to-string (`*symbol* `[`*string*`]` `&key` *element-type declaration* expression**`)` макрос

Вычисляет выражения *expression*, связав *symbol* с выходным потоком для строки. Поток существует только внутри выражения `with-output-to-string` и закрывается при возврате или прерывании. Если строка не задана, возвращается строка, состоящая из содержимого потоков. Если строка задается, она должна иметь указатель заполнения; символы, передаваемые в поток, добавляются в конец строки так же, как через `vector-push-extend`, а выражение `with-output-to-string` возвращает значение последнего выражения.

- `(write-byte` *i stream*`)` функция

Записывает *i* в бинарный выходной поток. Возвращает *i*.

- `(write-char` *character* `&optional` *stream*`)` функция

Записывает один знак в поток.

- `(write-line` *string* `&optional` *stream* `&key` *start end*`)` функция

Действует подобно *write-string*, но добавляет в конце перевод на новую строку.

- `(write-sequence` *proseq* `&optional` *stream* `&key` *start end*`)` функция

Записывает элементы последовательности в поток.

- `(write-string` *string* `&optional` *stream* `&key` *start end*`)` функция

Записывает элементы строки с поток.

- `(yes-or-no-p &optional` *format* `&rest` *args*`)` функция

Действует подобно `y-or-n-p`, но требует явного указания слов `yes` или `no` вместо одиночных знаков.

- `(y-or-n-p &optional` *format* `&rest` *args*`)` функция

Отображает вопрос *format* (по умолчанию `""`) в `*query-io*` так, как это сделал бы вызов `format` с аргументами *args*. Затем выдает приглашение и ожидает ввода `y` или `n`, возвращая истину или `nil` в зависимости от ответа. В случае некорректного ответа повторяет приглашение.

Печать

- `(copy-pprint-dispatch &optional pprint-dispatch)` функция

Возвращает копию таблицы управления аккуратной печатью *pprint-dispatch*, по умолчанию `*print-pprint-dispatch*`. Если *pprint-dispatch* `nil`, копирует исходное значение `*print-pprint-dispatch*`.

- `(format` *dest format* `&rest` *args*`)` функция

Записывает вывод в *dest*: это либо поток, либо `t` (запись в `*standard-output*`), либо `nil` (запись в строку). Возвращает `nil` или строку. Параметр *format* должен быть либо строкой, либо функцией, возвращаемой `formatter`. Если это функция, она примеряется к *dest* и *args*. Если это строка, она может содержать директивы форматирования, обычно начинающиеся со знака `~`, за которым следуют префиксные параметры, разделяемые запятыми, за которыми следуют необязательные модификаторы `:` и `@`, после которых следует определенный тег. Префиксом может быть: целое число; знак, предваряемый кавычкой; `V` или `v`, представляющими следующий аргумент (или отсутствие параметра, если аргумент `nil`); `#`, представляющим число оставшихся аргументов. Префиксные параметры могут пропускаться (запятые остаются), в этом случае используется значение по умолчанию. Завершающие запятые также могут опускаться.

Допустимы следующие директивы:

`~`*w*`,`*g*`,`*m*`,`*p`A`

Печатает следующий аргумент как `princ`, сдвинутый вправо (или влево, с помощью `@`) на по меньшей мере *m* (по умолчанию `0`) элементов *p* (по умолчанию `#\Space`), а также элементов *p*, собранных в группы по *g* (по умолчанию `1`) пока суммарное количество знаков (включая представление аргумента) не будет *w* (по умолчанию `0`) или более. С `:` пустые списки будут печататься как `()` вместо `nil`.

`~`*w*`,`*g*`,`*m*`,`*p`S`

То же, что и `~A`, но печатает аргумент как `prin1`.

`~W`

Печатает следующий аргумент как `write`. С : используется аккуратная печать, с `@` не учитываются ограничения на длину и вложенность списка.

`~C`

Следующий аргумент будет напечатан как знак. Если это простой знак, он будет напечатан как `write-char`. С : непечатные знаки убираются; `:@` делает то же, но также добавляются правила поведения для необычных знаков. С `@` знаки печатаются в синтаксисе `#\`.

`~n%`

Выводит *n* (по умолчанию `1`) переводов строки.

`~n&`

То же, что и `~%`, но первый перевод строки выполняется как с помощью `fresh-line`.

`~`*n*`~`

Печатает *n* (по умолчанию `1`) `~s`.

~*r*,*w*,*p*,*c*,*i*R

Очередной аргумент должен быть целым числом. Он печатается в системе с основанием *r* (по умолчанию 10), со смещением влево на столько *p* (по умолчанию #\Space), сколько требуется для того, чтобы суммарным количеством знаков было по меньшей мере *w*. С : группы *i* (по умолчанию 3) знаков разделяются элементами *c* (по умолчанию #\,). С @ знак печатается даже для положительных чисел.

Если никакие префиксы не предоставляются, ~R имеет совершенно другое значение, отображая числа в разнообразных нечисловых форматах. Без модификаторов 4 будет напечатано как four, с : как fourth, с @ как IV, а с :@ как IIII.

~*w*,*p*,*c*,*i*D

Отображает число в десятичном формате. Эквивалент ~10,*w*,*p*,*c*,*i*R.

~*w*,*p*,*c*,*i*B

Отображает число в бинарном формате. Эквивалент ~2,*w*,*p*,*c*,*i*R.

~*w*,*p*,*c*,*i*O

Отображает число в восьмеричном формате. Эквивалент ~8,*w*,*p*,*c*,*i*R.

~*w*,*p*,*c*,*i*X

Отображает числа в шестнадцатеричном формате. Эквивалент ~16,*w*,*p*,*c*,*i*R.

~*w*,*d*,*s*,*x*,*p*F

Если очередной аргумент является рациональной дробью, он печатается как десятичная дробь, смещенная на *s* цифр влево (по умолчанию 0) с помощью *d* цифр (по умолчанию сколько потребуется) после десятичного разделителя. Число может округляться, но направление округления может задаваться реализацией. Если задано *w*, число будет сдвинуто влево на столько *p* (по умолчанию #\Space), сколько потребуется, чтобы полное количество знаков было равно *w*. (Если само представление числа содержит больше знаков, чем *w*, и задан *x*, он будет напечатан.) Если опущены *w* и *d*, *s* игнорируется. С @ знак печатается даже для положительных чисел.

~*w*,*d*,*e*,*s*,*x*,*p*,*m*E

То же, что и ~F, но если первый аргумент рациональная дробь, она печатается в экспоненциальной записи с *s* (по умолчанию 1) знаками перед десятичным разделителем, *d* (по умолчанию сколько потребуется) знаков после него и *e* (по умолчанию сколько потребуется) знаков в экспоненте. Если задан *m*, он заменит знак экспоненты.

~*w*,*d*,*e*,*s*,*x*,*p*,*m*G

Если очередной аргумент является рациональной дробью, он будет отображен с помощью ~F или ~E в зависимости от числа.

~*d*,*n*,*w*,*p*$

Используется для печати денежных сумм. Если очередной аргумент является рациональной дробью, он будет напечатан в десятичном представлении с по меньшей мере *n* (по умолчанию 1) цифр перед десятичным разделителем и *d* цифрами (по умолчанию 2) после него. Будет напечатано по меньшей мере *w* (по умолчанию *w*) знаков; если потребуется, то число будет смещено влево на *p* (по умолчанию #\Space). С @ знак будет напечатан даже перед положительными числами. С : знак будет напечатан перед смещением.

Подобным образом действует *pprint-newline* с аргументами, зависящими от модификаторов: отсутствие соответствует :linear, @ соответствует :miser, : соответствует :fill, а :@ соответствует :mandatory.

~<*prefix*~;*body*~;*suffix*~:>

Действует подобно pprint-logial-block, требуя, чтобы очередной аргумент был списком, *prefix* и *suffix* аналогичны :prefix (или :per-line-prefix, если за ним следует ~@) и :suffix, а *body* играет роль выражений, составляющих тело. Оно может быть любой форматирующей строкой, а аргументы для нее получаются из списка аргументов, как с помощью pprint-pop. Внутри тела ~^ соответствует

`pprint-exit-if-list-exhausted`. Если заданы лишь два из трех параметров: *prefix*, *body* и *suffix*, значением *suffix* по умолчанию считается `""`; если лишь один из трех, *prefix* также считается `""`. С модификатором `:` *prefix* и *suffix* имеют умолчальные значения `"("` и `")"` соответственно. С `@` список оставшихся аргументов становится аргументом логическому блоку. Если вся директива заканчивается `~:@>`, то новая условная строка `:fill` добавляется после каждой группы пробелов в *body*.

`~`*n*`I`

Эквивалент `(pprint-indent :block` *n*`)`. С `:` аналогично `(pprint-indent :current` *n*`)`.

`~/name/`

Вызывает функцию с именем *name* и, по меньшей мере, четырьмя аргументами: поток; очередной аргумент; два значения, истинных когда используется `:` и `@` соответственно, а также любыми другими параметрами, переданными директиве.

`~`*m*`,`*n*`T`

Печатает достаточное количество пробелов, чтобы переместить курсор в колонку *m* (по умолчанию `1`), а если данная колонка уже пройдена, то в ближайшую колонку с несколькими интервалами n (по умолчанию `1`) после `m`. С `@` печатает *m* пробелов, затем достаточное количество пробелов, кратное *n*. С `:` является эквивалентом `(pprint-tab :section` *m n*`)`. С `@` является эквивалентом `(pprint-tab :section-relative` *m n*`)`.

`~`*w*`,`*n*`,`*m*`,`*p*`<`*text0*`~;...~;`*textn*`~>`

Отображает знаки, полученные смещением значений *text* на поле шириной *w* знаков, с по меньшей мере *m* (по умолчанию `0`) дополнительными элементами *p* (по умолчанию `#\Space`), при необходимости вставляемыми между *text*. Если ширина вывода с учетом минимального отступа больше w, ограничение увеличивается на число, кратное *n*. С `:` отступ будет также пред текстом; с `@` после него; `~^` завершает обработку директивы.

Если text0 следует за `~`*a*`,`*b*, а не `~;`, знаки, полученные из *text0*, будут выводиться, если оставшихся знаков больше, чем *b* (по умолчанию длина строки потока или `72`) с учетом *a* (по умолчанию *0*) зарезервированных знаков.

`~`*n*`*`

Игнорирует следующие *n* (по умолчанию 0) аргументов. С `:` сохраняет n аргументов. С `@` делает *n*-ный аргумент текущим.

`~`*i*`{`*text*`~}`

Действует подобно повторным вызовам `format` с *text* в качестве форматирующей строки и параметрами, взятыми из очередного аргумента, являющегося списком. Выполняется до тех пор, пока не закончатся аргументы, при достижении *n*-ного аргумента (по умолчанию без ограничений) начинает сначала. Если директива завершается `~:}` вместо `~}`, будет сделан по крайней мере один вызов `format`, пока *n* не будет равным 0. Если *text* не задан, очередной аргумент (должен быть строкой) используется вместо него. `~^` завершает обработку директивы. С `:` аргумент должен быть списком списков, а их элементы будут аргументами последовательных вызовов `format`. С `@` вместо очередного аргумента будет использоваться список оставшихся аргументов.

`~?`

Эквивалент вызову `format` с очередным аргументом, являющимся строкой, и списком из последующих аргументов. С `@` очередной аргумент будет использоваться как строка форматирования, но сооветствующие параметры будут получаться из аргументов основного вызова `format`.

`~(`*text*`~)`

Печатает *text*, преобразуя регистр в зависимости от модификатора: с `:` первая буква каждого слова преобразуется к верхнему регистру; с `@` помимо этого все остальные буквы преобразуются к нижнему регистру; с `:@` все буквы преобразуются к верхнему регистру.

`~P`

Если первый аргумент равен 1, не печатает ничего, иначе печатает s, с : предварительно сохраняет первый аргумент. С @, если первый аргумент равен 1, печатает y, иначе ies; с :@ предварительно сохраняет один аргумент.

~*a,b,c*^

Завершает директиву format (или любую другую, если используется внутри нее) при следующих условиях: если не заданы никакие префиксные параметры; если задан один и это ноль; если заданы два равных; если заданы три, и выполняется (<= *a b c*).

Если за ~ следует перевод строки, то она и все следующие за ним пробелы игнорируются. С @ игнорируется перевод строки, но не пробелы.

- (formatter *string*) макрос

Возвращает функцию, принимающую поток и ряд других параметров и применяет format к потоку, форматирующей строке и другим параметрам, возвращая лишние параметры.

- (pprint *object* &optional *stream*) функция

Действует подобно print, но пытается расставить отступы более аккуратно, а также не печатает пустых клеток на конце.

- (pprint-dispatch *object* &optional *pprint-dispatch*) функция

Возвращает наиболее приоритетную функцию в *pprint-dispatch*, для которой *object* имеет ассоциированный тип. Если *pprint-dispatch* не предоставляется, используется текущее значение *print-pprint-dispatch*, если передан nil, используется исходное значение. Если ни один тип в таблице диспетчеризации не соответствует *object*, возвращается функция, печатающая его через print-object. Второе возвращаемое значение истинно, когда первое было получено из таблицы диспетчеризации.

- (pprint-exit-if-list-exhausted) функция

Употребляется вместе с pprint-logical-block. Завершает блок, если не больше нечего печатать, иначе возвращает nil.

- (pprint-fill *stream object* &optional *colon at*) функция

Печатает объект в поток, но делает это иначе, если это список, и значение *print-pretty* истинно. печатает столько элементов списка, сколько способно уместиться в строке, окруженной скобками в случае истинности *colon* (умолчание). Аргумент *at* игнорируется. Возвращается nil.

- (pprint-indent *keyword r* &optional *stream*) функция

Если *stream* создан с помощью pprint-logical-block, и значение *print-pretty* истинно, устанавливает отступ для текущего логического блока. Если задан ключ :current, отступ устанавливается в текущую позицию, смещенную на r; если :block, но позиция отсчитывается от первого знака в текущем блоке, к которому добавляется r.

- (pprint-linear *stream object* &optional *colon at*) функция

Действует подобно pprint-fill, но умещает весь список на одной строке, или каждый элемент на отдельной строке.

- (pprint-logical-block (*symbol object* &key *prefix per-line-prefix suffix* макрос

declaration expression**)

Вычисляет выражения, связав *symbol* с новым потоком (действующим только внутри выражения pprint-logical-block), который направляет вывод в исходное значение *symbol* (которое должно быть поток). Весь вывод, направляемый в новый поток, находится в связанном с ним логическом блоке. Выражения *expression* не должны содержать побочных эффектов в данном окружении.

Объект *object* должен быть списком, который будут печатать выражения *expression*; его элементы будут получаться по очереди вызовом pprint-pop, иначе он будет напечатан с помощью *write*. При печати списка его разделяемые и вложенные компоненты будут отображаться согласно *print-circle* и *print-level*.

Ключевой аргумент, если задан, должен вычисляться в строку; *prefix* будет напечатан перед логическим блоком, *suffix* после него, а *per-line-prefix* в начале каждой строки. Параметры *prefix* и *per-line-prefix* взаимоисключающие.

- `(pprint-newline` *keyword* `&optional` *stream*`)` функция

Если поток создан с помощью `pprint-logical-block`, и значение `*print-pretty*` истинно, записывает новую строку в поток с учетом *keyword*: `:mandatory` означает всегда; `:linear`, если то результат аккуратной печати не умещается на строке; `:miser` для экономной записи; `:fill`, если предыдущая или следующая строки будут разорваны.

- `(pprint-pop)` макрос

Используется внутри `pprint-logical-block`. Если списком элементов, печатаемых в текущем логическом блоке, не пуст, возвращается следующий элемент. Если в списке остался атом (не `nil`), печатает его вслед за точкой и возвращает `nil`. Если значение `*print-length*` истинно и соответствует количеству уже напечатанных элементов, печатает многоточие и возвращает `nil`. Если значение `*print-circle*` истинно, и остаток списка является разделяемой структурой, печатает точку, за которой следует `#n#`, и возвращает `nil`.

- `(pprint-tab` *keyword i1 i2* `&optional` *stream*`)` функция

Если поток был создан с помощью `pprint-logical-block`, и значение `*print-pretty*` истинно, выводится табуляция, как с помощью директивы форматирования `~T` с префиксными параметрами i1 и i2. Возможные ключевые слова: `:line` соответствует `~T`, `:section` – `~:T`, `:line-relative` – `~@T`, `:section-relative` – `~:@T`.

- `(pprint-tabluar` *stream object* `&optional` *colon at tab*`)` функция

действует подобно `pprint-fill`, но печатает элементы списка так, что они выстраиваются в столбцы. Параметр *tab* (по умолчанию `16`) соответствует расстоянию между столбцами.

- `(princ` *object* `&optional` *stream*`)` функция

Отображает объект в поток в удобном для восприятия (по возможности) виде.

- `(princ-to-string` *object*`)` функция

Действует подобно `princ`, но направляет вывод в строку, которую затем возвращает.

- `(print` *object* `&optional` *stream*`)` функция

Действует подобно `prin1`, но окружает объект переносами строк.

- `(print-object` *object stream*`)` обобщенная функция

Вызывается системой при печати объекта в поток.

- `(print-not-readable-object` *condition*`)` функция

Возвращает объект, который не удалось напечатать в читаемом виде при возникновении исключения *condition*.

- `(print-unreadable-object` (*object stream* `&key` *type identity*) *expression**`)` макрос

Используется для отображения объектов в синтаксисе `#<...>`. Все аргументы вычисляются. Записывает в поток `#<`; затем, если *type* является типом, выводит метку типа для объекта; затем вычисляет выражения, которые должны отображать объект в поток; затем, если значение *identity* истинно, записывает идентификатор объекта; завершает записью `>`. Возвращает `nil`.

- `(prin1` *object* `&optional` *stream*`)` функция

Отображает объект в поток таким образом, чтобы он (по возможности) мог быть затем прочитан с помощью `read`.

- `(prin1-to-string` *object*`)` функция

Действует подобно `prin1`, но направляет вывод в строку, которую затем возвращает.

- `(set-pprint-dispatch` *type function* `&optional` *r pprint-dispatch*`)` функция

Если значение *function* истинно, добавляет элемент в *pprint-dispatch* (по умолчанию `*print-pprint-dispatch*`) с заданным типом, функцией и приоритетом *r* (по умолчанию `0`). Функция должна принимать два аргумента: поток и объект для печати. Если *function* `nil`, удаляет из таблицы все элементы с типом *type*. Возвращает `nil`.

- `(write` *object* `&key` *array base case circle escape gensym length level lines miser-width* функция

  pprint-dispatch pretty radix right-margin stream)

Записывает объект в поток, связывая каждую специальную переменную вида `*print-...*` со значением соответствующего ключевого параметра.

- `(write-to-string` *object* `&key ...)` функция

Действует по аналогии с *write*, но перенаправляет вывод в строку, которую затем возвращает.

Считыватель

- `(copy-readtable &optional` *from* `<`*to*`>)` функция

Если *to* – `nil`, возвращает копию таблицы чтения (по умолчанию `*readtable*`); если *to* – таблица чтения, возвращает ее после копирования в нее таблицы *from*.

- `(get-dispatch-macro-character` *char1 char2* `&optional` *readtable*`)` функция

Возвращает функцию (или *nil*, если ничего не найдено), вызываемую в *readtable*, когда *char1* следует за *char2*.

- `(get-macro-character` *char* `&optional` *readtable*`)` функция

Возвращает два значения: функцию (или `nil`, если ничего не найдено), вызываемую в *readtable*, когда во вводе встречается *char*, а также вторым значением истину, когда *char* был прочитан как часть имени символа.

- `(make-dispatch-macro-character` *char* `&optional` *nonterm readtable*`)` функция

Добавляет в *readtable* управляющий макросимвол. Если значение *nonterm* истинно, *char* ведет себя как нормальный знак в середине символа. Возвращает `t`.

- `(read &optional` *stream eof-error eof-value recursive*`)` функция

Считывает один Лисп-объект из потока, обрабатывает и возвращает его. Если достигнут конец файла, либо сигнализируется ошибка, либо возвращается *eof-value*, в зависимости от значения *eof-error* (по умолчанию истина). При вызове `read` из другой считывающей функции будет передан истинный параметр *recursive*.

- `(read-delimited-list` *char* `&optional` *stream recursive*`)` функция

Действует подобно `read`, но продолжает обработку объектов до тех пор, пока не встретит *char*, после чего немедленно возвращает список обработанных объектов. Сигнализирует ошибку, если до конца файла не остается ни одного знака *char*.

- `(read-from-string` *string* `&optional` *eof-error eof-value* `&key` *start end preserve-whitespace*`)` функция

Действует по отношению к строке, как `read` к потоку. Возвращает два значения: обработанный объект и положение (индексация с нуля) первого непрочитанного знака. Если значение *preserve-whitespace* истинно, действует как `read-preserving-whitespace` вместо `read`.

- `(read-preserving-whitespace &optional` *stream eof-error eof-value recursive*`)` функция

Действует подобно `read`, но оставляет в потоке завершающие пробелы.

- `(readtable-case` *readtable*`)` функция

Возвращает один из вариантов: `:upcase`, `:downcase`, `:preserve` или `:invert`, в зависимости от способа работы *readtable* с регистрами. Может быть первым аргументом `setf`.

- `(readtablep` *object*`)` функция

Возвращает истину, когда объект является таблицей чтения.

- `(set-dispatch-macro-character` *char1 char2 function* `&optional` *readtable*`)` функция

Добавляет в *readtable* элемент, сообщающей, что *function* будет вызываться при последовательном считывании *char1* и *char2* (который преобразуется к верхнему регистру и не может быть цифрой); *function* должна быть функцией трех аргументов: входного потока, *char1* и *char2*, и возвращать объект, прочитанный из потока, либо не возвращать ничего. Возвращает `t`.

- `(set-macro-character` *char function* `&optional` *nonterm readtable* `)` функция

Добавляет в *readtable* элемент, сообщающий, что *function* будет вызываться при считывании *char*; *function* должна быть функцией двух аргументов: входного потока и *char*, и возвращать либо считанный объект, либо ничего. Если значение *nonterm* истинно, *char* ведет себя как нормальный знак, считанный в середине символа. Возвращает `t`.

- `(set-syntax-from-char` *to-char from-char to-readtable from-readtable* `)` функция

Дает знаку *to-char* в таблице *to-readtable* (по умолчанию `*readtable*`) синтаксические свойства знака *from-char* из таблицы *from-readtable* (по умолчанию стандартная таблица). Возвращает `t`.

- `(with-standard-io-syntax` *expression*`*)` макрос

Вычисляет выражения, связав все специальные переменные, управляющие чтением и печатью (т.е. с именами вида `*read-` или `*print-`), с их исходными значениями.

Сборка системы

- `(compile-file` *path* `&key` *output-file verbose print external-format*`)` функция

Компилирует содержимое файла, на который указывает *path*, и записывает результат в файл *output-file*. Если *verbose* – истина, факт компиляции записывается в `*standard-output*`. Если *print* – истина, информация о toplevel-выражениях выводится в `*standard-output*`. Единственным (по стандарту) форматом *external-format* является `:default`. После завершения компиляции устанавливаются исходные значения `*readtable*` и `*package*`. Возвращает три значения: имя выходного файла (или `nil`, если файл не смог записаться); значение, истинное, когда в результате компиляции были получены ошибки или предупреждения; еще одно, истинное, когда были получены ошибки или предупреждения, отличные от стилевых предупреждений.

- `(compile-file-pathname` *path* `&key` *output-file*`)` функция

Возвращает имя выходного файла, который создаст *compile-file* с теми же аргументами. Допускает использование других ключей.

- `(load` *path* `&key` *verbose print if-does-not-exist external-format*`)` функция

Загружает файл, на который указывает *path*. Если это файл с исходным кодом, то выполняется последовательное вычисление всех toplevel-выражений; аналогично для скомпилированного файла, но для соответствующих функций из него справедлив предикат `compiled-function-p`. Если параметр *verbose* истинен, информация о загрузке файла анонсируется в `*standard-output*`. Если параметр *print* истинен, процесс загрузки также выводится в `*standard-output*`. Если параметр *if-does-not-exist* истинен (по умолчанию), в случае отсутствия файла сигнализируется ошибка; в противном случае возвращается `nil`. Единственным (по стандарту) форматом *external-format* является `:default`. Возвращает `t`.

- `(provide` *name*`)` [функция]

Добавляет строку *name* (или имя соотв. символа) в `*modules*`.

- (`require` *name* `&optional` *paths*`)` [функция]

Если строка *name* (или имя соотв. символа) не находится в `*modules*`, пытается загрузить файл, содержащий соответствующий модуль. Параметр *path*, если задается, должен быть списком путей, потоков или строк, при этом будут загружаться указываемые файлы.

- `(with-compilation-unit ([:override` *val*`]) ` *expression**`)` макрос

Вычисляет выражения *expression*. Вывод предупреждений, задерживаемых до конца компиляции, будет задержан до завершения вычисления последнего выражения. В случае вложенного вызова эффект будет получен только при сообщении истинного значения *val*.

Окружение

- `(apropos` *name* `&optional` *package*`)` функция

Выводит информацию о каждом интернированном символе, чье имя содержит подстроку *name* (или имя *name*, если это символ). Если задан пакет, поиск осуществляется только по нему. Не возвращает значений.

- `(apropos-list` *name* `&optional` *package*`)` функция

Действует подобно `apropos`, но не выводит информацию, а возвращает список символов.

- `(decode-universal-time` *i* `&optional` *time-zone*`)` функция

Интерпретирует i как количество секунд с момента 1 января 1990 года, 00:00:00 (GMT), возвращая девять значений: секунды, минуты, час, день, месяц (январю соответствует `1`), год, день недели (понедельнику соответствует `0`), действует ли летнее время и рациональное число, соответствующее смещению временной зоны относительно GMT. Параметр *time-zone* должен быть рациональным числом в интервале от `-24` до `24` включительно; если он задается, летнее время не учитывается.

- `(describe` *object* `&optional` *stream*`)` функция

Записывает описание объекта в поток. Не возвращает значений.

- `(describe-object` *object* `&optional` *stream*`)` обобщенная функция

Вызывается из `describe`.

- `(disassemble` *fn*`)` функция

Выводит объектный код, сгенерированный для *fn*, которая может быть функцией, именем функции или лямбда-выражением.

- `(documentation` *object symbol*`)` обобщенная функция

Возвращает документацию для *object* с ключом *symbol* или `nil`, если ничего не найдено. Может устанавливаться через `setf`.

- `(dribble &optional` *path*`)` функция

Если задан путь *path*, то записывает в соответствующий файл всю историю чтения/записи данной Лисп-сессии. Если вызывается без аргументов, то закрывает данный файл.

- `(ed &optional` *arg*`)` функция

Вызывает редактор (если имеется хотя бы один). Если *arg* – путь или строка, будет редактироваться соответствующий файл. Если это имя функции, редактироваться будет ее определение.

- `(encode-universal-time` *second minute hour date month year* `&optional` *time-zone*`)` функция

Функция, обратная `decode-unversal-time`.

- `(get-decoded-time)` функция

Эквивалент `(decode-universal-time (get-universal-time))`.

- `(get-internal-real-time)` функция

Возвращает текущее системное время в тактах системных часов, которые составляют `internal-time-units-per-second` долю секунды.

- `(get-internal-run-time)` функция

Действует подобно `get-internal-real-time`, но возвращает значение, соответствующее чистому времени работы Лисп-процесса в системных тактах.

- `(get-universal-time)` функция

Возвращает текущее время как количество секунд с 1 января 1900 года, 00:00:00 (GMT).

- `(inspect` *object*`)` функция

Интерактивная версия `describe`, позволяющая перемещаться по составным объектам.

- `(lisp-implementation-type)` функция

Возвращает строку, характеризующую используемую реализацию, или `nil`.

- `(lisp-implementation-version)` функция

Возвращает строку, характеризующую версию используемой реализации, или `nil`.

- `(long-site-name)` функция

То же, что и `short-site-name`, но выводит больше информации.

- `(machine-instance)` функция

Возвращает строку, характеризующую конкретную машину, на которой выполняется Лисп-сессия, или `nil`.

- `(machine-type)` функция

Возвращает строку, характеризующую тип машины, на которой выполняется Лисп-сессия, или `nil`.

- `(machine-version)` функция

Возвращает строку, сообщающую версию машины, на которой выполняется Лисп-сессия, или `nil`.

- `(room &optional (`*arg* `:default))` функция

Выводит сообщение, описывающее текущее состояние памяти. Более лаконичный ответ будет получен с аргументом `nil`, более многословный – с `t`.

- `(short-site-name)` функция

Возвращает строку, характеризующую текущее физическое расположение, или `nil`.

- `(software-type)` функция

Возвращает строку, характеризующую тип нижележащего программного обеспечения, или `nil`.

- `(software-version)` функция

Возвращает строку, характеризующую версию нижележащего программного обеспечения, или `nil`.

- `(step expression)` макрос

Делает шаг в вычислении выражения *expression*, возвращая значение(я), которые будут получены.

- `(time expression)` макрос

Вычисляет выражение *expression*, возвращая полученное значение(я), а также выводя в `*trace-output*` информацию о затраченном времени.

- `(trace` *fname**`)` макрос

Выводит информацию о вызовах функций с заданными именами в `*trace-output*`. Может не работать для функций, скомпилированных с `inline`-декларацией. Без аргументов выводит список функций, отслеживаемых на данный момент.

- `(untrace` *fname**`)` макрос

Отменяет вызов `trace`. Без аргументов отменяет трассировку всех отслеживаемых функций.

- `(user-homedir-pathname &optional` *host*`)` функция

Возвращает путь к домашней директории пользователя или `nil`, если на заданном хосте ее нет. Параметр *host* используется так же, как в `make-pathname`.

Константы и переменные

- `array-dimension-limit` константа

Положительный `fixnum`, на единицу превышает ограничение на количество элементов в одной размерности массива. Зависит от реализации, но не менее `1024`.

- `array-rank-limit` константа

Положительный `fixnum`, на единицу превышает максимальное количество размерностей в массиве. Зависит от реализации, но не менее `8`.

- `array-total-size-limit` константа

Положительный `fixnum`, на единицу превышает ограничение на количество суммарное количество элементов в массиве. Зависит от реализации, но не менее `1024`.

- `boole-1` ... `boole-xor` константы

Положительные целые числа, используемые как первый аргумент `boole`.

- `*break-on-signals*` переменная

Обобщение более старой `*break-on-warnings*`. Значение должно быть спецификатором типа. Если сигнализируется исключение заданного типа, вызывается отладчик. Исходно `nil`.

- `call-arguments-limit` константа

Положительное целое число, на единицу большее максимального количества аргументов в вызове функции. Зависит от реализации, но не менее `50`.

- `char-code-limit` константа

Положительное целое число, на единицу большее максимального значения, возвращаемого `char-code`. Зависит от реализации.

- `*compile-file-pathname*` переменная

В процессе вызова `compile-file` содержит путь, полученный из первого аргумента. Вне процесса компиляции `nil`.

- `*compile-file-truename*` переменная

Настоящее имя для `*compile-file-pathname*`.

- `*compile-print*` переменная

Используется как значение по умолчанию параметра `:print` в `compile-file`. Исходное значение зависит от реализации.

- `*compile-verbose*` переменная

Используется как значение по умолчанию параметра `:verbose` в `compile-file`. Исходное значение зависит от реализации.

- `*debug-io*` переменная

Поток, используемый для интерактивной отладки.

- `*debugger-hook*` переменная

Если не `nil`, должна быть функцией *f* двух аргументов. Непосредственно перед вызовом отладчика *f* вызывается с возникшим исключением и самой *f*. Если вызов *f* завершается нормально, управление передается в отладчик. В процессе вызова *f* `*debugger-hook*` будет установлен в `nil`.

- `*default-pathname-defaults*` переменная

Используется как значение по умолчанию, когда функциям типа `make-pathname` не передан аргумент `:defaults`.

- `short-float-epsilon` константа
- `single-float-epsilon` константа
- `double-float-epsilon` константа
- `long-float-epsilon` константа

Для каждого соответствующего типа определяет наименьшее положительное число того же формата, которое, при добавлении к нему `1.0` в том же формате, приводит к результату, отличному от `1.0`. Зависит от реализации.

- `short-float-negative-epsilon` константа
- `single-float-negative-epsilon` константа
- `double-float-negative-epsilon` константа
- `long-float-negative-epsilon` константа

Для каждого соответствующего типа определяет наименьшее положительное число того же формата, которое, при вычитании его из `1.0` в том же формате, приводит к результату, отличному от `1.0`. Зависит от реализации.

- `*error-output*` переменная

Поток, используемый для вывода сообщений об ошибках.

- `*features*` переменная

Зависящий от реализации список символов, представляющих возможности, поддерживаемые данной реализацией. Эти символы используются как тестовые компоненты в `#+` и `#-`.

- `*gensym-counter*` переменная

Неотрицательное целое число, используемое `gensym` для создания имен символов. Исходное значение зависит от реализации.

- `internal-time-units-per-second` константа

Если разница между двумя вызовами `get-internal-run-time` делится на данное целое число, результат будет представлен в виде количества секунд системного времени между ними.

- `lambda-list-keywords` константа

Список ключей в списках параметров (т.е. `&optional`, `&rest` и т.д.), поддерживаемых реализацией.

- `lambda-parameters-limit` константа

Положительное целое число, на единицу большее максимального количества переменных в списке параметров. Зависит от реализации, но не меньше `50`.

- `least-negative-short-float` константа
- `least-negative-single-float` константа
- `least-negative-double-float` константа
- `least-negative-long-float` константа

Наименьшее по модулю отрицательное число с плавающей точкой каждого типа, поддерживаемое реализацией.

- `least-negative-normalized-short-float` константа
- `least-negative-normalized-single-float` константа
- `least-negative-normalized-double-float` константа
- `least-negative-normalized-long-float` константа

Наименьшее по модулю нормализованное отрицательное число с плавающей точкой каждого типа, поддерживаемое реализацией.

- `least-positive-short-float` константа
- `least-positive-single-float` константа
- `least-positive-double-float` константа
- `least-positive-long-float` константа

Наименьшее по модулю положительное число с плавающей точкой каждого типа, поддерживаемое реализацией.

- `least-positive-normalized-short-float` константа
- `least-positive-normalized-single-float` константа
- `least-positive-normalized-double-float` константа

- `least-positive-normalized-long-float` константа

Наименьшее по модулю нормализованное положительное число с плавающей точкой каждого типа, поддерживаемое реализацией.

- `*load-pathname*` переменная

В процессе вычисления вызова `load` связывается с путем, полученным из первого аргумента. Вне процесса загрузки `nil`.

- `*load-print*` переменная

Используется как значение по умолчанию параметра `:print` в вызове `load`. Исходное значение зависит от реализации.

- `*load-truename*` переменная

Настоящее имя для `*load-pathname*`.

- `*load-verbose*` переменная

Используется как значение по умолчанию параметра `:verbose` в вызове `load`. Исходное значение зависит от реализации.

- `*macroexpand-hook*` переменная

Функция трех аргументов: раскрывающая функция, макровызов и окружение; вызывается из `macroexpand-1` для генерации раскрытий макросов. Исходное значение эквивалентно `funcall` или имени соответствующей функции.

- `*modules*` переменная

Список строк, полученных в результате вызовов `provide`.

- `most-negative-fixnum` константа

Наименьшее возможное значение типа `fixnum`, поддерживаемое реализацией.

- `most-negative-short-float` константа
- `most-negative-single-float` константа
- `most-negative-double-float` константа
- `most-negative-long-float` константа

Набольшее по модулю отрицательное число с плавающей точкой каждого типа, поддерживаемое реализацией.

- `most-positive-fixnum` константа

Наименьшее возможное значение типа `fixnum`, поддерживаемое реализацией.

- `most-positive-short-float` константа
- `most-positive-single-float` константа
- `most-positive-double-float` константа
- `most-positive-long-float` константа

Набольшее по модулю положительное число с плавающей точкой каждого типа, поддерживаемое реализацией.

- `multiple-values-limit` константа

Положительное целое число, на единицу большее максимального количества возвращаемых значений. Зависит от реализации, но не менее `20`.

- `nil` константа

Вычисляется сам в себя. В равной степени представляет ложь и пустой список.

- `*package*` переменная

Текущий пакет. Исходно `common-lisp-user`.

- `pi` константа

Приближение числа π в формате `long-float`.

- `*print-array*` переменная

Если истинна, массивы всегда печатаются в читаемом формате. Исходное значение зависит от реализации.

- `*print-base*` переменная

Целое число между 2 и 36 включительно. Определяет основание системы счисления, в которой печатаются числа. Исходное значение 10 (десятичный формат).

- *print-case* переменная

Управляет печатью символов, имена которых находятся в верхнем регистре. Возможны три значения: :upcase (исходно), :downcase и :capitalize (печатает символы так, как после применения string-capitalize к их именам).

- *print-circle* переменная

Если истинна, разделяемые структуры будут отображаться с помощью макросов чтения #n= и #n#. Исходно nil.

- *print-escape* переменная

Если nil, все будет печататься как с помощью princ. Исходно t.

- *print-gensym* переменная

Если истинна, перед неинтернированными символами будет печататься #:. Исходно t.

- *print-length* переменная

Либо nil (исходно), либо положительное целое число. Число определяет максимальное количество элементов, отображаемых для одного объекта (лишние будут скрыты). Если nil, ограничения нет.

- *print-level* переменная

Либо nil (исходно), либо положительное целое число. Число определяет максимальный уровень вложенности отображаемых объектов (остальное будет скрыто). Если nil, ограничения нет.

- *print-lines* переменная

Либо nil (исходно), либо положительное целое число. Число определяет максимальное количество строк, используемых для отображения объекта при аккуратной печати (лишние будут скрыты). Если nil, ограничения нет.

- *print-miser-width* переменная

Либо nil, либо положительное целое число. Число задает компактный стиль аккуратной печати, используемый, если необходимо ужать представление объекта. Исходное значение зависит от реализации.

- *print-pprint-dispatch* переменная

Либо nil, либо таблица управления аккуратной печатью. Исходно установлена таблица, выполняющая аккуратную печать согласно обычным правилам.

- *print-pretty* переменная

Когда истинна, для отображения объектов используется аккуратная печать. Исходное значение зависит от реализации.

- *print-radix* переменная

Когда истинна, задает основание системы счисления печатаемых чисел.

- *print-right-margin* переменная

Либо nil (исходно), либо положительное целое число, представляющее величину правого поля, на котором ничего не будет печататься. Если nil, то задается самим потоком.

- *query-io* переменная

Поток, используемый для двустороннего общения с пользователем.

- *random-state* переменная

Объект, представляющий состояние генератора случайных чисел.

- *read-base* переменная

Целое число от 2 до 36 включительно. Определяет основание системы счисления, в которой будут печататься числа. Исходно 10 (десятичный формат).

- *read-default-float-format* переменная

Показывает используемый по умолчанию формат, используемый `read` для чтения чисел с плавающей запятой. Должна быть спецификатором типа чисел с плавающей запятой. Исходно `single-float`.

- `*read-eval*` переменная

Если `nil`, `#.` сигнализирует ошибку. Исходно `t`.

- `*read-suppress*` переменная

Если истинна, `read` будет более толерантной к синтаксическим различиям. Исходно `nil`.

- `*readtable*` переменная

Текущая таблица чтения. Исходная таблица задает стандартный синтаксис Common Lisp.

- `*standard-input*` переменная

Входной поток по умолчанию.

- `*standard-output*` переменная

Выходной поток по умолчанию.

- `t` константа

Вычисляется сама в себя. Представляет истину.

- `*terminal-io*` переменная

Поток, используемый консолью (если имеется).

- `*trace-output*` переменная

Поток, в который отображается результат трассировки.

- `*` `**` `***` переменные

Содержат значения соответственно трех последних выражений, введенных в toplevel.

- `+` `++` `+++` переменные

Содержат соответственно три последних выражения, введенных в toplevel.

- `-` переменная

В процессе вычисления выражения в toplevel содержит само выражение.

- `/` `//` `///` переменные

Содержат список значений, которые вернули соответственно три последних выражений, введенных в toplevel.

Спецификаторы типов

Спецификаторы типов могут быть простыми или составными. Простой спецификатор является символом, соответствующим имени типа (например, `integer`). Составной спецификатор является списком из символа и набора аргументов. В этом разделе будут рассмотрены варианты составных спецификаторов.

- `(and` *type** `)`

Соответствует пересечению типов.

- `(array` *type dimensions*`)`
- `(simple-array` *type dimensions*`)`

Соответствуют множеству массивов с типом *type* и размерностями, соответствующими *dimensions*. Если *dimensions* – неотрицательное целое число, оно указывает на количество размерностей; если это список, то величину каждой размерности (как в вызове `make-array`).

Знак `*`, встречающийся на месте типа или одной из размерностей, означает отсутствие соответствующих ограничений.

- `(base-string` *i*`)`
- `(simple-base-string` *i*`)`

Эквиваленты `(vector base-character i)` и `(simple-array base-character (i))`, соответственно.

- `(bit-vector i)`
- `(simple-bit-vector i)`

Эквиваленты `(array bit (i))` и `(simple-array bit (i))`, соответственно.

- `(complex type)`

Соответствует множеству комплексных чисел, мнимые и действительные части которых принадлежат типу *type*.

- `(cons type1 type2)`

Соответствует множеству ячеек, чей `car` принадлежит типу *type1*, а `cdr` – *type2*. Знак `*` в одной из позиций соответствует `t`.

- `(eql object)`

Соответствует множеству из одного элемента: *object*.

- `(float min max)`
- `(short-float min max)`
- `(single-float min max)`
- `(double-float min max)`
- `(long-float min max)`

Указывают на множество чисел с плавающей запятой определенного типа со значениями между *min* и *max*, где *min* и *max* – либо *f* (включая число *f*), либо `(f)` (исключая число *f*), где *f* – число соответствующего типа, либо `*`, что соответствует отсутствию ограничения.

- `(function parameters type)`

Используется только в декларациях. Соответствует множеству функций, чьи аргументы принадлежат заданным типам, возвращающих значение(я) типа *type*. Список *parameters* соответствует списку спецификаторов типов для аргументов функции (ключевые параметры сообщаются как `(key type)`). Спецификатор, следующий за `&rest`, соответствуют типу последующих аргументов, а не типу самого параметра, который всегда список. (Также изучите спецификатор `values`.)

- `(integer min max)`

Аналог `float` для целых чисел.

- `(member object*)`

Соответствует множеству объектов *object*.

- `(mod i)`

Соответствует множеству целых чисел, меньших *i*.

- `(not type)`

Соответствует дополнению до множества *type*.

- `(or type*)`

Соответствует объединению типов.

- `(rational min max)`

Аналог `float` для рациональных чисел.

- `(real min max)`

Аналог `float` для действительных чисел.

- `(satisfies symbol)`

Соответствует множеству объектов, удовлетворяющих функции с одним аргументом и именем *symbol*.

- `(signed-byte i)`

Соответствует множеству целых чисел между $-2^{(i-1)}$ и $2^{(i-1)}-1$ включительно. Эквивалент `integer`, если *i* это `*`.

- `(string i)`

- `(simple-string i)`

Соответствует множеству строк и простых строк, соответственно, длиной *i*.

- `(unsigned-byte i)`

Соответствует множеству неотрицательных целых чисел, меньших `2`*i*. Если *i* это `*`, эквивалент `(integer 0 *)`.

- `(values . parameters)`Ё050in

Используется только в спецификаторах `function` и выражениях `the`. Соответствует множеству наборов значений, которые могут передаваться в `multiple-value-call` с функцией типа (`function parameters`).

- `(vector type i)`
- `(simple-vector i)`

Эквивалент `(array type (`*i*`))` и `(simple-array t (`*i*`))`, соответственно. Необходимо помнить, что простой вектор – не просто простой одномерный массив. Простой вектор может также хранить объекты любых типов.

Макросы чтения

Макросами, состоящими из одного знака. являются `(`, `)`, `'`, `;` и `` ` ``. Все предопределенные управляемые макросы чтения имеют управляющий знак `#`:

`#\`*c*	Соответствует знаку *c*.
`#'`*f*	Эквивалент `(function` *f*`)`.
`#(...)`	Соответствует простому вектору.
`#n(...)`	Соответствует простому вектору из *n* элементов. Если задано меньше, недостающие заполняются значением последнего.
`#*`*bbb*	Соответствует простому бит-вектору.
`#`*n*`*`*bbb*	Соответствует простому бит-вектору из *n* элементов. Если задано меньше, недостающие заполняются значением последнего.
`#:`*sym*	Создает новый неинтернированный символ с именем *sym*.
`#.`*expr*	Создает значение *expr* на момент считывания.
`#B`*ddd*	Двоичное число.
`#O`*ddd*	Восьмеричное число
`#X`*ddd*	Шестнадцатеричное число
`#`*n*`R`*ddd*	Число с основанием *n*, которое должно быть десятичным целым числом от 2 до 36 включительно.
`#C(`*a b*`)`	Соответствует комплексному числу *a*+*bi*.
`#`*n*`A`*expr*	Соответствует многомерному массиву, созданному со значением `'`*expr* параметра `:initial-contents` в `make-array`.
`#S(`*sym ...*`)`	Создает структуру типа *sym*, заполняя аргументы в соответствии с заданными значениями, присваивая остальным значения так же, как с помощью соответствующего конструктора.

#P*expr*	Эквивалент (parse-namestring '*expr*).
#*n*=*expr*	Эквивалент *expr*, но создается циклическая структура со ссылкой на элемент с меткой *n*.
#*n*#	Создает объект, являющийся меткой *n* в циклической структуре.
#+*test expr*	Если *test* пройден, то эквивалентно *expr*, иначе просто пробел.
#-*test expr*	Если *test* не пройден, то эквивалент *expr*, иначе просто пробел.
#|...|#	Многострочный комментарий. Игнорируется считывателем.
#<	Вызывает ошибку.

Обратную кавычку легко понять, если рассмотреть значения, возвращаемые выражением с ней.° Ё118in Чтобы вычислить такое выражение, удалим обратную кавычку и все соответствующей ей запятые, заменяя выражение после каждой кавычки на соответствующее ему значение. Вычисление выражения, начинающегося с запятой (вне обратной кавычки) приводит к ошибке.

Запятая соответствует обратной кавычке, когда между ними имеется одинаковое количество запятых и обратных кавычек (a находится между b и c, если a находится спереди выражения, содержащего c; это значит, что в корректном выражении последняя обратная кавычка совпадает с наиболее глубоколежащей запятой).

Ё061inПредположим, что x вычисляется в a, которая вычисляется в 1; y вычисляется в b, которая вычисляется в 2. Чтобы вычислить выражение:

```
``(w ,x ,,y)
```

удалим первую кавычку и вычислим все, что следует за соответствующими ей запятыми. Ей соответствует лишь крайняя справа запятая. Если мы удалим ее и заменим оставшееся выражение его значением, то получим:

```
`(w ,x ,b)
```

Легко увидеть, что это выражение раскрывается в:

```
(w a 2)
```

Знак запятая-собака (,@) ведет себя аналогично запятой, но должен находиться в списке, а соответствующее выражение должно вычисляться в список. Получаемый список сшивается со списком, содержащим этот знак.

```
``(w ,x ,,@(list 'a 'b))
```

вычисляется в

```
`(w ,x ,a ,b)
```

Знак запятая-точка (,.) аналогичен запятой с собакой, но действует деструктивно.

Замечания

Этот раздел может считаться библиографией. Все книги и статьи, перечисленные в нем, рекомендуются к прочтению.

Стр. Ё063out. Steele, Guy L., Jr., with Scott E. Fahlman, Richard P. Gabriel, David A. Moon, Daniel L. Weinreb, Daniel G. Bobrow, Linda G. DeMichiel, Sonya E. Keene, Gregor Kiczales, Crispin Perdue, Kent M. Pitman, Richard C. Waters, and Jon L White. Common Lisp: the Language, 2nd Edition. Digital Press, Bedford (MA), 1990.

Стр. Ё064out. McCarthy, John. «Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I». CACM, 3:4 (April 1960), pp. 184-195.

McCarthy, John. «History of Lisp».

In Wexelblat, Richard L. (Ed.) «History of Programming Languages». Academic Press, New York, 1981, pp. 173-197.

На момент издания оригинала[1] были доступны на *http://www-formal.stanford.edu/jmc/*.

Стр. Ё065out. Brooks, Frederick P. «The Mythical Man-Month»[2]. Addison-Wesley, Reading (MA), 1975, p. 16.

Быстрое прототипирование – не просто способ написания качественных программ в краткие сроки. Это способ написания программ, которые без него не будут написаны вовсе.

Даже очень амбициозные люди начинают с небольших дел. Проще всего вначале заняться несложной (или, по крайней мере, кажущейся несложной) задачей, которую вы в состоянии сделать в одиночку. Как раз поэтому многие большие дела происходили из малых начинаний. Быстрое прототипирование дает возможность начать с малого.

Стр. Ё066out. Там же.

Стр. Ё067out. Murray, Peter and Linda. «The Art of the Renaissance». Thames and Hudson, London, 1963, p. 85.

Стр. Ё068out. Janson, W J. «History of Art», 3rd Edition. Abrams, New York, 1986, p. 374.

Аналогия применима, разумеется, только к станковой живописи и, позже, живописи по холсту. Настенная живопись по-прежнему выполнялась в виде фресок. Я также не пытаюсь утверждать, что стили живописи полностью следовали за технологическим прогрессом, но противоположное мнение более жизнеспособно.

Стр. Ё069out. Имена car и cdr происходят из внутреннего представления списков в первой реализации Лиспа: car соответствовал «Contents of Address part of Register», то есть, содержимому адресной части регистра, а cdr – «Contents of the Decrement part of the Register», то есть, декрементной части регистра.

Стр. Ё070out. Читатели, столкнувшиеся с серьезными затруднениями в понимании рекурсии, могут обратиться к одной из следующих книг:

Touretzky, David S. Common Lisp: «A Gentle Introduction to Symbolic Computation». Benjamin/Cummings, Redwood City (CA), 1990, Chapter 8.

Friedman, Daniel P., and Matthias Felleisen. «The Little Lisper». MIT Press, Cambridge, 1987.

Стр. Ё071out. В ANSI Common Lisp имеется также макрос `lambda`, позволяющий писать `(lambda (x) x)` вместо `#'(lambda (x) x)`. Он редуцирует симметрию между лямбда-выражениями и символьными именами функций (где вы по-прежнему можете использовать `#'`), что привносит особую элегантность.

Стр. Ё072out Gabriel, Richard P. Lisp: «Good News, Bad News, How to Win Big». AI Expert, June 1991, p. 34.

---

[1] А также на момент подготовки перевода. – *Прим. перев.*

[2] В сети имеется русский перевод. – *Прим. перев.*

Стр. Ё073out. Следует остерегаться еще одного момента при использовании `sort`: она не гарантирует сохранение порядка следования элементов, равных с точки зрения функции сравнения. Например, при сортировке `(2 1 1.0)` с помощью `<`, корректная реализация может вернуть, как `(1 1.0 2)`, так и `(1.0 1 2)`. Чтобы сохранить как можно больше исходных позиций, используйте более медленную функцию `stable-sort` (также деструктивную), которая возвращает только первое значение.

Стр. Ё074out О пользе комментариев было сказано достаточно, но практически ничего – о затратах на них. Но комментарии имеют свою цену. Хороший код, как хорошая проза, требует постоянного пересмотра. Для этого код должен быть компактным и податливым. Междустрочные комментарии делают код менее гибким и более расплывчатым, поэтому задерживают его эволюцию.

Стр. Ё075out. Несмотря на то, что подавляющее число реализаций используют набор знаков ASCII, Common Lisp гарантирует лишь следующее их упорядочение: 26 знаков нижнего регистра, расположенных по возрастанию в алфавитном порядке, их аналоги в верхнем регистре, а также цифры от 0 до 9.

Стр. Ё076out. Стандартный способ реализации очередей с приоритетом – структура, называемая *кучей*. См. Sedgewick, Robert. «Algorithms». Addison-Wesley, Reading (MA), 1988.

Стр. Ё077out. Определение `progn` похоже на правило вычисления вызовов функций в Common Lisp (см. стр. Ё051out). Хотя `progn` и является специальным оператором, мы можем определить похожую функцию:

```
(defun our-progn (&rest args)
  (car (last args)))
```

Она будет совершенно неэффективна, но функционально эквивалентна реальному `progn` (если последний аргумент возвращает только одно значение).

Стр. Ё078out. Аналогия с лямбда-выражениями нарушается, если именами переменных являются символы, имеющие в списке параметров особое значение. Например, выражение:

```
(let ((&key 1) (&optional 2)))
```

корректно, а вот соответствующее ему лямбда-выражение – уже нет:

```
((lambda (&key &optional)) 1 2)
```

При построении аналогии между `do` и `labels` вы столкнетесь с той же проблемой. Выражаю благодарность Дэвиду Кужнику (David Kuzhnik) за указание на нее.

Стр. Ё079out. Guy L., Jr., and Richard P. Gabriel. «The Evolution of Lisp». ACM SIGPLAN Notices 28:3 (March 1993).

Пример, процитированный в этом отрывке, переведен на Common Lisp со Scheme.

Стр. Ё080out. Чтобы сделать вывод времени более читаемым, необходимо убедиться, что минуты и секунды представляются в виде пары цифр:

```
(defun get-time-string ()
  (multiple-value-bind (s m h) (get-decoded-time)
    (format nil "~A:~2,,,'0@A:~2,,,'0@A" h m s)))
```

Стр. Ё081out. В своем письме 18 марта (по старому стилю) 1751 года Честерфилд писал:

> Хорошо известно, что юлианский календарь неточен, так как переоценивает продолжительность года на одиннадцать дней. Папа Григорий Тринадцатый исправил эту ошибку [в 1582]; его календарная реформа немедленно была принята в Католической Европе, а затем и протестантами, но не была принята в России, Швеции и Англии. На мой взгляд, отрицание грубой общеизвестной ошибки не делает Англии особой чести, особенно в такой компании. Вытекающие из этого факта неудобства сходны со сложностями в политической или финансовой корреспонденции. По этой причине я предпринял попытку реформы: проконсультировался с лучшими юристами и самыми опытными астрономами, после чего мы подготовили по этому вопросу билль. Но здесь

> начались первые трудности: я должен был использовать юридический язык и астрономические вычисления, но в этих вопросах я дилетант. Тем не менее, совершенно необходимо, чтобы в Палате Лордов сочли меня разбирающимся в этих вопросах, а также, чтобы они сами не чувствовали себя профанами. Со своей стороны, я уже изъяснялся с ними на Кельтском и Славонском на темы, связанные с астрономией, и там меня вполне поняли. Поэтому я решил, что лучше заниматься делом, а не разглагольствовать, и удовлетворять их вместо информирования. Я лишь дал им небольшой экскурс в историю календарей, от египетского к грегорианскому, развлекая их занимательными историческими эпизодами. Но я был крайне осмотрителен в выборе слов, применил свое ораторское искусство, чтобы представить гармоничность и завершенность интересующего меня случая. Это сработало, и сработало бы всегда. Я усладил их слух и донес нужную информацию, после чего многие из них сообщили, что проблема теперь совершенно ясна. Но, видит Бог, я даже не пытался ничего доказывать.

См. Roberts, David (Ed.) «Lord Chesterfield's Press»[1], Oxford, 1992.

Стр. Ё082out. В Common Lisp вселенским (universal) временем считается целое число, представляющее количество секунд, прошедших с начала 1900 года. Функции `encode-universal-time` и `decode-universal-time` осуществляют перевод дат между форматами. Таким образом, для дат после 1900 года в Common Lisp есть более простой способ выполнения этого преобразования:

```
(defun num->date (n)
  (multiple-value-bind (ig no re d m y)
      (decode-universal-time n)
    (values d m y)))

(defun date->num (d m y)
  (encode-universal-time 1 0 0 d m y))

(defun date+ (d m y n)
  (num->date (+ (date->num d m y)
                (* 60 60 24 n))))
```

Помимо вышеупомянутого ограничения, имеется предел, за которым представления дат не будут принадлежать типу `fixnum`.

Стр. Ё083out. Хотя вызов `setf` может пониматься как обращение к ссылке на конкретный объект, он реализует более общий механизм. Пусть, например, `marble` – это структура с одним полем `color`:

```
(defstruct marble
  color)
```

Следующая функция принимает список разновидностей мрамора и возвращает их цвет, если он один для всех экземпляров, и `nil` в противном случае:

```
(defun uniform-color (lst)
  (let ((c (marble-color (car lst))))
    (dolist (m (cdr lst))
      (unless (eql (marble-color m) c)
        (return nil)))
    c))
```

Хотя `uniform-color` не ссылается на конкретный адрес, возможно (и даже может иметь смысл) иметь возможность вызывать `setf` с этой функцией. Определив:

```
(defun (setf uniform-color) (val lst)
  (dolist (m lst)
    (setf (marble-color m) val)))
```

мы сможем сказать:

```
(setf (uniform-color *marbles*) 'red)
```

чтобы сделать каждый элемент в `*marbles*` красным.

---

[1] Вообще говоря, книга называется Lord Chesterfield's Letters. – *Прим. перев.*

Стр. Ё084out. В ранних реализациях Common Lisp приходилось использовать `defsetf` для определения раскрытия `setf`-выражения. При определении порядка следования аргументов необходимо проявлять аккуратность и помнить, что новое значение располагается *последним* в определении функции, передаваемой вторым аргументом `defsetf`.

Так, вызов:

```
(defun (setf primo) (val lst) (setf (car lst) val))
```

эквивалентен

```
(defsetf primo set-primo)
(defun set-primo (lst val) (setf (car lst) val))
```

Стр. Ё085out. Си, например, позволяет передавать указатель на функцию, но на деле вы не можете передавать эту функцию (в Си нет замыканий), а получатель на деле не может применять ее (в Си нет аналога `apply`). Кроме того, вы обязаны декларировать для нее тип возвращаемого значения. Сможете ли вы написать в Си аналог `map-int` или `filter`? Разумеется, нет. Вам придется подавлять проверку типов аргументов и возвращаемого значения, что опасно и, вероятно, возможно лишь для 32-битных значений.

Стр. Ё086out. За примерами многостороннего использования замыканий обращайтесь к Abelson, Harold, abd Gerald Jay Sussman, with Julie Sussman. «Structure and Interpretation of Computer Programs».[1] MIT Press, Cambridge, 1985.

Стр. Ё087out. За дополнительной информацией о языке Dylan обращайтесь к Shalit, Andrew, with Kim Barrett, David Moon, Orca Starbuck, and Steve Strassmann. «Dylan Interim Reference Manual». Apple Computer, 1994.

На момент печати[2] этот документ был доступен на нескольких ресурсах, в том числе *http://www.harlequin.com* и *http://www.apple.com*.

Scheme – очень маленький, аккуратный диалект Лиспа. Он был разработан Гаем Стилом и Джеральдом Сассманом в 1975 году и на настоящий момент определен в Clinger, William, and Jonathan A. Rees (Eds.) «Revised[4] Report on the Algorithmic Language Scheme»[3]. 1991.

Этот отчет, а также разнообразные реализации Scheme были доступны на момент печати через анонимный FTP: *swiss-ftp.ai.mit.edu:pub*.

Можно отметить две замечательные книги по языку Scheme: Структура и интерпретация, указанная ранее, и Springer, George and Daniel P. Friedman. «Scheme and the Art of Programming». MIT Press, Cambridge, 1989.

Стр. Ё088out. Наиболее неприятные баги с Лиспе связаны, вероятно, с динамическим окружением. Подобные вещи практически никогда не возникнут в Common Lisp, использующем по умолчанию лексический контекст. Многие диалекты Лиспа, используемые, как правило, как встраиваемые языки, работают с динамическим контекстом, и об этом необходимо помнить.

Один подобный баг напоминает проблему захвата переменных (см. стр. Ё059out). Представьте, что вы передаете одну функцию как аргумент другой, и эта функция получает некоторую переменную в качестве аргумента. Но внутри этой функции данная переменная не определена или имеет другое значение.

Предположим теперь, что мы написали ограниченную версию `mapcar`:

```
(defun our-mapcar (fn x)
  (if (null x)
      nil
      (cons (funcall fn (car x))
```

[1] Изучение этой книги принесет вам много пользы. В сети имеется русский перевод (под названием «Структура и интерпретация компьютерных программ»; книга также известна как SICP), решения большинства задач из книги, а также соответствующие видеолекции MIT с русскими субтитрами. – *Прим. перев.*

[2] На момент подготовки перевода ни один из этих ресурсов, похоже, не содержал упомянутого материала. Документация и прочая актуальная информация о языке Dylan доступна по адресу *http://www.opendylan.org*. – *Прим. перев.*

[3] На момент подготовки перевода момент актуальным являлась шестая версия стандарта (R6RS), доступная по адресу *http://www.r6rs.org*. – *Прим. перев.*

```
                (our-mapcar fn (cdr x)))))
```

Пусть теперь эта функция используется в другой, `add-to-all`, которая принимает число и добавляет его к каждому элементу списка:

```
(defun add-to-all (lst x)
  (our-mapcar #'(lambda (num) (+ num x))
              lst))
```

В Common Lisp этот код отработает без проблем, но в Лиспе с динамическим окружением мы получили бы ошибку. Функция, передаваемая как аргумент `our-mapcar`, ссылается на `x`. В этот момент `x` будет числом, переданным вторым аргументом `add-to-all`. Но внутри вызова функции внутри `our-mapcar`, `x` будет иметь иное значение. Когда список будет передан вторым аргументом +, будет получена ошибка.

Ё089out. Более новые реализации Common Lisp содержат переменную `*read-eval*`, которая может отключать макрос чтения `#.`. При вызове `read-from-string` с пользовательским вводом будет разумно связать `*read-eval*` с `nil`. В противном случае пользователь может получить побочные эффекты, используя #. во вводе.

Ё090out. Существует множество оригинальных алгоритмов быстрого поиска совпадений в строках, но, применительно к поиску в файлах, наш метод грубой силы оказывается вполне быстрым. За другими алгоритмами поиска совпадений с строках обращайтесь к Sedgewick, Robert. «Algorithms». Addison-Wesley, Reading (MA), 1988.

Стр. Ё091out. В 1984 году Common Lisp определял `reduce` без ключевого параметра `:key`, и `random-next` мог быть определен следующим образом:

```
(defun random-next (prev)
  (let* ((choices (gethash prev *words*))
         (i (random (let ((x 0))
                      (dolist (c choices)
                        (incf x (cdr c)))
                      x))))
    (dolist (pair choices)
      (if (minusp (decf i (cdr pair)))
          (return (car pair))))))
```

Стр. Ё092out. Программа, подобная Хенли, использовалась в 1989 году для симуляции новостных лент авторства известных скандалистов. Подобные сообщения одурачили значительное количество читателей. Как и все хорошие шутки такого рода, она была небеспочвенна. Что можно сказать насчет содержимого таких статей, если не всякий отличит их от случайно сгенерированного текста?

Серьезным вкладом в развитие искусственного интеллекта можно считать умение разделять задачи по сложности. Некоторые задачи оказываются вполне тривиальными, другие – почти неразрешимыми. Работы, связанные с искусственным интеллектом, посвящены в первую очередь последним задачам, отсюда родился шуточный термин для первых – *искусственная глупость* (*artificial stupidity*). Но это не значит, что они не важны, просто у них другие области применения.

Оказывается, что иногда слова вовсе не обязаны передавать какие-либо мысли, и этому способствует склонность людей усиленно искать смысл там, где его нет. Если оратор позаботится о привнесении малейшего смысла в свои речи, доверчивая публика несомненно сочтет их глубокомысленными.

Пожалуй, это явление ничуть не моложе самого человечества. Однако новые технологии позволяют шагнуть еще дальше, примером чему может служить наша программа по генерации случайного текста. Она предельно проста, но может заставить людей искренне считать свою «поэзию» делом рук человеческих (проверьте на своих друзьях). Более сложные программы, несомненно, смогут генерировать более хитрые тексты.[1]

[1] Описанная идея породила немало других шуток, в том числе в России. Многие из них играют на псевдонаучности получаемого текста. Например, сервис Яндекс.Рефераты предлагает генерацию

Интересное обсуждение случайной генерации стихов как полноценной литературной формы вы можете найти здесь: Low, Jackson M. Poetry, Chance, Silence, Etc. In Hall, Donald (Ed.) «Claims for Poetry». University of Michigan Press, Ann Arbor, 1982. Стараниями Online Book Initiative текстовые версии многих классических работ, включая эту, доступны в сети. На момент издания[1] они были доступны по анонимному FTP: *ftp.std.com:obi*.

Также вам может показаться интересным режим dissociated-press в Emacs.

Стр. Ё093out. Следующая функция отображает значения шестнадцати констант, определяющих ограничения на размер чисел с плавающей запятой в используемой реализации:

```
(defun float-limits ()
  (dolist (m '(most least))
    (dolist (s '(positive negative))
      (dolist (f '(short single double long))
        (let ((n (intern (string-upcase
                          (format nil "~A-~A-~A-float"
                                  m s f)))))
          (format t "~30A  ~A~%" n (symbol-value n)))))))
```

Стр. Ё094out. Алгоритм `quicksort` был предложен Хоаром в 1962 году и описан Кнутом в Knuth, D. E. «Sorting and Searching». Addison-Wesley, Reading (MA), 1973.

Стр. Ё095out. Foderaro, John K. «Introduction to the Special Lisp Section». CACM 34:9 (September 1991), p. 27.

Стр. Ё096out. За более детальной информацией об использовании CLOS обращайтесь к:

Keene, Sonya E. «Object Oriented Programming in Common Lisp». Addison-Wesley, Reading (MA), 1989.

Kiczales, Gregor, Jim des Rivieres, and Daniel G. Bobrow. «The Art of the Metaobject Protocol». MIT Press, Cambridge, 1991.

Стр. Ё097out. Давайте еще раз отмотаем назад: *мы можем выполнить модификации такого рода даже без просмотра остального кода*. Эти слова могут прозвучать тревожно для некоторых читателей. Это рецепт спагетти-кода[2].

Объектно-ориентированная модель позволяет упростить построение программ с помощью их расширения. Такая возможность в неумелых руках может привести к спагетти-коду. Это не обязательно плохо, но ничего хорошего в таком явлении тоже нет.

Приличная часть когда-либо написанного кода является именно таким кодом, и он вряд ли подлежит модификации в будущем. Объектная ориентированность слегка облегчает ситуацию, немного структурируя код. Но для программ, которые желали бы избежать подобной участи, использование объектной ориентированности может быть довольно опасным.

Стр. Ё098out.Ё037in При наследовании экземпляром слота с одинаковым именем от нескольких суперклассов наследуемый слот содержит суперпозицию свойств всех родительских слотов. Механизм комбинации может различаться для разных свойств:

1. Свойства `:allocation`, `:initform` (если имеется) и `:documentation` (если имеется), будут получены из наиболее специфичного класса.
2. Свойства `:initarg` будет объединением параметров `:initarg` всех суперклассов. Это справедливо также для свойств `:accessor`, `:reader` и `:writer`.

рефератов на заданные темы.
Скандалом обернулся также эксперимент М. Гельфанда, в котором псевдонаучная статья под названием «Корчеватель: алгоритм типичной унификации точек доступа и избыточности», сгенерированная программой SCIgen (написанной в MIT), и переложенная на русский язык с помощью отечественной разработки «Этап-3», была допущена до печати в один из журналов ВАК (в 2008 г). Подобная шутка была позже организована в одном из американских журналов. – *Прим. перев.*

[1] Похоже, этот документ, как и сама организация OBI, канул в лету. – *Прим. перев.*

[2] Спагетти-кодом называют запутанный код, содержащий множество исключений и условных переходов. – *Прим. перев.*

3. Свойство `:type` будет пересечением параметров `:type` всех суперклассов.

Стр. Ё099out. Вы можете избежать необходимости удалять символы вручную, изначально используя неинтернированные символы:

```
(progn
  (defclass counter 0) ((#1=#:state :initform 0)))

  (defmethod increment ((c counter))
    (incf (slot-value c '#1#)))

  (defmethod clear ((c counter))
    (setf (slot-value c '#1#) 0)))
```

Особого смысла `progn` здесь не несет, он просто показывает, что эти выражения должны выполняться вместе в заданном порядке. Это не совсем удобно, и вы можете воспользоваться следующим макросом чтения:

```
(defvar *symtab* (make-hash-table :test #'equal))

(defun pseudo-intern (name)
  (or (gethash name *symtab*)
      (setf (gethash name *symtab*) (gensym))))

(set-dispatch-macro-character #\# #\[
  #'(lambda (stream charl char2)
      (do ((ace nil (cons char ace))
           (char (read-char stream) (read-char stream)))
          ((eql char #\]) (pseudo-intern ace)))))
```

Теперь вы сможете сделать следующее:

```
(defclass counter () ((#[state] :initform 0)))

(defmethod increment ((c counter))
  (incf (slot-value c '#[state])))

(defmethod clear ((c counter))
  (setf (slot-value c '#[state]) 0))
```

Стр. Ё100out. Данный макрос добавляет новый элемент в двоичное дерево поиска:Ё056in

```
(defmacro bst-push (obj bst <)
  (multiple-value-bind (vars forms var set access)
      (get-setf-expansion bst)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
              ,@(mapcar #'list vars forms)
              (,(car var) (bst-insert! ,g ,access ,<)))
         ,set))))
```

Стр. Ё101out. Knuth, Donald E. «Structured Programming with goto Statements». Computing Surveys, 6:4 (December 1974), pp. 261-301.

Стр. Ё102out. Knuth, Donald E. «Computer Programming as an Art». In ACM Turing Award Lectures: The First Twenty Years. ACM Press, 1987.

Эта статья, как и предыдущая, была переиздана в: Knuth, Donald E. «Literate Programming». CSLI Lecture Notes #27, Stanford University Center for the Study of Language and Information, Palo Alto, 1992.

Стр. Ё103out. Steele, Guy L., Jr. «Debunking the "Expensive Procedure Call" Myth or, Procedural Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO». Proceedings of the National Conference of the ACM, 1977, p. 157.

Суть оптимизация хвостовой рекурсии в замене рекурсивных вызовов аналогичным итеративным выражением. К сожалению, многие компиляторы, актуальные на момент написания книги, справлялись с этой задачей не лучшим образом.

Стр. Ё104out. Некоторые примеры использования `disassemble` на различных процессорах вы сможете найти в Norvig, Peter. «Paradigms of Artificial Intelligence

Programming: Case Studies in Common Lisp». Morgan Kaufmann, San Mateo (CA), 1992.

Стр. Ё105out. Популярность объектно-ориентированного программирования часто связывают с популярностью C++, которая возникла благодаря усовершенствованиям в плане типизации. Концептуально C++ связан с Си, но, в числе прочего, позволяет создавать операторы, работающие с аргументами разных типов. Но для этого вовсе не обязательно использовать объектно-ориентированное программирование, достаточно динамической типизации. Если вы понаблюдаете за людьми, использующими C++, то обратите внимание на плоскость создаваемых ими иерархий классов. C++ получил распространение не потому, что позволяет работать с классами и методами, но потому, что позволяет обойти некоторые ограничения языка Си.

Стр. Ё106out. Создание деклараций значительно упрощается с помощью макросов. Следующий макрос получает имя типа и выражение (вероятно, численное), и генерирует раскрытие, в котором задекларирован заданный тип для всех промежуточных результатов. Чтобы гарантировать, что выражение `e` будет вычислено с помощью только fixnum-арифметики, достаточно сказать `(with-type fixnum e)`.

```
(defmacro with-type (type expr)
  '(the ,type ,(if (atom expr)
                   expr
                   (expand-call type (binarize expr)))))

(defun expand-call (type expr)
  `(,(car expr) ,@(mapcar #'(lambda (a)
                              `(with-type ,type ,a))
                          (cdr expr))))

(defun binarize (expr)
  (if (and (nthcdr 3 expr)
           (member (car expr) '(+ - * /)))
      (destructing-bind (op a1 a2 . rest) expr
        (binarize `(,op (,op ,a1 ,a2) ,@rest)))
      expr))
```

Вызов `binarize` гарантирует, что никакой арифметический оператор не вызывается более, чем с двумя аргументами. Вызов типа:

```
(the fixnum (+ (the fixnum a)
               (the fixnum b)
               (the fixnum c)))
```

не может быть скомпилирован в сложения чисел типа `fixnum`, так как промежуточный результат (`a+b`) может не принадлежать типу `fixnum`.

С помощью `with-type` мы можем заменить версию `poly`, содержащую полный набор деклараций (см. стр. Ё060out):

```
(defun poly (a b x)
  (with-type fixnum (+ (* a (expt x 2)) (* b x ))))
```

Если вам предстоит выполнять много подобных действий, разумно даже определить макрос чтения, раскрывающийся в вызов (`with-type fixnum ...`).

Стр. Ё107out. На многих Unix-системах пригодным файлом со словами является /usr/dict/words.

Стр. Ё108out. T является диалектом Scheme с множеством полезных дополнений, включая поддержку пулов. Больше по теме: Rees, Jonathan A., Norman I. Adams, and James R.Meehan. «The T Manual», 5th Edition. Yale University Computer Science Department, New Haven, 1988.

На момент публикации[1] руководство по T, а также его реализация, были доступны по анонимному FTP: *hing.lcs.mit.edu:pub/t3.1*.

[1] На момент подготовки перевода этот ресурс не был доступен. Данный документ легко отыскать в сети, но не похоже, чтобы данный диалект представлял на настоящий момент какой-либо существенный интерес. – *Прим. перев.*

Стр. Ё109out. Различия между спецификациями и программами можно считать количественными, но не качественными. Однажды поняв это, становится неестественно требовать, чтобы кто-либо составлял документацию до начала реализации проекта. Если программа пишется на низкоуровневом языке, то будет полезно предварительно описать задачу в терминах высокого уровня. Но при переходе к высокоуровневым языкам необходимость в подобных действиях исчезает. В некотором роде, реализация и промежуточная спецификация становятся одним и тем же.

Если программа, ранее написанная на низкоуровневом языке, переписывается на более абстрактном языке, ее реализация начинает напоминать спецификацию даже в большей степени, чем использовавшаяся ранее спецификация. Перефразируя основную мысль раздела 13.7, спецификация программы на Си может быть написана на Лиспе.

Стр. Ё110out. История отливки скульптуры «Персей» Бенвенутто Челлини, вероятно, наиболее известная (и наиболее курьезная), по сравнению с другими литыми бронзовыми скульптурами. Cellini, Benvenuto. «Autobiography». Перевод George Bull, Penguin Books, Harmonds worth, 1956.

Стр. Ё111out. Даже опытные Лисп-хакеры иногда находят работу с пакетами неудобной. Потому ли это, что они и правда сложны, или мы просто не привыкли думать, что происходит с ними в момент чтения?

Подобная неустойчивость также связана с `defmacro`. Была бы очень кстати более абстрактная версия `defmacro`. Однако лишь подход, реализуемый `defmacro`, может обеспечить решение, наиболее близкое по духу к определению функций. Да, неожиданно возникающая проблема захвата переменных может обескуражить. Но и к ней можно привыкнуть со временем, и обход этой проблемы станет не сложнее обхода деления на ноль.

Также и со сложностями пакетной механики. Пакеты предоставляют удобную реализацию модульности. Даже при беглом взгляде очевидно, что они сильно напоминают методики, используемые программистами при отсутствии официальной поддержки модульности.

Стр. Ё112out. Конструкция `loop` вызывает множество споров, и вопрос, нужны ли операторы, дублирующие возможности `loop`, небеспочвенен. Единственным преимуществом `loop` является то, что это один оператор. С этой точки зрения `eval` также является одним оператором, а с точки зрения пользователя `loop` содержит столько операторов, сколько поддерживает различных типов предложений. Эти предложения не могут использоваться отдельно, как полноценные операторы в Лиспе: невозможно разорвать оор-выражение и вставить промежуточное вычисление с `map-int`, например.

Стр. Ё113out. Больше по теме логического вывода: Russell, Stuart, and Peter Norvig. «Artificial Intelligence: A Modern Approach»[2]. Prentice Hall, Englewood Cliffs (NJ), 1995.

Стр. Ё114out. Программа в главе 17 использует преимуществе, даваемое возможностью `setf` формировать первый аргумент `defun`. Эта возможность отсутствовала в ранних реализациях Common Lisp. Чтобы иметь возможность запускать приведенный код на ранних реализациях, потребуется небольшая модификация:

```
(proclaim '(inline lookup set-lookup))

(defsetf lookup set-lookup)

(defun set-lookup (prop obj val)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off
```

[2] Книга переведена на русский: Рассел С., Норвиг П. Искусственный интеллект: современный подход. Пер. с англ. и ред. К.А.Птицына. – 2-е изд. – М.: Вильямс, 2006.
Книга (известная также, как AIMA) представляет собой фундаментальный труд, посвященный проблемам ИИ. Она ценна тем, что дает систематизированное представление об ИИ и объединяет основные его концепции, но вряд ли может служить полноценным руководством по какой-либо отдельно взятой области ИИ. Вас также может заинтересовать код, сопровождающий книгу. Код на Common Lisp свободно доступен по адресу: *http://aima.cs.berkeley.edu/lisp/*. – *Прим. перев.*

```
          (setf (svref obj (+ off 3)) val)
          (error "Can't set "A of ~A." val obj))))

(defmacro defprop (name ftoptional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj nil)))
  (defsetf ,name (obj) (val)
    `(setf (lookup ',',name ,obj) ,val))))
```

Стр. Ё115out. Определив `defmeth` следующим образом:

```
(defmacro defmeth (name obj parms ferest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (setf (gethash ',name ,gobj)
             #'(lambda ,parms
                 (labels ((next ()
                            (funcall (get-next ,gobj ',name)
                                     ,@parms)))
                   ,@body))))))
```

то получили бы возможность вызывать следующий метод с помощью просто `next`:

```
(defmeth area grumpy-circle (c)
  (format t "How dare you stereotype me!~%")
  (next))
```

Выглядит проще. Однако, чтобы выполнять работу `next-method-p`, нам придется определить еще одну функцию.

Стр. Ё116out. Для по-настоящему быстрого доступа к слотам может пригодиться следующий макрос:

```
(defmacro with-slotref ((name prop class) &rest body)
  (let ((g (gensym)))
    `(let ((,g (+ 3 (position ,prop (layout ,class)
                              :test #;eq))))
       (macrolet ((,name (obj) `(svref ,obj ,',g)))
         ,@body))))
```

Он определяет локальный макрос, ссылающийся непосредственно на элемент вектора, соответствующий слоту. Попытка доступа к слоту будет оттранслирована непосредственно в вызов `svref`.

Пусть, например, определен класс `balloon`:

```
(setf balloon-class (class nil size))
```

тогда мы сможем воспользоваться нашим макросом, чтобы быстро лопнуть все шарики.

```
(defun popem (balloons)
  (with-slotref (bsize 'size balloon-class)
    (dolist (b balloons)
      (setf (bsize b) 0))))
```

Стр. Ё117out. Gabriel, Richard P. Lisp: «Good News, Bad News, How to Win Big». AI Expert, June 1991, p. 35.

Еще в далеком 1973 году Ричард Фейтман сумел показать, что компилятор MacLisp для PDP-10 производил более быстрый код, чем реализация Фортрана от производителя архитектуры. См. Fateman, Richard J. «Reply to an editorial». ACM SIGSAM Bulletin, 25 (March 1973), pp. 9-11.

Стр. Ё118out. Еще более простым способом понять суть обратной кавычки, вероятно, является следующая идея. Будем рассматривать ее подобно обычной, а `` `,x `` будем раскрывать в `(bq (comma x))`. Тогда мы сможем обработать обратную кавычку, расширяя `eval`, как показано в следующем наброске:

```
(defun eval2 (expr)
  (case (and (consp expr) (car expr))
    (comma (error "unmatched comma"))
    (bq (eval-bq (second expr) 1))
    (t (eval expr))))

(defun eval-bq (expr n)
  (cond ((atom expr)
         expr)
        ((eql (car expr) 'comma)
         (if (= n 1)
             (eval2 (second expr))
             (list 'comma (eval-bq (second expr)
                                   (1- n)))))
        ((eql (car expr) 'bq)
         (list 'bq (eval-bq (second expr) (1+ n))))
        (t
         (cons (eval-bq (car expr) n)
               (eval-bq (cdr expr) n)))))
```

Параметр `n`, используемый в `eval-bq`, используется для нахождения совпадающих на текущий момент запятых. Каждая обратная кавычка увеличивает значение `n`, а каждая запятая уменьшает. Запятая, встреченная с `n`=1, считается совпадающей.

Вот пример со стр. Ё061out.

```
> (setf x 'a a 1 y ' b b 2)
2
> (eval2 ' (bq (bq (w (comma x) (comma (comma y))))))
(BQ (W (COMMA X) (COMMA B)))
> (eval2 *)
(W A 2)
```

В некоторый момент случайным образом образовалась особенная молекула. Назовем ее Репликатором. Она не обязательно была самой большой или самой сложной в окрестности себя, но она обладала особым свойством – умением копировать саму себя.

Ричард Докинз

«Эгоистичный Ген»

Для начала нам следует определить набор символьных выражений в терминах ориентированных пар и списков. Затем следует определить пять элементарных функций и предикатов и построить на их основе условные выражения и рекурсивные определения более широкого набора функций, с которыми мы приведем ряд примеров. Затем мы покажем, как сами эти функции могут быть выражены в символьном виде, а также определим универсальную функцию apply, позволяющую вычислять значение заданной функции с заданными аргументами.

Джон Маккарти

Рекурсивные функции в символьных выражениях и их аппаратное вычисление, часть I.

Предметный указатель

1- функция....251
1+ функция....251
алгоритм....
  быстрой сортировки....120
  Горнера....116
  кодирования повторов....32
  логического вывода....174
  обратного логического вывода....174
  поиска в ширину....43
  поиска дихотомией....48
  поиска кратчайшего пути....41
  сопоставления с образцом....174
  трассировки лучей....111
атом....15
бредогенерация....102
Брукс, Фредерик....12
буфер....93
  кольцевой....93
встроенный язык....188
выражение....14
  вложенное....15
  правило вычисления....15
  самовычисляемое....14
  цитирование....16
дерево....34
  бинарное (двоичное)....35, 55
  сбалансированное....55
замыкание....81
значение....
  множественные....21, 68
  присвоение....23
  указатель на значение....31
инкапсуляция....137
инфиксная нотация....14
истинность....17
итерация....24
кавычка....
  обратная....119
класс....129
  родительский....132
Кнут, Дональд....9, 151
код....
  выравнивание....20
  парные скобки....20
  повторное использование....80
  спагетти....294
  чтение....20
кодогенерация....16, 54, 117
комбинация методов....136
  операторная....136
комментирование....36
константа....22
куча....290
Лисп....
  выразительная сила....10
  для численных расчетов....106
  единообразие....15
  интерактивность....14, 19, 79, 152
  код как списки....16
  происхождение....10
  символьные вычисления....174
  типы данных....16
  функциональная парадигма. .23, 75, 78, 177, 191
ложность....18
лямбда-выражение....25
Маккарти, Джон....10, 299
макрос чтения....96
  управляемый....97
макросимвол....96, 165
математическая индукция....35
метод....128
  вспомогательный....134
  квалификатор....135
  первичный....134
  применимый....133
  специфичность....133
наследование....129
наследование множественное....189
окружение....
  лексическое....64, 80
оператор....
  деструктивный....142
  логический....18, 178
  специальный....18, 118
  условный....18, 65
отладчик....70
очередь....142
палиндром....38
память....
  автоматическое управление....44
  висячий указатель....44
  выделение....44, 156
  куча....44
  мусор....44
  сборка мусора....44
  утечка....44
параметр....
  ключевой....36, 77
  необязательный....77
  обязательный....77
  остаточный....77
переменная....19
  глобальная....22, 84
  локальная....22, 84
  свободная....81
  специальная....84
печатные знаки....53
побочный эффект....21, 23
предшествование....132
префиксная нотация....14, 191
программа....
  загрузка....34
программирование....
  прототипирование....12
  сверху-вниз....78
  снизу-вверх....11, 78, 80, 180
  спецификация....12
  функциональное....23
профилировщик....151
пул....159
рекурсия....
  базовый случай....36, 202
  понимание....20, 35, 86
  хвостовая....86, 152

Си.....
выразительная сила.....10
сложный синтаксис.....15
символ.....16
импортированный.....168
интернированный.....100
как переменная.....19
как слово.....102
ключевое слово.....101
неинтернированный.....121
экспортируемый.....101, 168
слот.....129
список.....16, 29
ассоциативный.....40
блочное представление.....29
вложенный.....30
как код.....117
как множество.....36
как очередь.....43
как последовательность.....37
как стек.....39
нормальный.....40
предшествования.....132, 190
пустой.....17
свойств.....99
структура верхнего уровня.....140
устройство.....17
циклический.....148
стек.....158
считыватель.....21
тип.....26
атомарный.....164
встроенный.....26
иерархия.....26, 164
как множество.....164
конечный.....164
целочисленный.....106
типизация.....
динамическая.....154
строгая.....153
точечная пара.....40
функция.....
высшего порядка.....80, 82
глобальная.....75
деструктивная.....38, 44, 143
значение.....19
имя.....18
как агрегат.....20
как данные.....25
как значение.....80
как процесс.....20
комбинация функций.....80
локальная.....76
определение.....18
отображающая.....34
параметры.....18
рекурсивная.....19
тело.....18
утилита.....78, 183
экземпляр.....129
ячейка.....29
- переменная.....285
- функция.....251
; макрос чтения.....287
' макрос чтения.....287
(макрос чтения.....287
) макрос чтения.....287
* переменная.....285
* функция.....251
** переменная.....285
*** переменная.....285
break-on-signals переменная.....281
compile-file-pathname переменная.....281
compile-file-truename переменная.....281
compile-print переменная.....281
compile-verbose переменная.....281
debug-io переменная.....281
debugger-hook переменная.....281
default-pathname-defaults переменная.....281
error-output переменная.....282
features переменная.....282
gensym-counter переменная.....282
load-pathname переменная.....283
load-print переменная.....283
load-truename переменная.....283
load-verbose переменная.....283
macroexpand-hook переменная.....283
modules переменная.....283
package переменная.....283
print-array переменная.....283
print-base переменная.....283
print-case переменная.....284
*print-circle переменная.....284
print-escape переменная.....284
print-gensym переменная.....284
print-length переменная.....284
print-level переменная.....284
print-lines переменная.....284
print-miser-width переменная.....284
print-pprint-dispatch переменная.....284
print-pretty переменная.....284
print-radix переменная.....284
print-right-margin переменная.....284
random-state переменная.....284
read-base переменная.....284
read-default-float-format переменная.....284
read-eval переменная.....285
read-suppress переменная.....285
readtable переменная.....285
standard-input переменная.....285
standard-output переменная.....285
terminal-io переменная.....285
trace-output переменная.....285
/ переменная.....285
/ функция.....251
// переменная.....285
/// переменная.....285
/= функция.....251
#
макрос чтения.....287
#- макрос чтения 288
#. макрос чтения.....287
#' макрос чтения.....287
#(макрос чтения.....287
#* макрос чтения.....287
#\ макрос чтения.....287
#+ макрос чтения 288
#< макрос чтения 288
#| макрос чтения 288
#A макрос чтения.....287
#B макрос чтения.....287
#C макрос чтения.....287
#n# макрос чтения.....288
#n= макрос чтения.....288
#O макрос чтения.....287
#P макрос чтения.....288
#R макрос чтения.....287
#S макрос чтения.....287
#X макрос чтения.....287
` макрос чтения.....287

+ переменная....285
+ функция....251
++ переменная....285
+++ переменная....285
< функция....251
<= функция....251
= функция....251
> функция....251
>= функция....251
abort функция....239
abs функция....245
acons функция....253
acos функция....245
acosh функция....245
add-method обобщенная функция....231
adjoin функция....253
adjust-array функция....257
adjustable-array-p функция....257
allocate-instance обобщенная функция....231
alpha-character-p функция....251
alphanumericp функция....252
and макрос....221
append функция....253
apply функция....221
apropos функция....279
apropos-list функция....279
aref функция....257
arithmetic-error-operands функция....245
arithmetic-error-operation функция....245
array-dimension функция....257
array-dimension-limit константа....281
array-dimensions функция....257
array-displacement функция....258
array-element-type функция....258
array-has-fill-pointer-p функция....258
array-in-bounds-p функция....258
array-rank функция....258
array-rank-limit константа....281
array-row-major-index функция....258
array-total-size функция....258
array-total-size-limit константа....281
arrayp....258
ash функция....245
asin функция....246
asinh функция....246
assert макрос....239
assoc функция....253
assoc-if функция....253
assoc-if-not функция....253
atan функция....246
atanh функция....246
atom функция....253
Autocad....11
bit функция....258
bit-and функция....258
bit-andc1 функция....258
bit-andc2 функция....258
bit-eqv функция....258
bit-ior функция....258
bit-nand функция....258
bit-nor функция....258
bit-not функция....258
bit-orc1 функция....258
bit-orc2 функция....258
bit-vector-p функция....259
bit-xor функция....258
block специальный оператор....222
boole функция....246
boole-1 константа....281
boole-2 константа....281
boole-xor константа....281
both-case-p функция....251
boundp функция....242
break loop....15
break функция....239
broadcast-stream-streams функция....268
butlast функция....253
byte функция....246
byte-position функция....246
byte-size функция....246
caaaar функция....253
call-arguments-limit константа....281
call-method....231
call-next-method функция....231
car функция....253
case специальный оператор....222
catch специальный оператор....222
ccase макрос....222
cdr функция....253
ceiling функция....246
cell-error-name функция....239
cerror функция....239
change-class обобщенная функция....232
char функция....260
char-code функция....252
char-code-limit константа....281
char-downcase функция....252
char-equal функция....252
char-greaterp функция....252
char-int функция....252
char-lessp функция....252
char-name функция....252
char-not-equal функция....252
char-not-greaterp функция....252
char-not-lessp функция....252
char-upcase функция....252
char/= функция....252
char< функция....252
char<= функция....252
char= функция....252
char> функция....252
char>= функция....252
character функция....252
characterp функция....252
check-type макрос....239
cis функция....246
class-name....232
class-of функция....232
clear-input функция....268
clear-output &optional функция....268
close функция....268
clrhash функция....264
code-char функция....252
coerce функция....221
compile функция....219
compile-file функция....278
compile-file-pathname функция....278
compiled-function-p функция....222
compiler-macro-function функция....219
complement функция....222
complex функция....246
complexp функция....246
compute-applicable-methods обобщенная функция....232
compute-restarts функция....239
concatenate функция....261
concatenated-stream-streams функция....268
cond макрос....222
conjugate функция....247
cons функция....253

consp функция....254
constantly функция....222
constantp функция....219
continue функция....239
copy-alist функция....254
copy-list функция....254
copy-pprint-dispatch функция....272
copy-readtable функция....277
copy-seq функция....262
copy-structure функция....237
copy-symbol функция 242
copy-tree функция....254
cos функция....247
cosh функция....247
count функция....262
count-if функция....262
count-if-not функция....262
ctypecase макрос....222
decf макрос....247
declaim макрос....219
declare....219
decode-float функция....247
decode-universal-time функция....279
defclass макрос....232
defconstant макрос....222
defgeneric макрос....233
define-compiler-macro макрос....219
define-condition макрос....239
define-method-combination макрос....233, 234
define-modify-macro макрос....222
define-setf-expander макрос....223
define-symbol-macro макрос....219
defmacro макрос....219
defmethod макрос....234
defpackage макрос....243
defparameter макрос....223
defsetf макрос....223
defstruct макрос....237
deftype макрос....221
defun макрос....223
defvar макрос....223
delete функция....263
delete-duplicates функция....263
delete-file функция....267
delete-if функция....263
delete-if-not функция....263
delete-package функция....243
denominator функция....247
deposit-field функция....247
describe функция....279
destructing-bind макрос....223
digit-char функция....253
digit-char-p функция....253
directory функция....267
directory-namestring функция....265
disassemble функция....279
do макрос....228
do-all-symbols макрос....243
do-external-symbols макрос....243
do-symbols макрос....243
do* макрос....228
documentation обобщенная функция....279
dolist макрос....228
dotimes макрос....228
double-float-epsilon константа....281
double-float-negative-epsilon константа....282
dpb функция....247
dribble функция....279
dynamic-extent декларация....219
ecase макрос....223
echo-stream-input-stream функция....268
echo-stream-output-stream функция....268
ed функция....279
elt функция....262
Emacs....11, 20
encode-universal-time функция....279
endp функция....254
enough-namestring функция....265
ensure-directories-exist функция....267
ensure-generic-function функция....235
eq функция....223
eql функция....223
equal функция....223
equalp функция....224
error функция....239
etypecase макрос....224
eval функция....220
eval-when....220
evenp функция....247
every функция....224
exp функция....247
export функция....244
expt функция....247
fboundp функция....224
fceiling функция....247
fdefinition функция....224
ffloor функция....247
file-author функция....267
file-error-pathname функция....267
file-length функция....268
file-namestring функция....265
file-position функция....268
file-string-length функция....268
file-write-date функция....267
fill функция....262
fill-pointer функция....259
find функция....262
find-all-symbols функция....244
find-class функция....235
find-if функция....262
find-if-not функция....262
find-method обобщенная функция....235
find-package функция....244
find-restart функция....239
find-symbol функция....244
finish-output функция....268
first функция....254
flet специальный оператор....224
float функция....247
float-digits функция....247
float-precision функция....247
float-radix функция....247
float-sign функция....247
floatp функция....247
floor функция....247
fmakunbound функция....224
force-output функция....268
format функция....272
formatter макрос....275
fresh-line функция....268
fround функция....247
ftruncate функция....248
ftype декларация....219
funcall функция....224
function специальный оператор....224
function-keywords обобщенная функция....235
function-lambda-expression функция....224
functionp функция....224
gcd функция....248
gensym функция....242

gentemp функция....242
get функция....242
get-decoded-time функция....279
get-dispatch-macro-character функция....277
get-internal-real-time функция....279
get-internal-run-time функция....280
get-macro-character функция....277
get-output-stream-string функция....268
get-properties функция....254
get-setf-expansion функция....225
get-universal-time функция....280
getf функция....254
gethash функция....264
go специальный оператор....225
graphic-char-p функция....253
handler-bind макрос....240
handler-case....240
hash-table-count функция....265
hash-table-p функция....265
hash-table-rehash-size функция....265
hash-table-size функция....265
hash-table-test функция....265
host-namestring функция....266
identity функция....225
if специальный оператор....225
ignorable декларация....219
ignore декларация....219
ignore-errors макрос....240
imagpart функция....248
import функция....244
incf макрос....248
initialize-instance обобщенная функция....235
inline декларация....219
input-stream-p функция....269
inspect функция....280
integer-decode-float функция....248
integer-length функция....248
integerp функция....248
interactive-stream-p функция....269
Interleaf....11
intern функция....244
internal-time-units-per-second константа....282
intersection функция....254
invalid-method-error функция....240
invoke-debugger функция....240
invoke-restart функция....240
invoke-restart-interactively функция....240
isqrt функция....248
keywordp функция....242
labels специальный оператор....224
lambda макрос....220
lambda-list-keywords константа....282
lambda-parameters-limit константа....282
last функция....254
lcm функция....248
ldb функция....248
ldb-test функция....248
ldiff функция....254
least-negative-double-float константа....282
least-negative-long-float константа....282
least-negative-normalized-double-float константа....282
least-negative-normalized-long-float константа....282
least-negative-normalized-short-float константа....282
least-negative-normalized-single-float константа....282
least-negative-short-float константа....282
least-negative-single-float константа....282
least-positive-double-float константа....282
least-positive-long-float константа....282
least-positive-normalized-double-float константа....282
least-positive-normalized-long-float константа....283
least-positive-normalized-short-float константа....282
least-positive-normalized-single-float константа....282
least-positive-short-float константа....282
least-positive-single-float константа....282
length функция....262
let специальный оператор....225
let* специальный оператор....225
lisp-implementation-type функция....280
lisp-implementation-version функция....280
list функция....254
list-all-packages функция....244
list-length функция....254
list* функция....254
listen функция....269
listp функция....254
load функция....278
load-logical-pathname-translations функция. 266
load-time-value специальный оператор....220
locally специальный оператор....220
log функция....248
logand функция....248
logandc1 функция....248
logbitp функция....248
logcount функция....248
logeqv функция....248
logical-pathname функция....266
logical-pathname-translations функция....266
logior функция....249
lognand функция....249
lognor функция....249
lognot функция....249
logorc1 функция....249
logorc2 функция....249
logtest функция....249
logxor функция....249
long-float-epsilon константа....281
long-float-negative-epsilon константа....282
long-site-name функция....280
loop макрос....228
loop-finish макрос....231
lower-case-p функция....253
machine-instance функция....280
machine-type функция....280
machine-version функция....280
macro-function функция....220
macroexpand функция....220
macroexpand-1 функция....220
macrolet специальный оператор....225
make-array функция....259
make-broadcast-stream функция....269
make-concatenated-stream функция....269
make-condition функция....240
make-dispatch-macro-character функция....277
make-echo-stream функция....269
make-hash-table функция....265
make-instance обобщенная функция....235
make-instance-obsolete обобщенная функция....235
make-list функция....254
make-load-form обобщенная функция....236
make-load-form-saving-slots функция....236
make-package функция....244

make-pathname функция....266
make-random-state функция....249
make-sequence функция....262
make-string функция....260
make-string-input-stream функция....269
make-string-output-stream функция....269
make-symbol функция....242
make-synonym-stream функция....269
make-two-way-stream функция....269
makunbound функция....242
map функция....262
map-into функция....262
mapc функция....254
mapcan функция....255
mapcar функция....255
mapcon функция....255
maphash функция....265
mapl функция....255
maplist функция....255
mask-field функция....249
max функция....249
member функция....255
member-if функция....255
member-if-not функция....255
merge функция....262
merge-pathnames функция....266
method-combination-error функция....240
method-qualifiers обобщенная функция....236
min функция....249
minusp функция....249
mismatch функция....262
mod функция....249
most-negative-double-float константа....283
most-negative-fixnum константа....283
most-negative-long-float константа....283
most-negative-short-float константа....283
most-negative-single-float константа....283
most-positive-double-float константа....283
most-positive-fixnum константа....283
most-positive-long-float константа....283
most-positive-short-float константа....283
most-positive-single-float константа....283
muffle-warning функция....240
multiple-value-bind макрос....225
multiple-value-call специальный оператор. .225
multiple-value-list макрос....225
multiple-value-prog1 специальный оператор 225
multiple-value-setq макрос....225
multiple-values-limit константа....283
name-char функция....253
namestring функция....266
nbutlast функция....253
nconc функция....255
nerverse функция....264
next-method-p функция....236
nil константа....283
nintersection функция....254
no-applicable-method обобщенная функция 236
no-next-method обобщенная функция....236
not функция....225
notany функция....226
notevery функция....226
notinline декларация....219
nreconc функция....256
nreverse функция....264
nset-difference функция....256
nset-exclusive-or функция....256
nstring-capitalize функция....260
nstring-downcase функция....260
nstring-upcase функция....261
nsublis функция....256
nsubst функция....256
nsubst-if функция....256
nsubst-if-not функция....257
nsubstitute функция....264
nsubstitute-if функция....264
nsubstitute-if-not функция....264
nth функция....255
nth-value макрос....226
nthcdr функция....255
null функция....255
numberp функция....249
numerator функция....249
nunion функция....257
oddp функция....249
open функция....269
open-stream-p функция....270
optimize декларация....219
or макрос....226
OS/360....12
package-error-package функция....244
package-name функция....244
package-nicknames функция....244
package-shadowing-symbols функция....244
package-use-list функция....244
package-used-by-list функция....244
packagep функция....244
pairlis функция....255
parse-integer функция....250
parse-namestring функция....266
pathname функция....266
pathname-device функция....266, 267
pathname-directory функция....267
pathname-host функция....266
pathname-match-p функция....267
pathname-name функция....267
pathname-type функция....267
pathnamep функция....267
phase функция....250
pi константа....283
plus функция....250
pop макрос....255
position функция....262
position-if функция....263
position-if-not функция....263
pprint функция....275
pprint-dispatch функция....275
pprint-exit-if-list-exhausted функция....275
pprint-fill функция....275
pprint-indent функция....275
pprint-linear функция....275
pprint-logical-block макрос....275
pprint-newline функция....276
pprint-pop макрос....276
pprint-tab функция....276
pprint-tabluar функция....276
prin1 функция....276
prin1-to-string функция....276
princ функция....276
princ-to-string функция....276
print функция....276
print-not-readable-object функция....276
print-object обобщенная функция....276
print-unreadable-object макрос....276
probe-file функция....267
proclaim функция....220
prog макрос....226
prog* макрос....226
prog1 макрос....226

prog2 макрос....226
progn специальный оператор....226
progv специальный оператор....226
provide функция....278
psetf макрос....226
psetq макрос....226
push макрос....255
pushnew макрос....255
query-io* переменная....284
quote специальный оператор....221
random функция....250
random-state-p функция....250
rassoc функция....256
rassoc-if функция....256
rassoc-if-not функция....256
rational функция....250
rationalize функция....250
rationalp функция....250
read функция....277
read-byte функция....270
read-char функция....270
read-char-no-hang функция....270
read-delimited-list функция....277
read-from-string функция....277
read-line функция....270
read-preserving-whitespace функция....277
read-sequence функция....270
readtable-case функция....277
readtablep функция....277
realp функция....250
realpart функция....250
reduce функция....263
reinitialize-instance обобщенная функция...236
rem функция....250
remf макрос....256
remhash....265
remove функция....263
remove-duplicates функция....263
remove-if функция....263
remove-if-not функция....263
remove-method обобщенная функция....236
remprop функция....242
rename-file функция....267
rename-package функция....244
replace функция....263
require функция....278
rest функция....256
restart-bind макрос....240
restart-case макрос....241
restart-name функция....241
return макрос....226
return-from специальный оператор....226
revappend функция....256
room функция....280
rotatef макрос....226
round функция....250
row-major-aref функция....259
rplaca функция....256
rplacd функция....256
sbit функция....259
scale-float функция....250
schar функция....260
search функция....264
second функция....254
set функция....242
set-difference функция....256
set-dispatch-macro-character функция....278
set-exclusive-or функция....256
set-macro-character функция....278
set-pprint-dispatch функция....276
set-syntax-from-char функция....278
setf макрос....227
setq специальный оператор....227
shadow функция....245
shadowing-import функция....245
shared-initialize обобщенная функция....236
shiftf макрос....227
short-float-epsilon константа....281
short-float-negative-epsilon константа....282
short-site-name функция....280
signal функция....241
signum функция....250
simple-bit-vector-p функция....259
simple-condition-format-arguments функция241
simple-condition-format-control функция....241
simple-string-p функция....260
simple-vector-p функция....259
sin функция....250
single-float-epsilon константа....281
single-float-negative-epsilon константа....282
sinh функция....250
slot-boundp функция....236
slot-exists-p функция....236
slot-makunbound функция 236
slot-missing обобщенная функция....236
slot-unbound обобщенная функция....236
slot-value функция 237
software-type функция....280
software-version функция....280
some функция....227
sort функция....264
special декларация....219
special-operator-p функция....220
sqrt функция....250
stable-sort функция....264
standard-char-p функция....253
step макрос....280
store-value функция....241
stream-element-type функция....270
stream-error-stream функция....270
stream-external-format функция....270
streamp функция....270
string функция....260
string-capitalize функция....260
string-downcase функция....260
string-equal функция....260
string-greaterp функция....260
string-left-trim функция....261
string-lessp функция....261
string-not-equal функция....261
string-not-greaterp функция....261
string-not-lessp функция....261
string-right-trim функция....261
string-trim функция....261
string-upcase функция....260
string/= функция....261
string< функция....261
string<= функция....261
string= функция....261
string> функция....261
string>= функция....261
stringp функция....261
sublis функция....256
subseq функция....264
subsetp функция....256
subst функция....256
subst-if функция....256
subst-if-not функция....256
substitute функция....264
substitute-if функция....264

substitute-if-not функция....264
subtypep функция....221
svref функция....259
sxhash функция....265
symbol функция....242
symbol-macrolet специальный оператор....221
symbol-name функция 242
symbol-package функция....242
symbol-plist функция....242
symbol-value функция....242
synonym-stream-symbol функция....271
t константа....285
tagbody специальный оператор....227
tailp функция....257
tan функция....251
tanh функция....251
tenth функция....254
terpri функция....271
the специальный оператор....221
throw специальный оператор....227
time макрос....280
topevel....14
trace макрос....280
translate-logical-pathname функция....267
translate-pathname функция....267
tree-equal функция....257
truename функция....268
truncate функция....251
two-way-stream-input-stream функция....271
two-way-stream-output-stream функция....271
type декларация....219
type-error-datum функция....221
type-error-expected-type функция....221
type-of функция....221
typecase макрос....227
typep функция....221
unbound-slot-instance функция....237
unexport функция....245
unintern функция....245
union функция....257
unless макрос....227
unread-char функция....271
untrace....280
unuse-package функция....245
unwind-protect специальный оператор....227
update-instance-for-different-class обобщенная функция....237
update-instance-for-redefined-class обобщенная функция....237
upgraded-array-element-type функция....259
upgraded-complex-part-type функция....251
upper-case-p функция....253
use-package функция....245
use-value функция 241
user-homedir-pathname функция....280
values функция....227
values-list функция....227
vector функция....260
vector-pop функция....260
vector-push функция....260
vector-push-extend функция....260
vectorp функция....260
vi 20
warn функция....241
when макрос....228
wild-pathname-p функция....267
with-accessors макрос....237
with-compilation-unit макрос....279
with-condition-restarts макрос....241
with-hash-table-iterator макрос....265
with-input-from-string макрос....271
with-open-file макрос....271
with-open-stream макрос....271
with-output-to-string макрос....271
with-package-iterator макрос....245
with-simple-restart макрос....241
with-slots макрос....237
with-standard-io-syntax макрос....278
write функция....277
write-byte функция....271
write-char функция....271
write-line функция....271
write-sequence функция....271
write-string функция....271
write-to-string функция....277
y-or-n-p функция....272
yes-or-no-p функция....272
zerop функция....251

Книга **ANSI Common Lisp** сочетает введение в программирование на Лиспе и актуальный справочный материал по ANSI-стандарту языка. Новички найдут в ней примеры интересных программ с их тщательным объяснением. Профессиональные разработчики оценят всесторонний практический подход. Книга содержит:

- **Актуальный справочный материал** по ANSI Common Lisp.
- Детальное рассмотрение **объектно-ориентированного программирования**: не только описание CLOS, но и пример собственного объектно-ориентированного языка.
- **Более 20 самостоятельных примеров**, в том числе: трассировщик лучей, бредогенератор, сопоставление с образцом, логический вывод, программу для генерации HTML, алгоритмы поиска и сортировки, файлового ввода-вывода, сжатия данных, а также вычислительных задач.
- Особое внимание **критически важным концепциям**, включая префиксный синтаксис, связь кода и данных, рекурсию, функциональное программирование, типизацию, неявное использование указателей, динамическое выделение памяти, замыкания, макросы, предшествование классов, суть методов обобщенных функций и передачи сообщений.
- Полноценное руководство по **оптимизации**.
- Простое, но всестороннее объяснение **макросов**.
- Примеры различных **стилей программирования**, включая быстрое прототипирование, разработку снизу-вверх, объектно-ориентированное программирование, применение встраиваемых языков.
- Приложение, посвященное **отладке**, с примерами распространенных ошибок.

Понятное, отлично написанное руководство и справочник по азам и продвинутым возможностям ANSI Common Lisp. Это больше чем просто введение в язык – с таким обширным справочным материалом для большинства читателей книга будет неплохой альтернативой дядюшке Стилу.

Ричард Фейтман, Калифорнийский университет в Беркли.

Эта книга, соответствующая стандарту ANSI, будет идеальным учебным пособием.

Джон Фодераро, Franz. Inc.

Пол Грэм снова сделал это. Его первая книга, «On Lisp», содержала превосходное описание ряда продвинутых возможностей Лиспа, а эта предоставляет понятное и обстоятельное введение в язык, включая такие моменты как оптимизация производительности кода.

Томас Читам, Гарвардский университет.

Последняя глава книги восхитительна. В ней на фоне объектно-ориентированного программирования объясняются сразу несколько ключевых идей, она проводит читателя через несколько реализаций объектно-ориентированной системы, постепенно усложняя ее.

Дэвид Турецки, университет Карнеги-Меллон.

На настоящий момент Пол Грэм является независимым консультантом и использует Лисп. Он получил степень PhD в области информатики в Гарвардском университете.