

КАК ПРОЕКТИРОВАТЬ ПРОГРАММЫ

Введение в программирование
и компьютерные вычисления

Маттиас
Фелляйзен

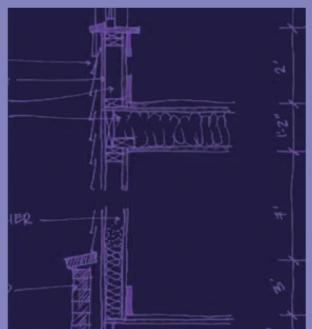
Роберт Брюс
Финдлер

Шрирам
Кришнамурти

Мэтью
Флэтт



АМК
издательство



Маттиас Фелляйзен
Роберт Брюс Финдлер
Мэтью Флэтт
Шрирам Кришнамурти

Как проектировать программы

How to Design Programs

**An Introduction to Programming and
Computing**

Second Edition

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi

The MIT Press
Cambridge, Massachusetts
London, England

Как проектировать программы

Введение в программирование и компьютерные вычисления

Маттиас Фелляйзен
Роберт Брюс Финдлер
Мэттью Флэтт
Шрирам Кришнамурти



Москва, 2022

УДК 004.2

ББК 32.97

Ф37

Фелляйзен М., Финдлер Р. Б., Флэтт М., Кришнамурти Ш.

Ф37 Как проектировать программы / пер. с англ. А. Н. Киселева; под ред. П. Б. Иванова, А. Д. Чичигина, Ю. А. Сыровецкого, С. В. Бронникова. – М.: ДМК Пресс, 2022. – 724 с.: ил.

ISBN 978-5-97060-926-2

Эта книга повествует о методах «хорошего программирования» – то есть о таком подходе к созданию программного обеспечения, который опирается на системное мышление, планирование и понимание задач разработчика на каждом этапе. В числе рассматриваемых тем – фундаментальные понятия систематического проектирования, типы данных, способы записи объемных данных, создание и использование абстракций, тестирование программ и функций и др.

Издание адресовано профессионалам и энтузиастам программирования, не имеющим прежнего опыта систематического проектирования программ, а также преподавателям технических вузов, которые могут использовать представленный материал в рамках учебного курса.

УДК 004.2

ББК 32.97

The rights to the russian language edition obtained through Alexander Korgzhenevski Agency (Moscow). All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-0-26253-480-2 (англ.)

ISBN 978-5-97060-926-2 (рус.)

© Massachusetts Institute of Technology, 2018

Illustrations © Torrey Butzer, 2000

© Перевод, оформление, издание, ДМК Пресс, 2022

Содержание

От редакторов	10
От издательства	11
Вступление	12
Пролог: как писать программы	29
I ДАННЫЕ ФИКСИРОВАННОГО РАЗМЕРА	55
1 Арифметика.....	56
1.1. Арифметика чисел	57
1.2. Арифметика строк.....	59
1.3. А теперь все смещаем	61
1.4. Арифметика изображений	63
1.5. Арифметика логических значений.....	66
1.6. Смешанные операции с логическими значениями	67
1.7. Предикаты: знай свои данные.....	69
2 Функции и программы.....	72
2.1. Функции.....	72
2.2. Вычисления	76
2.3. Композиция функций.....	80
2.4. Глобальные константы.....	83
2.5. Программы	85
3 Как проектировать программы	98
3.1. Проектирование функций	99
3.2. Практические упражнения: функции	106
3.3. Знание предметной области	106
3.4. От функций к программам	107
3.5. О тестировании	108
3.6. Проектирование интерактивных программ	110
3.7. Миры виртуальных питомцев	120
4 Интервалы, перечисления и детализация	122
4.1. Программирование с условиями	122
4.2. Условные вычисления	124
4.3. Перечисления	127
4.4. Интервалы	131
4.5. Детализация	135
4.6. Проектирование с использованием детализации	143
4.7. Миры с конечными состояниями.....	146
5 Добавляем структуру	154
5.1. От позиций к структурам posn	154
5.2. Вычисления со структурами posn	155
5.3. Программирование с posn	156
5.4. Определение структурных типов.....	158
5.5. Вычисления со структурами	163
5.6. Программирование со структурами	167
5.7. Вселенная данных	174

5.8. Проектирование с использованием структур	178
5.9. Структура в мире.....	181
5.10. Графический редактор.....	182
5.11. Больше виртуальных питомцев	184
6 Структуры и детализация	187
6.1. Проектирование с использованием детализации, снова.....	187
6.2. Смешивание миров.....	200
6.3. Ошибки ввода.....	203
6.4. Проверка состояния мира.....	207
6.5. Предикаты равенства.....	209
7 Итоги.....	211
Интермеццо 1. Язык для начинающих студентов	212
Словарь BSL.....	212
Грамматика BSL	213
Значение в языке BSL	217
Значения и вычисления	220
Ошибки в BSL.....	220
Логические выражения	223
Определения констант	224
Определения структур.....	226
Тесты в BSL	228
Сообщения об ошибках в BSL	229
II ДАННЫЕ ПРОИЗВОЛЬНОГО РАЗМЕРА.....	237
8 Списки	238
8.1. Создание списков.....	238
8.2. Что такое '(), что такое cons	243
8.3. Программирование со списками	245
8.4. Вычисления со списками.....	249
9 Проектирование с определениями данных, ссылающимися на самих себя	251
9.1. Практические упражнения: списки	258
9.2. Непустые списки	260
9.3. Натуральные числа	266
9.4. Русская матрёшка.....	270
9.5. Списки в интерактивных программах	274
9.6. Замечания о списках и множествах.....	279
10 Еще о списках.....	284
10.1. Функции, создающие списки	284
10.2. Структуры в списках	287
10.3. Списки в списках, файлы	291
10.4. И снова о графическом редакторе	300
11 Проектирование методом композиции	312
11.1. Функция list	312
11.2. Композиция функций	314

11.3. Повторяющиеся вспомогательные функции	316
11.4. Обобщающие вспомогательные функции.....	323
12 Проекты: списки	333
12.1. Реальные данные: словари	333
12.2. Реальные данные: iTunes.....	335
12.3. Игры со словами, иллюстрация приема композиции	340
12.4. Игры со словами, суть проблемы	345
12.5. «Питон»	347
12.6. Простой «Тетрис»	350
12.7. Полная игра «Космические захватчики»	353
12.8. Конечные автоматы	354
13 Итоги.....	362
Интермеццо 2. Quote, unquote	364
Цитирование	364
Квазицитирование и антицитирование	365
Объединение с антицитированием.....	370
III АБСТРАКЦИИ.....	375
14 Сходства повсюду.....	376
14.1. Сходства в функциях.....	376
14.2. Отличающиеся сходства	378
14.3. Сходства в определениях данных	381
14.4. Функции – это значения	384
14.5. Вычисления с функциями	385
15 Проектирование абстракций	389
15.1. Абстрагирование примеров	389
15.2. Сходства в сигнатурах.....	394
15.3. Единая точка управления	399
15.4. Абстрагирование макетов	400
16 Использование абстракций.....	402
16.1. Имеющиеся абстракции	403
16.2. Локальные определения	405
16.3. Локальные определения добавляют выразительности	409
16.4. Вычисления с локальными определениями.....	411
16.5. Использование абстракций на примерах.....	415
16.6. Проектирование с использованием абстракций	420
16.7. Практические упражнения: абстракция	422
16.8. Проекты: абстракция	423
17 Безымянные функции.....	426
17.1. Определение функций с помощью лямбда-выражений.....	427
17.2. Вычисления с лямбда-выражениями	429
17.3. Абстрагирование с помощью лямбда-выражений.....	432
17.4. Определение спецификаций с помощью лямбда-выражений	435
17.5. Представление с помощью лямбда-выражений	442
18 Итоги.....	447

Интермеццо 3. Область видимости и абстракции.....	448
Область видимости	448
Циклы в языке ISL.....	453
Сопоставление с образцом.....	461
IV ПЕРЕПЛЕТАЮЩИЕСЯ ДАННЫЕ.....	468
19 Поэзия S-выражений	469
19.1. Деревья.....	469
19.2. Леса.....	477
19.3. S-выражения.....	479
19.4. Проектирование с использованием взаимосвязанных данных	485
19.5. Проект: BST	487
19.6. Упрощение функций	491
20 Итеративное уточнение.....	494
20.1. Анализ данных	494
20.2. Уточнение определений данных.....	496
20.3. Уточнение функций	498
21 Уточнение интерпретатора	501
21.1. Интерпретация выражений.....	501
21.2. Интерпретация переменных.....	504
21.3. Интерпретация функций	507
21.4. Интерпретация всего и вся.....	509
22 Проект: обработка XML.....	512
22.1. XML как S-выражения.....	512
22.2. Отображение XML-перечислений.....	518
22.3. Предметно-ориентированные языки.....	523
22.4. Чтение XML.....	528
23 Одновременная обработка	533
23.1. Одновременная обработка двух списков: случай 1	533
23.2. Одновременная обработка двух списков: случай 2	534
23.3. Одновременная обработка двух списков: случай 3	537
23.4. Упрощение функций	541
23.5. Проектирование функций с двумя сложными аргументами	542
23.6. Практические упражнения: два аргумента	544
23.7. Проект: база данных.....	548
24 Итоги.....	560
Интермеццо 4. Природа чисел	561
Арифметика с числами фиксированного размера.....	561
Переполнение	567
Потеря значимости.....	567
Числа в *SL.....	568
V ГЕНЕРАТИВНАЯ РЕКУРСИЯ.....	574
25 Нестандартная рекурсия	575
25.1. Рекурсия без структуры	575

25.2. Рекурсия, игнорирующая структуру	579
26 Проектирование алгоритмов	585
26.1. Адаптация рецепта проектирования.....	585
26.2. Завершимость рекурсии	587
26.3. Структурная и генеративная рекурсии.....	590
26.4. Выбор	591
27 Вариации на тему	597
27.1. Фракталы, первое знакомство	597
27.2. Бинарный поиск	600
27.3. Синтаксический анализ	606
28 Математические примеры	610
28.1. Метод Ньютона.....	610
28.2. Интегрирование	614
28.3. Проект: гауссово исключение.....	621
29 Алгоритмы с возвратами	627
29.1. Обход графов	627
29.2. Проект: возврат	636
30 Итоги.....	643
Интермеццо 5. Стоимость вычислений.....	644
Конкретное время, абстрактное время	645
Определение термина «порядка»	651
Почему программы используют предикаты и селекторы?.....	654
VI АККУМУЛЯТОРЫ	658
31 Потеря знаний	659
31.1. Проблема структурной обработки	659
31.2. Проблема генеративной рекурсии.....	663
32 Проектирование функций с аккумулятором	668
32.1. Условия применения аккумулятора.....	668
32.2. Добавление аккумуляторов	670
32.3. Преобразование простых функций в функции с аккумуляторами	672
32.4. Графический редактор с поддержкой мыши	684
33 Дополнительные примеры использования аккумуляторов	687
33.1. Аккумуляторы и деревья	687
33.2. Представления данных с аккумуляторами.....	693
33.3. Аккумуляторы как результаты	699
34 Итоги.....	706
Эпилог: что дальше	708
Предметный указатель	714

От редакторов

Добрый день! Спасибо, что вы открыли эту книгу, даже если взяли с полки магазина просто посмотреть. Нам приятно в любом случае. Для нас удача и честь принять участие в переводе такой легендарной книги, чтобы вы могли прочесть ее на родном языке.

Что же делает ее легендарной? Вы наверняка заметили, что это перевод *второго* издания, которое вышло уже почти 5 лет назад, а первое – более 20 лет назад. Оставаться востребованным столько лет в стремительно меняющемся мире информационных технологий и языков программирования – достижение само по себе.

Авторы книги – легенды мира компьютерных наук и информатики, авторы фундаментальных работ на протяжении последних 30 лет. Они же являются одними из пионеров систематического, научного подхода к *преподаванию* программирования и информатики. Как результат, данная книга является отражением глубокого понимания как самого предмета, так и методики его преподавания, сформировавшегося у авторов за десятилетия практики.

Другой фактор долголетия этого учебника – то, что он *живой*. Книга продолжает развиваться даже после публикации: в ней постоянно вносят правки и уточнения, исправляют ошибки, она используется в качестве основы для курсов в университетах и на онлайн-площадках.

В результате получился один из лучших в мире учебников по программированию для новичков, который увлекательно читать, который никогда не перегружает лишними деталями, всегда ясно указывает направление развития и ненавязчиво прививает правила «хорошего тона».

Авторы справедливо отмечают, что это учебник по чему-то большему, чем просто программирование. В первую очередь он учит тому, что в последнее время принято называть «computational thinking», «алгоритмическим (или вычислительным) мышлением». Этим стилем мышления пользуются не только программисты, но и повара, учителя, спортсмены, врачи и многие, многие другие.

Авторы книги скромничают, когда говорят, что книга ориентирована на начинающих. Профессионалы встретят в ней как отсылки к фундаментальным понятиям и методам компьютерных наук, так и эффективные методы анализа повседневных задач. Поэтому от всего сердца советуем вам купить эту книгу! Мы от этого богаче не станем, но верим, что вас она способна обогатить интеллектуально, творчески и профессионально, даже если разработка программного обеспечения – не ваша основная деятельность.

С уважением от редакторов,
Павел Борисович Иванов,
Александр Дмитриевич Чичигин,
Юрий Алексеевич Сыровецкий,
Сергей Викторович Бронников

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и The MIT Press очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Вступление

Многие современные профессии требуют умения программировать в той или иной форме. Бухгалтеры программируют электронные таблицы; музыканты программируют синтезаторы; писатели программируют текстовые процессоры; а веб-дизайнеры программируют таблицы стилей. Когда мы писали эти слова для первого издания книги (1995–2000), читатели могли счесть их футуристическими, однако к настоящему времени умение программировать стало обязательным, и появились многочисленные книги, онлайн-курсы и предметы в общеобразовательной школе, которые удовлетворяют эту потребность и улучшают шансы людей на трудоустройство.

Типичный курс программирования учит подходу «Пробуй, пока не заработает». Добившись нужного результата, учащийся восклицает: «Работает!» – и идет дальше. К сожалению, эта фраза также является самой короткой небылицей в информатике и многим людям стоила многих часов их жизни. Эта книга, напротив, фокусируется на навыках *хорошего программирования* и адресована всем – и профессиональным программистам, и любителям.

Под «хорошим программированием» мы подразумеваем подход к созданию программного обеспечения, который изначально опирается на системное мышление, планирование и понимание на каждом этапе и на каждом шаге. Чтобы подчеркнуть это, мы говорим о *системном проектировании программ* и *системно спроектированных программах*. Что особенно важно, последнее словосочетание ясно выражает требование к желаемой функциональности. Хорошее программирование также удовлетворяет эстетическое чувство выполненного долга; хорошая программа по своей элегантности сравнима с хорошими стихами или черно-белыми фотографиями ушедшей эпохи. Проще говоря, программирование отличается от хорошего программирования как наброски карандашом на салфетке, сделанные в закусочной, от картин маслом в музее.

Нет, эта книга не превратит вас в мастера живописи. Но мы не стали бы тратить пятнадцать лет на подготовку данного издания, если бы не верили, что

каждый может разрабатывать программы

и

каждый может испытывать удовлетворение от творческого процесса.

Более того, мы утверждаем, что

проектирование программ – но не программирование – в традиционном для Запада высшем образовании должно стоять рядом с математикой и лингвистикой.

Студент, изучающий проектирование, который никогда больше не коснется программ, все равно приобретет универсально полезные

навыки решения задач, приобретет опыт творческой деятельности и научится ценить новую форму эстетики. Остальная часть этого вступления подробно объясняет, что мы имеем в виду под «системным проектированием», кому и чем это выгодно и как мы обучаем всему этому.

Системное проектирование программ

Программа взаимодействует с людьми, которых называют *пользователями*, и другими программами, которые могут быть *серверами* или *клиентами*. Соответственно, любая более или менее полная программа состоит из множества строительных блоков, одни из которых обрабатывают ввод, другие производят вывод, а третьи соединяют первые со вторыми. В качестве фундаментальных строительных блоков мы предпочитаем использовать функции, потому что все мы хорошо знакомы с функциями по курсу школьной математики и потому что простейшие программы являются именно такими функциями. Главное – выяснить, какие функции необходимы, как их соединить между собой и как их сконструировать из основных ингредиентов.

В этом контексте «системное проектирование программ» означает сочетание двух составляющих: рецептов проектирования и итеративного уточнения. Рецепты проектирования – это изобретение авторов, обеспечивающее возможность итеративного уточнения.

Рецепты проектирования применимы как к целым программам, так и к отдельным функциям. В этой книге есть всего два рецепта для целых программ: один для программ с графическим пользовательским интерфейсом (Graphical User Interface, GUI) и другой для неинтерактивных программ. Рецепты проектирования функций, напротив, намного разнообразнее: для данных атомарных типов, таких как числа; для перечислений разных видов данных; для данных, которые фиксированным образом объединяют другие данные; для конечных, но произвольно больших данных; и так далее.

Рецепты проектирования для функций объединяются общим **процессом проектирования**. В списке в рецепте 1 перечислены шесть основных шагов этого процесса. Название каждого шага сообщает ожидаемый результат(ы), а «команды» определяют ключевые действия. Центральную роль в этом процессе играют примеры. Для представления данных, выбранного на шаге 1, примеры иллюстрируют, как фактическая информация кодируется в данные и как данные интерпретируются в информацию. В шаге 3 говорится, что человек, решающий задачу, должен проработать конкрет-

Мы черпали вдохновение в методе, предложенном Майклом Джексоном (Michael Jackson) для создания программ на языке COBOL, а также в беседах с Дэниелом Фридманом (Daniel Friedman) о рекурсии, Робертом Харпером (Robert Harper) о теории типов и Дэниелом Джексоном (Daniel Jackson) о проектировании программного обеспечения.

Преподавателям.
Попросите учащихся скопировать рецепт 1 на картонную карточку. Когда у кого-то из них возникнет проблема, попросите их предъявить карточку и указать шаг, на котором они застряли.

ные сценарии, чтобы понять, что должна вычислить функция в каждом конкретном примере. Это понимание используется на шаге 5, когда наступает время определения функции. Наконец, шаг 6 требует, чтобы примеры были преобразованы в автоматизированные тесты, проверяющие правильность работы функции в некоторых случаях. Запуск функции на реальных данных может выявить другие расхождения между ожиданиями и результатами.

Рецепт 1. Базовый рецепт проектирования функций

- 1. Анализ задачи и определение данных.** Определите, какая информация и как должна быть представлена в выбранном языке программирования. Сформулируйте определения данных и проиллюстрируйте их примерами.
- 2. Сигнатура, описание назначения, заголовок.** Укажите, какие данные функция принимает и выдает. Сформулируйте краткий ответ на вопрос: «что вычисляет функция?» Определите заглушку с соответствующей сигнатурой.
- 3. Примеры использования функции.** Представьте примеры, иллюстрирующие назначение функции.
- 4. Создание макета функции.** Используя определения данных, напишите набросок функции.
- 5. Определение функции.** Заполните недостающие части в макете функции. Используйте описание назначения и примеры.
- 6. Тестирование.** Переформулируйте примеры в тесты и убедитесь, что функция успешно выполняет их все. Это поможет обнаружить вкравшиеся ошибки. Кроме того, тесты дополнят примеры и помогут другим понять определение функции, если в этом возникнет необходимость, а она всегда возникает в любой серьезной программе.

Преподавателям.

Наиболее важные вопросы возникают на шагах 4 и 5. Попросите учащихся записать эти вопросы своими словами на обратной стороне картонных карточек.

На каждом шаге процесса проектирования возникают вопросы. На некоторых из них, например на шаге создания примеров использования функции или на шаге создания макета, вопросы могут затрагивать определение данных. Ответы на них почти автоматически создают промежуточный продукт. Затраты на эти подготовительные шаги окупиваются, когда приходит время сделать творческий шаг – завершить определение функции, потому что оказывают необходимую помощь во всех случаях.

Необычность этого подхода заключается в создании новичками промежуточного продукта. Когда новичок застрянет на каком-то шаге, эксперт или преподаватель сможет проверить созданные промежуточные продукты, задать наводящие вопросы, характерные для процесса проектирования, и тем самым помочь новичку преодолеть затруднение. Этот аспект самообразования является коренным отличием проектирования программ от программирования.

Итеративное уточнение решает проблему сложности и многосторонности задач. Сделать все и сразу практически невозможно. Для решения этой проблемы специалисты по информатике заимствовали метод итеративного уточнения из естественных наук. Согласно методу итеративного уточнения рекомендуется сначала отбросить все несущественные детали и найти решение основной задачи. Затем шаг уточнения добавляет одну из отброшенных деталей, и расширенная задача решается повторно с максимальным использованием существующего решения. Повторение этих шагов уточнения и поиска решения в конечном итоге приводит к полному решению.

В этом смысле программист является своего рода ученым. Ученые создают приблизительные модели идеализированной версии мира, чтобы получить возможность делать прогнозы. Пока прогнозы сбываются, модель используется как есть; когда прогнозы начинают отличаться от реальности, ученые пересматривают свои модели, стремясь уменьшить расхождения. Точно так же, когда программист получает задание, он создает первую модель, превращает ее в код, оценивает с помощью пользователей и итеративно уточняет модель, пока поведение программы не будет достаточно точно соответствовать желаемому.

В этой книге мы раскрываем итеративное уточнение с двух сторон. Поскольку проектирование методом итеративного уточнения становится особенно полезным при проектировании сложных программ, книга подробно знакомит с этой техникой в четвертой части, когда рассматриваемые задачи достигают определенной степени сложности. Кроме того, в первых трех частях итеративное уточнение используется для формулирования все более сложных вариантов одной и той же задачи. То есть в одной главе мы берем базовую задачу, решаем еще в одной главе, а в следующей главе ставим аналогичную задачу, но с дополнительными деталями, отражающими новые, свежие, только что введенные понятия.

DrRacket и учебные языки

Чтобы научиться проектировать программы, нужно постоянно практиковаться. Как нельзя стать пианистом, не играя на пианино, так же нельзя стать разработчиком программ, не создавая программ и не доводя их до корректного поведения. По этой причине наша книга сопровождается некоторой программной поддержкой: языком, на котором можно писать программы, и *средой разработки программ*, в которой программы редактируются как текстовые документы и с помощью которой можно запускать программы.

Многие, встречавшиеся нам и говорившие, что хотели бы научиться программированию, спрашивали, какой язык программирования им следует выучить. Учитывая рекламную шумиху, развернутую вокруг некоторых языков программирования, такой вопрос не вызывает удивления. Но

Преподавателям.
На курсах, предназначенных для опытных разработчиков, можно использовать другой подходящий язык.

проблема в том, что он совершенно неуместен. Обучение программированию на языке, модном в настоящее время, часто приводит обучающихся к неудачам. Мода в этом мире очень недолговечна. Типичная книга «Короткий курс программирования на X» не может научить принципам, которые будут перенесены на следующий модный язык. Хуже того, сам язык часто отвлекает от приобретения передаваемых навыков на уровне формулирования решений и на уровне исправления ошибок.

Обучение проектированию программ, напротив, заключается прежде всего в изучении принципов и приобретении передаваемых навыков. Идеальный язык программирования должен поддерживать обе эти цели, чего нельзя сказать ни об одном стандартном промышленном языке. Ключевая проблема в том, что новички совершают ошибки еще до того, как более или менее существенно овладевают языком, однако языки программирования диагностируют эти ошибки, как если бы программист уже знал все его тонкости. В результате сообщения об ошибках часто ставят новичков в тупик.

Наше решение: начать со знакомства с нашим собственным

Преподавателям. Вы можете объяснить, что *BSL* – это школьная алгебра с дополнительными формами данных и множеством предопределенных функций для работы с ними. нынешним специализированным языком обучения, который мы назвали «язык для начинающих» (*Beginning Student Language*, *BSL*). По сути, это почти тот же «иностранный» язык, который изучается в школьном курсе математики. Он включает обозначения для определений функций, применения функций и условных выражений. Кроме того, он допускает вложенность выражений. То есть этот язык настолько мал, что диагностика ошибок в терминах всего языка доступна читателям, у которых нет никаких знаний, кроме начального курса математики.

Учащийся, овладевший принципами структурного проектирования, может затем перейти к «языку для учащихся промежуточной сложности» (*Intermediate Student Language*, *ISL*) и другим продвинутым диалектам с групповым названием **SL*. В книге эти диалекты используются для обучения принципам абстракции и рекурсии. Мы твердо уверены, что использование такой последовательности обучающих языков позволяет читателям подготовить себя к созданию программ на широком спектре профессиональных языков программирования (*JavaScript*, *Python*, *Ruby*, *Java* и др.).

ПРИМЕЧАНИЕ. Обучающие языки реализованы на *Racket*, языке программирования для создания языков программирования. *Racket* ускользнул из лаборатории в реальный мир и постепенно стал применяться для решения самых разных задач, от создания игр до реализации управления массивами телескопов. Обучающие языки заимствуют некоторые элементы из языка *Racket*, но эта книга **не** учит программированию на *Racket*. Однако учащийся, прочитавший эту книгу, с легкостью сможет начать программировать на *Racket*. **КОНЕЦ.**

Выбирай среду программирования, мы оказываемся в такой же плохой ситуации, как при выборе языка программирования. Среда

программирования для профессионалов подобна кабине современного реактивного самолета. Она имеет множество элементов управления и дисплеев, непостижимых для любого, кто впервые запускает такое программное приложение. Начинающим программистам нужен эквивалент двухместного одномоторного поршневого самолета, на котором они могут практиковать базовые навыки. Поэтому мы создали среду программирования DrRacket для новичков.

DrRacket поддерживает очень увлекательное обучение с обратной связью с использованием всего двух простых интерактивных панелей: области определений, содержащей определения функций, и области взаимодействия, которая позволяет программисту запросить вычисление выражений, связанных с определениями. В этом контексте исследовать сценарии «что, если» так же легко, как в приложении для работы с электронными таблицами. Экспериментирование можно начать при первом же контакте, используя обычные примеры в стиле калькулятора и быстро переходя к вычислениям с изображениями, словами и другими формами данных.

Интерактивная среда разработки программ, такая как DrRacket, упрощает процесс обучения. Во-первых, она позволяет начинающим программистам напрямую манипулировать данными. Поскольку нет никаких средств для чтения входной информации из файлов или устройств хранения, новичкам не нужно тратить драгоценное время на выяснение особенностей их использования. Во-вторых, среда разработки четко отделяет данные и операции с ними от ввода и вывода информации из «реального мира». В настоящее время это разделение считается настолько фундаментальным для системного проектирования программного обеспечения, что получило собственное название: *архитектура модель-представление-контроллер* (Model-View-Controller, MVC). Работая в DrRacket, начинающие программисты естественным путем знакомятся с этой фундаментальной идеей программной инженерии.

Применение навыков

Приобретенные в процессе обучения навыки проектирования программ находят системное применение в двух направлениях: в программировании вообще и в программировании электронных таблиц, синтезаторов, таблиц стилей и даже текстовых процессоров в частности. Как показывают наши наблюдения, процесс проектирования, представленный в рецепте 1, легко перенести практически на любой язык программирования и можно использовать для разработки как коротких, насчитывающих десяток строк, так и длинных программ, состоящих из десятков тысяч строк кода. Конечно, необходимо некоторое время, чтобы осмыслить и адаптировать процесс проектирования ко всему спектру языков и к разным масштабам программ; но как

только он станет второй натурой, его применение становится естественным и начинает приносить выгоду.

Обучение проектированию программ также означает обретение двух универсальных навыков. Проектирование программ, безусловно, дает те же аналитические навыки, что и математика, особенно алгебра и геометрия. Но, в отличие от математики, работа с программами – это активный подход к обучению. Процесс создания программного обеспечения включает немедленную обратную связь и тем самым способствует исследованиям, экспериментам и самооценке. Результатом, как правило, являются интерактивные продукты, создание которых дает более мощное чувство удовлетворенности, чем решение упражнений в учебниках.

Проектирование программ тренирует не только математические навыки, но также навыки чтения и письма. Даже самые маленькие задачи проектирования формулируются в текстовом виде. Без прочных навыков чтения и понимания прочитанного невозможно проектировать программы, которые решают более или менее сложные задачи. И наоборот, методы проектирования программ заставляют разработчика излагать свои мысли правильным и точным языком. Фактически, усваивая рецепт проектирования, учащийся одновременно совершенствует свои навыки артикуляции.

Для иллюстрации взгляните еще раз на описание процесса проектирования в рецепте 1. В нем говорится, что проектировщик должен:

- 1) проанализировать постановку задачи, обычно обозначаемую словом «задача»;
- 2) извлечь и абстрактно выразить ее суть;
- 3) проиллюстрировать суть примерами;
- 4) определить макет на основе этого анализа;
- 5) получить результаты и сопоставить их с ожиданиями;
- 6) доработать продукт с учетом неудачных проверок и тестов.

Каждый шаг требует анализа, описания, точности, сосредоточенности и внимания к деталям. Любой опытный предприниматель, инженер, журналист, юрист, ученый и любой другой профессионал сможет подтвердить, насколько эти навыки важны в повседневной работе. Практика проектирования программ – на бумаге и в DrRacket – это приятный способ приобретения навыков.

Точно так же совершенствование проекта не ограничивается информатикой и созданием программ. Этим занимаются и архитекторы, и композиторы, и писатели, и другие профессионалы. Они начинают с идей в голове и каким-то образом формулируют их суть. Они уточняют идеи на бумаге, пока продукт не будет максимально точно отражать мысленное представление. Воплощая идеи на бумаге, они используют навыки, аналогичные рецептам проектирования: рисование, письмо или игра на музыкальном инструменте, чтобы выразить определенные элементы стиля здания, описать характер че-

ловека или сформулировать элементы мелодии. Их продуктивность в итеративном процессе разработки обусловлена знанием и умением применять базовые рецепты проектирования в своей сфере и правильно выбирать наиболее подходящий для текущей ситуации.

Структура книги

Цель этой книги – познакомить читателей, не имеющих практического опыта, с *системным проектированием программ*. Параллельно она знакомит с *символическим представлением вычислений* – методом объяснения, как работает применение программы к данным. Проще говоря, этот метод обобщает сведения по арифметике и алгебре, которые учащиеся получают в начальной и средней школах соответственно. Но пусть вас это не пугает. В DrRacket имеется механизм пошаговых вычислений, способный иллюстрировать такие вычисления по шагам.

Книга состоит из шести частей, разделенных пятью интермеццо, и обрамляется прологом и эпилогом. Основные части посвящены проектированию программ, а промежуточные интермеццо вводят дополнительные понятия, касающиеся механики программирования и вычислений.

Пролог – это краткое введение в простое программирование. В нем объясняется, как реализовать простую анимацию на *SL. Прочитав его, любой новичок почувствует воодушевление и подавленность одновременно. Поэтому в последнем примечании объясняется, почему обычное программирование – это ошибочный путь, и как системный и последовательный подход к проектированию программ устраниет чувство страха, которое обычно испытывает каждый начинающий программист.

За прологом следуют основные части книги.

- **Часть I** описывает наиболее фундаментальные понятия системного проектирования на простых примерах. Основная идея заключается в том, что проектировщики обычно имеют некоторое представление о том, какие данные программа должна принимать и производить. По этой причине при системном подходе к проектированию необходимо извлечь как можно больше подсказок из описания данных, поступающих в программу и исходящих из нее. Для простоты эта часть начинается с атомарных данных – чисел, изображений и т. д., – а затем постепенно вводит новые способы описания данных: интервалы, перечисления, структуры и их комбинации.
- **Интермеццо 1** подробно описывает язык обучения: его словарь, грамматику и значение. Все вместе это обычно называют синтаксисом и семантикой. Разработчики программ используют эту модель вычислений для прогнозирования действий их творения после запуска или для анализа ошибок.

- **Часть II** дополняет часть I средствами описания наиболее интересных и полезных форм данных: составных данных произвольного размера. Программист может продолжать использовать типы данных из части I для представления информации, но эти типы всегда имеют фиксированную глубину и ширину. Эта часть демонстрирует, как небольшое обобщение позволяет перейти к данным произвольного размера. Затем мы переключим свое внимание на системное проектирование программ, обрабатывающих такие данные.
- **Интермеццо 2** вводит краткую и мощную нотацию для записи больших объемов данных: цитирование и антицитирование.
- **Часть III** наглядно демонстрирует сходство многих функций из части II. Никакой язык программирования не должен заставлять программистов создавать фрагменты кода, настолько похожие друг на друга. И наоборот, во всяком хорошем языке программирования есть способы устранения подобного сходства. Ученые-информатики называют этап устранения сходства *абстрагированием*, а его результат – *абстракцией*. Они знают, что абстракции значительно повышают продуктивность программиста. По этой причине в данной части будут представлены рецепты создания и использования абстракций.
- **Интермеццо 3** преследует две цели. Во-первых, здесь вводится понятие *лексической области видимости*, когда язык программирования связывает каждое вхождение имени с его определением, которое программист может найти, просматривая код. Во-вторых, объясняется суть библиотеки с дополнительными механизмами абстракции, включая так называемые *циклы for*.
- **Часть IV** обобщает часть II и явно вводит идею итеративного уточнения в словарь понятий проектирования.
- **Интермеццо 4** объясняет и иллюстрирует, почему десятичные числа работают таким странным образом во всех языках программирования. Представленные здесь основные факты должен знать каждый начинающий программист.
- **Часть V** добавляет новый принцип дизайна. Структурного проектирования и абстракции вполне достаточно для решения большинства задач, с которыми сталкиваются программисты, но иногда этого мало для создания «производительных» программ. То есть программам, созданным с применением принципов структурного проектирования, может потребоваться слишком много времени или энергии для вычисления желаемых ответов. Поэтому ученые-информатики заменяют такие программы, созданные с применением принципов структурного проектирования, программами, способными извлекать выгоду из глубокого понимания предметной области. В этой части книги показано, как спроектировать большой класс именно таких программ.

- **Интермеццо 5** использует примеры из части V для иллюстрации представлений ученых-информатиков о производительности.
- **Часть VI** добавляет последний трюк в инструментарий проектировщиков: аккумуляторы. Если говорить упрощенно, аккумулятор добавляет «память» в функции. Добавление памяти значительно улучшает производительность функций из первых четырех частей книги, созданных с применением принципов структурного проектирования. Для специальных программ из части V аккумуляторы могут даже гарантировать нахождение ответа.
- **Эпилог: что дальше** – это одновременно оценка пройденного и взгляд в будущее.

Читатели, занимающиеся самообразованием самостоятельно, должны проработать всю книгу, от первой до последней страницы. Под словом «проработать» мы подразумеваем, что они должны решить все упражнения или, по крайней мере, знать, как их решать.

Преподаватели тоже должны охватить как можно больше, начиная с пролога и заканчивая эпилогом. Наш опыт преподавания показывает, что это выполнимо. Как правило, мы организуем наши курсы так, чтобы слушатели в течение семестра писали большую и увлекательную программу. Однако мы понимаем, что могут быть обстоятельства, диктующие значительные сокращения, и вкусы некоторых преподавателей требуют иных способов использования книги.

На рис. 1 изображена навигационная схема для тех, кто предпочитает получать знания выборочно. Эта схема представляет собой граф зависимостей. Сплошная стрелка от одного элемента к другому предполагает обязательный порядок чтения; например, прежде чем перейти к части II, обязательно нужно изучить часть I. Пунктирные стрелки, напротив, обозначают предлагаемый маршрут; например, читать пролог перед остальной частью книги необязательно.

Вот три возможных пути изучения книги, которые можно предложить, основываясь на этой схеме:

- преподаватель старшей школы может пройти с учениками (насколько это возможно) части I и II, включая небольшой проект;
- преподаватель университета с квартальной системой обучения может сосредоточиться на части I, части II, части III и части V, а также интермеццо по *SL и области видимости;
- преподаватель университета с семестровой системой обучения может предпочесть как можно раньше охватить компромиссы производительности в проектировании. В этом случае мы можем порекомендовать изучить части I и II, а затем раздел об аккумуляторах в части VI, который не зависит от части V. После этого можно углубиться в интермеццо 5 и затем охватить остальную часть книги.

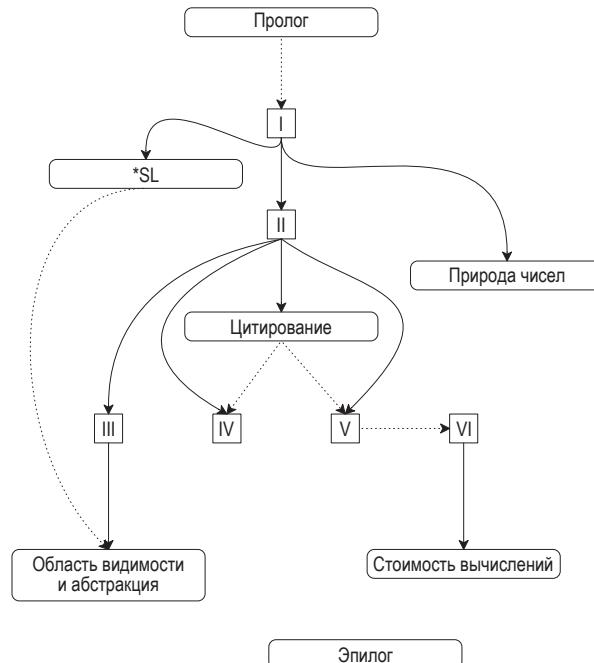


Рис. 1. Зависимости между частями книги и интермеццо

Повторение примеров и упражнений. Повествование в книге снова и снова возвращается к определенным упражнениям и примерам. Например, виртуальные домашние животные встречаются повсюду в части I и иногда даже в части II. Точно так же в обеих частях, I и II, рассматриваются альтернативные подходы к реализации интерактивного текстового редактора. В части V появляются графы, которые перекочевывают в часть VI. Цель этих повторений – последовательное уточнение и закрепление изученного. Мы призываем преподавателей использовать в своей работе эти примеры и упражнения или создать свои подобные последовательности.

Различия между изданиями

Второе издание «Как проектировать программы» имеет несколько важных отличий от первого издания:

- 1) подчеркивает разницу между проектированием всей программы и отдельных функций, составляющих ее. В частности, в этом издании основное внимание уделяется программам двух типов: управляемым событиями (с графическим интерфейсом и сетевым) и неинтерактивным;
- 2) проектирование программы на этапе планирования осуществляется сверху вниз, а на следующем за ним этапе построения –

снизу вверх. Мы явно показываем, как интерфейсы библиотек определяют форму элементов программы. В частности, на самом первом этапе проектирования программы создается список желаемых функций. Да, идея списка желаний присутствует и в первом издании, но во втором издании она рассматривается как явный элемент дизайна;

- 3) выполнимость пункта из списка желаний зависит от рецепта проектирования функции, о чем рассказывается в шести основных частях книги;
- 4) ключевым элементом структурного проектирования является определение функций, являющихся композицией других функций. Такая композиционная организация особенно полезна в мире неинтерактивных программ. Как и порождающая рекурсия, для правильной композиции требуется *озарение* и признание того факта, что последовательная обработка промежуточных результатов несколькими функциями упрощает общий процесс проектирования. Этот подход тоже требует составить список желаний, но при формулировании этих желаний необходимо тщательно проработать определения промежуточных данных. Это издание книги включает ряд явных упражнений по композиционному проектированию;
- 5) тестирование всегда было частью нашей философии проектирования, однако языки обучения и DrRacket начали обеспечивать достаточно полную его поддержку только в 2002 году, уже после выхода первого издания. Данное новое издание в значительной степени полагается на эту поддержку тестирования;
- 6) из этого издания мы исключили тему проектирования императивных программ. Старые главы можно найти на нашем сайте¹, а их адаптированные версии войдут во второй том данной серии «How to Design Components»;
- 7) в этом издании изменены обучающие пакеты с примерами и упражнениями. Предпочтительным стилем связывания этих библиотек считается применение инструкции `require`, но вы все еще можете добавлять их через меню в DrRacket;
- 8) наконец, во втором издании несколько изменились терминология и обозначения:

Мы благодарим Кэти Фислер (Kathi Fisler) за то, что обратила наше внимание на этот момент.

Второе издание	Первое издание
сигнатура	контракт
детализация	объединение
'()	<code>empty</code>
#true	<code>true</code>
#false	<code>false</code>

Последние три отличия значительно улучшают цитирование списков.

¹ <https://htdp.org/2003-09-26/Book/>. – Прим. перев.

Благодарности из первого издания

Особую благодарность мы хотим выразить четырем нашим коллегам: Роберту Картрайту (Robert «Corky» Cartwright), который совместно с первым автором (Маттиасом Фелляйзеном) разработал вводный курс для университета имени Уильяма Марша Райса; Даниэлю П. Фридману (Daniel P. Friedman), предложившему первому автору переписать книгу «The Little LISPer» (также изданную в MIT Press) в 1984 году, которая положила начало этому проекту; Джону Клементсу (John Clements), разработавшему, внедрившему и обслуживающему механизм пошаговых вычислений для DrRacket; и Полу Стеклеру (Paul Steckler), который поддерживал команду, внося свой вклад в разработку нашего набора инструментов программирования.

Участие в создании книги приняли многие другие наши друзья и коллеги, которые использовали ее в своих курсах или давали подробные комментарии к рукописи. Мы благодарны вам за помощь и терпение: Ян Барланд (Ian Barland), Джон Клементс (John Clements), Брюс Дуба (Bruce Duba), Майк Эрнст (Mike Ernst), Кэти Фислер (Kathi Fisler), Даниэль П. Фридман (Daniel P. Friedman), Джон Грейнер (John Greiner), Джеральдин Морен (Géraldine Morin), Джон Стоун (John Stone) и Вальдемар Тамез (Valdemar Tamez).

Десятки поколений студентов курса Comp 210 в университете Райса использовали ранние версии текста и предложили различные улучшения. Кроме того, многочисленные участники нашей конференции TeachScheme! использовали рукопись в своих курсах. Многие из них прислали свои комментарии и предложения. Хотелось бы отметить наиболее активных участников: г-жа Барбара Адлер (Ms. Barbara Adler), д-р Стивен Блох (Dr. Stephen Bloch), г-жа Карен Бурас (Ms. Karen Buras), г-н Джек Клей (Mr. Jack Clay), д-р Ричард Клеменс (Dr. Richard Clemens), г-н Кайл Джиллетт (Mr. Kyle Gillette), г-н Марвин Эрнандес (Mr. Marvin Hernandez), г-н Майкл Хант (Mr. Michael Hunt), г-жа Карен Норт (Ms. Karen North), г-н Джейми Рэймонд (Mr. Jamie Raymond) и г-н Роберт Рид (Mr. Robert Reid). Кристофер Фелляйзен (Christopher Felleisen) участвовал в проработке нескольких первых частей книги вместе со своим отцом и помог получить представление о взглядах молодого студента. Хрвое Блажевич (Hrvoje Blazevic, в то время яхтсмен, а ныне капитан танкера *LPG/C Harriette*), Джо Захарий (Joe Zachary, университет штата Юта) и Даниэль П. Фридман (Daniel P. Friedman, университет штата Индиана) помогли найти и исправить многочисленные опечатки в первом издании. Спасибо всем вам.

Наконец, Маттиас выражает свою благодарность Хельге (Helga) за ее многолетнее терпение и за окружение уютом ее рассеянных мужа и отца. Робби выражает благодарность Синь-Хуэй Хуан (Hsing-Huei Huang) – без ее поддержки и терпения он не смог бы работать так плодотворно. Мэттью благодарит Вэнь Юань (Wen Yuan) за ее постоянную поддержку и непреходящую музыку. Шрирам выражает призна-

тельность Кати Фислер (Kathi Fisler) за поддержку, терпение и бесконечные шутки, а также за ее участие в этом проекте.

Благодарности

Как и в 2001 году, мы благодарны Джону Клементсу (John Clements) за разработку, проверку, внедрение и поддержку механизма пошаговых вычислений для DrRacket. Он занимается им уже почти 20 лет, и его движок стал незаменимым инструментом обучения.

За последние несколько лет некоторые наши коллеги передали нам свои комментарии и предложения по улучшению. Мы с благодарностью отмечаем содержательные беседы и обмен мнениями с этими людьми:

Кэти Фислер (Kathi Fisler, Вустерский политехнический институт и университет Брауна), Грегор Кичалес (Gregor Kiczales, университет Британской Колумбии), Прабхакар Рагде (Prabhakar Ragde, университет Ватерлоо) и Норман Рэмси (Norman Ramsey, университет Тафтса).

Тысячи преподавателей посетили наши семинары, проводившиеся на протяжении многих лет, и многие из них давали нам ценные отзывы. Но особенно нам хотелось бы отметить Дэна Андерсона (Dan Anderson), Стивена Блоха (Stephen Bloch), Джека Клея (Jack Clay), Надима Абдул Хамида (Nadeem Abdul Hamid) и Виера Пру (Viera Proulx), внесших большой вклад в это издание.

Гийом Марсо (Guillaume Marceau), работая вместе с Кэти Фислер (Kathi Fisler) и Шрирам, потратил много месяцев на изучение сообщений об ошибках в DrRacket и их устранение. Мы благодарны ему за его прекрасную работу.

Селеста Холленбек (Celeste Hollenbeck) – самая удивительная наша читательница. Она не уставала спрашивать снова и снова, пока не усваивала материал до конца. Она никогда не останавливалась на полу пути, продвигая свои тезисы. Большое спасибо за ваши невероятные усилия.

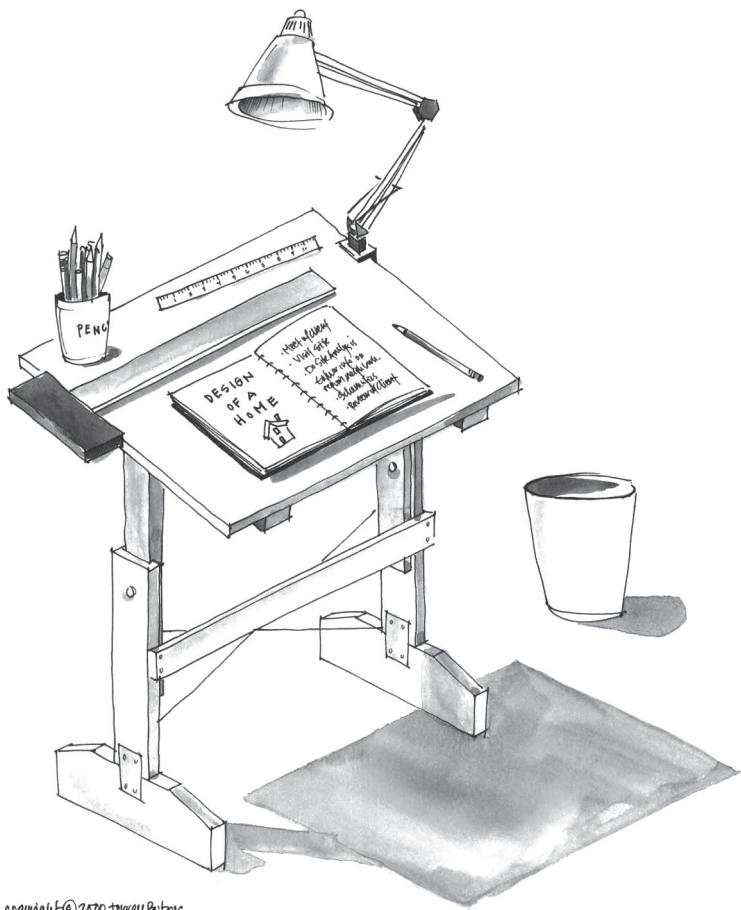
Мы также благодарим Эннаса Абдуссалама (Ennas Abdussalam), Марка Олдрича (Mark Aldrich), Мехмета Акифа Аккуса (Mehmet Akif Akkus), Анису Ануар (Anisa Anuar), Кристофа Бадура (Christoph Badura), Франко Барбайта (Franco Barbeite), Саада Башира (Saad Bashir), Аарона Баумана (Aaron Bauman), Сюзанну Беккер (Suzanne Becker), Майкла Бауша (Michael Bausch), Стивена Белкнапа (Steven Belknap), Стивена Блоха (Stephen Bloch), Илью Боткина (Elijah Botkin), Джозефа Богарта (Joseph Bogart), Уильяма Брауна (William Brown), Томаса Кабрера (Tomas Cabrera), Сююкун Си (Xuyuqun Si), Колина Кейна (Colin Caine), Энтона Каррико (Anthony Carrico), Родольфо Карвалью (Rodolfo Carvalho), Эстево Кастро (Estevo Castro), Марию Чакон (Maria Chacon), Стивена Чанга (Stephen Chang), Дэвида Чатмана (David

Chatman), Берли Харитон (Burleigh Chariton), Тунг Ченг (Tung Cheng), Нельсона Чиу (Nelson Chiu), Томаша Хршновича (Tomasz Chrzczonewicz), Джека Клея (Jack Clay), Ричарда Клейса (Richard Cleis), Джона Клементса (John Clements), Скотта Краймбла (Scott Crymble), Пирса Дарра (Pierce Darragh), Йонаса Декрекера (Jonas Decraecker), Ку Дунфанг (Qu Dongfang), Доминика Дейкхейзена (Dominique Dijkhuizen), Марка Энгельберга (Mark Engelberg), Эндрю Фаллоуза (Andrew Fellows), Цзянькун Фань (Jiankun Fan), Кристофера Фелляйзена (Christopher Felleisen), Себастьяна Фелляйзена (Sebastian Felleisen), Владимира Гаджика (Vladimir Gajić), Синь Гао (Xin Gao), Адриана Германа (Adrian German), Джека Гительсона (Jack Gitelson), Кайл Джиллетт (Kyle Gillette), Джонатана Гордона (Jonathan Gordon), Скотта Грина (Scott Greene), Бена Гринмана (Ben Greenman), Райана Голбека (Ryan Golbeck), Джоша Грэмса (Josh Grams), Григориоса (Grigorios), Джейн Грискти (Jane Griscti), Альберто Э. Ф. Герреро (Alberto E. F. Guerrero), Тайлер Хаммонд (Tyler Hammond), Нана Халберга (Nan Halberg), Ли Цзюнсон (Li Junsong), Надима Абдул Хамида (Nadeem Abdul Hamid), Джереми Хэнлона (Jeremy Hanlon), Тони Хенка (Tony Henk), Крейга Холбрука (Craig Holbrook), Коннора Хецлера (Connor Hetzler), Бенджамина Хоссейнзала (Benjamin Hosseinzahl), Уэйн Иба (Wayne Iba), Джона Джекмана (John Jackaman), Джордана Джонсона (Jordan Johnson), Блейка Джонсона (Blake Johnson), Эрвина Юнга (Erwin Junge), Марка Кауфманна (Marc Kaufmann), Коула Кендрика (Cole Kendrick), Грего-ра Кичалеса (Gregor Kiczales), Юджина Колбекера (Eugene Kohlbecker), Ярослава Колосовского (Jaroslaw Kolosowski), Кейтлин Крамер (Caitlin Kramer), Романа Кунин (Roman Kunin), Джексона Лоулера (Jackson Lawler), Девона Лепажа (Devon LePage), Бена Лернера (Ben Lerner), Ши-ченг Ли (Shicheng Li), Чен Элджей (Chen Lj), Эда Мафиса (Ed Maphis), Юшенг Мэй (YuSheng Mei), Андреса Меза (Andres Meza), Саада Махмуда (Saad Mhmood), Елену Мачкасову (Elena Machkasova), Джая Мартина (Jay Martin), Александра Мартинеса (Alexander Martinez), Юрия Машика (Yury Mashika), Джая Э. Маккарти (Jay McCarthy), Джеймса Макдоннелла (James McDonell), Майка Макхью (Mike McHugh), Уэйда Макрейнольдса (Wade McReynolds), Дэвида Мозеса (David Moses), Энн Е. Москол (Ann E. Moskol), Скотта Ньюсона (Scott Newson), Штепана Немеца (Štěpán Němec), Пола Оджанена (Paul Ojanen), проф. Роберта Ордоñеса (Prof. Robert Ordóñez), Лоран Орсо (Laurent Orseau), Клауса Остерманна (Klaus Ostermann), Аллану Паско (Alanna Pasco), Синана Пехливаноглу (Sinan Pehlivanoglu), Эрика Паркера (Eric Parker), Дэвида Портера (David Porter), Ника Плицикаса (Nick Pleatsikas), Пратьюша Прамода (Prathyush Pramod), Алока Рая (Alok Rai), Нормана Рамси (Norman Ramsey), Кришнана Равикумара (Krishnan Ravikumar), Джекоба Рубина (Jacob Rubin), Ильнара Салимзянова (Ilnar Salimzianov), Луиса Санжуана (Luis Sanjuán), Брайана Шака (Brian Schack), Райана «Хевви» Шила (Ryan «Havvy» Scheel), Лизу Шойинг (Lisa Scheuing), Вилли Шигеля (Willi Schiegel), Винита Шаха (Vinit Shah), Ника Шелли (Nick Shelley), Эдварда Шена (Edward Shen), Тубо Ши (Tubo Shi), Хеен

Шин (Hyeyoung Shin), Атхарва Шукла (Atharva Shukla), Мэтью Сингера (Matthew Singer), Майкла Сигела (Michael Siegel), Стивена Сигела (Stephen Siegel), Милтона Сильву (Milton Silva), Картика Сингхала (Kartik Singhal), Джо Сникериса (Joe Snikeris), Марка Смита (Marc Smith), Маттиса Смита (Matthijs Smith), Дэйва Смайли (Dave Smylie), Винсента Сент-Амура (Vincent St-Amour), Рида Стивенса (Reed Stevens), Уильяма Стивенсона (William Stevenson), Кевина Салливана (Kevin Sullivan), Асуму Такикава (Asumu Takikawa), Эрика Тантера (Éric Tanter), Сэма Тобина-Хохштадта (Sam Tobin-Hochstadt), Таноса Цуанаса (Thanos Tsouanas), Аарона Цая (Aaron Tsay), Маришку Тваалфховен (Mariska Twaalfhoven), Бора Гонсалеса Усача (Bor Gonzalez Usach), Рикардо Руи Валле-мена (Ricardo Ruy Valle-mena), Мануэля дель Валле (Manuel del Valle), Дэвида Ван Хорна (David Van Horn), Ника Вона (Nick Vaughn), Симеона Вельдстру (Simeon Veldstra), Андре Вентера (Andre Venter), Яна Витека (Jan Vitek), Марко Виллотта (Marco Villotta), Митча Ванда (Mitch Wand), Юсюй (Эвен) Ван (Yuxu (Ewen) Wang), Майкла Виджайю (Michael Wijaya), Дж. Клиффорда Уильямса (G. Clifford Williams), Эвана Уиттакера-Уокера (Ewan Whittaker-Walker), Джюлию Влоховски (Julia Wlochowski), Рулофа Воббена (Roelof Wobben), Дж. Т. Райта (J. T. Wright), Мардина Ядегара (Mardin Yadegar), Хуан ИчАО (Huang Yichao), Ю Ван Инь (Yuwang Yin), Эндрю Ципперера (Andrew Zipperer) и Ари Цви (Ari Zvi) за комментарии к рукописи этого второго издания.

Верстка сайта htdp.org создана Мэтью Баттериком (Matthew Butterick), который также разработал стили оформления для нашей онлайн-документации.

Наконец, мы благодарны Аде Брунштейн (Ada Brunstein) и Мари Луфкин Ли (Marie Lufkin Lee), нашим редакторам из MIT Press, которые дали нам разрешение на публикацию в интернете рукописей второго издания «Как проектировать программы». Мы также благодарим Кристин Бриджит Сэвидж (Christine Bridget Savage) из Массачусетского технологического института и Джона Хоуи (John Hoey) из Westchester Publishing Services за управление процессом публикации. Джон Донохью (John Donohue), Дженифер Робертсон (Jennifer Robertson) и Марк Вудворт (Mark Woodworth) проделали большую работу по редактированию рукописи.



copyright©2020 Torrey Buttar

Пролог: как писать программы

Когда вы были маленьким ребенком, родители учили вас считать на пальцах: «1 + 1 равно 2»; «1 + 2 равно 3» и т. д. Затем они спрашивали: «А сколько будет 3 + 2?» – и вы считали пальцы одной руки. Они программирували, а вы вычисляли. В каком-то смысле это все, что нужно для программирования и вычислений.

Теперь пришло время поменяться ролями. Запустите DrRacket. Перед вами откроется окно, как показано на рис. 2¹. Выберите пункт **Choose language** (Выбрать язык...) в меню **Language** (Язык), после чего в открывшемся диалоге выберите пункт **Teaching Languages** (Учебные языки) и внутри этого пункта, в списке под заголовком **How to Design Programs** (Как проектировать программы), выберите пункт **Beginning Student** (Начинающий студент, то есть язык для начинающих студентов – BSL) и щелкните на кнопке **OK**. Теперь вы можете начать программировать, а DrRacket будет вашим ребенком. Начните с простейших вычислений. Введите

```
| (+ 1 1)
```

в верхней половине окна DrRacket, щелкните на кнопке **RUN** (Выполнить) – и в нижней половине появится число 2.

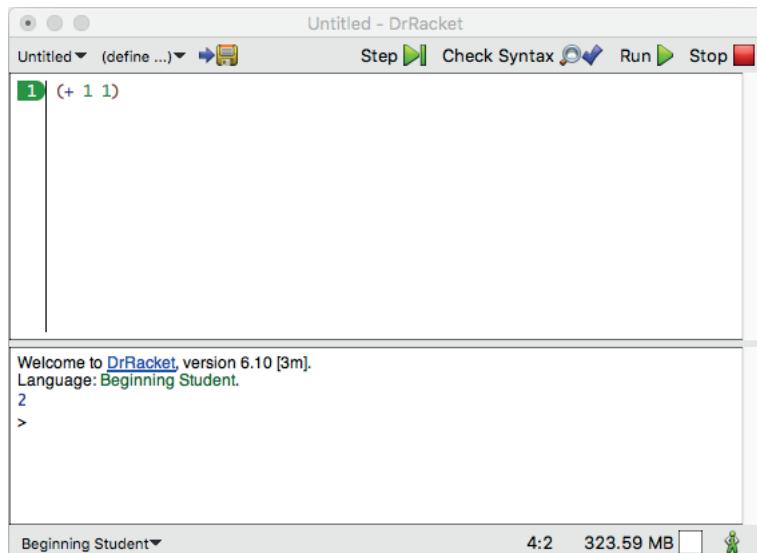


Рис. 2. Общий вид окна DrRacket

¹ Редактор DrRacket имеет русифицированный интерфейс. Чтобы включить его, выберите пункт меню **Help > Работать с русским интерфейсом DrRacket**. После этого откроется диалог, предупреждающий, что для смены языка интерфейса необходимо перезапустить редактор. Щелкните на кнопке **Accept and Exit**, или **Применить и выйти**, а затем снова запустите DrRacket. – Прим. перев.

Как видите, программировать ничуть не сложно. Вы задаете вопросы, как если бы DrRacket был ребенком, а DrRacket выполняет вычисления. Вы также можете попросить DrRacket обработать сразу несколько запросов:

```
(+ 2 2)
(* 3 3)
(- 4 2)
(/ 6 2)
```

После щелчка на кнопке **RUN** (Выполнить) и в нижней половине окна появятся числа 4 9 2 3 – ожидаемые результаты.

Теперь приостановимся ненадолго и проясним некоторые термины.

- Верхняя половина окна DrRacket называется *областью определений*. В этой области создаются программы, а процесс их создания называется *редактированием*. Сразу после добавления нового слова или изменения чего-либо в области определений в верхнем левом углу появляется кнопка **SAVE** (Сохранить). После первого щелчка на кнопке **SAVE** (Сохранить) DrRacket попросит указать имя файла, чтобы сохранить вашу программу. После того как область определений будет связана с файлом, последующие щелчки на кнопке **SAVE** (Сохранить) помогут вам гарантировать своевременное сохранение содержимого области определений в этом файле.
- *Программы* состоят из *выражений*. Вы уже не раз видели выражения на уроках математики. Выражение представляет собой либо обычное число, либо что-то, что начинается с открывающей круглой скобки «(» и заканчивается парной ей закрывающей круглой скобкой «)». DrRacket распознает парные скобки и закрашивает область между скобками.
- После щелчка на кнопке **RUN** (Выполнить) DrRacket вычисляет выражения в области определений и выводит полученные результаты в *области взаимодействий*. Затем DrRacket, ваш верный слуга, выводит *приглашение к вводу* (>) и ждет ваших команд. Этим приглашением DrRacket сигнализирует, что готов к вводу дополнительных выражений, которые он вычислит и выведет результат точно так же, как если бы выражение было введено в области определений:

```
> (+ 1 1)
2
```

Ведите выражение рядом с приглашением, нажмите клавишу **Return** или **Enter** на клавиатуре и посмотрите, как DrRacket отреагирует на результат. Вы можете ввести столько выражений, сколько пожелаете, например:

```
> (+ 2 2)
4
```

```

> (* 3 3)
9
> (- 4 2)
2
> (/ 6 2)
3
> (sqrt 3)
9
> (expt 2 3)
8
> (sin 0)
0
> (cos pi)
#i-1.0

```

Внимательно рассмотрите последний номер. Префиксом «#i» DrRacket сообщает: «На самом деле я не знаю точного числа, поэтому получите то, что у меня есть, – неточное (*inexact*) число». В отличие от вашего калькулятора или других систем программирования, DrRacket честен. Когда точное число неизвестно, в ответ добавляется специальный префикс. Позже мы покажем настоящие странности, которые творятся с «компьютерными числами», и тогда вы по достоинству оцените предупреждения, которые выводит DrRacket.

Возможно, вам интересно узнать: может ли DrRacket складывать больше двух чисел сразу? Да, может! Сделать это можно двумя разными способами:

```

> (+ 2 (+ 3 4))
9
> (+ 2 3 4)
9

```

Первый способ – использование *вложенных арифметических выражений*. Он известен всем нам еще со школы. Второй – *арифметические выражения на языке BSL*; это более естественный способ, потому что в языке BSL операции и числа всегда заключаются в круглые скобки.

В BSL всегда, когда требуется выполнить арифметическую операцию, выражение начинается с открывающей скобки, за которым следуют: символ операции, скажем +; числа, к которым нужно применить операцию (через пробел или даже через разрывы строк); и, наконец, закрывающая скобка. Элементы, следующие за операцией, называются *операндами*. При использовании вложенных выражений эти выражения сами выступают в роли operandов во вмещающем выражении, поэтому

```

> (+ 2 (+ 3 4))
9

```

является вполне допустимой программой. Вложенные выражения можно использовать в любом месте и в любых количествах:

```

> (+ 2 (+ (* 3 3) 4))
15

```

Эта книга не научит вас программированию на языке Racket, даже притом что редактор называется DrRacket. Прочтайте вступление, особенно раздел «DrRacket и языки обучения», где подробно рассказывается о выборе языка.

```

> (+ 2 (+ (* 3 (/ 12 4)) 4))
15
> (+ (* 5 5) (+ (* 3 (/ 12 4)) 4))
38

```

И нет никаких ограничений на вложенность, кроме вашего терпения.

Естественно, выполняя вычисления, DrRacket использует правила, которые известны вам из математики. Как и вы, он может определить результат сложения, только когда все операнды являются обычными числами. Если operand представлен операторным выражением в скобках, начинающимся с открывающей скобки «(» и символа операции, то DrRacket сначала вычислит результат этого вложенного выражения. В отличие от вас, ему не приходится задумываться о том, какое выражение вычислить первым, потому что это первое правило есть единственное правило.

За удобства DrRacket приходится платить скрупулезным отношением к скобкам. Вы должны ввести все необходимые круглые скобки, и при этом не должно быть лишних скобок. Например, ваш учитель математики может терпимо относиться к присутствию лишних скобок, но это не относится к BSL. Выражение $(+ (1) (2))$ содержит лишние круглые скобки, и DrRacket однозначно сообщит вам об этом:

```

> (+ (1) (2))
function call:expected a function after the open parenthesis,
found a number

```

(вызов функции: после открывающей круглой скобки ожидается функция, а обнаружено число).

Однако, привыкнув к особенностям языка BSL, вы увидите, что эта цена не так уж высока. Во-первых, вы можете использовать операции сразу с несколькими operandами, например:

```

> (+ 1 2 3 4 5 6 7 8 9 0)
45
> (* 1 2 3 4 5 6 7 8 9 0)
0

```

Как можно заметить, в онлайн-версии книги [названия операций связаны с документацией в HelpDesk](#).

Если вы не знаете, что делает операция с несколькими operandами, введите пример в области взаимодействия и нажмите **return**; DrRacket позволяет экспериментировать и узнавать, работает ли тот или иной прием, и как. Или обратитесь к документации HelpDesk. Во-вторых, читая программы, написанные другими, вам никогда не придется задаваться вопросом, какие выражения вычисляются в первую очередь. Круглые скобки и вложенность сразу скажут вам об этом.

В этом контексте «программировать» значит записывать понятные арифметические выражения, а «вычислять» – определять их значение. С DrRacket вы легко освоите этот вид программирования и вычислений.

Арифметика, арифметика...

Если бы в программировании использовались только числа и арифметические операции, то этот вид деятельности был бы таким же скучным, как уроки математики. К счастью, в программировании можно использовать не только числа, но также текст, флаги истинности, изображения и многое другое.

Шучу: математика – увлекательнейший предмет, но нам она пока не особенно нужна.

Прежде всего вы должны запомнить, что текст в BSL – это любая последовательность символов, введенных с клавиатуры, заключенная в двойные кавычки (""). Мы называем это строкой. То есть "hello world" – это типичная строка, и, «вычисляя» такие строки, DrRacket просто выводит их в области взаимодействий, как число:

```
| > "hello world"
| "hello world"
```

Многие люди пишут свои первые программы, которые выводят именно эту строку.

Вам также необходимо знать, что DrRacket поддерживает не только арифметику чисел, но и арифметику строк. Вот два примера, иллюстрирующих эту форму арифметики:

```
| > (string-append "hello" "world")
| "helloworld"
| > (string-append "hello" " " "world")
| "hello world"
```

`string-append`, как и `+`, тоже является оператором; он создает новую строку, объединяя все строки, следующие за ним. Как показывает первое взаимодействие, `string-append` объединяет строки буквально, не добавляя ничего между ними: ни пробелов, ни запятых, ничего. Поэтому если вы хотите увидеть фразу "hello world", то должны сами добавить пробел к одному из этих слов; именно это показывает второе взаимодействие. Конечно, самый естественный способ составить фразу из двух слов – ввести

```
| (string-append "hello" " " "world")
```

потому что `string-append`, так же как `+`, может обрабатывать любое количество operandов.

Со строками можно выполнять не только сложение. Вы можете извлекать фрагменты из строк, переворачивать их, преобразовывать все буквы в верхний (или нижний) регистр, удалять пробелы слева и справа и т. д. И что самое важное, вам не нужно ничего запоминать. Чтобы узнать, какие операции можно выполнять со строками, достаточно поискать в HelpDesk.

Заглянув в раздел с описанием простейших операций, доступных в BSL, можно увидеть, что *простейшие* (иногда их

Открыть HelpDesk можно, нажав клавишу F1 или выбрав соответствующий пункт в контекстном меню. Загляните в руководство по языку BSL, в раздел с описанием предопределенных операций, особенно операций со строками.

называют *предопределеными* или *встроеннымы*) операции могут потреблять строки и производить числа:

```
| > (+ (string-length "hello world") 20)
| 31
| > (number->string 42)
| "42"
```

Также есть операция, преобразующая строку в число:

```
| > (string->number "42")
| 42
```

Если вы ожидали увидеть в результате строку «forty-two» («сорок два») или что-то в этом роде, то извините: строковый калькулятор – не совсем то, что вам нужно.

Тем не менее последнее выражение вызывает вопрос: что получится, если применить операцию `string->number` к строке, которая не является изображением числа в кавычках? В этом случае операция вернет результат другого типа:

```
| > (string->number "hello world")
| #false
```

Это не число и не строка; это логическое значение. В отличие от чисел и строк, логические значения бывают только двух видов: `#true` и `#false`. Первое обозначает истину, а второе – ложь. В DrRacket имеется несколько операций для объединения логических значений:

```
| > (and #true #true)
| #true
| > (and #true #false)
| #false
| > (or #true #false)
| #true
| > (or #false #false)
| #false
| > (not #false)
| #true
```

возвращающих результаты, которые соответствуют названиям операций. (Не знаете, что означают операции `and`, `or` и `not`? Все просто: `(and x y)` вернет истину, если `x` и `y` истинны; `(or x y)` вернет истину, если либо `x`, либо `y`, либо оба истинны; и `(not x)` вернет истину, только если `x` ложно.)

Иногда бывает полезно «преобразовать» два числа в логическое значение:

```
| > (> 10 9)
| #true
| > (< -1 0)
| #true
| > (= 42 9)
| #false
```

Стоп! Попробуйте выполнить следующие три выражения: ($> = 10$ 10), ($<= -1 0$) и ($(string=? "Design" "tinker")$). Последнее выражение выглядит необычно, но не волнуйтесь, DrRacket справится с ним.

Со всеми этими новыми видами данных – числа, строки и логические значения являются данными – и операций легко забыть некоторые основы, такие как вложенные выражения:

```
| (and (or (= (string-length "hello world")
|           (string->number "11"))
|           (string=? "hello world" "good morning"))
|       (>= (+ (string-length "hello world") 60) 80))
```

Что получится в результате вычисления данного выражения? Как вы это поняли? Вы сами догадались об этом? Или просто ввели выражение в области взаимодействий и нажали клавишу **return**? Если вы поступили именно так, то как вы думаете, вы смогли бы определить результат самостоятельно? В конце концов, если вы не научитесь предсказывать результат, возвращаемый DrRacket для небольших выражений, вы не сможете доверять результатам вычислений более сложных задач.

Прежде чем приступать к изучению «настоящего» программирования, давайте обсудим еще один вид данных, который поможет оживить процесс: изображения. Если вставить изображение в область взаимодействий и нажать клавишу **return**, вот так:

```
| > 
```

в ответ DrRacket выведет изображение. В отличие от многих других языков программирования, BSL понимает изображения и поддерживает арифметику с изображениями, по аналогии арифметики с числами и строками. Проще говоря, ваши программы могут выполнять вычисления с изображениями, и вы можете оперировать ими в области взаимодействий. Более того, програмисты на BSL, как и программисты на других языках программирования, создают библиотеки, которые могут окаться полезными для других. Использование таких библиотек напоминает расширение словаря новыми словами. Мы называем такие библиотеки *учебными пакетами*, потому что они помогают в обучении.

Одна из важнейших библиотек – *2htdp/image* – поддерживает операции определения ширины и высоты изображения:

```
| (* (image-width ) (image-height 

```

После добавления библиотеки в программу щелчок на кнопке **RUN** (Выполнить) даст вам число 1176 – площадь изображения с размерами 28 на 42.

Для вставки изображений в DrRacket, например изображения ракеты, используйте меню *Insert* (Вставка). Или скопируйте и вставьте изображение из вашего браузера.

Добавьте выражение (*require 2htdp/image*) в область определений или выберите пункт *Add Teachpack* (Добавить учебный пакет) в меню *Language* (Язык) и выберите пакет *image* в списке *Preinstalled HtDP/2e Teachpack* (Предустановленные учебные пакеты HtDP/2e).

Вам не обязательно искать изображения в Google, чтобы вставлять их в программы DrRacket с помощью меню **Insert** (Вставка). Можете поручить DrRacket создавать простые изображения с нуля:

```
> (circle 10 "solid" "red")
●
> (rectangle 30 20 "outline" "blue")
□
```

Когда результатом выражения является изображение, DrRacket рисует его в области взаимодействия. Но в остальном программа BSL работает с изображениями как с данными, подобными числам. В частности, в BSL есть операции для объединения изображений также, как и операции для сложения чисел или добавления строк:

```
> (overlay (circle 5 "solid" "red")
            (rectangle 20 20 "solid" "blue"))
■
```

Наложение этих изображений в обратном порядке дает в результате сплошной синий квадрат:

```
> (overlay (rectangle 20 20 "solid" "blue")
            (circle 5 "solid" "red"))
■
```

Давайте остановимся и на мгновение задумаемся над последним результатом.

Как видите, операция `overlay` больше похожа на `string-append`, чем на `:`: она «складывает» изображения так же, как `string-append` «складывает» строки, а `+` вычисляет сумму чисел. Вот еще одна иллюстрация:

```
> (image-width (square 10 "solid" "red"))
10
> (image-width
    (overlay (rectangle 20 20 "solid" "blue")
             (circle 5 "solid" "red")))
20
```

Эти взаимодействия с DrRacket вообще ничего не рисуют; они просто измеряют ширину получившегося изображения.

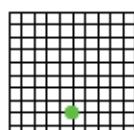
Следует упомянуть еще две операции: `empty-scene` и `place-image`. Первая создает сцену, особый вид прямоугольника, а вторая помещает изображение в эту сцену:

Не совсем так. На самом деле в полученном изображении отсутствует сетка.

Мы наложили сетку на пустую сцену, чтобы вы могли видеть, где именно находится зеленая точка.

```
(place-image (circle 5 "solid" "green")
            50 80
            (empty-scene 100 100))
```

В результате получается:



Как можно видеть на этом изображении, начало координат (или $(0,0)$) находится в верхнем левом углу. В отличие от геометрии, ось Y направлена **вниз**, а не вверх. В остальном изображение показывает именно то, что можно было ожидать: зеленый диск с центром в точке с координатами $(50,80)$ в пустом прямоугольнике 100 на 100.

Подведем некоторые итоги. Под программированием подразумевается запись арифметических выражений, но при этом вы не ограничены одними только скучными числами. В языке BSL под арифметикой подразумевается арифметика чисел, строк, логических значений и даже изображений. Однако под вычислением по-прежнему подразумевается определение значения выражения, разве что это значение может быть строкой, числом, логическим значением или изображением.

Теперь вы готовы писать программы, которые заставляют ракеты летать.

Входы и выходы

Программы, которые мы писали до сих пор, довольно незатейливы. Мы записывали выражение или несколько выражений, щелкали на кнопке **RUN** (Выполнить) и просматривали получившиеся результаты. Если снова щелкнуть на кнопке **RUN** (Выполнить), DrRacket выведет точно такие же результаты. Фактически вы можете щелкать на кнопке **RUN** (Выполнить) сколько угодно раз, и результаты от этого не изменятся. Проще говоря, наши программы больше похожи на вычисления на карманном калькуляторе, с той лишь разницей, что DrRacket может выполнять вычисления с любыми видами данных, а не только с числами.

Это и хорошо, и плохо. Хорошо, потому что программирование и вычисления являются естественным обобщением калькулятора. Плохо, потому что цель программирования – обрабатывать большое количество данных и получать много разных результатов, выполняя более или менее одинаковые вычисления. (Программы также должны вычислять эти результаты быстро, по крайней мере быстрее, чем мы.) То есть вам нужно еще многое узнать, прежде чем вы научитесь программировать. Но не волнуйтесь: обладая знаниями об арифметике чисел, строк, логических значений и изображений, вы почти готовы написать программу, которая создает фильмы, а не просто выводит какое-нибудь простенькое сообщение, такое как «hello world». И этим мы займемся дальше.

На всякий случай, если вы не знали этого, фильм – это последовательность изображений, которые быстро сменяют друг друга на экране. Если бы ваши учителя математики знали об «арифметике изображений», которую вы видели в предыдущем разделе, то наверняка научили бы вас создавать фильмы вместо скучных числовых последовательностей. Вот еще одна такая таблица:

$x =$	1	2	3	4	5	6	7	8	9	10
$y =$	1	4	9	16	25	36	49	64	81	?

Ваши учителя могли бы попросить вас вставить недостающее число в ячейку со знаком вопроса «?».

Как оказывается, снять фильм не сложнее, чем заполнить такую таблицу чисел. Действительно, все дело в таких таблицах:

$x =$	1	2	3	4
$y =$	•	•	•	?

Говоря более конкретно, ваш учитель мог бы предложить вам нарисовать четвертое, пятое и 1273-е изображения, потому что фильм – это просто длинная последовательность изображений, сменяющих друг друга примерно 20 или 30 раз в секунду. То есть вам понадобится от 1200 до 1800 таких изображений, чтобы из них сконструировать фильм продолжительностью в одну минуту.

Вы также можете вспомнить, что ваши учителя могли просить не только вставить четвертое или пятое число в некоторой последовательности, но и указать выражение, определяющее любой элемент последовательности по заданному значению x . В словесном примере учитель мог бы пожелать увидеть что-то вроде этого:

$$y = x \cdot x.$$

Если в эту формулу вместо x подставить 1, 2, 3 и т. д., то в результате получатся числа 1, 4, 9 и т. д., в точности как показано в таблице. Для последовательности изображений то же самое можно выразить примерно так:

y = изображение, содержащее точку на x^2 пикселей
ниже верхнего края.

Важно отметить, что эта формулировка обозначает не простое выражение, но функцию.

На первый взгляд функции похожи на выражения с символом u слева, за которым следует знак $=$ и выражение. Однако это не выражения, а функции, которые вы могли часто видеть в школе на уроках математики. Но в DrRacket функции записываются немного иначе:

```
| (define (y x) (* x x))
```

Слово `define` говорит: «считать у функцией», которая вычисляет значение подобно выражению. Однако значение функции зависит от значения того, что называется *входом*. Этот факт мы выражаем с помощью `(y x)`. Поскольку входное значение заранее неизвестно, то для его представления мы используем имя. Здесь, следуя математиче-

ской традиции, мы использовали имя x для обозначения неизвестного входа; но довольно скоро мы будем использовать другие имена.

Эта вторая часть означает, что в функцию нужно передать одно число – для x , – чтобы вычислить конкретное значение для y . В этом случае DrRacket подставит полученное значение x в выражение, связанное с функцией, в данном примере это выражение $(* x x)$. После замены x значением, например 1, DrRacket сможет вычислить результат выражения, который также называется *выходом* функции.

Щелкните на кнопке **RUN** (Выполнить) и обратите внимание, что ничего не произошло. В области взаимодействия не появилось ничего нового, как будто вы ничего и не вводили и в DrRacket ничего не изменилось. Но на самом деле это не так. Вы фактически определили функцию и сообщили DrRacket о ее существовании. Теперь редактор готов использовать эту функцию. Введите

| (y 1)

в области взаимодействий и убедитесь, что в ответ DrRacket вывел число 1. В DrRacket выражение $(y 1)$ называется *применением функции*. Попробуйте выполнить

В математике запись $y(1)$ тоже называется применением функции, просто ваши учителя забыли вам сказать об этом.

| (y 2)

и убедитесь, что в ответ DrRacket вывел 4. Конечно, все эти выражения можно также ввести в области определений и щелкнуть на кнопке **RUN** (Выполнить):

```
(define (y x) (* x x))
(y 1)
(y 2)
(y 3)
(y 4)
(y 5)
```

В ответ DrRacket выведет: 1 4 9 16 25 – числа из таблицы. Теперь определите недостающее число.

С нашей точки зрения функции дают весьма экономичный способ вычисления множества интересующих нас значений с помощью одного выражения. В действительности программы – это функции; и, освоив функции, вы будете знать о программировании почти все, что нужно. Учитывая важность функций, давайте обобщим то, что мы уже знаем о них.

- Во-первых,

| (define (ИмяФункции ИмяВхода) Тело)

– это *определение функции*. Об этом говорит ключевое слово `define` (определить). По сути, определение состоит из трех частей: двух имен и выражения. Первое имя – это имя функции. Вы будете использовать его, когда вам понадобится применить

функцию. Второе имя, называемое *параметром*, – это вход функции, который неизвестен до фактического применения функции. Выражение с именем *Тело* вычисляет выход (результат) функции для определенного входа.

- Во-вторых,

```
| (ИмяФункции ВыражениеАргумента)
```

– это *применение функции*. Первая часть сообщает DrRacket, какую функцию следует применить, а вторая часть – это вход, к которому применяется функция. Если бы вы сейчас читали руководство для Windows или Mac, в нем было бы написано, что это выражение *запускает приложение* с именем ИмяФункции, которое получает на вход значение ВыражениеАргумента и обрабатывает его. Как всякое другое выражение, ВыражениеАргумента может быть простым фрагментом данных или глубоко вложенным выражением.

Функции могут принимать и возвращать не только числа, но и все остальные виды данных. Давайте проверим это и создадим функцию, имитирующую вторую таблицу, с изображениями цветной точки, подобно тому, как первая функция имитировала числовую таблицу. Поскольку в школе вам не рассказывали, как в выражениях создавать изображения, начнем с простого. Помните пустую сцену? Мы кратко упоминали о ней в конце предыдущего раздела. Если создать ее в области взаимодействий, как показано ниже:

```
| > (empty-scene 100 60)
```



то DrRacket нарисует пустой прямоугольник, который называется сценой. В сцену можно добавлять изображения с помощью `place-image`:

```
| > (place-image 🚀 50 23 (empty-scene 100 60))
```



Представьте, что ракета – это точка на рисунках, показанных в таблице выше. Разница лишь в том, что видеть ракету интереснее.

Теперь вы должны заставить ракету опуститься, как точку в таблице выше. В предыдущем разделе вы узнали, как добиться этого эффекта: нужно увеличивать координату *y*, передаваемую в `place-image`:

```
| > (place-image 🚀 50 20 (empty-scene 100 60))
```



```
| > (place-image 🚀 50 30 (empty-scene 100 60))
```





```
> (place-image  50 40 (empty-scene 100 60))
```



Теперь осталось только определить способ, который позволит с легкостью создать множество таких сцен, и быстро отобразить по порядку.

Листинг 1. Посадка ракеты (версия 1)

```
(define (picture-of-rocket height)
  (place-image  50 height (empty-scene 100 60)))
```

Первую цель можно достичь с помощью функции в листинге 1. Да, это определение функции. Вместо у ей дано имя `picture-of-rocket`, которое ясно сообщает, что выводит функция: сцену с ракетой. Вместо `x` параметру в определении функции дано имя `height`, которое четко сообщает, что это число, которое определяет высоту местоположения ракеты. Выражение с телом функции в точности повторяет выражения, с которыми мы только что экспериментировали, за исключением того, что в нем вместо числа используется `height`. Теперь мы легко можем создать все необходимые изображения с помощью одной функции:

```
(picture-of-rocket 0)
(picture-of-rocket 10)
(picture-of-rocket 20)
(picture-of-rocket 30)
```

Ведите эти выражения в области определений или в области взаимодействий, и вы получите ожидаемые сцены.

Для достижения второй цели вы должны познакомиться с одной элементарной операцией из библиотеки `2htdp/universe`: `animate`. Щелкните на кнопке **RUN** (Выполнить) и введите следующее выражение:

```
| > (animate picture-of-rocket)
```

Обратите внимание, что выражение аргумента в этом примере – функция. Не беспокойтесь об использовании функций в качестве аргументов; этот прием прекрасно работает с `animate`, но пока не пытайтесь сами определять такие функции, как `animate`.

Как только вы нажмете клавишу **return**, DrRacket вычислит выражение, но не отобразит ни результата, ни даже приглашения к вводу. Вместо этого откроется другое окно – холст – и запустятся часы, тикающие 28 раз в секунду. С каждым тактом часов DrRacket будет применять `picture-of-rocket` к количеству тактов, прошедших с момента вызова `animate`. Результаты применения этой функции будут отобра-

язык `BSL` позволяет использовать в именах любые символы, включая «-» и «.».

Не забудьте добавить библиотеку `2htdp/universe` в область определений.

жаться на холсте и создавать эффект анимационного фильма. Моделирование продолжается до тех пор, пока вы не закроете окно. После этого `animate` вернет количество обработанных тактов.

В упражнении 298 объясняется, как то: `animate` применяет функцию в своем операнде к числам 0, организована функция 1, 2 и т. д. и отображает полученные изображения. Вот более `animate`. подробное объяснение:

- `animate` запускает часы и считает количество тактов;
- часы идут со скоростью 28 тактов в секунду;
- каждый раз, когда завершается очередной такт, `animate` применяет функцию `picture-of-rocket` к порядковому номеру текущего такта; и
- сцена, созданная в результате этого применения, отображается на холсте.

Это означает, что ракета сначала появляется на высоте 0, затем 1, потом 2 и т. д., то есть постепенно опускается вниз. Наша трехстрочная программа создает около 100 изображений примерно за 3,5 секунды, а быстрое их отображение создает эффект приземления ракеты.

А теперь обобщим все, что вы узнали в этом разделе. Функции – это удобный способ обработки больших объемов данных за короткое время. Вы можете запустить функцию вручную, передав несколько разных входов, чтобы проверить правильность выходных результатов. Это называется тестированием функции. DrRacket может запустить функцию для множества входов с помощью некоторых библиотек. Естественно, DrRacket может также запускать функции, когда вы нажимаете клавиши на клавиатуре или манипулируете мышью. Чтобы узнать, как это сделать, продолжайте читать. И независимо от того, как запускается применение функции, помните, что программы (простые) – это функции.

Множество способов вычисления

Если запустить (`animate picture-of-rocket`), спустя какое-то время ракета исчезнет под землей. Это выглядит странно. Ракеты в старых фантастических фильмах не тонут в земле; они изящно приземляются на опоры, и на этом фильм должен заканчиваться.

Эта идея предполагает, что в зависимости от ситуации вычисления должны выполняться по-разному. В нашем примере программа `picture-of-rocket` должна работать «как есть», пока ракета находится в полете. Однако как только ракета коснется нижней части холста, ее дальнейшее снижение должно остановиться.

Эта идея не должна быть для вас новой. Даже ваши учителя математики показывали вам функции, различающие разные ситуации:

$$\text{sign}(x) = \begin{cases} +1, & \text{если } x > 0 \\ 0, & \text{если } x = 0. \\ -1, & \text{если } x < 0 \end{cases}$$

Эта функция *sign* различает три вида входных значений: которые больше 0, равны 0 и меньше 0. В зависимости от входа результат функции равен +1, 0 или -1.

Эту функцию легко определить в DrRacket, используя условное выражение *cond*:

```
(define (sign x)
  (cond
    [(> x 0) 1]
    [(= x 0) 0]
    [(< x 0) -1]))
```

После щелчка на кнопке **RUN** (Выполнить) вы сможете использовать функцию *sign* как любую другую функцию:

```
> (sign 10)
1
> (sign -5)
-1
> (sign 0)
0
```

В общем случае условное выражение имеет следующий вид:

```
(cond
  [ВыражениеУсловия1 ВыражениеРезультата1]
  [ВыражениеУсловия2 ВыражениеРезультата2]
  ...
  [ВыражениеУсловияN ВыражениеРезультатаN])
```

То есть условное выражение *cond* состоит из некоторого необходимого количества *условных строк*. Каждая строка содержит два выражения: левое обычно называют *условием*, а правое – *результатом*; иногда также используются термины *вопрос* и *ответ*. Чтобы вычислить выражение *cond*, DrRacket вычисляет первое выражение условия, ВыражениеУсловия1. Если оно возвращает #true, то DrRacket заменяет все выражение *cond* выражением результата ВыражениеРезультата1, вычисляет его и получившееся значение возвращает как результат всего выражения *cond*. Если в результате вычисления ВыражениеУсловия1 получится #false, то DrRacket пропускает первую строку и переходит ко второй. Если все выражения условий вернут #false, то DrRacket сообщит об ошибке.

Теперь, зная это, вы можете изменить ход процесса. Цель состоит в том, чтобы не дать ракете опуститься ниже уровня земли в сцене размером 100 на 60 пикселей. Поскольку функция *picture-of-grocket* принимает высоту, на которой она должна изобразить ракету в сцене,

Откройте новую вкладку в DrRacket и начните с чистого листа.

*Сейчас самое время познакомиться с кнопкой **STEP** (Шаг). Введите (sign -5) в области определений (после ввода определения функции *sign*) и щелкните на кнопке **STEP** (Шаг).*

Когда появится новое окно, попробуйте пощелкать на кнопках со стрелками влево и вправо.

кажется, что достаточно просто сравнить заданную высоту с максимально допустимой.

В листинге 2 приводится уточненное определение функции. В нем определяется функция с именем `picture-of-rocket.v2`, чтобы мы могли различать две версии. Применение разных имен также позволяет использовать обе функции в области взаимодействий и сравнивать результаты.

Листинг 2. Посадка ракеты (версия 2)

```
(define (picture-of-rocket.v2 height)
  (cond
    [(<= height 60)
     (place-image 🚀 50 height
                   (empty-scene 100 60))]
    [(> height 60)
     (place-image 🚀 50 60
                   (empty-scene 100 60))]))
```

Вот как работает оригинальная версия:

```
> (picture-of-rocket 5555)

```

А так – вторая:

```
> (picture-of-rocket.v2 5555)

```

Какое бы число вы не передали функции `picture-of-rocket.v2`, если оно окажется больше 60, то вы получите ту же сцену. Если, к примеру, выполнить такое выражение:

```
| > (animate picture-of-rocket.v2)
```

то ракета опустится вниз и на половину своего корпуса уйдет под землю, после чего остановится.

Стоп! Это именно то, что мы хотели увидеть?

Ракета, погрузившаяся наполовину под землю, смотрится некрасиво. Но вы знаете, как исправить этот огрех. Как вы уже видели, язык BSL поддерживает арифметику изображений. Когда функция `place-image` добавляет изображение в сцену, она ориентируется на его центр, как если бы все изображение было представлено точкой, даже если оно имеет реальную высоту и реальную ширину. Как вы знаете, мы можем измерить высоту изображения с помощью `image-height`. Эта функция пригодится нам и поможет остановить спуск ракеты в тот момент, когда ее нижняя часть коснется земли.

Сложив два плюс два, нетрудно догадаться, что высота, на которой ракета должна прекратить спуск, вычисляется так:

```
| (- 60 (/ (image-height 🚀) 2))
```

Вы можете убедиться в этом, поиграв с самой программой или поэкспериментировав в области взаимодействий.

Вот первая попытка:

```
| (place-image 🚀 50 (- 60 (image-height 🚀))
  (empty-scene 100 60))
```

Теперь замените третий аргумент в примере выше выражением

```
| (- 60 (/ (image-height 🚀) 2))
```

Стоп! Поэкспериментируйте сами и оцените полученные результаты. Какой из них вам больше нравится?

Листинг 3. Посадка ракеты (версия 3)

```
(define (picture-of-rocket.v3 height)
  (cond
    [(<= height (- 60 (/ (image-height 🚀) 2)))
     (place-image 🚀 50 height
                 (empty-scene 100 60))]
    [(> height (- 60 (/ (image-height 🚀) 2)))
     (place-image 🚀 50 (- 60 (/ (image-height 🚀) 2))
                 (empty-scene 100 60)))]))
```

Размышляя и экспериментируя, вы в конечном итоге дойдете до программы в листинге 3. Если задано какое-то число, представляющее высоту местоположения ракеты, то сначала проверяется, достиг ли нижний край ракеты на земле. Если не достиг, то местоположение ракеты в сцене меняется, как и раньше. Иначе изображение ракеты позиционируется так, чтобы ее нижняя часть касалась земли.

Одна программа, множество определений

Теперь предположим, что ваши друзья посмотрели анимацию и им не понравился размер холста. Они попросили дать им версию, в которой сцена имеет размер 200×400 . Эта простая просьба заставит вас заменить 100 на 400 в пяти местах программы и 60 на 200 еще в двух местах, не говоря уже о числе 50, обозначающем «середину холста».

А теперь попробуйте проделать это, чтобы понять, насколько сложно выполнить данную просьбу для простенькой пятистрочной программы. Читая дальше, имейте в виду, что в мире не редкость программы, состоящие из 50 000, 500 000 или даже 5 000 000 или более строк программного кода.

В идеальной программе подобные просьбы, такие как изменение размеров холста, не должны требовать вносить такое большое коли-

чество изменений. В BSL этого легко добиться с помощью `define`. Эта инструкция способна определять не только функции, но также **константы**, присваивая имена некоторым значениям. Вот как выглядит определение константы в общем виде:

```
| (define Имя Выражение)
```

То есть вы можете добавить в программу такое определение:

```
| (define HEIGHT 60)
```

а в программе использовать `HEIGHT` везде, где прежде использовалось число 60. Смысл такого определения очевиден. Каждый раз, встретив имя `HEIGHT` во время вычислений, DrRacket будет заменять его числом 60.

Теперь взгляните на код в листинге 7, который реализует это простое изменение, а также присваивает имя изображению ракеты. Скопируйте эту программу в DrRacket, щелкните на кнопке **RUN** (Выполнить) и введите следующее выражение в области взаимодействий:

```
| > (animate picture-of-rocket.v4)
```

Убедитесь, что программа работает так же, как и раньше.

Программа в листинге 4 включает четыре определения: одно определение функции и три определения констант. Числа 100 и 60 встречаются в программе всего один раз – в определениях констант `WIDTH` и `HEIGHT`. Вы также могли заметить, что обновленная программа использует имя `h` вместо `height` для параметра функции `picture-of-rocket.v4`. Строго говоря, в этом изменении нет особой необходимости, потому что DrRacket не спутает `height` и `HEIGHT`, но мы сделали это, чтобы не сбить с толку вас.

Вычисляя выражение (`animate picture-of-rocket.v4`), DrRacket заменяет `HEIGHT` на 60, `WIDTH` на 100 и `ROCKET` на изображение ракеты каждый раз, когда встречает эти имена. Чтобы испытать радость настоящих программистов, замените число 60 в определении `HEIGHT` на 400 и щелкните на кнопке **RUN** (Выполнить). Вы увидите приземляющуюся ракету в сцене размером 100 на 400 пикселей. Чтобы увеличить высоту сцены, потребовалось всего одно небольшое изменение!

Листинг 4. Посадка ракеты (версия 4)

```
(define (picture-of-rocket.v4 h)
  (cond
    [(<= h (- HEIGHT (/ (image-height ROCKET) 2)))
     (place-image ROCKET 50 h (empty-scene WIDTH HEIGHT))]
    [(> h (- HEIGHT (/ (image-height ROCKET) 2)))
     (place-image ROCKET
                  50 (- HEIGHT (/ (image-height ROCKET) 2))
                  (empty-scene WIDTH HEIGHT)))])

(define WIDTH 100)
(define HEIGHT 60)
(define ROCKET 
```

Выражаясь современным языком, вы только что выполнили свой первый рефакторинг программы. Каждый раз, реорганизуя свою программу, чтобы подготовиться к возможным просьбам изменить что-нибудь в ней, вы выполняете рефакторинг программы. Добавьте этот пункт в свое резюме. Он смотрится неплохо, и вашему будущему работодателю, вероятно, понравится читать такие модные словечки, даже если это не делает вас хорошим программистом. Однако хороший программист никогда не смиряется с наличием в программе трех одинаковых выражений:

```
| (- HEIGHT (/ (image-height ROCKET) 2))
```

Каждый раз, когда ваши друзья и коллеги будут читать эту программу, им придется приостанавливаться, чтобы понять, что вычисляет это выражение – расстояние между верхним краем холста и центральной точкой ракеты, покоящейся на земле. Каждый раз, вычисляя эти выражения, DrRacket должен выполнить три шага: (1) определить высоту изображения; (2) разделить ее на 2 и (3) вычесть результат из HEIGHT. И каждый раз будет получаться одно и то же число.

Это наблюдение требует от нас добавить еще одно определение:

```
| (define ROCKET-CENTER-TO-TOP
  (- HEIGHT (/ (image-height ROCKET) 2)))
```

Теперь подставьте ROCKET-CENTER-TO-TOP вместо выражения `(- HEIGHT (/ (image-height ROCKET) 2))` в остальной части программы. Возможно, вас волнует вопрос: где разместить это определение – выше или ниже определения HEIGHT? Или в общем случае: имеет ли значение порядок следования определений? Ответ заключается в следующем: для определений констант порядок имеет значение, а для определений функций – нет. Встретив определение константы, DrRacket вычисляет выражение в определении, а затем связывает имя константы с полученным результатом. Например, следующая последовательность определений:

```
| (define HEIGHT (* 2 CENTER))
  (define CENTER 100)
```

вызовет сообщение об ошибке «CENTER is used before its definition» (константа CENTER используется до ее определения), когда DrRacket встретит определение константы HEIGHT.

Если переупорядочить определения:

```
| (define CENTER 100)
  (define HEIGHT (* 2 CENTER))
```

они будут вычислены без ошибок. Здесь DrRacket сначала свяжет имя CENTER с числом 100, а затем вычислит выражение `(* 2 CENTER)` и получит в результате число 200, которое благополучно свяжет с именем HEIGHT.

Программа также может содержать односрочечные комментарии, начинающиеся с точки с запятой (;). DrRacket игнорирует такие комментарии, но люди, читающие программы, не должны этого делать, потому что комментарии предназначены для людей.

Это «канал обратной связи» между автором программы и всеми ее читателями в будущем для передачи информации о программе.

Порядок определений констант имеет значение, но совершенно не важно, где поместить определения констант относительно определений функций. Если ваша программа включает множество определений функций, их порядок тоже не имеет значения, хотя лучше сначала ввести все определения констант, а затем определения функций в порядке убывания важности. Начав писать свои программы с множеством определений, вы поймете, почему этот порядок важен.

После устранения всех повторяющихся выражений вы получите программу, показанную в листинге 5. Она состоит из одного определения функции и пяти определений констант. Кроме положения центра ракеты, эти константы определяют также размеры самого изображения и сцены.

Листинг 5. Посадка ракеты (версия 5)

```
; константы
(define WIDTH 100)
(define HEIGHT 60)
(define MTSCN (empty-scene WIDTH HEIGHT))

(define ROCKET 
)
(define ROCKET-CENTER-TO-TOP
  (- HEIGHT (/ (image-height ROCKET) 2)))

; функции
(define (picture-of-rocket.v5 h)
  (cond
    [(<= h ROCKET-CENTER-TO-TOP)
     (place-image ROCKET 50 h MTSCN)]
    [(> h ROCKET-CENTER-TO-TOP)
     (place-image ROCKET 50 ROCKET-CENTER-TO-TOP MTSCN)]))
```

Прежде чем продолжить чтение, подумайте о следующих изменениях в вашей программе:

- Как бы вы изменили программу, чтобы создать сцену размером 200×400?
- Как бы вы изменили программу, чтобы она демонстрировала приземление зеленого НЛО (неопознанного летающего объекта)? Нарисовать НЛО легко:

```
(overlay (circle 10 "solid" "green")
        (rectangle 40 4 "solid" "green"))
```

- Как бы вы изменили программу, чтобы фон сцены окрасить в синий цвет?
- Как бы вы изменили программу, чтобы ракета приземлялась на плоскую каменную площадку, которая на 10 пикселей выше уровня земли? Не забудьте также добавить эту площадку в сцену.

Практика – лучший способ изучения. Поэтому не останавливайтесь и просто сделайте это.

Магические числа. Взгляните еще раз на функцию `picture-of-rocket.v5`. Мы убрали из определения функции все повторяющиеся выражения и все числа, кроме одного – числа 50. В мире программирования такие числа называют *магическими числами*, и большинство программистов не любят их. По прошествии времени легко забыть, какую роль играет число и можно ли его изменить. Лучше всего для таких чисел определить отдельные константы.

В данном случае мы знаем, что 50 – это выбранная нами координата `x` для ракеты. Несмотря на то что число 50 не похоже на выражение, в действительности оно является повторяющимся выражением. Таким образом, у нас есть две причины исключить 50 из определения функции, и мы предлагаем вам сделать это самостоятельно.

Еще одно определение

Напомним, что `animate` фактически применяет переданные ей функции к количеству тактов часов, прошедших с момента первого вызова. То есть аргументом для `picture-of-rocket` является не высота, а время. В наших предыдущих определениях `picture-of-rocket` использовалось неправильное имя для аргумента функции; вместо `h` (сокращенно от «height» – высота) следует использовать `t` (сокращенно от «time» – время):

```
(define (picture-of-rocket t)
  (cond
    [(<= t ROCKET-CENTER-TO-TOP)
     (place-image ROCKET 50 t MTSCN)]
    [(> t ROCKET-CENTER-TO-TOP)
     (place-image ROCKET
                  50 ROCKET-CENTER-TO-TOP
                  MTSCN)]))
```

*Будьте внимательны!
В этом разделе
используются некото-
рые знания из физики.
Если вас пугает физика,
пропустите этот
раздел при первом чте-
нии; программирование
не требует знаний
физики.*

Это небольшое изменение в определении сразу же проясняет, что эта программа использует время, как если бы оно было расстоянием. А это нехорошо.

Даже если вы пропускали уроки физики в школе, вы наверняка знаете, что время – это не расстояние. Так что наша программа работала по чистой случайности. Но не волнуйтесь, этот недостаток легко исправить. Для этого нужно немного знать ракетостроение, которое такие люди, как мы, называют физикой.

Физика?!? Возможно, вы уже забыли, чему вас учили на уроках физики. Или даже пропускали их, потому что были слишком молоды и ветрены. Но не волнуйтесь. Такое случается даже с лучшими программистами, потому что им приходится помогать людям, занимающимся музыкой, экономикой, фотографией, медициной и многими другими дисциплинами. Очевидно, что никакой программист не может знать всего этого. Поэтому они или ищут необходимые знания, или разгово-

ригают со специалистами. И если вы поговорите с физиком, то узнаете, что пройденное расстояние пропорционально времени:

$$d = v \cdot t.$$

То есть объект, движущийся со скоростью v , за t секунд переместится на d километров (метров, пикселей и т. п.).

Конечно, учитель должен показать вам правильное определение функции:

$$d(t) = v \cdot t,$$

потому что оно сразу говорит, что значение d зависит от t , а v является константой. Программисты обычно делают еще один шаг и заменяют однобуквенные сокращения осмысленными именами:

```
(define V 3)
(define (distance t)
  (* V t))
```

Этот фрагмент программы включает два определения: функцию `distance`, которая вычисляет расстояние, пройденное объектом, который движется с постоянной скоростью, и константу `V`, описывающую скорость.

Вы можете задаться вопросом: почему для скорости `V` здесь определено значение 3? Какой-то особой причины нет, просто мы посчитали, что 3 пикселя за такт – это хорошая скорость. Вы можете не согласиться с нами. Поэкспериментируйте с этим числом и посмотрите, что из этого получится.

Листинг 6. Посадка ракеты (версия 6)

```
; свойства "мира" и садящейся ракеты
(define WIDTH 100)
(define HEIGHT 60)
(define V 3)
(define X 50)

; константы, связанные с графикой
(define MTSCN (empty-scene WIDTH HEIGHT))

(define ROCKET 

```

Теперь можно еще раз исправить `picture-of-rocket`. Вместо сравнения `t` с высотой функция будет использовать выражение (`distance t`), вычисляющее высоту местоположения ракеты. Окончательная программа показана в листинге 6. Она включает определения двух функций: `picture-of-rocket.v6` и `distance`. Остальные определения констант делают определения функций легко читаемыми и изменяемыми. Как обычно, эту программу можно запустить с помощью `animate`:

```
| > (animate picture-of-rocket.v6)
```

По сравнению с предыдущими версиями, эта версия `picture-of-rocket` показывает, что программа может состоять из нескольких определений функций, ссылающихся друг на друга. Кроме того, даже в первой версии использовались функции `+ / -` просто вы думали, что они встроены в BSL.

Когда вы станете настоящим программистом, то обнаружите, что программы состоят из множества определений функций и множества определений констант. Вы также увидите, что функции все время ссылаются друг на друга. И ваша задача – организовать их так, чтобы вы могли легко читать эти определения даже спустя несколько месяцев после завершения работы над ними. В конце концов, вы или кто-то другой может пожелать внести изменения в эти программы, и если вы не сможете понять организацию программы, вам будет сложно выполнить даже самую простую задачу.

Теперь вы – программист

Это утверждение может стать для вас неожиданностью, но это правда. Теперь вы знаете всю механику языка BSL. Вы знаете, что в программировании используется арифметика чисел, строк, изображений и любых других данных, поддерживаемых вашими языками программирования. Вы знаете, что программы состоят из определений функций и констант. Вы знаете, как мы говорили выше, что все дело в правильной организации этих определений. И последнее, но не менее важное: вы знаете, что DrRacket и учебные пакеты поддерживают множество других функций, а документация в HelpDesk описывает эти функции.

Вы можете подумать, что еще недостаточно знаете, чтобы писать программы, реагирующие на нажатия клавиш, щелчки мыши и т. д. И это правда. Кроме `animate`, библиотека `2htdp/universe` содержит множество других функций, которые подключают ваши программы к клавиатуре, мыши, часам и другим механизмам в вашем компьютере. Более того, с ее помощью можно даже писать программы, способные связать ваш компьютер с любым другим компьютером, где бы тот не находился. Так что это не проблема.

Проще говоря, вы познакомились почти со всеми механизмами составления программ. Если вдобавок к этому вы познакомитесь со всеми доступными функциями, то сможете писать программы, играть в интересные компьютерные игры, запускать анимацию или отслеживать бизнес-аккаунты. Вопрос в том, действительно ли это означает, что вы программист. Как вы думаете?

Не спешите перевернуть страницу. Подумайте!

Нет!

Рассматривая полки в разделе «Программирование» в книжном магазине, можно увидеть множество книг, которые обещают превратить вас в программиста. Однако теперь, опробовав несколько первых примеров, вы, вероятно, понимаете, что это невозможно.

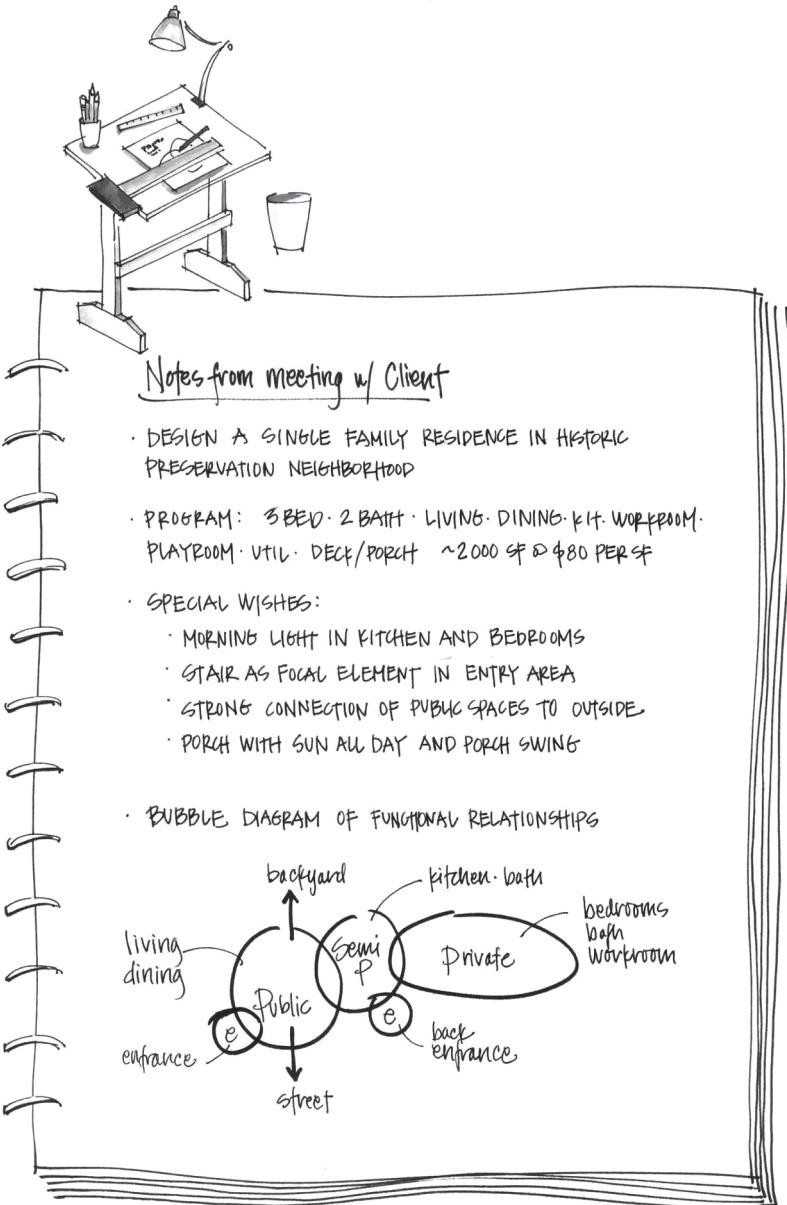
Механические навыки программирования – умение писать выражения, понятные компьютеру, знакомство с доступными функциями и библиотеками и прочие подобные знания и умения – мало помогают в реальном программировании. Если бы это было не так, то вы точно так же могли бы выучить иностранный язык, запомнив тысячу слов из словаря и несколько правил из учебника грамматики.

Умение программировать – это гораздо больше, чем простое знание языка. Особенно важно помнить, что программисты создают программы, которые другие люди смогут читать в будущем. Хорошая программа отражает постановку задачи и ее важные понятия. Она должна включать краткое описание самой себя и сопровождаться примерами, иллюстрирующими это описание и связывающими программу со стоящей перед ней задачей. Наличие примеров гарантирует, что будущий читатель поймет, почему и как работает ваш код. Проще говоря, умение программировать – это системный подход к решению задач и передача этой системы в коде. Самое замечательное, что такой подход делает программирование доступным для всех – он служит сразу двум хозяевам.

Остальная часть книги посвящена всему перечисленному; в ней очень мало внимания уделяется механике DrRacket, BSL или библиотек. Книга показывает, как хорошие программисты размышляют о задачах. И вы узнаете, что такой способ решения задач применим и к другим жизненным ситуациям, таким как работа врачей, журналистов, юристов и инженеров.

И кстати, в остальной части книги используется другой тон, более подходящий для серьезного обсуждения, чем в этом прологе.

Примечание о том, чего вы не найдете в этой книге. Вводные книги по программированию, как правило, содержат много сведений о любимых авторами прикладных дисциплинах: головоломках, математике, физике, музыке и т. д. И это естественно, потому что программирование используется во всех этих областях, но такие сведения одновременно отвлекают от сути программирования. Поэтому мы постарались свести к минимуму использование знаний из других областей и сосредоточиться на решении вычислительных задач.



I ДАННЫЕ ФИКСИРОВАННОГО РАЗМЕРА

Всякий язык программирования включает язык данных и язык операций с данными. Первый всегда определяет некоторые формы атомарных данных для представления разнообразной информации, и программист должен уметь составлять элементарные данные для описания более сложных композиций. Второй язык определяет некоторые базовые операции с атомарными данными, и задача программиста состоит в том, чтобы научиться объединять эти операции в программы, выполняющие желаемые вычисления. Для описания комбинации этих двух частей языка программирования мы используем слово *арифметика*, потому что оно обобщает то, что вы узнали во время учебы в школе.

Эта первая часть книги знакомит с арифметикой языка программирования BSL, с которым мы познакомились в прологе. От арифметики рукой подать до ваших первых простых программ, которые в математике называют функциями. Но прежде чем вы это осознаете, процесс написания программ будет казаться вам запутанным, и вы будете искать способ упорядочить свои мысли. Мы приравниваем «организацию мыслей» к *проектированию*, и эта первая часть книги знакомит вас с систематическим подходом к проектированию программ.

1. Арифметика

Быстро пролистайте эту первую главу и переходите ко второй.

А когда встретите незнакомую вам «арифметику», возвращайтесь сюда.

- ввести «(»,
- ввести имя операции op,

- ввести аргументы, разделяя их пробелами, и
- ввести «)».

Просто ради напоминания, вот простое выражение:

```
| (+ 1 2)
```

Здесь используется операция + сложения, за которой следуют два аргумента – обычные числа. А вот другой пример:

```
| (+ 1 (+ 1 (+ 1 1) 2) 3 4 5)
```

Отметим два важных момента в этом втором примере. Во-первых, операции могут принимать больше двух аргументов. Во-вторых, аргументы необязательно должны быть числами; они могут быть выражениями.

Вычисляются выражения просто. Сначала BSL вычисляет все аргументы операции, а затем передает полученные значения операции, которая возвращает результат. Таким образом:

Мы используем ==, чтобы сказать: «равно, согласно законам вычислений».

и

```
| (+ 1 (+ 1 (+ 1 1) 2) 3 (+ 2 2) 5)
| ==
| (+ 1 (+ 1 2 2) 3 4 5)
| ==
| (+ 1 5 3 4 5)
| ==
| 18
```

Эти вычисления должны быть вам знакомы, потому что подобные вычисления вы производили на уроках математики. Кто-то мог бы записать порядок вычислений иначе, если его не учили выстраивать вычисления в правильной последовательности. Как бы то ни было, BSL выполняет вычисления точно так же, как и вы, и этот факт должен принести вам облегчение. Он гарантирует, что вы понимаете работу элементарных операций с элементарными данными, поэтому есть некоторая надежда, что вы сможете предсказать результаты, вычисляемые вашими программами. Вообще говоря, для программиста

В прологе вы узнали, как записать выражение, знакомое вам с первого класса, следуя правилам языка BSL:

- ввести «(»,
- ввести имя операции op,

важно знать, как выполняет вычисления выбранный им язык, потому что иначе поведение программы может нанести ущерб людям, которые их используют.

В оставшейся части этой главы представлены четыре формы *атомарных данных* в языке BSL: числа, строки, изображения и логические значения. Мы используем слово «атомарный», следуя за аналогией с физикой. Вы не сможете заглянуть внутрь атомарных фрагментов данных, но у вас есть функции, позволяющие объединить несколько фрагментов атомарных данных, извлечь их «свойства», так же в терминах атомарных данных, и т. д. В следующих разделах представлены некоторые из этих функций, еще называемых *примитивными*, или *предопределенные операциями*. Полный перечень функций, доступных в языке BSL, вы найдете в документации, поставляемой с DrRacket.

В следующем томе «How to Design Components» мы расскажем, как проектировать атомарные данные.

1.1. Арифметика чисел

Услышав слово «арифметика», многие начинают думать о «числах» и «операциях с числами». К «операциям с числами» можно отнести: сложение двух чисел для получения третьего, вычитание одного числа из другого, определение наибольшего общего делителя двух чисел и многие другие. Если не воспринимать слово «арифметика» слишком буквально, то в этот список можно также включить вычисление синуса угла, округление действительного числа до ближайшего целого и т. д.

Язык BSL поддерживает числа и арифметику с ними. Как обсуждалось в прологе, арифметические операции, такие как `+`, используются следующим образом:

```
| (+ 3 4)
```

то есть в форме *префиксной записи*. Вот некоторые из операций с числами, которые поддерживает наш язык: `+`, `-`, `*`, `/`, `abs`, `add1`, `ceiling`, `denominator`, `exact->inexact`, `expt`, `floor`, `gcd`, `log`, `max`, `numerator`, `quotient`, `random`, `remainder`, `sqr` и `tan`. Мы прошлись по всему алфавиту, чтобы показать разнообразие операций. Загляните в документацию, чтобы узнать, что они вычисляют и заодно – сколько вообще подобных операций поддерживается.

Если вам понадобится операция с числами, знакомая вам по урокам математики, то, скорее всего, BSL поддерживает ее. Угадайте ее название и поэкспериментируйте в области взаимодействий. Представим, что вам нужно вычислить синус некоторого угла. Вы могли бы попробовать сделать это:

```
| > (sin 0)  
| 0
```

Возможно, вы знакомы с числом e . Это действительное число, примерно 2,718, которое называется «постоянной Эйлера».

а потом долго и счастливо пользоваться своим открытием. Но можно заглянуть в HelpDesk. Там вы обнаружите, что кроме операций язык BSL также распознает имена некоторых широко используемых чисел, например π и e .

Что еще можно сказать о числах? Программы на BSL могут использовать натуральные, целые, рациональные, действительные и комплексные числа. Мы уверены, что вы слышали обо всех этих видах чисел, кроме последнего. Комплексные числа могли упоминаться на уроках математики в старших классах средней школы. Если нет, то не волнуйтесь; несмотря на несомненную пользу комплексных чисел, новичкам необязательно знать о них.

По-настоящему важное различие касается точности чисел. На данный момент важно понимать, что BSL различает *точные* и *неточные* числа. Когда в вычислениях участвуют точные числа, BSL старается сохранить точность. Например, $(/ 4 6)$ дает точную дробь $2/3$, которую DrRacket может отобразить в виде правильной, неправильной или десятичной дроби. Поэкспериментируйте с компьютерной мышкой и найдите пункт в меню, который заменяет дробь десятичной дробью.

Некоторые числовые операции BSL не могут дать точного результата. Например, операция `sqr` над числом 2 дает иррациональное число, которое нельзя описать конечным числом цифр. Поскольку компьютеры имеют ограничения в представлении данных, язык BSL вынужден учитывать эти ограничения и поэтому выводит приближенный результат операции: 1.4142135623730951. Как упоминалось в прологе, об этой неточности начинающих программистов предупреждает префикс `#i`. Однако большинство языков программирования предпочитают жертвовать точностью молча, лишь немногие сообщают о ней в документации и еще меньше предупреждают об этом программистов.

ПРИМЕЧАНИЕ О ЧИСЛАХ. Слово «число» относится к большому разнообразию чисел, включая натуральные, целые, рациональные, действительные и даже комплексные числа. В большинстве случаев слово «число» можно приравнять к «числовой прямой», известной вам из начальной школы, хотя иногда такое сравнение не особенно точное. Для большей точности в выражении своих мыслей мы используем подходящие слова: *целое*, *действительное* и т. д. Мы можем даже уточнять эти понятия, используя такие стандартные термины, как *положительное целое*, *неотрицательное число*, *отрицательное число* и т. д. **КОНЕЦ.**

Упражнение 1. Добавьте следующие определения для x и y в области определений в DrRacket:

```
| (define x 3)
| (define y 4)
```

Теперь представьте, что x и y – это координаты точки. Напишите выражение, которое вычисляет расстояние от этой точки до начала координат, то есть до точки с координатами $(0,0)$.

Правильный результат для этих значений – число 5, но ваше выражение должно давать правильный результат даже после изменения определений x и y .

На всякий случай, если вы не изучали геометрию или забыли формулу, напомним, что расстояние от точки (x,y) до начала координат вычисляется как:

$$\sqrt{x^2 + y^2}.$$

В конце концов, мы учим вас проектировать программы, а не готовим из вас геометров.

Лучший способ прийти к желаемому выражению – щелкнуть на кнопке **RUN** (Выполнить) и поэкспериментировать в области взаимодействий. После щелчка на кнопке **RUN** (Выполнить) DrRacket определит текущие значения x и y , и вы сможете использовать их в своих экспериментах с выражениями:

```
> x  
3  
> y  
4  
> (+ x 10)  
13  
> (* x y)  
12
```

Получив выражение, которое дает правильный результат, скопируйте его из области взаимодействий в область определений.

Чтобы убедиться, что выражение работает правильно, замените число 5 на 12 в определении x и число 4 на число 5 в определении y , а затем щелкните на кнопке **RUN** (Выполнить). В результате должно получиться число 13.

Ваш учитель математики сказал бы, что вы вычислили значение по **формуле расстояния**. Чтобы использовать формулу с другими входными значениями, нужно открыть DrRacket, отредактировать определения x и y , подставив желаемые координаты, и щелкнуть на кнопке **RUN** (Выполнить). Но такой способ повторного использования формулы расстояния слишком громоздкий и неудобный. Вскоре мы покажем вам, как определять функции, которые упрощают повторное использование формул. А здесь мы просто использовали это упражнение, чтобы привлечь внимание к идее функций и подготовить вас к программированию с их помощью. ■

1.2. Арифметика строк

Существует распространенное предубеждение относительно внутреннего устройства компьютеров: многие считают, что все дело в битах и байтах – какими бы они ни были – и, возможно, в числах, пото-

му что все знают, что компьютеры предназначены для вычислений. С одной стороны, это верно, и инженеры-электронщики должны использовать именно такое представление, но начинающие программисты и все остальные никогда не должны делать этого.

Языки программирования предназначены для выполнения вычислений с информацией, а информация может иметь любую форму. Например, программа может работать с цветами, именами, деловыми письмами или бытовой перепиской между людьми. Даже если бы мы могли кодировать такую информацию как числа, то это было бы совершенно неправильно. Только представьте, что вам придется запомнить огромные таблицы с числовыми обозначениями, например 0 означает «красный», а 1 означает «привет» и т. д.

Вместо этого большинство языков программирования поддерживают, по крайней мере, один вид данных для представления такой символьной информации. На данный момент мы используем строки BSL. Вообще говоря, *строка* (String) – это последовательность символов, которые можно вводить с клавиатуры, плюс некоторые другие, которые мы пока не будем трогать, заключенная в двойные кавычки. В прологе мы видели несколько строк на языке BSL: "hello", "world", "blue", "red" и др. Первые две – это слова, которые могут употребляться в разговоре или в письме; остальные – названия цветов, которые мы, возможно, захотим использовать.

ПРИМЕЧАНИЕ. Мы используем термин *1String* (односимвольная строка) для обозначения символов, вводимых с клавиатуры и составляющих строку. Например, "red" состоит из трех таких *1String*: "r", "e", "d". В действительности *1String* – это нечто большее, но сейчас будем представлять данные этого типа как строки, состоящие из одного символа. **КОНЕЦ.**

В языке BSL есть только одна операция, принимающая и возвращающая исключительно строки: *string-append*, которая, как мы видели в прологе, объединяет две строки в одну. Операцию *string-append* можно считать операцией сложения строк, похожей на +, только, в отличие от последней, принимающей два (или более) числа и возвращающей новое число, первая принимает две или более строк и возвращает новую строку:

```
| > (string-append "what a" "lovely" "day" " 4 BSL")
| "what a lovely day 4 BSL"
```

Исходные числа не меняются, когда складываются операцией +, и исходные строки не меняются, когда объединяются операцией *string-append*. Если вам понадобится вычислять такие выражения в уме, то просто помните, что при сложении строк используются очевидные законы, аналогичные законам для +:

```
| (+ 1 1) == 2 (string-append "a" "b") == "ab"
| (+ 1 2) == 3 (string-append "ab" "c") == "abc"
| (+ 2 2) == 4 (string-append "a" " ") == "a "
| ...
| ...
```

Упражнение 2. Добавьте следующие две строки в область определений:

```
| (define prefix "hello")
| (define suffix "world")
```

Затем используйте элементарные операции со строками, чтобы создать выражение, которое объединяет `prefix` и `suffix` и вставляет `"_"` между ними. Получившаяся в результате программа должна после запуска выводить `"hello_world"` в области взаимодействий.

См. упражнение 1, где показано, как создавать выражения в DrRacket. ■

1.3. А теперь все смешаем

Все остальные операции со строками (в языке BSL) принимают или возвращают данные, не являющиеся строками. Вот несколько примеров:

- `string-length` принимает строку и возвращает число;
- `string-ith` принимает строку `s` и число `i` и возвращает 1String (символ), находящийся в `i`-й позиции в строке `s` (счет начинается с 0);
- `number->string` принимает число и возвращает строку.

Также обратите внимание на `substring` и узнайте, что она делает.

Если документация в HelpDesk покажется вам путаной, поэкспериментируйте с функциями в области взаимодействий. Передайте им подходящие аргументы и выясните, что они вычисляют. Также попробуйте передать **неподходящие** аргументы, чтобы узнать, как на это реагирует BSL:

```
| > (string-length 42)
| string-length:expects a string, given 42
```

(`string-length`: ожидалась строка, а получено число 42).

Как видите, в таких случаях BSL сообщает об ошибке. В первой части сообщения («`string-length`») указывается название операции, в которой обнаружилась ошибка, а во второй половине описывается сама ошибка. В данном конкретном примере BSL сообщает, что `string-length` должна применяться к строке, а мы передали ей число 42.

Операции можно вкладывать друг в друга, **если следить за тем, чтобы передавались подходящие данные**. Вернемся к выражению из пролога:

```
| (+ (string-length "hello world") 20)
```

Внутреннее выражение применяет `string-length` к `"hello world"` – нашей любимой строке. Внешнее выражение `+` получает результат вложенного выражения и число 20.

Давайте пройдем это выражение по шагам и определим его результат:

```
(+ (string-length "hello world") 20)
==
(+ 11 20)
==
31
```

Как видите, вычисления с такими вложенными выражениями, обрабатывающими данные разных типов, ничем не отличаются от вычислений с числовыми выражениями. Вот еще один пример:

```
(+ (string-length (number->string 42)) 2)
==
(+ (string-length "42") 2)
==
(+ 2 2)
==
4
```

Прежде чем продолжить, попробуйте создать несколько вложенных выражений, которые **неправильно** смешивают данные, например:

```
| (+ (string-length 42) 1)
```

Запустите их в DrRacket. Прочтайте сообщение об ошибке, а также обратите внимание, какие области подсвечиваются в области определений.

Упражнение 3. Добавьте следующие две строчки кода в область определений:

```
| (define str "helloworld")
| (define i 5)
```

Затем, используя операции со строками, создайте выражение, которое добавляет символ "_" в строку str в позицию i. В результате должна получиться строка длиннее исходной; ожидаемый результат: "hello_world".

Под термином *позиция* подразумевается символ, находящийся на i-м месте слева от начала, но программисты начинают счет с 0, поэтому 5-я буква в этом примере – "w", потому что 0-я буква – "h". **Подсказка.** Столкнувшись с подобной «проблемой отсчета», выпишите символы строки и подпишите ниже их номера, начав с 0, это облегчит подсчет:

```
| (define str "helloworld")
| (define ind "0123456789")
| (define i 5)
```

См. упражнение 1, где показано, как создавать выражения в DrRacket. ■

Упражнение 4. Используйте те определения, что и в упражнении 3, и создайте выражение, удаляющее из str символ в *i*-й позиции. Очевидно, что это выражение создаст строку короче исходной. Какие значения *i* допустимы? ■

1.4. Арифметика изображений

Изображение – это прямоугольный фрагмент визуальных данных, например фотография или геометрическая фигура и ее рамка. Изображения можно вставлять в DrRacket везде, где можно вставлять выражения, потому что изображения являются значениями, такими же как числа и строки.

Открыв новую вкладку, не забудьте подключить библиотеку `2htdp/image`.

Ваши программы могут манипулировать изображениями с помощью элементарных операций трех видов. Операции первого вида создают элементарные изображения:

- `circle` создает изображение круга из радиуса, строку, определяющую необходимость заливки, и строку с названием цвета;
- `ellipse` создает эллипс и принимает два диаметра, строку, определяющую необходимость заливки, и строку с названием цвета;
- `line` создает отрезок по двум точкам и строке с названием цвета;
- `rectangle` создает прямоугольник и принимает ширину, высоту, строку режима и строку с названием цвета;
- `text` создает текстовое изображение и принимает строку с текстом, размер шрифта и строку с названием цвета;
- `triangle` создает равносторонний треугольник, направленный вверх, и принимает размер, строку режима и строку с названием цвета.

Названия этих операций однозначно определяют создаваемое изображение. Вам только нужно запомнить строки режима `"solid"` (со сплошной заливкой цветом) и `"outline"` (только контур) и строки цветов, такие как `"orange"` (оранжевый), `"black"` (черный) и т. д.

Попробуйте с этими операциями в окне взаимодействий:

```
> (circle 10 "solid" "green")
●
> (rectangle 10 20 "solid" "blue")
█
> (star 12 "solid" "gray")
★
```

А теперь взгляните еще раз на примеры выше! В последнем примере используется не упомянутая выше операция. Загляните в докумен-

тацию (<https://docs.racket-lang.org/teachpack/2htdpimage.html>) и узнайте, сколько еще таких операций имеется в библиотеке *2htdp/image*. Поэкспериментируйте с этими операциями.

Операции второго вида возвращают свойства изображений:

- `image-width` определяет ширину изображения в пикселях;
- `image-height` определяет высоту изображения.

Они извлекают эти значения непосредственно из изображений, например:

```
| > (image-width (circle 10 "solid" "red"))
| 20
| > (image-height (rectangle 10 20 "solid" "blue"))
| 20
```

А теперь остановитесь и объясните, что вернет DrRacket, если ввести следующее выражение:

```
| (+ (image-width (circle 10 "solid" "red"))
|   (image-height (rectangle 10 20 "solid" "blue")))
```

Для правильного понимания третьего вида операций с изображениями необходимо познакомиться с одной новой идеей: *точкой привязки*. Изображение – это не единственный пиксель, оно состоит из множества пикселей. Каждое изображение чем-то похоже на фотографию, то есть на прямоугольник, заполненный пикселями. Один из этих пикселей считается точкой привязки. При использовании операций, объединяющих два изображения, объединение осуществляется относительно точек привязки, если явно не указать какую-либо другую точку:

- `overlay` накладывает все изображения, перечисленные в операции, друг на друга, используя центр в качестве точки привязки;
- `overlay/x/y` подобна операции `overlay`, но принимает два числа – `x` и `y` – между двумя аргументами с изображениями. Она сдвигает второе изображение на `x` пикселей вправо и на `y` пикселей вниз относительно верхнего левого угла первого изображения; отрицательное значение `x` вызывает сдвиг второго изображения влево, а отрицательное значение `y` – вверх;
- `overlay/align` подобна операции `overlay`, но принимает две строки, которые смещают точки привязки указанных изображений. Всего существует девять разных позиций; поэкспериментируйте с ними!

Библиотека *2htdp/image* содержит множество других элементарных функций для объединения изображений. Когда захотите познакомиться с ними поближе, вам придется прочитать документацию с их описанием. А пока мы представим еще три операции, которые могут пригодиться для создания анимированных сцен и изображений для игр:

- `empty-scene` создает прямоугольник заданной ширины и высоты;
- `place-image` помещает изображение в сцену в указанное место. Если изображение не помещается в сцену, оно будет соответствующим образом обрезано;
- `scene+line` принимает сцену, четыре числа и цвет и рисует линию на указанном изображении. Поэкспериментируйте самостоятельно, чтобы увидеть, как работает эта операция.

Законы арифметики изображений аналогичны законам арифметики чисел; см. табл. 1, где приводится несколько примеров и сравнение с арифметикой чисел. Повторю еще раз, что ни одна операция не изменяет и не уничтожает исходное изображение. Так же как `+`, эти операции просто создают новые изображения, которые определенным образом объединяют исходные данные.

Таблица 1. Правила создания изображений

Арифметика чисел	Арифметика изображений
$(+ 1 1) == 2$	<code>(overlay (square 4 "solid" "orange") (circle 6 "solid" "yellow"))</code> == 
$(+ 1 2) == 3$	<code>(underlay (circle 6 "solid" "yellow") (square 4 "solid" "orange"))</code> == 
$(+ 2 2) == 4$	<code>(place-image (circle 6 "solid" "yellow") 10 10 (empty-scene 20 20))</code> == 
...	...

Упражнение 5. Воспользуйтесь библиотекой `2htdp/image` и создайте изображение простой лодки или дерева. Предусмотрите простую возможность изменения размеров изображения. ■

Упражнение 6. Добавьте следующую строку в область *Скопируйте и вставьте изображение в DrRacket.*



Создайте выражение, подсчитывающее пиксели в изображении. ■

1.5. Арифметика логических значений

Прежде чем мы сможем проектировать программы, нам нужно познакомиться с последним видом элементарных данных: *логическими (boolean) значениями*. Существует только два вида логических значений: `#true` и `#false`. Программы используют логические значения для представления решений или состояния переключателей.

Вычисления с логическими значениями тоже очень просты. В частности, программы на BSL используют в основном три операции: `or`, `and` и `not`. Эти операции похожи на сложение, умножение и изменение знака чисел. Конечно, поскольку существует всего два логических значения, мы имеем возможность продемонстрировать работу этих функций во всех возможных ситуациях:

- `or` проверяет, есть ли `#true` среди заданных логических значений:

```
> (or #true #true)
#true
> (or #true #false)
#true
> (or #false #true)
#true
> (or #false #false)
#false
```

- `and` проверяет, равны ли **все** указанные логические значения значению `#true`:

```
> (and #true #true)
#true
> (and #true #false)
#false
> (and #false #true)
#false
> (and #false #false)
#false
```

- `not` всегда выбирает другое логическое значение, отличающееся от заданного:

```
> (not #true)
#false
```

Неудивительно, что операции `or` и `and` могут принимать больше двух выражений. Наконец, операции `or` и `and` имеют еще ряд отличительных особенностей, но для их объяснения необходимо вернуться к вложенным выражениям.

Формулировку этого упражнения предложил

Надим Хамид (*Nadeem Hamid*). Упражнение 7. Логические выражения могут выражать некоторые повседневные проблемы. Предположим, вам нужно решить, подходит ли сегодняшний день для посещения

торгового центра. Вы ходите в торговый центр либо когда пасмурно, либо по пятницам (потому что именно по пятницам в магазинах проводятся распродажи).

Теперь попробуйте принять решение, используя новые знания о логических значениях. Сначала добавьте эти две строки в область определений DrRacket:

```
| (define sunny #true)
| (define friday #false)
```

Теперь создайте выражение, которое проверит, что `sunny` имеет значение `#false` или `friday` имеет значение `#true`. В данном случае результат должен получиться равным `#false`. (Почему?)

См. упражнение 1, где показано, как создавать выражения в DrRacket. Сколько всего разных комбинаций из значений `sunny` и `friday` может быть? ■

1.6. Смешанные операции с логическими значениями

Одно из важных применений логических значений – помочь в вычислениях с другими видами данных. В прологе уже говорилось, что в программах на BSL можно давать значениям имена с помощью определений. Например, программа может начинаться с определения

```
| (define x 2)
```

и затем вычислять обратную величину:

```
| (define inverse-of-x (/ 1 x))
```

И все будет хорошо, пока мы не отредактируем программу и не изменим значение `x` на `0`.

В таких ситуациях нам могут помочь логические значения, в частности условные вычисления. Сначала с помощью элементарной функции `=` можно проверить равенство двух (или более) чисел. Если они равны, то операция `=` вернет `#true`, иначе – `#false`. Затем использовать разновидность выражения на BSL, о которой мы пока не упоминали: выражение `if`. В нем используется слово «`if`», как если бы это была элементарная функция, но это не так. За словом «`if`» следуют три выражения, разделенных пробельными символами (включая табуляцию, разрывы строк и т. д.). Естественно, все выражение заключено в круглые скобки, например:

```
| (if (= x 0) 0 (/ 1 x))
```

Это выражение `if` содержит три подвыражения: `(= x 0)`, `0` и `(/ 1 x)`. Вычисление этого выражения происходит в два этапа:

- Первое подвыражение вычисляется всегда. Его результат должен быть логическим значением.

- Если первое подвыражение возвращает #true, то вычисляется второе подвыражение; в противном случае – третье. Результат

Щелкнув правой кнопкой мыши на результате, вы сможете выбрать

другую форму его

Ведите определение x , показанное выше, и поэкспериментируйте с выражением if в области взаимодействий:

```
| > (if (= x 0) 0 (/ 1 x))
| 0.5
```

Опираясь на законы арифметики, вы можете сами предугадать результат:

```
(if (= x 0) 0 (/ 1 x))
== ; поскольку x имеет значение 2
(if (= 2 0) 0 (/ 1 2))
== ; 2 не равно 0, подвыражение (= 2 0) даст #false
(if #false 0 (/ 1 x))
(/ 1 2)
== ; после нормализации в десятичное представление получается
0.5
```

Другими словами, DrRacket знает, что x обозначает 2, а оно не равно 0. Поэтому ($= x 0$) вернет #false, и функция if выберет третье подвыражение для этапа вычислений.

А сейчас представьте, что вы исправили определение x так, что теперь оно выглядит следующим образом:

```
| (define x 0)
```

Какое значение теперь вернет наше условное выражение?

```
| (if (= x 0) 0 (/ 1 x))
```

Почему? Напишите на листке бумаги последовательность вычислений, как она видится вам.

Кроме $=$, в BSL имеется множество других элементарных операций сравнения. Объясните, что делают следующие четыре операции сравнения в отношении чисел: $<$, \leq , $>$, \geq .

Строки нельзя сравнивать с помощью $=$ и родственных ей операций. Вместо этого надо использовать $string=?$, $string<=?$ или $string>=?$. Совершенно очевидно, что $string=?$ проверяет равенство двух заданных строк, но две другие операции требуют пояснений. Загляните в документацию с их описанием. Или экспериментальным путем определите общие закономерности, а затем проверьте свои выводы, заглянув в документацию.

У кого-то может возникнуть вопрос: зачем вообще сравнивать строки друг с другом? Представьте программу, которая управляет светофор-

рами. Она может использовать строки "green", "yellow" и "red" для обозначения цветов. Программа может содержать такой фрагмент:

```
| (define current-color ...)

| (define next-color
  (if (string=? "green" current-color) "yellow" ...))
```

Точки в определении `current-color`, конечно же, не являются частью программы. Замените их строкой с названием цвета.

Легко представить, что этот фрагмент связан с вычислениями, которые определяют, какую лампочку нужно включить, а какую выключить.

В следующих нескольких главах мы рассмотрим более эффективные способы выражения условных вычислений, чем `if`, и, что особенно важно, системные способы их проектирования.

Упражнение 8. Добавьте следующую строку в область определений:



```
| (define cat )
```

Создайте условное выражение, которое определяет, какая сторона изображения больше – ширина или высота. Изображение должно быть помечено как "tall" (высокое), если его высота больше или равна ширине; иначе метка должна быть строкой "wide" (широкое). См. упражнение 1, где показано, как создавать выражения в DrRacket. В ходе экспериментов замените изображение кота прямоугольником по вашему выбору и убедитесь, что ваше выражение возвращает правильный ответ.

После этого попробуйте изменить выражение так, чтобы оно определяло, является ли изображение "tall" (высоким), "wide" (широким) или "square" (квадратным). ■

1.7. Предикаты: знай свои данные

Вспомните выражение (`string-length 42`) и его результат. На самом деле это выражение не дает результата, оно сообщает об ошибке. DrRacket выводит сообщения об ошибках красным цветом в области взаимодействий и выделяет ошибочные выражения (в области определений). Этот способ выделения ошибок особенно полезен, когда ошибочное выражение глубоко вложено в какое-то другое выражение:

```
| (* (+ (string-length 42) 1) pi)
```

Поэкспериментируйте с этим выражением, введя его в область взаимодействий и в область определений (а затем щелкните на кнопке **RUN** (Выполнить)).

Конечно, никому не хочется, чтобы в его программе были подобные выражения, сигнализирующие об ошибках. И обычно мало кто допускает такие очевидные ошибки, как использование числа 42 вместо строки. Однако довольно часто программы имеют дело с переменными, которые могут хранить число или строку:

```
| (define in ...)
| (string-length in)
```

Переменная, такая как `in`, может играть роль заменителя любого значения, включая число, и использоваться в выражении `string-length`.

Один из способов предотвратить подобные случайности – использовать *предикат*, то есть функцию, которая принимает значение и определяет, принадлежит ли оно какому-либо классу данных. Например, предикат `number?` определяет, является ли данное значение числом:

```
| > (number? 4)
| #true
| > (number? pi)
| #true
| > (number? #true)
| #false
| > (number? "fortytwo")
| #false
```

Как видите, предикаты возвращают логические значения. Поэтому, комбинируя предикаты с условными выражениями, можно предотвратить неправильное использование выражений:

```
| (define in ...)
| (if (string? in) (string-length in) ...)
```

*Ведите выражение (`sqrt -1`) в области взаимодействий и нажмите клавишу **Enter**. Посмотрите, что получилось в результате. Вы должны увидеть так называемое комплексное*

число, с которым рано или поздно сталкивается каждый. Ваш учитель математики мог говорить вам, что нельзя вычислить квадратный корень из отрицательного числа, но правда в том, что математики и некоторые программисты уверены в обратном. Не волнуйтесь: понимание особенностей комплексных чисел не является обязательным требованием для проектировщиков программ.

Для всех классов данных, с которыми мы познакомились в этой главе, имеются соответствующие предикаты. Поэкспериментируйте с `number?`, `string?`, `image?` и `boolean?`, чтобы понять, как они работают.

В дополнение к предикатам, которые различают разные формы данных, языки программирования также имеют предикаты, различающие разные типы чисел. В BSL числа классифицируются по строению и по точности. Под строением понимаются знакомые всем числа: целые, рациональные, действительные и комплексные, но многие языки программирования, включая BSL, также используют конечные приближения хорошо известных

констант, что приводит к несколько неожиданным результатам с предикатом `rational?` :

```
| > (rational? pi)  
#true
```

Что касается точности, то мы уже упоминали это понятие. А теперь поэкспериментируйте с предикатами `exact?` и `inexact?` , чтобы убедиться, что они выполняют проверки, о которых говорят их имена (точное число и неточное число соответственно). Позже мы обсудим природу чисел более подробно.

Упражнение 9. Добавьте следующую строку в область определений в DrRacket:

```
| (define in ...)
```

Затем создайте выражение, преобразующее значение `in` в положительное число. Для строки в переменной `in` выражение должно вычислять длину строки; для изображения – площадь; для числа оно должно уменьшать это число на 1, если оно не равно 0 или неотрицательное; для `#true` должно возвращаться значение 10, а для `#false` – значение 20. *Подсказка:* прочтите (еще раз) описание условного выражения `cond` в разделе «Пролог: как писать программы».

См. упражнение 1, где показано, как создавать выражения в DrRacket. ■

Упражнение 10. Теперь отдохните, поешьте, послите и переходите к следующей главе. ■

2. Функции и программы

«Арифметика» в программировании – это только половина дела; другая половина – «алгебра». В данном случае слово «алгебра» относится к школьному предмету алгебры так же, как понятие «арифметика» из предыдущей главы относится к школьному предмету арифметики. В частности, алгебра в программировании включает такие понятия, как переменная, определение функции, применение функции и композиция функций. Эта глава повторно познакомит вас с этими понятиями в увлекательной и доступной форме.

2.1. Функции

Программы – это функции. Так же как функции, программы принимают входные данные и возвращают результат. В отличие от функций, с которыми вы, возможно, хорошо знакомы, программы работают с разными данными: числами, строками, изображениями и их комбинациями. Кроме того, программы могут реагировать на события в реальном мире, а их результаты – влиять на реальный мир. Например, программа для работы с электронными таблицами может реагировать на нажатия клавиш бухгалтером и заполнять некоторые ячейки числами, или программа-календарь может запускать программу ежемесячного расчета заработной платы в последний день каждого месяца. Наконец, программа не может использовать все свои входные данные сразу, вместо этого она обрабатывает их поэтапно.

Определения. Многие языки программирования скрывают связь между программами и функциями, но BSL, наоборот, выдвигает ее на передний план. Каждая программа на языке BSL состоит из нескольких определений, за которыми обычно следует выражение, использующее эти определения. Есть два вида определений:

- *определения констант* в форме (define Переменная Выражение), которые мы видели в предыдущей главе;
- *определения функций*, которые бывают разных видов; один из них мы использовали в прологе.

Подобно выражениям, определения функций в BSL имеют единобразную форму:

```
(define (Имяфункции Переменная ... Переменная)
      Выражение)
```

То есть, чтобы определить функцию, нужно ввести:

- «(define (»,
- имя функции,

- необходимые переменные, перечислив их через пробел, и завершить список переменных закрывающей круглой скобкой «)»,
- затем выражение и закрывающую скобку «)».

Вот несколько коротких примеров:

- (define (f x) 1)
- (define (g x y) (+ 1 1))
- (define (h x y z) (+ (* 2 2) 3))

Прежде чем объяснить, почему эти примеры не имеют практической ценности, мы должны рассказать, что означают определения функций. Если выражаться простым языком, то определение функции вводит новую операцию с данными, то есть добавляет новую операцию в наш словарь, содержащий элементарные операции, которые всегда доступны. Подобно элементарной функции, определяемая нами функция принимает входные данные. Количество переменных определяет количество входных данных, также называемых *аргументами* или *параметрами*. Таким образом, f – это функция с одним аргументом, такие функции иногда называют *унарными* функциями. Функция g – это функция с двумя аргументами, такие функции иногда называют *бинарными*, а h – это функция с тремя аргументами, или *тернарная* функция. Выражение, которое часто называют *телом функции*, определяет результат.

Примеры демонстрируют бесполезные функции, потому что выражения внутри них не включают переменные. Поскольку переменные относятся к входным параметрам, отсутствие упоминания их в выражениях означает, что выходные данные функции не зависят от их входных данных и, следовательно, всегда одинаковы. Нам не нужно писать функции или программы, если результат всегда один и тот же.

Переменные – это не данные; они представляют данные. Например, определение константы, такое как

```
| (define x 3)
```

говорит, что x всегда имеет значение 3. Переменные в заголовке функции, то есть следующие за именем функции, являются заглушками и представляют пока **неизвестные** входные данные. Ссылки на переменные в теле функции – это способ использовать эти данные во время применения функции, когда значения переменных становятся известными.

Рассмотрим следующий фрагмент определения:

```
| (define (ff a) ...)
```

Заголовок этой функции – (ff a) – означает, что ff принимает одно входное значение, а переменная a является представителем этого значения. Определяя функцию, мы не знаем, какие значения будут

иметь входные данные. На самом деле весь смысл определения функции состоит в том, чтобы дать возможность использовать функцию много раз для множества разных входных данных.

Тело функции ссылается на ее параметры. Ссылка на параметр функции на самом деле является ссылкой на входные данные функции. Если завершить определение `ff` следующим образом:

```
| (define (ff a)
|   (* 10 a))
```

то тем самым мы скажем, что результатом функции является число, в десять раз большее входного числа. Вероятно, этой функции будут передаваться числа, потому что нет смысла умножать на 10 изображения, логические значения или строки.

На данный момент нам осталось прояснить единственный вопрос – как функция получает входные данные. С этой целью обратимся к понятию применения функции.

Применение. Применение функции заставляет определения функций работать и выглядит так же, как применение любой предопределенной операции:

- введите «();
- введите имя функции, например `f`;
- добавьте столько аргументов, сколько может принять функция, перечислив их через пробел;
- и добавьте «)» в конце.

Теперь, чтобы закрепить новые знания, поэкспериментируйте с функциями в области взаимодействий, как мы это делали с элементарными операциями, и убедитесь, что понимаете, как они работают. Например, следующие три эксперимента подтверждают, что функция `f`, которую мы определили выше, возвращает одно и то же значение при применении к любым данным:

```
| > (f 1)
| 1
| > (f "hello world")
| 1
| > (f #true)
| 1
```

Не забудьте добавить выражение (`require 2htdp/image`) в области определений.

Что вернет выражение `(f (circle 3 "solid" "red"))?`

Как видите, поведение функции `f` не меняется, даже если ее применить к изображению. Но вот что произойдет, если попытаться применить функцию к слишком малому или слишком большому количеству аргументов:

```
| > (f)
| f:expects 1 argument, found none (f: ожидается 1 аргумент, не найдено ни одного)
| > (f 1 2 3 4 5)
| f:expects only 1 argument, found 5 (f: ожидается только 1 аргумент, найдено 5)
```

DrRacket сигнализирует об ошибке сообщением, которое похоже на те, что выводятся при применении элементарной операции к неправильному количеству аргументов:

```
> (+)
+:expects at least 2 arguments, found none (+: ожидается по меньшей мере 2 аргумента, не
найдено ни одного)
```

Функции можно применять не только в приглашении к вводу в области взаимодействий. Их также можно применять внутри вложенных выражений:

```
> (+ (ff 3) 2)
32
> (* (ff 4) (+ (ff 3) 2))
1280
> (ff (ff 1))
100
```

Упражнение 11. Определите функцию, которая принимает два числа, x и y , и вычисляет расстояние от точки (x, y) до начала координат.

В упражнении 1 мы разработали правую часть этой функции для конкретных значений x и y . Теперь добавьте заголовок. ■

Упражнение 12. Определите функцию `cvolume`, которая принимает длину ребра куба и вычисляет его объем. Если у вас есть время, определите также функцию `csurface`, вычисляющую площадь поверхности куба.

Подсказка. Куб – это трехмерная объемная фигура, ограниченная шестью квадратами – гранями. Площадь поверхности куба легко определить, зная площадь одного квадрата, которая равна квадрату длины его стороны. Объем куба – это произведение длины ребра на площадь одной грани. (Почему?) ■

Упражнение 13. Определите функцию `string-first`, которая извлекает первый символ (`1String`) из **непустой** строки. ■

Упражнение 14. Определите функцию `string-last`, которая извлекает последний символ (`1String`) из непустой строки. ■

Упражнение 15. Определите функцию `==>`. Она должна принимать два логических значения (пусть это будут параметры с именами `sunny` и `friday`) и возвращать `#true`, если `sunny` имеет ложное значение или `friday` – истинное. **Примечание.** В алгебре логики эта логическая операция называется *импликацией* и обозначается как `sunny => friday`. ■

Упражнение 16. Определите функцию `image-area`, которая подсчитывает количество пикселей в заданном изображении. Идеи, как это можно реализовать, см. в упражнении 6. ■

Упражнение 17. Определите функцию `image-classify`, которая принимает изображение и возвращает "tall" (высокое), если высота изображения больше ширины; "wide" (широкое), если высота изображения меньше ширины, и "square" (квадратное), если ширина равна высоте. Идеи, как это можно реализовать, см. в упражнении 8. ■

Упражнение 18. Определите функцию `string-join`, которая принимает две строки и объединяет их в одну строку, добавляя символ "_" между ними. Идеи, как это можно реализовать, см. в упражнении 2. ■

Упражнение 19. Определите функцию `string-insert`, которая принимает строку `str` и число `i` и вставляет "_" в *i*-ю позицию строки `str`. Предполагается, что `i` – это число в диапазоне от 0 до длины заданной строки (включительно). Идеи, как это можно реализовать, см. в упражнении 3. Подумайте, как `string-insert` может справиться со вставкой "" . ■

Упражнение 20. Определите функцию `string-delete`, которая принимает строку `str` и число `i` и удаляет из `str` символ в *i*-й позиции. Предполагается, что `i` – это число в диапазоне от 0 до длины данной строки (не включая). Идеи, как это можно реализовать, см. в упражнении 4. Сможет ли `string-delete` работать с пустыми строками? ■

2.2. Вычисления

Определение и применение функций всегда идут рука об руку. Если вы хотите проектировать программы, то должны понимать эту взаимосвязь, потому что вам придется в уме представлять, как DrRacket выполняет ваши программы, и, соответственно, замечать, что идет не так, когда что-то идет не так, а рано или поздно что-то **обязательно** пойдет не так.

Возможно, вы уже сталкивались с этой идеей на уроках алгебры, тем не менее мы попробуем объяснить ее по-своему. Итак, поехали. Применение функции происходит в три этапа: DrRacket вычисляет значения выражений аргументов; проверяет равенство числа аргументов и числа параметров функции; и если это условие соблюдается, то вычисляет значение тела функции, заменяя все параметры значениями соответствующих аргументов. Это последнее значение является значением выражения применения функции. Данное объяснение выглядит довольно сложным, поэтому обратимся к примерам.

Бот пример применения функции `f`:

```
(f (+ 1 1))
== ; DrRacket знает, что (+ 1 1) == 2
(f 2)
== ; DrRacket заменяет все вхождения x числом 2
1
```

Последнее уравнение выглядит странным, потому что `x` не используется в теле `f`. В результате замена вхождения `x` на 2 в теле функции дает в результате 1 – число, которое является телом функции.

Для `ff` вычисления выполняются несколько иначе:

```
(ff (+ 1 1))
== ; DrRacket знает, что (+ 1 1) == 2
(ff 2)
```

```

== ; DrRacket заменяет все вхождения x в теле ff числом 2
(* 10 2)
== ; а затем DrRacket выполняет самую обычную арифметическую операцию
20

```

Самое замечательное, что, объединяя эти законы вычислений с законами арифметики, можно успешно предсказать результат любой программы на BSL:

```

(+ (ff (+ 1 2)) 2)
== ; DrRacket знает, что (+ 1 2) == 3
(+ (ff 3) 2)
== ; DrRacket заменяет все вхождения x в теле ff числом 3
(+ (* 10 3) 2)
== ; теперь DrRacket применяет законы арифметики
(+ 30 2)
==
32

```

Естественно, полученный результат можно использовать в других вычислениях:

```

(* (ff 4) (+ (ff 3) 2))
== ; DrRacket заменяет все вхождения x в теле ff числом 4
(* (* 10 4) (+ (ff 3) 2))
== ; DrRacket знает, что (* 10 4) == 40
(* 40 (+ (ff 3) 2))
== ; теперь используется результат, вычисленный выше
(* 40 32)
==
1280 ; потому что это просто математика

```

В целом можно сказать, что DrRacket прекрасно владеет алгеброй, знает все законы арифметики и отлично справляется с заменой. Более того, DrRacket может не только определять значения выражений, но также может показать, **как** это делается, то есть может показать пошаговое решение алгебраической задачи, которая должна определить значение выражения.

Еще раз взгляните на кнопки в окне DrRacket. Одна из них выглядит как кнопка «перейти к следующему треку» на аудиоплеере. Если щелкнуть на этой кнопке, то появится окно **движка пошаговых вычислений**, в котором можно проверить порядок выполнения программы в области определений.

Ведите в область определений определение функции ff. Там же, в области определений, ниже определения функции, добавьте выражение (ff (+ 1 1)). Теперь щелкните на кнопке **STEP** (Шаг). Появится окно движка пошаговых вычислений; на рис. 3 показано, как выглядит это окно в версии DrRacket 6.2. Далее можно использовать стрелки вперед и назад, чтобы увидеть все этапы вычислений, выполняемые для определения значения выражения. Обратите внимание, что движок пошаговых вычислений выполняет те же вычисления, что и мы.

Да, вы могли бы использовать DrRacket для решения домашних заданий по алгебре! Поэкспериментируйте с разными вариантами, которые предлагает движок.

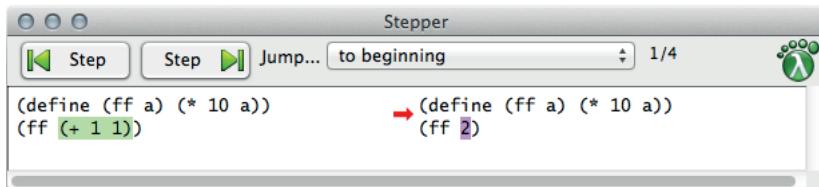


Рис. 3. Движок пошаговых вычислений в DrRacket

Упражнение 21. Используйте движок пошаговых вычислений в DrRacket, чтобы увидеть, как вычисляется выражение $(ff (ff 1))$. Также попробуйте выражение $(+ (ff 1) (ff 1))$. Использует ли движок пошаговых вычислений в DrRacket результаты вычислений повторно? ■

Читая обсуждение всех этих вычислений, включающих скучные функции и числа, вы можете подумать, что снова изучаете курс алгебры. К счастью, этот подход распространяется на **все** программы, включая самые интересные, в данной книге.

Для начала рассмотрим функции, обрабатывающие строки. Вспомним некоторые законы арифметики строк:

```
(string-append "hello" " " "world") == "hello world"
(string-append "bye" ", " "world") == "bye, world"
...
```

Теперь допустим, что мы определили функцию, которая создает начало письма:

```
(define (opening first-name last-name)
  (string-append "Dear " first-name ","))
```

Применив эту функцию к двум строкам, вы получаете начало письма:

```
> (opening "Matthew" "Fisler")
"Dear Matthew,"
```

Однако самое важное – как законы вычислений определяют этот результат и как, используя их, можно предвидеть, что сделает DrRacket:

```
(opening "Matthew" "Fisler")
== ; DrRacket подставляет "Matthew" вместо first-name
(string-append "Dear " "Matthew" ",")
==
"Dear Matthew,"
```

Поскольку `last-name` не встречается в определении функции `opening`, подстановка "Fisler" не имеет никакого эффекта.

Далее в книге мы познакомимся с другими формами данных. Для объяснения выполнения операций с данными всегда используются законы, подобные тем, которые действуют в арифметике в этой книге.

Упражнение 22. Используйте движок пошаговых вычислений в DrRacket, чтобы увидеть, как выполняется следующий фрагмент программы:

```
(define (distance-to-origin x y)
  (sqrt (+ (sqr x) (sqr y))))
(distance-to-origin 3 4)
```

Соответствует ли ход вычислений вашему пониманию? ■

Упражнение 23. Первый символ (1String) в строке "hello world" – буква "h". Как следующая функция вычисляет этот результат?

```
(define (string-first s)
  (substring s 0 1))
```

Подтвердите свои рассуждения с помощью движка пошаговых вычислений. ■

Упражнение 24. Вот определение функции `==>`:

```
(define (==> x y)
  (or (not x) y))
```

С помощью движка пошаговых вычислений определите значение выражения `(==> #true #false)`. ■

Упражнение 25. Взгляните на следующую попытку решить упражнение 17:

```
(define (image-classify img)
  (cond
    [(>= (image-height img) (image-width img)) "tall"]
    [(= (image-height img) (image-width img)) "square"]
    [(<= (image-height img) (image-width img)) "wide"]))
```

Помогает ли пошаговое выполнение применения функции предложить исправление? ■

Упражнение 26. Какое значение вернет эта программа:

```
(define (string-insert s i)
  (string-append (substring s 0 i)
                " "
                (substring s i)))

(string-insert "helloworld" 6)
```

Подтвердите свои рассуждения с помощью движка пошаговых вычислений. ■

Рано или поздно вы столкнетесь с императивными операциями, которые не объединяют и не извлекают значения, а изменяют их. Чтобы производить вычисления с помощью таких операций, вам понадобится изучить дополнительные законы, помимо законов арифметики и подстановки.

2.3. Композиция функций

Редкая программа состоит из определения единственной функции. Подавляющее большинство программ состоит из определения *главной функции* и множества определений других функций, и практически всегда результат одной функции передается на вход другой функции. По аналогии с алгеброй мы называем такой способ определения функций *композицией*, а дополнительные функции – *вспомогательными функциями*.

Рассмотрим программу в листинге 7, которая формирует шаблон письма. Она состоит из четырех функций. Первая из них – это главная функция, она создает полный шаблон письма, включающий имя и фамилию адресата плюс подпись. Главная функция обращается к трем вспомогательным функциям, которые создают три части письма – заголовок, тело и подпись, – и объединяет эти части в правильном порядке с помощью `string-append`.

Листинг 7. Пакетная программа

```
(define (letter fst lst signature-name)
  (string-append
    (opening fst)
    "\n\n"
    (body fst lst)
    "\n\n"
    (closing signature-name)))
(define (opening fst)
  (string-append "Dear " fst ","))
(define (body fst lst)
  (string-append
    "We have discovered that all people with the" "\n"
    "last name " lst " have won our lottery. So, " "\n"
    fst ", " "hurry and pick up your prize."))
(define (closing signature-name)
  (string-append
    "Sincerely,"
    "\n\n"
    signature-name
    "\n"))
```

Ведите эти определения в область определений DrRacket, щелкните на кнопке **RUN** (Выполнить) и вычислите следующие выражения в области взаимодействий:

```
> (letter "Matthew" "Fisler" "Felleisen")
"Dear Matthew,\n\nWe have discovered that ... \n"
> (letter "Kathi" "Felleisen" "Findler")
"Dear Kathi,\n\nWe have discovered that ... \n"
```

ПРИМЕЧАНИЕ. Результатом обоих выражений является длинная строка, содержащая "\n" – символ, представляющий перевод на но-

вую строку при **печати**. Теперь добавьте в программу инструкцию (`require 2htdp/batchio`), подключающую библиотеку с функцией `write-file`, которая позволит вам вывести эту строку в консоль:

```
> (write-file 'stdout (letter "Matt" "Fiss" "Fell"))
Dear Matt,
We have discovered that all people with the
last name Fiss have won our lottery. So,
Matt, hurry and pick up your prize.
Sincerely,
Fell
'standard
```

Пока считайте '`stdout`' простой строкой.

В разделе 2.5 мы подробно разберем такие пакетные программы.
КОНЕЦ.

В общем случае, когда решаемая программой проблема делится на несколько отдельных задач, каждая такая задача должна решаться отдельной функцией, а общая проблема – главной функцией, объединяющей все вместе. Выразим эту идею простым правилом:

Определяйте отдельную функцию для каждой задачи.

Следуя этому правилу, вы получаете достаточно короткие и простые для понимания функции. Когда вы научитесь разрабатывать функции, вы поймете, что маленькие функции намного проще заставить работать правильно, чем большие. Более того, если вам когда-нибудь понадобится изменить часть программы из-за изменения формулировки проблемы, то вам будет гораздо проще найти соответствующие части кода, если он организован как набор небольших функций, а не как большой монолитный блок.

Вот иллюстрация действия этого правила на примере.

Проблема. Владелец монопольного кинотеатра в маленьком городке имеет полную свободу устанавливать цены на билеты. Чем выше цена, тем меньше людей могут позволить себе купить билеты. Чем ниже цена, тем больше затраты на организацию показов из-за высокой посещаемости. Немного поэкспериментировав, владелец кинотеатра установил зависимость между ценой билета и средней посещаемостью.

При цене 5 долларов за билет на показ приходят 120 человек. На каждые 10 центов изменения стоимости билета средняя посещаемость меняется на 15 человек. То есть если поднять стоимость билета до 5,10 доллара, то на сеансы будет приходить в среднем 105 человек; если опустить цену до 4,90 доллара, то средняя посещаемость увеличится до 135 человек. Давайте переведем это в математическую формулу:

средняя посещаемость =

$$120 \text{ человек} - \frac{\text{величина изменения цены}}{0.10} \cdot 15 \text{ человек.}$$

Прежде чем продолжить, объясните, почему в этом уравнении стоит знак минус.

К сожалению, увеличение посещаемости удорожает обслуживание сеансов. Каждый сеанс обходится владельцу по фиксированной цене в 180 долларов плюс переменные затраты в размере 0,04 доллара на одного зрителя.

Владелец хотел бы знать точное соотношение между прибылью и ценой билета, чтобы максимизировать прибыль.

Итак, задача ясна, но как ее решить – пока не понятно. На данный момент мы можем лишь сказать, что у нас имеется несколько величин, зависящих друг от друга.

Сталкиваясь с такой ситуацией, желательно выявить различные зависимости, одну за другой:

1. В постановке задачи указано, что количество зрителей зависит от цены билета. Очевидно, вычисление этого числа является отдельной задачей и, следовательно, заслуживает отдельного определения функции:

```
(define (attendees ticket-price)
  (- 120 (* (- ticket-price 5.0) (/ 15 0.1))))
```

2. Выручка зависит исключительно от продажи билетов, то есть определяется произведением цены билета на количество зрителей:

```
(define (revenue ticket-price)
  (* ticket-price (attendees ticket-price)))
```

3. Затраты на проведение сеанса состоят из двух частей: фиксированной (180 долларов) и переменной, зависящей от количества зрителей. Учитывая, что количество зрителей является функцией от цены билета, то функция вычисления затрат на организацию сеанса должна также принимать цену билета, чтобы повторно использовать функцию attendees:

```
(define (cost ticket-price)
  (+ 180 (* 0.04 (attendees ticket-price))))
```

4. Наконец, *прибыль* – это разность между выручкой и затратами при заданной цене билета:

```
(define (profit ticket-price)
  (- (revenue ticket-price)
      (cost ticket-price)))
```

Определение функции `profit` явно отражает неформальное описание проблемы.

Эти четыре функции – все, что нужно для вычисления прибыли, и теперь мы можем использовать функцию `profit`, чтобы найти оптимальную цену билета.

Упражнение 27. Наше решение проблемы содержит несколько констант, используемых в функциях. Как отмечалось в разделе «Одна программа, множество определений» в прологе, таким константам лучше давать имена, чтобы будущие читатели кода понимали суть этих чисел. Определите все константы в области определений DrRacket и замените все магические числа именами констант. ■

Упражнение 28. Определите потенциальную прибыль для следующих цен на билеты: 1 доллар, 2 доллара, 3 доллара, 4 доллара и 5 долларов. При какой цене прибыль кинотеатра получится больше всего? Определите лучшую цену билета до цента. ■

Вот альтернативная версия той же программы в виде одной функции:

```
(define (profit price)
  (- (* (+ 120
            (* (/ 15 0.1)
                (- 5.0 price)))
        price)
      (+ 180
         (* 0.04
            (+ 120
               (* (/ 15 0.1)
                   (- 5.0 price)))))))
```

Ведите это определение в DrRacket и убедитесь, что оно дает те же результаты для цен для 1, 2, 3, 4 и 5 долларов, что и исходная версия. Одного взгляда на эту монолитную версию достаточно, чтобы убедиться, насколько сложнее понять эту функцию, чем четыре предшествующие.

Упражнение 29. Изучив затраты на сеанс, владелец обнаружил несколько способов снизить их. В результате оптимизации он избавился от постоянной составляющей затрат; осталась только переменная составляющая в размере 1,50 доллара на зрителя.

Измените обе программы, чтобы отразить эту находку, а затем протестируйте их еще раз с ценами на билеты 3, 4 и 5 долларов и сравните результаты. ■

2.4. Глобальные константы

Как уже говорилось в прологе, такие функции, как `profit`, выигрывают от использования глобальных констант. Любой язык программирования позволяет программистам определять константы. В BSL такие определения оформляются так:

- введите «(define»,
- добавьте имя константы,
- затем пробел и выражение
- и добавьте «)» в конце.

Константа – это, по сути, *глобальная переменная*, а ее определение называется *определением константы*. Выражение в определении константы мы обычно называем *правой частью определения*.

Определения констант задают имена для любых форм данных: чисел, изображений, строк и т. д. Вот несколько простых примеров:

```
; текущая стоимость билета на сеанс:  
(define CURRENT-PRICE 5)  
  
; удобно использовать для вычисления площади круга:  
(define ALMOST-PI 3.14)  
  
; пустая строка:  
(define NL "\n")  
  
; пустая сцена:  
(define MT (empty-scene 100 100))
```

Два первых определения объявляют числовые константы, два последних – константы со строкой и изображением. По соглашению в именах глобальных констант мы используем только прописные буквы. Это позволяет легко определять такие переменные при беглом просмотре программы.

На эти глобальные переменные могут ссылаться все функции в программе. Ссылка на переменную подобна непосредственному использованию соответствующего выражения. Преимущество использования именованных переменных вместо данных заключается в том, что изменение определения константы отражается на всех вычислениях, в которых она участвует. Например, мы можем добавить цифры в определение ALMOST-PI или увеличить размер пустой сцены:

```
(define ALMOST-PI 3.14159)  
  
; пустая сцена:  
(define MT (empty-scene 200 800))
```

В большинстве наших примеров определений в правой части используются *константы-литералы*, а в последнем – выражение. В действительности программист может использовать любые выражения в определениях констант, если они вычислимы в точке определения. Предположим, программе требуется обрабатывать изображения строго определенного размера и выполнять операции с его центром:

```
(define WIDTH 100)  
(define HEIGHT 200)  
  
(define MID-WIDTH (/ WIDTH 2))  
(define MID-HEIGHT (/ HEIGHT 2))
```

Мы можем определить две константы с литералами в правой части и две *вычисляемые константы*, то есть переменные, значения которых являются не просто литералами, но результатами вычисления выражений.

И снова обозначим это как правило:

Для каждой константы, используемой в формулировке задачи, определите константу в программе.

Упражнение 30. Определите константы в программе оптимизации цен на билеты в кинотеатре, преобразовав изменение посещаемости в зависимости от изменения цены (15 человек на каждые 10 центов) в константу. ■

2.5. Программы

Теперь вы готовы к созданию простых программ. С точки зрения написания кода, программа – это просто набор определений функций и констант. Обычно одна функция выделяется как «главная», и эта главная функция ссылается на другие функции. Однако с точки зрения выполнения программы делятся на две большие категории:

- *пакетные программы* получают сразу все свои входные данные и вычисляют результат. Главная функция такой программы является композицией из вспомогательных функций, которые могут ссылаться на другие вспомогательные функции и т. д. Когда мы запускаем пакетную программу, операционная система вызывает главную функцию, передает ей входные данные и ждет вывода программы;
- *интерактивные программы* получают лишь часть своих входных данных, выполняют вычисления и производят некоторые выходные данные, затем получают дополнительные данные и т. д. Появление дополнительных данных мы называем *событием*, а интерактивные программы – программами, которые *управляются событиями*. Главная функция такой программы, управляемой событиями, использует выражение, чтобы описывать, какие функции вызывать для тех или иных типов событий. Эти функции называются *обработчиками событий*.

Когда мы запускаем интерактивную программу, главная функция сообщает это описание операционной системе. Когда происходят события ввода, операционная система вызывает соответствующий обработчик событий. Точно так же операционная система знает из описания, когда и как представлять результаты выполнения этих обработчиков.

В этой книге основное внимание уделяется программам, взаимодействующим с пользователями через *графический пользовательский интерфейс* (ГПИ) (англ. graphical user interface, GUI); существуют так-

же другие виды интерактивных программ, и вы обязательно познакомитесь с ними, продолжив изучать информатику.

Пакетные программы. Как уже упоминалось, пакетная программа сразу получает все свои входные данные и, опираясь на них, вычисляет результат. Ее главная функция ожидает получить некоторые аргументы, передает их вспомогательным функциям, получает обратно их результаты и объединяет эти результаты в свой окончательный ответ.

После создания программы мы, естественно, приступаем к их использованию. В DrRacket пакетные программы запускаются в области взаимодействий, что позволяет наблюдать за их работой.

Еще более полезные программы могут извлекать входные данные из какого-то файла и выводить результаты в другой файл. В действительности название «пакетная программа» восходит к временам развития вычислительной техники, когда программа читала файл (или несколько файлов) из пакета перфокарт и помещала результат в другой файл(ы) и также в пакет перфокарт. По идее, пакетная программа целиком и сразу читает входной файл(ы) и выводит все результаты в другой файл(ы).

Подобные пакетные программы на основе файлов можно создавать с использованием библиотеки *2htdp/batch-io*, которая добавляет в наш словарь две функции (кроме прочих):

- `read-file`, читает содержимое файла и представляет его как строку;
- `write-file`, создает файл и записывает в него указанную строку.

Эти функции читают и записывают строки в файлы:

```
> (write-file "sample.dat" "212")
"sample.dat"
> (read-file "sample.dat")
"212"
```

После первой итерации файл с именем "sample.dat" будет содержать

| 212

Имена *stdout* и *stdin* являются сокращениями от «*standard output*» (стандартное устройство вывода, или просто стандартный вывод) и «*standard input*» (стандартное устройство ввода, или просто стандартный ввод) соответственно.

Результатом `write-file` является подтверждение, что строка помещена в файл. Если файл уже существует, его содержимое будет затерто заданной строкой; в противном случае функция создаст файл и запишет в него данную строку. Второе взаимодействие (`(read-file "sample.dat")`) прочитает содержимое файла "sample.dat" и вернет строку "212".

По pragматическим соображениям `write-file` может принимать в первом аргументе особый токен '`stdout`'. В этом случае вывод будет осуществляться на экран, в области взаимодействий, например:

```
> (write-file 'stdout "212\n")
212
'stdout
```

Аналогично функция `read-file` может принимать токен `'stdin` вместо имени файла и читать данные, вводимые с клавиатуры.

Давайте рассмотрим простой пример создания пакетной программы. Предположим, мы хотим создать программу, которая преобразует температуру, измеренную на градуснике со шкалой Фаренгейта, в температуру в градусах Цельсия. Не волнуйтесь, мы не собираемся проверять ваши знания по физике; вот формула преобразования:

Мы не требуем запоминать факты, но надеемся, что вы будете помнить, где их найти. Вы знаете, где найти формулу конвертирования температур?

$$C = \frac{5}{9} \cdot (f - 32).$$

Естественно, f в этой формуле – это температура по Фаренгейту, а C – температура по Цельсию. Этой формулы вполне достаточно для учебника алгебры, но математик или программист должен написать $C(f)$ в левой части уравнения, чтобы напомнить читателям, что f – это заданное значение, а C вычисляется из f .

Эта формула легко переводится на язык BSL:

```
(define (C f)
  (* 5/9 (- f 32)))
```

Напомним, что $5/9$ – это число, точнее рациональная дробь, и что C зависит от f , что и выражает обозначение функции.

Запускается эта пакетная программа в области взаимодействий как обычно:

```
> (C 32)
0
> (C 212)
100
> (C -40)
-40
```

Но давайте предположим, что мы решили использовать эту функцию в программе, которая читает температуру по Фаренгейту из файла, преобразует ее в температуру по Цельсию и выводит результат в другой файл.

У нас уже есть формула преобразования на языке BSL, осталось только написать главную функцию, объединяющую `C` с имеющимися элементарными функциями:

```
(define (convert in out)
  (write-file out
    (string-append
      (number->string
        (C
          (string->number
```

```
|     (read-file in)))
|     "\n")))
```

Мы дали главной функции имя `convert`. Она принимает имена двух файлов: `in` – файл со значением температуры по Фаренгейту и `out` – куда должен быть записан результат с температурой по Цельсию. Комбинируя пять функций, `convert` вычисляет результат. Давайте внимательно рассмотрим тело `convert`:

- 1) `(read-file in)` извлекает содержимое указанного файла и возвращает его в виде строки;
- 2) `string->number` преобразует строку в число;
- 3) С интерпретирует число как температуру по Фаренгейту и преобразует его в температуру по Цельсию;
- 4) `number->string` получает число с температурой по Цельсию и преобразует его в строку;
- 5) `(write-file out ...)` записывает строку с результатом в файл с именем, указанным в параметре `out`.

Этот длинный список шагов может показаться ошеломляющим, даже притом что в нем и не упоминается ссылка на функцию `string-append`. Приостановитесь и попробуйте сами объяснить, что делает

```
| (string-append ... "\n")
```

Когда мы изучали арифметику программирования, средняя композиция функций включала две, а иногда три функции. Но имейте в виду, что программы решают реальные задачи, тогда как упражнения по алгебре просто иллюстрируют идею композиции функций.

Файл "sample.dat" Теперь можно поэкспериментировать с `convert`. Для *намного также создать* чала используем `write-file`, чтобы создать файл с исходным *с помощью текстового* значением для `convert`: *редактора.*

```
> (write-file "sample.dat" "212")
"sample.dat"
> (convert "sample.dat" 'stdout)
100
'stdout
> (convert "sample.dat" "out.dat")
"out.dat"
> (read-file "out.dat")
"100"
```

В первом эксперименте мы использовали `stdout`, чтобы видеть результат работы `convert` в области взаимодействий. Во втором эксперименте мы указали имя выходного файла `"out.dat"`. Как и ожидалось, вызов `convert` вернул строку с именем этого файла, которую возвращает `write-file`; мы уже видели этот эффект, когда использовали `write-file` для записи температуры по Фаренгейту. Затем мы прочитали содержимое выходного файла с помощью `read-file`, но то же самое можно сделать с помощью текстового редактора.

Помимо простого запуска пакетной программы, полезно также выполнить вычисления в пошаговом режиме. Проверьте еще раз наличие файла "sample.dat" и что в нем находится единственное число, а затем щелкните на кнопке **STEP** (Шаг) в DrRacket. После этого откроется другое окно, в котором можно увидеть, как протекает процесс вычислений, запускаемый вызовом главной функции пакетной программы. Сделав это, вы увидите, что процесс в точности совпадает с описанием выше.

Упражнение 31. Вспомните программу `letter` из раздела 2.3. Вот как можно запустить эту программу и заставить ее выводить результат в область взаимодействий:

```
> (write-file
  'stdout
  (letter "Matthew" "Fisler" "Felleisen"))
Dear Matthew,
```

We have discovered that all people with the
last name Fisler have won our lottery. So,
Matthew, hurry and pick up your prize.

Sincerely,

Felleisen
'stdout

Конечно, одно из основных достоинств программ заключается в том, что они могут производить разные результаты для разных входных данных. Запустите `letter` с тремя входными строками по вашему выбору.

Вот пример пакетной программы, составляющей заготовки писем, которая читает имена из трех разных файлов и сохраняет письмо в четвертом файле:

```
(define (main in-fst in-lst in-signature out)
  (write-file out
    (letter (read-file in-fst)
           (read-file in-lst)
           (read-file in-signature))))
```

Функция `main` принимает четыре строки: первые три – имена входных файлов, а последняя – имя выходного файла. Она читает по одной строке из первых трех файлов, передает эти строки в `letter` и полученный результат записывает в файл с именем в `out` – четвертом параметре функции `main`.

Создайте соответствующие файлы, запустите главную функцию `main` и проверьте, выводит ли она ожидаемое письмо в заданный файл. ■

Интерактивные программы. Когда-то пакетные программы являлись основой использования компьютеров в производстве, но в настоящее время люди в основном работают с интерактивными программами. Обычно они взаимодействуют с настольными приложениями посредством мыши и клавиатуры. Кроме того, интерактивные

программы могут реагировать на события, генерируемые компьютером, такие как такты системных часов или поступление сообщения с другого компьютера.

Упражнение 32. Большинство людей запускают программы не только на настольных компьютерах, но также на сотовых телефонах, планшетах и бортовых компьютерах своих автомобилей. Очень скоро люди будут использовать носимые компьютеры, встроенные в очки, одежду и спортивное снаряжение. В более отдаленном будущем могут появиться встроенные биокомпьютеры, вживляемые в организм человека и напрямую взаимодействующие с функциями тела. Попробуйте представить десяток различных форм событий, с которыми придется иметь дело программам на таких компьютерах. ■

Цель этого раздела – познакомить с механикой написания **интерактивных** программ на BSL. Поскольку многие примеры в этой книге описывают интерактивные программы, мы будем представлять новые идеи постепенно и осторожно. Начав заниматься проектами интерактивных программ, вы сможете вернуться к этому разделу и прочитать его вновь; второе или третье чтение часто помогает прояснить особенно сложные аспекты механики.

Чистый компьютер без ничего – бесполезное физическое оборудование. К нему можно прикоснуться, и только. Компьютер становится полезным после установки *программного обеспечения*, то есть набора программ. Обычно в первую очередь на компьютер устанавливается *операционная система*. Ее задача – управлять компьютером, в том числе и подключенными устройствами, такими как монитор, клавиатура, мышь, динамики и т. д. Вот как примерно это работает: когда пользователь нажимает клавишу на клавиатуре, операционная система вызывает функцию, обрабатывающую нажатия клавиш. Мы говорим, что нажатие клавиши является *событием клавиатуры*, а функция – *обработчиком события*. Аналогично операционная система вызывает обработчик тактов системных часов, событий мыши и т. д. После того как обработчик событий выполнит свою работу, операционной системе может потребоваться изменить изображение на экране, издать звуковой сигнал, распечатать документ или выполнить какое-то другое действие. Для выполнения этих задач она вызывает функции, преобразующие данные в звуки, изображения, операции с принтером и т. д.

Естественно, у разных программ разные потребности. Одна программа может интерпретировать нажатия клавиш как сигналы управления ядерным реактором; другая передает их текстовому процессору. Чтобы компьютеры могли выполнять эти радикально разные задачи, разные программы устанавливают разные обработчики событий. То есть программа, управляющая запуском ракеты, использует одни функции для обработки тактов системных часов, а программное обеспечение управления духовой – другие.

При проектировании интерактивной программы необходимо иметь способ обозначить одну функцию как отвечающую за обработ-

ку событий клавиатуры, другую функцию – за обработку событий тактов системных часов, третью – за представление некоторых данных в виде изображения и т. д. Задача главной функции в интерактивной программе – сообщить эти обозначения операционной системе – программной платформе, на которой действует программа.

DrRacket – это небольшая операционная система, а BSL – один из ее языков программирования. Последний поставляется с библиотекой *2htdp/universe*, включающей механизм *big-bang*, посредством которого программа может сообщить операционной системе, какая функция и какое событие обрабатывает. Кроме того, *big-bang* следит за *состоянием программы*. Для этого в нем есть одно обязательное подвыражение, значение которого становится *начальным состоянием* программы. В остальном можно считать, что *big-bang* состоит из одного обязательного предложения и множества дополнительных предложений. Обязательное предложение *to-draw* сообщает DrRacket, как отобразить состояние программы, включая начальное. Все остальные необязательные предложения сообщают операционной системе, что та или иная функция обрабатывает определенное событие. Под обработкой событий в BSL подразумевается, что функция получает состояние программы и описание события и генерирует следующее состояние программы. Поэтому для описания состояния программы мы часто используем слова *текущее состояние* программы.

ТЕРМИНОЛОГИЯ. В некотором смысле выражение *big-bang* описывает, как программа взаимодействует с небольшим сегментом мира. Этот мир может быть игрой, в которую играет пользователь программы; анимацией, которую смотрит пользователь; или текстовым редактором, в котором пользователь ведет некоторые заметки. Поэтому исследователи языка программирования часто говорят, что *big-bang* – это описание маленького мира, включающее его начальное состояние, порядок преобразования состояний, порядок отображения состояний и как *big-bang* будет определять другие атрибуты текущего состояния. Учитывая это, мы также часто говорим о *состоянии мира* и даже называем программы *мировыми программами*.

КОНЕЦ.

Давайте исследуем эту идею шаг за шагом, начав со следующего определения:

```
| (define (number->square s)
|   (square s "solid" "red"))
```

Функция принимает положительное число и отображает красный квадрат этого размера. Щелкните на кнопке **RUN** (Выполнить) и поэкспериментируйте с этой функцией, например:

```
| > (number->square 5)
| ■
| > (number->square 10)
| ■
```

```
| > (number->square 20)
```



Он ведет себя как пакетная программа, принимая число и создавая изображение, которое DrRacket показывает на экране.

Теперь попробуйте выполнить следующее выражение `big-bang` в области взаимодействий:

```
| > (big-bang 100 [to-draw number->square])
```

Появится отдельное окно с красным квадратом 100×100 пикселей. Кроме того, в области взаимодействий DrRacket не отобразит приглашение к вводу, как если бы программа продолжала работать, и это действительно так. Чтобы остановить программу, нажмите кнопку **STOP** (Остановить) или просто закройте дополнительное окно:

```
| > (big-bang 100 [to-draw number->square])
```

100

Когда DrRacket останавливает вычисление выражения `big-bang`, он возвращает текущее состояние, которым в данном случае является начальное состояние: 100.

Вот более интересное выражение `big-bang`:

```
| > (big-bang 100
|   [to-draw number->square]
|   [on-tick sub1]
|   [stop-when zero?])
```

Это выражение `big-bang` добавляет два необязательных предложения к предыдущему: предложение `on-tick` сообщает DrRacket, как обрабатывать такты системных часов, а предложение `stop-when` сообщает, когда остановить программу. Это выражение читается так, с учетом начального состояния 100:

1. С каждым тактом часов вычесть 1 из текущего состояния.
2. Затем проверить истинность выражения `zero?` для нового состояния, и если оно истинно, то остановить программу.
3. Каждый раз, когда обработчик события возвращает значение, использовать `number->square` для отображения состояния в форме квадрата.

Теперь нажмите клавишу **return** и посмотрите, что произойдет. Спустя какое-то время выполнение выражений прекратится, и DrRacket выведет число 0.

Выражение `big-bang` отслеживает текущее состояние. Первоначально это состояние было равно 100. Каждый раз, когда отмеряется очередной тик часов, вызывается обработчик тактов, и текущее состояние обновляется. То есть состояние `big-bang` меняется следующим образом:

```
| 100, 99, 98, ..., 2, 1, 0
```

Когда значение состояния станет равным 0, вычисления прекращаются. Для любого другого состояния от 100 до 1 выражение `big-bang` преобразует состояние в изображение, используя `number->square`, как указано в предложении `to-draw`, и в результате в дополнительном окне отображается красный квадрат, который постепенно уменьшается в размерах в течение 100 тактов часов.

Давайте добавим предложение с обработчиком событий клавиатуры. Прежде всего определим функцию, которая принимает текущее состояние и строку, описывающую событие клавиатуры, и возвращает новое состояние:

```
(define (reset s ke)
  100)
```

Эта функция игнорирует свои аргументы и просто возвращает 100 – начальное состояние выражения `big-bang`, которое мы хотим изменить.

Затем добавим в выражение `big-bang` предложение `on-key`:

```
> (big-bang 100
  [to-draw number->square]
  [on-tick sub1]
  [stop-when zero?]
  [on-key reset])
```

Стоп! Попробуйте объяснить, что произойдет, если ввести это выражение в область взаимодействий, нажать клавишу `return`, сосчитать до 10 и нажать клавишу `a`.

Вы увидите, что красный квадрат сначала уменьшается в размерах на один пиксель с каждым тиком часов. Однако как только вы `a`, красный квадрат снова увеличится до полного размера, потому что нажатие клавиши `a` вызывает функцию `reset`, которая возвращает число 100. Это число станет новым состоянием `big-bang`, и `number->square` превратит его в полноразмерный красный квадрат.

Чтобы понять, как вычисляются выражения `big-bang`, рассмотрим следующую схематическую версию:

```
(big-bang cw0
  [on-tick tock]
  [on-key ke-h]
  [on-mouse me-h]
  [to-draw render]
  [stop-when end?]
  ...)
```

Это выражение `big-bang` определяет три обработчика событий – `tock`, `ke-h` и `me-h` – и использует предложение `stop-when`.

Вычисление данного выражения `big-bang` начинается с текущего состояния `cw0`, которое обычно является выражением. DrRacket, наша операционная система, устанавливает значение `cw0` как текущее состояние и вызывает `render` для преобразования текущего состояния в изображение, которое затем отображается в отдельном окне. На

самом деле `render` – это единственное средство, с помощью которого выражение `big-bang` представляет данные.

Вот как обрабатываются события:

- с каждым тактом часов DrRacket применяет `tock` к текущему состоянию `big-bang` и получает результат; `big-bang` рассматривает этот результат как следующее текущее состояние;
- при каждом нажатии клавиши DrRacket применяет `ke-h` к текущему состоянию выражения `big-bang` и к строке, представляющей клавишу; например, нажатие клавиши `a` будет представлено строкой "`a`", а нажатие клавиши со стрелкой влево – строкой "`left`". Значение, возвращаемое обработчиком `ke-h`, `big-bang` рассматривает как следующее текущее состояние;
- каждый раз, когда указатель мыши входит в окно, покидает его, перемещается по окну или выполняется щелчок мышью, DrRacket применяет `me-h` к текущему состоянию выражения `big-bang`, координатам `x` и `y` – события и строке, представляющей событие мыши; например, нажатие кнопки мыши будет представлено строкой "`button-down`". Значение, возвращаемое обработчиком `me-h`, `big-bang` рассматривает как следующее текущее состояние.

Все события обрабатываются по порядку; если кажется, что два события произошли одновременно, то DrRacket выступит в роли арбитра и выстроит их в определенном порядке.

После обработки события `big-bang` использует `end?` и `render` для проверки и отображения текущего состояния:

- (`end? cw`) возвращает логическое значение; если это `#true`, то `big-bang` немедленно останавливает вычисления, в противном случае продолжает их;
- предполагается, что (`render cw`) создаст изображение, и `big-bang` отображает это изображение в отдельном окне.

Таблица 2. Принцип действия выражения `big-bang`

Событие	cw_0	cw_1	...
Событие	e_0	e_1	...
Такт часов	(<code>tock cw₀</code>)	(<code>tock cw₁</code>)	...
Нажатие клавиши	(<code>ke-h cw₀ e₀</code>)	(<code>ke-h cw₁ e₁</code>)	...
Событие мыши	(<code>me-h cw₀ e₀ ...</code>)	(<code>me-h cw₁ e₁ ...</code>)	...
Отображение состояния	(<code>render cw₀</code>)	(<code>render cw₁</code>)	...

Таблица 2 кратко описывает этот процесс. В первой строке перечислены имена текущих состояний. Во второй строке – событий, возникающих в DrRacket: e_0 , e_1 и т. д. Каждое событие e_i может быть тактом часов, нажатием клавиши или событием мыши. В следующих трех строках описывается, как обрабатываются события:

- если e_0 – это такт часов, то big-bang вычисляет (`tock cw0`) и получает cw_1 ;
- если e_0 – это событие клавиатуры, то big-bang вычисляет (`ke-h cw0 e0`) и получает cw_1 . Обработчик должен применяться к самому событию, потому что часто программы по-разному реагируют на нажатия разных клавиш;
- если e_0 – это событие мыши, то big-bang вычисляет (`me-h cw0 e0 ...`) и получает cw_1 . Этот вызов показан в схематичной форме, потому что событие мыши e_0 сопровождается несколькими элементами данных – его природой и координатами, – и троеточием мы просто указываем на это;
- наконец, `render` превращает текущее состояние в изображение, как указывает последняя строка. DrRacket отображает такие изображения в отдельном окне.

Столбец под заголовком cw_1 показывает, как генерируется следующее текущее состояние cw_2 в зависимости от характера события e_1 .

Давайте применим эту таблицу для интерпретации конкретной последовательностью событий: пользователь нажимает клавишу `a`, затем истекает очередной такт часов, и, наконец, пользователь щелкает мышью, вызывая событие «button down» в позиции (90, 100). В обозначениях Racket:

- 1) cw_1 является результатом (`ke-h cw0 "a"`);
- 2) cw_2 – результатом (`tock cw1`);
- 3) cw_3 – результатом (`me-h cw2 90 100 "button down"`).

Мы можем выразить эти три шага в виде последовательности трех определений:

```
(define cw1 (ke-h cw0 "a"))
(define cw2 (tock cw1))
(define cw3 (me-h cw2 "button-down" 90 100))
```

Стоп! Как big-bang отображает каждое из этих трех состояний?

Теперь рассмотрим случай обработки последовательности из трех тактов часов. В этом случае:

- 1) cw_1 является результатом (`tock cw0`);
- 2) cw_2 является результатом (`tock cw1`);
- 3) cw_3 является результатом (`tock cw2`).

Или если сформулировать на языке BSL:

```
(define cw1 (tock cw0))
(define cw2 (tock cw1))
(define cw3 (tock cw2))
```

Фактически то же самое можно выразить в виде единственного выражения:

```
| (tock (tock (tock cw0)))
```

Оно определяет состояние, которое вычисляет `big-bang` после трех тактов часов. Стоп! Переформулируйте первую последовательность событий в виде единственного выражения.

Проще говоря, последовательность событий определяет, в каком порядке `big-bang` пересекает приведенную выше табл. 2 возможных состояний, чтобы достичь текущего состояния в каждый момент времени. Конечно, само выражение `big-bang` никак не изменяет текущее состояние; оно просто хранит его и при необходимости передает обработчикам событий и другим функциям.

Отсюда легко определить первую интерактивную программу. Взгляните на листинг 8. Программа состоит из двух определений констант, за которыми следуют три определения функций: `main`, запускающей интерактивную программу `big-bang`; `place-dot-at`, преобразующей текущее состояние в изображение; и `stop`, игнорирующей свои входные данные и возвращающей 0.

Листинг 8. Первая интерактивная программа

```
(define BACKGROUND (empty-scene 100 100))
(define DOT (circle 3 "solid" "red"))

(define (main y)
  (big-bang y
    [on-tick sub1]
    [stop-when zero?]
    [to-draw place-dot-at]
    [on-key stop]))

(define (place-dot-at y)
  (place-image DOT 50 y BACKGROUND))

(define (stop y ke)
  0)
```

После щелчка на кнопке **RUN** (Выполнить) мы можем попросить DrRacket применить эти функции-обработчики. Вот один из способов проверить их работу:

```
> (place-dot-at 89)

> (stop 89 "q")
0
```

Стоп! Попробуйте теперь объяснить, как `main` среагирует на нажатие клавиши.

Один из способов узнать, насколько верно ваше предположение, – запустить функцию `main` с некоторым разумным числом:

```
| > (main 90)
```

Передохните!

К настоящему времени вы могли почувствовать ошеломление от этих первых двух глав. Они познакомили вас со множеством новых понятий, включая новый язык, его словарь, его идеи, его идиомы, инструмент для записи новых слов в этот словарь и способ запуска программ. Столкнувшись с этим изобилием информации, у многих из вас может возникнуть вопрос: как создать программу, когда имеется сформулированная постановка задачи. Чтобы ответить на этот важный вопрос, в следующей главе мы сделаем шаг назад и подробно рассмотрим систематический подход к проектированию программ. Так что сделайте передышку и продолжайте, когда будете готовы.

3. Как проектировать программы

Как показали первые главы в этой книге, обучение программированию требует овладения многими понятиями. С одной стороны, нужен язык программирования, определяющий систему обозначений для описания вычислений. Языки, используемые для формулирования программ, – это искусственные конструкции, однако процесс освоения языка программирования имеет много общего с освоением естественного языка. Оба имеют словарный запас, грамматику и представление о «фразах».

С другой стороны, очень важно научиться переходить от постановки задачи к программе. Мы должны определить, что уместно в постановке задачи, а что можно игнорировать. Мы обязаны выяснить, какие данные должна получать программа, какие данные производить и как входные данные соотносятся с выходными. Мы должны знать или выяснить, поддерживает ли выбранный язык и его библиотеки базовые операции с данными, которые наша программа должна обрабатывать. В противном случае нам, возможно, придется разработать вспомогательные функции, реализующие эти операции. Наконец, создав программу, мы должны проверить, действительно ли она выполняет предполагаемые вычисления. При этом могут быть выявлены всевозможные ошибки, которые необходимо исследовать и исправить.

Все это звучит довольно сложно, и у кого-то из вас может возникнуть вопрос: почему бы просто не начать экспериментировать тут и там, оставляя в покое те участки, результаты работы которых выглядят верными? Такой подход к программированию часто называют «гаражным программированием», он широко распространен и во многих случаях действительно приводит к успеху; иногда с этого начинается развитие новой компании. Однако никакая компания не может продавать такие «гаражные программы», потому что разобраться в них смогут только ее авторы.

Хорошая программа поставляется с коротким описанием, объясняющим, что делает программа, какие данные принимает и производит. В идеале такое описание дает некоторую уверенность, позволяя убедиться, что программа действительно работает. Связь идеальной программы с постановкой задачи очевидна, поэтому небольшое изменение в постановке задачи легко трансформировать в небольшое изменение программы. Инженеры-программисты называют такие программы «программным продуктом».

Слово «другие» также подразумевает самих программистов, которые обычно забывают все мысли, которые они вкладывали в создаваемые программы.

Вся эта дополнительная работа необходима, потому что программисты создают программы не для себя. Программисты пишут программы для чтения другими программистами, и, так уж сложилось, иногда люди запускают эти программы, чтобы выполнить работу. Большинство программ представляют собой большие и сложные наборы

взаимодействующих функций, и никто не может написать все эти функции за один день. Программисты подключаются к проектам, пишут код, покидают проекты; иные берут их программы и работают над ними. Другая трудность заключается в склонности клиентов менять свое мнение о задаче, которую они действительно хотят решить. Обычно они правильно формулируют задачу в целом, но часто ошибаются в некоторых деталях. Хуже того, такие сложные логические конструкции, как программы, почти всегда страдают от человеческих ошибок; проще говоря, программисты тоже могут ошибаться. В конце концов, кто-то обнаруживает эти ошибки, и программисты должны их исправить. Им нужно прочитать программный код, написанный месяцы, год или двадцать лет тому назад, и изменить его.

Упражнение 33. Почитайте о проблеме «2000 года». ■

Здесь мы представляем рецепт проектирования, который объединяет пошаговый процесс со способом организации программ вокруг обрабатываемых данных. Для читателей, которые не любят долго смотреть на пустой экран, этот рецепт предлагает способ систематического движения вперед. Для тех из вас, кто учит других проектировать программы, рецепт послужит инструментом выявления трудностей, которые испытывают новички. Кто-то сможет применить наш рецепт в других областях, например в медицине, журналистике или инженерии. Тем, кто хочет стать настоящим программистом, рецепт проектирования поможет научиться разбираться в существующих программах и работать с ними, даже если они написаны программистами, не использующими подобные рецепты проектирования. Остальная часть этой главы посвящена первым шагам в мире рецептов проектирования; а последующие главы и части так или иначе будут уточнять и расширять рецепт.

3.1. Проектирование функций

Информация и данные. Цель программы – описать вычислительный процесс, потребляющий некоторую информацию и производящий новую информацию. В этом смысле программа похожа на инструкции, которые учитель математики дает ученикам начальной школы. Однако, в отличие от ученика, программа работает не только с числами: она обрабатывает навигационную информацию, отыскивает адрес человека, переключает флаги или проверяет состояние видеоигры. Вся эта информация поступает из области реального мира, часто называемой *предметной областью* программы, и результаты вычислений, производимых программой, представляют дополнительную информацию об этой области.

Информация играет центральную роль в нашем описании. Информация – это факты о предметной области программы. Для программы, обслуживающей каталог мебели, такие понятия, как «стол на пяти ножках» или «квадратный стол размером два на два метра»,

являются фрагментами информации. Игровая программа имеет дело с предметной областью другого типа, где «пять» может описывать количество пикселей, на которое перемещается объект за такт часов. А для программы расчета заработной платы «пять» может означать процент удержаний.

Чтобы программа могла обрабатывать информацию, она должна преобразовать ее в некоторую форму представления данных на языке программирования, затем обработать эти данные и по завершении преобразовать полученные данные в информацию. Интерактивная программа может даже смешивать эти шаги, собирая больше информации из внешнего мира по мере необходимости, и передавать ее между шагами.

Мы используем BSL и DrRacket, поэтому вам не нужно беспокоиться о преобразовании информации в данные. В языке BSL функцию можно применить непосредственно к данным и посмотреть, что она производит. Это позволяет избежать серьезной «проблемы курицы и яйца» при создании функций, преобразующих информацию в данные и наоборот. Для простых видов информации такие элементы программы создаются легко и просто, но для более сложной информации может потребоваться знание, например, приемов синтаксического анализа, а это, в свою очередь, требует большого опыта в разработке программ.

Инженеры-программисты широко используют методологию *модель-представление-контроллер* (Model-View-Controller, MVC), чтобы отделить обработку данных от преобразования информации в данные и данных в информацию. В настоящее время общепризнано, что хорошо спроектированные программные системы обеспечивают это разделение, хотя большинство вводных книг по программированию все еще объединяют эти этапы. Таким образом, работа с BSL и DrRacket позволяет сосредоточиться на разработке ядра программы, а когда вы накопите достаточно опыта в этом, вы сможете научиться проектировать элементы программ, отвечающие за преобразование информации в данные и обратно.

В этой книге мы используем два предустановленных учебных пакета, чтобы продемонстрировать разделение данных и информации: `2htdp/batch-io` и `2htdp/universe`. В этой главе мы приступим к формированию рецептов проектирования интерактивных и неинтерактивных программ, чтобы вы могли получить представление о том, как создаются полноценные программы. Имейте в виду, что библиотеки многих развитых языков программирования предлагают гораздо больше контекстов для создания программ, и вам придется, так или иначе, адаптировать рецепты проектирования под конкретные особенности языка.

Учитывая центральную роль информации и данных, проектирование программы должно начинаться с определения связей между фрагментами информации. В частности, мы, программисты, должны решить, как на выбранном языке программирования *представлять*

соответствующие фрагменты информации в виде данных и как *интерпретировать* данные при их преобразовании в информацию. Эту идею поясняет абстрактная диаграмма на рис. 4.



Рис. 4. Преобразование информации в данные и данных в информацию

Чтобы конкретизировать эту идею, рассмотрим несколько примеров. Предположим, вы проектируете программу, которая принимает и производит числовую информацию. Выбор способа представления в этом случае прост, но при интерпретации необходимо пояснить, что означает число, например 42, в данной предметной области:

- в программе, обрабатывающей изображения, 42 может означать расстояние в пикселях от верхнего края;
- в игровой программе 42 может означать расстояние, на которое переместился игровой объект за один такт часов;
- в программе, выполняющей физические расчеты, 42 может означать температуру по шкале Фаренгейта, Цельсия или Кельвина;
- в программе, которая ведет каталог мебели, 42 может означать размер стола;
- в любой программе, независимо от предметной области, 42 может просто означать количество символов в строке.

Поэтому очень важно знать, как перейти от чисел-информации к числам-данным и наоборот.

Поскольку эти знания так важны для всех, кто читает программу, мы часто записываем их в виде комментариев, которые называем *определениями данных*. Определение данных служит двум целям. Во-первых, дает осмысленное имя набору данных – *классу*. А во-вторых, информирует читателей, как создать экземпляры этого класса и как определить, принадлежит ли некоторый произвольный фрагмент данных коллекции.

Вот определение данных для одного из примеров, приведенных выше:

Для обозначения чего-то, подобного «математическому множеству», инженеры-программисты используют термин «класс».

| ; Температура -- это число.
| ; интерпретация: число выражает температуру в градусах Цельсия

Первая строка вводит имя для коллекции данных – *Температура* – и сообщает, что класс состоит из чисел. Например, если вас спросят, является ли число 102 температурой, то вы сможете ответить «да», потому что 102 – это число, а все числа представляют температуру. Точно так же если вас спросят, является ли строка «холодный» температурой, то вы сможете уверенно ответить «нет», потому что никакая строка в этой программе не является температурой. А если вас попросят представить образец температуры, то вы сможете показать число, например -400.

Если вам известно, что минимально возможная температура имеет величину примерно -274°C , то возникает естественный вопрос: можно ли выразить это знание в определении данных. Поскольку наши определения данных на самом деле являются простыми описаниями классов на простом человеческом языке, мы действительно можем дать гораздо более точное определение класса температур, чем показано выше. Для таких определений данных мы используем в этой книге стилизованную форму естественного языка, а в следующей главе представим способ наложения ограничений, таких как «больше -274».

До настоящего момента нам встречались имена четырех классов данных: Число (Number), Стока (String), Изображение (Image) и Логическое значение (Boolean). При этом формулирование нового определения данных означает не что иное, как введение нового имени для существующей формы данных, скажем имени «температура» для чисел. Однако даже этих ограниченных знаний достаточно, чтобы объяснить схему процесса проектирования.

Теперь вы можете вернуться к разделу «Системное проектирование программ» во вступлении и особенно к рецепту 1.

Процесс проектирования. Как только вы поймете, как представлять входную информацию в виде данных и как интерпретировать выходные данные в виде информации, проектирование каждой отдельной функции превращается в простой процесс:

1. Выразить представление информации в виде данных. Для этого достаточно одностороннего комментария:

| ; Мы обозначаем числами длину в сантиметрах.

Сформулировать определения классов данных, которые считаются критически важными для успеха программы, как мы сделали это с классом Температура.

2. Записать сигнатуру, описание назначения и заголовок функции.

Сигнатурой функции – это комментарий, сообщающий читателям вашего проекта, сколько входных параметров принимает ваша функция, каким классам они принадлежат и какие данные эта функция производит. Вот три примера:

- на входе – строка, на выходе – число:

| ; Стока -> Число

- на входе – температура, на выходе – строка:

| ; Температура -> Стока

Как указывает эта сигнатура, введение псевдонима для существующей формы данных позволяет легко понять ваши намерения.

Тем не менее мы советуем пока воздержаться от определения таких псевдонимов, потому что это может вызвать путаницу. Чтобы уметь находить правильный баланс между потребностью в новых именах и удобочитаемостью программ, нужна практика, а нам еще нужно понять более важные идеи;

- на входе число, строка и изображение:

| ; Число Стока Изображение -> Изображение

Стоп! Ответьте на вопрос: что получается на выходе этой функции?

Описанием назначения в языке BSL называется односторонний комментарий, описывающий цель функции. Если вы затрудняетесь сформулировать цель, то запишите как можно более краткий ответ на следующий вопрос:

Что вычисляет эта функция?

Каждый читатель вашей программы должен понимать, что вычисляют ваши функции, **без необходимости** читать саму функцию.

Для программы, состоящей из множества функций, тоже должно быть сформулировано описание назначения. Вообще говоря, хорошие программисты пишут два описания назначения: одно – для читателя, которому, возможно, понадобится изменить код, и другое – для человека, который захочет использовать программу, не читая ее код.

Наконец, заголовок – это упрощенное определение функции. Выберите одно имя переменной для каждого класса входных данных в сигнатуре, а роль тела функции может играть любой фрагмент данных выходного класса. Вот три заголовка функций, которые соответствуют трем сигнатурам, упомянутым выше:

- (define (f a-string) 0);
- (define (g n) "a");
- (define (h num str img) (empty-scene 100 100)).

Имена параметров здесь отражают тип данных, представляемых этими параметрами. Иногда предпочтительнее использовать имена, подсказывающие назначение параметра.

Формулируя описание назначения, часто бывает полезно использовать имена параметров, поясняющие вычисления. Например:

```
| ; Число Стока Изображение -> Изображение
| ; добавляет строку s в изображение img,
| ; с отступом w в пикселях от верхнего края и 10 -- от левого края
| (define (add-image y s img)
|   (empty-scene 100 100))
```

Теперь можете щелкнуть на кнопке **RUN** (Выполнить) и поэкспериментировать с функцией. Конечно, результат всегда будет одним и тем же, что делает эксперименты довольно скучными.

- Проиллюстрируйте сигнатуру и описание назначения несколькими примерами применения функции. Для этого выберите по одному экземпляру данных каждого входного класса из сигнатуры и покажите, каким должен быть результат.

Предположим, что вы разрабатываете функцию, которая вычисляет площадь квадрата. Очевидно, что эта функция принимает длину стороны квадрата, которую проще всего представить числом (положительным). Допустим, вы сделали первый шаг в соответствии с рецептом и теперь добавляете примеры между описанием назначения и заголовком:

```
| ; Число -> Число
| ; вычисляет площадь квадрата со стороной len
| ; дано: 2, ожидаемый результат: 4
| ; дано: 7, ожидаемый результат: 49
| (define (area-of-square len) 0)
```

- Следующий шаг – составить *инвентарь*, который поможет понять, что дано и что нужно вычислить. В случае с простыми функциями, которые мы рассматриваем в данный момент, мы знаем, что они получают данные через параметры.

Параметры представляют пока неизвестные нам значения, но мы знаем, что, опираясь на эти неизвестные данные, функция должна вычислить свой результат. Чтобы напомнить себе об этом факте, заменим тело функции *макетом*.

На данный момент макет содержит только параметры, поэтому предыдущий пример выглядит так:

```
(define (area-of-square len)
  (... len ...))
```

Многоточия напоминают, что это не полная функция, а макет, который нужно заполнить.

Макеты в этом разделе выглядят скучно. Но как только мы введем новые формы данных, они станут намного интереснее.

Термин
«инвентарь»
предложил
Стивен Блох
(Stephen Bloch).

5. Настал момент написать код. В общем случае написание кода означает программирование, но в более узком смысле этого слова, а именно – написание выполняемых выражений и определений функций.

Для нас написание кода означает замену тела функции выражением, которое использует элементы макета и вычисляет то, о чём говорится в описании назначения. Вот полное определение `area-of-square`:

```
| ; Число -> Число
| ; вычисляет площадь квадрата со стороной len
| ; дано: 2, ожидаемый результат: 4
| ; дано: 7, ожидаемый результат: 49
(define (area-of-square len)
  (sqr len))
```

Чтобы завершить функцию `add-image`, требуется немного больше усилий: см. листинг 9. В частности, функция должна преобразовать заданную строку `s` в изображение и поместить его в заданную сцену.

6. Последний шаг в правильном процессе проектирования – проверка функции на заранее придуманных вами примерах. Пока что тестирование выполняется так: щелкните на кнопке **RUN** (Выполнить) и введите примеры выражений применения функции в области взаимодействий:

```
| > (area-of-square 2)
| 4
| > (area-of-square 7)
| 49
```

Листинг 9. Окончательный результат после пятого шага

```
; Число Страна Изображение -> Изображение
; добавляет строку s в изображение img,
; с отступом y пикселей от верхнего края и 10 -- от левого края
; дано:
;     y = 5,
;     s = "hello", и
;     img = (empty-scene 100 100)
; ожидаемый результат:
;     (place-image (text "hello" 10 "red") 10 5 ...)
;     где ... -- это (empty-scene 100 100)
(define (add-image y s img)
  (place-image (text s 10 "red") 10 y img))
```

Фактические результаты должны совпадать с ожидаемыми; обязательно проверьте все результаты и убедитесь, что они в точности соответствуют указанным в примерах. Если какой-то из фактических результатов не соответствует ожидаемому, проверьте три возможных случая:

- вы ошиблись в расчетах и неверно определили ожидаемый результат для некоторых примеров;

- b) определение функции содержит ошибку и дает неверный результат. В этом случае в вашей программе имеется **логическая ошибка**, известная также как **жучок (bug)**;
- c) ошибки имеют место и в примерах, и в определении функции.

Обнаружив несоответствие между ожидаемыми и фактическими результатами, мы рекомендуем сначала проверить правильность ожидаемых результатов. Если они верные, то ошибка, скорее всего, скрыта в определении функции. Иначе исправьте пример и снова выполните тесты. Если проблема осталась, то, возможно, вы столкнулись с третьей, более редкой ситуацией.

3.2. Практические упражнения: функции

Несколько первых упражнений из представленных ниже почти в точности повторяют упражнения из раздела 2.1, отличаясь только тем, что вместо слова «определите» здесь используется слово «спроектируйте». Это означает, что вы должны создать эти функции, следуя рецепту проектирования, и ваши решения должны включать все элементы, предполагаемые рецептом.

Как следует из названия раздела, эти упражнения имеют целью помочь вам закрепить процесс проектирования на практике. Пока шаги не вошли у вас в привычку, никогда не пропускайте их, потому что иначе увеличивается риск допустить ошибки, которых легко избежать. В программировании еще много места для сложных ошибок, и не стоит тратить время на глупости.

Упражнение 34. Спроектируйте функцию `string-first`, которая извлекает первый символ (`1String`) из непустой строки. Считайте пока, что функция никогда не будет получать пустые строки. ■

Упражнение 35. Спроектируйте функцию `string-last`, которая извлекает последний символ (`1String`) из непустой строки. ■

Упражнение 36. Спроектируйте функцию `image-area`, которая подсчитывает количество пикселей в заданном изображении. ■

Упражнение 37. Спроектируйте функцию `string-rest`, которая создает строку, подобную данной, но без первого символа. ■

Упражнение 38. Спроектируйте функцию `string-remove-last`, которая создает строку, подобную данной, но без последнего символа. ■

3.3. Знание предметной области

У многих начинающих программистов возникает естественный вопрос: какие знания необходимы, чтобы определить тело функции. Нетрудно догадаться, что для этого необходимо знать и понимать

предметную область программы. Существует две формы такого знания *предметной области*:

- 1) знания из внешних предметных областей, таких как математика, музыка, биология, гражданское строительство, искусство и т. д. Поскольку программисты не могут знать все предметные области, где применяются вычисления, они должны быть готовы понимать язык различных предметных областей, чтобы обсуждать проблемы с экспертами в этих областях. Математика находится на пересечении многих, но не всех предметных областей. Поэтому программистам приходится осваивать новые термины, решая проблемы с экспертами в предметной области;
- 2) знание библиотечных функций, доступных в выбранном языке программирования. Если ваша задача – перевести на язык программирования математическую формулу, включающую функцию тангенса, то выбранный вами язык должен иметь такую функцию, как, например, `tan` в BSL. Если ваша задача связана с графикой, то вам пригодится знание возможностей библиотеки `2htdp/image`.

Поскольку нельзя заранее предсказать, в какой предметной области вам придется работать или какой язык программирования вы будете использовать, крайне важно, чтобы у вас было четкое представление обо всех возможностях любых языков программирования, которые существуют и подходят для решения ваших задач. В противном случае вашу работу возьмет на себя какой-нибудь эксперт в предметной области с недостаточными знаниями в области программирования.

Распознать задачи, требующие знания предметной области, можно по определениям данных. В определениях данных используются классы, поддерживаемые выбранным языком программирования, поэтому определение тела функции (и программы) в основном зависит от опыта в данной области. Позже, когда мы познакомимся с составными формами данных, проектирование функций потребует от нас знаний в области информатики.

3.4. От функций к программам

Не все программы состоят из определения единственной функции. Большинство программ состоят из нескольких функций; многие также используют определения констант. Но в любом случае важно систематически проектировать каждую функцию, хотя глобальные константы и вспомогательные функции немного меняют процесс проектирования.

После определения глобальных констант ваши функции могут использовать их для вычисления результатов. Чтобы напомнить себе об их существовании, эти константы можно добавлять в макеты; в кон-

це концов, они относятся к элементам, которые могут способствовать определению функции.

Потребность в множестве функций возникает, например, потому, что интерактивным программам необходимы функции, обрабатывающие события от клавиатуры и мыши, преобразующие текущее состояние в музыку, и, возможно, многие другие. Даже неинтерактивным программам может потребоваться несколько различных функций для решения нескольких отдельных задач. Иногда постановка проблемы сама предлагает эти задачи, а иногда, в процессе разработки какой-либо функции, может возникнуть потребность во вспомогательных функциях.

Термином «список желаний» мы обязаны Джону Стоуну (John Stone).

По этим причинам мы рекомендуем создать и сохранить список необходимых функций – *список желаний*. Каждый элемент в списке желаний должен включать: описательное имя функции, ее сигнатуру и описание назначения. Создание неинтерактивной программы начинайте с добавления главной функции в список желаний и приступайте к ее проектированию. Создание интерактивной программы можно начать с добавления в список обработчиков событий, функции `stop-when` и функции `scene-rendering`. Пока список не опустеет, выбирайте из него очередной элемент и проектируйте функцию. Если во время проектирования обнаружится, что вам нужна еще одна функция, добавьте ее в список. Когда список опустеет, программа будет готова.

3.5. О тестировании

Тестирование быстро превращается в трудоемкую работу. Конечно, запускать небольшие программы в области взаимодействий несложно, но для этого требуется много механического труда и сложных проверок. По мере развития своих систем программистам приходится проводить множество проверок. Вскоре эта работа становится непосильной, и программисты начинают пренебрегать ею. В то же время тестирование – это самый главный инструмент для обнаружения и устранения недостатков. Небрежное тестирование влечет просачивание ошибок в функции, то есть появление функций со скрытыми дефектами, а функции с дефектами тормозят развитие проекта, часто самым неожиданным образом.

Поэтому очень важно автоматизировать тестирование, чтобы не проводить его вручную. Подобно многим языкам программирования, BSL имеет свои средства тестирования, и DrRacket знает об этом. Для знакомства с этими средствами тестирования вернемся к функции, преобразующей температуру по Фаренгейту в температуру по Цельсию, представленную в разделе 2.5. Вот ее определение:

```
| ; Число -> Число
| ; преобразует температуру по Фаренгейту в температуру по Цельсию
```

```
| ; дано: 32, ожидаемый результат: 0
| ; дано: 212, ожидаемый результат: 100
| ; дано: -40, ожидаемый результат: -40
(define (f2c f)
  (* 5/9 (- f 32)))
```

Чтобы протестировать функцию, необходимо трижды применить ее и сравнить полученные результаты с ожидаемыми. Вот как сформулировать эти тесты в области определений DrRacket:

```
| (check-expect (f2c -40) -40)
| (check-expect (f2c 32) 0)
| (check-expect (f2c 212) 100)
```

Если теперь щелкнуть на кнопке **RUN** (Выполнить), то вы увидите отчет BSL о том, что программа успешно прошла все три теста и от вас ничего больше не требуется.

Кроме возможности автоматически запускать тесты в форме `check-expect`, поддержка тестирования предлагает еще одно преимущество: вывод сообщений об ошибках, если тесты терпят неудачу. Чтобы увидеть, как это работает, измените один из тестов так, чтобы результат сравнения ожидаемого результата с фактическим оказался ложным, например:

```
| (check-expect (f2c -40) 40)
```

Теперь после щелчка на кнопке **RUN** (Выполнить) появится дополнительное окно с текстом, сообщающим, что один из трех тестов потерпел неудачу. Для неудачного теста в окне будут показаны: вычисленное значение, результат вызова функции (`-40`); ожидавшееся значение (`40`) и гиперссылка на определение теста, потерпевшего неудачу.

Листинг 10. Тестирование в BSL

```
| ; Число -> Число
| ; преобразует температуру по Фаренгейту в температуру по Цельсию

(check-expect (f2c -40) -40)
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)

(define (f2c f)
  (* 5/9 (- f 32)))
```

Определения `check-expect` можно разместить выше или ниже определений тестируемых функций. После щелчка на кнопке **RUN** (Выполнить) DrRacket выберет все определения `check-expect` и выполнит их после добавления определений всех функций в «словарь» операций. В листинге 10 показано, как использовать эту свободу и объединить примеры с шагом тестирования. Вместо записи примеров в виде комментариев их можно преобразовать непосредственно в тесты. Закончив проектировать функцию, щелкните на кнопке **RUN** (Выполнить) –

и тесты выполняются автоматически. И если позже вам понадобится изменить функцию по какой-либо причине, следующий щелчок на кнопке **RUN** (Выполнить) поможет вам проверить функцию повторно.

И последнее, но не менее важное: `check-expect` также может выполнять проверки с изображениями, то есть позволяет протестировать функции, создающие изображения. Предположим, вы решили спроектировать функцию `render`, которая помещает изображение автомобиля, с именем `CAR`, в сцену с именем `BACKGROUND`. Для этой функции можно сформулировать следующие тесты:

```
(check-expect (render 50)
  

```

Дополнительные способы формулирования тестов вы найдете в интермецо 1.

Альтернативный вариант:

```
(check-expect (render 50)
  (place-image CAR 50 Y-CAR BACKGROUND))
(check-expect (render 200)
  (place-image CAR 200 Y-CAR BACKGROUND))
```

Этот альтернативный подход помогает понять, как выразить тело функции, и поэтому он предпочтительнее. Один из способов разработки таких выражений – проведение экспериментов в области взаимодействий.

Поскольку DrRacket дает такую удобную возможность автоматизации тестов, мы немедленно переключимся на этот стиль тестирования и будем использовать его на протяжении всей оставшейся части книги. Эта форма тестирования называется *модульным тестированием*, и фреймворк модульного тестирования в BSL специально предназначен для начинающих программистов. Однажды вы перейдете на какой-нибудь другой язык программирования, и одной из ваших первых задач станет выбор подходящего фреймворка модульного тестирования.

3.6. Проектирование интерактивных программ

В предыдущей главе мы специально познакомили вас с библиотекой `2htdp/universe`, чтобы в этом разделе показать, как рецепт проектирования, кроме всего прочего, помогает систематически проектировать программы. Он начинается с краткого обзора библиотеки `2htdp/universe`, где будет представлен список определений данных и сигнатур функций, а затем представит рецепт проектирования программ.

Учебный пакет предполагает, что программист разработает определение данных, представляющее состояние программы, и функцию `render`, которая знает, как создать изображение для каждого возмож-

ного состояния. В зависимости от потребностей программы программист должен также спроектировать функции, обрабатывающие такты часов, нажатия клавиш и события мыши. Наконец, интерактивной программе может потребоваться остановиться по достижении конечного состояния; функция `end?` должна распознавать такие конечные состояния. Эта идея схематично представлена в листинге 11.

Листинг 11. Список желаний для проектирования программ

```
; СостояниеМира: данные, представляющие состояние мира в некоторый момент (cw)

; СостояниеМира -> Изображение
; когда это необходимо, big-bang получит изображение, соответствующее
; текущему состоянию мира, вычислив выражение (render cw)
(define (render cw) ...)

; СостояниеМира -> СостояниеМира
; с каждым тактом часов big-bang будет получать следующее
; состояние мира из (clock-tick-handler cw)
(define (clock-tick-handler cw) ...)

; СостояниеМира Стока -> СостояниеМира
; по нажатии каждой клавши big-bang будет получать следующее
; состояние из (keystroke-handler cw ke), где ke представляет клавишу
(define (keystroke-handler cw ke) ...)

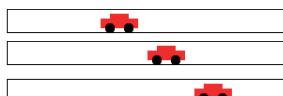
; СостояниеМира Число Число Стока -> СостояниеМира
; для каждого действия мышью big-bang будет получать следующее
; состояние из (mouse-event-handler cw x y me), где x и y --
; координаты события, а me -- его описание
(define (mouse-event-handler cw x y me) ...)

; СостояниеМира -> ЛогическоеЗначение
; после каждого события big-bang вычислит (end? cw)
(define (end? cw) ...)
```

Предположим, что вы уже имеете упрощенное представление о работе `big-bang` и можете сосредоточиться на действительно важной задаче проектирования программ. Давайте создадим конкретный пример, следуя рецепту дизайна.

Задача. Спроектируйте программу, которая перемещает автомобиль в пределах холста слева направо, по три пикселя за такт часов.

С такой постановкой задачи легко представить, как будут выглядеть сцены, генерируемые программой:



В этой книге мы часто будем называть предметную область интерактивной программы `big-bang` «миром» и говорить о проектировании «мировых программ».

Рецепт проектирования мировых программ, так же как рецепт проектирования функций, – это инструмент, помогающий постепенно перейти от постановки задачи к действующей программе. Он включает три больших шага и один маленький.

1. Для всех свойств мира, которые остаются неизменными с течением времени и необходимы для визуализации в виде изображения, создайте константы. В BSL такие константы создаются через определения. Применительно к мировым программам мы различаем два вида констант:

a) «физические» константы описывают общие атрибуты объектов в мире, такие как скорость движения объекта, его цвет, высоту, ширину, радиус и т. д. Конечно, на самом деле эти константы не относятся к физическим фактам, но многие из них аналогичны физическим аспектам реального мира.

В контексте нашей задачи такими «физическими» константами являются радиус колес автомобиля и расстояние между колесами:

```
(define WIDTH-OF-WORLD 200)
(define WHEEL-RADIUS 5)
(define WHEEL-DISTANCE (* WHEEL-RADIUS 5))
```

Обратите внимание, что вторая константа вычисляется на основе первой;

- b) графические константы – изображения предметов в мире. Программа будет объединять их в изображения, представляющие полное состояние мира.

Вот графические константы с изображениями колес нашего автомобиля:

```
(define WHEEL
  (circle WHEEL-RADIUS "solid" "black"))
(define SPACE
  (rectangle ... WHEEL-RADIUS ... "white"))
(define BOTH-WHEELS
  (beside WHEEL SPACE WHEEL))
```

Обычно графические константы вычисляются, и в вычислениях часто используются физические константы и другие графические константы.

Хорошей практикой считается снабжать определения констант комментариями, объясняющими их назначение.

2. Свойства, меняющиеся со временем – с каждым тиком часов, после нажатия клавиш или действий мышью, – определяют текущее состояние мира. Ваша задача – разработать представление данных для всех возможных состояний мира. Результатом должно быть определение данных, сопровождаемое коммен-

*Мы настоятельно
советуем позэкспери-
ментировать в обла-
сти взаимодействий
DrRacket и попробовать
самостоятельно
создать такие графи-
ческие константы.*

тарием, из которого читатели смогут узнать, как информация о мире преобразуется в данные и как данные интерпретируются в информацию о мире.

Для представления состояния мира выбирайте простые формы данных.

В нашем примере с течением времени изменяется расстояние от автомобиля до левого края холста. Расстояние до правого края тоже меняется, но для создания изображения достаточно расстояния до одного края, левого или правого. Расстояние измеряется числами, поэтому вполне подойдет следующее определение данных:

```
| ; СостояниеМира -- это число.  
| ; интерпретация: число пикселей от левого края  
| ; сцены до автомобиля
```

Как вариант в роли состояния мира можно использовать счетчик прошедших тактов часов. Реализацию этого варианта мы оставляем вам в качестве самостоятельного упражнения.

3. После определения представления данных, описывающих состояние мира, нужно спроектировать ряд функций, чтобы на их основе можно было сформировать правильное выражение `big-bang`.

Прежде всего вам понадобится функция, преобразующая любое заданное состояние в изображение, с помощью которой выражение `big-bang` сможет отобразить последовательность состояний в виде изображений:

```
| ; render
```

Затем вы должны решить, какие события будут изменять те или иные аспекты состояния мира. В зависимости от решения вам потребуется спроектировать все или часть из следующих трех функций:

```
| ; clock-tick-handler  
| ; keystroke-handler  
| ; mouse-event-handler
```

Наконец, если в постановке задачи оговаривается, что программа должна остановиться после достижения некоторого состояния, то вы должны спроектировать функцию:

```
| ; end?
```

Обобщенные сигнатуры и описание назначения этих функций вы найдете в листинге 11. Измените эти обобщенные формулировки в соответствии с конкретными задачами, которые вы решаете, чтобы читатели знали, что эти функции вычисляют. Проще говоря, желание спроектировать интерактивную про-

грамму автоматически влечет добавление нескольких начальных пунктов в список желаний. Выполните эти пункты один за другим, и вы получите законченную мировую программу.

Давайте воплотим этот шаг. Функция `big-bang` требует в обязательном порядке передать ей функцию отображения, поэтому нам остается только выяснить, понадобятся ли нам какие-либо функции обработки событий. Согласно постановке нашей задачи, автомобиль должен двигаться слева направо с течением времени, поэтому нам определенно потребуется функция, обрабатывающая события хода часов. Таким образом, получаем такой список желаний:

```
; СостояниеМира -> Изображение
; помещает изображение автомобиля в сцену BACKGROUND
; на расстоянии x пикселей от левого края
(define (render x)
  BACKGROUND)

; СостояниеМира -> СостояниеМира
; прибавляет 3 к x, чтобы переместить автомобиль вправо
(define (tock x)
  x)
```

Обратите внимание, как мы адаптировали формулировки целей к нашей задаче, чтобы рассказать, как `big-bang` будет использовать эти функции.

- Наконец, необходимо определить главную функцию (`main`). В отличие от всех других функций, главные функции мировых программ не требуют проектирования или тестирования. Единственное их предназначение – дать простую возможность запустить мировую программу из области взаимодействий в DrRacket. Единственное решение, которое потребуется принять, касается аргументов `main`. В данной задаче главная функция принимает один аргумент: начальное состояние мира:

```
; СостояниеМира -> СостояниеМира
; запускает программу с некоторым начальным состоянием
(define (main ws
  (big-bang ws
    [on-tick tock]
    [to-draw render])))
```

Теперь можно запустить эту интерактивную программу:

```
| > (main 13)
```

и наблюдать, как автомобиль начнет движение слева направо, отступив 13 пикселей от левого края. Программа остановится, когда вы закроете окно `big-bang`. Помните, что после остановки `big-bang` возвращает текущее состояние мира.

Конечно же, совсем необязательно использовать имя «Состояние-Мира» для обозначения класса данных, представляющих состояния мира. Вы можете использовать любое другое имя, при условии что будете последовательно применять его в сигнатурах функций, занимающихся обработкой событий. Точно так же необязательно использовать имена `tock`, `end?` или `render`. Эти функции можно назвать как угодно, главное, чтобы вы использовали те же имена в выражении `big-bang`. Наконец, вы, возможно, заметили, что элементы в выражении `big-bang` могут следовать в любом порядке, с единственным условием: начальное состояние всегда должно следовать первым.

Теперь рассмотрим остальную часть процесса проектирования программы, используя рецепт проектирования для функций и другие идеи, с которыми мы познакомились к данному моменту.

Упражнение 39. Хороший программист обеспечит возможность увеличения или уменьшения изображения, такого как `CAR`, изменением значения в одном месте программы. Мы начали разработку изображения автомобиля с простого определения:

Хорошие программисты устанавливают единственную точку управления для любых аспектов своих программ, а не только для графических констант. К этому вопросу мы еще не раз вернемся в последующих главах.

```
| (define WHEEL-RADIUS 5)
```

Определение `WHEEL-DISTANCE` основано на радиусе колеса. Соответственно, увеличение значения `WHEEL-RADIUS` с 5 до 10 увеличит размер изображения автомобиля вдвое. Такой способ организации программ называют *единственной точкой управления*, и эта идея должна максимально использоваться при проектировании.

Определите свое изображение автомобиля так, чтобы `WHEEL-RADIUS` оставался единственной точкой управления. ■

Следующий пункт в списке желаний – функция обработки тактов часов:

```
; СостояниеМира -> СостояниеМира
; перемещает автомобиль на 3 пикселя вправо с каждым тактом часов
(define (tock ws) ws)
```

Поскольку роль состояния мира играет расстояние между левым краем холста и автомобилем и автомобиль движется со скоростью три пикселя за такт часов, краткое описание назначения объединяет эти два факта в один. Это также упрощает создание примеров и определение функции:

```
; СостояниеМира -> СостояниеМира
; перемещает автомобиль на 3 пикселя вправо с каждым тактом часов
; примеры:
;   дано: 20, ожидаемый результат 23
;   дано: 78, ожидаемый результат 81
(define (tock ws)
  (+ ws 3))
```

Последний шаг в процессе проектирования требует убедиться в верности примеров. Итак, щелкаем на кнопке **RUN** (Выполнить) и проверяем следующие выражения:

```
> (tock 20)
23
> (tock 78)
81
```

Поскольку полученные результаты совпали с ожидаемыми, проектирование функции `tock` можно считать завершенным.

Упражнение 40. Оформите примеры в виде тестов на языке BSL, то есть в форме выражений `check-expert`. Введите ошибку в код функции и повторно выполните тесты. ■

Второй пункт в списке желаний – функция, преобразующая состояние мира в изображение:

```
; СостояниеМира -> Изображение
; помещает изображение автомобиля в сцену BACKGROUND
; в соответствии с состоянием мира
(define (render ws
  BACKGROUND))
```

Чтобы добавить примеры для функции `render`, составим таблицу (см. табл. 3). В ней перечислены заданные и ожидаемые состояния мира, а также изображения сцен (в верхней половине), которые должны получиться. Для ваших первых нескольких функций отображения состояния мира вы можете нарисовать эти изображения вручную.

Таблица 3. Примеры для программы, перемещающей автомобиль

св	Изображение
50	
100	
150	
200	

св	Выражение
50	(place-image CAR 50 Y-CAR BACKGROUND)
100	(place-image CAR 100 Y-CAR BACKGROUND)
150	(place-image CAR 150 Y-CAR BACKGROUND)
200	(place-image CAR 200 Y-CAR BACKGROUND)

Такая таблица наглядно объясняет, что будет возвращать функция – движущийся автомобиль, – но она не объясняет, как функция получает этот результат. Чтобы исправить данный недостаток, мы рекомендуем записывать выражения, подобные тем, что показаны в нижней половине табл. 3, которые создают изображения, показанные в верхней половине. Имена, состоящие исключительно из заглав-

ных букв, – это, как нетрудно догадаться, константы: изображение автомобиля (CAR), фиксированная координата у (Y-CAR) и фоновая сцена (BACKGROUND), которая в начальный момент времени пуста.

Эта расширенная таблица представляет образец формулы, используемой в теле функции render:

```
; СостояниеМира -> Изображение
; помещает изображение автомобиля в сцену BACKGROUND
; в соответствии с состоянием мира
(define (render cw)
  (place-image CAR cw Y-CAR BACKGROUND))
```

И это почти все, что нужно для создания простой мировой программы.

Упражнение 41. Завершите пример и запустите программу. То есть после решения упражнения 39 определите константы BACKGROUND и Y-CAR. Затем добавьте все определения функций, включая тесты для них. Добившись успешного выполнения всех тестов, добавьте дерево для создания декораций. Мы использовали такой код:

```
(define tree
  (underlay/xy (circle 10 "solid" "green")
               9 15
               (rectangle 2 20 "solid" "brown")))
```

чтобы создать фигуру, напоминающую дерево. Также добавьте в выражение big-bang предложение, останавливающее анимацию, как только автомобиль скроется за правым краем сцены. ■

Определившись с представлением состояния мира в виде данных, внимательный программист, возможно, не раз пересмотрит это фундаментальное проектное решение на последующих этапах процесса проектирования. Например, в нашей задаче мы представляем автомобиль как точку. Но изображение автомобиля – это не просто математическая точка, не имеющая ни ширины, ни высоты. Поэтому интерпретация предложения «количество пикселей от левого края» не выглядит однозначной. Что имеется в виду? Расстояние от левого края сцены до левой стороны автомобиля? До точки в его центре? Или до правой стороны? Мы проигнорировали эту проблему и положились в принятии решений на примитивы обработки изображений в BSL. Если вам не нравится результат, вернитесь к определению данных и измените его или его интерпретацию по своему вкусу.

Упражнение 42. Измените интерпретацию определения данных так, чтобы состояние обозначало координату X правой стороны автомобиля. ■

Упражнение 43. Рассмотрим ту же постановку задачи с определением данных на основе времени:

```
; СостояниеАнимации – это число.
; интерпретация: число тактов часов,
; прошедших с начала анимации
```

Как и в предыдущем определении данных, это также относит состояние мира к классу чисел. Однако его интерпретация объясняет, что данное число означает нечто совершенно иное.

Спроектируйте функции `tock` и `render`. Затем определите выражение `big-bang`, чтобы вновь получить анимацию автомобиля, движущегося слева направо.

*Иногда бывает сложно
правильно обрабаты-
вать движения мыши,
потому что они
не совсем такие,
какими кажутся.
Чтобы коротко познако-
миться с причинами,
прочтите примечание
«О мышах и клавишиах»
в онлайн-версии книги.*

Как, по вашему мнению, эта программа связана с `animate` из пролога?

Используйте определение данных, чтобы спроектировать программу, которая перемещает автомобиль по синусоидальной траектории. (В жизни не надо так ездить!) ■

Мы заканчиваем этот раздел иллюстрацией обработки событий мыши, которая также демонстрирует преимущества разделения представления и модели. Предположим, мы решили дать людям возможность перемещать автомобиль через «гиперпространство»:

Задача. Спроектируйте программу, которая перемещает автомобиль в пределах холста слева направо, по три пикселя за такт часов. **В ответ на щелчок мышью в любой точке в пределах холста программа должна переместить автомобиль в точку с координатой x , соответствующей координате x щелчка.**

Здесь жирным выделено предложение, дополняющее формулировку предыдущей задачи.

Столкнувшись с изменившейся задачей, мы используем процесс проектирования, чтобы внести необходимые изменения. При правильном применении этот процесс естественным образом определяет, что нужно добавить в существующую программу, чтобы удовлетворить расширенную постановку задачи. Итак, начнем.

- Новых свойств не добавилось, а значит, нам не нужны новые константы.
- Программа все так же обрабатывает единственное свойство, изменяющееся с течением времени, – координату x автомобиля. Следовательно, имеющегося представления данных достаточно.
- Обновленная постановка задачи требует добавить обработчик событий мыши, но не требует отказаться от перемещения автомобиля по часам. Поэтому добавляем следующий пункт в список желаний:

```
; СостояниеМира Число Число Стока -> СостояниеМира
; перемещает автомобиль в координату x события,
; если это событие "button-down"
(define (hyper x-position-of-car x-mouse y-mouse me)
  x-position-of-car)
```

- Наконец, нам нужно изменить функцию `main` и включить обработку событий мыши. Для этого достаточно добавить пред-

ложение `on-mouse`, которое напрямую связано с новым пунктом в нашем списке желаний:

```
(define (main ws)
  (big-bang ws
    [on-tick tock]
    [on-mouse hyper]
    [to-draw render]))
```

Как видите, изменившаяся постановка задачи требует лишь добавить обработку щелчков мышью, а все остальное остается прежним.

Теперь осталось лишь спроектировать еще одну функцию, и для этого мы используем рецепт проектирования функций.

Первые два шага рецепта проектирования функций мы уже выполнили. Теперь следующий шаг – определить несколько примеров применения функции:

```
; СостояниеМира Число Число Стока -> СостояниеМира
; перемещает автомобиль в координату x события,
; если это событие "button-down"
; дано: 21 10 20 "enter"
; ожидаемый результат: 21
; дано: 42 10 20 "button-down"
; ожидаемый результат: 10
; дано: 42 10 20 "move"
; ожидаемый результат: 42
(define (hyper x-position-of-car x-mouse y-mouse me)
  x-position-of-car)
```

В примерах говорится, что если в строковом параметре передается строка "button-down", то функция должна вернуть значение `x-mouse`; во всех остальных случаях должно возвращаться значение `x-position-of-car`.

Упражнение 44. Оформите примеры в виде тестов BSL. Щелкните на кнопке **RUN** (Выполнить) и посмотрите, как все они терпят неудачу. ■

Чтобы завершить определение функции, вспомним, о чем рассказывалось в прологе, в частности об условном выражении `cond`. Используя `cond`, можно уместить определение `hyper` в две строки:

```
; СостояниеМира Число Число Стока -> СостояниеМира
; перемещает автомобиль в координату x события,
; если это событие "button-down"
(define (hyper x-position-of-car x-mouse y-mouse me)
  (cond
    [(string=? "button-down" me) x-mouse]
    [else x-position-of-car]))
```

В следующей главе мы подробно расскажем о проектировании с помощью `cond`.

Если вы выполнили упражнение 41, запустите программу еще раз и обратите внимание, что на этот раз все тесты должны выполниться успешно. Если никаких ошибок не возникло, выполните выражение

```
| (main 1)
```

в области взаимодействий DrRacket и попробуйте переместить автомобиль через гиперпространство.

Вы можете спросить, почему нам так легко удалось изменить программу. Тому есть две причины. Во-первых, эта книга и описываемое в ней программное обеспечение строго разделяют данные, за которыми следит программа, – модель – и изображение, которое она показывает, – представление (вид). В частности, функции, обрабатывающие события, никак не затрагивают отображение состояния. Чтобы изменить способ отображения состояния, нам достаточно будет сосредоточиться на функции, указанной в предложении `to-draw`. Во-вторых, рецепты проектирования программ и функций правильно организуют программы. Если что-то изменится в постановке задачи, то повторное выполнение пунктов рецепта проектирования естественным образом укажет, где необходимо внести изменения в первоначальное решение. Это может показаться очевидным для простых задач, с которыми мы сейчас имеем дело, но такой подход критически важен для задач, с которыми программисты сталкиваются в реальном мире.

3.7. Мирь виртуальных питомцев

Упражнения в этом разделе вводят два первых элемента игры с виртуальным питомцем. В самом начале мы имеем изображение кошки, которая прогуливается по холсту. В процессе прогулки кошка устает и показывает свою усталость. Как и с любым домашним питомцем, вы можете попробовать погладить кошку, чтобы немного поднять ей настроение, или даже покормить, что поднимет ей настроение еще больше.

Итак, начнем с изображения нашей любимой кошки:



Скопируйте изображение кошки и вставьте его в DrRacket, затем дайте изображению имя с помощью `define`, как показано выше.

Упражнение 45. Спроектируйте мировую программу «виртуальная кошка», которая непрерывно перемещает кошку слева направо. Назовем ее `cat-prog` и предположим, что она принимает аргумент с начальной позицией кошки. Кроме того, заставьте кошку перемещаться на три пикселя за каждый тик часов. Всякий раз, когда кошка будет исчезать за правым краем сцены, она должна появляться из-за левого края. Для этого вам может пригодиться функция `modulo`. ■

Упражнение 46. Улучшите анимацию кошки, используя немного другое изображение:



```
| (define cat2 )
```

Измените функцию отображения из упражнения 45 так, чтобы она использовала первое или второе изображение кошки, в зависимости от четности или нечетности координаты x . Прочтайте описание функции `odd?` в HelpDesk и используйте выражение `cond` для выбора изображения кошки. ■

Упражнение 47. Спроектируйте мировую программу, которая поддерживает и отображает «шкалу счастья». Назовем это `gauge-program` и предположим, что она принимает аргумент с максимальным уровнем счастья. В первый момент программа должна показать на шкале указанный максимальный уровень счастья. С каждым тиком часов этот уровень должен уменьшаться на -0.1 , но никогда не опускаться ниже 0 . Каждый раз, когда нажимается клавиша со стрелкой вниз, уровень счастья должен увеличиваться на $1/5$; а каждый раз, когда нажимается стрелка вверх, уровень счастья должен увеличиваться на $1/3$.

Для отображения уровня счастья используйте сцену со сплошным красным прямоугольником в черной рамке. Если уровень счастья равен 0 , прямоугольник должен исчезнуть; если уровень счастья максимальный, прямоугольник должен пересекать всю сцену.

Примечание. Когда вы получите достаточный объем знаний, мы покажем, как совместить программу `gauge-program` с решением упражнения 45. После этого вы сможете помочь кошке стать более счастливой. Если погладить кошку, ее настроение немного улучшится. Если покормить ее, ее настроение улучшится намного больше.

Итак, теперь у вас наверняка проснулось желание больше узнать о разработке мировых программ, чем рассказали первые три главы. ■

4. Интервалы, перечисления и детализация

На данный момент вы знаете четыре варианта представления информации в виде данных: числа, строки, изображения и логические значения. Для многих задач этого достаточно, но есть много других задач, для которых этих четырех типов данных в BSL (или других языках программирования) недостаточно. Настоящим проектировщикам нужны дополнительные способы представления информации в виде данных.

Хорошие программисты должны уметь проектировать программы с ограничениями на эти встроенные типы данных. Один из примеров подобных ограничений – перечислить несколько элементов некоторого типа и сказать, что только эти элементы будут использоваться для решения поставленной задачи. Перечисление элементов возможно, только когда их количество ограничено. Чтобы приспособить под свои нужды типы с «бесконечным» количеством элементов, мы вводим интервалы – наборы элементов, удовлетворяющих определенному свойству.

Под словом «бесконечное» может просто подразумеваться «настолько большое количество элементов, что прямое их перечисление не представляется возможным».

Определение перечислений и интервалов предполагает различие элементов разных типов. Для их различия в коде требуются условные функции, то есть функции, выбирающие разные способы вычисления результата в зависимости от значения некоторого аргумента. Как писать такие функции, мы рассказывали в разделе «Множество способов вычисления» в прологе и в разделе 1.6. Однако ни там, ни там не использовались рецепты проектирования. В обоих разделах мы просто представили некоторые конструкции на нашем любимом языке программирования (BSL) и показали несколько примеров их использования.

В этой главе мы обсудим общий подход к проектированию перечислений и интервалов – новых форм описания данных. Начнем с того, что вновь рассмотрим выражение `cond`. Затем обсудим три разных типа описаний данных: перечисления, интервалы и детализации. Перечисление указывает каждый отдельный элемент данных, принадлежащий перечислению, а интервал определяет диапазон данных. Последний тип описания, детализация, смешивает первые два, указывая диапазоны в одном предложении своего определения и конкретные элементы данных в другом. В заключение главы мы представим общую стратегию проектирования для таких ситуаций.

4.1. Программирование с условиями

Давайте вспомним краткое введение в условные выражения в прологе. Поскольку `cond` – самая сложная форма выражения в этой книге, внимательно рассмотрим его общую форму определения:

```
(cond
  [ВыражениеУсловия1 ВыражениеРезультата1]
  [ВыражениеУсловия2 ВыражениеРезультата2]
  ...
  [ВыражениеУсловияN ВыражениеРезультатаN])
```

Квадратные скобки отделяют друг от друга разные ветви условного выражения. Вместо [...] можно использовать (...).

Условное выражение начинается с (`cond` и заканчивается закрывающей круглой скобкой). Вслед за ключевым словом программист записывает столько ветвей `cond`, сколько необходимо; каждая ветвь состоит из двух выражений, заключенных в квадратные скобки: [и].

Ветви условного выражения `cond` также часто называют *условиями* или *предложениями cond*.

Вот пример функции, в которой используется условное выражение:

```
(define (next traffic-light-state)
  (cond
    [(string=? "red" traffic-light-state) "green"]
    [(string=? "green" traffic-light-state) "yellow"]
    [(string=? "yellow" traffic-light-state) "red"]))
```

Подобно математическому примеру в прологе, этот пример иллюстрирует удобство использования условных выражений `cond`. Во многих задачах функция должна различать несколько разных ситуаций. В выражении `cond` можно определить по одной ветви для каждой ситуации и тем самым напомнить читателю кода, что в постановке задачи говорится о разных условиях.

Замечание с прагматической точки зрения: сравните выражение `cond` с выражениями `if` из раздела 1.6. Последние отличают одну конкретную ситуацию от всех остальных и хуже подходят для задач с несколькими ситуациями; их лучше всего использовать, когда мы хотим сказать: «или так, или иначе». Поэтому мы всегда используем `cond`, когда хотим напомнить читателю нашего кода, что из определений данных вытекает несколько разных ситуаций. В других фрагментах кода мы используем любую наиболее удобную конструкцию.

В ситуациях со сложными условиями в выражении `cond` иногда бывает желательно иметь возможность сказать: «во всех остальных случаях». В таких случаях выражения `cond` разрешают использовать ключевое слово `else` в самой последней строке в определении `cond`:

```
(cond
  [ВыражениеУсловия1 ВыражениеРезультата1]
  [ВыражениеУсловия2 ВыражениеРезультата2]
  ...
  [else ВыражениеРезультатаПоУмолчанию])
```

Если по ошибке ключевое слово `else` окажется в любой другой строке в определении выражения `cond`, то DrRacket сообщит об ошибке:

```
> (cond
  [(> x 0) 10]
  [else 20]
  [(< x 10) 30])
```

```
| cond:found an else clause that isn't the last clause in its
| cond expression
```

(cond: встретено условие else, находящееся не в последней строке в выражении cond).

То есть BSL отвергает грамматически неправильные фразы, потому что нет смысла выяснять, что такие фразы могли бы означать.

Представьте проект функции в игровой программе, которая вычисляет некоторую награду в конце игры. Вот ее заголовок:

```
; ПоложительноеЧисло -- это Число, которое больше или равно 0.
; ПоложительноеЧисло -> Страна
; вычисляет награду по заданному числу очков s
```

А вот два варианта ее реализации для сравнения:

<pre>(define (reward s) (cond [(<= 0 s 10) "bronze"] [(and (< 10 s) (<= s 20)) "silver"] [(< 20 s) "gold"]))</pre>	<pre>(define (reward s) (cond [(<= 0 s 10) "bronze"] [(and (< 10 s) (<= s 20)) "silver"] [else "gold"]))</pre>
---	--

Вариант слева использует выражение cond с тремя полноценными условиями; вариант справа содержит условие else. Чтобы сформулировать последнее условие в функции слева, необходимо проверить выполнение условия ($< 20 s$), даже притом что мы знаем, что:

- s – это ПоложительноеЧисло;
- выражение ($<= 0 s 10$) вернуло #false;
- выражение (and ($< 10 s$) ($<= s 20$)) также вернуло #false.

В этом примере используется простое условие, но вообще в подобных ситуациях легко допустить небольшие ошибки или опечатки, которые сделают поведение программы непредсказуемым. Поэтому лучше определить функцию, как показано справа, если известно, что вам нужна полная противоположность – называемая *дополнением* – всех предыдущих условий в cond.

4.2. Условные вычисления

Из разделов «Множество способов вычисления» в прологе и разделе 1.6 вы примерно знаете, как DrRacket вычисляет условные выражения. Давайте теперь подробнее рассмотрим работу выражений cond. Взгляните еще раз на уже знакомое определение функции:

```
| (define (reward s)
|   (cond
```

```
| [(<= 0 s 10) "bronze"]
| [(and (< 10 s) (<= s 20)) "silver"]
| [else "gold"]])
```

Эта функция принимает число очков – положительное число – и в результате дает достоинство награды.

Просто глядя на выражение cond, нельзя сказать, какое из трех ветвей в cond будет использоваться. И в этом вся суть функции. Она может получать разные входные значения, например 2, 3, 7, 18, 29. Для каждого из этих значений может потребоваться действовать по-разному. Цель выражения cond – дать возможность различать разные классы входных данных.

Возьмем, например:

```
| (reward 3)
```

Вы знаете, что DrRacket замещает применение функции ее телом после подстановки аргументов вместо параметров. То есть

```
(reward 3) ; далее читаем "'равно'"
==
(cond
  [(<= 0 3 10) "bronze"]
  [(and (< 10 3) (<= 3 20)) "silver"]
  [else "gold"])
```

На этом этапе DrRacket вычисляет условия друг за другом и для первого, которое дало #true, продолжает процесс вычисления с выражением-результатом:

```
(reward 3)
==
(cond
  [(<= 0 3 10) "bronze"]
  [(and (< 10 3) (<= 3 20)) "silver"]
  [else "gold"])
 ==
(cond
  [#true "bronze"]
  [(and (< 10 3) (<= 3 20)) "silver"]
  [else "gold"])
 ==
"bronze"
```

Здесь выполнилось первое условие, потому что 3 находится между 0 и 10.

Вот второй пример:

```
(reward 21)
==
(cond
  [(<= 0 21 10) "bronze"]
  [(and (< 10 21) (<= 21 20)) "silver"]
  [else "gold"])
 ==
(cond
```

```

[#false "bronze"]
[(and (< 10 21) (≤ 21 20)) "silver"]
[else "gold"])
== 
(cond
  [(and (< 10 21) (≤ 21 20)) "silver"]
  [else "gold"])

```

Обратите внимание, что первое условие на этот раз дает `#false`, и, как упоминалось в разделе «Множество способов вычисления», вся эта ветка отбрасывается. Далее вычисления выполняются в полном соответствии с ожиданиями:

```

(cond
  [(and (< 10 21) (≤ 21 20)) "silver"]
  [else "gold"])
== 
(cond
  [(and #true (≤ 21 20)) "silver"]
  [else "gold"])
== 
(cond
  [(and #true #false) "silver"]
  [else "gold"])
== 
(cond
  [#false "silver"]
  [else "gold"])
== 
(cond
  [else "gold"])
== 
"gold"

```

Второе условие, как и первое, тоже дает `#false`, после чего вычисление переходит к третьей строчке в `cond`. Ключевое слово `else` указывает DrRacket, что все выражение `cond` должно заменить ответом из этой ветки.

Упражнение 48. Введите определение функции `reward` и ее вызов (`reward 18`) в области определений DrRacket и, используя движок пошаговых вычислений, посмотрите, как DrRacket применяет функцию. ■

Упражнение 49. Выражение `cond` на самом деле является самым обычным выражением, и поэтому его можно вставить внутрь другого выражения:

```
| (- 200 (cond [(> y 200) 0] [else y]))
```

Используя движок пошаговых вычислений, посмотрите, как вычисляется это выражение со значениями `y`, равными 100 и 210.

Вложенные выражения `cond` можно использовать взамен обычных выражений `cond`. Взгляните на определение функции запуска ракеты в листинге 12. Обе ветви выражения `cond` выглядят абсолютно одинаково, за исключением фрагмента, обозначенного троеточием `...`:

```
| (place-image ROCKET X ... MTSCN)
```

Листинг 12. Программа из раздела «Одна программа, множество определений»

```
(define WIDTH 100)
(define HEIGHT 60)
(define MTSCN (empty-scene WIDTH HEIGHT))

(define ROCKET )
(define ROCKET-CENTER-TO-TOP
  (- HEIGHT (/ (image-height ROCKET) 2)))

(define (create-rocket-scene.v5 h)
  (cond
    [(<= h ROCKET-CENTER-TO-TOP)
     (place-image ROCKET 50 h MTSCN)]
    [(> h ROCKET-CENTER-TO-TOP)
     (place-image ROCKET 50 ROCKET-CENTER-TO-TOP MTSCN)]))
```

Реорганизуйте функцию `create-rocket-scene.v5` так, чтобы она использовала вложенное выражение `cond`; новая функция должна содержать только одну ссылку на `place-image`. ■

4.3. Перечисления

Не все строки представляют события мыши. Если заглянуть в Help-Desk, в раздел с описанием предложения `on-mouse` для `big-bang`, то можно обнаружить, что только шесть строк используются для уведомления программ о событиях мыши:

```
; События мыши -- это одна из следующих строк:
; -- "button-down"
; -- "button-up"
; -- "drag"
; -- "move"
; -- "enter"
; -- "leave"
```

Интерпретация этих строк очевидна. Первые две строки появляются, когда пользователь компьютера нажимает и отпускает кнопку мыши соответственно. Третья и четвертая строки сопровождают событие перемещения указателя мыши, возможно, с одновременным удержанием нажатой кнопки мыши. Наконец, последние две строки представляют события перемещения указателя мыши через границу холста: либо внутрь области холста, либо наружу за его пределы.

Что особенно важно, определение данных для представления строковых событий мыши сильно отличается от определений данных, которые мы видели до сих пор. Это называется *перечислением*: в этом представлении перечислены все возможные варианты. Неудивительно, что перечисления являются обычным явлением. Вот простой пример:

```
; Светофор -- это одна из следующих строк:
```

```

; -- "red"
; -- "green"
; -- "yellow"
; интерпретация: три строки представляют три
; состояния, которые может принимать светофор

```

Это упрощенное представление состояний, которые может принимать светофор. В отличие от других, это определение данных также использует несколько иную формулировку объяснения значения термина Светофор, но это несущественная разница.

Пользоваться перечислениями в большинстве случаев несложно. Когда входные данные функции представляют класс данных, описание которого определяет элементы для каждого конкретного случая, функция должна различать только эти случаи и вычислять результат для каждого из них. Например, вот как можно определить функцию, которая вычисляет следующее состояние светофора на основе текущего:

```

; Светофор -> Светофор
; возвращает следующее состояние, опираясь на текущее состояние s
(check-expect (traffic-light-next "red") "green")
(define (traffic-light-next s)
  (cond
    [(string=? "red" s) "green"]
    [(string=? "green" s) "yellow"]
    [(string=? "yellow" s) "red"]))

```

Поскольку определение данных Светофор включает три отдельных элемента, функция `traffic-light-next` различает три разных случая. Результатом для каждого случая является другая строка, описывающая следующее состояние светофора.

Упражнение 50. Если скопировать и вставить определение функции `traffic-light-next` в область определений DrRacket и щелкнуть на кнопке **RUN** (Выполнить), то DrRacket выделит два из трех условий в операторе `cond`. Таким способом DrRacket подсказывает, что представленные тесты охватывают не все условия. Добавьте еще пару тестов, чтобы удовлетворить DrRacket. ■

Упражнение 51. Спроектируйте программу `big-bang`, которая имитирует работу светофора в течение заданного интервала времени. Программа должна отображать состояние светофора в виде круга соответствующего цвета и менять состояние с каждым тактом часов. Какое начальное состояние наиболее подходящее? Спросите своих друзей-инженеров. ■

Основная идея перечисления состоит в том, чтобы определить набор данных как конечное число экземпляров. Каждый экземпляр явно указывает, какие данные принадлежат к определяемому классу. Обычно экземпляры данных отображаются как есть, но иногда элементы перечисления являются обычными предложениями на естественном языке, которое описывает конечное число элементов данных с помощью одной фразы.

Например:

```
; Символ (1String) -- это строка с длиной, равной 1,
; включая
; -- "\\\" (обратный слеш),
; -- " " (пробел),
; -- "\\t" (символ табуляции),
; -- "\\r" (возврат каретки)
; -- "\\b" (забой).
; интерпретация: клавиши на клавиатуре
```

Мы знаем, что такое определение данных является правильным, если все его экземпляры можно описать с помощью теста BSL. Вот как в случае с символом (1String) можно узнать, принадлежит ли некоторая строка *s* классу символов:

```
| (= (string-length s) 1)
```

Альтернативный способ проверки – перечислить все экземпляры описываемого класса данных:

```
; Символ (1String) -- это одна из строк:
; -- "q"
; -- "w"
; -- "e"
; -- "г"
;
; ...
; -- "\\t"
; -- "\\r"
; -- "\\b"
```

Если посмотреть на клавиатуру, то можно обнаружить \leftarrow , \uparrow и другие подобные обозначения кнопок. Выбранный нами язык программирования BSL использует собственное определение данных для представления этой информации. Например:

Вы уже знаете, где найти полное определение.

```
; СобытиеКлавиатуры -- это одна из строк:
; -- 1String
; -- "left"
; -- "right"
; -- "up"
; -- ...
```

Первый элемент в этом перечислении описывает ту же группу строк, что и класс символа 1String. Последующие элементы описывают строки, обозначающие специальные события, такие как нажатие или отпускание одной из четырех клавиш со стрелками.

Теперь мы можем систематически проектировать обработчики событий клавиатуры. Например:

```
; СостояниеМира СобытиеКлавиатуры -> ...
(define (handle-key-events w ke)
  (cond
    [(= (string-length ke) 1) ...]
    [(string=? "left" ke) ...]
```

```
[(string=? "right" ke) ...]
[(string=? "up" ke) ...]
[(string=? "down" ke) ...]
...))
```

Эта функция обработки событий использует выражение `cond` с отдельным условием для каждой строки в перечислении определения данных. Первое условие в `cond` идентифицирует первый элемент перечисления СобытиеКлавиатуры, второе условие – второй элемент перечисления и т. д.

Программы, полагающиеся на определения данных, зависящие от выбранного языка программирования (например, BSL) или его библиотек (таких как библиотека `2htdp/universe`), обычно используют только часть перечисления. Чтобы проиллюстрировать этот момент, давайте рассмотрим типичную задачу.

Задача. Спроектируйте обработчик событий клавиатуры, который перемещает красную точку влево или вправо по горизонтали линии в ответ на нажатие клавиш со стрелками влево и вправо.

Листинг 13. Функции с условиями и специальные перечисления

```
; Позиция -- это Число.
; интерпретация: расстояние от левого края до шарика

; Позиция СобытиеКлавиатуры -> Позиция
; вычисляет следующее местоположение для шарика
(check-expect (keh 13 "left") 8)
(check-expect (keh 13 "right") 18)
(check-expect (keh 13 "a") 13)

(define (keh p k)
  (cond
    [(= (string-length k) 1)
     p]
    [(string=? "left" k)
     (- p 5)]
    [(string=? "right" k)
     (+ p 5)]
    [else p]))

(define (keh p k)
  (cond
    [(string=? "left" k)
     (- p 5)]
    [(string=? "right" k)
     (+ p 5)]
    [else p]))
```

В листинге 13 представлены два решения этой задачи. Функция слева организована в соответствии с основной идеей использования одного условия `cond` одному элементу в определении класса входных данных СобытиеКлавиатуры. Функция справа использует только три необходимые строки: две для клавиш со стрелками влево и вправо и одна для всего остального. Переупорядочивание условий вполне допустимо, потому что для этой задачи значение имеют только две строки в `cond` и их можно четко отделить от других строк. Естественно, такая перестановка возможна только при надлежащем подходе к проектированию функции.

4.4. Интервалы

Представьте, что вам поручено реализовать следующую задачу.

Задача. Спроектируйте программу, имитирующую спуск НЛО.

После некоторых размышлений вы могли бы написать такой код, как в листинге 14. Стоп! Изучите определения и замените многоточия, прежде чем читать дальше.

Листинг 14. Спуск НЛО

```
; СостояниеМира -- это Число.  
; интерпретация: расстояние в пикселях от верхнего края до НЛО  
  
(define WIDTH 300) ; distances in terms of pixels  
(define HEIGHT 100)  
(define CLOSE (/ HEIGHT 3))  
(define MTSCN (empty-scene WIDTH HEIGHT))  
(define UFO (overlay (circle 10 "solid" "green") ...))  
  
; СостояниеМира -> СостояниеМира  
(define (main y0)  
  (big-bang y0  
    [on-tick nxt]  
    [to-draw render]))  
  
; СостояниеМира -> СостояниеМира  
; вычисляет следующее местоположение НЛО  
(check-expect (nxt 11) 14)  
(define (nxt y)  
  (+ y 3))  
  
; СостояниеМира -> Изображение  
; помещает НЛО на заданной высоте в центре сцены MTSCN по горизонтали  
(check-expect (render 11) (place-image UFO ... 11 MTSCN))  
(define (render y)  
  (place-image UFO ... y MTSCN))
```

Однако перед выпуском этой «игровой» программы хорошо бы добавить в холст строку с описанием состояния:

Задача. Добавьте строку с описанием состояния. Стока должна сообщать "descending" (спуск), когда высота НЛО превышает одну треть высоты холста. Затем должно появиться сообщение "closing in" (сближение). И наконец, когда НЛО достигает нижнего края холста, строка состояния должна уведомить игрока, что НЛО приземлилось, сообщением "landed" (посадка завершена).

Вы можете использовать подходящие цвета для строки состояния.

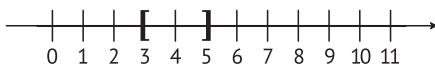
В данном случае у нас нет перечисления отдельных элементов или подклассов данных. Концептуально интервал между 0 и HEIGHT содержит бесконечное количество чисел и большое количество целых чисел. Поэтому, чтобы придать некоторую организацию слишком об-

щему определению данных, использующему просто «числа» для описания классов координат, мы используем интервалы.

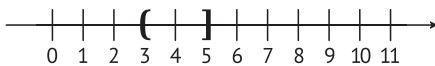
Интервал – это описание класса чисел с использованием границ. Самый простой интервал имеет две границы: левую и правую. Если левая граница входит в интервал, то мы говорим, что интервал *закрыт слева*. Точно так же если правая граница входит в интервал, то мы говорим, что интервал *закрыт справа*. Наконец, если интервал не включает какую-то границу, то мы говорим, что он *открыт* на этой границе.

Закрытые границы в изображениях интервалов обозначаются квадратными скобками, а открытые – круглыми. Вот четыре таких интервала:

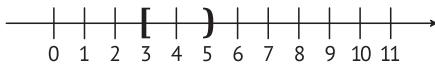
- $[3,5]$ – закрытый интервал:



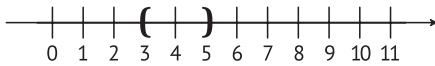
- $(3,5]$ – интервал, открытый слева:



- $[3,5)$ – интервал, открытый справа:



- $(3,5)$ – открытый интервал:

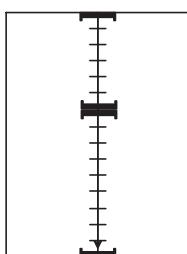


Упражнение 52. Какие целые числа входят в четыре интервала, указанных выше? ■

Идея интервала помогает сформулировать определение данных, лучше отражающее постановку задачи, чем определение, основанное на «числах»:

```
; СостояниеМира входит в один из трех интервалов:  
; -- между 0 и CLOSE  
; -- между CLOSE и HEIGHT  
; -- меньше HEIGHT
```

Вот как можно изобразить эти три интервала:



Здесь показана стандартная числовая прямая, повернутая вертикально и разбитая на интервалы. Каждый интервал начинается с квадратной скобки, обращенной вниз (└), и заканчивается квадратной скобкой, обращенной вверх (┘). Таким образом на изображении отмечены три интервала:

- верхний интервал простирается от 0 до CLOSE;
- средний интервал простирается от CLOSE до HEIGHT;
- нижний, невидимый интервал отмечен просто линией в точке HEIGHT.

На простой числовой прямой последний интервал начинается с точки HEIGHT и уходит в бесконечность.

Визуальное представление определения данных помогает при проектировании функций. Во-первых, сразу подсказывает, как определять принадлежность к тому или иному интервалу. Наша задача – реализовать функцию так, чтобы она правильно работала внутри всех интервалов на концах каждого интервала. Во-вторых, изображение сообщает, что мы должны сформулировать условие, которое определяет, находится ли некоторая «точка» в пределах одного из интервалов.

При изучении рисунка сразу возникает вопрос, а именно: как функция должна обрабатывать конечные точки интервалов. В нашем примере есть две точки на числовой прямой, которые относятся к двум интервалам: CLOSE, принадлежащая верхнему и среднему интервалам, и HEIGHT, принадлежащая среднему и нижнему интервалам. Такие совпадения обычно вызывают проблемы в программах, и их следует избегать.

Функции в BSL избегают их естественным образом благодаря особенностям вычисления выражений cond. Рассмотрим эту естественную организацию функции, которая получает элементы из класса СостояниеМира:

```
; СостояниеМира -> СостояниеМира
(define (f y)
  (cond
    [(<= 0 y CLOSE) ...]
    [(<= CLOSE y HEIGHT) ...]
    [(>= y HEIGHT) ...]))
```

Три условия в cond соответствуют трем интервалам. Каждое условие определяет значения y, находящиеся в пределах своего интервала. Однако из-за того, что условия в cond проверяются одно за другим, значение y, равное CLOSE, выбирается первым условием, а значение y, равное HEIGHT, – вторым.

Чтобы сделать этот выбор более очевидным для каждого, кто будет читать данный код, можно использовать другие условия:

```
; СостояниеМира -> СостояниеМира
(define (g y)
  (cond
    [(<= 0 y CLOSE) ...]
    [(and (< CLOSE y) (<= y HEIGHT)) ...]
    [(> y HEIGHT) ...]))
```

Обратите внимание, как во втором условии используется оператор `and` для объединения условий «строго меньше» и «меньше или равно» вместо условия `<=` с тремя аргументами.

Теперь мы можем завершить определение функции, которая добавляет строку состояния в анимацию посадки НЛО; см. листинг 15. Функция использует выражение `cond`, чтобы определить принадлежность заданного значения к одному из трех интервалов. В каждом условии в `cond` используется выражение с функцией `render` (из листинга 14), которая создает изображение со снижающимся НЛО, а затем помещает соответствующий текст в позицию (10,10), вызывая `place-image`.

Листинг 15. Отображение НЛО со строкой состояния

```
; СостояниеМира -> Изображение
; добавляет строку состояния в сцену, созданную функцией render
(check-expect (render/status 10)
  (place-image (text "descending" 11 "green")
    10 10
    (render 10)))

(define (render/status y)
  (cond
    [(<= 0 y CLOSE)
     (place-image (text "descending" 11 "green")
       10 10
       (render y))]
    [(and (< CLOSE y) (<= y HEIGHT))
     (place-image (text "closing in" 11 "orange")
       10 10
       (render y))]
    [(> y HEIGHT)
     (place-image (text "landed" 11 "red")
       10 10
       (render y))]))
```

Чтобы запустить эту версию, нужно немного изменить функцию `main` из листинга 14:

```
; СостояниеМира -> СостояниеМира
(define (main y0)
  (big-bang y0
    [on-tick nxt]
    [to-draw render/status]))
```

Кое-что в этом определении функции должно вызвать у вас беспокойство, и чтобы сделать это более явным, давайте немного уточним постановку задачи:

Задача. Добавьте строку с описанием состояния, **расположенную в точке (20,20)**. Стока должна сообщать "descending" (спуск), когда высота НЛО превышает одну третью высоты холста...

Такое уточнение может поступить от клиента, который впервые посмотрел вашу анимацию.

На этом этапе у вас нет другого выбора, кроме как изменить функцию `render/status` в **шести** местах, потому что у нас есть три копии одного внешнего элемента информации: местоположение строки состояния. Чтобы избавиться от многократных изменений одного элемента, программисты стараются избегать использования копий. Решить эту проблему можно двумя способами. Первый – использовать константы, с которыми вы познакомились в первых главах. Второй – рассматривать выражения `cond` как самые обычные выражения, которые могут появляться где угодно в функции, в том числе в середине любого другого выражения; взгляните на листинг 16 и сравните его с листингом 15. В этом новом определении `render/status` выражение `cond` является первым аргументом в `place-image`. Как видите, результатом всегда является изображение `text`, которое помещается в позицию (20,20) в изображение, созданное с помощью (`render y`).

Листинг 16. Отображение НЛО со строкой состояния, улучшенная версия

```
; СостояниеМира -> Изображение
; добавляет строку состояния в сцену, созданную функцией render
(check-expect (render/status 42)
  (place-image (text "closing in" 11 "orange")
    20 20
    (render 42)))
(define (render/status y)
  (place-image
    (cond
      [(<= 0 y CLOSE)
        (text "descending" 11 "green")]
      [(and (< CLOSE y) (<= y HEIGHT))
        (text "closing in" 11 "orange")]
      [(> y HEIGHT)
        (text "landed" 11 "red")])
    20 20
    (render y)))
```

4.5. Детализация

Интервал выделяет особый подкласс чисел, которых в принципе бесконечно много. Перечисление определяет новый класс, состоящий из полезных элементов существующего класса данных. Однако иногда определения данных должны включать элементы обеих этих категорий. Они используют *детализации*, обобщающие интервалы и перечисления, и позволяют комбинировать любые уже определенные классы данных друг с другом и с отдельными экземплярами данных.

Вернемся к примеру из раздела 4.3 с одним важным определением данных:

```
; СобытиеКлавиатуры -- это одна из строк:
; -- 1String
; -- "left"
```

```
| ; -- "right"
| ; -- "up"
| ; -- ...
```

В данном случае определение СобытиеКлавиатуры ссылается на определение произвольного символа 1String. Функции, обслуживающие события клавиатуры, часто обрабатывают простые символы 1String отдельно и используют для этого вспомогательные функции, соответственно, мы получаем удобный способ выразить сигнатуры и для этих функций.

В описании функции string->number как раз используется идея детализации. Вот ее сигнатура:

```
| ; Стока -> ЧислоИлиЛожь
| ; преобразует строку в число;
| ; если преобразование невозможно, то возвращает #false
(define (string->number s) (... s ...))
```

Согласно описанию в этой сигнатуре, результатом этой функции является класс данных:

```
| ; ЧислоИлиЛожь -- это одно из значений:
| ; -- #false
| ; -- Число
```

Эта детализация объединяет один экземпляр данных (#false) с большим классом данных другого типа (Число).

Теперь представьте функцию, которая принимает результат функции string->number и прибавляет к нему 3, интерпретируя #false как число 0:

```
| ; ЧислоИлиЛожь -> Число
| ; прибавляет 3 к заданному числу;
| ; если аргумент не является числом, то результат -- 3
(define (add3 x)
  (cond
    [(false? x) 3]
    [else (+ x 3)]))
```

Тело функции состоит из выражения cond с количеством условий, равным количеству элементов в перечислении определения данных. Первое условие определяет ситуацию, когда функция применяется к #false и возвращает результат 3. Второе условие касается чисел и прибавляет 3 к указанному числу.

Рассмотрим более содержательную задачу проектирования:

Задача. Спроектируйте программу, которая запускает ракету, когда пользователь программы нажимает клавишу пробела. Программа сначала отображает ракету, стоящую внизу холста. После запуска она перемещает ракету вверх со скоростью три пикселя за такт.

Эта исправленная версия использует представление с двумя классами состояний:

```
; ЗР (запуск ракеты) -- это одно из двух значений:  
; -- "resting" (покой)  
; -- Неотрицательное Число  
; интерпретация: "resting" представляет ракету, стоящую на поверхности земли  
; число обозначает высоту, на которой находится ракета
```

В отличие от строки "resting" (в состоянии покоя), толкование чисел неоднозначно в своем понятии высоты местоположения:

1. Слово «высота» может обозначать расстояние между землей (нижним краем холста) и точкой отсчета, обозначающей, например, центр ракеты.
2. Слово «высота» может обозначать расстояние между верхним краем холста и точкой отсчета.

Обе интерпретации вполне пригодны. Но во втором случае используется обычное компьютерное значение слова «высота». То есть вторая интерпретация немного удобнее для функций, которые переводят состояние мира в изображение, и поэтому мы будем использовать именно эту интерпретацию.

Чтобы убедить вас в правильности этого выбора, в упражнении 57 ниже вам будет предложено выполнить упражнения из этого раздела с использованием первой интерпретации понятия «высота».

Упражнение 53. Рецепт проектирования мировых программ требует переводить информацию в данные и наоборот, чтобы гарантировать полное понимание определения данных. Лучше всего нарисовать какие-то мировые сценарии и представить их в виде данных и, наоборот, выбрать несколько примеров данных и нарисовать картинки, которые им соответствуют. Сделайте это для определения ЗР, включив примеры с использованием как минимум HEIGHT и 0. ■

На самом деле запуск ракеты начинается с обратного отсчета:

Задача. Спроектируйте программу, которая запускает ракету, когда пользователь нажимает клавишу пробела. После нажатия клавиши пробела программа начинает обратный отсчет и продолжает его в течение трех тактов, после чего начинает изображать взлетающую ракету. Ракета должна двигаться вверх со скоростью три пикселя за такт часов.

Следуя рецепту разработки программ, сначала определим константы:

```
(define HEIGHT 300) ; размеры в пикселях  
(define WIDTH 100)  
(define YDELTA 3)

(define BACKG (empty-scene WIDTH HEIGHT))
(define ROCKET (rectangle 5 30 "solid" "red"))

(define CENTER (/ (image-height ROCKET) 2))
```

Константы `WIDTH` и `HEIGHT` описывают размеры холста и сцены. Константа `YDELTA` описывает скорость взлета ракеты вдоль оси `Y`, как указано в постановке задачи. Константа `CENTER` – это вычисленный центр ракеты.

Затем переходим к определению данных. Эта версия задачи явно требует выделить три разных подкласса состояний:

```
; ЗР00 (запуск ракеты с обратным отсчетом) -- это одно из значений:  
; -- "resting"  
; -- Число в диапазоне от -3 до -1  
; -- НеотрицательноеЧисло  
; интерпретация: ракета стоит на поверхности земли, идет обратный отсчет,  
; число обозначает расстояние в пикселях от верхнего края холста  
; до ракеты (высота местоположения ракеты)
```

Второй, новый подкласс данных – три отрицательных числа – представляет состояние мира после нажатия клавиши пробела и до взлета ракеты.

Теперь добавим в список желаний функцию, отображающую состояние в виде изображений, а также функции обработки событий, которые могут нам понадобиться:

```
; ЗР00 -> Изображение  
; отображает состояние ракеты, покоящейся на земле или взлетающей  
(define (show x)  
  BACKG)  
  
; ЗР00 СобытиеКлавиатуры -> ЗР00  
; начинает обратный отсчет по нажатии клавиши пробела,  
; если ракета находилась в состоянии покоя  
(define (launch x ke)  
  x)  
  
; ЗР00 -> ЗР00  
; изменяет высоту ракеты на YDELTA,  
; если взлет начался  
(define (fly x)  
  x)
```

Создания этих сигнатур требует рецепт проектирования мировых программ, но выбор имен для данных и обработчиков событий остается за нами. Кроме того, мы специально выбрали имена, чтобы они соответствовали нашей постановке задачи.

Теперь используем рецепт проектирования и напишем законченные определения всех трех функций, начав с примеров для первой из них:

```
(check-expect  
  (show "resting")  
  (place-image ROCKET 10 HEIGHT BACKG))  
  
(check-expect  
  (show -2)  
  (place-image (text "-2" 20 "red")  
              10 (* 3/4 WIDTH))
```

```
(place-image ROCKET 10 HEIGHT BACKG))  
  
(check-expect  
  (show 53)  
  (place-image ROCKET 10 53 BACKG))
```

Как и прежде в этой главе, мы создали по одному тесту для каждого подкласса в определении данных. Первый тест проверяет состояние покоя, второй – состояние в середине обратного отсчета, а последний – ракету в полете. Кроме того, ожидаемые значения мы выражаем в виде выражений, создающих соответствующие изображения. Чтобы создать эти выражения, мы поэкспериментировали в области взаимодействий DrRacket; а как поступили бы вы?

При внимательном рассмотрении примеров легко заметить, что их создание также связано с выбором некоторых значений. На самом деле в постановке задачи нет ничего, что указывало бы, как именно отображается ракета перед запуском, но это естественно. Точно так же нет ничего, что указывало бы, как должны отображаться числа в процессе обратного отсчета, но это добавляет приятный штрих. Наконец, если вы решили упражнение 53, то знаете, что θ и HEIGHT имеют особое значение для третьего предложения в определении данных.

В общем случае интервалы заслуживают особого внимания при составлении примеров, то есть для каждого интервала желательно представить как минимум три примера: по одному с каждого конца и один внутри. Поскольку второй подкласс в ЗРОО представляет (конечный) интервал, а третий – полуоткрытый интервал, то есть смысл рассмотреть их конечные точки:

- совершенно очевидно, что (show -3) и (show -1) должны создавать точно такие же изображения, как и (show -2). В конце концов, ракета все еще стоит на поверхности земли, даже если числа обратного отсчета различаются;
- случай с (show HEIGHT) отличается. Согласно нашему соглашению, значение HEIGHT представляет состояние, когда ракета только что была запущена, хотя на рисунке ракета все еще стоит на земле. Основываясь на последнем тестовом примере, это можно выразить так:

```
(check-expect  
  (show HEIGHT)  
  (place-image ROCKET 10 HEIGHT BACKG))
```

Если теперь вычислить выражение «ожидаемого значения» в области взаимодействий DrRacket, то можно заметить, что ракета находится на полкорпуса под землей. Конечно, так быть не должно, а это означает, что нам нужно скорректировать этот и последний тестовый пример в наборе выше:

```
(check-expect  
  (show HEIGHT)
```

```
(place-image ROCKET 10 (- HEIGHT CENTER) BACKG))

(check-expect
  (show 53)
  (place-image ROCKET 10 (- 53 CENTER) BACKG))
```

- наконец, нужно определить ожидаемый результат для (`show 0`). Это простое, но показательное упражнение.

Следуя прецедентам в данной главе, `show` использует выражение `cond` для проверки трех условий:

```
(define (show x)
  (cond
    [(string? x) ...]
    [(<= -3 x -1) ...]
    [(>= x 0) ...]))
```

Каждое условие идентифицирует соответствующий подкласс: (`string?` `x`) выбирает первый подкласс, который состоит только из одного элемента – строки "resting"; ($\leq -3 \leq x \leq -1$) описывает второй подкласс данных; ($\geq x \geq 0$) выбирает любые неотрицательные числа.

Упражнение 54. Почему первое условие в функции `show` нельзя записать как (`string=? "resting" x`)? Сформулируйте точное условие, то есть логическое выражение, дающее `#true`, только когда `x` принадлежит первому подклассу ЗРОО. ■

Объединив примеры и приведенную выше заготовку функции `show`, легко создать полное определение:

```
(define (show x)
  (cond
    [(string? x)
     (place-image ROCKET 10 (- HEIGHT CENTER) BACKG)]
    [(<= -3 x -1)
     (place-image (text (number->string x) 20 "red")
                 10 (* 3/4 WIDTH)
                 (place-image ROCKET
                             10 (- HEIGHT CENTER)
                             BACKG))]
    [(>= x 0)
     (place-image ROCKET 10 (- x CENTER) BACKG)])))
```

Такой способ определения функций действительно очень эффективен и является важной составляющей полноценного подхода к проектированию, описанного в этой книге.

Упражнение 55. Еще раз взгляните на функцию `show`. Она содержит три выражения похожего вида:

```
| (place-image ROCKET 10 (- ... CENTER) BACKG)
```

Два выражения используются для рисования ракеты, стоящей на поверхности земли, и одно – для рисования взлетающей ракеты. Определите вспомогательную функцию, которая выполняет эту ра-

боту, и тем самым сократите количество строк в `show`. Какие преимущества это дает? Чтобы ответить на этот вопрос, можете повторно прочитать пролог. ■

Перейдем ко второй функции, которая обрабатывает события клавиатуры и определяет момент запуска ракеты. Вся необходимая информация у нас уже есть в заголовке, поэтому сразу же сформулируем примеры в виде тестов:

```
(check-expect (launch "resting" " ") -3)
(check-expect (launch "resting" "a") "resting")
(check-expect (launch -3 " ") -3)
(check-expect (launch -1 " ") -1)
(check-expect (launch 33 " ") 33)
(check-expect (launch 33 "a") 33)
```

Взглянув на эти шесть примеров, легко заметить, что первые два тестируют обработку событий клавиатуры, когда ракета находится на поверхности земли перед стартом, третий и четвертый – во время обратного отсчета, а последние два – во время полета, когда ракета уже находится в воздухе.

Прием с предварительной записью макета выражения `cond` хорошо зарекомендовал себя при проектировании функции `show`, поэтому воспользуемся им снова:

```
(define (launch x ke)
  (cond
    [(string? x) ...]
    [(<= -3 x -1) ...]
    [(>= x 0) ...]))
```

Оглядываясь на примеры, можно заметить, что состояние мира не меняется, если его значение относится ко второму или третьему подклассу данных ЗРМО. То есть в этих случаях `launch` просто должна возвращать свой аргумент `x`:

```
(define (launch x ke)
  (cond
    [(string? x) ...]
    [(<= -3 x -1) x]
    [(>= x 0) x]))
```

Наконец, первый пример иллюстрирует точный случай, когда функция `launch` создает новое состояние мира:

```
(define (launch x ke)
  (cond
    [(string? x) (if (string=? " " ke) -3 x)]
    [(<= -3 x -1) x]
    [(>= x 0) x]))
```

В частности, когда мир находится в состоянии "resting" и пользователь нажимает клавишу пробела, то функция запускает обратный отсчет с -3.

Скопируйте этот код в область определений DrRacket и убедитесь, что все определения, приведенные выше, работают нормально. Теперь можно добавить функцию, запускающую программу:

```
; 3P00 -> 3P00
(define (main1 s)
  (big-bang s
    [to-draw show]
    [on-key launch]))
```

Эта функция **не** определяет, что должно происходить с каждым тиком часов; в конце концов, мы еще не спроектировали функцию `fly`. Тем не менее `main1` уже позволяет запустить эту неполную версию программы и проверить, как запускается обратный отсчет. Какой аргумент, по вашему мнению, может передаваться в вызов `main1`?

Листинг 17. Запуск обратного отсчета и взлет

```
; 3P00 -> 3P00
; изменяет высоту ракеты на YDELTA,
; если взлет начался

(check-expect (fly "resting") "resting")
(check-expect (fly -3) -2)
(check-expect (fly -2) -1)
(check-expect (fly -1) HEIGHT)
(check-expect (fly 10) (- 10 YDELTA))
(check-expect (fly 22) (- 22 YDELTA))

(define (fly x)
  (cond
    [(string? x) x]
    [(<= -3 x -1) (if (= x -1) HEIGHT (+ x 1))]
    [(>= x 0) (- x YDELTA)]))
```

Функция `fly` – обработчик тиков часов – проектируется точно так же, как две предыдущие функции. В листинге 17 показан результат процесса проектирования. И снова основная идея заключается в охвате пространства возможных входных данных набором тестовых примеров. Особенно это относится к двум интервалам. Эти примеры гарантируют правильную работу обратного отсчета и переход к взлету ракеты.

Упражнение 56. Определите функцию `main2`, чтобы можно было запустить ракету и понаблюдать, как она взлетает. Загляните в документацию с описанием `op-tick` и узнайте продолжительность одного тика часов и как ее изменить.

Если понаблюдать за взлетом ракеты до конца, то можно заметить, что как только ракета достигает верхнего края сцены, происходит нечто любопытное. Попробуйте объяснить это. Добавьте в `main2` предложение `stop-when`, чтобы анимация взлета прекращалась после того, как ракета скроется из поля зрения. ■

Решив упражнение 56, вы получите законченную программу, но она ведет себя немного странно. Опытные программисты скажут вам,

что использование отрицательных чисел для обозначения этапа обратного отсчета – слишком «хрупкое» решение. В следующей главе мы познакомимся с дополнительными средствами, позволяющими получить более надежное определение данных и решить эту проблему. Но прежде, в следующем разделе, мы посмотрим, как проектировать программы, которые принимают данные, описываемые методом детализации.

Упражнение 57. Напомним, что понятие «высота» имеет два возможных толкования, и это поставило нас перед выбором. Теперь, когда вы выполнили все упражнения в этом разделе, решите их еще раз, используя первое толкование «высоты». Сравните решения. ■

4.6. Проектирование с использованием детализации

В предыдущих трех разделах говорилось, что при проектировании функций может и должна использоваться организация, заданная в определениях данных. В частности, если определение данных выделяет определенные экземпляры или диапазоны данных, то примеры и организация функции должны отражать это.

В этом разделе мы уточним рецепт проектирования из раздела 3.4, чтобы вы могли действовать системно, столкнувшись с функциями, которые принимают на входе детализации, включая перечисления и интервалы. Для наглядности мы проиллюстрируем шесть этапов проектирования упрощенным примером:

Задача. В стране Налогия решили ввести трехступенчатый налог с продаж, чтобы сократить дефицит бюджета. Недорогие товары, со стоимостью менее 1000 долларов, не облагаются налогом. Предметы роскоши, со стоимостью выше 10 000 долларов, облагаются налогом в восемь процентов (8,00 %). Товары из промежуточной ценовой категории облагаются налогом в пять процентов (5,00 %).

Спроектируйте функцию для кассового аппарата, которая вычисляет размер налога, исходя из цены товара.

Вспоминайте эту постановку задачи, когда будете рассматривать пункты рецепта проектирования:

1. Когда в постановке задачи выделяются разные классы входной информации, необходимо со всем тщанием отнести к формулировке определений данных.

В определениях данных должны использоваться отдельные *пункты* для каждого подкласса, а иногда и для отдельных экземпляров данных. Каждое предложение должно определять представление данных для конкретного подкласса информации. При этом каждый подкласс данных должен отличаться от

любых других классов, чтобы можно было строить функцию на основе анализа непересекающихся случаев.

В данном примере задача связана с ценами и налогами, которые обычно представлены положительными числами. Также здесь четко различаются три диапазона:

; Цена делится на три диапазона:
; --- от 0 до 1000
; --- от 1000 до 10000
; --- от 10000 и выше.

; интерпретация: цена единицы товара

Разработчики, создающие приложения для реального мира, не используют обычные числа для обозначения денежных сумм. См. интермеццо 4, где описываются некоторые проблемы, связанные с числами.

Вам понятно, как эти диапазоны связаны с постановкой задачи?

2. Сигнатура функции, заголовок и назначение формулируются так же, как и раньше.

Вот отправная точка для нашего рабочего примера:

; Цена -> Число
; вычисляет сумму налога для р

(define (sales-tax p) 0)

3. При определении тестовых примеров необходимо выбрать хотя бы один образец из каждого подкласса в определении данных. Кроме того, если подкласс является конечным диапазоном, желательно также определить примеры с границами диапазона и хотя бы одним значением внутри диапазона.

В нашем определении данных упоминаются три разных интервала, поэтому определим примеры для всех границ и по одному примеру для значений внутри каждого интервала, вычисляющие сумму налога в следующих случаях: 0, 537, 1000, 1282, 10000 и 12017.

Стоп! Попробуйте вручную вычислить налог для каждой из этих цен.

Вот наша первая попытка с округленными суммами налогов:

0 537 1000 1282 10000 12017
0 0 ??? 64 ??? 961

Знаки вопроса указывают на некоторую расплывчатость в формулировках, использованных в постановке задачи: «со стоимостью менее 1000 долларов» и «со стоимостью выше 10 000 долларов». Программист может решить, что эти слова означают «строго меньше» или «строго больше», но законодатели могли иметь в виду «меньше или равно» или «больше или равно» соответственно. Будучи скептически настроенными, предположим, что законодатели штата Таксленд хотели собрать больше налогов, поэтому ставка налога для цены 1000 долларов составляет 5 %, а ставка для цены 10 000 долларов – 8 %. Но, вообеще говоря, программист, работающий в налоговой инспекции,

должен уточнить размер ставки для этих случаев у специалиста по налоговому праву.

Теперь, выяснив, как интерпретировать границы интервалов в данной задаче, можно уточнить определение данных. Мы на-деемся, что вы сможете сделать это самостоятельно.

Прежде чем продолжить, напишем тестовые примеры:

```
(check-expect (sales-tax 537) 0)
(check-expect (sales-tax 1000) (* 0.05 1000))
(check-expect (sales-tax 12017) (* 0.08 12017))
```

Обратите внимание. Вместо ожидаемых результатов мы использовали выражения, вычисляющие эти ожидаемые результаты. Это упростит определение функции.

Стоп! Допишите остальные тестовые примеры. Подумайте, почему иногда тестовых примеров должно быть больше, чем подклассов в определении данных.

4. Самое заметное новшество – макет условного выражения. Вообще говоря,

макет условного выражения отражает организацию подклассов с помощью cond.

Это означает, во-первых, что тело функции должно быть условным выражением, количество условий в котором совпадает с количеством подклассов в определении данных. Если в определении данных упоминаются три подкласса данных, то выражение `cond` должно содержать три условия; если имеется семнадцать подклассов, то выражение `cond` должно содержать семнадцать условий. Во-вторых, для каждого условия `cond` необходимо сформулировать одно выражение, вычисляющее результат. Каждое выражение должно включать параметр функции и определять один из подклассов в определении данных:

```
(define (sales-tax p)
  (cond
    [(and (<= 0 p) (< p 1000)) ...]
    [(and (<= 1000 p) (< p 10000)) ...]
    [(>= p 10000) ...]))
```

5. Закончив с макетом, можно приступать к определению функции. Учитывая, что тело функции уже содержит заготовку выражения `cond`, естественно начать с заполнения различных строк в `cond`. Для каждой строки в `cond` можно предположить, что входной параметр удовлетворяет условию, и поэтому допустимо использовать за основу соответствующие тестовые примеры. Чтобы сформулировать соответствующее выражение результата, запишите вычисление для данного примера как выражение, которое использует параметр функции. При работе с каждой

отдельной строкой игнорируйте все другие возможные типы входных данных; о них позаботятся другие условия.

```
(define (sales-tax p)
  (cond
    [(and (<= 0 p) (< p 1000)) 0]
    [(and (<= 1000 p) (< p 10000)) (* 0.05 p)]
    [(>= p 10000) (* 0.08 p)]))
```

6. Наконец, запустите тесты и убедитесь, что они охватывают все условия в `cond`.

Что вы будете делать, если один из тестовых примеров выполнится с ошибкой? Прочтите конец раздела 3.1, посвященный ошибкам тестирования. ■

Упражнение 58. Добавьте определения констант, отделяющих диапазон низких цен от цен на предметы роскоши и другие товары, чтобы законодатели в Таксленде могли с легкостью поднять налоги, если того пожелают. ■

4.7. Мирь с конечными состояниями

С новыми знаниями в области проектирования, полученными в этой главе, вы сможете разработать полноценную модель работы светофора. Когда горит зеленый свет и пора останавливать движение, то светофор последовательно включает сначала желтый сигнал, а затем красный. Когда горит красный свет и пора возобновить движение, то светофор просто включает зеленый сигнал.

Вышесказанное иллюстрирует *диаграмма переходов* между состояниями на рис. 5. Такая диаграмма состоит из состояний и стрелок, соединяющих их. Каждое состояние изображено в виде светофора с одним конкретным сигналом: красным, желтым или зеленым. Каждая стрелка показывает, как может меняться состояние мира, из какого состояния и в какое другое состояние он может *перейти*. В примере диаграммы показаны три стрелки, потому что существует только три способа изменения состояния светофора. Надписи на стрелках указывают причину изменения состояния; светофор переходит из одного состояния в другое с течением времени.

Во многих случаях диаграммы переходов содержат конечное число состояний и стрелок. В информатике такие диаграммы называют *конечными автоматами* (Finite State Machine, FSM). Несмотря на свою простоту, конечные автоматы играют важную роль в информатике.

Чтобы создать мировую программу для конечного автомата, нужно сначала выбрать представление данных для возможных «состояний мира», которое, согласно разделу 3.6, представляет те аспекты мира, которые могут некоторым образом измениться. В случае со светофором меняется цвет сигнала, то есть номер включенной лампы. Размер ламп, их расположение (горизонтальное или вертикальное) и другие

аспекты не меняются. Поскольку существует только три состояния, вновь используем определение данных Светофор, сформулированное нами ранее.

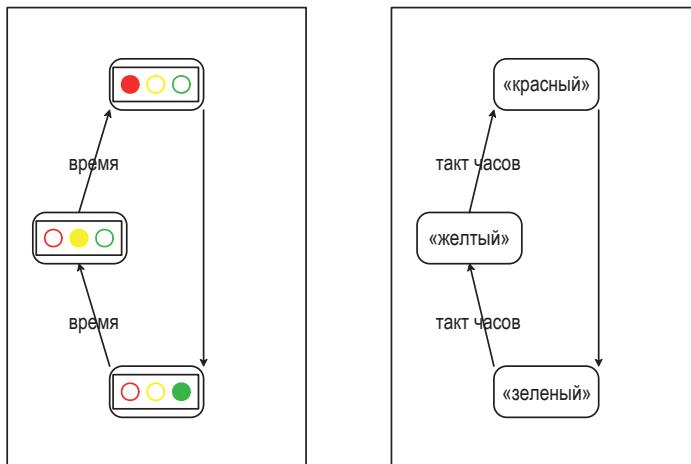


Рис. 5. Как действует светофор

Справа на рис. 5 изображена схематическая интерпретация определения данных Светофор. Как и диаграмма переходов слева на рис. 5, она включает три состояния, расположенных таким образом, что каждый экземпляр данных четко определяет конкретную конфигурацию. Кроме того, стрелки теперь подписаны словами «такт часов», подсказывая, что в качестве триггера, изменяющего состояние светофора, наша мировая программа использует течение времени. Если бы мы решили смоделировать переключение светофора вручную, то могли бы обозначить переходы соответствующими клавишами на клавиатуре.

Теперь, зная, как представлять разные состояния мира, как переходить от одного состояния к другому и что состояние должно меняться с каждым тактом часов, мы можем написать сигнатуры, назначения и заготовки двух функций, которые необходимо спроектировать:

```
; Светофор -> Светофор
; выдает следующее состояние, опираясь на текущее состояние cs
(define (tl-next cs) cs)

; Светофор -> Изображение
; преобразует текущее состояние cs в изображение
(define (tl-render current-state)
  (empty-scene 90 30))
```

В предыдущих разделах для функций, преобразующих состояние мира в изображение и обрабатывающих такты часов, использовались имена `render` и `next`. Здесь мы добавили префикс «`tl-`» (от «`traffic`

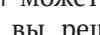
light» – светофор) к этим именам, чтобы подсказать, к какому миру принадлежат функции. Поскольку вы уже писали похожие функции, оставляем их создание вам как самостоятельное упражнение.

Упражнение 59. Завершите проектирование мировой программы, моделирующей светофор. Вот главная функция для этой программы:

```
| ; Светофор -> Светофор
| ; моделирует работу светофора,
| ; автоматически переключающегося с течением времени
(define (traffic-light-simulation initial-state)
  (big-bang initial-state
    [to-draw tl-render]
    [on-tick tl-next 1]))
```

Аргументом функции является начальное состояние мира для выражения `big-bang`, которое DrRacket преобразует в изображение с помощью `tl-render`. События хода системных часов обрабатываются функцией `tl-next`. Также обратите внимание, что здесь явно настраивается продолжительность одного такта часов равной одной секунде.

Завершите проектирование `tl-render` и `tl-next` и скопируйте их в область определений DrRacket. Вот несколько тестовых примеров для `tl-render`:

```
| (check-expect (tl-render "red") 
| 
| 
| (check-expect (tl-render "yellow") 
| 
| 
```

Ваша функция `tl-render` может использовать эти изображения непосредственно. Но если вы решите конструировать изображения с помощью функций из библиотеки `2htdp/image`, то спроектируйте вспомогательную функцию для создания изображения одноцветной лампочки. Затем используйте функцию `place-image` для размещения лампочек в сцене. ■

Упражнение 60. Как вариант в представлении данных для программы, моделирующей работу светофора, можно использовать числа вместо строк:

```
| ; Светофор -- это одно из трех значений:
| ; -- 0 интерпретация: красный сигнал
| ; -- 1 интерпретация: зеленый сигнал
| ; -- 2 интерпретация: желтый сигнал
```

Это представление значительно упрощает определение `tl-next`:

```
| ; Ч-Светофор -> Ч-Светофор
| ; возвращает следующее состояние, опираясь на текущее состояние cs
(define (tl-next-numeric cs) (modulo (+ cs 1) 3))
```

Сформулируйте тесты для `tl-next-numeric`.

Какая функция точнее передает свое предназначение, `tl-next` или `tl-next-numeric`? Объясните почему. ■

Упражнение 61. Как отмечалось в разделе 3.4, программы должны определять константы и использовать имена констант вместо фактических значений. Определение данных, описывающее состояния светофора, тоже должно использовать константы:

```
(define RED 0)
(define GREEN 1)
(define YELLOW 2)

; С-Светофор -- это одно из трех значений:
; -- RED
; -- GREEN
; -- YELLOW
```

При выборе удачных имен определение данных не требует описания правил их интерпретации.

Листинг 18. Символический светофор

```
; С-Светофор -> С-Светофор
; выдает следующее состояние, опираясь на текущее состояние cs
(check-expect (tl-next-... RED) YELLOW)
(check-expect (tl-next-... YELLOW) GREEN)

(define (tl-next-numeric cs)
  (modulo (+ cs 1) 3))

(define (tl-next-symbolic cs)
  (cond
    [(equal? cs RED) GREEN]
    [(equal? cs GREEN) YELLOW]
    [(equal? cs YELLOW) RED]))
```

В листинге 18 показаны две функции, изменяющие состояние светофора. Какая из них спроектирована правильно, в соответствии с рецептом проектирования? Какая из них продолжит работать после изменения констант, как показано ниже:

```
(define RED "red")
(define GREEN "green")
(define YELLOW "yellow")
```

Это поможет вам ответить на вопросы?

ПРИМЕЧАНИЕ. Функция `equal?`, использованная в листинге 18, сравнивает два произвольных значения, независимо от их типов. Равенство – сложная тема в мире программирования. **КОНЕЦ** ■

Вот еще одна задача с конечным числом состояний, добавляющая несколько дополнительных сложностей:

Постановка задачи. Спроектируйте мировую программу, которая моделирует работу двери с автоматическим доводчиком. Если такая дверь заперта, ее можно открыть ключом. Незапертая дверь автоматически закрывается, но ее можно толкнуть

Эту форму определения данных использовал бы опытный проектировщик.

и распахнуть. Как только человек пройдет через дверь и отпустит ее, она автоматически закроется, и после этого ее можно снова запереть.

Чтобы выделить основные элементы, нарисуем диаграмму переходов (см. слева на рис. 6). Подобно светофору, дверь имеет три состояния: заперта, закрыта и открыта. При отпирании и запирании дверь переходит из состояния «заперта» в состояние «закрыта» и наоборот. Чтобы открыть незапертую дверь, ее нужно **толкнуть**. Оставшийся переход отличается от других, потому что не требует никаких действий со стороны кого-либо или чего-либо. Дверь сама закрывается со временем. Соответствующая стрелка перехода подписана словом «время», чтобы подчеркнуть это.

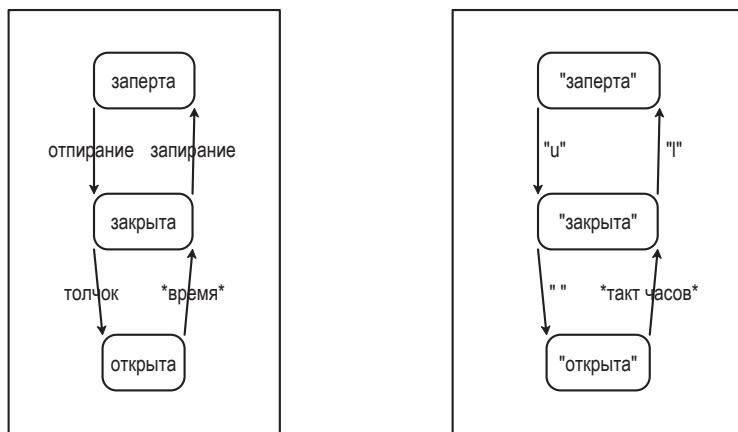


Рис. 6. Диаграмма переходов для автоматически закрывающейся двери

Следуя рецепту, начнем с перевода трех возможных состояний в данные:

```

(define LOCKED "locked")
(define CLOSED "closed")
(define OPEN "open")

; Состояние двери -- это одно из значений:
; -- LOCKED (заперта)
; -- CLOSED (закрыта)
; -- OPEN (открыта)
  
```

Вспомним также урок из упражнения 61, а именно что лучше определить символические константы и формулировать определения данных в терминах этих констант.

Следующий шаг в рецепте проектирования мировой программы требует преобразовать действия в предметной области – стрелки на диаграмме слева – во взаимодействия с компьютером, поддержку которых можно получить с помощью библиотеки *2htdp/universe*.

Наша диаграмма состояний и переходов двери, в частности стрелка от состояния «открыта» к состоянию «закрыта», предполагает использование системных часов. Для реализации взаимодействий, обозначенных другими стрелками, можно использовать нажатия клавиш или щелчки мышью. Давайте используем нажатия клавиш: «u» – для отпирания двери, «l» – для запирания и клавишу пробела – для имитации толчка, открывающего дверь. На диаграмме справа на рис. 6 эти варианты представлены графически; она транслирует диаграмму конечного автомата из мира информации в мир данных.

Решив использовать время и нажатия клавиш для обозначения действий, мы должны спроектировать функции, отображающие текущее состояние мира, представленное как СостояниеДвери, и преобразующие его в следующее состояние мира. Разумеется, их следует внести в список желаний:

- door-closer, закрывает открытую дверь по истечении одного такта часов;
- door-action, выполняет действие в зависимости от нажатой клавиши;
- door-render, преобразует текущее состояние в изображение.

Стоп! Сформулируйте соответствующие сигнатуры.

Начнем с door-closer. Поскольку door-closer играет роль обработчика событий часов, его сигнатура определяется нашим выбором представления СостояниеДвери в качестве коллекции набора состояний мира:

```
; СостояниеДвери -> СостояниеДвери
; закрывает открытую дверь по истечении одного такта часов
(define (door-closer state-of-door) state-of-door)
```

Придумывать примеры не составляет никакого труда, когда мир может находиться только в одном из трех состояний. Здесь, как и в некоторых математических примерах выше, для выражения основной идеи мы использовали таблицу:

исходное состояние	желаемое состояние
LOCKED	LOCKED
CLOSED	CLOSED
OPEN	CLOSED

Стоп! Выразите эти примеры в виде тестов на языке BSL.

Чтобы получить макет, нам понадобится условное выражение с тремя условиями:

```
(define (door-closer state-of-door)
  (cond
    [(string=? LOCKED state-of-door) ...]
    [(string=? CLOSED state-of-door) ...]
    [(string=? OPEN state-of-door) ...]))
```

а превратить этот макет в определение функции нам помогут примеры:

```
(define (door-closer state-of-door)
  (cond
    [(string=? LOCKED state-of-door) LOCKED]
    [(string=? CLOSED state-of-door) CLOSED]
    [(string=? OPEN state-of-door) CLOSED]))
```

Не забудьте запустить свои тесты.

Вторая функция, *door-action*, обрабатывает остальные три стрелки на диаграмме. Функции, обслуживающие события клавиатуры, используют два элемента информации: состояние мира и описание события. То есть сигнатура этой функции выглядит так:

```
; Состояниедвери Событиеклавиатуры -> Состояниедвери
; выполняет определенное действие
; в зависимости от события клавиатуры k и состояния s
(define (door-action s k)
  s)
```

И снова представим примеры в табличной форме:

исходное состояние	событие клавиатуры	желаемое состояние
LOCKED	"u"	CLOSED
CLOSED	"l"	LOCKED
CLOSED	" "	OPEN
OPEN	-	OPEN

Примеры объединяют информацию из нашего рисунка с выбранными нами соответствиями между действиями и событиями клавиатуры. В отличие от таблицы с примерами для светофора, эта таблица неполная. Подумайте, какие еще примеры можно было бы привести, а затем попробуйте понять, почему нашей таблицы достаточно.

Опираясь на эти примеры, легко сформулировать законченную реализацию:

```
(check-expect (door-action LOCKED "u") CLOSED)
(check-expect (door-action CLOSED "l") LOCKED)
(check-expect (door-action CLOSED " ") OPEN)
(check-expect (door-action OPEN "a") OPEN)
(check-expect (door-action CLOSED "a") CLOSED)

(define (door-action s k)
  (cond
    [(and (string=? LOCKED s) (string=? "u" k))
     CLOSED]
    [(and (string=? CLOSED s) (string=? "l" k))
     LOCKED]
    [(and (string=? CLOSED s) (string=? " " k))
     OPEN]
    [else s]))
```

Обратите внимание на использование оператора `and` для объединения двух условий: одно условие проверяет текущее состояние двери, а другое – событие клавиатуры.

Наконец, нам нужно отобразить состояние мира в виде сцены:

```
; Состояние двери -> Изображение
; преобразует состояние s в изображение с текстом
(check-expect (door-render CLOSED)
               (text CLOSED 40 "red"))
(define (door-render s)
  (text s 40 "red"))
```

Эта упрощенная функция создает изображение с крупным текстом. А вот так программа использует все эти функции:

```
; Состояние двери -> Состояние двери
; имитирует дверь с автоматическим доводчиком
(define (door-simulation initial-state)
  (big-bang initial-state
            [on-tick door-closer]
            [on-key door-action]
            [to-draw door-render]))
```

Теперь пришло время собрать все вместе и запустить в DrRacket, чтобы увидеть, правильно ли работает наша программа.

Упражнение 62. Во время работы программы состояние «открыта» практически не видно. Измените программу так, чтобы продолжительность такта часов составляла три секунды. Запустите программу и понаблюдайте за ней. ■

5. Добавляем структуру

Предположим, что мы решили спроектировать мировую программу, имитирующую мяч, прыгающий вверх и вниз между полом и потолком воображаемой идеальной комнаты. Предположим, что мяч перемещается с постоянной скоростью – два пикселя за один такт часов.

Если следовать рецепту проектирования, то первая наша цель – определить представление данных, меняющихся с течением времени. В данном случае со временем меняются местоположение и направление движения мяча, но это два значения, а *big-bang* может контролировать только одно значение. Возникает вопрос: как в один блок данных уместить два изменяющихся блока информации?

Математикам известны приемы «объединения» двух чисел в одно, из которого потом можно обратно получить исходные числа. Программисты считают такие уловки вредными, потому что они скрывают истинные цели программы.

Вот еще один сценарий, где возникает тот же вопрос. Мобильный телефон – это несколько миллионов строк кода, заключенных в пластик. Кроме всего прочего, он управляет вашими контактами. У каждого контакта есть имя, номер телефона, адрес электронной почты и, возможно, другая информация. Когда приходится работать со множеством контактов, каждый отдельный контакт лучше всего представить в виде единого экземпляра данных; в противном случае составные части могут случайно перепутаться.

Для решения подобных проблем каждый язык программирования предоставляет некоторый механизм объединения нескольких значений в единый *составной фрагмент данных* и извлечения составляющих значений, когда это необходимо. Эта глава знакомит с подобным механизмом, имеющимся в языке BSL, так называемой поддержкой определения *структурных типов*, и рассказывает, как проектировать программы, работающие с составными данными.

5.1. От позиций к структурам posn

Местоположение на холсте однозначно определяется двумя элементами данных: расстоянием от левого края и расстоянием от верхнего края. Первое называется *координатой x*, а второе – *координатой y*.

Среда программирования DrRacket, которая, по сути, является программой на BSL, представляет такие позиции в виде структуры *posn*. Структура *posn* объединяет два числа в одно значение. Создать экземпляр структуры *posn* можно с помощью операции *make-posn*, которая принимает два числа и создает экземпляр *posn*. Например, вот выражение, которое создает экземпляр структуры *posn*, в котором координата *x* имеет значение 3, а координата *y* – значение 4:

```
| (make-posn 3 4)
```

Структура `posn` – это такой же тип данных, как число, логическое значение или строка. В частности, элементарные операции и функции могут принимать и создавать структуры. При создании экземпляра структуры `posn` программа может дать ему имя:

```
| (define one-posn (make-posn 8 6))
```

Стоп! Опишите `one-posn` в терминах координат.

Прежде чем продолжить, познакомимся с правилами вычислений с использованием структур `posn`. Это поможет нам создавать функции, обрабатывающие структуры `posn`, и понимать, что они вычисляют.

5.2. Вычисления со структурами `posn`

Особенности функций и законы, связанные с ними, хорошо знакомы из начальной алгебры, но структуры `posn` – это, кажется, что-то новенькое. С другой стороны, экземпляры `posn` должны выглядеть как декартовы точки на плоскости, с которыми вы, возможно, сталкивались раньше.

Выбор отдельных координат декартовой точки – тоже знакомый процесс. Например, если учитель скажет вам: «Взгляните на график на рис. 7 и скажите, что такое p_x и p_y », – то, скорее всего, вы ответите: «31 и 26», – потому что знаете, что должны прочитать значения в точках пересечения с осями координат вертикальной и горизонтальной линий, исходящих из p .

Спасибо Нилу Торонто
(Neil Toronto)
за библиотеку `plot`.

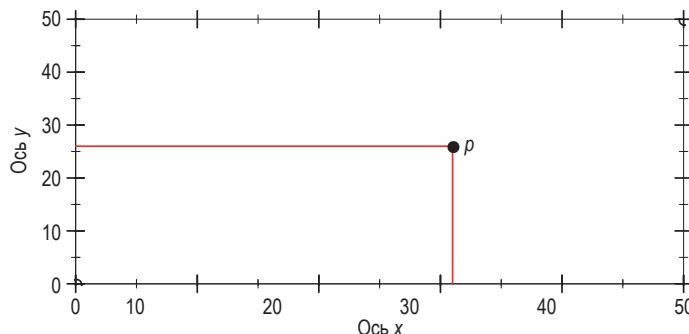


Рис. 7. Декартова точка

Вот как эту идею выразить на BSL. Предположим, вы ввели код

```
| (define p (make-posn 31 26))
```

в область определений и щелкнули на кнопке **RUN** (Выполнить), а затем произвели следующие взаимодействия:

```
| > (posn-x p)
31
| > (posn-y p)
26
```

Определение `p` напоминает обозначение точки на декартовой плоскости, а ссылки `posn-x` и `posn-y` подобны индексам при `p`: p_x и p_y .

С вычислительной точки зрения для структур `posn` имеют силу два тождества:

```
(posn-x (make-posn x0 y0)) == x0
(posn-y (make-posn x0 y0)) == y0
```

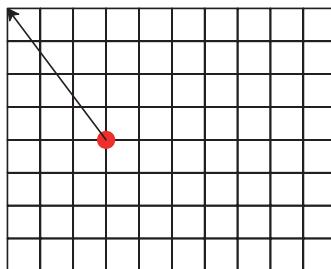
DrRacket использует эти тождества во время вычислений. Вот пример вычислений с использованием структур `posn`:

```
(posn-x p)
== ; DrRacket замещает p выражением (make-posn 31 26)
(posn-x (make-posn 31 26))
== ; DrRacket использует тождество для posn-x
31
```

Стоп! Подтвердите верность второго взаимодействия, выполнив вычисления вручную. Также используйте движок пошаговых вычислений в DrRacket для перепроверки.

5.3. Программирование с `posn`

Теперь рассмотрим порядок проектирования функции, которая вычисляет расстояние от некоторой точки на холсте до начала координат:



Как показывает этот рисунок, под «расстоянием» в данном случае подразумевается длина прямого отрезка, соединяющего точку с верхним левым углом холста.

Вот описание назначения и заголовок:

```
; вычисляет расстояние от точки ap до начала координат
(define (distance-to-0 ap)
  0)
```

Главная особенность функции `distance-to-0` состоит в том, что она принимает одно значение, структуру `posn`, и возвращает одно значение – расстояние от заданной точки до начала координат.

Чтобы составить примеры, нужно знать, как вычислить это расстояние. Для точек со значением 0 в одной из координат результатом будет другая координата:

```
(check-expect (distance-to-0 (make-posn 0 5)) 5)
(check-expect (distance-to-0 (make-posn 7 0)) 7)
```

Для всех других случаев можно попробовать вывести формулу самостоятельно или вспомнить школьный курс геометрии. Эта формула относится к предметной области, которую вы можете хорошо знать, но если это не так, то мы подскажем вам эту формулу; в конце концов, эта предметная область не является информатикой. Итак, вот как выглядит формула вычисления расстояния (x, y):

$$\sqrt{x^2 + y^2}.$$

Используя эту формулу, легко можно составить еще несколько примеров применения функции:

```
(check-expect (distance-to-0 (make-posn 3 4)) 5)
(check-expect (distance-to-0 (make-posn 8 6)) 10)
(check-expect (distance-to-0 (make-posn 5 12)) 13)
```

Мы специально подобрали такие примеры, чтобы вам было проще проверить их результаты. Но подобная простота характерна не для всех структур `posn`.

Стоп! Подставьте координаты x и y из примеров в формулу. Проверьте вручную верность ожидаемых результатов во всех пяти примерах.

Теперь можно перейти к определению функции. Как показывают примеры, при проектировании `distance-to-0` не требуется различать разные ситуации; можно просто вычислить расстояние от точки с координатами x и y внутри данной структуры `posn`. Но функция должна каким-то образом получить эти координаты из структуры `posn`. Для этого можно использовать элементарные функции `posn-x` и `posn-y`. В частности, функция `distance-to-0` должна вычислить выражения `(posn-x ap)` и `(posn-y ap)`, потому что `ap` – это имя заданной структуры `posn`:

```
(define (distance-to-0 ap)
  (... (posn-x ap) ...
        ... (posn-y ap) ...))
```

Имея макет и примеры, определить реализацию функции не составит большого труда:

```
(define (distance-to-0 ap)
  (sqrt
    (+ (sqr (posn-x ap))
        (sqr (posn-y ap))))))
```

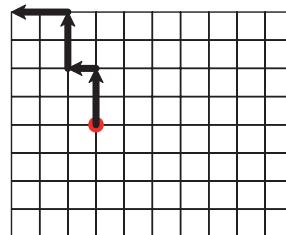
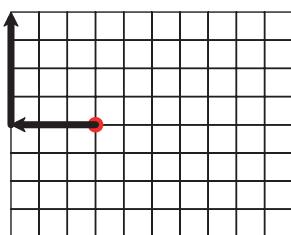
Функция вычисляет квадраты значений выражений `(posn-x ap)` и `(posn-y ap)`, представляющих координаты x и y , находит их сумму и извлекает из суммы квадратный корень. В DrRacket можно быстро убедиться, что наша новая функция возвращает правильные результаты для всех наших примеров.

Упражнение 63. Вычислите вручную следующие выражения:

- (`(distance-to-0 (make-posn 3 4))`);
- (`(distance-to-0 (make-posn 6 (* 2 4)))`);
- (`(+ (distance-to-0 (make-posn 12 5)) 10)`).

Запишите все шаги, предположив, что вычисление `sqr` – это один шаг. Сравните записанные шаги с результатами вычислений с помощью движка пошаговых вычислений в DrRacket. ■

Упражнение 64. Манхэттенским расстоянием от точки до начала координат называют длину пути, проложенного по прямоугольной сетке улиц в Манхэттене. Вот пара примеров:



Слева показан «прямой» способ прокладки пути, когда сначала путь прокладывается максимально влево, а затем вверх, насколько необходимо. Для сравнения справа показан способ «случайного блуждания», когда путь прокладывается на несколько кварталов влево, затем на несколько кварталов вверх и т. д., до достижения пункта назначения, в данном случае начало координат.

Стоп! Имеет ли значение выбор того или иного способа прокладки пути?

Спроектируйте функцию `manhattan-distance`, которая измеряет манхэттенское расстояние от данной точки до начала координат. ■

5.4. Определение структурных типов

В отличие от чисел или логических значений, структуры, такие как `posn`, обычно не являются частью языка программирования. Язык предоставляет только механизм определения структур, а все остальное оставляется на усмотрение программиста. То же верно и для BSL.

Определение структурного типа – это еще одна форма определений, помимо определений констант и функций. Вот как создатель DrRacket определил структуру `posn` на BSL:

Использование квадратных скобок в определении структуры не является обязательным требованием, и здесь мы использовали их исключительно для удобства, чтобы отделить имена полей. Квадратные скобки можно заменить круглыми скобками.

```
| (define-struct posn [x y])
```

В общем виде определение структуры выглядит так:

```
| (define-struct ИмяСтруктуры [Имя поля ...])
```

Ключевое слово `define-struct` сообщает, что определяется новый тип структуры. За ним следует имя структуры. Третья часть в определении структуры – это последовательность имен *полей*, заключенная в скобки.

Определение структуры фактически определяет функции. Но, в отличие от обычного определения функции, **определение структуры определяет сразу несколько функций**. В частности, определяются три вида функций:

- один *конструктор* – функция, которая создает экземпляры структуры. Конструктор принимает столько значений, сколько полей имеется в структуре; как уже упоминалось, *структурой* называют экземпляр структуры. Фраза *тип структуры* – это общее название набора всех возможных экземпляров;
- по одному *селектору* для каждого поля, извлекающему значение поля из экземпляра структуры;
- один *предикат структуры*, который, как и обычные предикаты, отличает экземпляры от всех других типов значений.

Программа может использовать их, как если бы они были обычными функциями.

Любопытно отметить, что определение типа структуры автоматически создает имена для множества новых операций. Так конструктор получает имя структуры с префиксом «`make-`», а селекторы – имя структуры с окончаниями, соответствующими именам полей. Наконец, предикат – это просто имя структуры со знаком вопроса «`?`», который произносится как «да?» при чтении вслух.

Это соглашение об именах выглядит сложным и, возможно, даже вносит некоторую путаницу. Но, немного попрактиковавшись, вы быстро освоите его. Оно также объясняет функции, поддерживаемые структурами `posn`: `make-posn` – конструктор, `posn-x` и `posn-y` – селекторы. Мы пока не сталкивались с `posn?`, тем не менее теперь мы знаем, что эта функция существует; назначение предикатов мы подробно рассмотрим в следующей главе.

Упражнение 65. Взгляните на следующие определения типов структур:

- `(define-struct movie [title producer year]);`
- `(define-struct person [name hair eyes phone]);`
- `(define-struct pet [name number]);`
- `(define-struct CD [artist title price]);`
- `(define-struct sweater [material size producer]).`

Выпишите имена функций (конструкторов, селекторов и предикатов), которые создаются одновременно с ними. ■

Но давайте оставим структуры `posn` и посмотрим на определение структуры, которое мы могли бы использовать для отслеживания контактов, например, в вашем сотовом телефоне:

```
| (define-struct entry [name phone email])
```

Вот имена функций, которые создаются этим определением:

- `make-entry`, принимает три значения и создает экземпляр контакта `entry`;
- `entry-name`, `entry-phone` и `entry-email`, принимают один экземпляр `entry` и возвращают значение соответствующего поля;
- `entry?`, предикат.

Поскольку каждый экземпляр `entry` объединяет три значения, выражение

```
| (make-entry "Al Abe" "666-7771" "lee@x.me")
```

создаст экземпляр структуры `entry` со строкой "Al Abe" в поле `name`, строкой "666-7771" в поле `phone` и строкой "lee@x.me" в поле `email`.

Упражнение 66. Вернитесь к определениям структур в упражнении 65 и попробуйте предположить, какие значения могли бы храниться в их полях. Затем создайте хотя бы по одному экземпляру для каждого определения структуры. ■

Каждое определение структуры создает новый тип структур, отличный от всех остальных. Эта выразительность необходима программистам, чтобы передать **назначение** структуры с помощью ее имени. Каждый раз, когда структура создается, проверяется или из нее извлекаются поля, текст программы явно будет напоминать читателю об этом назначении. Если бы программисты не думали о том, что кто-то будет читать их код в будущем, то они могли бы использовать одно определение структуры для структур с одним полем, другое для структур с двумя полями, третье для структур с тремя и т. д.

А теперь давайте решим задачу программирования.

Задача. Определите тип структуры для программы, имитирующей «прыгающий мяч», которая упоминалась в самом начале этой главы. Местоположение мяча – это одно число, определяющее расстояние в пикселях от верхнего края холста. Абсолютная скорость – это количество пикселей, на которое мяч перемещается за такт часов. Фактическая скорость – это абсолютная скорость **плюс** направление движения.

Поскольку мяч перемещается только по вертикали, для представления фактической скорости достаточно обычного числа:

- положительное число означает, что мяч движется вниз;
- отрицательное число означает, что мяч движется вверх.

На основе этого знания предметной области можно сформулировать следующее определение типа структуры:

```
| (define-struct ball [location velocity])
```

Оба поля будут содержать числа, поэтому (`make-ball 10 -3`) – хороший пример данных. Согласно этому примеру, мяч находится на расстоянии 10 пикселей от верхнего края холста и перемещается вверх со скоростью 3 пикселя за такт часов.

Обратите внимание, что структура `ball` просто объединяет два числа, подобно структуре `posn`. Встретив в программе выражение (`ball-velocity a-ball`), вы сразу поймете, что эта программа имеет дело с представлением мяча и его скоростью. Напротив, если бы программа использовала структуры `posn`, то выражение (`posn-y a-ball`) могло бы ввести читателя кода в заблуждение: он мог бы подумать, что выражение относится к координате `y`.

Упражнение 67. Вот еще один способ представления прыгающего мяча:

```
| (define SPEED 3)
| (define-struct balld [location direction])
| (make-balld 10 "up")
```

Попробуйте интерпретировать этот фрагмент кода и создайте другие экземпляры `balld`. ■

Поскольку структуры являются значениями, так же как числа, логические значения или строки, то логично предположить, что экземпляр одной структуры может находиться внутри экземпляра другой структуры. Возьмем для примера игровые объекты. В отличие от прыгающих мячей, такие объекты не всегда движутся исключительно по вертикали, они могут двигаться по диагонали и вообще как угодно. Для описания местоположения и скорости мяча, движущегося по двумерному мировому холсту, требуются два числа: по одному для каждого направления. Местоположение такого объекта определяется двумя числами – координатами `x` и `y`. Скорость описывает характер изменения местоположения по горизонтали и вертикали. Иначе говоря, чтобы узнать, где будет находиться объект в следующий момент, необходимо эти «числа, описывающие характер изменений» прибавить к соответствующим координатам.

Согласно законам физики, чтобы определить следующее местоположение объекта, нужно прибавить скорость к его местоположению. Разработчики должны узнатъ, к кому обращаться с вопросами, касающимися предметной области.

Как мы уже знаем, местоположение можно представить с помощью структуры `posn`. Для представления скоростей определим тип структуры `vel`:

```
| (define-struct vel [deltax deltay])
```

В ней имеется два поля: `deltax` и `deltay`. Слово «`delta`» обычно используется для обозначения приращений, когда речь идет о моделировании физических действий, а окончания `x` и `y` указывают, какая ось координат подразумевается.

Теперь объединим структуры `posn` и `vel` в структуре `ball`, представляющей мяч, движущийся по прямым линиям, но не обязательно по вертикали или горизонтали:

```
| (define ball1
  (make-ball (make-posn 30 40) (make-vel -10 5)))
```

Экземпляр этой структуры можно представить как мяч, находящийся в 30 пикселях от левого и в 40 пикселях от верхнего края холста и перемещающийся за каждый такт часов на 10 пикселей влево (потому что вычитание 10 из координаты x приблизит его к левому краю) и на 5 вниз (потому что прибавление положительного числа к координате у увеличивает расстояние от верхнего края).

Альтернативное решение – использовать комплексные числа. Если вы знакомы с ними, то могли бы использовать их для представления местоположения и скорости. Например, в *BSL* комплексное число $4\text{-}3i$ можно использовать для обозначения местоположения или скорости $(4, -3)$.

Упражнение 68. Вместо использования вложенных структур для представления мяча можно определить структуру с четырьмя полями:

```
| (define-struct ballf [x y deltax deltay])
```

На языке программистов такое представление называется *плоским представлением*. Создайте экземпляр *ballf*, интерпретируемый так же, как экземпляр *ball1*. ■

Рассмотрим еще один пример вложенных структур: списков контактов. Многие сотовые телефоны поддерживают списки контактов, которые позволяют указывать несколько телефонных номеров для каждого имени: домашний, рабочий и сотовый. Для телефонных номеров можно также указать код города и местный номер. Поскольку эта информация имеет вид вложений, лучше определить вложенное представление данных:

```
| (define-struct centry [name home office cell])
| (define-struct phone [area number])
|
| (make-centry "Shriram Fisler"
|               (make-phone 207 "363-2421")
|               (make-phone 101 "776-1099")
|               (make-phone 208 "112-9981"))
```

Идея состоит в том, что запись в списке контактов имеет четыре поля: имя и три телефонных номера. Последние представлены экземплярами *phone*, которые отделяют код города и местный номер телефона.

В окружающей нас действительности вложенная информация существует повсюду, и лучший способ представить такую информацию в виде данных – отразить вложенность с помощью вложенных экземпляров структур. Это упрощает интерпретацию данных в прикладной области программы, а также переход от примеров информации к данным. Конечно, на самом деле задача определения данных состоит в том, чтобы указать, как преобразовывать информацию в данные и обратно. Однако, прежде чем перейти к приемам определения данных с использованием структур, мы систематически исследуем вычисления со структурами и поразмышляем о них.

5.5. Вычисления со структурами

Типы структур обобщают данные двумя способами. Во-первых, структура может иметь произвольное количество полей: ноль, одно, два, три и т. д. Во-вторых, поля в структурах именуются, а не нумеруются. Это упрощает чтение кода, потому что гораздо легче запомнить, что фамилия доступна в поле с именем `last-name`, чем в поле с номером 7.

Аналогично вычисления с экземплярами структур обобщают манипуляции с данными. Чтобы оценить эту идею, рассмотрим схематический способ представления экземпляров структуры в виде ящиков с количеством отсеков, совпадающим с количеством полей. Вот пример определения структуры:

```
| (define pl (make-entry "Al Abe" "666-7771" "lee@x.me"))
```

и схема:

entry		
name	phone	email
"Al Abe"	"666-7771"	"lee@x.me"

Надпись `entry`, напечатанная наклонным шрифтом, указывает, что здесь изображен экземпляр данного типа структуры; каждая ячейка тоже имеет свою подпись. Вот еще один пример:

```
| (make-entry "Tara Harp" "666-7770" "th@smlu.edu")
```

Ему соответствует похожая схема, но с другим содержимым:

entry		
name	phone	email
"Tara Harp"	"666-7770"	"th@smlu.edu"

Экземпляры вложенных структур тоже можно изобразить как наборы ячеек, вложенные в другие ячейки. Вот пример схемы для экземпляра структуры `ball1`, который был создан выше:

ball	
location	velocity
posn	vel
x	deltax
30	-10
y	deltay
40	+5

В этом случае внешняя ячейка содержит две вложенные ячейки, по одной для каждого поля.

Упражнение 69. Нарисуйте схему представления для решения упражнения 65. ■

Большинство языков программирования поддерживают также структуры или похожие на структуры типы, в которых используются числовые имена полей.

В контексте таких схем селектор можно сравнить с кнопкой, открывающей конкретный отсек в ящике определенного типа и тем самым позволяющей владельцу извлечь содержимое. Следуя этой логике, применение `entry-name` к `pl` даст строку:

```
| > (entry-name pl)
| "Al Abe"
```

Но попытка применить `entry-name` к экземпляру структуры `posn` за-кончится ошибкой:

```
| > (entry-name (make-posn 42 5))
| entry-name: expects an entry, given (posn 42 5)
```

(`entry-name`: ожидалась `entry`, а получена (`posn 42 5`)).

Если в отсеке находится другой ящик, может потребоваться использовать два селектора подряд, чтобы добраться до нужного числа:

```
| > (ball-velocity ball1)
| (make-vel -10 5)
```

Применение `ball-velocity` к `ball1` вернет значение поля скорости – экземпляр `vel`. Чтобы получить скорость по оси `x`, нужно применить селектор к результату первого селектора:

```
| > (vel-deltax (ball-velocity ball1))
| -10
```

Внутреннее выражение извлекает скорость из `ball1`, а внешнее – значение поля `deltax`, которое в данном случае равно `-10`.

Кроме того, взаимодействия показывают, что экземпляры структуры являются значениями. DrRacket выводит их точно так, как они были введены:

```
| > (make-vel -10 5)
| (make-vel -10 5)
| > (make-entry "Tara Harp" "666-7770" "th@smlu.edu")
| (make-entry "Tara Harp" "666-7770" "th@smlu.edu")
| > (make-centry
|   "Shriram Fisler"
|   (make-phone 207 "363-2421")
|   (make-phone 101 "776-1099")
|   (make-phone 208 "112-9981"))
| (make-centry ...)
```

Стоп! Попробуйте выполнить последнее действие самостоятельно, чтобы увидеть правильный результат.

Вообще говоря, определение структуры не только создает новые функции и новые способы создания значений, но также добавляет новые законы вычислений. Эти законы являются обобщением законов, с которыми мы познакомились в разделе 5.2, когда исследовали структуру `posn`. Чтобы лучше понять их, рассмотрим пример.

Когда DrRacket встречает определение типа структуры с двумя полями:

```
| (define-struct ball [location velocity])
```

вводятся два закона, по одному для каждого селектора:

```
| (ball-location (make-ball l0 v0)) == l0
| (ball-velocity (make-ball l0 v0)) == v0
```

Для других структур вводятся аналогичные законы. Например, для определения

```
| (define-struct vel [deltax deltay])
```

DrRacket добавит следующие два закона в свою базу знаний:

```
| (vel-deltax (make-vel dx0 dy0)) == dx0
| (vel-deltay (make-vel dx0 dy0)) == dy0
```

Используя эти законы, можно объяснить результаты взаимодействий из примера выше:

```
(vel-deltax (ball-velocity ball1))
== ; DrRacket заменит структуру ball1 ее значением
(vel-deltax
  (ball-velocity
    (make-ball (make-posn 30 40) (make-vel -10 5))))
== ; DrRacket использует закон для ball-velocity
(vel-deltax (make-vel -10 5))
== ; DrRacket использует закон для vel-deltax
-10
```

Упражнение 70. Опишите законы, действующие для следующих определений структур:

```
| (define-struct centry [name home office cell])
| (define-struct phone [area number])
```

Используйте движок пошаговых вычислений в DrRacket, чтобы убедиться, что следующее выражение действительно возвращает 101:

```
(phone-area
  (centry-office
    (make-centry "Shriram Fisler"
      (make-phone 207 "363-2421")
      (make-phone 101 "776-1099")
      (make-phone 208 "112-9981")))) ■
```

В заключение, чтобы окончательно понять идею определения структур, мы должны обсудить предикаты. Как уже упоминалось, каждое определение структуры вводит один новый предикат. DrRacket использует эти предикаты, чтобы определить, правильно ли применяется селектор к указанному значению. Более подробно эта идея объясняется в следующей главе, а здесь мы просто покажем, что предикаты похожи на предикаты из «арифметики». Предикат `number?` распознает числа, предикат `string?` – строки, предикаты `posn?` и `entry?` распознают экземпляры структур `posn` и `entry`. Эту нашу догадку мож-

но подтвердить экспериментами в области взаимодействий. Предположим, что область определений содержит следующие определения:

```
| (define ap (make-posn 7 0))
| (define pl (make-entry "Al Abe" "666-7771" "lee@x.me"))
```

Если верна наша догадка, что предикат `posn?` отличает экземпляры структуры `posn` от всех других значений, то можно предположить, что он вернет `#false` для чисел и `#true` для `ap`:

```
| > (posn? ap)
#true
| > (posn? 42)
#false
| > (posn? #true)
#false
| > (posn? (make-posn 3 4))
#true
```

Аналогично предикат `entry?` отличает экземпляры структуры `entry` от любых других значений:

```
| > (entry? pl)
#true
| > (entry? 42)
#false
| > (entry? #true)
#false
```

В общем случае предикат распознает именно те значения, которые созданы с помощью конструктора с тем же именем. В интермэццо 1 подробно объясняется этот закон, а также представлены другие законы вычислений в BSL.

Упражнение 71. Добавьте следующий код в область определений DrRacket:

```
; расстояния определяются в пикселях:
(define HEIGHT 200)
(define MIDDLE (quotient HEIGHT 2))
(define WIDTH 400)
(define CENTER (quotient WIDTH 2))

(define-struct game [left-player right-player ball])

(define game0
  (make-game MIDDLE MIDDLE (make-posn CENTER CENTER)))
```

Щелкните на кнопке **RUN** (Выполнить) и вычислите следующие выражения:

```
| (game-ball game0)
| (posn? (game-ball game0))
| (game-left-player game0)
```

Объясните результаты шаг за шагом. Проверьте свои объяснения с помощью движка пошаговых вычислений в DrRacket. ■

5.6. Программирование со структурами

Правильное программирование требует правильного определения данных. С введением определений типов структур определение данных становится еще интереснее. Помните, что определение данных обеспечивает способ представления информации в виде данных и интерпретацию этих данных как информации. В случае со структурами это требует описания того, какие данные в каких полях хранятся. Для некоторых определений структур сформулировать такие описания легко и просто:

```
(define-struct posn [x y])
; Posn -- это структура:
;   (make-posn Число Число)
; интерпретация: точка, находящаяся на расстоянии
; x пикселей от левого и y пикселей от верхнего края холста
```

Нет смысла использовать другие типы данных для создания posn. Точно так же все поля в entry – в структуре записей в списке контактов – явно должны быть строками, как описывалось в предыдущем разделе:

```
(define-struct entry [name phone email])
; Entry -- это структура:
;   (make-entry Стока Стока Стока)
; интерпретация: имя, номер телефона и адрес электронной почты
```

Встретив posn или entry, читатель легко сможет интерпретировать экземпляры этих структур.

Сравните эту простоту с определением структуры ball, которое допускает, как минимум, две разные интерпретации:

```
(define-struct ball [location velocity])
; Ball-1d -- это структура:
;   (make-ball Число Число)
; интерпретация 1: расстояние от верхнего края и скорость
; интерпретация 2: расстояние от левого края и скорость
```

Какую бы из них мы ни использовали в программе, мы должны постоянно придерживаться ее. Однако, как показано в разделе 5.4, структуры ball можно использовать совершенно иначе:

```
; Ball-2d -- это структура:
;   (make-ball posn vel)
; интерпретация: 2-мерные местоположение и скорость

(define-struct vel [deltax deltay])
; Vel -- это структура:
;   (make-vel Число Число)
; интерпретация: (make-vel dx dy) означает приращение
; на dx пикселей [за такт] координаты x и
; на dy пикселей [за такт] координаты y
```

Здесь мы назвали вторую коллекцию данных Ball-2d, в противовес Ball-1d, чтобы описать представление данных для мяча, способного перемещаться по прямым линиям по мировому холсту. Проще говоря, одну и ту же структуру можно использовать двумя разными способами. Конечно, в рамках одной программы лучше придерживаться одного и только одного способа использования, иначе вы создадите себе проблемы на ровном месте.

Кроме того, Ball-2d ссылается на другое определение данных, а именно на определение Vel. Несмотря на то что все иные определения данных, использовавшиеся до сих пор, относились к встроенным коллекциям данных (Число, Логическое значение, Стока), вполне приемлемо (и это часто используется на практике), когда одно из наших определений данных ссылается на другое.

Упражнение 72. Сформулируйте определение данных для приведенного выше определения структуры phone, соответствующее приведенным примерам.

Затем сформулируйте определение данных для телефонных номеров, используя это определение структуры:

```
| (define-struct phone# [area switch num])
```

Исторически сложилось так, что первые три цифры определяют код города, следующие три – код телефонной станции в вашем районе, а последние четыре цифры – телефонный номер в этом районе. Опишите содержимое трех полей как можно точнее с использованием интервалов. ■

На этом этапе вам, возможно, интересно понять, что на самом деле означают определения данных. Этот вопрос и ответ на него – тема следующего раздела. А пока мы продолжим обсуждать особенности использования определений данных при проектировании программ.

Вот описание задачи для контекста.

Задача. Ваша команда разрабатывает интерактивную игровую программу, которая перемещает красную точку по холсту размером 100×100 и позволяет игрокам использовать мышь для перемещения точки в исходную позицию. Вот как далеко вы продвинулись:

```
(define MTS (empty-scene 100 100))
(define DOT (circle 3 "solid" "red"))

; A Posn represents the state of the world.
; Posn -> Posn
(define (main p0)
  (big-bang p0
    [on-tick x+]
    [on-mouse reset-dot]
    [to-draw scene+dot]))
```

Ваша задача – создать функцию `scene+dot`, которая добавляет красную точку на пустой холст в указанной позиции.

Описание задачи диктует следующую сигнатуру вашей функции:

```
| ; Posn -> Изображение
| ; добавляет красную точку в MTS в позицию p
| (define (scene+dot p) MTS)
```

Добавить описание назначения несложно. Как упоминалось в разделе 3.1, такое заявление использует параметр функции для выражения результата функции.

Теперь сформулируем пару примеров и оформим их в виде тестов:

```
| (check-expect (scene+dot (make-posn 10 20))
|               (place-image DOT 10 20 MTS))
| (check-expect (scene+dot (make-posn 88 73))
|               (place-image DOT 88 73 MTS))
```

Так как функция принимает структуру posn, очевидно, что она должна извлекать значения полей x и y:

```
| (define (scene+dot p)
|   (... (posn-x p) ... (posn-y p) ...))
```

После добавления этих дополнительных элементов в макет тела функции мы легко сможем дописать остальное определение. Используя place-image, данная функция помещает DOT в MTS в координаты, содержащиеся в p:

```
| (define (scene+dot p)
|   (place-image DOT (posn-x p) (posn-y p) MTS))
```

Функции могут создавать и возвращать структуры. Вернемся к нашему примеру выше, потому что он как раз предусматривает такую задачу.

Задача. Вашему коллеге предложено определить функцию x+, которая принимает структуру posn, увеличивает координату x на 3 и возвращает новую структуру с обновленной координатой x.

Как вы наверняка помните, функция x+ должна обрабатывать такты часов.

Мы можем взять за основу несколько первых шагов из процесса проектирования scene+dot:

```
| ; Posn -> Posn
| ; увеличивает координату x в p на 3
| (check-expect (x+ (make-posn 10 0)) (make-posn 13 0))
| (define (x+ p)
|   (... (posn-x p) ... (posn-y p) ...))
```

Сигнатура, описание назначения и пример – все это вытекает из постановки задачи. Вместо заголовка – функции с результатом по умолчанию – наш макет содержит два выражения с селекторами для posn. В конце концов, информация для вычисления результата долж-

на поступать из входных данных, а входные данные – это структура, содержащая два значения.

Теперь мы легко можем завершить определение. Поскольку желаемый результат – структура posn, функция использует make-posn для объединения составных частей:

```
| (define (x+ p)
|   (make-posn (+ (posn-x p) 3) (posn-y p)))
```

Упражнение 73. Спроектируйте функцию posn-up-x, которая принимает структуру posn p и число n и создает новую структуру posn, подобную p, с числом n в поле x.

Интересно отметить, что x+ можно определить, используя posn-up-x:

```
| (define (x+ p)
|   (posn-up-x p (+ (posn-x p) 3)))
```

Примечание. Такие функции, как posn-up-x, называются *функциями обновления, или функциями-сеттерами*. Они чрезвычайно полезны при разработке больших программ. ■

Функция также может создавать экземпляры структур из элементарных данных. Конечно, у нас уже естьстроенная функция make-posn, которая делает именно это, но давайте рассмотрим еще один пример.

Задача. Другому коллеге поручено спроектировать функцию reset-dot, которая устанавливает точку в координаты, где выполнен щелчок мышью.

Чтобы решить эту задачу, вспомним, как рассказывалось в разделе 4.3, что обработчики событий мыши принимают четыре значения: текущее состояние мира, координаты x и y указателя мыши в момент события и описание события MouseEvt.

Добавив знания из постановки задачи в рецепт проектирования программы, получаем сигнатуру, описание назначения и заголовок:

```
| ; Posn Число Число MouseEvt -> Posn
| ; для щелчков мыши, (make-posn x y); иначе p
| (define (reset-dot p x y me) p)
```

В примерах вызова обработчика событий мыши нам понадобятся структура posn, два числа и MouseEvt – специально сформированная строка. Например, щелчок мыши может быть представлен одной из двух строк: "button-down" и "button-up". Первая сообщает, что пользователь нажал кнопку мыши, вторая – что отпустил. Вот два примера, которые вы, возможно, захотите изучить и погонять в интерпретаторе:

```
| (check-expect
|   (reset-dot (make-posn 10 20) 29 31 "button-down")
|   (make-posn 29 31))
| (check-expect
|   (reset-dot (make-posn 10 20) 29 31 "button-up")
|   (make-posn 10 20))
```

Эта функция использует только элементарные данные, но описание назначения и примеры предполагают, что она различает два вида событий мыши MouseEvts: "button-down" и все остальные. Такое разделение предполагает использование выражения `cond`:

```
(define (reset-dot p x y me)
  (cond
    [(mouse=? "button-down" me) (... p ... x y ...)]
    [else (... p ... x y ...)]))
```

Следуя рецепту проектирования, этот макет ссылается на параметры, чтобы напомнить, какие данные доступны.

Дописать остальное определение снова не представляет труда, потому что описание назначения явно говорит, что должна вычислить функция в каждом из двух случаев:

```
(define (reset-dot p x y me)
  (cond
    [(mouse=? me "button-down") (make-posn x y)]
    [else p]))
```

Как и прежде, мы могли бы отметить, что `make-posn` создает экземпляры `posn`, но вы это и так знаете, и нет необходимости постоянно напоминать об этом.

Упражнение 74. Скопируйте все необходимые определения констант и функций в область определений DrRacket. Добавьте тесты и убедитесь, что они выполняются успешно. Затем запустите программу и с помощью мыши поместите красную точку в выбранное вами место. ■

Во многих программах приходится иметь дело с вложенными структурами. Проиллюстрируем этот момент еще одним небольшим отрывком из мировой программы.

Задача. Ваша команда проектирует игровую программу, которая перемещает объект по холstu с изменяющейся скоростью. Выбранное представление данных требует наличия двух определений:

```
(define-struct ufo [loc vel])
; UFO -- это структура:
; (make-ufo Posn Vel)
; интерпретация: (make-ufo p v) определяет местоположение
; p и скорость перемещения v
```

Не забывайте, что все это имеет отношение к физике.

Ваша задача – разработать функцию `ufo-move-1`, которая вычисляет местоположение НЛО спустя один такт часов.

Начнем с определения примеров, которые помогут понять суть определения данных:

```
(define v1 (make-vel 8 -3))
(define v2 (make-vel -5 -3))
(define p1 (make-posn 22 80))
(define p2 (make-posn 30 77))
```

Порядок следования этих определений имеет значение.
См. интермецо 1.

```
(define u1 (make-ufo p1 v1))
(define u2 (make-ufo p1 v2))
(define u3 (make-ufo p2 v1))
(define u4 (make-ufo p2 v2))
```

Первые четыре определения создают элементы *Vel* и *Posn*. Последние четыре – создают все возможные комбинации из первых четырех.

Теперь напишем сигнатуру, описание назначения, несколько примеров и заголовок функции:

```
; UFO -> UFO
; определяет, куда переместится и за один такт часов;
; скорость остается неизменной

(check-expect (ufo-move-1 u1) u3)
(check-expect (ufo-move-1 u2)
              (make-ufo (make-posn 17 77) v2))

(define (ufo-move-1 u) u)
```

В примерах применения функции используем примеры данных и наши знания о местоположениях и скоростях. В частности, мы знаем, что транспортное средство, движущееся на север со скоростью 60 миль в час и на запад со скоростью 10 миль в час, через час окажется в 60 милях к северу от начальной точки и в 10 милях к западу. Через два часа оно будет находиться в 120 милях к северу от начальной точки и в 20 милях к западу.

Как всегда, функция, принимающая экземпляр структуры, может (и, вероятно, должна) извлечь информацию из структуры для вычисления результата. Итак, снова добавим выражения с селекторами в определение функции:

```
(define (ufo-move-1 u)
  (... (ufo-loc u) ... (ufo-vel u) ...))
```

Примечание. После добавления селекторов в макет функции возникает вопрос: потребуется ли нам продолжить уточнение извлекаемых данных? В конце концов, эти два селектора извлекают экземпляры *posn* и *vel* соответственно, которые в свою очередь являются структурами, и мы можем извлечь из них значения их полей. Вот как будет выглядеть макет функции в случае утвердительного ответа:

```
; UFO -> UFO
(define (ufo-move-1 u)
  (... (posn-x (ufo-loc u)) ...
        ... (posn-y (ufo-loc u)) ...
        ... (vel-deltax (ufo-vel u)) ...
        ... (vel-deltay (ufo-vel u)) ...))
```

Однако такой подход делает макет функции довольно сложным. При проектировании реалистичных программ доведение этой идеи до логического конца может приводить к созданию невероятно сложных схем. Поэтому установим следующее правило:

Если функция имеет дело с вложенными структурами, для обработки каждого уровня вложенности должна быть создана отдельная функция.

Во второй части книги это правило станет еще более важным, и мы немного уточним его. **Конец.**

Теперь сосредотачиваемся на том, как объединить данные экземпляры `posn` и `vel`, чтобы вычислить следующее местоположение НЛО, как того требуют наши знания физики. В частности, мы знаем, что нужно «сложить» их вместе, где слово «сложить» может не означать операцию, которую мы обычно применяем к числам. Итак, представим, что у нас есть функция сложения `Vel` и `Posn`:

```
| ; Posn Vel -> Posn
| ; складывает v и p
| (define (posn+ p v) p)
```

Запишем сигнатуру, описание назначения и заголовок, следуя установленному нами рецепту проектирования. Это называется «загадывать желание» и является частью «составления списка желаний», как описано в разделе 3.4.

Загадывать желания нужно так, чтобы потом мы смогли реализовать функцию, над которой работаем. Для этого можно использовать методику деления сложных задач на более простые подзадачи, которая поможет нам решить задачу небольшими шагами. Вот полное определение `uffo-move-1` для данной постановки задачи:

```
| (define (uffo-move-1 u)
|   (make-uffo (posn+ (uffo-loc u) (uffo-vel u))
|             (uffo-vel u)))
```

Поскольку `uffo-move-1` и `posn+` у нас полностью определены, можно даже щелкнуть на кнопке **RUN** (Выполнить) и проверить, не сообщит ли DrRacket о грамматических ошибках. Естественно, тесты завершаются с ошибками, потому что `posn+` – это пока простое желание, но не функция, которая нам нужна.

Теперь сосредоточимся на `posn+`. Мы выполнили первые два шага из рецепта проектирования (определенны данные, сигнатура/назначение/заголовок), поэтому дальше создадим примеры. Один простой способ создать примеры для «желания» – использовать примеры для исходной функции и превратить их в примеры для новой функции:

В геометрии операция, соответствующая функции `posn+`, называется переносом.

```
| (check-expect (posn+ p1 v1) p2)
| (check-expect (posn+ p1 v2) (make-posn 17 77))
```

В данном случае мы знаем, что выражение `(uffo-move-1 (make-uffo p1 v1))` должно дать в результате `p2`. В то же время мы знаем, что `uffo-move-1` применяет `posn+` к `p1` и `v1`, то есть для этих входных данных `posn+` должна дать `p2`.

Стоп! Проверьте наши расчеты вручную, чтобы убедиться, что вы понимаете все, что мы делаем.

Теперь добавим селекторы в наш макет:

```
(define (posn+ p v)
  (... (posn-x p) ... (posn-y p) ...
    ... (vel-deltax v) ... (vel-deltay v) ...))
```

Поскольку `posn+` использует экземпляры `Posn` и `Vel` и каждый экземпляр является структурой с двумя полями, мы получаем четыре выражения. В отличие от выражений с вложенными селекторами, приведенных выше, это простые применения селекторов к параметрам.

Если вспомнить, что означают эти четыре выражения, или как мы вычисляли желаемый результат на основе двух структур, то мы с легкостью сможем завершить определение `posn+`:

```
(define (posn+ p v)
  (make-posn (+ (posn-x p) (vel-deltax v))
    (+ (posn-y p) (vel-deltay v))))
```

Первый шаг – прибавить скорость по горизонтали к координате `x` и скорость по вертикали к координате `y`. Это дает два выражения, по одному для каждой новой координаты. С помощью `make-posn` новые координаты можно объединить в одну структуру `Posn`.

Упражнение 75. Введите эти определения и тестовые примеры в область определений DrRacket и убедитесь, что они выполняются без ошибок. Это первый случай, когда вы имеете дело с «желанием», и вы должны убедиться, что понимаете, как работает эта пара функций. ■

5.7. Вселенная данных

Помните, что математики называют это коллекциями данных, или множеством классов данных.

У каждого языка своя вселенная данных. Эти данные представляют информацию из внешнего мира, описывающую его. Именно этими данными манипулируют программы. Эта вселенная данных представляет собой коллекцию, которая содержит не только все встроенные данные, но также любые экземпляры данных, которые любая программа может когда-либо создать.

Слева на рис. 8 показан один из способов представления вселенной данных в языке BSL. Поскольку существует бесконечно много чисел и строк, коллекция всех данных бесконечна. Мы обозначили «бесконечность» на рис. 8 многоточием «...», но настоящие определения должны избежать таких неточностей.

Ни программы, ни отдельные функции в программах никогда не будут сталкиваться со всей вселенной данных. Цель определения данных – описать части этой вселенной и дать им имена, чтобы потом

можно было кратко ссылаться на них. Другими словами, определение именованных данных – это описание коллекции данных, а выбранное имя можно использовать в других определениях данных и в сигнатурах функций. Имя в сигнатуре функции указывает, какие данные будет обрабатывать функция и, неявно, с какой частью вселенной данных она не будет иметь дела.

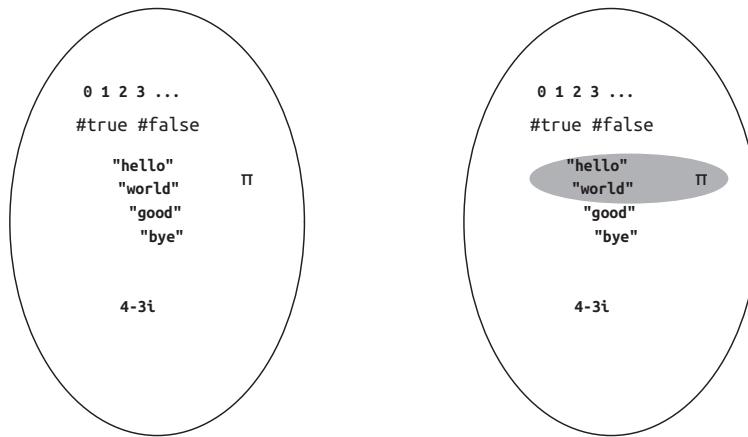


Рис. 8. Вселенная данных

Определения данных, которые мы использовали в первых четырех главах, ограничивались встроенными коллекциями данных. Для этого мы использовали явную или неявную детализацию всех значений, включенных в коллекцию. Например, область с серым фоном справа на рис. 8 соответствует следующему определению данных:

```
; BS -- одно из значений:  
; --- "hello",  
; --- "world",  
; --- число π.
```

Это конкретное определение данных выглядит довольно бессмысленным, но обратите внимание на стилизованное сочетание слов на естественном языке и на языке BSL. Это определение является точным и недвусмысленным. Оно точно определяет, какие элементы принадлежат BS, а какие нет.

Определение структур полностью меняет картину. Когда программист определяет структуры, вселенная расширяется и дополняется всеми возможными экземплярами структуры. Например, добавление *posn* означает появление экземпляров *posn* со всеми возможными значениями в двух полях. Кружок в центре на рис. 9 изображает добавление этих значений, включая такую кажущуюся бессмыслицу, как (*make-posn* "hello" 0) и (*make-posn* (*make-posn* 0 1) 2). И да, некоторые из этих экземпляров *posn* не имеют для нас никакого смысла. Но программа на BSL может сконструировать любой из них.

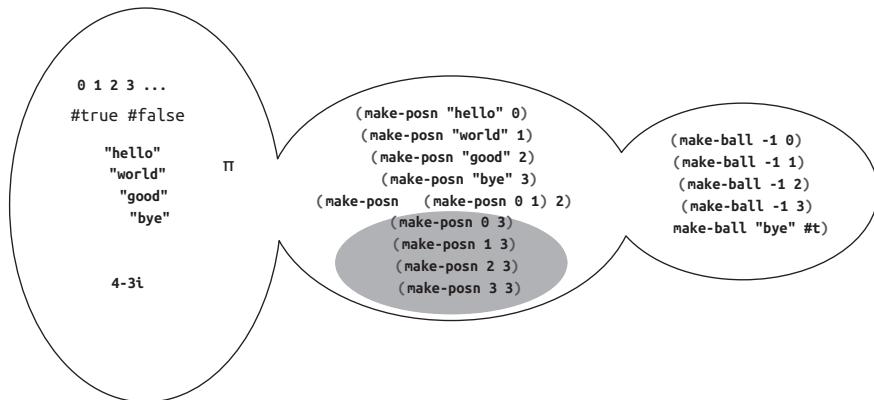


Рис. 9. Добавление структуры во вселенную данных

Добавление еще одного определения структуры снова расширяет вселенную данных всеми возможными комбинациями. Допустим, мы добавили определение структуры *ball* с двумя полями. Как показано в третьем круге на рис. 9, добавление этого определения создает коллекцию экземпляров *ball*, которые содержат числа, структуры *posn* и т. д., а также экземпляры *posn*, которые содержат экземпляры *ball*. Попробуйте создать такие экземпляры в DrRacket! Добавьте

```
| (define-struct ball [location velocity])
```

в область определений, щелкните на кнопке **RUN** (Выполнить) и создайте несколько экземпляров структур.

С практической точки зрения, определение данных с использованием структур описывает большие коллекции данных через комбинации существующих определений данных. Когда мы пишем

```
| ; Posn -- это (make-posn Число Число)
```

мы описываем бесконечное количество возможных экземпляров *posn*. Как и выше, в определениях данных используются комбинации выражений на естественном языке, наборов данных, определенных в другом месте, и конструкторов данных. На данный момент в определении данных больше ничего не должно присутствовать.

Определение данных с использованием структур определяет новую коллекцию данных, состоящую из экземпляров этих структур, которые будут использоваться нашими функциями. Например, определение данных с использованием структуры *posn* определяет заштрихованную область в центральном круге вселенной на рис. 9, который включает все структуры *posn*, два поля которых содержат числа. В то же время есть возможность создать экземпляр *posn*, не удовлетворяющий требованию, что оба поля должны содержать числа:

```
| (make-posn (make-posn 1 1) "hello")
```

Эта структура содержит экземпляр *posn* в поле *x* и строку в поле *y*.

Упражнение 76. Сформулируйте определения данных для следующих определений типов структур:

- (define-struct movie [title producer year]);
- (define-struct person [name hair eyes phone]);
- (define-struct pet [name number]);
- (define-struct CD [artist title price]);
- (define-struct sweater [material size producer]).

Сделайте разумные предположения о том, какие значения могут содержаться в каждом поле. ■

Упражнение 77. Определите структуру и данные для представления времени суток. Время состоит из трех чисел: часов, минут и секунд. ■

Упражнение 78. Определите структуру и данные для представления трехбуквенных слов. Слово состоит из строчных букв, представленных односимвольными строками *1String* от "a" до "z", плюс #false.

Примечание. Это упражнение является частью проекта игры «Виселица»; см. упражнение 396. ■

Программисты не только пишут определения данных, но и читают их, чтобы понять, как работает та или иная программа, расширить типы данных, с которыми они могут работать, устраниТЬ ошибки и т. д. Мы читаем определения данных, чтобы понять, как создавать данные, принадлежащие указанной коллекции, и выяснить, принадлежит ли экземпляр данных указанному классу.

Поскольку определения данных играют такую важную роль в процессе проектирования, часто желательно сопровождать определения данных примерами, как мы сопровождаем функции примерами, иллюстрирующими их поведение. Создавать примеры данных из определений довольно просто:

- для встроенной коллекции данных (число, строка, логическое значение, изображение) используйте любые примеры, какие вам нравятся;

ПРИМЕЧАНИЕ. Иногда для обозначения встроенных коллекций данных люди используют описательные имена, например NegativeNumber или OneLetterString. Но они не являются заменой хорошо написанного определения данных. **КОНЕЦ.**

- для перечисления используйте несколько элементов перечисления;
- для интервалов используйте конечные точки (если они есть) и хотя бы одну точку внутри;
- для детализаций проиллюстрируйте каждую часть отдельно;
- для структур следуйте описанию на естественном языке, то есть используйте конструктор и выберите примеры из коллекций данных, соответствующих каждому полю.

Это все, что потребуется для создания примеров на основе определений данных в большей части этой книги, правда, сами определения данных будут становиться все сложнее с каждой последующей главой.

Упражнение 79. Сконструируйте примеры для следующих определений данных:

- ; Color -- одна из строк, обозначающих цвет:
 ; --- "white"
 ; --- "yellow"
 ; --- "orange"
 ; --- "green"
 ; --- "red"
 ; --- "blue"
 ; --- "black"

ПРИМЕЧАНИЕ. DrRacket распознает многие другие строки как цвета. **КОНЕЦ.**

- ; N -- это Число между 0 и 100.
 ; интерпретация: представляет уровень счастья
 - (define-struct person [fname lname male?])
 ; Person -- это структура:
 ; (make-person Стока Стока Логическое_значение)
- Насколько оправданно использовать имя поля, похожее на имя предиката?
- (define-struct dog [owner name age happiness])
 ; Dog -- это структура:
 ; (make-dog Person Стока ПоложительноеЧисло N)
- Добавьте интерпретацию в это определение данных.
- ; Weapon -- одно из значений:
 ; --- #false
 ; --- Posn
 ; интерпретация: #false означает, что ракета еще не запущена;
 ; Posn представляет местоположение запущенной ракеты

Последнее определение является примером необычной детализации, сочетающей встроенные данные с типом структуры. Подобные определения мы подробно рассмотрим в следующей главе.

5.8. Проектирование с использованием структур

Добавление структурных типов усиливает необходимость неукоснительного выполнения всех шести шагов в рецепте проектирования. Мы больше не можем полагаться на встроенные коллекции данных

для представления информации. Теперь ясно, что программисты должны создавать определения данных для всех задач, кроме самых простых.

Этот раздел добавляет еще один рецепт проектирования, иллюстрируя его следующим образом:

Задача. Спроектируйте функцию, которая вычисляет расстояние от объектов в трехмерном пространстве до начала координат.

Поехали:

1. Когда задача требует представления информации, части которой неразделимо связаны друг с другом или описывают естественное целое, необходимо определить структуру. Структура должна включать столько полей, сколько имеется соответствующих свойств. Экземпляр такой структуры соответствует целому, а значения полей – атрибутам этого целого.

Определение данных для структуры предполагает определение имени для коллекции допустимых экземпляров. Кроме того, определение данных должно описывать, какие данные какому полю соответствуют. В определении можно использовать только имена встроенных коллекций данных или ранее объявленных определений данных.

В конце концов, мы (и другие) должны иметь возможность создавать экземпляры структуры, используя определение данных. В противном случае наше определение данных будет считаться неполным. Чтобы гарантировать возможность создания экземпляров, определения данных должны сопровождаться примерами данных.

Вот как можно применить эту идею к нашей задаче:

```
(define-struct r3 [x y z])
; R3 -- это структура:
; (make-r3 Число Число Число)

(define ex1 (make-r3 1 2 13))
(define ex2 (make-r3 -1 0 3))
```

Определение структуры вводит новый тип структуры, `r3`, а определение данных – имя `R3` для всех экземпляров структуры `r3`, содержащих только числа.

2. По-прежнему нужно определить сигнатуру, описание назначения и заголовок функции, но этот шаг ничем не отличается от аналогичного шага в прежнем рецепте проектирования.
Стоп! Выполните этот шаг самостоятельно для данной постановки задачи.
3. Используйте примеры из первого шага для создания примеров применения функции. Для каждого поля, представляющего интервал или перечисление, обязательно создайте примеры с использованием конечных и промежуточных точек. Мы наде-

емся, что вы сможете выполнить этот шаг самостоятельно для данной постановки задачи.

- Функция, принимающая структуры, часто, но не всегда извлекает значения из различных полей структур. Чтобы напомнить себе о такой возможности, добавьте селектор для каждого поля в макеты таких функций.

Вот как это можно представить для данной постановки задачи:

```
; R3 -> Число
; определяет расстояние от р до начала координат
(define (r3-distance-to-0 p)
  (... (r3-x p) ... (r3-y p) ... (r3-z p) ...))
```

При желании рядом с каждым выражением применения селектора можно показать, какие данные оно извлекает из этой структуры; подобную информацию можно взять из определения данных. Стоп! Просто сделайте это!

- При определении функции используйте селекторы из макета. Но имейте в виду, что некоторые из них могут не понадобиться.
- Тестирование. Протестируйте функцию сразу же, как только будет написан заголовок функции. Повторяйте тестирование, пока не будут охвачены все выражения. Тестируйте снова всякий раз, когда будете вносить изменения.

Там вы найдете такую формулу: $\sqrt{x^2 + y^2 + z^2}$. Завершите решение поставленной задачи. Если вы не можете вспомнить формулу вычисления расстояния от точки до начала координат в трехмерном пространстве, то поищите ее в книге по геометрии.

Упражнение 80. Создайте макеты для функций, принимающих экземпляры следующих структур:

- (define-struct movie [title director year]);
- (define-struct pet [name number]);
- (define-struct CD [artist title price]);
- (define-struct sweater [material size color]).

Для этой задачи не нужно создавать определения данных. ■

Упражнение 81. Спроектируйте функцию time->seconds, которая принимает экземпляр структуры с представлением времени суток (см. упражнение 77) и возвращает количество секунд, прошедших с полуночи. Например, если в структуре передается время 12 часов 30 минут и 2 секунды, то применение time->seconds к этому экземпляру должно вернуть 45002. ■

Упражнение 82. Спроектируйте функцию compare-word. Функция должна принимать два трехбуквенных слова (см. упражнение 78) и возвращать слово, указывающее, где указанные слова совпадают и не совпадают. Функция должна сохранять содержимое полей в структуре с результатом, соответствующие поля в исходных словах

совпадают, а в поле, где обнаружено несовпадение, она должна поместить значение `#false`.

Подсказка. Упражнение подразумевает две задачи: сравнение слов и сравнение «букв». ■

5.9. Структура в мире

Когда мировая программа должна следить за двумя независимыми элементами информации, для представления данных о состоянии мира необходимо использовать коллекцию структур. Одно поле хранит одну часть информации, а другое поле – вторую. Естественно, если мир предметной области характеризуется большим количеством элементов информации, определение структуры должно содержать столько полей, сколько имеется отдельных элементов информации.

Рассмотрим игру с космическими захватчиками, в которой имеются два игровых объекта: НЛО и танк. НЛО спускается вертикально вниз, а танк движется по горизонтали вдоль нижнего края сцены. Если оба объекта движутся с известной постоянной скоростью, то для их описания достаточно определить по одному элементу информации для каждого объекта: координату y для НЛО и координату x для танка. Для объединения этих элементов информации требуется структура с двумя полями:

```
| (define-struct space-game [ufo tank])
```

Мы оставляем вам, как самостоятельное задание, сформулировать адекватное определение данных для этого определения структуры, включая интерпретацию. Обратите внимание на дефис в имени структуры. Язык BSL позволяет использовать всевозможные символы в именах переменных, функций, структур и имен полей. Какие имена получат селекторы для этой структуры? Какое имя получит предикат?

Каждый раз, когда мы говорим «элемент информации», мы не всегда имеем в виду одно число или одно слово. Элемент информации сам может объединять несколько элементов информации. Создание представления данных для такого рода информации естественным образом приводит к вложенным структурам.

Добавим немного остроты в нашу воображаемую игру с космическими захватчиками. НЛО, спускающееся только по вертикали, – это скучно. Чтобы сделать игру более интересной, в которой танк атакует НЛО, НЛО должен иметь возможность спускаться не по прямой, возможно, прыгая случайным образом. Для реализации этой идеи нам понадобятся две координаты, описывающие местоположение НЛО, поэтому пересмотрим наше определение данных для космической игры:

```
| ; SpaceGame -- это структура:  
| ; (make-space-game Posn Число).
```

```
| ; интерпретация: (make-space-game (make-posn ux uy) tx)
| ; описывает конфигурацию, согласно которой НЛО находится в позиции
| ; (ux,uy), а танк -- на поверхности земли с x-координатой tx
```

Чтобы научиться понимать, какие представления данных необходимы для мировых программ, необходима практика. В следующих двух разделах мы рассмотрим несколько достаточно сложных постановок задач. Решите их, прежде чем переходить к играм, которые вы, возможно, захотите разработать самостоятельно.

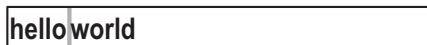
5.10. Графический редактор

Чтобы написать программу на языке BSL, нужно запустить среду программирования DrRacket и набрать текст программы на клавиатуре. Посмотрите, как набирается текст. Нажатие клавиши со стрелкой влево перемещает курсор влево; нажатие клавиши **Backspace** (или **Delete**) стирает одну букву слева от курсора (или справа соответствен-но), если, конечно, такая буква имеется.

Этот процесс называется «редактированием», хотя точнее его было бы называть «редактированием текста программы», потому что далее мы будем использовать слово «редактирование» для обозначе-ния более сложной задачи, чем набор текста на клавиатуре. Когда вы пишете и редактируете другие виды документов, то, вероятно, используете иные программные приложения, называемые текстовыми процессорами, хотя специалисты по информатике называют их все редакторами или даже графическими редакторами.

Теперь вы обладаете достаточным объемом знаний, чтобы разработать универсальную программу, которая действует как однострочный редактор для простого текста. Редактирование в данном случае включает ввод букв и какое-либо изменение уже существующего тек-ста, в том числе удаление и вставку букв. Это подразумевает наличие некоторого представления о позиции в тексте. Люди называют эту позицию *курсором*; большинство графических редакторов отображают его особым образом, чтобы его легко было обнаружить.

Взгляните на следующую конфигурацию редактора:



Кто-то уже ввел текст «helloworld» и пять раз нажал клавишу со стрелкой влево, в результате чего курсор переместился из конца текс-та в положение между буквами «о» и «w». Если теперь нажать кла-вишу пробела, то изображение в редакторе изменится следующим образом:



Проще говоря, это действие приведет к вставке пробела и перемещению курсора вправо, между пробелом и буквой «w».

Учитывая вышесказанное, редактор должен поддерживать два элемента информации:

- 1) введенный текст;
- 2) текущее положение курсора.

А это предполагает использование структуры с двумя полями.

Мы можем представить несколько разных способов преобразования информации в данные и обратно. Например, одно поле в структуре может содержать весь введенный текст, а другое – индекс, то есть количество символов между началом строки и курсором. Другое представление данных – использовать две строки в двух полях: в одном поле хранится часть текста слева от курсора, а в другом – часть текста справа. Мы выбрали этот второй подход к представлению состояния редактора:

```
(define-struct editor [pre post])
; Editor -- это структура:
;   (make-editor Стока Стока)
; интерпретация: (make-editor s t) описывает редактор
; с видимым в нем текстом (string-append s t), где
; курсор отображается между s и t
```

Решите несколько следующих упражнений, исходя из этого представления данных.

Упражнение 83. Спроектируйте функцию `render`, которая принимает состояние редактора `Editor` и создает изображение.

Назначение функции – показать текст в пустой сцене размером 200×20 пикселей. Курсор должен отображаться как красный прямоугольник с размерами 1×20 , а текст строки – черным шрифтом с размером 16.

Сконструируйте в области взаимодействия DrRacket изображение, которое будет играть роль образца. Мы сами начали со следующего выражения:

```
(overlay/align "left" "center"
  (text "hello world" 11 "black")
  (empty-scene 200 20))
```

Возможно, вам понадобится заглянуть в документацию с описанием `beside`, `above` и других подобных функций. Если вам понравится внешний вид изображения, используйте сконструированное выражение в качестве теста и руководства по проектированию функции `render`. ■

Упражнение 84. Спроектируйте функцию `edit`. Эта функция принимает два параметра, состояние редактора `ed` и KeyEvent `ke`, и создает редактор. Ее назначение – добавить символ из события `ke` в конец поля `pre` в структуре `ed`, если `ke` не является событием нажатия клавиши **Backspace** ("\"b"). В противном случае функция должна удалить

символ слева от курсора (если он есть). Функция игнорирует нажатие клавиш табуляции ("\") и **Return** ("\").

Функция обращает внимание только на два события длиннее одной буквы: "left" и "right". По нажатии клавиши со стрелкой влево курсор должен переместиться на один символ влево (если он есть), а по нажатии клавиши со стрелкой вправо курсор должен переместиться на один символ вправо (если он есть). Все другие события клавиатуры игнорируются.

Сконструируйте несколько тестовых примеров для `edit`, обращая внимание на особые случаи. Решая это упражнение, мы создали 20 примеров и превратили их все в тесты.

Подсказка. Рассуждайте об этой функции как о принимающей события клавиатуры – коллекции, которая определена как перечисление. Она может использовать вспомогательные функции для работы со структурой `Editor`, представляющей состояние редактора. Держите список желаний под рукой; вам потребуется спроектировать все эти вспомогательные функции, такие как `string-first`, `string-rest`, `string-last` и `string-remove-last`. Если вы еще не сделали этого, выполните упражнения из раздела 2.1. ■

Упражнение 85. Определите функцию `rep`. Принимая поле `rge` структуры `Editor`, она должна запускать интерактивный редактор, используя функции `render` и `edit` из двух предыдущих упражнений в предложениях `to-draw` и `on-key` соответственно. ■

Упражнение 86. Обратите внимание, что если попытаться ввесить длинный текст, то наша программа-редактор не сможет отобразить его весь. Не вместившийся в сцену текст просто будет обрезан по правому краю. Измените функцию `edit` из упражнения 84 так, чтобы она игнорировала нажатия клавиш, если после добавления нового символа в конец поля `rge` текст окажется слишком длинным для холста. ■

Упражнение 87. Разработайте представление данных для редактора на основе нашей первой идеи с использованием строки и индекса. Затем снова выполните предыдущие упражнения, **следуя рецепту проектирования**. **Подсказка:** выполните упражнения из раздела 2.1, если вы еще не сделали этого.

Замечание о выборе дизайна. Главная цель упражнений – научить вас проектированию. Как показывают примеры, самое первое решение, принимаемое при проектировании, касается представления данных. Чтобы сделать правильный выбор, необходимо заранее обдумать и взвесить каждый из возможных вариантов. Конечно, чтобы добиться успеха в этом вопросе, нужно набраться опыта. ■

5.11. Больше виртуальных питомцев

В этом разделе мы продолжим наш проект виртуального зоопарка из раздела 3.7. В частности, цель этого упражнения – объединить

программу кошачьего мира с программой управления индикатором счастья. Объединенная программа должна отображать кошку, перемещающуюся по холсту, и с каждым шагом ее усталость нарастает, а уровень счастья падает. Единственный способ поднять настроение кошке – покормить ее (нажать клавишу со стрелкой вниз) или погладить (нажать клавишу со стрелкой вверх). Наконец, цель последнего упражнения в этом разделе – создать еще одного счастливого виртуального питомца.

Упражнение 88. Определите структуру, которая хранит координаты кошки и уровень ее счастья. Затем сформулируйте определение данных для кошек с названием VCat, включая интерпретацию. ■

Упражнение 89. Спроектируйте мировую программу happy-cat, которая управляет гуляющей кошкой и ее уровнем счастья. В первый момент после запуска программы кошка должна быть совершенно довольна.

Подсказки. (1) Повторно используйте функции из мировых программ в разделе 3.7. (2) Используйте структуру из предыдущего упражнения для представления состояния мира. ■

Упражнение 90. Измените программу happy-cat из предыдущих упражнений так, чтобы она прекращала выполнение всякий раз, когда уровень счастья кошки падает до 0. ■

Упражнение 91. Расширьте определение структуры и определение данных из упражнения 88, включив в него поле направления движения. Измените программу happy-cat так, чтобы кошка двигалась в направлении, заданном в этом поле. Программа должна перемещать кошку в текущем направлении и разворачивать ее по достижении любого края сцены. ■



| (define cham

Приведенный выше рисунок хамелеона – это **прозрачное** изображение. Чтобы вставить его в DrRacket, выберите в меню пункт **Insert Image** (Вставить рисунок). Вставка таким способом сохранит прозрачность пикселей рисунка.

При объединении частично прозрачного изображения с цветной фигурой, например прямоугольником, изображение принимает цвет фигуры. На рисунке хамелеона внутренняя часть прозрачная, а область снаружи – белая. Попробуйте выполнить следующее выражение в DrRacket:

```
(define background
  (rectangle (image-width cham)
             (image-height cham)
             "solid"
             "red"))

(overlay cham background)
```

Упражнение 92. Спроектируйте программу `cham`, которая непрерывно перемещает хамелеона по холсту слева направо. По достижении правого края хамелеон должен исчезать и сразу же появляться слева. Подобно кошке, хамелеон постепенно устает от прогулки, и его уровень счастья падает.

Для управления шкалой уровня счастья хамелеона можно повторно использовать шкалу уровня счастья виртуальной кошки. Чтобы поднять настроение хамелеону, его можно покормить (нажав клавишу со стрелкой вниз); гладить хамелеона нельзя. Конечно, как и все хамелеоны, наш тоже может менять цвет: нажатие клавиши "г" делает его красным, "б" – синим, а "з" – зеленым. Объедините программы хамелеона и кошки и по возможности повторно используйте функции из последней.

Начните с определения данных `VCham` для представления хамелеона. ■

Упражнение 93. Скопируйте решение упражнения 92 и измените копию так, чтобы хамелеон прогуливался по трехцветному фону. В нашем решении используются следующие цвета:

```
(define BACKGROUND
  (beside (empty-scene WIDTH HEIGHT "green")
          (empty-scene WIDTH HEIGHT "white")
          (empty-scene WIDTH HEIGHT "red")))
```

Поехьте итальянской пиццы, когда закончите!

Но вы можете использовать любые другие цвета. Посмотрите, как хамелеон меняет цвет, пересекая границу между двумя цветами.

Примечание. Если внимательно посмотреть анимацию, то можно заметить, что хамелеон находится внутри белого прямоугольника. Если вы умеете пользоваться программами для редактирования изображений, измените изображение так, чтобы белый прямоугольник был невидим. Тогда хамелеон будет по-настоящему сливаться с фоном. ■

6. Структуры и детализация

В двух предыдущих главах были представлены два способа формулировки определений данных. Способ с применением детализации (перечислений и интервалов) используется для создания небольших коллекций из больших. Способ с применением структур используется для объединения нескольких коллекций. Конструирование представлений данных является основой для правильного проектирования программ, поэтому неудивительно, что у программистов часто возникает желание детализировать определения данных, включающие структуры, или использовать структуры для объединения детализированных данных.

Вспомните игру с воображаемыми космическими захватчиками из раздела 5.9 в предыдущей главе. Пока что в ней имеются один НЛО, спускающийся из космоса, и один танк на земле, движущийся по горизонтали. В нашем представлении данных используется структура с двумя полями: одно для представления данных об НЛО и другое – для представления данных о танке. Естественно, игровой танк должен иметь возможность запускать ракеты. Вот так неожиданно мы подошли к состоянию еще одного типа, содержащему три объекта, движущихся независимо: НЛО, танк и ракету. Теперь у нас есть две разные структуры: одна представляет два независимо движущихся объекта, а другая – третий. Поскольку состояние мира может быть только одной структурой, естественно использовать детализацию для описания всех возможных состояний:

- 1) состояние мира – это структура с двумя полями;
- 2) состояние мира – это структура с тремя полями.

Что касается нашей предметной области – реальной игры, – первое состояние представляет игру до того, как танк запустит свою единственную ракету, а второе – после запуска ракеты.

*Не волнуйтесь,
в следующей части
книги мы расскажем, как
реализовать запуск
любого количества
ракет без перезарядки.*

В этой главе мы представим основной замысел детализации определений данных, содержащих структуры. Поскольку у нас уже есть все необходимые ингредиенты, то сразу же приступим к детализации структур. После этого мы обсудим некоторые примеры, в том числе примеры мировых программ, извлекающих выгоду из наших новых возможностей. А в заключительном разделе поговорим об ошибках в программировании.

6.1. Проектирование с использованием детализации, снова

Для начала уточним постановку задачи для игры с космическими захватчиками из раздела 5.6.

Задача. Используя библиотеку *2htdp/universe*, спроектируйте игровую программу, имитирующую бой с захватчиками из космоса. Игрок управляет танком (маленький прямоугольник), который должен защищать нашу планету (нижняя часть холста) от НЛО (одну из возможных реализаций вы найдете в разделе 4.4), который спускается сверху вниз. Чтобы помешать НЛО приземлиться, игрок может запустить одну ракету (треугольник меньше танка), нажав клавишу пробела. Если ракета попадет в НЛО, игрок выигрывает; иначе НЛО приземляется, и игрок проигрывает.

Вот некоторые подробности о трех игровых объектах и их перемещениях. Во-первых, танк движется с постоянной скоростью вдоль нижнего края холста. Для изменения направления движения танка игрок может использовать клавиши со стрелками влево и вправо. Во-вторых, НЛО спускается с постоянной скоростью, но совершает небольшие случайные прыжки влево или вправо. В-третьих, после запуска ракеты поднимается вертикально вверх с постоянной скоростью, которая в два раза больше скорости спуска НЛО. Наконец, считается, что ракета попала в НЛО, если их координаты достаточно близки, независимо от того, что понимается под «достаточной близостью».

Эта постановка задачи будет использоваться в следующих двух подразделах в качестве рабочего примера, поэтому, прежде чем продолжить, внимательно изучите ее и решите следующее упражнение. Это поможет вам понять задачу достаточно глубоко.

Упражнение 94. Нарисуйте несколько набросков игрового пейзажа на разных этапах. Используйте наброски, чтобы определить постоянные и переменные элементы игры. Для начала определите физические и графические константы, которые описывают размеры мира (холста) и его объектов. Затем сконструируйте фоновый пейзаж. Наконец, создайте начальную сцену с использованием констант, определяющих параметры танка, НЛО и фона. ■

Определение детализации. Первый шаг в рецепте проектирования требует написать определение данных. Одна из целей определения данных – описать конструкцию данных, представляющих состояние мира; другая цель – описать все возможные элементы данных,

которые могут принимать функции обработки событий в мировой программе. Мы еще не встречались с перечислениями, включающими структуры, поэтому в первом подразделе мы рассмотрим эту идею. Возможно, вам эта идея не покажется новой, тем не менее обратите на нее пристальное внимание.

Как отмечалось во введении к этой главе, игра, имитирующая бой ракетного танка с космическими захватчиками, требует представления данных для двух различных состояний игры. Мы выбрали вариант с двумя структурами:

Для этой игры с космическими захватчиками мы могли бы обойтись одним определением структуры с тремя полями, где третье поле содержит `#false` до запуска ракеты и структуру `posn` с координатами ракеты после ее запуска. См. ниже.

```
| (define-struct aim [ufo tank])
| (define-struct fired [ufo tank missile])
```

Первая предназначена для представления периода времени, когда игрок пытается поставить танк в позицию для выстрела, а вторая – для представления состояний после выстрела ракеты. Однако, прежде чем сформулировать полное определение данных для состояния игры, нужны создать представления данных для танка, НЛО и ракеты.

Допустим, что вы уже объявили такие физические константы, как WIDTH и HEIGHT, которые являются предметом упражнения 94, и сформулируем следующие определения данных:

```
; UFO -- это Posn.
; интерпретация: (make-posn x y) -- местоположение НЛО
; (используется соглашение о системе координат с началом в левом верхнем углу)

(define-struct tank [loc vel])
; Tank -- это структура:
;   (make-tank Число Число).
; интерпретация: (make-tank x dx) определяет позицию:
; (x, HEIGHT) и скорость перемещения танка: dx пикселей за такт часов

; Missile -- это Posn.
; интерпретация: (make-posn x y) -- местоположение ракеты
```

Каждое из этих определений данных описывает структуру, либо вновь определенную, как tank, либо встроенную коллекцию данных Posn. Вас может немного удивить то, что определения Posn используются для представления двух разных аспектов мира. С другой стороны, в реальном мире мы широко используем числа (а также строки и логические значения) для представления самых разных видов информации, поэтому повторное использование коллекции структур, таких как Posn, не является чем-то особенным.

Теперь можно сформулировать определения данных для состояния игры с космическими захватчиками:

```
; SIGS -- это одно из следующих значений:
; -- (make-aim UFO Tank)
; -- (make-fired UFO Tank Missile)
; интерпретация: представляет полное состояние игры
; с космическими захватчиками
```

Определение данных имеет форму детализации. При этом каждое предложение описывает содержимое структуры точно так же, как определения структур, которые мы видели до сих пор. Однако, как показывает это определение, не всякое определение данных включает только одну структуру; в данном случае определение данных включает две разные структуры.

Смысл такого определения прост. Оно вводит имя SIGS для коллекции всех экземпляров структур, которые можно создать в соответствии с определением. Давайте создадим несколько примеров для наглядности:

- вот пример, описывающий состояние игры, когда танк перемещается к позиции пуска ракеты:

```
| (make-aim (make-posn 20 10) (make-tank 28 -3))
```

- следующий пример соответствует состоянию игры после пуска ракеты:

```
| (make-fired (make-posn 20 10)
             (make-tank 28 -3)
             (make-posn 28 (- HEIGHT TANK-HEIGHT)))
```

разумеется, имена, состоящие только из заглавных букв, – это имена констант, которые вы должны были определить в упражнении 94;

- и последний пример соответствует состоянию, когда ракета попала в НЛО:

```
| (make-fired (make-posn 20 100)
             (make-tank 100 3)
             (make-posn 22 103))
```

Этот пример предполагает, что холст имеет ширину больше 100 пикселей.

Обратите внимание, что первый экземпляр SIGS создается в соответствии с первым предложением в определении данных, а второй и третий – в соответствии со вторым предложением. Естественно, числа в каждом поле зависят от вашего выбора значений глобальных игровых констант.

Упражнение 95. Покажите, как при получении этих трех экземпляров использовался первый или второй вариант из определения данных. ■

Упражнение 96. Нарисуйте примерный вид игровой сцены для каждого из трех состояний на холсте размером 200×200 пикселей. ■

Рецепт проектирования. После знакомства с новым способом формулировки определений данных пришла пора уточнить рецепт проектирования. В этой главе исследуется возможность объединения двух и более средств описания данных, и уточненный рецепт проектирования отражает это обстоятельство, особенно первый его шаг:

- 1) когда может понадобиться этот новый способ определения данных? Вы уже знаете, что необходимость в детализации обусловлена различиями между разными классами информации в постановке задачи. Точно так же потребность в определении данных на основе структуры возникает из-за необходимости сгруппировать несколько разных элементов информации.
Детализация разных форм данных, включая коллекции структур, требуется, когда в постановке задачи присутствуют разные виды информации и когда хотя бы некоторые из них отличаются разным составом элементов.

Следует иметь в виду, что определения данных могут ссылаться на другие определения данных. Поэтому если конкретное предложение в определении данных выглядит слишком сложным, допускается добавить отдельное определение данных для этого предложения и сослаться на это вспомогательное определение. И как всегда, нужно сформулировать примеры данных, используя их определения;

- 2) второй шаг остается прежним. Сформулируйте сигнатуру функции, в которой упоминаются только имена вновь добавленных или встроенных коллекций данных, добавьте описание назначения и определите заголовок функции;
- 3) третий шаг тоже не изменился. Вы все так же должны сформулировать примеры применения функций, иллюстрирующие описание назначения из второго шага, и все так же должны записать хотя бы по одному примеру для каждого элемента в детализации;
- 4) при разработке макета теперь необходимо учитывать два разных аспекта: саму детализацию и структуры в ее предложениях. Для учета первого аспекта тело макета должно состоять из выражения `cond`, включающего столько условий, сколько имеется элементов в детализации. Также в каждое условие нужно добавить выражение `cond`, идентифицирующее подклассы данных в соответствующем элементе.

Для учета второго аспекта, если элемент представлен структурой, макет должен содержать выражения применения селекторов в каждом условии `cond`, идентифицирующем подкласс данных, описанный в элементе.

Однако если вы решили описать данные с использованием отдельного определения, то выражения селекторов добавлять **не** нужно. Вместо этого следует создать макет для отдельного определения данных в текущей задаче и ссылаться на этот макет путем применения функции. Это будет указывать на то, что данный подкласс данных обрабатывается отдельно.

Прежде чем приступить к разработке макета, поразмышляйте о характере функции. Если формулировка задачи предполагает необходимость решения нескольких задач, вполне вероятно, что вместо одной функции потребуется спроектировать и написать несколько функций. В таком случае пропустите этот шаг;

- 5) заполните пробелы в макете. Чем сложнее определение данных, тем сложнее этот шаг. Но не отчаивайтесь, потому что данный рецепт дизайна может помочь даже в самых сложных ситуациях.

Если вы застопорились, то сначала заполните простые случаи, а для остальных используйте значения по умолчанию. Некото-

рые тестовые примеры могут показывать неверный результат, тем не менее это будет вполне видимый шаг вперед.

Если вы застопорились на некоторых случаях детализации, проанализируйте примеры, соответствующие им. Определите, что вычисляют части макета, соответствующие заданным входным данным. Затем подумайте, как объединить эти части (и некоторые константы) для получения желаемого результата. Имейте в виду, что вам может понадобиться вспомогательная функция.

Кроме того, если ваш макет «вызывает» другой макет из-за ссылок друг на друга в определениях данных, то исходите из предположения, что та другая функция делает именно то, что обещают ее описание назначения и примеры, даже если определение той другой функции еще не завершено;

- 6) тестирование. Если тесты не проходят, определите, где находится причина: в функциях, в тестах или там и там. Вернитесь к соответствующему шагу.

Вернитесь к разделу 3.1, прочитайте рецепт простого проектирования и сравните его с этой версией.

А теперь проиллюстрируем использование этого рецепта на примере проектирования функции отображения для постановки задачи в начале данного раздела. Напомним, что для выражения `big-bang` нужна функция отображения, преобразующая состояние мира в изображение после каждого такта часов, щелчка мыши или нажатия клавиши.

Сигнатура этой функции ясно говорит о том, что она отображает экземпляр класса состояния мира в экземпляре класса изображений:

```
| ; SIGS -> Изображение
| ; добавляет TANK, UFO и, возможно, MISSILE в
| ; сцену BACKGROUND
(define (si-render s) BACKGROUND)
```

Здесь `TANK`, `UFO`, `MISSILE` и `BACKGROUND` – это имена констант, определяющие изображения, которые вы должны были создать в упражнении 94. Напомним, что эта сигнатура лишь следует обобщенной сигнатуре для функций отображения, которые всегда принимают коллекции состояний мира и создают какое-то изображение.

Детализация в определении данных включает два элемента, но мы создадим три примера, используя примеры данных, приведенные выше (см. табл. 4). В отличие от табличных функций, которые можно найти в книгах по математике, эта таблица отображается вертикально. Левый столбец содержит примеры входных данных, а в правом столбце представлены желаемые результаты. Как видите, мы использовали примеры данных из первого шага рецепта проектирования, охватывающие оба элемента детализации.

Таблица 4. Примеры отрисовки состояний игры

s	(si-render s)
(make-aim (make-posn 10 20) (make-tank 28 -3))	
(make-fired (make-posn 20 100) (make-tank 100 3) (make-posn 22 103))	
(make-fired (make-posn 10 20) (make-tank 28 -3) (make-posn 32 (- HEIGHT TANK-HEIGHT 10)))	

Теперь перейдем к разработке макета – самому важному этапу процесса проектирования. Во-первых, мы знаем, что тело `si-render` должно содержать выражение `cond` с двумя условиями. Согласно рецепту проектирования, этими условиями должны быть `(aim? s)` и `(fired? s)`, они различают два возможных вида данных, которые может получить `si-render`:

```
(define (si-render s)
  (cond
    [(aim? s) ...]
    [(fired? s) ...]))
```

Во-вторых, добавим селекторы в каждое условие `cond`, обрабатывающее структуры. В данном случае оба условия обрабатывают структуры: `aim` и `fired`. Первая имеет два поля и, следовательно, требует добавить в первое условие `cond` два селектора, а вторая имеет три поля и, соответственно, требует добавить три селектора:

```
(define (si-render s)
  (cond
    [(aim? s) (... (aim-tank s) ... (aim-ufo s) ...)]
    [(fired? s) (... (fired-tank s) ... (fired-ufo s)
                     ... (fired-missile s) ...)]))
```

Макет содержит почти все, что нужно для завершения поставленной задачи. Чтобы завершить определение, нужно выяснить для каждого условия в `cond`, как объединить имеющиеся значения, дабы получить желаемый результат. Кроме входных данных, мы можем использовать глобальные константы, например `BACKGROUND`, что явно не будет лишним; элементарные или встроенные операции; и, если

Упражнение 97. Спроектируйте функции `tank-render`, `ufo-render` и `missile-render`. Сравните первое выражение:

```
(tank-render
  (fire-tank s)
  (ufo-render (fire-ufo s)
    (missile-render (fire-missile s)
      BACKGROUND)))
```

СО ВТОРЫМ:

```
(ufo-render
  (fire-ufo s)
  (tank-render (fire-tank s)
    (missile-render (fire-missile s)
      BACKGROUND)))
```

В каких случаях эти два выражения дают одинаковый результат? ■

Упражнение 98. Спроектируйте функцию `si-game-over?` для использования в качестве обработчика `stop-when`. Игра должна останавливаться, когда НЛО достигает поверхности земли или ракета попадает в НЛО. В обоих случаях мы рекомендуем проверить близость одного объекта к другому.

Предложение `stop-when` допускает второе необязательное подвыражение – функцию, которая отображает конечное состояние игры. Спроектируйте функцию `si-render-final` и используйте ее как второе подвыражение в предложении `stop-when` в функции `main` в упражнении 100. ■

Упражнение 99. Спроектируйте функцию `si-move`. Она будет вызываться после каждого такта часов, чтобы определить новые позиции игровых объектов. Соответственно, она должна принимать один экземпляр `SIGS` и порождать другой.

Вычислить новую позицию танка и ракеты (если она была запущена) относительно несложно. Они движутся по прямой с постоянной скоростью. Но НЛО при перемещении может совершать случайные прыжки влево или вправо. Поскольку мы выше не рассказывали о функциях, возвращающих случайные числа, в оставшейся части данного упражнения мы покажем, как решить эту проблему.

В BSL имеется функция, которая возвращает случайные числа. Введение этой функции наглядно показывает, почему сигнатуры и описания назначения играют такую важную роль в проектировании. Вот соответствующие сведения о нужной вам функции:

```
; Число -> Число
; порождает число в интервале [0,n),
; число, полученное при каждом последующем вызове, может отличаться от предыдущего
(define (random n) ...)
```

Сигнатура и описание назначения точно описывают, что вычисляет функция, поэтому теперь вы сможете поэкспериментировать с функцией `random` в области взаимодействий DrRacket. Стоп! Сделайте это!

Идея об использовании `random` определяется знанием языка BSL и не является одним из навыков проектирования, которые вы должны приобрести, поэтому мы включили эту подсказку. Кроме того, `random` – это первая и единственная элементарная функция языка BSL, которая не является математической функцией. Функции в программировании имеют множество сходств с математическими функциями, но это не идентичные понятия.

Так как при каждом (почти) применении `random` возвращает разные числа, тестирование функций, использующих `random`, может вызывать определенные трудности. Для начала разделите `si-move` на две части:

```
(define (si-move w)
  (si-move-proper w (random ...)))

; SIGS Число -> SIGS
; перемещает НЛО на предсказуемое расстояние delta
(define (si-move-proper w delta)
  w)
```

Такое определение позволяет отделить получение случайного числа от процесса перемещения игровых объектов. Даже при том что `random` может возвращать разные результаты при каждом новом применении, `si-move-proper` можно протестировать с предопределенными числовыми входными данными и получить одинаковый результат при одинаковых входных данных. Проще говоря, при таком подходе большая часть кода останется доступной для тестирования.

Вместо прямого вызова `random` можно спроектировать функцию, которая создает случайную координату x для НЛО. Для тестирования такой функции можно использовать функцию `check-random` из фреймворка тестирования в языке BSL. ■

Упражнение 100. Спроектируйте функцию `si-control`, которая будет играть роль обработчика событий клавиатуры. Она должна принимать состояние игры и событие клавиатуры и создавать новое состояние игры. Функция должна реагировать на три события:

- нажатие клавиши со стрелкой влево должно вызывать перемещение танка влево;
- нажатие клавиши со стрелкой вправо должно вызывать перемещение танка вправо;
- нажатие клавиши пробела должно запускать ракету, если она еще не была запущена.

После создания этой функции вы сможете определить функцию `si-main`, которая использует выражение `big-bang` для создания окна игры. Желаем приятно провести время! ■

Листинг 20. Отображение состояния игры, снова

```
; SIGS.v2 -> Изображение
; отрисовывает состояние игры поверх BACKGROUND
(define (si-render.v2 s)
  (tank-render
    (sigs-tank s)
    (ufo-render (sigs-ufo s)
      (missile-render.v2 (sigs-missile s)
        BACKGROUND))))
```

Представления данных редко бывают уникальными. Например, мы могли бы использовать одну структуру для представления всех возможных состояний игры:

```
(define-struct sigs [ufo tank missile])
; SIGS.v2 (т. е. SIGS версии 2) -- это структура:
; (make-sigs UFO Tank MissileOrNot)
; интерпретация: представляет полное состояние
; с космическими захватчиками

; MissileOrNot -- одно из значений:
; -- #false
; -- Posn
; интерпретация: #false означает, что ракета находится в танке;
; Posn определяет местоположение ракеты после пуска
```

В отличие от первого представления данных, описывающего состояние игры, эта вторая версия не делает различий между состояниями до и после пуска ракеты. Вместо этого каждое состояние содержит некоторые данные о ракете, правда, этот элемент данных может быть простым значением `#false`, указывающим, что пуск ракеты еще не был произведен.

В результате функции, обрабатывающие это второе представление состояния, отличаются от функций для первого представления. В частности, функции, использующие элемент `SIGS.v2`, не используют выражение `cond`, потому что в коллекции имеется только один тип элементов. С точки зрения подхода к проектированию достаточно использовать рецепт проектирования со структурами из раздела 5.8. В листинге 20 показан результат проектирования функции отображения для этого представления данных.

Напротив, для проектирования функций с использованием представления `MissileOrNot` требуется рецепт из этого раздела. Давайте рассмотрим последовательность проектирования функции `missile-render.v2`, задача которой состоит в том, чтобы добавить ракету в изображение. Вот определение заголовка:

```
; MissileOrNot Изображение -> Изображение
; добавляет изображение ракеты m в сцену s
(define (missile-render.v2 m s)
  s)
```

При создании примеров мы должны учесть как минимум два случая: первый, когда `m` имеет значение `#false`, и второй, когда `m` является представлением `Posn`. В первом случае ракета не была запущена, а значит, ее изображение не нужно добавлять в сцену. Во втором случае мы имеем координаты ракеты, в которых должно появиться изображение ракеты. Таблица 5 демонстрирует работу функции в этих двух сценариях.

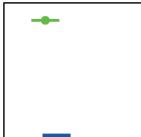
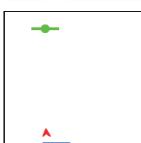
Упражнение 101. Преобразуйте примеры в табл. 5 в тесты. ■

Теперь мы готовы приступить к определению макета. Поскольку определение данных для главного аргумента (`m`) является детализа-

цией с двумя элементами, то тело функции, вероятно, будет состоять из выражения cond с двумя условиями:

```
(define (missile-render.v2 m s)
  (cond
    [(boolean? m) ...]
    [(posn? m) ...]))
```

Таблица 5. Примеры отображения состояний игры с танком и ракетой

m	(missile-render.v2 m s)
#false	
(make-posn 32 (- HEIGHT TANK-HEIGHT 10))	

Согласно определению данных, первое условие в выражении cond проверяет, является ли *m* логическим значением, а второе проверяет, является ли оно экземпляром Posn. Если кто-то по ошибке применит *missile-render.v2* к значению #true и к какому-нибудь изображению, то функция выполнит первое предложение в выражении cond. Далее мы подробнее расскажем о таких ошибках.

Второй шаг в процессе создания макета требует использовать селекторы во всех условиях в выражении cond, которые имеют дело со структурами. В нашем примере это относится ко второму условию, поэтому добавим применение селекторов для извлечения координат *x* и *y* из заданного экземпляра Posn:

```
(define (missile-render.v2 m s)
  (cond
    [(boolean? m) ...]
    [(posn? m) (... (posn-x m) ... (posn-y m) ...)]))
```

Сравните этот макет с макетом функции *si-render* выше. Определение данных для *si-render* включает два разных типа структур, поэтому макет данной функции содержит селекторы в обоих условиях в выражении cond. Определение данных *MissileOrNot* смешивает элементарные значения со структурой, поэтому селекторы потребовались только во втором условии. Оба способа определения данных допустимы, и ваша главная задача – следуя рецепту, найти организацию кода, которая соответствует определению данных.

Вот полное определение функции:

```
(define (missile-render.v2 m s)
  (cond
```

```
[(boolean? m) s]
[(posn? m)
  (place-image MISSILE (posn-x m) (posn-y m) s))])
```

Действуя шаг за шагом, сначала определяются простые условия; в данной функции простым является первое условие. Поскольку оно идентифицирует ситуацию, когда ракета не была запущена, то должно просто вернуть заданное значение *s*. Работая над вторым условием, нужно помнить, что (*posn-x m*) и (*posn-y m*) выбирают координаты для изображения ракеты. Данная функция должна добавить изображение MISSILE в сцену *s*, поэтому вам нужно выяснить лучшую комбинацию элементарных операций и ваших собственных функций для объединения этих четырех значений. Выбор операции комбинирования – это именно то место, где в игру вступает ваше творческое чутье программиста.

Упражнение 102. Спроектируйте все остальные функции, необходимые для завершения игры с этим вторым определением данных. ■

Упражнение 103. Сконструируйте представление данных для следующих четырех видов животных:

- **пауки**, атрибутами которых являются: количество оставшихся ног (мы предполагаем, что пауки могут терять ноги в результате несчастных случаев) и необходимый объем контейнера для их транспортировки;
- **слоны**, единственный атрибут которых – необходимый объем контейнера для транспортировки;
- **удавы**, атрибутами которых являются длина и обхват;
- **броненосцы**, для которых вы сами должны определить соответствующие атрибуты, в том числе необходимый объем контейнера для транспортировки.

Напишите макеты функций, которые принимают аргументы, представляющие этих животных.

Спроектируйте функцию *fits?*, которая принимает экземпляр животного и описание контейнера. Она должна определить, достаточно ли объема указанного контейнера для транспортировки заданного животного. ■

Упражнение 104. В любом городе есть свой парк транспортных средств: автомобилей, фургонов, автобусов и внедорожников. Разработайте представление данных для автомобилей. Представление каждого транспортного средства должно описывать максимальное количество пассажиров, номерной знак и расход топлива (литров на 100 км). Напишите макеты функций, которые принимают автомобили. ■

Упражнение 105. Некоторая программа содержит следующее определение данных:

```
; Координата -- это одно из значений:
; -- NegativeNumber
```

```

; интерпретация: для оси y, расстояние от верхнего края
; -- PositiveNumber
; интерпретация: для оси x, расстояние от левого края
; -- Posn
; интерпретация: обычные декартовы координаты точки

```

Составьте не менее, чем по два примера данных для каждого предложения в этом определении данных. Для каждого из примеров объясните его значение с помощью наброска на холсте. ■

6.2. Смешивание миров

В этом разделе мы исследуем несколько проблем проектирования мировых программ и начнем с простых упражнений, касающихся наших виртуальных питомцев.

Упражнение 106. В разделе 5.11 обсуждалось создание виртуальных питомцев с индикаторами довольства. Один из этих питомцев – кошка; другой – хамелеон. Однако каждая из обсуждавшихся программ управляет только одним питомцем.

Спроектируйте мировую программу *cat-cham*. Она должна управлять заданным животным, кошкой или хамелеоном, перемещая его по холstu, начиная с заданного местоположения. Вот описание представления данных для животных:

```

; VAnimal -- одно из значений
; -- a VCat
; -- a VCham

```

где *VCat* и *VCham* – определения данных, которые вы должны были сконструировать в упражнениях 88 и 92.

Учитывая, что *VAnimal* является коллекцией состояний мира, вы должны спроектировать:

- функцию преобразования *VAnimal* в изображение;
- функцию для обработки тактов часов и преобразующую состояние *VAnimal* в новое состояние *VAnimal*;
- функцию обработки событий клавиатуры, чтобы дать возможность кормить, гладить и раскрашивать свое животное в тех случаях, когда это применимо.

Изменить окрас кошки или погладить хамелеона по-прежнему невозможно. ■

Упражнение 107. Спроектируйте программу *cham-and-cat*, которая управляет сразу двумя животными: виртуальной кошкой и виртуальным хамелеоном. Вам потребуется определение данных «зоопарка», содержащее обоих животных и функции для работы с ними.

Постановка задачи не уточняет, какие клавиши использовать для управления двумя животными. Вот два возможных толкования:

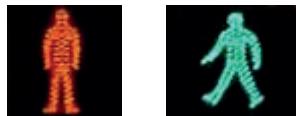
- 1) каждое событие клавиатуры затрагивает оба животных;
- 2) каждое событие клавиатуры затрагивает только одно из животных.

В этом случае вам понадобится добавить в представление данных некоторый признак, определяющий животное в фокусе, то есть животное, находящееся под управлением. Для переключения фокуса функция обработки событий клавиатуры может интерпретировать нажатие клавиши «`k`» как команду «`kitty`» (кошечка), а нажатие клавиши «`l`» как команду «`lizard`» (ящерица). После того как игрок нажмет клавишу «`k`», все следующие нажатия клавиш будут применяться только к кошке, пока игрок не нажмет клавишу «`l`».

Выберите один из вариантов и спроектируйте подходящую программу. ■

Упражнение 108. По умолчанию светофор для пешеходного перехода показывает оранжевую фигурку стоящего человека на черном фоне. Когда наступает момент разрешить пешеходам перейти улицу, светофор получает сигнал и показывает зеленую фигурку идущего человека. Этот период длится 10 секунд. Затем на табло светофора появляются цифры 9, 8, ..., 0, причем нечетные числа имеют оранжевый цвет, а четные – зеленый. Когда обратный отсчет достигает 0, светофор возвращается в состояние по умолчанию.

Спроектируйте мировую программу, реализующую такой пешеходный светофор. Светофор должен переключаться из состояния по умолчанию по нажатии клавиши пробела на клавиатуре. Все остальные изменения состояния светофора должны происходить автоматически, с течением времени. Вы можете использовать следующие изображения:



или нарисовать свои фигуры. ■

Упражнение 109. Спроектируйте мировую программу, которая распознает шаблон в последовательности событий клавиатуры. Первоначально программа показывает белый прямоугольник 100×100 . Встретив первую букву из заданного шаблона, она должна показать желтый прямоугольник того же размера. После встречи с последней буквой программа должна показать зеленый прямоугольник. Если нажимается «неверная» клавиша, не соответствующая шаблону, программа должна показать красный прямоугольник.

В частности, программа должна распознавать любые последовательности, начинающиеся с символа «`a`», за которым следует произвольное сочетание символов «`b`» и «`c`» любой длины, и всякая по-

следовательность должна заканчиваться буквой «d». Очевидно, что «acbd» – это одна из допустимых последовательностей; последовательности «ad» и «abcbbcd» тоже допустимы. Но «da», «aa» или «d» – это недопустимые последовательности.

Подсказка. Используйте в своем решении конечный автомат, идея которого была представлена в разделе 4.7 как один из принципов проектирования, лежащий в основе мировых программ. Как следует из названия, программа, реализующая конечный автомат, может находиться в одном из конечного числа состояний. Первое состояние называется *начальным состоянием*. Каждое событие клавиатуры заставляет автомат пересматривать свое текущее состояние; он может остаться в том же состоянии или перейти в другое. Когда программа распознает правильную последовательность событий, она переходит в *конечное состояние*.

Таблица 6. Два способа определения данных для конечного автомата

Общепринятый	Сокращенный
; ExpectsToSee.v1 -- одно из значений:	; ExpectsToSee.v2 -- одно из значений:
; -- "начало, ожидается 'a'"	; -- AA
; -- "ожидается 'b', 'c' или 'd'"	; -- BB
; -- "конец"	; -- DD
; -- "ошибка, неверная клавиша"	; -- ER (define AA "начало, ...") (define BB "ожидается ...") (define DD "конец") (define ER "ошибка, ...")

В определении данных справа используется прием именования, представленный в упражнении 61.

Для задачи распознавания последовательностей состояния обычно обозначаются буквами, которые конечный автомат ожидает увидеть дальше (см. табл. 6). Взгляните на последнее состояние, в котором говорится, что обнаружено нажатие неверной клавиши. На рис. 10 показано, как представить эти состояния, и связи между ними в виде диаграммы. Каждый узел соответствует одному из четырех состояний; каждая стрелка указывает, какое событие клавиатуры вызывает переход из одного состояния в другое.

Историческая справка. В 1950-х годах Стивен К. Клини (Stephen C. Kleene), которого мы называли бы специалистом по информатике, изобрел *регулярные выражения* для записи задач распознавания текстовых шаблонов. Нашу задачу Клини выразил бы таким регулярным выражением:

| a (b|c)* d

которое означает: «символ a, за которым могут следовать символы b и c в любых комбинациях, пока не встретится символ d». ■

6.3. Ошибки ввода

Одной из центральных тем, обсуждаемых в этой главе, является роль предикатов. Предикаты имеют решающее значение при проектировании функций, обрабатывающих смешанные данные. Смешение данных возникает естественным образом, когда в формулировке задачи упоминается много разных видов информации, но оно также возникает, когда вы передаете свои функции и программы другим. В конце концов, вы знаете и учитываете особенности своих определений данных и сигнатур функций. Однако вы можете не знать, чем занимаются ваши друзья и коллеги и как кто-то, незнакомый с языком BSL и программированием, использует ваши программы. Поэтому в этом разделе мы покажем один из способов защиты программ от недопустимого ввода.

Это самообман – ожидать, что мы всегда будем строго следовать определениям сигнатур своих функций. Вызов функции с недопустимыми данными случается даже с самыми лучшими из нас. Многие языки программирования похожи на BSL и ожидают, что программисты будут внимательно читать сигнатурные, но есть и такие, которые сами проверяют соответствие вызовов сигнатурным, пусть и за счет некоторой дополнительной сложности.

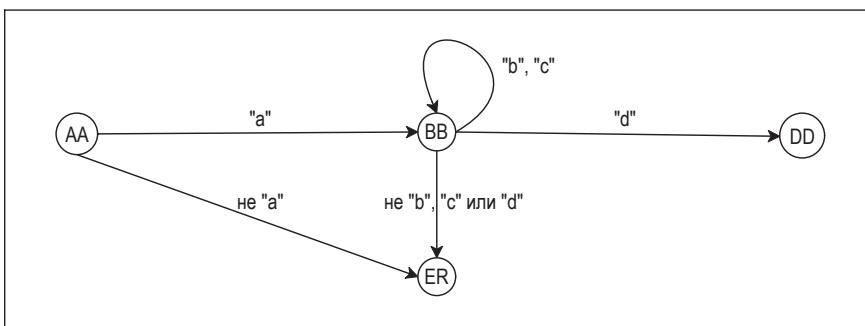


Рис. 10. Диаграмма конечного автомата

Давайте продемонстрируем этот момент с помощью простой программы. Вот функция для вычисления площади круга:

```

; Число -> Число
; вычисляет площадь круга с радиусом r
(define (area-of-disk r)
  (* 3.14 (* r r)))
  
```

Наши друзья могут решить использовать эту функцию для решения домашних заданий по геометрии. К сожалению, они могут случайно применить эту функцию к строке, а не к числу. В случае такой ошибки функция остановит выполнение программы с загадочным сообщением:

```

> (area-of-disk "my-disk")
*:expects a number as 1st argument, given "my-disk"
  
```

(*:ожидалось число в 1-м аргументе, а получено «my-disk»).

С помощью предикатов можно предотвратить появление подобных загадочных сообщений и вывести более понятное описание ошибки.

Такие версии проверяющих функций можно создавать для передачи нашим друзьям. Друзья могут не знать языка BSL, поэтому мы должны быть готовыми к тому, что они попробуют применить *проверяющую функцию* к произвольным значениям: числам, строкам, изображениям, структурам posn и т. д. Мы не можем предугадать, какие структуры будут определены в BSL, но мы знаем приблизительную форму определения данных для коллекции всех значений в языке BSL. Эта форма определения данных показана в листинге 21. Как обсуждалось в разделе 5.7, определение данных для Any (любое значение) является открытым, потому что каждое определение структуры добавляет новые экземпляры. Эти экземпляры сами могут содержать значения Any, а это подразумевает, что определение данных Any должно ссылаться на самого себя – первое время эта мысль может пугать.

Листинг 21. Вселенная данных в языке BSL

```
| ; Any -- одно из значений в языке BSL:
| ; -- Число
| ; -- Логическое значение
| ; -- Стока
| ; -- Изображение
| ; -- (make-posn Any Any)
| ; ...
| ; -- (make-tank Any Any)
| ; ...
```

С учетом этой детализации макет проверяющей функции имеет примерно следующую форму:

```
| ; Any -> ???
(define (checked-f v)
  (cond
    [(number? v) ...]
    [(boolean? v) ...]
    [(string? v) ...]
    [(image? v) ...]
    [(posn? v) (...(posn-x v) ... (posn-y v) ...) ]
    ...
    ; какой селектор понадобится в следующем условии?
    [(tank? v) ...]
    ...))
```

Конечно, невозможно перечислить все варианты из этого определения, но, к счастью, в этом нет необходимости. Мы знаем, что для всех значений, допустимых для исходной функции, ее проверяющая версия должна давать точно такие же результаты, а при получении любых других значений – сообщать об ошибке.

В частности, наша функция checked-area-of-disk принимает произвольное значение BSL и использует area-of-disk для вычисления площади круга, если аргумент является числом. В противном случае она должна остановить выполнение с сообщением об ошибке; в языке

BSL для этого используется функция `errgog`, которая принимает строку и останавливает программу:

```
| (errgog "area-of-disk: number expected")
```

Вот как могло бы выглядеть определение функции `checked-area-of-disk`:

```
(define MESSAGE "area-of-disk: number expected")
(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [(boolean? v) (error MESSAGE)]
    [(string? v) (error MESSAGE)]
    [(image? v) (error MESSAGE)]
    [(posn? v) (error MESSAGE)]
    ...
    [(tank? v) (error MESSAGE)]
    ...))
```

Использование предложения `else` поможет завершить это определение естественным образом:

```
; Any -> Число
; вычисляет площадь круга с радиусом v,
; если v является числом
(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [else (errgog "area-of-disk: number expected")]))
```

Давайте поэкспериментируем, чтобы убедиться, что у нас получилось то, что мы хотели:

```
| > (checked-area-of-disk "my-disk")
area-of-disk:number expected
```

Создавать проверяющие функции важно, если вы собираетесь передавать свои программы другим. Однако гораздо важнее уметь проектировать программы, которые работают правильно. В этой книге основное внимание уделяется процессу проектирования правильно работающих программ, и, чтобы не отвлекаться, мы будем считать, что всегда строго следуем определениям данных и сигнатур. Мы, по крайней мере, поступаем так почти всегда, но в редких случаях можем попросить вас спроектировать проверяющие версии функций или программ.

Упражнение 110. Проверяющая версия функции `area-of-disk` может также проверить, является ли аргумент функции положительным числом. Внесите соответствующие изменения в `check-area-of-disk`. ■

Упражнение 111. Взгляните на следующее определение:

```
(define-struct vec [x y])
; vec -- это
;   (make-vec ПоложительноеЧисло ПоложительноеЧисло)
; интерпретация: представляет вектор скорости
```

Напишите функцию `checked-make-vec` – проверяющую версию элементарной операции `make-vec`. Она должна гарантировать возможность создания вектора, только если ее аргументы будут положительными числами. Другими словами, `checked-make-vec` должна обеспечить соблюдение неформального определения данных. ■

Предикаты. У многих из вас наверняка давно возник вопрос: как создавать свои предикаты? В конце концов, если представить в общем виде, проверяющие функции имеют следующую форму:

```
| ; Any -> ...
| ; проверяет допустимость входного аргумента для функции g
(define (checked-g a)
  (cond
    [(XYZ? a) (g a)]
    [else (error "g: bad input")]))
```

где функция `g` определяется так:

```
| ; XYZ -> ...
| (define (g some-x) ...)
```

Мы предполагаем, что существует определение данных с именем `XYZ` и выражение `(XYZ? A)` должно возвращать `#true`, если `a` является экземпляром `XYZ`, и `#false` в противном случае.

Для функции `area-of-disk`, которая принимает число, вполне подходит предикат `number?`. Но для некоторых функций, таких как `missile-render` (см. выше), желательно определить свой предикат, потому что `MissileOrNot` – это придуманная нами, а не встроенная коллекция данных. Итак, давайте определим предикат для `MissileOrNot`.

Вспомним, как выглядит сигнатура предикатов:

```
| ; Any -> Boolean
| ; это элемент коллекции MissileOrNot?
(define (missile-or-not? a) #false)
```

Хорошей практикой считается формулировка описания назначения предиката в форме вопроса, потому что применение предиката напоминает вопрос о значении. Знак вопроса «`?`» в конце имени предиката еще больше усиливает эту идею; некоторые могут добавлять слово «да?» при произнесении имен таких функций.

Придумать примеры тоже несложно:

```
| (check-expect (missile-or-not? #false) #true)
| (check-expect (missile-or-not? (make-posn 9 2)) #true)
| (check-expect (missile-or-not? "yellow") #false)
```

Первые два примера напоминают, что экземпляр `MissileOrNot` может быть либо значением `#false`, либо экземпляром `Posn`. Третий пример утверждает, что строка не является допустимым экземпляром этой коллекции. Вот еще три теста:

```
| (check-expect (missile-or-not? #true) #false)
| (check-expect (missile-or-not? 10) #false)
| (check-expect (missile-or-not? empty-image) #false)
```

Объясните ожидаемые ответы!

Поскольку предикаты принимают любые значения BSL, их макеты подобны макетам проверяющих функций checked-f. Стоп! Найдите этот макет и еще раз внимательно рассмотрите, прежде чем читать дальше.

По аналогии с проверяющими функциями, в предикате не требуется определять все возможные условия. Достаточно добавить только те, которые могут вернуть #true:

```
(define (missile-or-not? v)
  (cond
    [(boolean? v) ...]
    [(posn? v) (... (posn-x v) ... (posn-y v) ...)]
    [else #false]))
```

Все остальные случаи обобщаются условием else, которое возвращает #false.

Согласно этому макету, в определении missile-or-not? достаточно перечислить все допустимые случаи:

```
(define (missile-or-not? v)
  (cond
    [(boolean? v) (boolean=? #false v)]
    [(posn? v) #true]
    [else #false]))
```

Коллекция MissileOrNot включает только значение #false; #true не входит в нее. Мы выразили эту идею в форме выражения (boolean=? #false v), но также можно было бы использовать выражение (false? v):

```
(define (missile-or-not? v)
  (cond
    [(false? v) #true]
    [(posn? v) #true]
    [else #false]))
```

Естественно, все экземпляры Posn тоже являются членами коллекции MissileOrNot, что объясняет #true во втором условии.

Упражнение 112. Переформулируйте предикат, используя выражение ог. ■

Упражнение 113. Спроектируйте предикаты для следующих определений данных из предыдущего раздела: SIGS, Координата (упражнение 105) и VAnimal. ■

В заключение упомянем также два важных предиката, key-event? и mouse-event?, которые наверняка пригодятся вам при создании мировых программ. Об их назначении вполне можно догадаться по именам, но вы все равно загляните в документацию с их описанием, чтобы убедиться, что понимаете, что именно они проверяют.

6.4. Проверка состояния мира

В мировой программе многое может пойти не так. Мы просто согласились с тем, что наши функции всегда применяются к надлежащему типу

данных, но мировая программа может манипулировать слишком многими разными данными одновременно, поэтому у нас не может быть абсолютного доверия самим себе. Проектируя мировую программу, которая обрабатывает такты часов, щелчки мышью, нажатия клавиш и формирует изображение, очень легко ошибиться в одном из этих взаимодействий. При этом язык BSL может не распознать ошибку немедленно. Например, одна из наших функций может вернуть результат, не являющийся экземпляром нашего представления данных для состояния мира. В такой ситуации `big-bang` примет этот экземпляр данных и сохранит его до следующего события. И только когда следующий обработчик событий получит эти несоответствующие данные, программа может завершиться с ошибкой. Хуже того, даже второй, третий и четвертый этапы обработки событий могут пропустить дальше эти несоответствующие данные, но затем, намного позже, произойдет взрыв.

Чтобы справиться с подобными проблемами, `big-bang` поддерживает дополнительное предложение `check-with`, которое принимает предикат для проверки состояния мира. Если, например, состояние мира представлено числом, мы с легкостью могли бы отразить этот факт следующим образом:

```
| (define (main s0)
|   (big-bang s0 ... [check-with number?] ...))
```

Как только какая-либо функция обработки событий вернет какое-то значение, не являющееся числом, мир остановится с соответствующим сообщением об ошибке.

Предложение `check-with` особенно полезно, когда определение данных является не простым классом со встроенным предикатом, таким как `number?`, а чем-то другим, как, например, следующее определение интервала:

```
| ; UnitWorld -- это число
| ; между 0 (включительно) и 1 (не включая его).
```

В этом случае необходимо сформулировать предикат для интервала:

```
; Any -> Логическое значение
; является ли x числом между 0 (включительно) и 1 (не включая его)?

(check-expect (between-0-and-1? "a") #false)
(check-expect (between-0-and-1? 1.2) #false)
(check-expect (between-0-and-1? 0.2) #true)
(check-expect (between-0-and-1? 0.0) #true)
(check-expect (between-0-and-1? 1.0) #false)

(define (between-0-and-1? x)
  (and (number? x) (≤ 0 x) (< x 1)))
```

С этим предикатом можно контролировать все переходы состояния в мировой программе:

```
| (define (main s0)
|   (big-bang s0
```

```

...
[check-with between-0-and-1?]
...)
```

Если какой-либо из обработчиков вернет число за пределами указанного интервала или, что еще хуже, нечисловое значение, то программа немедленно обнаружит эту ошибку и даст нам шанс исправить ее.

Упражнение 114. Используйте предикаты из упражнения 113 для проверки в мировой программе, имитирующей бой с космическими захватчиками, в программе, управляющей виртуальными питомцами (упражнение 106), и в программе редактора (раздел 5.10). ■

6.5. Предикаты равенства

Предикат равенства – это функция, сравнивающая два экземпляра из одной коллекции данных. Вспомните определение Светофор в разделе 4.3, включающее коллекцию из трех строк: "red", "green" и "yellow". Вот один из способов определить функцию `light=?:`

```

; Светофор Светофор -> Логическое значение
; сравнивает два (состояния) светофора

(check-expect (light=? "red" "red") #true)
(check-expect (light=? "red" "green") #false)
(check-expect (light=? "green" "green") #true)
(check-expect (light=? "yellow" "yellow") #true)

(define (light=? a-value another-value)
  (string=? a-value another-value))
```

После щелчка на кнопке **RUN** (Выполнить) все тесты выполняются успешно, но, к сожалению, другие взаимодействия обнаружат противоречие в наших намерениях:

```

> (light=? "salad" "greens")
#false
> (light=? "beans" 10)
string=?:expects a string as 2nd argument, given 10
```

(`string=?:`ожидается строка во 2-м аргументе, а получено значение 10).

Сравните эти взаимодействия с применением других встроенных предикатов равенства:

```

> (boolean=? "#true" 10)
boolean=?:expects a boolean as 1st argument, given "#true"
```

(`boolean=?:`ожидается логическое значение в 1-м аргументе, а получено значение «#true»).

Попробуйте выполнить выражения (`string=? 10 #true`) и (`= 20 "help"`). Все они сообщат об ошибке применения к неверному аргументу.

Проверяющая версия `light=?` предварительно проверяет, являются ли оба аргумента экземплярами коллекции Све-

Регистр символов
имеет значение; строка
"red" считается отличной от "Red" и "RED".

тофор. Если нет, то она сообщает об ошибке, подобной встроенным предикатам равенства. Вот определение предиката `light?:`:

```
; Any -> Логическое значение
; является ли данный экземпляр Светофором?
(define (light? x)
  (cond
    [(string? x) (or (string=? "red" x)
                      (string=? "green" x)
                      (string=? "yellow" x))]
    [else #false]))
```

Теперь можно завершить предикат `light=?`, просто следуя первонаучальному анализу. Функция проверяет принадлежность аргументов к коллекции значений Светофор, и если это условие нарушается, то применяет функцию `error`, чтобы сообщить об ошибке:

```
(define MESSAGE
  "traffic light expected, given some other value")

; Any Any -> Логическое значение
; являются ли аргументы значениями Светофор, и если да,
; то являются ли они одинаковыми?

(check-expect (light=? "red" "red") #true)
(check-expect (light=? "red" "green") #false)
(check-expect (light=? "green" "green") #true)
(check-expect (light=? "yellow" "yellow") #true)

(define (light=? a-value another-value)
  (if (and (light? a-value) (light? another-value))
      (string=? a-value another-value)
      (error MESSAGE)))
```

Упражнение 115. Исправьте предикат `light=?` так, чтобы он сообщал, какой из двух аргументов не является экземпляром коллекции Светофор. ■

Трудно представить, что ваши программы будут использовать `light=?`, скорее, они будут использовать `key=?` и `mouse=?`, два предиката равенства, упоминавшихся в конце предыдущего раздела. Естественно, `key=?` – предикат, сравнивающий два события клавиатуры; аналогично `mouse=?` сравнивает два события мыши. Оба типа событий представлены строками, однако важно понимать, что не все строки представляют события клавиатуры или мыши.

Мы рекомендуем использовать `key=?` в обработчиках событий клавиатуры и `mouse=?` в обработчиках событий мыши. Использование `key=?` в обработчике событий клавиатуры гарантирует, что функция действительно будет сравнивать строки, представляющие события клавиатуры, а не какие-то другие. Как только, например, функция случайно получит в аргументе строку `"hello\n world"`, предикат `key=?` тут же заметит ошибку и сообщит нам об этом.

7. Итоги

В этой первой части книги мы исследовали ряд простых, но важных уроков. Вот краткое их изложение.

1. Хороший программист проектирует программы. Плохой программист движется вперед методом проб и ошибок, пока не будет создана видимость, что программа работает.
2. Рецепт проектирования имеет два измерения. С одной стороны, он описывает процесс проектирования, то есть последовательность выполняемых шагов. С другой – объясняет, как выбранное представление данных влияет на процесс проектирования.
3. Каждая хорошо спроектированная программа состоит из множества определений констант, структур, данных и функций. В пакетных программах одна функция является «главной», и обычно она состоит из нескольких других функций, выполняющих вычисления. В интерактивных программах роль главной функции играет функция `big-bang`; она определяет начальное состояние программы, функцию вывода изображения и до трех обработчиков событий: тактов часов, щелчков мышью и нажатий клавиш на клавиатуре. В программах обоих типов функции определяются «сверху вниз», начиная с главной функции, за которой следуют функции, вызываемые в главной функции, и т. д.
4. Подобно многим другим языкам программирования, язык для начинающих студентов (Beginning Student Language, BSL) имеет **словарь и грамматику**. Программисты должны уметь определять значение каждого предложения на языке, чтобы предсказать, что программа получит в результате вычислений. Следующее интермеццо подробно объясняет эту идею.
5. Языки программирования, включая BSL, поставляются с богатым набором библиотек, чтобы программистам не приходилось постоянно изобретать велосипед. Программист должен привыкнуть к таким библиотечным функциям, особенно к их сигнатурам и описаниям назначений. Это упростит жизнь.
6. Программист должен знать «инструменты», которые может предложить выбранный язык программирования. Эти инструменты либо являются частью языка, как, например, `cond` или `max`, либо «импортируются» из библиотек. В этом смысле убедитесь, что понимаете следующие термины: **определение типа структуры, определение функции, определение константы, экземпляр структуры, определение данных, big-bang и функция обработки событий**.

Интермеццо 1. Язык для начинающих студентов

В части I этой книги BSL рассматривается как естественный язык. В ней были представлены «основные слова» языка, показано, как составлять «предложения» из «слов» и как использовать знания алгебры для «чтения» этих «предложений». Такое введение помогает понять язык до некоторой степени, но чтобы освоить его по-настоящему, необходимо формальное изучение.

Во многих отношениях аналогии, приводившиеся в первой части, верны. В языке программирования есть словарь и грамматика, хотя программисты эти элементы языка обычно называют *синтаксисом*. Предложение в языке BSL – это выражение или определение. Грамматика BSL определяет правила формирования этих фраз. Но не все грамматически верные предложения на естественном языке или языке программирования имеют смысл. Например, фраза на естественном языке «кошка свернулась клубком» имеет определенный смысл, но фраза «кирпич – это автомобиль» не имеет смысла, хотя и является грамматически верной. Чтобы определить, имеет ли предложение смысл, необходимо знать *значение языка*; программисты называют это *семантикой*.

В этом интермеццо мы рассмотрим язык BSL как расширение знакомого нам языка арифметики и алгебры. В конце концов, вычисления начинаются с этой формы простой математики, и мы должны понимать связь между законами математики и вычислениями. Первые три раздела описывают синтаксис и семантику значительной части BSL. А в четвертой, опираясь на эти новые знания BSL, возобновляется обсуждение ошибок. Остальные разделы заполняют оставшиеся пробелы, и в последнем обсуждаются инструменты, предназначенные для определения тестов.

Словарь BSL

Врезке ниже представлен базовый словарь языка BSL. Он состоит из литературных констант, таких как числа или логические значения; имен, имеющих значение в языке BSL, например `cond` или `+`, и имен, которым программы могут придавать значение с помощью `define` или параметров функции.

Каждый пункт списка определяет множество посредством детализации его элементов. Можно, конечно, указать эти коллекции целиком, но мы считаем это излишним и доверяем вашей интуиции. Просто имейте в виду, что каждое из этих множеств может включать некоторые дополнительные элементы.

Базовый словарь BSL

Имя или переменная – это последовательность любых символов, кроме следующих: " , ' ` () [] { } | ; #:

- примитив – это имя, изначально имеющее смысл в языке BSL, например + или sqrt;
- переменная – это имя, не имеющее предопределенного смысла.

Значение – это одно из следующих:

- число – одно из: 1, -1, 3/5, 1.22, #i1.22, 0+1i и т. д. Синтаксис чисел в языке BSL довольно сложный, потому что охватывает множество форматов представления чисел: положительные и отрицательные числа, дроби натуральные и десятичные, точные и приблизительные числовые значения, вещественные и комплексные числа, числа в системах счисления с основанием, отличным от 10, и многие другие. Понимание форм записи чисел требует глубокого понимания грамматики и особенностей синтаксического анализа, обсуждение которых выходит за рамки этого интермешко;
- логическое значение – всего два: #true и #false;
- строка – одна из: "", "he says \"hello world\" to you", "doll" и т. д. Как правило, строка – это последовательность символов, заключенная в пару двойных кавычек ";
- изображение – это изображение в формате png, jpg, tiff и многих других форматах. Мы намеренно опускаем точное определение изображения.

«Любое допустимое значение» мы обычно обозначаем как v, v-1, v-2 и т. д.

Грамматика BSL

Врезке «Базовая грамматика BSL» показана большая часть грамматики языка BSL, которая отличается удивительной простотой, по сравнению с другими языками. Что касается выразительной силы BSL, то пусть внешняя простота не обманывает вас. Однако прежде всего мы должны обсудить правила чтения грамматики. Каждая строка со знаком «равно» (=) представляет *синтаксическую категорию*, сам знак = можно произнести как «один из», а знак вертикальной черты (|) – как «или». Многоточия обозначают любое количество повторений того, что предшествует многоточию. Например, определение программа означает, что программа не содержит ничего, или содержит одно определение-выражения, или содержит последовательность из двух, трех, четырех, пяти или более определений выражений. Поскольку

При чтении вслух грамматика звучит как определение данных. Грамматику можно использовать для записи многих определений данных.

этот пример не особо информативен, рассмотрим вторую синтаксическую категорию. Она утверждает, что определение – это либо

| (define (переменная переменная) выражение)

потому что «любое количество повторений» подразумевает ноль повторений, либо

| (define (переменная переменная переменная) выражение)

с одним повторением, либо

| (define (переменная переменная переменная переменная) выражение)

с двумя повторениями.

Базовая грамматика BSL

программа = определение-или-выражение ...

определение-или-выражение = определение
| выражение

определение = (define (переменная переменная переменная ...) выражение)

выражение = переменная
| значение
| (примитив выражение выражение ...)
| (переменная выражение выражение ...)
| (cond [выражение выражение] ... [выражение выражение])
| (cond [выражение выражение] ... [else выражение])

И наконец, следует отметить три «слова», выделенных ненаклонным шрифтом: *define*, *cond* и *else*. Согласно определению словаря BSL, эти три слова являются именами. Но мы не упомянули, что эти имена имеют предопределенное значение. В языке BSL эти слова служат маркерами, которые отделяют одни составные предложения от других, и в знак признания их роли такие слова называются *ключевыми словами*.

Теперь мы готовы сформулировать цель грамматики. Грамматика языка программирования диктует правила составления предложений на основе словаря. Некоторые предложения – это просто элементы словаря. Например, как описывается во врезке «Базовая грамматика BSL», 42 – это предложение на языке BSL:

Программа в DrRacket
в действительности
состоит из двух
отдельных частей:
области определений
и выражений в области
взаимодействий.

- первая синтаксическая категория говорит, что программа – это определение-выражения. Выражения могут ссылаться на определения;
- вторая синтаксическая категория говорит, что определение-выражения является либо определением, либо выражением;

- последнее определение перечисляет все способы формирования выражения, и второе из них – это значение.

Врезка «Базовая грамматика BSL» утверждает, что 42 – это значение, подтверждая последний пункт.

Определение грамматики показывает, как составлять сложные предложения, построенные из других предложений. Например, определение сообщает, что определение функции начинается с символа «(», за которым следуют: ключевое слово `define`, еще один символ «(», последовательность, по крайней мере, из двух переменных, символ «)», выражение и, наконец, заключительная закрывающая круглая скобка «)», которая соответствует самой первой открывающей круглой скобке. Обратите внимание, как ведущее ключевое слово `define` отделяет определение от выражений.

Выражения бывают шести видов: переменные, константы, примитивы, функции и две разновидности условных выражений `cond`. Первые два – это атомарные предложения, последние четыре – составные. Так же как `define`, ключевое слово `cond` отделяет условные выражения от всего остального.

Вот три примера выражений: "all", `x` и `(f x)`. Первый пример принадлежит к классу строк и, следовательно, является выражением. Второй пример – это переменная, а всякая переменная – это выражение. Третий пример – это применение функции, потому что `f` и `x` – это переменные.

Напротив, следующие предложения, заключенные в скобки, не являются допустимыми выражениями: `(f define)`, `(cond x)` и `((f 2) 10)`. Первое частично соответствует форме применения функции, но использует ключевое слово `define`, как если бы оно было переменной. Второе предложение тоже не является допустимым выражением `cond`, потому что содержит переменную во втором элементе, а не пару выражений, заключенных в круглые скобки. Последнее предложение не является ни условным выражением, ни применением функции, потому что первая часть является выражением.

Наконец, можно заметить, что в грамматике не упоминаются пробельные символы: пробелы, табуляции и переводы строки. BSL – достаточно вольный язык. Если между элементами любой последовательности в программе имеются пробелы, DrRacket будет понимать ваши программы на BSL. Однако хорошим программистам может не понравиться то, что вы пишете. Такие программисты используют пробелы, чтобы упростить чтение и понимание программ. Что особенно важно, они предпочитают стиль, более удобный для людей, чем для программ-трансляторов, обрабатывающих исходный код программ (таких как DrRacket). Они осваивают этот стиль, читая примеры кода в книгах и обращая внимание на их форматирование.

Имейте в виду, что ваши программы на BSL будут изучаться двумя категориями читателей: людьми и средой программирования DrRacket.

Упражнение 116. Взгляните на следующие предложения:

1. x
2. (= y z)
3. (= (= y z) 0)

Объясните, почему они считаются синтаксически допустимыми выражениями. ■

Упражнение 117. Взгляните на следующие предложения:

1. (3 + 4)
2. number?
3. (x)

Объясните, почему они считаются синтаксически неверными выражениями. ■

Упражнение 118. Взгляните на следующие предложения:

1. (define (f x) x)
2. (define (f x) y)
3. (define (f x y) 3)

Объясните, почему они считаются синтаксически допустимыми определениями. ■

Упражнение 119. Взгляните на следующие предложения:

1. (define (f "x") x)
2. (define (f x y z) (x))

Объясните, почему они считаются синтаксически неверными определениями. ■

Упражнение 120. Определите, какие из следующих предложений являются допустимыми, а какие – нет:

1. (x)
2. (+ 1 (not x))
3. (+ 1 2 3)

Объясните, почему те или иные предложения являются допустимыми или неверными. Определите категорию – выражение или определение – допустимых предложений. ■

ПРИМЕЧАНИЕ О ГРАММАТИЧЕСКОЙ ТЕРМИНОЛОГИИ. Компоненты составных предложений имеют имена. Мы ввели некоторые из этих имен неофициально. В листинге 22 перечислены основные условные обозначения.

Листинг 22. Синтаксические соглашения о терминологии

```

; применение функции:
(функция аргумент ... аргумент)

; определение функции:
(define (имя-функции параметр ... параметр)
  тело-функции)

; условное выражение:
(cond
  )

```

```

условное-предложение
...
условное-предложение)

; условное-предложение
[условие ответ]

```

В дополнение к терминам в листинге 22 мы также используем термин заголовок функции для обозначения второго компонента определения. Соответственно, компонент выражение называется *телом функции*. Люди, которые рассматривают языки программирования с точки зрения математики, заголовок называют *левой частью*, а тело – *правой частью*. Иногда также можно услышать или увидеть термин *фактические аргументы*, обозначающий аргументы в выражении применения функции. **КОНЕЦ**.

Значение в языке BSL

Нажимая клавишу **Enter** на клавиатуре, вы фактически просите DrRacket вычислить выражение. В процессе вычислений DrRacket использует законы арифметики и алгебры, чтобы получить значение. Во врезке «Базовый словарь BSL» определяется, что значение (точнее множество значений) – это просто подмножество всех выражений. В множество входят логические значения, строки и изображения.

Правила вычислений делятся на две категории. Бесконечное количество правил, таких как правила арифметики, объясняет, как определить значение (результат) применения элементарной операции к значениям:

```

(+ 1 1) == 2
(- 2 1) == 1
...

```

Как уже отмечалось, пара символов `==` говорит нам, что два выражения равны, согласно законам вычислений в BSL. Но арифметика BSL является более универсальной, чем простые вычисления с числами. Она также включает правила вычислений с логическими значениями, строками и т. д.:

```

(not #true)      == #false
(string=? "a" "a") == #true
...

```

И так же как в алгебре, равенство всегда можно заменить тождественными преобразованиями; как показано в листинге 23.

Листинг 23. Замена равенства тождественными преобразованиями

```

(boolean? (= (string-length (string-append "h" "w"))
              (+ 1 3)))
==

```

```
(boolean? (= (string-length (string-append "h" "w")) 4))
==
(boolean? (= (string-length "hw") 4))
==
(boolean? (= 2 4))
==
(boolean? #false)
== #true
```

Кроме того, важно знать правила из алгебры, чтобы понимать порядок применения функции к аргументам. Предположим, программа содержит такое определение:

```
(define (f x-1 ... x-n)
  f-body)
```

Тогда применение функции описывается правилом:

```
(f v-1 ... v-n) == f-body
; где все вхождения x-1 ... x-n
; заменяются v-1 ... v-n соответственно
```

Более подробно это правило рассматривается в разделе 17.2. Традиционно в таких языках, как BSL, мы называем это правило правилом *бета* или *бета-значения*.

Это правило сформулировано максимально обобщенно, поэтому лучше взглянуть на конкретный пример. Допустим, у нас есть такое определение:

```
(define (poly x y)
  (+ (expt 2 x) y))
```

и в области взаимодействий мы ввели выражение (*poly* 3 5). На первом этапе вычислений DrRacket применит правило бета:

```
| (poly 3 5) == (+ (expt 2 3) 5) ... == (+ 8 5) == 13
```

Помимо правила бета, нам также нужны правила, определяющие значения выражений *cond*. Эти правила являются алгебраическими, даже притом что они не изучаются в школьном курсе алгебры. Если первое условие оценивается как *#false*, то оно отбрасывается, а остальные условия остаются нетронутыми:

```
(cond
  [#false ...]
  [условие2 ответ2]
  ...)
== (cond
      ; первая строка отбрасывается
      [условие2 ответ2]
      ...)
```

Это правило имеет имя *cond_{false}*. А вот пример правила *cond_{true}*:

```
(cond
  [#true ответ-1]
  [условие2 ответ2]
  ...)
== ответ-1
```

Это правило применяется также в случаях, когда первое условие – *else*.

Рассмотрим следующий пример:

```
(cond
  [(zero? 3) 1]
  [(= 3 3) (+ 1 1)]
  [else 3])
== ; согласно правилам арифметики и замены равного равным
(cond
  [#false 1]
  [(= 3 3) (+ 1 1)]
  [else 3])
== ; согласно правилу condfalse
(cond
  [= 3 3) (+ 1 1)]
  [else 3])
== ; согласно правилам арифметики и замены равного равным
(cond
  [#true (+ 1 1)]
  [else 3])
== ; согласно правилу condtrue
(+ 1 1)
```

Эти вычисления иллюстрируют применение арифметических правил и обоих правил cond.

Упражнение 121. Вычислите следующие выражения по шагам:

1. $(+ (* (/ 12 8) 2/3) (- 20 (\sqrt{4})))$
2. $(\text{cond} [(= 0 0) \#false] [(> 0 1) (\text{string=?} "a" "a")) [\text{else} (= (/ 1 0) 9)])$
3. $(\text{cond} [(= 2 0) \#false] [(> 2 1) (\text{string=?} "a" "a")) [\text{else} (= (/ 1 2) 9)])$

Используйте движок пошаговых вычислений в DrRacket, чтобы проверить свои рассуждения. ■

Упражнение 122. Пусть программа содержит следующее определение:

```
(define (f x y)
  (+ (* 3 x) (* y y)))
```

Покажите по шагам, как DrRacket вычислит следующие выражения:

1. $(+ (f 1 2) (f 2 1))$
2. $(f 1 (* 2 3))$
3. $(f (f 1 (* 2 3)) 19)$

Используйте движок пошаговых вычислений в DrRacket, чтобы проверить свои рассуждения. ■

Значения и вычисления

Специалисты называют движок пошаговых вычислений в DrRacket моделью вычислений.

В главе 21 будет представлена еще одна модель – *интерпретатор*.

Движок пошаговых вычислений в DrRacket имитирует действия ученика, изучающего алгебру. В отличие от вас, этот механизм прекрасно справляется с применением законов арифметики и алгебры, представленных выше.

Вы можете и должны использовать этот механизм, когда затрудняетесь определить, как работает новая языковая конструкция. В разделах «Вычисления» предлагаются специальные упражнения для этого, но вы можете составить свои примеры, пропустить их через движок пошаговых вычислений и попытаться понять, почему он выполняет те или иные шаги.

Наконец, движок пошаговых вычислений можно использовать, увидев результат программы, который может показаться вам ошибочным. Эффективное использование движка пошаговых вычислений для этой цели требует практики. Например, проводя подобные исследования, часто приходится копировать блоки программы и удалять ненужные части. Но, освоив движок пошаговых вычислений, вы обнаружите, что он помогает четко объяснить ошибки времени выполнения и логические ошибки в ваших программах.

Ошибки в BSL

Практически полный список сообщений об ошибках приводится в последнем разделе этого интермецо.

Когда DrRacket обнаруживает, что какая-то фраза в круглых скобках не принадлежит языку BSL, он сообщает о *синтаксической ошибке*. Чтобы определить, является ли синтаксически допустимой программа, заключенная в скобки, DrRacket использует грамматику, представленную во врезке «Базовая грамматика BSL» выше, и рассуждает в соответствии с правилами, описанными выше. Однако не все синтаксически допустимые программы являются осмысленными.

Когда DrRacket выполняет синтаксически допустимую программу и обнаруживает, что какая-то операция применяется к значению неправильного типа, возникает *ошибка времени выполнения*. Рассмотрим синтаксически допустимое выражение $(/ 1 0)$, которое, как вы знаете из математики, не имеет значения. Поскольку все вычисления в BSL основываются на правилах математики, DrRacket сообщает об ошибке:

```
| > (/ 1 0)
|/:division by zero (/:деление на ноль)
```

Естественно, что та же ошибка будет обнаружена, даже если выражение, такое как $(/ 1 0)$, будет глубоко вложено в другое выражение:

```
| > (+ (* 20 2) (/ 1 (- 10 10)))
|/:division by zero (/: деление на ноль)
```

Поведение DrRacket отражается в наших вычислениях, как описывается далее. Когда обнаруживается выражение, не имеющее значения, и правила вычислений не допускают дальнейшего упрощения, мы говорим, что вычисления *застопорились*. Такое застопоривание соответствует ошибке времени выполнения. Например, вычисление выражения, приведенного выше, вызывает застопоривание:

```
(+ (* 20 2) (/ 1 (- 10 10)))
==
(+ (* 20 2) (/ 1 0))
==
(+ 40 (/ 1 0))
```

Эта последовательность вычислений также показывает, что DrRacket исключает контекст застопорившегося выражения, потому что оно сигнализирует об ошибке. В этом конкретном примере исключается сложение числа 40 с застопорившимся выражением ($/ 1 0$).

Не все вложенные застопорившиеся выражения сигнализируют об ошибках. Допустим, программа содержит следующее определение:

```
(define (my-divide n)
  (cond
    [(= n 0) "inf"]
    [else (/ 1 n)]))
```

Если применить функцию `my-divide` к числу 0, то DrRacket выполнит следующую последовательность вычислений:

```
(my-divide 0)
==
(cond
  [(= 0 0) "inf"]
  [else (/ 1 0)])
```

В данной ситуации было бы неправильно говорить, что функция сигнализирует об ошибке деления на ноль, даже притом, что выделенное подвыражение позволяет предположить это. Причина в том, что выражение $(= 0 0)$ вернет #true, поэтому второе условие `cond` не играет никакой роли:

```
(my-divide 0)
==
(cond
  [(= 0 0) "inf"]
  [else (/ 1 0)])
 ==
(cond
  [#true "inf"]
  [else (/ 1 0)])
 == "inf"
```

К счастью, наши законы вычислений автоматически решают подобные ситуации. Мы просто должны помнить, когда они применяются. Например, в выражении

```
| (+ (* 20 2) (/ 20 2))
```

сложение не будет выполнено до умножения или деления. Аналогично выделенное деление в

```
| (cond
|   [(= 0 0) "inf"]
|   [else (/ 1 0)])
```

не заменит полное выражение cond, пока соответствующая строка не станет первым условием.

В любых случаях следует руководствоваться следующим правилом:

Всегда выбирайте самое внешнее и крайнее левое вложенное выражение, готовое к вычислению.

Оно может показаться несколько упрощенным, но всегда объясняет результаты BSL.

Часто программисты также предпочитают знать, в какой функции произошла ошибка. Вспомните проверяющую версию area-of-disk из раздела 6.3:

```
| (define (checked-area-of-disk v)
|   (cond
|     [(number? v) (area-of-disk v)]
|     [else (error "number expected")]))
```

Теперь представьте попытку применения checked-area-of-disk к строке:

```
| (- (checked-area-of-disk "a")
|   (checked-area-of-disk 10))
== 
| (- (cond
|   [(number? "a") (area-of-disk "a")]
|   [else (error "number expected")])
|   (checked-area-of-disk 10))
== 
| (- (cond
|   [#false (area-of-disk "a")]
|   [else (error "number expected")])
|   (checked-area-of-disk 10))
== 
| (- (error "number expected"))
|   (checked-area-of-disk 10))
```

В этой точке можно попытаться вычислить второе выражение, но даже притом что оно вернет результат, примерно равный 314, ваши вычисления все равно столкнутся с выражением error, которое подобно застопорившемуся выражению. Проще говоря, вычисления закончатся на:

```
| (error "number expected")
```

Логические выражения

В нашем текущем определении языка BSL отсутствуют выражения `ог` и `and`. Их добавление может служить примером, как изучать новые языковые конструкции. Сначала нужно понять их синтаксис, а затем семантику.

Вот дополненная грамматика выражений:

```
выражение = ...
| (and выражение выражение)
| (ог выражение выражение)
```

Грамматика утверждает, что `and` и `ог` являются ключевыми словами, за каждым из которых следуют два выражения. Это не применение функции.

Чтобы понять, почему `and` и `ог` в языке BSL не являются функциями, рассмотрим практику их использования. Предположим, нам нужно сформулировать условие, которое определяет равенство выражения (`/ 1 n`) и значения `r`:

```
(define (check n r)
  (and (not (= n 0)) (= (/ 1 n) r)))
```

В данном примере проверка сформулирована через выражение `and`, чтобы избежать ошибки деления на 0. Теперь применим `check` к 0 и 1/5:

```
(check 0 1/5)
== (and (not (= 0 0)) (= (/ 1 0) 1/5))
```

Если бы выражение `and` было обычной операцией, то мы должны были бы вычислить оба подвыражения и в результате столкнулись бы с ошибкой. Однако `and` просто не вычислит второе выражение, если первое вернет `#false`. Проще говоря, `and` выполняет вычисления по короткой схеме.

Теперь вы без труда смогли бы сами сформулировать правила вычисления для выражений `and` и `ог`. Вот еще один способ объяснить их суть – преобразовать их в другие выражения:

```
| (and выражение-1 выражение-2)
```

– это краткая форма записи для

```
(cond
  [выражение-1 выражение-2]
  [else #false])
```

и

```
| (ог выражение-1 выражение-2)
```

Чтобы гарантировать, что выражение-2 вычисляет логическое значение, в этих сокращениях стоило использовать (`if выражение-2 #true #false`) вместо простого выражение-2. Мы тут немного приукрасили.

– это краткая форма записи для

```
| cond
  [выражение-1 #true]
  [else выражение-2])
```

То есть если у вас возникнут сомнения в том, как вычислить выражение `and` или `or`, используйте приведенные выше эквиваленты. Но мы надеемся, что вы будете прекрасно понимать эти операции на интуитивном уровне, а этого почти всегда достаточно.

Упражнение 123. Использование `if` могло удивить вас, потому что в этом интермеццо такая форма условия больше нигде не упоминается. Может создаться впечатление, что интермеццо дает объяснения в форме, не имеющей объяснения. На данный момент мы полагаемся на ваше интуитивное понимание `if` как сокращения для `cond`. Напишите правило, показывающее, как переформулировать

```
| (if проверяемое-выражение выражение-тогда выражение-иначе)
```

в выражение `cond`. ■

Определения констант

Программы включают не только определения функций, но и определения констант, но константы не были включены в нашу первую грамматику. Итак, вот дополненная грамматика, включающая определения констант:

```
| определение = ...
| (define имя выражение)
```

Как оказывается, в DrRacket имеется другой способ определения функций; см. главу 17.

По своей форме определение константы подобно определению функции. Ключевое слово `define` отличает определения констант от выражений, но не отличает от определений функций. Чтобы понять, что определяется – функция или константа, читатель кода должен взглянуть на вторую часть определения.

Теперь давайте разберемся с особенностями определения констант. В определении константы с литералом в правой части, например

```
| (define RADIUS 5)
```

переменная – это просто сокращенное обозначение значения выражения. Когда в процессе вычислений среда программирования DrRacket встретит ссылку на `RADIUS`, она заменит ее числом 5.

Если в правой части определения указано выражение, например

```
| (define DIAMETER (* 2 RADIUS))
```

то это выражение должно быть вычислено немедленно. В таких выражениях допускается использовать любые предшествующие определения. То есть последовательность определений

```
(define RADIUS 5)
(define DIAMETER (* 2 RADIUS))
```

эквивалентна

```
(define RADIUS 5)
(define DIAMETER 10)
```

Это правило выполняется, даже если в определениях констант используются определения функций:

```
(define RADIUS 10)
(define DIAMETER (* 2 RADIUS))
(define (area r) (* 3.14 (* r r)))
(define AREA-OF-RADIUS (area RADIUS))
```

По мере оценки этой последовательности определений DrRacket сначала выяснит, что RADIUS имеет значение 10, DIAMETER – 20, а area – это имя функции. Наконец, будет вычислено выражение (area RADIUS) и полученное значение 314 связано с именем AREA-OF-RADIUS.

Возможность смешивания определений констант и функций также вводит новый вид ошибок времени выполнения. Взгляните на следующую программу:

```
(define RADIUS 10)
(define DIAMETER (* 2 RADIUS))
(define AREA-OF-RADIUS (area RADIUS))
(define (area r) (* 3.14 (* r r)))
```

Она похожа на предыдущую, но в ней два последних определения поменялись местами. Оценка первых двух определений будет выполнена так же, как прежде. Но при попытке оценить третье определение процесс пойдет по другому пути. Чтобы оценить третье определение, необходимо вычислить выражение (area RADIUS). Определение RADIUS предшествует этому выражению, но определение area еще не встречалось. При оценке программы в DrRacket вы получите сообщение об ошибке, поясняющее, что «эта функция не определена». Поэтому будьте внимательны и используйте функции в определениях констант, только если известно, что они уже определены.

Упражнение 124. Оцените следующую программу, шаг за шагом:

```
(define PRICE 5)
(define SALES-TAX (* 0.08 PRICE))
(define TOTAL (+ PRICE SALES-TAX))
```

Возникнет ли ошибка при оценке следующей программы?

```
(define COLD-F 32)
(define COLD-C (fahrenheit->celsius COLD-F))
(define (fahrenheit->celsius f)
  (* 5/9 (- f 32)))
```

А такой?

```
(define LEFT -100)
```

```
(define RIGHT 100)
(define (f x) (+ (* 5 (expt x 2)) 10))
(define f@LEFT (f LEFT))
(define f@RIGHT (f RIGHT))
```

Проверьте свои рассуждения с помощью движка пошаговых вычислений в DrRacket. ■

Определения структур

Как вы уже знаете, `define-struct` – самая сложная конструкция в BSL. Поэтому мы оставили ее объяснение напоследок. Вот ее грамматика:

```
определение = ...
| (define-struct имя [имя ...])
```

Определение структур – это третья форма определений. От определений констант и функций определение структур отличается ключевым словом.

Вот простой пример:

```
| (define-struct point [x y z])
```

Здесь `point`, `x`, `y` и `z` являются переменными, а круглые скобки помещены в соответствии с шаблоном грамматики, это правильное определение типа структуры. Напротив, следующие два предложения в круглых скобках:

```
| (define-struct [point x y z])
| (define-struct point x y z)
```

не являются допустимыми определениями, потому что за `define-struct` не следует одно имя переменной и последовательность имен переменных в скобках.

Синтаксис `define-struct` прост, но его трудно объяснить с помощью правил оценки. Как уже упоминалось несколько раз, определение `define-struct` определяет сразу несколько функций: конструктор, несколько селекторов и предикат. То есть определение

```
| (define-struct c [s-1 ... s-n])
```

вводит в программу следующие функции:

- 1) `make-c`: конструктор;
- 2) `c-s-1... c-s-n`: последовательность селекторов;
- 3) `c?:` предикат.

Эти функции имеют тот же статус, что и `+`, `-` или `*`. Однако, прежде чем переходить к правилам, управляющим созданием этих новых функций, мы должны вернуться к определению значений. В конце концов, одна из целей `define-struct` – представление класса значений, отличных от всех существующих значений.

Проще говоря, `define-struct` расширяет вселенную значений, добавляя в нее структуры, которые объединяют несколько значений в одно. Когда программа включает определение `define-struct`, оценка значений в ней изменяется:

Значение – это: число, логическое значение, строка, изображение

- или значение структурного типа:

```
(make-c _value-1 ... _value-n)
```

где предполагается, что структура с определена.

Например, определение `point` добавляет значения, имеющие следующую форму:

```
(make-point 1 2 -1)
(make-point "one" "hello" "world")
(make-point 1 "one" (make-point 1 2 -1))
...
...
```

Теперь можно переходить к правилам оценки новых функций. Если `c-s-1` применяется к структуре `c`, то она возвращает первый компонент значения. Точно так же второй селектор извлекает второй компонент, третий селектор – третий компонент и т. д. Связь между конструктором новых данных и селекторами лучше всего охарактеризовать с помощью n уравнений, добавляемых к правилам BSL:

```
(c-s-1 (make-c V-1 ... V-n)) == V-1
(c-s-n (make-c V-1 ... V-n)) == V-n
```

Для нашего текущего примера получаем конкретные уравнения:

```
(point-x (make-point V U W)) == V
(point-y (make-point V U W)) == U
(point-z (make-point V U W)) == W
```

Встретив выражение `(point-y (make-point 3 4 5))`, DrRacket заменит это выражение значением 4, а выражение `(point-x (make-point (make-point 1 2 3) 4 5))` будет оценено как `(make-point 1 2 3)`.

Предикат `c?` может применяться к любому значению. Он вернет `#true`, если значение является структурой `c`, и `#false` в противном случае. Оба этих условия можно преобразовать в два уравнения:

```
(c? (make-c V-1 ... V-n)) == #true
(c? V) == #false
```

где `V` не является значением, сконструированным с помощью `make-c`. И снова уравнения будет проще понять, если выразить их в терминах нашего примера:

```
(point? (make-point U V W)) == #true
(point? X) == #false
```

где `X` является значением, но не является экземпляром структуры `point`.

Упражнение 125. Определите, какие из следующих предложений являются допустимыми:

1. (define-struct oops [])
2. (define-struct child [parents dob date])
3. (define-struct (child person) [dob date])

Объясните, почему то или иное предложение является допустимым или недопустимым. ■

Упражнение 126. Определите значения следующих выражений, исходя из предположения, что в области определений объявлены следующие структуры:

```
| (define-struct point [x y z])
| (define-struct none [])
```

1. (make-point 1 2 3)
2. (make-point (make-point 1 2 3) 4 5)
3. (make-point (+ 1 2) 3 4)
4. (make-none)
5. (make-point (point-x (make-point 1 2 3)) 4 5)

Объясните, почему то или иное выражение имеет или не имеет значения. ■

Упражнение 127. Предположим, что программа содержит следующее определение:

```
| (define-struct ball [x y speed-x speed-y])
```

Предскажите результаты вычисления следующих выражений:

1. (number? (make-ball 1 2 3 4))
2. (ball-speed-y (make-ball (+ 1 2) (+ 3 3) 2 3))
3. (ball-y (make-ball (+ 1 2) (+ 3 3) 2 3))
4. (ball-x (make-posn 1 2))
5. (ball-speed-y 5)

Проверьте свои предсказания, вычислив эти выражения в области взаимодействий и с использованием движка пошаговых вычислений. ■

Тесты в BSL

Во врезке «Полная грамматика BSL» представлена полная грамматика языка BSL плюс несколько форм тестирования.

Общий смысл тестовых выражений легко объяснить. После щелчка на кнопке **RUN** (Выполнить) DrRacket соберет все тестовые выражения и переместит их в конец программы, сохраняя порядок их определения. Затем будет выполнено содержимое области определений. Каждый тест вычислит свои части, а затем сравнит их с ожидаемым результатом с помощью некоторого предиката. Помимо этого, тесты

взаимодействуют с DrRacket, способствуя сбору некоторой статистики и информации об ошибках тестирования.

Упражнение 128. Скопируйте следующие тесты в область определений DrRacket:

```
(check-member-of "green" "red" "yellow" "grey")
(check-within (make-posn #i1.0 #i1.1)
              (make-posn #i0.9 #i1.2) 0.01)
(check-range #i0.9 #i0.6 #i0.8)
(check-random (make-posn (random 3) (random 9))
              (make-posn (random 9) (random 3)))
(check-satisfied 4 odd?)
```

Убедитесь, что все они терпят неудачу, и объясните, почему. ■

Полная грамматика BSL

определение-выражения = определение
 | выражение
 | тест

определение = (define (имя переменная переменная ...) выражение)
 | (define имя выражение)
 | (define-struct имя [имя ...])

выражение = (имя выражение выражение ...)
 | (cond [выражение выражение] ... [выражение выражение])
 | (cond [выражение выражение] ... [else выражение])
 | (and выражение выражение выражение ...)
 | (or выражение выражение выражение ...)
 | имя
 | число
 | строка
 | изображение

тест = (check-expect выражение выражение)
 | (check-within выражение выражение выражение)
 | (check-member-of выражение выражение ...)
 | (check-range выражение выражение выражение)
 | (check-eq? выражение)
 | (check-random выражение выражение)
 | (check-satisfied выражение имя)

Сообщения об ошибках в BSL

Программы на BSL могут сигнализировать о многих видах синтаксических ошибок. Мы разрабатывали BSL и его систему сообщений об

ошибках специально для новичков, которые неизбежно допускают ошибки, и все же сообщения об ошибках требуют некоторого привыкания.

Ниже приводятся примеры сообщений об ошибках, с которыми вы можете столкнуться. Каждое сообщение состоит из трех частей:

- фрагмент кода, где обнаружена ошибка;
- текст сообщения об ошибке;
- объяснение с предлагаемым вариантом исправления ошибки.

Рассмотрим пример наихудшего сообщения об ошибке, которое вы когда-либо видели:

```
(define (absolute n)
  (cond
    [< 0 (- n)]      <: expected a function call, but there is
    [else n]))        no open parenthesis before this function
                      (<: предполагается вызов функции, но нет
                       открывающей круглой скобки перед этой
                       функцией).
```

Выражение `cond` состоит из ключевого слова, за которым следует произвольно длинная последовательность предложений. Каждое предложение, в свою очередь, состоит из двух частей: условного выражения и ответа, которые являются выражениями. В этом определении функции `absolute` первое предложение начинается с символа `<`, который, как предполагается, должен задавать условие, но даже не является выражением согласно определению. Это настолько сбивает BSL с толку, что он не «видит» открывающую скобку слева от `<`. Исправленное условное выражение должно выглядеть так: (`< n 0`).

Выделение символа `<` в определении функции подсказывает, где находится ошибка. Ниже определения показано сообщение об ошибке, которое DrRacket выведет в окне взаимодействий после щелчка на кнопке **RUN** (Выполнить). Изучите текст справа, поясняющий ошибку, чтобы понять, как среагировать на это несколько противоречивое сообщение. А теперь мы хотим вас успокоить, что никакие другие сообщения об ошибках не являются столь же туманными, как это.

Итак, если вы столкнетесь с ошибкой и вам потребуется помочь, найдите текст ошибки в списке ниже и прочитайте соответствующее подробное описание.

Сообщения об ошибках применения функций в BSL.

Предположим, что область определений содержит следующий код и ничего больше:

```
| ; Число Число -> Число
| ; вычисляет среднее двух чисел, x и у
```

```
(define (average x y)
  (/ (+ x y)
      2))
```

Щелкните на кнопке **RUN** (Выполнить). Если теперь ввести в области взаимодействий следующие выражения, то DrRacket выведет соответствующие сообщения об ошибках.

```
(f 1)
```

f: this function is not defined
(f: эта функция не определена)

Приложение ссылается на f как на функцию, но функция f не определена в области определений. Определите функцию или убедитесь, что имя переменной написано правильно.

```
(1 3 "three" #true)
```

function call: expected a function after
the open parenthesis, but found a number
(вызов функции: после открывающей круглой скобки ожидается функция, но встречено число)

За открывающей круглой скобкой всегда должно следовать ключевое слово или имя функции, но 1 не является ни тем, ни другим. Имя функции определяется либо самим языком BSL, как, например, +, либо программистом в области определений, как, например, average.

```
(average 7)
```

average: expects 2 arguments,
but found only 1
(average: ожидается два аргумента, а обнаружен один)

Этот вызов функции применяет average к одному аргументу, 7, но определение этой функции требует двух чисел.

```
(average 1 2 3)
```

average: expects 2 arguments, but found 3
(average: ожидается два аргумента, а обнаружено три)

Здесь функция average применяется к трем аргументам вместо двух.

```
(make-posn 1)
```

make-posn: expects 2 arguments,
but found only 1
(make-posn: ожидается два аргумента,
а обнаружен один)

Функции, определяемые в самом языке BSL, тоже должны применяться кциальному числу аргументов. Например, make-posn должна применяться к двум аргументам.

Сообщения об ошибках в данных

Далее снова предполагается, что область определений содержит только следующий код:

```
| ; Число Число -> Число
| ; вычисляет среднее двух чисел, x и у
(define (average x y) ...)
```

Напомним, что posn – это предопределенный тип структур.

```
(posn-x #true)
```

posn-x: expects a posn, given #true
(posn-x: ожидается posn, а получено #true)

Функция должна применяться к ожидаемым ею аргументам. Например, posn-x ожидает получить экземпляр posn.

```
(average "one" "two")
```

+: expects a number as 1st argument,
given "one"
(+: ожидается число в первом аргументе,
а получена строка «one»)

Если, согласно определению, функция принимает два числа, то она должна применяться к двум числам; здесь average применяется к двум строкам. Ошибка возникает, только когда average применяет операцию + к строкам. Подобно всем элементарным операциям, + является проверяющей функцией.

Сообщения об ошибках в условиях

На этот раз предполагается наличие определения константы в области определений:

```
; N -- случайное число в диапазоне [0,1,...10)
(define 0-to-9 (random 10))
```

```
(cond
  [(>= 0-to-9 5)])
  cond: expected a clause with a question
  and an answer, but found a clause with
  only one part
  (cond: ожидается предложение с услови-
  ем и ответом, а обнаружено предложение
  только с одной частью)
```

Каждое предложение в `cond` должно содержать точно две части: условие и ответ. Выражение `(>= 0-to-9 5)` здесь интерпретируется как условие; выражение ответа отсутствует.

```
(cond
  [(>= 0-to-9 5)
   "head"
   "tail"])
  cond: expected a clause with a question
  and an answer, but found a clause
  with 3 parts
  (cond: ожидается предложение с услови-
  ем и ответом, а обнаружено предложение
  с тремя частями)
```

В этом случае предложение `cond` состоит из трех частей, что также является нарушением грамматики. Выражение `(>= 0-to-9 5)` в этом примере явно предназначено для проверки условия, но за ним следуют два ответа: `"head"` и `"tail"`. Чтобы исправить ошибку, выберите из двух строк нужную и оставьте только ее.

```
(cond)
  cond: expected a clause after cond,
  but nothing's there
  (cond: после cond ожидается предложе-
  ние, но оно отсутствует)
```

Оператор `cond` должен сопровождаться, по меньшей мере, одним предложением с условием и ответом; на практике чаще используется форма `cond` с двумя и более предложениями.

Сообщения об ошибках в определениях функций

Во всех следующих сценариях предполагается, что в область определений был введен следующий код и нажата кнопка **RUN** (Выполнить).

```
(define f(x) x)
```

define: expected only one expression after
the variable name f, but found 1 extra part
(define: после имени переменной f ожида-
ется только одно выражение, но обнару-
жена одна дополнительная часть)

Определение функции состоит из трех частей: ключевого слова `define`, последовательности имен переменных, заключенных в круглые скобки, и выражения. Это определение состоит из четырех частей; вероятно, данное определение является попыткой использовать для заголовка стандартное обозначение из курса алгебры – $f(x)$ вместо $(f\ x)$.

```
(define (f x x) x)
```

define: found a variable that is used
more than once: x
(define: обнаружена переменная, исполь-
зумая более одного раза: x)

Последовательность параметров в определении функции не должна содержать повторяющиеся имена переменных.

```
(define (g) x)
```

define: expected at least one variable after
the function name, but found none
(define: после имени функции ожидается,
по крайней мере, одна переменная, но не
обнаружено ни одной)

В языке BSL заголовок функции должен содержать, по крайней мере, два имени. Первое – имя функции; остальные – имена переменных, которые являются параметрами, а в этом определении функции они отсутствуют.

```
(define (f (x)) x)
```

define: expected a variable, but found a part
(define: ожидается переменная, но найде-
но выражение)

Заголовок функции содержит выражение (x) , которое не является именем переменной.

```
(define (h x y) x y)
define: expected only one expression for the
function body, but found 1 extra part
(define: ожидается только одно выражение
для тела функции, но обнаружена одна до-
полнительная часть)
```

В этом определении функции за заголовком следуют два выражения: `x` и `y`.

Сообщения об ошибках в определениях структур

Для исследования ошибок, описываемых далее, необходимо добавить в область определений следующие определения структур и щелкнуть на кнопке **RUN** (Выполнить).

```
(define-struct (x))
(define-struct (x y)) define-struct: expected the structure name
                           after define-struct, but found a part
                           (define-struct: после define-struct ожидает-
                           ся имя структуры, а обнаружено выраже-
                           ние)
```

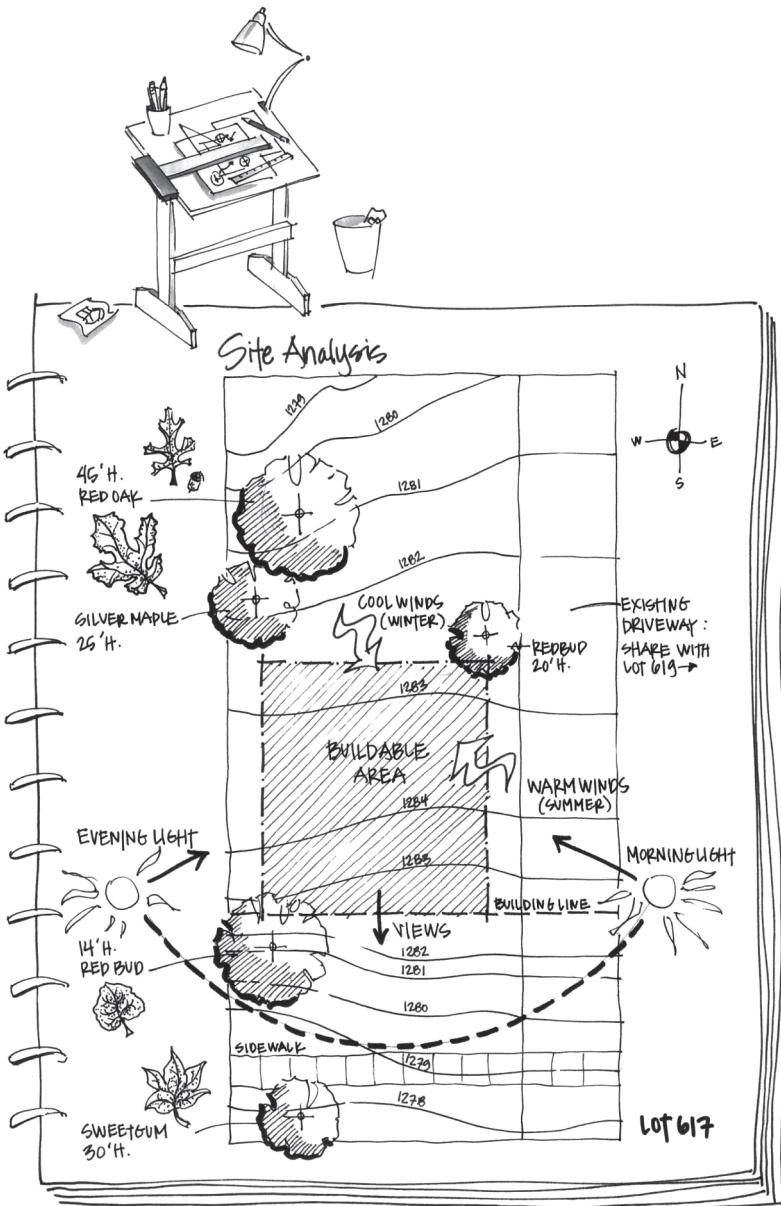
Определение структуры состоит из трех частей: ключевого слова `define-struct`, имени структуры и последовательности имен в круглых скобках. В этих примерах отсутствует имя структуры.

```
(define-struct x
  [y y]) define-struct: found a field name that is used
                           more than once: y
                           (define-struct: встречено имя поля, ис-
                           пользующееся более одного раза: y)
```

Последовательность имен полей в определении структуры не должна содержать повторяющихся имен.

```
(define-struct x y)
(define-struct x y z) define-struct: expected at least one field name
                           (in parentheses) after the structure name,
                           but found something else
                           (define-struct: после имени структуры
                           ожидается хотя бы одно имя поля (в скоб-
                           ках), но встречено что-то иное)
```

В этих определениях структур отсутствуют последовательности имен полей, заключенные в скобки.



II ДАННЫЕ ПРОИЗВОЛЬНОГО РАЗМЕРА

Все определения данных в части I описывают данные фиксированного размера. Для нас логические значения, числа, строки и изображения являются атомарными; специалисты в информатике говорят, что они имеют единичный размер. После знакомства со структурами вы научились объединять фиксированное количество фрагментов данных. Даже при создании глубоко вложенных структур с использованием языка определения данных вы всегда знаете точное количество элементарных фрагментов данных в любом конкретном экземпляре. Однако многие задачи в программировании предполагают вычисления с неопределенным количеством фрагментов информации, которые должны обрабатываться как одно целое. Например, программе может потребоваться вычислить среднее значение группы чисел или следить за произвольным количеством объектов в интерактивной игре. Однако с текущим багажом знаний невозможно сформулировать определение данных, с помощью которого можно было бы представить такую информацию в виде данных.

В этой части мы вновь вернемся к языку определения данных и рассмотрим его особенности, позволяющие описывать данные произвольного (но конечного) размера. Например, первая половина этой части будет посвящена спискам – форме данных, которая присутствует в большинстве современных языков программирования. Помимо дополнений в языке определения данных, в этой части мы обновим рецепт проектирования, чтобы работать с такими определениями данных. И в заключительных главах вы увидите, как эти определения данных и обновленный рецепт проектирования применяются в различных контекстах.

8. Списки

Возможно, прежде вам не доводилось сталкиваться с определениями данных, ссылающимися на самих себя. Ваши учителя родного языка и литературы почти наверняка старались держаться подальше от подобных определений, так же как и учителя математики. Но программисты не могут позволить себе нечеткости и неопределенности. Их работа требует точности. Определение данных, как правило, может содержать несколько ссылок на самого себя, и в этой главе мы покажем примеры таких определений, начав со списка.

Введение списков также расширяет круг приложений, которые мы можем изучать. В этой главе мы постараемся осветить все аспекты с помощью наглядных примеров, а также объясним причины для пересмотра рецепта проектирования в следующей главе, где рассказывается, как системно проектировать функции, которые работают с определениями данных, ссылающимися на самих себя.

8.1. Создание списков

Все мы постоянно составляем списки. Перед тем как пойти в магазин, мы составляем список продуктов, которые нужно приобрести. Некоторые каждое утро составляют список дел на день. В декабре многие дети готовят новогодние списки желаний. Чтобы спланировать вечеринку, мы составляем список приглашенных. Организация информации в виде списков – неотъемлемая часть нашей жизни.

Так как часто информация поступает в виде списков, мы должны научиться представлять такие списки в виде данных на языке BSL. Поскольку списки так важны, в языке BSL имеется все необходимое для создания списков и управления ими, по аналогии с точками на декартовой плоскости (`posn`). Но, в отличие от точек, определение данных, представляющих списки, всегда остается за вами. Однако не будем забегать вперед и начнем с создания списков.

Формирование списка всегда начинается с пустого списка. В BSL пустой список определяется так:

```
| '()
```

Это определение пустого списка. Так же как `#true` или `5`, `'()` – это всего лишь константа. Добавляя элемент в список, мы создаем другой список; для этой цели в BSL имеется операция `cons`. Например, выражение

```
| (cons "Mercury" '())
```

создает новый список из списка `'()` и строки `"Mercury"`. В табл. 7 этот список представлен в той же наглядной форме, какую мы использовали для иллюстрации структур. В блоке с подписью `cons` имеются

два поля: `first` и `rest`. В этом конкретном примере поле `first` содержит строку "Mercury", а поле `rest` – '()'.

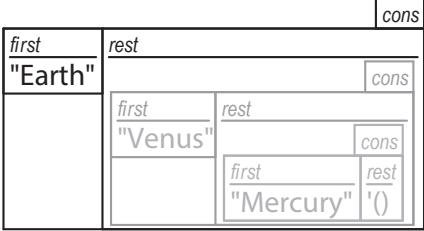
Получив список с одним элементом, мы можем сконструировать список с двумя элементами, вновь воспользовавшись операцией `cons`. Например, так:

```
| (cons "Venus" (cons "Mercury" '()))
```

или так:

```
| (cons "Earth" (cons "Mercury" '()))
```

Таблица 7. Конструирование списка

Список	Диаграмма
(<code>cons</code> "Mercury" '())	
(<code>cons</code> "Venus" (<code>cons</code> "Mercury" '()))	
(<code>cons</code> "Earth" (<code>cons</code> "Venus" (<code>cons</code> "Mercury" '())))	

В средней строке в табл. 7 показано, как можно представить список с двумя элементами. Это тот же блок из двух полей, но на этот раз поле `rest` содержит еще один похожий блок. На самом деле этот вложенный блок является блоком из первой строки в той же таблице.

Наконец, сконструируем список из трех пунктов:

```
| (cons "Earth" (cons "Venus" (cons "Mercury" '()))))
```

Последняя строка в табл. 7 демонстрирует список с тремя элементами. Его поле `rest` содержит блок, который снова содержит блок. Итак, создавая списки, мы помещаем блоки в блоки, которые, в свою очередь, помещаем в другие блоки, и т. д. На первый взгляд это может показаться странным, но вообще это очень похоже на матрешки. Единственное отличие состоит в том, что программы на BSL могут вложить друг в друга гораздо больше блоков, чем самый искусный мастер смог бы вложить матрешек.

Поскольку даже у хорошего художника возникнут проблемы с рисованием глубоко вложенных структур, специалисты в информатике прибегают к диаграммам в виде прямоугольников и стрелок. В табл. 8 показано, как можно переставить последнюю строку из табл. 7. Каждая структура cons изображается как отдельный блок. Если поле `rest` настолько сложное, что его нельзя нарисовать внутри блока, то вместо него изображается точка и линия со стрелкой, указывающей на содержащийся в нем блок. В зависимости от расположения блоков вы получаете два вида диаграмм. Первая, показанная в верхней строке в табл. 8, перечисляет блоки в порядке их создания. Вторая, показанная в нижней строке, перечисляет блоки в порядке их включения с помощью `cons`. То есть вторая диаграмма сообщает, что будет получено при применении `first` к списку, независимо от длины этого списка. Именно поэтому программисты предпочитают второй вариант.

Таблица 8. Изображение списка в виде диаграммы

Список	Диаграмма
<code>(cons "Earth" (cons "Venus" (cons "Mercury" '())))</code>	<pre> graph LR C1[cons first: "Earth" rest: C2] --> C2[cons first: "Venus" rest: C3] C2 --> C3[cons first: "Mercury" rest: "()'"] </pre>
	<pre> graph LR C1[cons first: "Earth" rest: C2] --> C2[cons first: "Venus" rest: C3] C2 --> C3[cons first: "Mercury" rest: "()'"] </pre>

Упражнение 129. Создайте списки, представляющие:

- 1) список небесных тел, скажем всех планет в нашей Солнечной системе;
- 2) список блюд, например стейк, картофель фри, бобы, хлеб, вода, сыр Бри и мороженое;
- 3) список цветов.

Нарисуйте диаграммы этих списков, как в табл. 7 и 8. Какой из рисунков вам нравится больше? ■

Аналогично можно составлять списки чисел. Вот пример списка из 10 цифр:

```

  (cons 0
    (cons 1
      (cons 2
        (cons 3
          (cons 4
            (cons 5
              (cons 6
                ...
  
```

```
(cons 7
  (cons 8
    (cons 9 '()))))))))
```

Чтобы построить этот список, потребуется 10 операций конструирования списков и один пустой список '(). Чтобы создать список с тремя произвольными числами, например:

```
(cons pi
  (cons e
    (cons -22.3 '())))
```

потребуется использовать три операции cons.

Вообще говоря, списки необязательно должны состоять из значений одного типа, например:

```
(cons "Robbie Round"
  (cons 3
    (cons #true
      '())))
```

Первый элемент в этом списке – строка, второй – число, а последний – логическое значение. Этот список можно рассматривать как запись в трудовой книжке с тремя элементами данных: имя сотрудника, количество лет, отработанных в компании, и наличие у сотрудника корпоративной медицинской страховки. Также можно вообразить, что этот список представляет виртуального игрока в какой-то игре. Без определения данных невозможно понять, что это за данные.

Имейте в виду, что если список должен представлять запись с фиксированным количеством полей, то вместо списка лучше использовать структуру.

Бот первое определение данных, включающее операцию cons:

```
; 3LON -- это список с тремя числами:
; (cons Number (cons Number (cons Number '())))
; интерпретация: координаты точки в 3-мерном пространстве
```

Это определение данных использует операцию cons, подобно тому, как другие определения используют конструкторы для создания экземпляров структур, и в некотором смысле cons – это всего лишь специальный конструктор. Но такое определение данных не демонстрирует, как формировать списки произвольной длины: списки, которые могут ничего не содержать, содержат один элемент, два элемента, десять элементов или, может быть, даже 1 438 901 элемент.

Что ж, попробуем еще раз:

```
; List-of-names -- это одно из значений:
; -- '()
; -- (cons String List-of-names)
; интерпретация: список фамилий приглашенных
```

Сделайте глубокий вдох и прочтите это определение еще раз. Это одно из самых необычных определений, с которыми вы когда-либо сталкивались. Прежде вы не видели определений, ссылающихся на

самых себя. Имеет ли такое определение смысл? В конце концов, если бы вы сказали своему учителю родного языка, что «стол – это стол», то почти наверняка вы услышали бы в ответ: «Вздор!» – потому что определение, ссылающееся на самого себя, не объясняет значения слова.

В информатике и программировании, однако, определения, ссылающиеся на себя, используются очень широко, и при некоторой осторожности такие определения действительно имеют смысл. Здесь «осмысленность» означает, что определение данных можно использовать для описания предназначения, а именно для представления примеров данных, которые принадлежат к определяемому классу, или для проверки принадлежности некоторого заданного фрагмента данных к определенному классу. С этой точки зрения определение List-of-names (список имен) имеет смысл. Как минимум мы можем генерировать '()' в качестве одного примера, используя первое предложение в детализации. А опираясь на '()', как элемент списка имен List-of-names, легко создать второй пример:

```
| (cons "Findler" '())
```

Здесь мы используем строку и список List-of-names, чтобы получить набор данных в соответствии со вторым предложением в детализации. Следуя этому правилу, можно генерировать массу других таких же списков:

```
| (cons "Flatt" '())
| (cons "Felleisen" '())
| (cons "Krishnamurthi" '())
```

И хотя все эти списки содержат одно имя (представленное в виде строки), мы можем использовать вторую строку из определения данных для создания списков с большим количеством имен:

```
| (cons "Felleisen" (cons "Findler" '()))
```

Этот фрагмент данных принадлежит списку имен List-of-names, потому что "Felleisen" является строкой, а (cons "Findler" '()) соответствует определению списка имен List-of-names.

Упражнение 130. Создайте элемент List-of-names, содержащий пять строк. Нарисуйте представление списка, подобное изображенному в табл. 7.

Объясните, почему

```
| (cons "1" (cons "2" '()))
```

является элементом List-of-names, а (cons 2 '()) – нет. ■

Упражнение 131. Напишите определение данных для представления списков логических значений. Определение класса должно соответствовать любым сколь угодно длинным спискам логических значений. ■

8.2. Что такое '(), что такое cons

Теперь приостановимся ненадолго и внимательнее рассмотрим '() и cons. Как уже упоминалось, '() – это просто константа. Но в отличие от таких констант, как 5 или "this is a string", она больше похожа на имя функции или переменную; но если сравнить ее с #true и #false, то легко увидеть, что в действительности это просто представление пустого списка на языке BSL.

Что касается правил вычислений, то '() – это новый вид элементарного (и атомарного) значения, отличный от чисел, логических значений, строк и т. д. Это не составное значение, как Posn. На самом деле константа '() настолько уникальна, что сама по себе принадлежит кциальному классу значений, который имеет предикат, распознавающий только '() и ничего больше:

```
; Любое значение -> логическое значение
; является ли заданное значение пустым списком '()
(define (empty? x) ...)
```

Подобно другим предикатам, empty? может применяться к любому значению из вселенной значений в языке BSL. Он дает #true, только если применяется к пустому списку '():

```
> (empty? '())
#true
> (empty? 5)
#false
> (empty? "hello world")
#false
> (empty? (cons 1 '()))
#false
> (empty? (make-posn 0 0))
#false
```

Теперь обратим внимание на cons. Исходя из примеров, что мы видели до сих пор, можно заключить, что cons – это конструктор, подобный конструкторам структур, которые мы видели выше. Если говорить точнее, cons выглядит как конструктор структуры с двумя полями, первое из которых может содержать любое значение, а второе – список любого вида. Вот как эта идея выражается на языке BSL:

```
(define-struct pair [left right])
; ConsPair -- это структура:
;   (make-pair Any Any).

; Любое значение Любое значение -> ConsPair
(define (our-cons a-value a-list)
  (make-pair a-value a-list))
```

Единственная загвоздка в том, что our-cons принимает любые возможные значения BSL во втором аргументе, а cons – нет, что подтверждает следующий эксперимент:

```
| > (cons 1 2)
| cons:second argument must be a list, but received 1 and 2
```

(cons: второй аргумент должен быть списком, а получены значения 1 и 2).

Иначе говоря, cons – это проверяющая функция, подобная той, что обсуждается в главе 6, которая предлагает следующее уточнение:

```
; ConsOrEmpty -- одно из значений:
; -- '()
; -- (make-pair Any ConsOrEmpty)
; интерпретация: ConsOrEmpty -- это класс всех списков

; Any Any -> ConsOrEmpty
(define (our-cons a-value a-list)
  (cond
    [(empty? a-list) (make-pair a-value a-list)]
    [(pair? a-list) (make-pair a-value a-list)]
    [else (error "cons: second argument ...")]))
```

Если cons – это проверяющая функция-конструктор, то вам может быть интересно узнать, как извлечь отдельные элементы из полученной структуры. В конце концов, в главе 5 говорится, что структуры невозможно использовать без селекторов. Поскольку структура cons имеет два поля, у нее есть два селектора: first и rest. Их легко определить в терминах нашей структуры pair:

```
; ConsOrEmpty -> Любое значение
; извлекает левый элемент из заданной пары
(define (our-first a-list)
  (if (empty? a-list)
      (error 'our-first ...)
      (pair-left a-list)))
```

Стоп! Определите our-rest.

Если ваша программа имеет доступ к определению структуры pair, то она легко сможет создавать экземпляры pair, не содержащие '()' или содержащие другие экземпляры pair в правом поле. Независимо от того, как были созданы такие «плохие» экземпляры, намеренно или случайно, они наверняка приведут к неправильному и странному поведению функций и программ. Поэтому BSL скрывает фактическое определение структуры для cons, чтобы избежать этих проблем. В разделе 16.2 показан один из способов сокрытия таких определений в программах, но на данный момент нам это не нужно.

Таблица 9. Примитивы, имеющие отношение к спискам

'()	специальное значение, представляющее пустой список
empty?	предикат, распознающий только '()
cons	проверяющий конструктор для создания экземпляров с двумя полями
first	селектор для извлечения элемента, добавленного последним
rest	селектор для извлечения расширяющего списка
cons?	предикат, распознающий экземпляры cons

Таблица 9 обобщает этот раздел. Важно запомнить, что `'()` – это уникальное значение, а `cons` – проверяющий конструктор, который создает списки. Кроме того, `first`, `rest` и `cons?` являются самыми обычными селекторами и предикатом. Подводя итог, можно сказать, что в этой главе рассказывается **не** о новом способе создания данных, а о **новом способе формулирования определений данных**.

8.3. Программирование со списками

Предположим, вы ведете список своих друзей и решили, что список может оказаться настолько большим, что вам понадобится программа, с помощью которой можно узнать, присутствует ли какое-то имя в списке. Чтобы конкретизировать эту идею, давайте сформулируем ее как упражнение:

Здесь слово "friend" (друг) используется в смысле, характерном для социальных сетей, а не для реальной жизни.

Задача. Вы работаете со списком контактов для нового сотового телефона. Владелец телефона постоянно обновляет этот список и просматривает его. На данный момент вам поручено разработать функцию, которая принимает этот список контактов и определяет, содержит ли он имя «Flatt».

Решив эту задачу, мы обобщим ее и напишем функцию, которая сможет отыскивать любое имя в списке.

Для представления списка имен, в котором функция должна выполнять поиск, вполне подойдет определение данных `List-of-names` из предыдущего раздела. Так как определение у нас уже есть, то перейдем сразу к заголовку функции:

```
| ; List-of-names -> Boolean
| ; определяет, присутствует ли имя "Flatt" в списке a-list-of-names
| (define (contains-flatt? a-list-of-names)
|   #false)
```

Имя `a-list-of-names` вполне подходит для списка имен, который принимает функция, но оно слишком длинное, поэтому сократим его до `alon`.

Следуя общему рецепту проектирования, приведем несколько примеров, иллюстрирующих назначение функции. Сначала определим результат для простейшего случая: `'()`. Поскольку этот список не содержит ничего, он определенно не содержит "Flatt":

```
| (check-expect (contains-flatt? '()) #false)
```

Затем рассмотрим варианты списков с одним элементом. Вот два примера:

```
| (check-expect (contains-flatt? (cons "Find" '())))
|   #false)
| (check-expect (contains-flatt? (cons "Flatt" '())))
|   #true)
```

В первом случае правильный ответ `#false`, потому что единствен-
ный элемент в списке не "Flatt"; во втором случае единственным
элементом является "Flatt", поэтому ответ – `#true`. И наконец, более
общий пример:

```
(check-expect
  (contains-flatt?
    (cons "A" (cons "Flatt" (cons "C" '()))))
  #true)
```

И снова правильный ответ `#true`, потому что список содержит
"Flatt".

Стоп! Приведите общий пример, правильным ответом на который
будет `#false`.

Сделайте глубокий вдох и запустите программу. Заголовок – это
определение-«пустышка»; в нашей программе есть несколько приме-
ров, которые автоматически преобразуются в тесты, и, что самое ин-
тересное, некоторые из них выполняются успешно. Они успешно вы-
полняются по ошибке. Если причина вам понятна, то читайте дальше.

Четвертый шаг – проектирование макета функции, соответствую-
щего определению данных. Поскольку определение данных для спис-
ков строк состоит из двух предложений, тело функции должно быть
выражением `cond` с двумя ветками. Два условия определяют, какой из
двух типов списков получила функция:

```
(define (contains-flatt? alon)
  (cond
    [(empty? alon) ...]
    [(cons? alon) ...]))
```

Вместо `(cons? alon)` во втором предложении можно использовать
`else`.

Добавим в макет еще одну подсказку. Как вы наверняка помните,
рецепт проектирования предлагает аннотировать каждое предложе-
ние в выражении `cond` выражениями с селекторами, если класс вход-
ных данных представлен составными экземплярами. В данном случае
мы знаем, что `'()` – это элементарное значение, не имеющее никаких
компонентов, составляющих его. В любых других случаях список со-
стоит из строки и другого списка строк, и мы напомним себе об этом
факте, добавив в макет `(first alon)` и `(rest alon)`:

```
(define (contains-flatt? alon)
  (cond
    [(empty? alon) ...]
    [(cons? alon)
      (... (first alon) ... (rest alon) ...)]))
```

Теперь перейдем к пятому шагу в нашем рецепте проектирова-
ния – непосредственной реализации функции. Для начала рассмот-
рим каждое условие в `cond` отдельно. Если `(empty? alon)` истинно, зна-
чит, функция получила пустой список, и результат должен быть `#false`.

Во втором случае, если (`cons?` `alon`) истинно, аннотации в макете напоминают нам, что мы имеем строку в первом элементе и остальную часть списка. Рассмотрим пример, соответствующий этому условию:

```
(cons "A"
  (cons ...
    ... '()))
```

Функция должна сравнить первый элемент с искомой строкой "Flatt". В этом примере первый элемент содержит строку "A", не совпадающую с "Flatt", поэтому операция сравнения дает #`false`. Если взять другой пример, скажем

```
(cons "Flatt"
  (cons ...
    ... '()))
```

то функция определила бы, что первый элемент соответствует искомой строке "Flatt", и ответила бы #`true`. Из всего этого следует, что второе предложение в выражении `cond` должно содержать выражение, сравнивающее строку в первом элементе со строкой "Flatt":

```
(define (contains-flatt? alon)
  (cond
    [(empty? alon) #false]
    [(cons? alon)
      (... (string=? (first alon) "Flatt")
        ... (rest alon) ...))])
```

Если сравнение дает #`true`, то результат функции должен быть #`true`. Если сравнение дает #`false`, то необходимо рассмотреть остальную часть списка: (`rest alon`). Очевидно, что в этом случае функция не может знать окончательного ответа, потому что ответ зависит от того, что скрыто в «...». Иначе говоря, если первый элемент не совпадает со строкой "Flatt", то нам нужно проверить, содержит ли остальная часть списка строку "Flatt".

К счастью, у нас есть функция `contains-flatt?`, отвечающая всем требованиям. В соответствии с назначением она определяет, содержит ли список строку "Flatt". А это значит, что (`contains-flatt? l`) сообщает, содержит ли список `l` искомую строку "Flatt". И, следуя той же логике, (`contains-flatt? (rest alon)`) определяет, содержит ли (`rest alon`) строку "Flatt", что нам и требуется.

Проще говоря, последняя строка должна содержать выражение (`contains-flatt? (rest alon)`):

```
; List-of-names -> Логическое значение
(define (contains-flatt? alon)
  (cond
    [(empty? alon) #false]
    [(cons? alon)
      (... (string=? (first alon) "Flatt") ...
        ... (contains-flatt? (rest alon)) ...))])
```

Вся хитрость в том, чтобы объединить значения двух выражений. Как уже упоминалось, если первое сравнение дает #true, то нет необходимости выполнять поиск в остальной части списка; но если оно дает #false, то второе выражение, в свою очередь, может дать #true, что означает, что имя "Flatt" находится в остальной части списка. То есть результат (contains-flatt? alon) будет #true, когда либо первое, либо второе выражение в последней строчке даст #true.

Листинг 24. Поиск в списке

```
; List-of-names -> Логическое значение
; определяет, присутствует ли имя "Flatt" в списке alon
(check-expect
  (contains-flatt? (cons "X" (cons "Y" (cons "Z" '())))) 
  #false)
(check-expect
  (contains-flatt? (cons "A" (cons "Flatt" (cons "C" '())))) 
  #true)
(define (contains-flatt? alon)
  (cond
    [(empty? alon) #false]
    [(cons? alon)
     (or (string=? (first alon) "Flatt")
         (contains-flatt? (rest alon)))]))
```

В листинге 24 показано полное определение функции. В целом оно мало отличается от определений в первой части книги. Оно состоит из сигнатуры, назначения, двух примеров и определения. Единственное отличие этого определения функции от всего, что вы видели раньше, – это ссылка функции на саму себя, то есть ссылка на contains-flatt? в теле define. С другой стороны, определение данных тоже содержит ссылку на самого себя, поэтому ссылка функции на саму себя не должна вызывать особого удивления.

Упражнение 132. С помощью DrRacket проверьте работу функции contains-flatt? со следующим примером:

```
(cons "Fagan"
  (cons "Findler"
    (cons "Fisler"
      (cons "Flanagan"
        (cons "Flatt"
          (cons "Felleisen"
            (cons "Friedman" '()))))))))
```

Какой результат, по вашему мнению, должна вернуть функция? ■

Упражнение 133. Вот другой способ сформулировать второе предложение cond в функции contains-flatt?:

```
... (cond
  [(string=? (first alon) "Flatt") #true]
  [else (contains-flatt? (rest alon))]) ...
```

Объясните, почему это выражение дает те же результаты, что и выражение в версии в листинге 24. Какая версия понятнее вам? Поясните почему. ■

Упражнение 134. Разработайте функцию `contains?`, которая определяет наличие заданной строки в заданном списке строк.

Примечание. В языке BSL имеется встроенная функция `member?`, которая принимает два значения и проверяет, встречается ли первое во втором, которое интерпретируется как список:

```
| > (member? "Flatt" (cons "b" (cons "Flatt" '())))
| #true
```

Не используйте `member?` в определении функции `contains?`. ■

Листинг 25. Вычисления со списками, шаг 1

```
(contains-flatt? (cons "Flatt" (cons "C" '())))
==
(cond
  [(empty? (cons "Flatt" (cons "C" '()))) #false]
  [(cons? (cons "Flatt" (cons "C" '())))
   (or
     (string=? (first (cons "Flatt" (cons "C" '()))) "Flatt")
     (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))])
```

8.4. Вычисления со списками

Мы все еще используем язык BSL, поэтому правила алгебры – см. интермэццо 1 – говорят нам, как определять значение таких выражений, как

```
| (contains-flatt? (cons "Flatt" (cons "C" '())))
```

без привлечения DrRacket. Программисты должны четко представлять, как выполняются подобные вычисления, поэтому мы по шагам разберем один простой пример.

В листинге 25 показан первый шаг, где для определения результата применения функции используется обычное правило подстановки. В результате получается условное выражение, потому что, как сказал бы учитель алгебры, функция задана кусочно.

Листинг 26. Вычисления со списками, шаг 2

```
...
==
(cond
  [#false #false]
  [(cons? (cons "Flatt" (cons "C" '())))
   (or (string=? (first (cons "Flatt" (cons "C" '()))) "Flatt")
        (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))]
  ==
  (cond
    [(cons? (cons "Flatt" (cons "C" '())))
     (or (string=? (first (cons "Flatt" (cons "C" '()))) "Flatt")
          (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))]
    ==
    (cond
```

```
[#true
  (or (string=? (first (cons "Flatt" (cons "C" '())))) "Flatt")
      (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))
  ==
  (or (string=? (first (cons "Flatt" (cons "C" '())))) "Flatt")
      (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))
```

В листинге 26 показано продолжение вычислений. Чтобы найти правильную ветвь в выражении `cond`, мы должны определить значения условий одно за другим. Поскольку список не пустой, результат первого условия – `#false`, и поэтому мы исключаем первую ветвь из дальнейшего рассмотрения. Условие во втором предложении вычисляется в `#true`, потому что для `cons`-списка условие `cons?` выполняется.

Листинг 27. Вычисления со списками, шаг 3

```
...
==
(or (string=? "Flatt" "Flatt")
    (contains-flatt? (rest (cons "Flatt" (cons "C" '())))))
== (#true (contains-flatt? ...))
== #true
```

От этого момента осталось сделать всего три шага, чтобы получить окончательный результат. Эти три шага показаны в листинге 27. Первый вычисляет `(first (cons "Flatt" ...))` и получает строку "Flatt". Второй обнаруживает, что "Flatt" совпадает с искомой строкой "Flatt". Третий сообщает, что `(ог #true X)` имеет значение `#true` независимо от значения `X`.

Упражнение 135. Используйте движок пошаговых вычислений в DrRacket, чтобы проверить порядок вычислений в

```
| (contains-flatt? (cons "Flatt" (cons "C" '())))
```

Также с помощью движка пошаговых вычислений определите результат

```
| (contains-flatt?
  (cons "A" (cons "Flatt" (cons "C" '()))))
```

Что случится, если строку "Flatt" заменить на "B"? ■

Упражнение 136. С помощью движка пошаговых вычислений в DrRacket проверьте

```
| (our-first (our-cons "a" '())) == "a"
| (our-rest (our-cons "a" '())) == '()
```

Определения этих функций вы найдете в разделе 8.2. ■

9. Проектирование с определениями данных, ссылающимися на самих себя

На первый взгляд, определения данных, ссылающиеся на самих себя, кажутся гораздо более сложными, чем определения смешанных данных. Но, как показал пример `contains-flatt?`, шесть шагов рецепта проектирования остаются действительными и для них. Тем не менее в этой главе мы обобщим рецепт проектирования, чтобы он еще лучше подходил для случаев использования определений данных, ссылающихся на самих себя. Новые правила касаются процесса выявления, когда действительно необходимо использовать самоссылающиеся определения данных, создания макета и определения тела функции:

- Если постановка задачи предполагает использование информации произвольного размера, то определение данных, представляющих ее, должно ссылаться на само себя. Выше вы видели только один такой класс – список имен *List-of-names*. Слева на рис. 11 показано, как подобным же образом определить список строк *List-of-strings*. Остальные списки атомарных данных определяются точно так же.

Чтобы определение данных, ссылающееся на самого себя, было допустимым, оно должно удовлетворять двум условиям. Во-первых, определение должно содержать как минимум два предложения. Во-вторых, хотя бы одно из предложений не должно ссылаться на определяемый класс данных. Хорошей практикой считается явное обозначение ссылок на себя с помощью стрелок от ссылок в определении данных на само это определение; пример такого обозначения см. на рис. 11.

Числа тоже кажутся произвольно большими.
Для неточных чисел это иллюзия. Для точных целых чисел это действительно так.
Поэтому работа с целыми числами является частью этой главы.

```
(define (fun-for-losa-list-of-strings)
  (cond
    [(empty? a-list-of-strings)
     [else ;(cons? a-list-of-strings)
           (... (first a-list-of-strings)
                 (... (rest a-list-of-strings) ...))])])
```

Рис. 11. Стрелками показаны ссылки определений данных на самих себя

Допустимость определений данных со ссылками на самих себя необходимо проверять путем создания примеров данных. Начните с предложения, которое не ссылается на определение данных; перейдите к другому предложению и представьте вариант, использующий предыдущий пример и ссылку на само опреде-

ление. Следуя таким путем, для определения данных на рис. 11 вы получите списки, подобные следующим:

<pre>'()</pre>	для первого предложения
<pre>(cons "a" '())</pre>	для второго предложения, с использованием предыдущего примера
<pre>(cons "b" (cons "a" '()))</pre>	для второго предложения, с использованием предыдущего примера

Если из определения данных не получается сгенерировать примеры, то это говорит о недопустимости такого определения. Если создать первые примеры получилось, но вы не понимаете, как создать более крупные примеры, то это говорит о том, что такое определение, вероятно, не соответствует его интерпретации.

- Правила определения заголовка остаются прежними: заголовок все так же должен включать сигнатуру, описание назначения и фиктивное определение. Формулируя описание назначения, сосредоточьтесь на том, что функция вычисляет, а не как она это делает и, в частности, не на том, как она выполняет обход экземпляров входных данных.

Вот пример, конкретизирующий этот рецепт проектирования:

<pre>; List-of-strings -> Number</pre>	
<pre>; подсчитывает количество строк в списке alos</pre>	
<pre>(define (how-many alos)</pre>	
<pre> 0)</pre>	

В описании назначения четко указано, что функция просто подсчитывает строки в заданном списке; и нет необходимости заранее думать о том, как сформулировать эту идею в виде функции на языке BSL.

- Когда дело доходит до примеров применения функции, обязательно представьте примеры входных данных, в которых несколько раз используется предложение со ссылкой на само определение данных. Это лучший способ позже сформулировать тесты, которые охватывают все определение функции. В нашем текущем примере описание назначения почти само подсказывает примеры применения функции:

дано	ожидается
'()	0
(cons "a" '())	1
(cons "b" (cons "a" '()))	2

Первая строка соответствует пустому списку, и мы знаем, что пустой список ничего не содержит. Вторая строка – это список с одной строкой, поэтому ожидаемый результат 1. Последняя строка представляет список с двумя строками.

4. По сути, определение данных, ссылающееся на само себя, выглядит так же, как определение смешанных данных. Поэтому разработка макета может продолжаться в соответствии с рецептом в главе 6. В частности, необходимо сформулировать выражение `cond`, в котором количество предложений совпадает с количеством предложений в определении данных. Каждое условие должно соответствовать своему предложению в определении данных и содержать соответствующие селекторы.

Таблица 10. Как преобразовать определение данных в макет

Вопрос	Ответ
Различаются ли в определении данных разные подклассы данных?	Макет должен содержать столько предложений <code>cond</code> , сколько имеется различных подклассов в определении данных
Чем подклассы отличаются друг от друга?	Используйте различия, чтобы сформулировать условие для каждого предложения
Имеются ли предложения, включающие структурированные значения?	Если да, добавьте в предложение соответствующие селекторы
Используются ли в определении данных ссылки на само определение?	Сформулируйте «естественные рекурсии» в макете, соответствующие ссылкам в определении данных на само определение
Имеются ли в определении данных ссылки на какие-то другие определения данных?	Специализируйте макет с учетом этого другого определения данных. Обратитесь к соответствующему макету. См. раздел 6.1, шаги 4 и 5 рецепта проектирования

В табл. 10 эта идея выражена в виде викторины с вопросами и ответами. В левом столбце задаются вопросы об определении данных, а в правом объясняется, что необходимо предпринять при построении макета.

Если проигнорировать последнюю строку и применить первые три вопроса к любой функции, принимающей список строк `List-of-strings`, то вы получите следующий макет:

```
(define (fun-for-los alos)
  (cond
    [(empty? alos) ...]
    [else
      (... (first alos) ... (rest alos) ...)]))
```

Напомним, однако, что цель макета – выразить определение данных в виде заготовки функции. То есть макет выражает в форме кода то, что в определении данных выражается как смесь слов на естественном языке и на языке BSL. Проще говоря, каждой важной части в определении данных должен соответствовать свой аналог в макете, и это правило должно неукоснительно соблюдаться, когда определение данных содержит

стрелку, указывающую изнутри определения на само определение. В частности, если определение данных ссылается на само себя в i -м предложении и k -м поле упомянутой там структуры, то макет должен включать ссылку в i -м предложении `cond` и содержать селектор для k -го поля. Для каждого такого селектора добавьте стрелку, указывающую обратно на параметр функции. В конечном итоге в макете должно быть столько же стрелок, сколько имеется в определении данных.

Рисунок 11 иллюстрирует эту идею на примере макетов функций, которые принимают список строк. Оба примера включают одну стрелку, которая берет начало во втором предложении – поле и селектор `rest` соответственно – и указывает на начало соответствующего определения.

Поскольку код на BSL, как и в большинстве языков программирования, записывается в виде текста, вы должны использовать альтернативу стрелке – рекурсивное применение функции к соответствующему селектору:

```
(define (fun-for-los alos)
  (cond
    [(empty? alos) ...]
    [else
      (... (first alos) ...)
      (... (fun-for-los (rest alos)) ...))])
```

Интересно отметить, что рецепт проектирования произвольно больших данных соответствует так называемым «доказательствам по индукции» в математике, а «прыжок веры» представляет использование гипотезы индукции в индуктивном шаге такого доказательства. Логика доказывает справедливость этого метода с помощью теоремы индукции.

5. Тело функции должно начинаться с предложений в `cond`, не имеющих рекурсивных вызовов. Такие предложения называются *базовыми вариантами*. Соответствующие ответы обычно легко формулируются или уже даны в примерах.

Далее следуют рекурсивные варианты. Для начала напомним, что вычисляет каждое из выражений в строке макета. Для естественной рекурсии предполагается, что функция уже работает, как указано в описании назначения. Этот последний шаг называют «прыжком веры», но, как вы увидите, он всегда дает верный результат.

Все остальное связано с объединением различных значений.

В табл. 11 сформулированы первые четыре вопроса и ответы для этого шага. Воспользуемся ею, чтобы завершить определение функции `how-many`. Переименование макета `fun-for-los` в `how-many` дает нам следующее:

```
; List-of-strings -> Число
; подсчитывает строки в списке alos
(define (how-many alos)
  (cond
    [(empty? alos) ...]
    [else
```

```
| (... (first alos) ...
|   ... (how-many (rest alos)) ...)))
```

Таблица 11. Как преобразовать определение данных в макет

Вопрос	Ответ
Что должны возвращать нерекурсивные предложения в <code>cond</code> ?	Примеры должны подсказать вам, какие значения должны возвращаться. Если нет, то сформулируйте соответствующие примеры и тесты
Что вычисляют селекторы в рекурсивных предложениях?	Определения должны подсказать вам, какие данные извлекают эти селекторы, а описания интерпретации в определениях данных сообщат, какую информацию эти данные представляют
Какой результат возвращает естественная рекурсия?	Используйте описание назначения, чтобы определить значение, возвращаемое рекурсивным вызовом, без учета того, как это значение вычисляется. Если описание назначения не дает ответа, улучшите его формулировку
Как функция может объединить значения, чтобы получить желаемый результат?	Найдите в BSL функцию, которая объединяет значения. Или, если такой функции нет, добавьте в список желаний вспомогательную функцию. В большинстве случаев этот последний шаг реализуется просто. Назначение, примеры и макет сообщат вам, какая функция или выражение поможет объединить полученные значения в правильный результат. Мы называем такие функции и выражения <i>комбинаторами</i> , слегка злоупотребляя существующей терминологией

Как показывают примеры применения функции, ответом для базового случая является число 0. Два выражения во втором предложении извлекают элемент `first` и вычисляют количество строк в `(rest alos)`. Чтобы вычислить количество строк во всем списке `alos`, достаточно просто прибавить 1 к значению последнего выражения:

```
| (define (how-many alos)
|   (cond
|     [(empty? alos) 0]
|     [else (+ (how-many (rest alos)) 1)])))
```

Этот табличный способ поиска комбинатора предложил Феликс Клок (Felix Klock).

Определить правильный способ объединения значений в желаемый результат не всегда просто. Начинающие программисты часто застревают на этом шаге. Как показано в табл. 12, сначала необходимо поместить в таблицу все примеры применения функции, а также значения выражений в макете. В табл. 13 показано, как выглядит подобная таблица для нашего примера `how-many`. В крайнем левом столбце перечислены примеры входных данных, а в крайнем правом – желаемые результаты для

этих входных данных. Три столбца между ними показывают значения выражений в макете: (`(first alos)`), (`(rest alos)`) и (`(how-many (rest alos))`), последний из которых является естественной рекурсией. Если рассматривать эту таблицу достаточно долго, можно заметить, что значения в столбце с результатами всегда на единицу больше значений в столбце, соответствующем естественной рекурсии. Таким образом, легко догадаться, что

```
| (+ (how-many (rest alos)) 1)
```

является тем самым выражением, которое вычисляет желаемый результат. Поскольку среда разработки DrRacket помогает быстро проверить такие предположения, используйте ее и щелкните на кнопке **RUN** (Выполнить). Если примеры, преобразованные в тесты, выполняются успешно, мысленно пройдитесь по выражению, чтобы убедиться, что оно будет правильно работать для всех списков; в противном случае продолжайте добавлять в таблицу все новые и новые строки с примерами, пока у вас не появится другая идея.

Таблица 12. Преобразование макета в функцию, табличный метод

Вопрос	Ответ
Итак, если вы застряли...	...оформите примеры из третьего шага в виде таблицы. Поместите входное значение в первый столбец, а желаемый результат – в последний. В промежуточные столбцы впишите значения, возвращаемые селекторами и рекурсивными вызовами. Добавляйте примеры в таблицу, пока не заметите общую закономерность, определяющую комбинатор
Если макет ссылается на макет вспомогательной функции, то как определить, что вычисляет эта функция?	Обратитесь к описанию назначения другой функции и примерам, чтобы определить, что она вычисляет, и исходите из того, что вы уже можете использовать результат, даже если проектирование этой вспомогательной функции еще не закончено

Таблица 13. Аргументы, промежуточные значения и результаты

alos	(first alos)	(rest alos)	(how-many (rest alos))	(how-many alos)
(<code>cons "a"</code> <code>'())</code>)	"a"	'()	0	1
(<code>cons "b"</code> (<code>cons "a"</code> <code>'())</code>)	"b"	(<code>cons "a"</code> <code>'())</code>)	1	2
(<code>cons "x"</code> (<code>cons "b"</code> (<code>cons "a"</code> <code>'())</code>))	"x"	(<code>cons "b"</code> (<code>cons "a"</code> <code>'())</code>))	2	3

В таблице также видно, что некоторые селекторы в макете не имеют отношения к вычислению фактического результата. В данном случае нет необходимости вычислять (`first alos`), что сильно отличает эту функцию от `contains-flatt?`, где используются оба выражения из макета.

Продолжая читать эту книгу, имейте в виду, что во многих случаях этап комбинирования можно выразить с помощью элементарных операций языка BSL, таких как `+`, `and` или `cons`. Но иногда вам, возможно, придется добавить вспомогательную функцию. Кроме того, в некоторых случаях могут понадобиться вложенные условия.

6. Наконец, преобразуйте все примеры в тесты и убедитесь, что все они выполняются успешно и охватывают все части функции.

Вот наши примеры для `how-many`, преобразованные в тесты:

```
(check-expect (how-many '()) 0)
(check-expect (how-many (cons "a" '())) 1)
(check-expect
  (how-many (cons "b" (cons "a" '())))) 2)
```

Примеры лучше сразу формулировать в виде тестов, и язык BSL позволяет это. Такой подход поможет также, если на предыдущем шаге вам придется прибегнуть к табличному методу поиска комбинатора.

В табл. 13 приводится обобщенная версия рецепта проектирования, представленного в этом разделе. В первом столбце перечислены шаги рецепта, а во втором – ожидаемые результаты после выполнения каждого шага. В третьем столбце описываются действия, которые помогут сделать тот или иной шаг. Рецепт учитывает определения списков, ссылающиеся на самих себя, которые мы использовали в этой главе. Как всегда, для овладения любым процессом нужна практика, поэтому мы настоятельно рекомендуем выполнить следующие упражнения, в которых предлагается применить рецепт для решения нескольких задач.

Вы можете скопировать табл. 13 на одну сторону картонной карточки для записей, записать свои варианты вопросов и ответов для этого рецепта проектирования на другой стороне и затем всегда держать эту карточку под рукой.

Таблица 14. Проектирование функций для обработки данных, ссылающихся на самих себя

Шаги	Результат	Действия
Анализ задачи	Определение данных	Разработка структуры данных, представляющих информацию; создание примеров конкретных элементов информации и описание интерпретации данных; идентификация ссылок на себя в определении данных

Таблица 14 (окончание)

Шаги	Результат	Действия
Определение заголовка	Сигнатура; описание назначения; фиктивное определение	Определение сигнатуры с использованием известных типов данных; формулирование краткого описания назначения функции; создание фиктивного определения функции, которое возвращает постоянное значение из указанного диапазона
Примеры	Примеры и тесты	Создание нескольких примеров, не менее одного для каждого предложения в определении данных
Макет	Макет функции	Преобразование определения данных в макет: по одному условию в выражении cond для каждого предложения в определении данных; селекторы, с помощью которых условия идентифицируют структуру; одна естественная рекурсия для каждой ссылки на само определение
Определение	Законченное определение функции	Выбор способа комбинирования в ожидаемый результат значений выражений в предложениях cond
Тестирование	Проверка выполнения тестов	Преобразование всех примеров в тесты check-expect и их выполнение

9.1. Практические упражнения: списки

Упражнение 137. Сравните макеты функций contains-flatt? и how-мапу. Кроме имен функций, они совершенно одинаковые. Объясните это сходство. ■

Упражнение 138. Вот определение данных для представления последовательности денежных сумм:

```
| ; List-of-amounts -- одно из значений:
| ; -- '()
| ; -- (cons PositiveNumber List-of-amounts)
```

Создайте несколько примеров, чтобы проверить себя, насколько правильно вы понимаете это определение данных. Добавьте стрелки, обозначающие ссылки на само определение.

Спроектируйте функцию sum, которая принимает список List-of-amounts и подсчитывает сумму всех денежных средств в списке. Используйте движок пошаговых вычислений в DrRacket и посмотрите, как применение (sum l) обрабатывает короткий список l типа List-of-amounts. ■

Упражнение 139. Взгляните на следующее определение данных:

```
| ; List-of-numbers -- одно из значений:
| ; -- '()
| ; -- (cons Number List-of-numbers)
```

Некоторые экземпляры этого класса данных можно передать на вход функции `sum` из предыдущего упражнения 138, а некоторые – нет.

Спроектируйте функцию `pos?`, которая принимает список чисел List-of-numbers и определяет, являются ли все числа в этом списке положительными. Иначе говоря, если `(pos? l)` возвращает `#true`, то это означает, что `l` является экземпляром списка денежных сумм List-of-amounts. Используйте движок пошаговых вычислений в DrRacket, чтобы выяснить, как `pos?` обрабатывает списки `(cons 5 '())` и `(cons -1 '())`.

Также спроектируйте функцию `checked-sum`. Функция должна принимать список чисел List-of-numbers и возвращать их сумму, если этот список является экземпляром списка денежных сумм List-of-amounts; в противном случае `checked-sum` должна сообщать об ошибке.

Подсказка. Используйте для этого `check-eogg`.

Что вычислит функция `sum`, получив список чисел List-of-numbers? ■

Упражнение 140. Спроектируйте функцию `all-true`, которая принимает список логических значений и определяет, все ли они равны `#true`. Иначе говоря, если в списке есть хотя бы одно значение `#false`, то функция должна вернуть `#false`.

Затем спроектируйте функцию `one-true`, которая принимает список логических значений и определяет, имеется ли в этом списке хотя бы одно значение `#true`. ■

Используйте табличный подход к проектированию. Это поможет определить базовый случай. Используйте движок пошаговых вычислений в DrRacket, чтобы увидеть, как эти функции обрабатывают списки `(cons #true '())`, `(cons #false '())` и `(cons #true (cons #false '()))`.

Упражнение 141. Представьте, что вам предложили спроектировать функцию `cat`, которая принимает список строк и объединяет все его элементы в одну длинную строку. Вы неизбежно придете к следующему частичному определению:

```
; List-of-string -> String
; объединяет все элементы из l в одну длинную строку

(check-expect (cat '()) "")
(check-expect (cat (cons "a" (cons "b" '()))) "ab")
(check-expect
  (cat (cons "ab" (cons "cd" (cons "ef" '()))))
  "abcdef")

(define (cat l)
  (cond
    [(empty? l) ""]
    [else (... (first l) ... (cat (rest l)) ...)]))
```

Заполните табл. 15. Определите функцию, которая может скомбинировать желаемый результат из значений, вычисленных с помощью подвыражений.

Используйте движок пошаговых вычислений в DrRacket, чтобы проверить порядок вычисления выражения `(cat (cons "a" '()))`. ■

Таблица 15. Таблица для функции cat

<i>l</i>	(first <i>l</i>)	(rest <i>l</i>)	(cat (rest <i>l</i>))	(cat <i>l</i>)
(cons "a" (cons "b" '()))	???	???	???	"ab"
(cons "ab" (cons "cd" (cons "ef" '())))	???	???	???	"abcdef"

Упражнение 142. Спроектируйте функцию *ill-sized?*, которая принимает список изображений *loi* и положительное число *n*. Она должна отыскать первое изображение в *loi*, ширина и высота которого не равны *n*; если такого изображения нет, то функция должна вернуть #false.

Подсказка. Используйте следующее неполное определение:

```
| ; ImageOrFalse -- одно из значений:  
| ; -- Image  
| ; -- #false ■
```

9.2. Непустые списки

Теперь вы знаете достаточно, чтобы уверенно создавать определения данных, представляющие списки. Решив упражнения (хотя бы некоторые) в конце предыдущего раздела, вы научились обрабатывать списки различных видов чисел, логических значений, изображений и т. д. В этом разделе мы продолжим знакомство со списками и приемами их обработки.

Начнем с простой на вид задачи вычисления среднего значения по списку температур. Для простоты сразу приведем определение данных:

```
| ; List-of-temperatures -- одно из значений:  
| ; -- '()  
| ; -- (cons CTemperature List-of-temperatures)  
| ; CTemperature -- число больше -272.
```

Вообще говоря, температуры можно представить простыми числами, но второе определение данных напоминает, что не все числа являются температурами, и вы должны иметь это в виду.

Заголовок выглядит просто:

```
| ; List-of-temperatures -> Number  
| ; вычисляет среднюю температуру  
(define (average allot) 0)
```

Придумать примеры для этой задачи тоже несложно, поэтому сформулируем только один тест:

```
(check-expect
  (average (cons 1 (cons 2 (cons 3 '())))) 2)
```

Ожидаемый результат – это, как нетрудно догадаться, сумма температур, деленная на их количество.

Немного подумав, можно заметить, что макет функции `average` должен напоминать макеты, виденные нами до сих пор:

```
(define (average alot)
  (cond
    [(empty? alot) ...]
    [(cons? alot)
      (... (first alot) ...
           ... (average (rest alot)) ...)]))
```

Два условия в выражении `cond` соответствуют двум предложениям в определении данных; вопросы отличают пустые списки от непустых; и из-за ссылки на само определение данных необходима естественная рекурсия.

Однако превратить этот макет в определение функции слишком непросто. Первое условие в `cond` должно возвращать число, представляющее среднее значение для пустого списка температур, но такого числа нет. Точно так же второе условие предполагает применение функции, которая объединит текущую температуру и среднее значение для остальной части списка температур в новое среднее значение. Это возможно, но такой способ вычисления среднего значения выглядит неестественным.

Среднее значение температур получается делением их суммы на количество. Мы отметили это, когда формулировали тривиальный пример. Наша формулировка предполагает, что `average` решает три задачи: суммирование, подсчет и деление. Рецепт, представленный в первой части, требует для каждой задачи написать отдельную функцию, и если поступить именно так, то организация вычисления среднего станет очевидна:

```
; List-of-temperatures -> Number
; вычисляет среднюю температуру
(define (average alot)
  (/ (sum alot) (how-many alot)))

; List-of-temperatures -> Number
; суммирует температуры в заданном списке
(define (sum alot) 0)

; List-of-temperatures -> Number
; считывает количество температур в заданном списке
(define (how-many alot) 0)
```

Определения последних двух функций, конечно же, являются элементами списка желаний, и для них нужно спроектировать полные определения. Сделать это легко, потому что `how-many` одинаково обрабатывает и списки строк `List-of-strings`, и списки температур

List-of-temperature (почему?), а также потому, что конструирование sum производится с использованием уже известной процедуры:

```
| ; List-of-temperatures -> Number
| ; суммирует температуры в заданном списке
(define (sum alot)
  (cond
    [(empty? alot) 0]
    [else (+ (first alot) (sum (rest alot))))]))
```

Стоп! Используя пример проектирования функции average, спроектируйте функцию sum и убедитесь, что тест выполняется правильно. Затем запустите тесты для average.

На данный момент правильность определения average не вызывает сомнений просто потому, что оно прямо соответствует процедуре вычисления среднего значения, о которой рассказывается в школе. Однако мы пишем программы не только для себя, но и для других. В частности, другие должны иметь возможность прочитать сигнатуру функции, применить ее и получить информативный ответ. Но наше определение average не предполагает обработку пустых списков температур.

Выражаясь языком математики, мы бы сказали, что функция average в упражнении 143 – это частичная функция, потому что она вызывает ошибку, если ей передать '()'.

Упражнение 143. Определите с помощью DrRacket, как поведет себя функция average, если ей передать пустой список. Затем спроектируйте функцию checked-average, которая выводит информативное сообщение об ошибке при применении к '()'.

Альтернативное решение – сообщить об ограничениях в сигнатуре, указав, что average не может применяться к пустым спискам. Для этого нужно определить представление данных, исключающее '()', например:

```
| ; NEList-of-temperatures -- одно из значений:
| ; -- ???
| ; -- (cons CTemperature NEList-of-temperatures)
```

Вопрос в том, чем заменить «??», чтобы показать, что пустой список '()' не является допустимым экземпляром данных. Как вариант можно показать пример кратчайшего из допустимых списков, который длиннее пустого списка. То есть первое предложение в определении данных должно описывать все возможные списки с единственной температурой:

```
| ; NEList-of-temperatures -- одно из значений:
| ; -- (cons CTemperature '())
| ; -- (cons CTemperature NEList-of-temperatures)
| ; интерпретация: непустые списки температур в шкале Цельсия
```

Это определение отличается от предыдущих определений списков: оно отделяет важные элементы: предложение со ссылкой на само определение от предложения **без** такой ссылки. Строгое соблюдение рецепта проектирования требует составить несколько примеров не-

пустого списка температур NEList-of-temperatures, чтобы придать смысл этому определению. Как всегда, начинать следует с базового предложения, то есть пример должен выглядеть так:

```
| (cons c '())
```

где *c* обозначает температуру CTemperature, например (cons -273 '()). Кроме того, такое определение ясно показывает, что все непустые элементы списка температур List-of-temperatures тоже являются элементами нового класса данных: (cons 1 (cons 2 (cons 3 '())))) соответствует всем требованиям, если им соответствует (cons 2 (cons 3 '())), а (cons 2 (cons 3 '())) является списком температур NEList-of-temperatures, потому что (cons 3 '()), в свою очередь, является экземпляром класса NEList-of-temperatures, как было подтверждено выше. Убедитесь сами, что размер списка температур NEList-of-temperatures не имеет ограничений.

Теперь вернемся к задаче проектирования average и покажем, что эта функция предназначена исключительно для работы с непустыми списками. Определив список температур NEList-of-temperatures, мы можем правильно сформулировать сигнатуру:

Этот альтернативный подход показывает, что в данном случае можно сузить область определения average и создать totальную функцию.

```
; NEList-of-temperatures -> Number
; вычисляет среднюю температуру

(check-expect (average (cons 1 (cons 2 (cons 3 '())))))
              2)
(define (average ne-l)
  (/ (sum ne-l)
     (how-many ne-l)))
```

Естественно, все остальное остается без изменений: описание назначения, тесты и определение функции. В конце концов, сама идея вычисления среднего предполагает обработку непустого набора чисел, и в этом весь смысл нашего обсуждения.

Упражнение 144. Будут ли sum и how-many правильно обрабатывать списки NEList-of-temperature, несмотря на то что изначально они проектировались для работы с списками List-of-temperature? Если вы уверены, что они не будут правильно обрабатывать списки NEList-of-temperature, то приведите контрпримеры. Если вы уверены в обратном, то объясните почему. ■

Как бы то ни было, определение данных поднимает вопрос, как спроектировать sum и how-many с учетом того, что теперь они должны применяться к экземплярам NEList-of-temperature. Вот очевидные результаты выполнения первых трех шагов из рецепта проектирования:

```
; NEList-of-temperatures -> Number
; суммирует температуры в заданном списке
(check-expect
  (sum (cons 1 (cons 2 (cons 3 '())))) 6)
(define (sum ne-l) 0)
```

Этот пример является измененной версией примера `average`; фиктивное определение возвращает число, но ошибочное для данного теста.

Четвертый шаг – это самая интересная часть проектирования функции `sum` для обработки непустых списков `NEList-of-temperatures`. Во всех предыдущих примерах требовалось использовать макет, отличающий пустые списки от непустых из-за особой формы определения данных. Это не относится к списку температур `NEList-of-temperatures`. Здесь в обоих предложениях упоминаются непустые списки. Однако эти два предложения различаются в части `rest` списков. В частности, в первом предложении в поле `rest` используется значение `'()`, а во втором вместо него используется непустой список. Чтобы отличить экземпляры данных первого вида от данных второго вида, следует извлечь поле `rest` и проверить его с помощью `empty?`:

```
| ; NEList-of-temperatures -> Number
| (define (sum ne-l)
|   (cond
|     [(empty? (rest ne-l)) ...]
|     [else ...]))
```

Здесь `else` заменяет `(cons? (rest ne-l))`.

Далее нужно определить, должны ли оба этих предложения или одно из них обрабатывать `ne-l` как структуру. Это тот случай, когда требуется безусловное использование поля `rest` в `ne-l`. Иначе говоря, нужно добавить соответствующие селекторы в два предложения:

```
| (define (sum ne-l)
|   (cond
|     [(empty? (rest ne-l)) (... (first ne-l) ...)]
|     [else (... (first ne-l) ... (rest ne-l) ...)]))
```

Прежде чем продолжить чтение, объясните, почему выражение, возвращающее ответ в первом условии, не содержит обращения к селектору `(rest ne-l)`.

Наконец, нужно решить вопрос, касающийся ссылок определения данных на само себя. Мы знаем, что `NEList-of-temperature` содержит одну такую ссылку, поэтому макет `sum` должен включать одно рекурсивное применение:

```
| (define (sum ne-l)
|   (cond
|     [(empty? (rest ne-l)) (... (first ne-l) ...)]
|     [else
|       (... (first ne-l) ... (sum (rest ne-l) ...))]))
```

В данном случае `sum` применяется к `(rest ne-l)` во втором предложении, потому что в этой точке определение данных ссылается на само себя.

Приступая к пятому шагу проектирования, разберемся с тем, что у нас уже есть. Поскольку первое предложение в выражении `cond` выглядит значительно проще второго с его рекурсивным вызовом,

начнем с него. В этом конкретном случае, согласно условию, `sum` применяется к списку, содержащему ровно одно значение температуры (`first ne-l`). Очевидно, что это значение температуры является суммой всех температур в данном списке:

```
(define (sum ne-l)
  (cond
    [(empty? (rest ne-l)) (first ne-l)]
    [else
      (... (first ne-l) ... (sum (rest ne-l)) ...))]))
```

Во втором случае список включает, по меньшей мере, две температуры; выражение `(first ne-l)` извлекает первую из них, а `(rest ne-l)` – остальные. Кроме того, макет предлагает использование результата выражения `(sum (rest ne-l))`. Но `sum` – это определяемая нами функция, и мы пока не можем знать, как она обработает `(rest ne-l)`. Однако у нас есть описание назначения, в котором говорится, что `sum` вычисляет сумму всех температур, имеющихся в указанном списке, то есть в `(rest ne-l)`. Если это описание верно, то `(sum (rest ne-l))` сложит вместе все числа в `ne-l`, кроме одного. Чтобы получить окончательный результат, достаточно просто прибавить первую температуру к этой сумме:

```
(define (sum ne-l)
  (cond
    [(empty? (rest ne-l)) (first ne-l)]
    [else (+ (first ne-l) (sum (rest ne-l))))]))
```

Если сейчас запустить тест для данной функции, вы увидите, что наш прыжок веры был оправдан. Действительно, по причинам, которые не обсуждаются в этой книге, такой прыжок веры оправдан всегда, поэтому он является неотъемлемой частью рецепта проектирования.

Упражнение 145. Создайте предикат `sorted>?`, который принимает непустой список температур `NEList-of-temperature` и возвращает `#true`, если температуры в списке отсортированы в порядке убывания. То есть если второе значение температуры меньше первого, третье меньше второго и т. д. В противном случае предикат должен возвращать `#false`.

Подсказка. Это еще одна задача, которая легко решается с помощью табличного метода определения комбинатора. В табл. 16 приводится неполная таблица с несколькими примерами. Заполните ячейки таблицы недостающими значениями. Затем попробуйте создать выражение, вычисляющее результат по частям. ■

Упражнение 146. Спроектируйте функцию `how-many` для `NEList-of-temperature`. Это последняя функция, необходимая для завершения `average`, поэтому убедитесь, что `average` тоже успешно проходит все свои тесты. ■

Упражнение 147. Разработайте определение данных для `NEList-of-Booleans`, представляющее непустые списки логических значений. Затем повторно спроектируйте функции `all-true` и `one-true` из упражнения 140. ■

Таблица 16. Таблица для предиката sorted>?

<i>l</i>	(first <i>l</i>)	(rest <i>l</i>)	(sorted>? (rest <i>l</i>))	(sorted>? <i>l</i>)
(cons 1 (cons 2 '()))	1	???	#true	#false
(cons 3 (cons 2 '()))	3	(cons 2 '())	???	#true
(cons 0 (cons 3 (cons 2 '())))	0	(cons 3 (cons 2 '()))	???	???

Упражнение 148. Сравните определения функций из этого раздела (*sum*, *how-many*, *all-true*, *one-true*) с определениями соответствующих функций из предыдущих разделов. С какими определениями данных проще работать, которые допускают пустые списки или которые не допускают этого? Объясните почему. ■

9.3. Натуральные числа

Язык программирования BSL предлагает множество функций, которые принимают списки, и несколько функций, которые их создают. Среди них *make-list*, которая принимает число *n* и некоторое значение *v* и создает список с *n* элементами, равными *v*. Вот некоторые примеры:

```
> (make-list 2 "hello")
(cons "hello" (cons "hello" '()))
> (make-list 3 #true)
(cons #true (cons #true (cons #true '())))
> (make-list 0 17)
'()
```

Проще говоря, несмотря на то что эта функция принимает атомарные данные, она может производить фрагменты данных произвольно большого размера. Возникает вопрос: как такое возможно?

Ответ заключается в том, что *make-list* принимает на входе не просто число, а число особого вида. В детском саду вы называли эти числа «счетными числами», то есть эти числа используются для подсчета предметов. В информатике эти числа называют *натуральными числами*. В отличие от обычных чисел, натуральные числа имеют такое определение данных:

```
; N -- одно из значений:
; -- 0
; -- (add1 N)
; интерпретация: представляет счетные числа
```

Первое предложение утверждает, что 0 – это натуральное число и используется, чтобы показать, что нет ни одного объекта для подсчета. Второе предложение сообщает, что если n – натуральное число, то и $n + 1$ тоже является натуральным числом, потому что `add1` – это функция, которая прибавляет 1 к любому заданному числу. Второе предложение можно было бы записать как `(+ n 1)`, но применение `add1` должно сигнализировать о том, что это сложение является особенным.

Особенность `add1` заключается в том, что она действует скорее как конструктор из некоторого определения структуры, чем как обычная функция. По этой причине в BSL имеется также функция `sub1`, которая является «селектором», соответствующим `add1`. К любому натуральному числу m , не равному 0 , можно применить `sub1`, чтобы получить число, использовавшееся при создании m . Другими словами, `add1` похожа на `cons`, а `sub1` на `first` и `rest`.

Теперь у многих из вас может возникнуть вопрос: какие предикаты отличают 0 от других натуральных чисел, не равных 0 . По аналогии со списками имеются два предиката: `zero?`, который определяет – равно ли указанное число нулю, и `positive?`, который определяет, больше ли указанное число нуля.

Теперь вы сможете самостоятельно спроектировать функции для натуральных чисел, такие как `make-list`. Определение данных уже имеется, поэтому просто добавим заголовок:

```
; N String -> List-of-strings
; создает список, содержащий n копий строки s

(check-expect (copier 0 "hello") '())
(check-expect (copier 2 "hello")
              (cons "hello" (cons "hello" '())))
(define (copier n s)
  '())
```

Следующий шаг – разработка макета. В соответствии с определением данных тело макета `copier` должно состоять из выражения `cond` с двумя условиями: одно для значения 0 и одно для положительных чисел. Кроме того, 0 считается атомарным значением, а положительные числа – структурированными значениями, то есть второе предложение в макете должно содержать выражение с селектором. И последнее, но не менее важное: определение данных для `N` во втором предложении ссылается на само себя, поэтому второе предложение в макете должно осуществлять рекурсивное применение `copier` к соответствующему выражению с селектором:

```
(define (copier n s)
  (cond
    [(zero? n) ...]
    [(positive? n) (... (copier (sub1 n) s) ...))]))
```

Листинг 28. Создание списка копий

```
; N String -> List-of-strings
; создает список, содержащий n копий строки s

(check-expect (copier 0 "hello") '())
(check-expect (copier 2 "hello")
              (cons "hello" (cons "hello" '())))

(define (copier n s)
  (cond
    [(zero? n) '()]
    [(positive? n) (cons s (copier (sub1 n) s))]))
```

Листинг 28 содержит полное определение функции `copier`, полученное на основе ее макета. Давайте восстановим все детали этого процесса. Как всегда, начнем с предложения в выражении `cond`, которое не имеет рекурсивных вызовов. Условие в этом предложении говорит, что (это важно) входное значение равно 0, то есть функция должна создать пустой список без элементов. Работая над вторым примером, мы уже прояснили этот случай. Теперь перейдем ко второму предложению в `cond` и вспомним, что вычисляют выражения в нем:

- 1) `(sub1 n)` возвращает натуральное число, которое использовалось при конструировании заданного натурального числа `n`, которое, как мы знаем, больше 0;
- 2) `(copier (sub1 n) s)` создает список, содержащий `(sub1 n)` копий строки `s`, как указано в описании назначения.

Но функция получила число `n` и должна создать список с `n` строками `s`. Если строк в списке на одну меньше, то нетрудно догадаться, что функция должна просто прибавить строку `s` к результату `(copier (sub1 n) s)` с помощью `cons`. Именно это и делает второе предложение.

Теперь можно запустить тесты, чтобы убедиться, что функция `copier` выполняется без ошибок хотя бы в двух представленных примерах. При желании можете создать еще несколько примеров входных данных и опробовать их.

Упражнение 149. Правильно ли работает функция `copier`, если вместо строки применить ее к натуральному числу, логическому значению или изображению? Или нужно спроектировать другую функцию? Ответ на этот вопрос вы найдете в части III.

Вот альтернативное определение `copier` с использованием предложения `else`:

```
(define (copier.v2 n s)
  (cond
    [(zero? n) '()]
    [else (cons s (copier.v2 (sub1 n) s))]))
```

Как поведут себя `copier` и `copier.v2` при применении к 0.1 и "x"? Объясните. Используйте движок пошаговых вычислений в DrRacket, чтобы подтвердить свое объяснение. ■

Упражнение 150. Спроектируйте функцию `add-to-pi`. Она принимает натуральное число n и прибавляет его к π без использования элементарной операции `+`. Вот начало:

```
; N -> Number
; вычисляет (+ n pi) без использования +
(check-within (add-to-pi 3) (+ 3 pi) 0.001)

(define (add-to-pi n)
  pi)
```

Закончив определение, обобщите функцию до версии `add`, которая прибавляет натуральное число n к некоторому произвольному числу x без использования `+`. Почему в тесте используется проверка `check-within?` ■

Упражнение 151. Спроектируйте функцию `multiply`. Она принимает натуральное число n и умножает его на число x без использования `*`.

С помощью движка пошаговых вычислений в DrRacket вычислите `(multiply 3 x)` с произвольным значением x по своему выбору. Как `multiply` соотносится со знаниями арифметики, полученными в начальной школе? ■

Упражнение 152. Спроектируйте две функции: `col` и `row`.

Функция `col` принимает натуральное число n и изображение `img` и создает столбец – изображение с n копиями `img`, расположенными вертикально.

Функция `row` принимает натуральное число n и изображение `img` и создает строку – изображение с n копиями `img`, расположенными горизонтально. ■

Упражнение 153. Цель этого упражнения – визуализировать результаты студенческого протеста в стиле 1968 года. Вот примерная идея: небольшая группа студентов наполняет воздушные шарики краской, входит в какую-нибудь аудиторию и беспорядочно разбрасывает шарики. Программа должна показать точки в аудитории, куда попали шарики.

Используйте две функции из упражнения 152, чтобы создать прямоугольник размером 8×18 квадратов, каждый из которых имеет размер 10×10 пикселей. Добавьте его в пустую сцену того же размера. Это – ваша аудитория.

Спроектируйте функцию `add-balloons`. Она должна принимать список `Posn` с координатами, попадающими в границы аудитории, и создавать изображение аудитории с красными точками, координаты которых соответствуют координатам в заданном списке структур `Posn`.

На рис. 12 показан результат нашего решения этого упражнения. Слева изображена чистая аудитория, в середине – после броска двух первых шариков, а справа – крайне маловероятное распределение 10 попаданий. Но где 10-я точка? ■

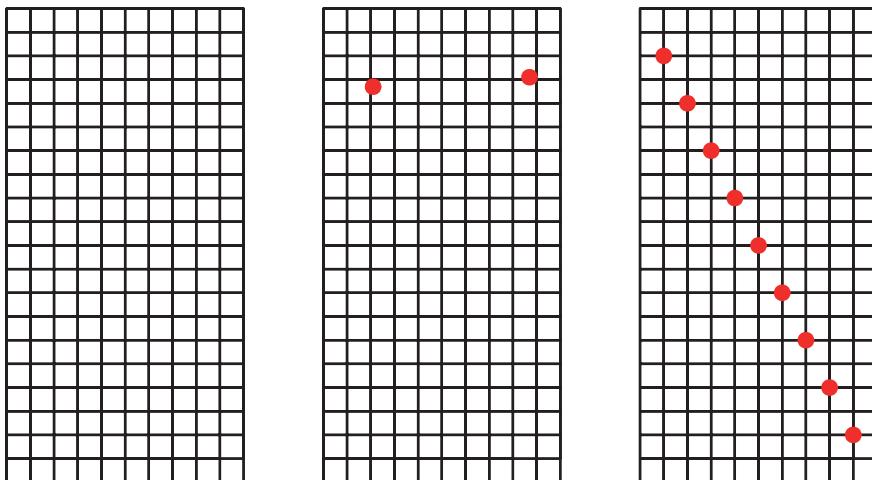


Рис. 12. Случайные атаки

9.4. Русская матрешка

В Википедии дается такое определение русской матрёшки: «Матрёшка – русская деревянная игрушка в виде расписной куклы, внутри которой находятся подобные ей куклы меньшего размера» – и иллюстрируется следующим изображением:



Задача может показаться вам слишком абстрактной или даже абсурдной; не понятно, зачем может понадобиться представлять матрешки в виде структуры данных или что делать с таким представлением. Но мы предлагаем отставить эти вопросы в сторону и просто подыграть нам.

На этом рисунке матрёшки расположены в ряд, чтобы зритель мог увидеть их все.

Теперь рассмотрим задачу представления таких матрешек в виде данных на BSL. Включив фантазию, можно представить, что мастер может создать произвольное количество матрешек, вкладываемых друг в друга, потому что для любой матрёшки можно создать матрёшку большего размера, вмещающую ее. С другой стороны, как известно, глубоко внутри находится цельная матрёшка, внутри которой ничего нет.

На каждом следующем уровне в сторону увеличения нас может интересовать множество разных характеристик: размер (хотя он связан с уровнем вложенности); цвет; орнамент на поверхности

матрешки и т. д. В этой задаче мы выберем только одну характеристику – цвет, который будем представлять в виде строки. С учетом этого каждая следующая матрешка имеет два свойства: цвет и матрешку, находящуюся внутри. Для представления информации с двумя свойствами мы всегда определяем структуры:

```
| (define-struct layer [color doll])
```

А теперь добавим определение данных:

```
| ; RD (сокращенно от Russian Doll – русская матрешка) -- это одно из значений:  
| ; -- Стока  
| ; -- (make-layer String RD)
```

Естественно, первое предложение в этом определении данных представляет самую внутреннюю матрешку, а точнее ее цвет. Второе предложение описывает каждую следующую матрешку, в которую вложена данная матрешка. Это предложение представляет ее в виде экземпляра `layer`, который, как следует из определения, содержит цвет и еще одно поле: матрешку, которая находится внутри этой матрешки.

Взгляните на следующее схематическое изображение:



Здесь изображены три матрешки. Красная – самая внутренняя, зеленая – в середине и желтая – текущая внешняя матрешка. Чтобы представить эту матрешку экземпляром `RD`, можно начать с любого конца. Давайте пойдем изнутри. Красную матрешку легко представить в виде `RD`. Поскольку внутри ничего нет и матрешка имеет красный цвет, для ее представления достаточно строки "red". Вторую матрешку можно представить так:

```
| (make-layer "green" "red")
```

Это представление говорит, что зеленая (полая) матрешка содержит красную матрешку. Наконец, чтобы получить самую внешнюю матрешку, просто заключим эту матрешку в еще одну:

```
| (make-layer "yellow" (make-layer "green" "red"))
```

Этот процесс дает хорошее представление о том, как перейти от любого набора цветных русских матрешек к представлению данных.

Но имейте в виду, что программист должен уметь делать и обратное, то есть переходить от конкретных данных к информации. Чтобы проверить это свое умение, нарисуйте схематичную матрешку для следующего экземпляра RD:

```
| (make-layer "pink" (make-layer "black" "white"))
```

При желании можете даже попробовать реализовать это на языке BSL.

Теперь, когда мы создали определение данных и понимаем, как представлять настоящие куклы и как интерпретировать экземпляры RD, мы готовы приступить к проектированию функций, принимающих RD. В частности, спроектируем функцию, которая подсчитывает, сколько матрешек содержится в заданном наборе. Это описание является прекрасным описанием назначения и одновременно определяет сигнатуру:

```
| ; RD -> Number
| ; сколько матрешек в данном наборе an-rd
```

Сначала определим пример данных и начнем с (make-layer "yellow" (make-layer "green" "red")). Изображение выше говорит нам, что ожидаемый ответ – 3, потому что имеется три матрешки: красная, зеленая и желтая. Этот пример также говорит нам, что когда входные данные представляют следующую матрешку



то ожидаемый ответ – 1.

Четвертый шаг требует создания макета. Следуя стандартной процедуре для этого шага, получаем следующий макет:

```
| ; RD -> Number
| ; сколько матрешек в данном наборе an-rd
(define (depth an-rd)
  (cond
    (cond
      [(string? an-rd) ...]
      [(layer? an-rd)
        (... (layer-color an-rd) ...
             ... (depth (layer-doll an-rd)) ...)])))
```

Количество предложений в cond задается количеством предложений в определении RD. В каждом предложении конкретно указывается, о каких данных идет речь и что мы должны использовать предикаты string? и layer?. Строки не являются составными данными, но экземпляры layer содержат два значения. Если функции нужны эти значения, она должна использовать селекторы: (layer-color an-rd) и (layer-doll an-rd). Наконец, второе предложение в определении

данных содержит ссылку на само определение в поле *doll* структуры *layer*. Следовательно, нам нужно рекурсивно применить функцию ко второму выражению с селектором.

Примеры и макет диктуют определение функции. Для нерекурсивного предложения в *cond* ответ, очевидно, равен 1. Рекурсивное предложение в макете вычисляет следующие результаты:

- (*layer-color an-rd*) извлекает строку, описывающую цвет текущей матрешки;
- (*layer-doll an-rd*) извлекает матрешку, содержащуюся в текущей;
- (*depth (layer-doll an-rd)*) определяет, сколько матрешек содержится в (*layer-doll an-rd*), в соответствии с описанием назначения функции *depth*.

Это последнее число близко к желаемому ответу, но не является им, потому что разница между *an-rd* и (*layer-doll an-rd*) составляет один уровень вложенности, то есть одну дополнительную матрешку. Иначе говоря, функция должна прибавить 1 к результату рекурсивного применения, чтобы получить фактический ответ:

```
; RD -> Number
; сколько матрешек в данном наборе an-rd
(define (depth an-rd)
  (cond
    [(string? an-rd) 1]
    [else (+ (depth (layer-doll an-rd)) 1)]))
```

Обратите внимание, что определение функции не использует (*layer-color an-rd*) во втором предложении. И снова мы видим, что макет отражает схему организации данных, но при этом нам могут понадобиться не все части фактического определения.

Теперь преобразуем примеры в тесты:

```
(check-expect (depth "red") 1)
(check-expect
  (depth
    (make-layer "yellow" (make-layer "green" "red")))
  3)
```

Если выполнить этот код в DrRacket, то можно увидеть, что при его выполнении затрагиваются все части определения *depth*.

Упражнение 154. Спроектируйте функцию *colors*. Он должна принимать матрешку и возвращать строку, перечисляющую все цвета этой и всех вложенных матрешек через запятую и пробел. Для нашего конкретного примера в результате должна получиться строка:

```
| "yellow, green, red" ■
```

Упражнение 155. Спроектируйте функцию *inner*, которая принимает RD и возвращает самую внутреннюю матрешку (ее цвет). Ис-

пользуйте движок пошаговых вычислений в DrRacket, чтобы вычислить (`(inner rd)`) для значения `rd` по вашему выбору. ■

9.5. Списки в интерактивных программах

Применение списков и определений данных, которые ссылаются на самих себя, позволяет проектировать и создавать гораздо более интересные интерактивные программы, чем те, что используют конечное

Если вы забыли, как проектируются и создаются интерактивные программы, вернитесь к разделу 3.6.

количество данных. Только представьте, что вы можете создать версию программы космических захватчиков из главы 6, которая позволяет игроку произвести столько выстрелов из танка, сколько он пожелает. Начнем с упрощенной версии этой задачи.

Задача. Спроектируйте интерактивную программу, имитирующую стрельбу. Каждый раз, когда «игрок» нажимает клавишу пробела, программа должна производить выстрел из нижней части холста. Выстреливаемые снаряды должны подниматься вертикально вверх со скоростью один пиксель за такт часов.

Проектирование интерактивной программы начинается с разделения информации на константы и элементы постоянно меняющегося состояния мира. В первом случае мы определим физические и графические константы; а для второго случая разработаем представление данных, описывающее состояние мира. Несмотря на некоторую расплывчатость постановки задачи, она явно предполагает наличие прямоугольной сцены со снарядами, перемещающимися вертикально вверх. Очевидно, что местоположение снарядов будет меняться с каждым тактом часов, но размер сцены и координата X пуска снарядов остаются неизменными:

```
(define HEIGHT 80) ; расстояния в пикселях
(define WIDTH 100)
(define XSHOTS (/ WIDTH 2))

; графические константы
(define BACKGROUND (empty-scene WIDTH HEIGHT))
(define SHOT (triangle 3 "solid" "red"))
```

Эти конкретные значения не указываются в постановке задачи, но мы легко можем исправить ситуацию в будущем, изменив пару определений, поэтому можно считать, что мы достигли нашей цели.

В постановке задачи упоминаются два аспекта «мира», которые меняются с течением времени. Во-первых, нажатие клавиши пробела производит выстрел и добавляет в сцену новый снаряд. Во-вторых, все снаряды летят вертикально вверх со скоростью один пиксель за такт часов. Мы не можем заранее предсказать, сколько выстрелов сделает игрок, поэтому используем список для их представления:

```
| ; List-of-shots (список выпущенных снарядов) -- одно из значений:
| ; -- '()
| ; -- (cons Shot List-of-shots)
; интерпретация: коллекция летящих снарядов
```

Остается решить еще один вопрос: как представить каждый отдельный снаряд. Мы уже знаем, что все они имеют одну и ту же координату X, которая никогда не меняется. Кроме того, все снаряды выглядят одинаково. Единственное, чем они отличаются друг от друга, – это координата Y. Поэтому каждый снаряд можно представить в виде числа:

```
| ; Shot (снаряд) -- это Number (число).
; интерпретация: представляет координату Y снаряда
```

Мы могли бы ограничить представление снарядов интервалом чисел меньше HEIGHT, потому что знаем, что все снаряды выстреливаются из нижней части холста и затем перемещаются вертикально вверх, то есть их координата Y непрерывно уменьшается.

Соответственно, для представления мира можно использовать такое определение данных:

```
| ; ShotWorld -- список чисел List-of-numbers.
; интерпретация: каждое число в этом списке
; представляет координату Y снаряда
```

Итак, два определения данных выше описывают списки чисел. У нас уже есть определение списков чисел, а имя ShotWorld сообщает назначение этого класса данных.

Следующая наша задача после определения констант и представления данных, описывающего состояния мира, состоит в том, чтобы выбрать обработчики событий и адаптировать их сигнатуры к данной проблеме. В текущем примере упоминаются такты часов и клавиша пробела; все это подталкивает нас к тому, чтобы добавить в список желаний три функции:

- функцию, превращающую состояние мира в изображение:

```
| ; ShotWorld -> Image
| ; для каждого значения у в списке w добавляет в сцену
| ; изображение снаряда в позиции (MID,y}
(define (to-image w) BACKGROUND)
```

потому что постановка задачи требует визуального отображения снарядов;

- функцию, обрабатывающую каждый тик часов:

```
| ; ShotWorld -> ShotWorld
| ; перемещает каждый снаряд из списка w вверх на один пиксель
(define (tock w) w)
```

- и функцию для обработки событий клавиатуры:

```
| ; ShotWorld KeyEvent -> ShotWorld
| ; добавляет новый снаряд в сцену,
| ; если игрок нажал клавишу пробела
| (define (keyh w ke) w)
```

Не забывайте, что кроме функций в списке желаний нам также нужно определить основную функцию, которая подготавливает мир и устанавливает обработчики. Листинг 29 включает эту функцию, которая единственная не была разработана нами, потому что она является лишь модификацией стандартной схемы.

Начнем с проектирования функции `to-image`. У нас уже есть ее сигнатура, описание назначения и заголовок, поэтому теперь мы должны определить примеры. Поскольку определение данных включает два предложения, нужно создать как минимум два примера: '() и список координат, скажем (`cons 9 '()`). Ожидаемым результатом для '(), очевидно, является первоначальная сцена `BACKGROUND`, а при наличии координат Y в списке функция должна поместить изображение снаряда в позицию `MID` с указанной координатой Y:

```
| (check-expect (to-image (cons 9 '()))
|                 (place-image SHOT XSHOTS 9 BACKGROUND))
```

Прежде чем продолжить чтение, исследуйте пример, который применяет `to-image` к списку с двумя снарядами. Это поможет понять, как работает функция.

Четвертый шаг – преобразование определения данных в макет:

```
| ; ShotWorld -> Image
| (define (to-image w)
|   (cond
|     [(empty? w) ...]
|     [else
|       (... (first w) ... (to-image (rest w)) ...)]))
```

Макет функции, соответствующий определению списка данных, теперь настолько знаком, что не требует подробных объяснений. Если у вас есть сомнения, прочитайте вопросы в табл. 10 и создайте макет самостоятельно.

На основе макета легко определить функцию. Основная идея состоит в том, чтобы объединить примеры с макетом и ответить на вопросы из табл. 11. Следуя установленному порядку, начнем с базового случая – пустого списка снарядов. Как мы уже знаем из примеров, ожидаемый результат – пустая сцена `BACKGROUND`. Затем определим, что должны вычислять выражения во втором условии в `cond`:

- (`first w`) извлекает первую координату из списка;
- (`rest w`) – остальные координаты;
- (`(to-image (rest w))`) добавляет в сцену снаряды из оставшейся части списка, согласно описанию назначения `to-image`.

Иначе говоря, (`(to-image (rest w))`) добавляет в сцену оставшиеся снаряды в списке и, соответственно, выполняет почти всю рабо-

ту. Единственное, чего не хватает, – отображения первого снаряда (`first w`). Если теперь применить описание назначения к этим двум выражениям, мы получим желаемое выражение для второго предложения `cond`:

```
| (place-image SHOT XSHOTS (first w)
|   (to-image (rest w)))
```

Добавленный значок – это стандартное изображение снаряда; две координаты прописаны в описании назначения, а последний аргумент `place-image` – это изображение, созданное из остальной части списка.

В листинге 29 приводится полное определение функции `to-image`, а также остальная часть программы. Функция `tock` проектируется точно так же, как `to-image`, и мы оставляем вам этот процесс как самостоятельное упражнение. Но на обработчике `keyh` мы остановимся подробнее. Сигнатура этой функции ставит интересный вопрос. Она указывает, что обработчик принимает два аргумента с нетривиальными определениями данных. С одной стороны, `ShotWorld` – это определение данных, ссылающееся само на себя. С другой стороны, `KeyEvents` – это фактически перечисление. Пока мы можем только «догадываться», какой из двух аргументов определяет структуру макета, но позднее мы подробно изучим такие ситуации.

Обработчик событий, такой как `keyh`, обрабатывает нажатия клавиш. Следовательно, будем считать, что его основным аргументом является событие от клавиатуры, и будем строить макет на его основе. В частности, следуя определению данных для событий клавиатуры `KeyEvent` из раздела 4.3, функция должна включать выражение `cond` со множеством предложений, подобных следующим:

```
(define (keyh w ke)
  (cond
    [(key=? ke "left") ...]
    [(key=? ke "right") ...]
    ...
    [(key=? ke " ") ...]
    ...
    [(key=? ke "a") ...]
    ...
    [(key=? ke "z") ...]))
```

Листинг 29. Интерактивная программа с состоянием в виде списка

```
; ShotWorld -> ShotWorld
(define (main w0)
  (big-bang w0
    [on-tick tock]
    [on-key keyh]
    [to-draw to-image]))

; ShotWorld -> ShotWorld
; перемещает каждый снаряд вверх на один пиксель
(define (tock w)
```

```

(cond
  [(empty? w) '()]
  [else (cons (sub1 (first w)) (tock (rest w))))]

; ShotWorld KeyEvent -> ShotWorld
; добавляет новый снаряд в сцену,
; если игрок нажал клавишу пробела
(define (keyh w ke)
  (if (key=? ke " ") (cons HEIGHT w) w))

; ShotWorld -> Image
; для каждого значения у в списке w добавляет в сцену BACKGROUND
; изображение снаряда в позицию (XSHOTS,y)
(define (to-image w)
  (cond
    [(empty? w) BACKGROUND]
    [else (place-image SHOT XSHOTS (first w)
                       (to-image (rest w))))]))

```

Конечно, подобно функциям, принимающим все возможные значения, обработчику событий клавиатуры обычно не нужно проверять все возможные варианты. В данном случае мы знаем, что обработчик реагирует только на клавишу пробела, а все остальные игнорируются. Поэтому мы можем объединить все условия `cond` в предложение `else` и отдельно обрабатывать только пробел " ".

Упражнение 156. Добавьте тесты для программы в листинге 29 и убедитесь, что все они выполняются успешно. Объясните, что делает функция `main`. Затем запустите программу, вызвав функцию `main`. ■

Упражнение 157. Поэкспериментируйте и определите, легко ли изменить принятые нами решения относительно констант. Например, определите, приводит ли изменение одной константы к желаемому результату:

- увеличьте высоту холста на 220 пикселей;
- увеличьте ширину холста на 30 пикселей;
- измените координату X линии выстрелов на «где-нибудь слева от середины»;
- измените фон сцены на зеленый;
- измените отображение снарядов в виде красного вытянутого прямоугольника.

Также проверьте, можно ли увеличить размер снаряда вдвое, ничего не меняя, или изменить его цвет на черный. ■

Упражнение 158. Запустите функцию `main`, нажмите клавишу пробела (произведите выстрел) и подождите некоторое время, пока снаряд не исчезнет за верхним краем холста. Даже притом что снаряд покинул сцену, он все еще будет содержаться в списке, описывающем состояние мира.

Спроектируйте альтернативную функцию `tock`, которая не просто перемещает снаряды на один пиксель с каждым тиком часов, но также удаляет снаряды, пересекшие верхнюю границу сцены. Под-

сказка. Для этого можно спроектировать вспомогательную функцию с рекурсивным предложением в cond. ■

Упражнение 159. Превратите решение упражнения 153 в интерактивную программу. Основная функция этой программы, с именем `riot` (бунт), должна принимать количество разбрасываемых шариков и отображать сцену с опускающимися шариками, которые появляются из-за верхнего края по одному в секунду. Функция должна создать список структур `Posn` с координатами точек падения шариков.

Подсказки. (1) Вот одно из возможных представлений данных:

```
(define-struct pair [balloon# lob])
; Pair -- это структура (make-pair N List-of-posns)
; List-of-posns -- одно из значений:
; -- '()
; -- (cons Posn List-of-posns)
; интерпретация: (make-pair n lob) означает n шариков,
; которые нужно бросить
```

(2) Выражение `big-bang` – это самое обычное выражение. Его можно вложить в другое выражение.

(3) Напомним, что случайные числа можно получить с помощью функции `random`. ■

9.6. Замечания о списках и множествах

Эта книга опирается на интуитивное понимание **множеств** как коллекций значений BSL. В разделе 5.7 конкретно говорится, что определение данных вводит имя для множества значений BSL. В этой книге постоянно задается один и тот же вопрос о множествах: входит ли какой-либо элемент в некоторое множество. Например, 4 входит в множество чисел, а строка «четыре» – нет. В книге также показано, как использовать определения данных для проверки, является ли какое-либо значение членом некоторого именованного множества, и как использовать некоторые определения данных для создания выборки из множества, но эти две процедуры касаются определений данных, а не множеств как таковых.

В то же время списки представляют коллекции значений. Следовательно, вам может быть интересно узнать, отличаются ли списки от множеств или это различие несущественное. Если да, то этот раздел для вас.

На данный момент основное различие между множествами и списками состоит в том, что множество – это идея, которую мы используем для обсуждения этапов проектирования кода, а список – одна из многих форм данных в BSL – выбранном нами языке программирования. Эти два понятия используются в наших разговорах на довольно разных уровнях. Однако, учитывая, что определение данных вводит представление фактической информации на языке BSL и множества

Большинство полноценных языков напрямую поддерживают представления данных в виде списков и множеств.

являются коллекциями информации, вы можете спросить: как внутри BSL представить множества в виде данных. В отличие от списков, множества не имеют особого статуса в BSL, но в то же время множества чем-то напоминают списки. Ключевое различие заключается в том, какие функции используют эти формы данных. В BSL имеется несколько констант и функций для работы со списками, например `empty`, `empty?`, `cons`, `cons?`, `first`, `rest`. Также есть возможность определить свои функции, например `member?`, `length`, `remove`, `reverse` и т. д. Вот пример функции, которая отсутствует в языке BSL, но которую можно определить самостоятельно:

```
| ; List-of-string String -> N
| ; подсчитывает количество вхождений строки s в список los
(define (count los s)
  0)
```

Стоп! Завершите проектирование этой функции.

Продолжим наши рассуждения и без лишних усложнений скажем, что множества – это фактически списки. А чтобы еще больше упростить рассуждения, сосредоточимся на списках чисел. Если согласиться с тем, что для нас имеет значение, является ли число частью множества или нет, то почти сразу становится ясно, что списки можно использовать для представления множеств двумя разными способами.

В табл. 17 показаны два определения данных. Оба говорят, что множество представлено списком чисел. Разница в том, что определение справа имеет ограничение, согласно которому никакое число не может присутствовать в списке более одного раза. В конце концов, ключевой вопрос, который мы задаем о множестве: присутствует ли какое-то число в множестве или нет, и не имеет значения, входит ли оно в набор один, два или три раза.

Таблица 17. Два представления множеств

;	Son.L -- одно из значений:	;	Son.R -- одно из значений:
;	-- пустой список	;	-- пустой список
;	-- (cons Number Son.L)	;	-- (cons Number Son.R)
;		;	
;	Son используется, когда	;	Ограничение: если s -- это Son.R, то никакое
;	применяется к Son.L и Son.R	;	число не может присутствовать дважды в s

Независимо от выбранного варианта, мы можем определить два важных понятия:

```
| ; Son (множество чисел)
(define es '())
|
| ; Number Son -> Boolean
| ; присутствует ли x в s
```

```
| (define (in? x s)
  (member? x s))
```

Первое – это **пустое множество**, которое в обоих случаях представлено пустым списком. Второе – проверка на членство.

Один из способов создать большое множество – использовать `cons` и приведенные выше определения. Допустим, мы решили создать представление множества, содержащего числа 1, 2 и 3. Вот одно из таких представлений:

```
| (cons 1 (cons 2 (cons 3 '()))))
```

Оно остается верным для обоих представлений данных. Но можно ли сказать, что следующее представление

```
| (cons 2 (cons 1 (cons 3 '()))))
```

определяет то же самое множество? А такое представление?

```
| (cons 1 (cons 2 (cons 1 (cons 3 '())))))
```

В первом случае ответ утвердительный, потому что главное – входит ли число в множество или нет. Но во втором случае ситуация меняется: даже при том что порядок элементов в списке не имеет значения, ограничение в правом определении данных не позволяет отнести последний список к множеству Son.R, потому что он содержит два экземпляра 1.

Разница между двумя определениями данных проявляется при проектировании функций. Представьте, что нам нужна функция, которая удаляет число из множества. Вот соответствующая запись в списке желаний, которая относится к обоим представлениям:

```
| ; Number Son -> Son
| ; вычитает x из s
(define (set- x s)
  s)
```

В описании назначения используется слово «вычитает», потому что именно так логики и математики обозначают операцию удаления элемента из множества.

В табл. 18 показаны результаты проектирования. Они имеют два отличия:

- 1) в тесте слева используется список, содержащий два числа 1, а в тесте справа то же множество определяется с помощью единственного оператора `cons`;
- 2) из-за этих различий слева должна использоваться функция `remove-all`, а справа – `remove`.

Стоп! Скопируйте код в область определений DrRacket и убедитесь, что тесты выполняются успешно. Затем читайте далее и продолжайте экспериментировать с кодом вместе с нами.

Таблица 18. Функции для двух представлений множеств

<pre>; Number Son.L -> Son.L ; удаляет x из s (define s1.L (cons 1 (cons 1 '())))</pre>	<pre>; Number Son.R -> Son.R ; удаляет x из s (define s1.R (cons 1 '()))</pre>
<pre>(check-expect (set-.L 1 s1.L) es)</pre>	<pre>(check-expect (set-.R 1 s1.R) es)</pre>
<pre>(define (set-.L x s) (remove-all x s))</pre>	<pre>(define (set-.R x s) (remove x s))</pre>

Неприятный аспект табл. 18 заключается в том, что тесты используют в качестве ожидаемого результата простой список `es`. На первый взгляд эта проблема может показаться несущественной. Однако взгляните на следующий пример:

```
| (set- 1 set123)
```

где `set123` представляет множество чисел 1, 2 и 3 одним из двух способов:

```
| (define set123-version1
  (cons 1 (cons 2 (cons 3 '()))))
| (define set123-version2
  (cons 1 (cons 3 (cons 2 '()))))
```

Независимо от выбора представления, выражение `(set- 1 set123)` возвращает один из двух списков:

```
| (define set23-version1
  (cons 2 (cons 3 '())))
| (define set23-version2
  (cons 3 (cons 2 '())))
```

Но мы не можем предсказать, какое из этих двух множеств вернет `set-`.

Для простого случая с двумя альтернативами можно использовать такой тест `check-member-of`:

```
| (check-member-of (set-.v1 1 set123.v1)
  set23-version1
  set23-version2)
```

Если ожидаемое множество содержит три элемента, то число возможных вариантов увеличивается до шести, если не считать представлений с повторяющимися элементами, которые допускаются определением данных слева.

Для решения этой проблемы необходимо объединить две идеи. Во-первых, давайте вспомним, что на самом деле `set-` гарантирует отсутствие указанного элемента в результате. Наш способ преобра-

зования примеров в тесты не реализует эту идею. Во-вторых, эту идею можно точно сформулировать на языке BSL с помощью теста `check-satisfied`.

В интермеццо 1 мы уже упоминали, что тест `check-satisfied` определяет, удовлетворяет ли выражение определенному свойству. Свойство – это функция, принимающая некоторое значение и возвращающая логическое значение. В данном случае мы хотим заявить, что 1 не является членом некоторого множества:

```
| ; Son -> Boolean
| ; #true, если 1 является членом s; иначе #false
| (define (not-member-1? s)
|   (not (in? 1 s)))
```

С помощью `not-member-1?` мы можем сформулировать тест, как показано ниже:

```
| (check-satisfied (set- 1 set123) not-member-1?)
```

и в этом варианте четко указано, что именно должна сделать функция. Более того, эта формулировка не зависит от того, как представлено входное или выходное множество.

Подводя итог, можно сказать, что списки и множества оба являются коллекциями значений, но имеют существенные отличия:

Свойство	Списки	Множества
членство	одно из многих	имеет критическое значение
порядок	имеет критическое значение	не имеет значения
количество вхождений	имеет значение	не имеет значения
размер	конечный, но произвольный	конечный или бесконечный

Последняя строка в этой таблице представляет хоть и новую, но все же очевидную идею. Многие из множеств, упоминаемых в данной книге, бесконечно велики, например число, строка или список строк `List-of-strings`. Список, напротив, всегда конечен, хотя может содержать сколь угодно большое количество элементов.

В этом разделе объяснялись существенные различия между множествами и списками, а также два способа представления конечных множеств с помощью конечных списков. Язык BSL недостаточно выразителен для представления бесконечных множеств; упражнение 299 познакомит вас с совершенно другим представлением множеств, которое также может выражать бесконечные множества. Однако вопрос о том, как реальные языки программирования представляют множества, выходит за рамки этой книги.

Упражнение 160. Спроектируйте функции `set+L` и `set+R`, создающие множество, добавляя число `x` в некоторое заданное множество `s`, для левого и правого определений данных соответственно. ■

10. Еще о списках

Списки – универсальная форма данных, доступная практически во всех языках программирования. Программисты используют их при создании больших приложений, искусственного интеллекта, распределенных систем и многое другое. В этой главе мы познакомимся с некоторыми идеями из мира списков, включая функции, создающие списки, представления данных со структурами внутри списков и способы представления текстовых файлов в виде списков.

10.1. Функции, создающие списки

Вот пример функции, вычисляющей сумму заработной платы на основе почасовой ставки:

```
| ; Number -> Number
| ; вычисляет заработную плату за h рабочих часов
(define (wage h)
  (* 12 h))
```

Он принимает количество отработанных часов и вычисляет сумму заработной платы за это время. Однако компании, заказавшей программное обеспечение для расчета заработной платы, не интересна эта функция. Ей нужна функция, вычисляющая заработную плату для всех ее сотрудников.

Назовем эту новую функцию `wage*`. Ее задача – обработать все часы, отработанные сотрудниками, и вычислить заработную плату, причитающуюся каждому из них. Для простоты предположим, что функции передается список чисел, каждое из которых представляет количество часов, отработанное одним сотрудником в течение недели, и на выходе должен получиться список с причитающимися суммами заработной платы, также являющийся списком чисел.

Поскольку у нас уже есть определение данных, описывающее входные и выходные данные, мы можем сразу перейти ко второму шагу проектирования:

```
| ; List-of-numbers -> List-of-numbers
| ; вычисляет недельную зарплату сотрудников с учетом почасовой ставки
(define (wage* whrs)
  '())
```

Далее нужно составить несколько примеров входных данных и ожидаемых результатов. Итак, сконструируем несколько коротких списков чисел, представляющих часы, отработанные за неделю:

Дано	Ожидается
'()	'()
(cons 28 '())	(cons 336 '())
(cons 4 (cons 2 '()))	(cons 48 (cons 24 '()))

Чтобы вычислить результат, мы определяем недельную заработную плату для каждого числа в данном входном списке. В первом примере во входном списке нет чисел, поэтому на выходе получится '(). Убедитесь, что понимаете, почему второй и третий ожидаемые результаты – это то, что нам нужно.

Учитывая, что `wage*` получает те же данные, что и некоторые другие функции из главы 8, и макет зависит только от формы определения данных, мы можем повторно использовать следующий макет:

```
| (define (wage* whrs)
  (cond
    [(empty? whrs) ...]
    [else (... (first whrs) ...
                ... (wage* (rest whrs)) ...)]))
```

Если вы захотите попрактиковаться в разработке шаблонов, используйте вопросы из табл. 10.

Пришло время для самого творческого шага в процессе проектирования. Следуя рецепту, рассмотрим каждое предложение `cond` отдельно. Нерекурсивный случай, когда выполняется условие `(empty? whrs)`, означает, что функция получила пустой список '(). Согласно примерам выше, в такой ситуации функция должна вернуть '().

Во втором случае рецепт проектирования требует указать, что вычисляет каждое выражение в макете:

- `(first whrs)` возвращает первое число из списка `whrs`, то есть первое количество отработанных часов;
- `(rest whrs)` – оставшаяся часть данного списка;
- `(wage * (rest whrs))` вызывает функцию, которую мы сейчас проектируем, для обработки оставшейся части списка. Как обычно, используем ее сигнатуру и описание назначения, чтобы определить результат этого выражения. Сигнатурой говорит нам, что эта функция принимает список чисел, а описание назначения объясняет, что в результате возвращается список с суммами заработной платы, соответствующими отработанным часам, указанным во входном списке.

Ключ к успеху состоит в том, чтобы довериться этим фактам при формулировании выражения, вычисляющего результат в этом случае, даже притом что функция еще не определена.

Поскольку у нас уже есть список сумм заработной платы для всех, кроме первого элемента в `whrs`, функция должна выполнить два действия, чтобы получить ожидаемый результат для всего списка `whrs`: вычислить недельную заработную плату для `(first whrs)` и сконструировать список, представляющий недельные заработные платы для исходного списка `whrs`. В первом действии мы повторно используем `wage`. Во втором – объединяем две части информации в один список:

```
| (cons (wage (first whrs)) (wage* (rest whrs)))
```

Законченное определение функции приводится в листинге 30.

Листинг 30. Вычисление заработной платы для всех сотрудников

```

; List-of-numbers -> List-of-numbers
; вычисляет недельную зарплату сотрудников с учетом почасовой ставки
(define (wage* whrs)
  (cond
    [(empty? whrs) '()]
    [else (cons (wage (first whrs)) (wage* (rest whrs))))]

  ; Number -> Number
  ; вычисляет заработную плату за h рабочих часов
  (define (wage h)
    (* 12 h)))

```

Упражнение 161. Преобразуйте примеры в тесты и убедитесь, что все они выполняются успешно. Затем измените функцию в листинге 30 так, чтобы сотрудники получали по 14 долларов за час работы. После этого реорганизуйте программу так, чтобы впоследствии можно было изменять величину заработной платы изменением единственного значения в программе. ■

Покажите результаты, получаемые на разных шагах рецепта проектирования. Если вы застопорились на каком-то шаге, покажите кому-нибудь, как далеко вы продвинулись, следя за рецепту проектирования.
Рецепт – это не просто инструмент проектирования, которым вы можете пользоваться; это также система диагностики, которая позволит другим помочь себе.

Упражнение 162. Никакие сотрудники не могут работать более 100 часов в неделю. Чтобы защитить компанию от мошенничества, функция должна проверить каждый элемент списка, который получает функция `wage*`, чтобы он не превышал 100. Если обнаружится хотя бы один такой элемент, функция должна немедленно сигнализировать об ошибке. Как следует изменить функцию в листинге 30, чтобы выполнить эту простую проверку? ■

Упражнение 163. Спроектируйте функцию `convertF`. Она должна принимать список температур по шкале Фаренгейта и возвращать список соответствующих температур по шкале Цельсия. ■

Упражнение 164. Спроектируйте функцию `convert-euro`, которая принимает список с денежными суммами в долларах США и возвращает список соответствующих денежных сумм в евро. Посмотрите текущий обменный курс в интернете.

Обобщите функцию `convert-euro` до функции `convert-euro*`, которая принимает обменный курс и список с денежными суммами в долларах США и возвращает список соответствующих денежных сумм в евро. ■

Упражнение 165. Спроектируйте функцию `subst-bot`, которая принимает списки описаний игрушек (каждое описание – это строка с единственным словом) и заменяет все вхождения слова "бот" на "г2д2"; все остальные описания должны оставаться неизменными.

Обобщите функцию `subst-bot` до функции `substitute`, которая принимает две строки – `new` и `old` и список описаний для обработки. Последний потребляет две строки, называемые новой и старой, и список строк. Она должна вернуть новый список строк, заменив все вхождения `old` на `new`. ■

10.2. Структуры в списках

Представление рабочей недели в виде числа – не лучший выбор, потому что для печати зарплатной ведомости требуется больше информации, чем простое количество часов, отработанных за неделю. Кроме того, разные сотрудники имеют разную почасовую ставку. К счастью, элементы списка могут быть не только атомарными, но и любыми другими значениями, включая структуры.

Наш текущий пример требует именно такого представления данных. Вместо чисел мы используем структуры, представляющие сотрудников, плюс отработанные ими часы работы и почасовые ставки:

```
| (define-struct work [employee rate hours])
; Work -- это структура:
;   (make-work String Number Number)
; интерпретация: (make-work n r h) объединяет имя n,
; почасовую ставку r и количество отработанных часов h
```

Это довольно упрощенное представление, но, несмотря на простоту, оно порождает дополнительную проблему, заставляя нас сформулировать определение данных для списков, содержащих структуры:

```
| ; Low (List of works -- список работ) -- это одно из значений:
| ; -- '()
| ; -- (cons Work Low)
; интерпретация: экземпляр Low представляет
; часы, отработанные сотрудниками компании
```

Вот три примера списка Low:

```
'()

(cons (make-work "Robby" 11.95 39)
      '())
(cons (make-work "Matthew" 12.95 45)
      (cons (make-work "Robby" 11.95 39)
            '()))
```

Используя это определение данных, объясните, почему эти примеры данных относятся к классу Low.

Стоп! Создайте еще пару примеров данных, соответствующих этому определению.

Теперь, прочувствовав определение Low, можно приступить к переделке функции `wage*`, чтобы вместо списков чисел она принимала списки Low:

```
; Low -> List-of-numbers
; вычисляет недельную зарплату по заданному списку записей
(define (wage*.v2 an-low)
  '())
```

Суффикс «`.v2`» в конце имени функции сообщает читателям кода, что это вторая, переделанная версия функции. В данном случае пере-

При работе над реальными проектами вместо подобных суффиксов вы будете использовать инструменты управления разными версиями кода.

делка начинается с определения новой сигнатуры и адаптации описания назначения. Заголовок при этом не изменился.

Третий шаг рецепта проектирования – проработка примеров. Начнем со второго списка из представленных выше. Он содержит одну запись Work, а именно (`(make-work "Robby" 11.95 39)`). Она интерпретируется так: "Robby" проработал 39 часов и за каждый час получает 11,95 доллара. То есть его зарплата за неделю составляет 466,05 доллара, или $(* 11.95 39)$. Следовательно, `wage*.v2` должна вернуть `(cons 466.05 '())`. Естественно, если бы входной список содержал две записи Work, функция выполнила бы два подобных вычисления и вернула в результате список с двумя числами.

Стоп! Определите ожидаемый результат для третьего примера из представленных выше.

ЗАМЕЧАНИЕ О ЧИСЛАХ. Имейте в виду, что BSL, в отличие от большинства других языков программирования, воспринимает десятичные числа точно так же, как и вы, а именно как точные дроби. Но подобное выражение на другом языке, таком как Java, вернуло бы для первой записи значение $466,04999999999995$. Поскольку заранее нельзя предсказать, когда операции с десятичными числами поведут себя таким странным образом, подобные примеры лучше записывать так:

```
(check-expect
  (wage*.v2
    (cons (make-work "Robby" 11.95 39) '())
    (cons (* 11.95 39) '()))
```

просто чтобы подготовиться к странностям в других языках программирования. Кроме того, оформление примеров в таком стиле также показывает, что вы действительно поняли, как вычисляется заработная плата. **КОНЕЦ.**

Теперь перейдем к разработке макета. Воспользовавшись шаблонными вопросами, вы быстро получите следующее:

```
(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) ...]
    [(cons? an-low)
      (... (first an-low) ...
           ... (wage*.v2 (rest an-low)) ...)])
```

потому что определение данных состоит из двух предложений, в первом из которых используется пустой список `'()`, а во втором список структур. Но мы также знаем о входных данных больше, чем выражено в этом макете. Например, мы знаем, что `(first an-low)` извлекает из данного списка структуру с тремя полями. Похоже, что мы должны добавить в макет еще три выражения:

```
(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) ...]
```

```
[(cons? an-low)
 (... (first an-low) ...
 ... ... (work-employee (first an-low)) ...
 ... ... (work-rate (first an-low)) ...
 ... ... (work-hours (first an-low)) ...
 (wage*.v2 (rest an-low)) ...)])
```

В этом шаблоне перечислены все данные, которые потенциально могут нас заинтересовать.

Здесь мы используем другую стратегию. В частности, мы должны **создать и использовать отдельный макет функции** всякий раз, когда разрабатываем макет для определения данных, который ссылается на другие определения данных:

```
(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) ...]
    [(cons? an-low)
      (... (for-work (first an-low))
           ... (wage*.v2 (rest an-low)) ...))]

    ; Work -> ???
    ; макет для обработки экземпляров Work
  (define (for-work w)
    (... (work-employee w) ...
         ... (work-rate w) ...
         ... (work-hours w) ...)))
```

Разделение на отдельные макеты приводит к естественному разделению работы на функции и между функциями; ни одна из них не становится слишком большой, и все они относятся к конкретному определению данных.

Наконец, мы готовы приступить к программированию. Как всегда, начнем с самого простого случая, которым здесь является первая строка. Если `wage*.v2` применяется к пустому списку `'()`, она должна вернуть `'()`. Затем переходим ко второй строке и вспоминаем, что вычисляют выражения в ней.

1. `(first an-low)` извлекает первую структуру `Work` из списка.
2. `(for-work ...)` говорит о том, что мы должны спроектировать функцию, которая обрабатывает структуры `Work`.
3. `(rest an-low)` извлекает оставшуюся часть данного списка.
4. `(wage*.v2 (rest an-low))` возвращает список с заработными платами для всех записей `Work`, кроме первой, в соответствии с описанием назначения функции.

Если вы застопорились здесь, воспользуйтесь табличным методом в табл. 12.

Если вам все понятно, то вы увидите, что достаточно сложить два выражения вместе:

```
... (cons (for-work (first an-low))
          (wage*.v2 (rest an-low))) ...
```

предположив, что `fog-work` вычисляет заработную плату для первой записи `Work`. Так мы завершили функцию, добавив еще одну функцию в список желаний.

Поскольку `fog-work` – это просто имя, выбранное навскидку, и не совсем подходит для этой функции, выберем другое имя, `wage.v2`, и добавим ее определение в списке желаний:

```
| ; Work -> Number
| ; вычисляет заработную плату для данной записи w типа work
(define (wage.v2 w)
  0)
```

Проектирование подобных функций подробно описано в части I, поэтому мы не будем приводить здесь дополнительных пояснений. В листинге 31 представлен окончательный результат разработки функций `wage` и `wage*.v2`.

Упражнение 166. Функция `wage*.v2` принимает список записей `Work` и создает список чисел. Однако функции могут также создавать списки структур.

Спроектируйте представление данных, описывающее зарплатный чек. Предположим, что зарплатный чек содержит имя сотрудника и сумму. Затем спроектируйте функцию `wage*.v3`. Она должна принимать список записей `Work` и возвращать список зарплатных чеков, по одному для каждой записи в исходном списке.

Листинг 31. Вычисление зарплат для списка записей `Work`

```
| ; Low -> List-of-numbers
| ; вычисляет недельную зарплату по заданному списку записей Work

(check-expect
  (wage*.v2 (cons (make-work "Robby" 11.95 39) '()))
  (cons (* 11.95 39) '()))

(define (wage*.v2 an-low)
  (cond
    [(empty? an-low) '()]
    [(cons? an-low) (cons (wage.v2 (first an-low))
                          (wage*.v2 (rest an-low)))]))

; Work -> Number
; вычисляет заработную плату для данной записи w типа work
(define (wage.v2 w)
  (* (work-rate w) (work-hours w)))
```

Настоящий зарплатный чек содержит также табельный номер сотрудника. Разработайте представление данных для информации о сотрудниках и измените определение данных для записей `Work`, чтобы в них использовалась эта дополнительная информация о сотруднике, а не только строка с его именем. Также измените представление данных для зарплатного чека, добавив в него имя и табельный номер сотрудника. Наконец, спроектируйте функцию `wage*.v4`, которая

принимает список обновленных записей Work и возвращает список зарплатных чеков.

Замечание об итеративном уточнении. Это упражнение демонстрирует прием *итеративного уточнения задачи*. Мы начали с упрощенного представления зарплаты и постепенно сделали его более реалистичным. Для такой простой программы, как эта, итеративное уточнение – избыточный прием, потому что можно сразу начать с полного представления данных, но позже мы столкнемся с ситуациями, когда итеративное уточнение превращается в необходимость. ■

Упражнение 167. Спроектируйте функцию `sum`, которая принимает список структур `Posn` и возвращает сумму координат X всех его элементов. ■

Упражнение 168. Спроектируйте функцию `translate`, которая принимает и создает списки со структурами `Posn`. Для каждой структуры (`make-posn x y`) во входном списке она должна создавать структуру (`make-posn x (+ y 1)`) в выходном списке. Мы заимствовали слово «*translate*» (перемещение) из геометрии, где сдвиг точки на постоянное расстояние по прямой называется *перемещением*. ■

Упражнение 169. Спроектируйте функцию `legal`. Так же как `translate` из упражнения 168, функция `legal` должна принимать и возвращать списки со структурами `Posn`. Выходной список должен содержать все экземпляры `Posn` из входного списка, чьи координаты X находятся в диапазоне от 0 до 100, а координаты Y – в диапазоне от 0 до 200. ■

Упражнение 170. Вот один из способов представления номера телефона:

```
(define-struct phone [area switch four])
; Phone -- это структура:
;   (make-phone Three Three Four)
; Three -- это число в диапазоне от 100 до 999.
; Four -- это число в диапазоне от 1000 до 9999.
```

Спроектируйте функцию `replace`. Она должна принимать и возвращать списки структур `Phone`, замещая все вхождения кода города (`area`) 713 на 281. ■

10.3. Списки в списках, файлы

В главе 2 мы познакомились с функцией `read-file`, которая читает указанный текстовый файл и возвращает его содержимое в виде строки. Судя по всему, создатель `read-file` выбрал на роль представления текстовых файлов простую строку, и эта функция создает представление данных для конкретного файла (заданного именем). Однако текстовые файлы – это не просто длинные последовательности символов. Они могут быть организованы в строки и слова, строки и ячейки и множеством других

Добавьте (`require 2htdp/batch-io`) в область определений.

способов. Проще говоря, представление содержимого файла в виде простой строки может быть пригодно в редких случаях, но часто это плохой выбор.

Чтобы не быть голословными, рассмотрим пример файла в листинге 32. Он содержит стихотворение Пита Хайна (Piet Hein) и состоит из множества строк и слов. Если прочитать этот файл с помощью выражения

```
| (read-file "ttt.txt")
```

то вы получите такую строку:

Как вы наверняка догадались, многоточия не являются частью результата.

```
| "TTT\n \nPut up in a place\nwhere ...."
```

где "\n" внутри строки соответствует разрывам строк.

Листинг 32. Стихотворение «Things take time»

```
ttt.txt
TTT

Put up in a place
where it's easy to see
the cryptic admonishment
T.T.T.

When you feel how depressingly
slowly you climb,
it's well to remember that
Things Take Time.

Piet Hein
```

Эту строку можно разбить на части, применив ряд элементарных операций со строками, например `explode`, но большинство языков программирования, включая BSL, поддерживают множество других представлений файлов и функций, которые создают такие представления из существующих файлов:

- один из способов представления таких файлов – список строк, где каждая строка в файле представлена одним строковым элементом в списке:

```
(cons "TTT"
  (cons ""
    (cons "Put up in a place"
      (cons ...
        '()))))
```

Здесь вторым элементом списка является пустая строка, потому что файл содержит пустую строку;

- другой способ – список слов, в котором каждое слово представлено строкой:

```
(cons "TTT"
  (cons "Put"
    (cons "up"
      (cons "in"
        (cons ...
          '())))))
```

Обратите внимание, что в этом представлении исчезла вторая пустая строка, потому что в пустой строке нет слов;

- и третье представление основано на списках списков слов:

```
(cons (cons "TTT" '())
  (cons '()
    (cons (cons "Put"
      (cons "up"
        (cons ... '())))
      (cons ...
        '())))))
```

Это представление имеет преимущество перед вторым в том, что оно сохраняет организацию файла, включая вторую пустую строку. Но проблема в том, что в таком представлении списки содержат... списки.

При первом знакомстве идея списков, содержащих списки, может показаться пугающей, но не нужно беспокоиться. Рецепт проектирования поможет справиться с этими сложностями.

Прежде чем продолжить, рассмотрите листинг 33. В нем показано несколько примеров функций чтения файлов. Это не исчерпывающий набор примеров: существует множество других способов чтения текстовых файлов, и вам потребуется узнать еще много нового, чтобы научиться обрабатывать все возможные виды текстовых файлов. Но для нашей цели – обучения и изучения принципов систематического проектирования программ – их вполне достаточно, и они помогут вам разрабатывать достаточно интересные программы.

В листинге 33 используются имена двух определений данных, с которыми мы пока незнакомы, включая списки списков. Как всегда, начнем с определения данных, но на этот раз мы предлагаем вам сделать это самостоятельно. Поэтому, прежде чем продолжить чтение, выполните следующие упражнения. Это необходимо, чтобы понять смысл функций в листинге, а кроме того, не решив эти упражнения, вам будет трудно понять остальную часть данного раздела.

Упражнение 171. Вы уже знаете, как выглядит определение данных для списка строк List-of-strings. Расскажите о нем вслух. Проверьте себя – сможете ли вы представить стихотворение Пита Хайна (Piet Hein) в виде такого списка, в котором каждая строка содержит одну текстовую строку из стихотворения, а также в виде списка, в котором каждая строка содержит одно слово. Используйте функции `read-lines` и `read-words`, чтобы подтвердить свои рассуждения.

Затем разработайте определение данных для списка списков строк *List-of-list-of-strings* и снова представьте стихотворение Пита Хайна в виде такого списка, в котором каждая текстовая строка из стихотворения представлена списком строк, по одной на слово, а все стихотворение – списком таких списков. Используйте функцию `read-words/line`, чтобы подтвердить свои рассуждения. ■

Листинг 33. Функции чтения файлов

```
; String -> String
; возвращает содержимое файла f в виде строки
(define (read-file f) ...)

; String -> List-of-string
; возвращает содержимое файла f в виде списка строк,
; по одной на каждую строку текста
(define (read-lines f) ...)

; String -> List-of-string
; возвращает содержимое файла f в виде списка строк,
; по одной на каждое слово
(define (read-words f) ...)

; String -> List-of-list-of-string
; возвращает содержимое файла f в виде списка списков строк,
; по одному списку на каждую строку текста и по одной строке на слово
(define (read-words/line f) ...)

; Функции принимают имя файла в виде строки.
; Если файл с указанным именем отсутствует в папке,
; где находится программа, то функции сигнализируют об ошибке.
```

Как вы, наверное, знаете, в операционных системах имеются программы для измерения файлов. Одни подсчитывают количество строк, другие определяют, сколько слов присутствует в строке. Начнем с последней программы, чтобы на ее примере посмотреть, как рецепт проектирования помогает справиться с созданием сложных функций.

Первый шаг – убедиться в наличии всех необходимых определений данных. Если вы выполнили упражнение 171, как мы просили, то у вас должно быть определение данных для представления входной информации в проектируемой функции, а в предыдущем разделе определяется список чисел *List-of-numbers*, описывающий все возможные выходные результаты. Для краткости обозначим класс списков списков строк как *LLS* (*List-of-list-of-string* – список списков строк) и используем это имя для оформления заголовка для желающей функции:

```
; LLS -> List-of-numbers
; подсчитывает количество слов в каждой текстовой строке
(define (words-on-line lls) '())
```

Мы дали функции имя `words-on-line`, потому что оно выражает ее назначение в одной фразе.

Далее необходимо создать набор примеров данных:

```
(define line0 (cons "hello" (cons "world" '())))
(define line1 '())

(define lls0 '())
(define lls1 (cons line0 (cons line1 '())))
```

Первые два определения представляют два примера текстовых строк: первая содержит два слова, а вторая не содержит ни одного. Последние два определения показывают, как создать экземпляры LLS из этих примеров строк. Определите ожидаемый результат применения функции к этим двум примерам.

Имея примеры данных, легко сформулировать примеры применения функции; просто представьте, как применить функцию к каждому из примеров данных. Когда `words-on-line` применяется к `lls0`, она должна вернуть пустой список, потому что в нем нет строк. Когда `words-on-line` применяется к `lls1`, она должна вернуть список с двумя числами, потому что `lls1` содержит две строки. Эти два числа равны 2 и 0 соответственно, потому что первая строка в `lls1` содержит два слова, а вторая – ни одного.

Бот как можно перевести все это в тестовые примеры:

```
(check-expect (words-on-line lls0) '())
(check-expect (words-on-line lls1)
              (cons 2 (cons 0 '())))
```

Выполнив этот второй шаг, вы получите законченную программу, которая, впрочем, не дает никаких результатов в некоторых тестовых примерах.

Следующий интересный шаг – разработка макета для этой демонстрационной задачи. Ответив на вопросы из табл. 10, вы без труда получите типичный макет функции, обрабатывающей список:

```
(define (words-on-line lls)
  (cond
    [(empty? lls) ...]
    [else
      (... (first lls) ; список строк
           ... (words-on-line (rest lls)) ...)]))
```

Как и в предыдущем разделе, мы знаем, что выражение `(first lls)` извлекает список строк List-of-strings, который тоже имеет сложную организацию. Возникает соблазн вставить вложенный макет для выражения этих знаний, но, как вы наверняка помните, лучше разработать второй вспомогательный макет и изменить первую строку во втором условии так, чтобы она ссылалась на этот вспомогательный макет.

Поскольку вспомогательный макет определяет функцию, которая принимает список, он почти не отличается от предыдущего:

```
(define (line-processor ln)
  (cond
```

```

| ((empty? lls) ...)
| [else
| (... (first ln) ; a string
| ... (line-processor (rest ln)) ...)])

```

Основное отличие заключается в том, что (*first ln*) извлекает строку из списка, а мы рассматриваем строки как атомарные значения. Получив этот макет, можно изменить первую строку во втором условии в *words-on-line*, как показано ниже:

```
| ... (line-processor (first lls)) ...
```

Это напомнит нам на пятом шаге о том, что определение *words-on-line* требует спроектировать вспомогательную функцию.

Теперь приступим к программированию. Как всегда, воспользуемся вопросами из табл. 11. Первое условие, обрабатывающее случай, когда функция получает пустой список строк, – самый простой. Наши примеры говорят нам, что в этом случае функция должна вернуть '(), то есть пустой список чисел. Второе условие, обрабатывающее случай, когда функция получает непустой список, содержит несколько выражений. Давайте вспомним, что они вычисляют:

- (*first lls*) извлекает первую строку из непустого списка строк;
- (*line-processor (first lls)*) напоминает, что мы должны спроектировать вспомогательную функцию для обработки строки;
- (*rest lls*) – оставшаяся часть списка строк;
- (*words-on-line (rest lls)*) подсчитывает слова в строках в оставшейся части списка. Откуда мы это знаем? Именно это указано в сигнатуре и в описании назначения функции *words-on-line*.

Если допустить, что у нас уже есть вспомогательная функция, которая принимает строку и подсчитывает слова в ней (назовем ее *words#*), то мы с легкостью можем закончить второе условие:

```

| (cons (words# (first lls))
|       (words-on-line (rest lls)))

```

Это выражение объединяет количество слов в первой строке *lls* со списком чисел слов, соответствующих строкам в оставшейся части списка *lls*.

Осталось спроектировать функцию *words#*. В макете мы обозначили ее именем *line-processor*. Ее цель – подсчитать количество слов в текстовой строке, которая представлена простым списком строк. Итак, вот запись в нашем списке желаний:

```

| ; List-of-strings -> Number
| ; подсчитывает слова в los
| (define (words# los) 0)

```

Здесь уместно вспомнить пример, использованный в главе 9 для иллюстрации рецепта проектирования определений данных, ссы-

лающихся на самих себя. Функция в том примере называется `how-many`, и она тоже подсчитывает количество строк в списке строк. Ее отличие в том, что она обрабатывает список имен, но в данном случае это не имеет никакого значения; она подсчитывает количество строк в списке строк, а значит, решает нашу проблему.

Повторное использование функций считается хорошей практикой, поэтому мы можем определить `words#` так:

```
| (define (words# los)
|   (how-many los))
```

На самом деле в языках программирования уже есть функции, которые решают такие задачи. В BSL эта функция называется `length`, она подсчитывает количество значений в любом списке, независимо от типов этих значений.

В листинге 34 представлено законченное решение нашей текущей задачи. Листинг включает два тестовых примера. Кроме того, вместо отдельной функции `words#` определение `words-on-line` вызывает функцию `length`, которая имеется в языке BSL. Поэкспериментируйте с этим решением в DrRacket и убедитесь, что два тестовых примера охватывают все определение функции.

Листинг 34. Подсчет слов в текстовой строке

```
; LLS -- одно из значений:
; -- '()
; -- (cons Los LLS)
; интерпретация: список текстовых строк, каждая из которых
; представлена списком строк

(define line0 (cons "hello" (cons "world" '())))
(define line1 '())

(define lls0 '())
(define lls1 (cons line0 (cons line1 '())))

; LLS -> List-of-numbers
; подсчитывает количество слов в каждой текстовой строке

(check-expect (words-on-line lls0) '())
(check-expect (words-on-line lls1) (cons 2 (cons 0 '())))

(define (words-on-line lls)
  (cond
    [(empty? lls) '()]
    [else (cons (length (first lls))
                (words-on-line (rest lls)))]))
```

Сделав еще один небольшой шаг, вы получили возможность создать свою первую файловую утилиту:

```
; String -> List-of-numbers
; подсчитывает слова во всех строках в заданном файле
```

Возможно, у вас появится желание познакомиться со списком функций, доступных в BSL.

Назначение некоторых из них может показаться туманным, но многие из них пригодятся для решения будущих задач. Использование таких функций экономит ваше время, а не наше.

```
(define (file-statistic file-name)
  (words-on-line
    (read-words/line file-name)))
```

Она просто объединяет библиотечную функцию с `words-on-line`. Первая читает файл и возвращает его содержимое в виде списка списков строк, который затем передается второй функции.

Идея объединения встроенной функции со вновь спроектированной широко используется в программировании. Естественно, люди проектируют функции не случайным образом и стараются использовать функции, уже имеющиеся в выбранном языке программирования. Разработчики программ тщательно планируют свои действия и проектируют функции в соответствии с данными, которые возвращают имеющиеся функции. Как уже отмечалось выше, решение обычно воспринимается как композиция двух вычислений и сводится к разработке соответствующего набора данных, с помощью которого можно передать результат одного вычисления второму, где каждое вычисление реализуется с помощью функции.

Упражнение 172. Спроектируйте функцию `collapse`, которая преобразует список текстовых строк в строку. При объединении слова в текстовых строках следует разделять пробелами (" "), а сами текстовые строки – символом перевода строки ("\n").

Листинг 35. Кодирование строк

```
; 1String -> String
; преобразует данный символ (1String) в трехбуквенную числовую строку

(check-expect (encode-letter "z") (code1 "z"))
(check-expect (encode-letter "\t")
  (string-append "00" (code1 "\t")))
(check-expect (encode-letter "a")
  (string-append "0" (code1 "a"))))
(define (encode-letter s)
  (cond
    [(>= (string->int s) 100) (code1 s)]
    [(< (string->int s) 10)
     (string-append "00" (code1 s))]
    [(< (string->int s) 100)
     (string-append "0" (code1 s)))))

; 1String -> String
; преобразует данный символ (1String) в строку

(check-expect (code1 "z") "122")

(define (code1 c)
  (number->string (string->int c)))
```

Усложненное задание. Когда вы закончите, опробуйте программу, как показано ниже:

```
(write-file "ttt.dat"
  (collapse (read-words/line "ttt.txt")))
```

и убедитесь в идентичности файлов «ttt.dat» и «ttt.txt», удалив все лишние пробелы в вашей версии стихотворения Т.Т.Т. ■

Упражнение 173. Спроектируйте программу, удаляющую все артикли из текстового файла. Программа должна принимать имя файла *n*, читать файл, удалять артикли и записывать результат в файл с именем, которое является результатом объединения строки «no-article-» с именем файла *n*. В этом упражнении под артиклями понимаются следующие три слова: «*a*», «*an*» и «*the*».

Используйте `read-words/line`, чтобы сохранить организацию исходного текста при преобразовании в строки и слова. Завершив разработку программы, примените ее к стихотворению Пита Хейна. ■

Упражнение 174. Спроектируйте программу, которая кодирует текстовые файлы числами. Каждая буква в слове должна кодироваться числовой трехбуквенной строкой со значением от 0 до 256. В листинге 35 показана наша версия функции кодирования букв. Прежде чем начать, объясните, как работает эта функция.

Подсказки. (1). Используйте `read-words/line`, чтобы сохранить организацию исходного текста при преобразовании в строки и слова. (2) Прочтайте описание функции `explode`. ■

Упражнение 175. Спроектируйте программу, имитирующую команду `wc` в Unix. Эта команда подсчитывает символы, слова и строки в данном файле. То есть она принимает имя файла и выводит три числа. ■

Листинг 36. Транспонирование матрицы

```
; Matrix -> Matrix
; транспонирует заданную матрицу

(define wor1 (cons 11 (cons 21 '())))
(define wor2 (cons 12 (cons 22 '())))
(define tam1 (cons wor1 (cons wor2 '())))

(check-expect (transpose mat1) tam1)

(define (transpose lln)
  (cond
    [(empty? (first lln)) '()]
    [else (cons (first* lln) (transpose (rest* lln))))]))
```

Упражнение 176. Возможно, учителя математики уже познакомили вас с матричными вычислениями. Матрица – это прямоугольная таблица чисел. Вот одно из возможных представлений данных для матриц:

```
; Matrix -- это одно из значений:
; -- (cons Row '())
; -- (cons Row Matrix)
; ограничение: все строки в матрице имеют одинаковую длину

; Row (строка в матрице) -- это одно из значений:
; -- '()
; -- (cons Number Row)
```

Обратите внимание на ограничение. Изучите определение данных и преобразуйте матрицу два на два, состоящую из чисел 11, 12, 21 и 22, в это представление данных. Стоп! Не продолжайте чтения, пока не разберетесь с примерами данных.

Вот решение этой простой задачи:

```
(define row1 (cons 11 (cons 12 '())))
(define row2 (cons 21 (cons 22 '())))
(define mat1 (cons row1 (cons row2 '())))
```

Если к этому решению вы пришли не сами, то внимательно изучите его.

Функция в листинге 36 реализует важную математическую операцию – транспонирование матрицы. Транспонировать означает отразить элементы относительно диагонали, то есть относительно линии, соединяющей верхний левый угол с правым нижним.

Стоп! Транспонируйте матрицу `mat1` вручную, затем прочитайте листинг 36. Что делает выражение (`empty? (first lln)`) в функции `transpose?`

Определение предполагает использование двух вспомогательных функций:

- `first*`, принимает матрицу и возвращает первый столбец в виде списка чисел;
- `rest*`, принимает матрицу и возвращает матрицу без первого столбца.

Даже не имея определений этих функций, вы должны понимать, как работает транспонирование. Вы также должны понимать, что эту функцию **нельзя** спроектировать с помощью рецептов проектирования, которые мы использовали до сих пор. Объясните почему.

Спроектируйте две функции из списка желаний. Затем завершите проектирование `transpose`, добавив несколько тестовых примеров. ■

10.4. И снова о графическом редакторе

В разделе 5.10 мы спроектировали интерактивный графический однострочный редактор. Там предлагалось два разных способа представления состояния редактора и настоятельно рекомендовалось изучить оба: структуру, содержащую пару строк, и структуру, объединяющую строку с индексом, соответствующим текущей позиции курсора (см. упражнение 87).

Третья альтернатива – использовать структуры, объединяющие два списка символов (`1String`):

```
(define-struct editor [pre post])
; Editor -- это структура:
;   (make-editor Lo1S Lo1S)
```

```
| ; Lo1S -- это одно из значений:
| ; -- '()
| ; -- (cons 1String Lo1S)
```

Прежде чем у вас появятся вопросы, рассмотрим два примера данных:

```
(define good
  (cons "g" (cons "o" (cons "o" (cons "d" '())))))
(define all
  (cons "a" (cons "l" (cons "l" '()))))
(define lla
  (cons "l" (cons "l" (cons "a" '()))))

; пример данных 1:
(make-editor all good)

; пример данных 2:
(make-editor lla good)
```

Эти два примера демонстрируют, насколько важно указывать в заголовке интерпретацию. Два поля в структуре Editor четко представляют буквы слева и справа от курсора, однако примеры выше демонстрируют, что существует как минимум два способа интерпретации структуры:

- 1) (`(make-editor pre post)`) может означать, что буквы в `pre` предшествуют курсору, а буквы в `post` следуют за ним и что текст отображается в редакторе как

```
| (string-append (implode pre) (implode post))
```

Напомним, что `implode` преобразует список символов (1String) в строку;

- 2) (`(make-editor pre post)`) также может означать, что буквы в `pre` предшествуют курсору в обратном порядке. Если это так, то текст в редакторе отображается следующим образом:

```
| (string-append (implode (rev pre))
|               (implode post))
```

Функция `rev` должна принимать список символов и переставлять их в обратном порядке.

Даже не имея полного определения функции `rev`, нетрудно представить, как она работает. Используйте это знание, чтобы убедиться, что понимаете, что преобразование первого примера данных в информацию в соответствии с первой интерпретацией и преобразование второго примера данных в соответствии со второй интерпретацией приводят к одному и тому же отображению в редакторе:

allgood

Обе интерпретации имеют право на жизнь, но, как оказывается, вторая значительно упрощает проект программы. Остальная часть этого раздела иллюстрирует этот момент, параллельно демонстрируя использование списков внутри структур. Чтобы оценить данный урок по достоинству, вы должны были выполнить упражнения из раздела 5.10.

Начнем с `rev`, потому что нам определенно нужна эта функция, чтобы понять смысл определения данных. Она имеет простой заголовок:

```
| ; Lo1s -> Lo1s
| ; возвращает копию входного списка, в которой элементы следуют в обратном порядке
|
| (check-expect
|   (rev (cons "a" (cons "b" (cons "c" '()))))
|   (cons "c" (cons "b" (cons "a" '()))))
|
| (define (rev l) l)
```

Для удобства мы добавили один «очевидный» пример. Вы можете добавить свои дополнительные примеры, чтобы убедиться, что понимаете, что вам нужно.

Макет функции `rev` – типичный для функций, обрабатывающих списки:

```
| (define (rev l)
|   (cond
|     [(empty? l) ...]
|     [else (... (first l) ...
|                 ... (rev (rest l)) ...))])
```

Здесь мы имеем два предложения, и во втором из них используется несколько селекторов и ссылок на само определение.

Заполнить первое предложение в макете легко: копией пустого списка с переупорядоченными элементами является сам пустой список. Для заполнения второго предложения снова используем вопросы:

- (`first l`) – это первый элемент списка символов;
- (`rest l`) – это остальная часть списка;
- (`rev (rest l)`) – это копия остальной части списка с переупорядоченными элементами.

Стоп! Попробуйте завершить проектирование `rev`, используя следующие подсказки.

Таблица 19. Табличный метод для rev

<code>l</code>	<code>(first l)</code>	<code>(rest l)</code>	<code>(rev (rest l))</code>	<code>(rev l)</code>
<code>(cons "a" '())</code>	"a"	'()	'()	<code>(cons "a" '())</code>
<code>(cons "a" (cons "b" (cons "c" '())))</code>	"a"	<code>(cons "b" (cons "c" '()))</code>	<code>(cons "c" (cons "b" '()))</code>	<code>(cons "c" (cons "b" (cons "a" '())))</code>

Если этих подсказок окажется недостаточно, то создайте таблицу из примеров. В табл. 19 показаны два примера: (`(cons "a" '())`) и (`(cons "a" (cons "b" (cons "c" '())))`). Второй пример особенно показателен. Взглядите на предпоследний столбец, он показывает, что основную работу выполняет применение (`(rev (rest l))`), возвращающее (`(cons "c" (cons "b" (cons "a" '())))`). Так как желаемым результатом является (`(cons "c" (cons "b" (cons "a" '())))`), то `rev` должна каким-то образом добавить "a" в конец результата, полученного рекурсивным вызовом. На самом деле, поскольку (`(rev (rest l))`) возвращает уже преобразованную копию оставшейся части списка, то достаточно будет просто добавить результат выражения (`(first l)`) в его конец. У нас нет функции, которая добавляет элементы в конец списка, но мы можем добавить ее в список желаний и завершить определение функции:

```
| (define (rev l)
|   (cond
|     [(empty? l) '()]
|     [else (add-at-end (rev (rest l)) (first l))]))
```

Вот расширенная запись в списке желаний для функции `add-at-end`:

```
| ; Lo1s 1String -> Lo1s
| ; создает новый список, добавляя s в конец l
|
| (check-expect
|   (add-at-end (cons "c" (cons "b" '())) "a")
|   (cons "c" (cons "b" (cons "a" '()))))
|
| (define (add-at-end l s)
|   l)
```

Мы назвали запись «расширенной», потому что она содержит пример, сформулированный в виде теста. Пример заимствован из примеров для `rev`, и именно он определяет необходимость добавления новой записи в список желаний. Прежде чем продолжить чтение, придумайте пример, когда `add-at-end` получает пустой список.

Поскольку `add-at-end` тоже является функцией, обрабатывающей списки, ее макет покажется вам хорошо знакомым:

```
| (define (add-at-end l s)
|   (cond
|     [(empty? l) ...]
|     [else (... (first l) ...
|                ... (add-at-end (rest l) s) ...)]))
```

Чтобы превратить его в определение функции, вновь воспользуемся вопросами для шага 5 рецепта. Наш первый вопрос: как сформулировать ответ для «базового» случая, то есть первого предложения в этом макете? Если вы неукоснительно выполняли все предлагаемые упражнения, то знаете, что его результатом всегда будет выражение

```
| (add-at-end '() s)
```

то есть (`(cons s '())`). В конце концов, результат должен быть списком, а список должен содержать данный символ (`1String`).

Следующие два вопроса касаются «сложного» случая со ссылкой на само определение. Мы знаем, во что вычисляются выражения во втором предложении в `cond`: первое извлекает первый символ из данного списка, а второе «создает новый список, добавляя `s` в конец (`(rest l)`)». Описание назначения четко определяет, что должна сделать функция в этом случае – она должна прибавить (`(first l)`) к результату рекурсивного вызова:

```
(define (add-at-end l s)
  (cond
    [(empty? l) (cons s '())]
    [else
      (cons (first l) (add-at-end (rest l) s))]))
```

Запустите тестовые примеры, чтобы убедиться, что эта функция работает правильно и, следовательно, правильно работает `rev`. Вас не должно удивлять, если вы узнаете, что в BSL уже имеется функция, которая переупорядочивает любые списки, включая списки символов. Она называется `reverse`.

Упражнение 177. Спроектируйте функцию `create-editor`. Она должна принимать две строки и создавать структуру `Editor`. Первая строка – это текст слева от курсора, а вторая – справа. В оставшейся части раздела мы не раз будем использовать эту функцию. ■

На данном этапе у вас должно сложиться ясное понимание нашего представления данных для графического однострочного редактора. Следуя стратегии разработки интерактивных программ, представленной в разделе 3.6, мы должны определить физические константы, например ширину и высоту редактора, и графические константы, например изображение курсора. Вот как эти константы определили мы:

```
(define HEIGHT 20) ; the height of the editor
(define WIDTH 200) ; its width
(define FONT-SIZE 16) ; the font size
(define FONT-COLOR "black") ; the font color

(define MT (empty-scene WIDTH HEIGHT))
(define CURSOR (rectangle 1 HEIGHT "solid" "red"))
```

Также важно не забыть добавить в список желаний обработчики событий и функцию, которая отображает состояние редактора. Напомним, что заголовки этих функций определяет библиотека `2htdp/universe`:

```
; Editor -> Image
; создает изображение редактора с двумя фрагментами текста,
; разделенными курсором
(define (editor-render e) MT)

; Editor KeyEvent -> Editor
```

```
| ; обрабатывает события клавиатуры
| (define (editor-kh ed ke) ed) (index "editor-kh")
```

Кроме того, стратегия из раздела 3.6 требует добавить главную функцию для программы:

```
| ; main : String -> Editor
| ; запускает программу редактора с некоторой начальной строкой
| (define (main s)
|   (big-bang (create-editor s ""))
|     [on-key editor-kh]
|     [to-draw editor-render]))
```

Вернитесь к упражнению 177 и спроектируйте функцию создания редактора, если вы этого еще не сделали.

В общем случае не важно, в каком порядке проектировать функции из списка желаний, но мы решили сначала заняться функцией `editor-kh`, а затем перейти к `editor-render`. Поскольку у нас уже есть заголовок, давайте объясним работу обработчика событий клавиатуры на двух примерах:

```
| (check-expect (editor-kh (create-editor "" "") "e")
|               (create-editor "e" ""))
| (check-expect
|   (editor-kh (create-editor "cd" "fgh") "e")
|   (create-editor "cde" "fgh"))
```

Оба примера показывают, что происходит при нажатии клавиши «`e`» на клавиатуре. Компьютер вызывает функцию `editor-kh` с текущим состоянием редактора и символом «`e`». В первом примере редактор не содержит текста, и в результате получается редактор с буквой «`e`», за которой следует курсор. Во втором примере курсор находится между строками «`cd`» и «`fgh`», поэтому в результате получается редактор с курсором между «`cde`» и «`fgh`». Проще говоря, функция всегда вставляет обычные буквы в позицию курсора.

Прежде чем продолжить, составьте примеры, иллюстрирующие работу `editor-kh`, когда на клавиатуре нажимается клавиша забоя **Backspace** («`\b`»), чтобы удалить какую-то букву, или клавиши со стрелками «влево» и «вправо» для перемещения курсора. Во всех случаях подумайте, что должно произойти, когда редактор не содержит текста, когда курсор находится на левом или правом конце непустой строки и когда он находится посередине. Даже притом что здесь не используются интервалы значений, обязательно создайте примеры для «краевых» случаев.

После подготовки тестовых примеров можно переходить к макету функции. Как мы знаем, `editor-kh` принимает два экземпляра составных данных: первый – это структура, содержащая списки, второй – длинное перечисление строк. Вообще говоря, для проектирования подобных функций необходим усовершенствованный рецепт проектирования, но в подобных ситуациях вполне очевидно, что первоочередное значение имеет событие нажатия клавиши.

С учетом вышесказанного макет функции конструируется как большое условное выражение `cond`, проверяющее событие клавиатуры `KeyEvent`, полученное функцией:

```
(define (editor-kh ed k)
  (cond
    [(key=? k "left") ...]
    [(key=? k "right") ...]
    [(key=? k "\b") ...]
    [(key=? k "\t") ...]
    [(key=? k "\r") ...]
    [(= (string-length k) 1) ...]
    [else ...]))
```

Выражение `cond` не совсем соответствует определению данных для событий клавиатуры `KeyEvent`, потому что некоторые события требуют особого внимания ("`left`", "`\b`" и т. д.). Некоторые события должны игнорироваться, потому что для них в редакторе не предусмотрено никаких действий ("`\t`" и "`\r`"), а некоторые можно объединить в одну большую группу (обычные алфавитно-цифровые клавиши).

Упражнение 178. Объясните, почему в макете `editor-kh` события "`\t`" и "`\r`" проверяются раньше событий нажатия обычных алфавитно-цифровых клавиш (строк с длиной 1). ■

На пятом шаге – определении функции – мы должны рассмотреть каждое предложение в условном выражении. Первое предложение должно только перемещать курсор и оставлять текстовое содержимое редактора неизменным. То же относится и ко второму предложению. Третье предложение, однако, должно удалить букву из содержимого редактора, если это возможно. Наконец, шестое предложение должно добавить букву в позицию курсора. Следуя рецепту проектирования, добавим в список желаний функции для каждой из этих задач:

```
(define (editor-kh ed k)
  (cond
    [(key=? k "left") (editor-lft ed)]
    [(key=? k "right") (editor-rgt ed)]
    [(key=? k "\b") (editor-del ed)]
    [(key=? k "\t") ed]
    [(key=? k "\r") ed]
    [(= (string-length k) 1) (editor-ins ed k)]
    [else ed]))
```

Судя по определению `editor-kh`, третья и четвертая функции в списке желаний имеют одинаковые сигнатуры:

```
| ; Editor -> Editor
```

А последняя принимает два аргумента:

```
| ; Editor 1String -> Editor
```

Мы оставляем вам самостоятельно сформулировать записи в списке желаний для первых трех функций и сосредоточимся на четвертой.

Начнем с описания назначения и заголовка функции:

```
| ; вставляет символ k между pre и post
| (define (editor-ins ed k)
|   ed)
```

Назначение прямо вытекает из постановки задачи. Поскольку *pre* и *post* являются составными частями текущего состояния редактора, мы просто объединили их в один аргумент.

Далее создадим примеры для *editor-ins*, опираясь на примеры для *editor-kh*:

```
| (check-expect
|   (editor-ins (make-editor '() '()) "e")
|   (make-editor (cons "e" '()) '()))
|
| (check-expect
|   (editor-ins
|     (make-editor (cons "d" '())
|                   (cons "f" (cons "g" '()))))
|     "e")
|
|   (make-editor (cons "e" (cons "d" '()))
|                 (cons "f" (cons "g" '()))))
```

Внимательно изучите эти примеры, используя интерпретацию в определении данных *Editor*, и убедитесь, что понимаете, что означают эти примеры с точки зрения информации и что должна получить функция в каждом случае. Лучше всего нарисовать визуальное представление редактора, потому что оно хорошо представляет информацию.

Четвертый шаг требует создания макета. Первый аргумент гарантированно является структурой, а второй – строкой, атомарным фрагментом данных. Иначе говоря, макет должен извлечь фрагменты данных из представления редактора:

```
| (define (editor-ins ed k)
|   (... ed ... k ...)
|   ... (editor-pre ed) ...
|   ... (editor-post ed) ...))
```

Параметры перечислены в макете, потому что они тоже доступны.

На основе макета и примеров относительно легко заключить, что *editor-ins* должна создать новое представление редактора из полей *pre* и *post* в текущем состоянии, добавив *k* перед первым из них:

```
| (define (editor-ins ed k)
|   (make-editor (cons k (editor-pre ed))
|                 (editor-post ed)))
```

Несмотря на то что *(editor-pre ed)* и *(editor-post ed)* являются списками символов, нет никакой необходимости проектировать вспомогательные функции. Чтобы получить желаемый результат, достаточно использовать оператор *cons*, который создает списки.

Теперь вы должны запустить тесты для этой функции и, используя интерпретацию Editor, объяснить общими словами, почему эта функция выполняет вставку. А если для вас что-то останется непонятным, то сравните это простое определение с определением из упражнения 84 и выясните, почему в том определении была нужна вспомогательная функция, а в этом нет.

Упражнение 179.

Спроектируйте следующие функции:

```
; Editor -> Editor
; перемещает курсор на одну позицию влево,
; если возможно
(define (editor-lft ed) ed)

; Editor -> Editor
; перемещает курсор на одну позицию вправо,
; если возможно
(define (editor-rgt ed) ed)

; Editor -> Editor
; удаляет символ слева от курсора,
; если возможно
(define (editor-del ed) ed)
```

И снова не забудьте создать набор хороших примеров. ■

При проектировании функции отображения редактора возникают новые, но небольшие проблемы. Первая – требуется определить достаточно большое количество тестов. С одной стороны, необходимо покрыть все возможные комбинации: пустая строка слева от курсора, пустая строка справа и обе строки пустые. С другой стороны, придется поэкспериментировать с функциями из библиотеки *2htdp/image*. В частности, нужно найти способ объединения двух строк в одно текстовое изображение, а также способ поместить текстовое изображение в пустую сцену (MT). Вот как мы реализовали создание изображения на основе результата, возвращаемого применением (*create-editor "pre" "post"*):

```
(place-image/align
  (beside (text "pre" FONT-SIZE FONT-COLOR)
          CURSOR
          (text "post" FONT-SIZE FONT-COLOR))
  1 1
  "left" "top"
  MT)
```

Если сравнить полученный результат с изображением редактора выше, то можно заметить некоторые различия, и в этом нет ничего необычного, потому что точный вид редактора не важен для цели этого упражнения, а также потому, что внесенные изменения не упрощают задачу. В любом случае, поэкспериментируйте в области взаимодействий DrRacket, чтобы выбрать такой вид экрана редактора, который вам нравится.

Теперь можно переходить к реализации макета, который выглядит так:

```
| (define (editor-render e)
|   (... (editor-pre e) ... (editor-post e)))
```

Аргумент – это обычная структура с двумя полями. Однако значения этих полей являются списками символов, поэтому у вас может возникнуть соблазн уточнить шаблон. Но не надо этого делать! Просто вспомните, что когда одно определение данных ссылается на другое сложное определение данных, лучше использовать список желаний.

Если вы создали достаточное количество примеров, то наверняка включили в свой список желаний функцию, которая превращает строку в текст нужного размера и цвета. Назовем эту функцию editor-text. Определение editor-render может просто дважды применить editor-text и объединить результаты с помощью beside и place-image:

```
; Editor -> Image
(define (editor-render e)
  (place-image/align
    (beside (editor-text (editor-pre e))
            CURSOR
            (editor-text (editor-post e)))
    1 1
    "left" "top"
    MT))
```

Несмотря на то что в этом определении мы получили выражение с тремя уровнями вложенности, использование воображаемой функции editor-text делает его вполне читаемым.

Остается спроектировать editor-text. Из реализации editor-render мы знаем, что editor-text принимает список символов и создает текстовое изображение:

```
; Lo1s -> Image
; преобразует список символов в текстовое изображение
(define (editor-text s)
  (text "" FONT-SIZE FONT-COLOR))
```

Это начальное определение создает пустое текстовое изображение.

Рассмотрим пример, чтобы понять, что должна возвращать функция editor-text. Вот пример исходного состояния редактора:

```
| (create-editor "pre" "post")
```

которое также использовалось для объяснения editor-render, а вот его эквивалент:

```
(make-editor
  (cons "e" (cons "r" (cons "p" '())))
  (cons "p" (cons "o" (cons "s" (cons "t" '())))))
```

Возьмем за основу второй список. Ожидаемый результат нам известен по примеру для `editor-render`:

```
(check-expect
  (editor-text
    (cons "p" (cons "o" (cons "s" (cons "t" '())))))
  (text "post" FONT-SIZE FONT-COLOR))
```

Если хотите, можете сейчас приостановиться и придумать второй пример.

Учитывая, что `editor-text` принимает список символов, создание макета не вызывает особых трудностей:

```
(define (editor-text s)
  (cond
    [(empty? s) ...]
    [else (... (first s)
                ... (editor-text (rest s)) ...))]))
```

В конце концов, структура макета диктуется определениями данных, описывающими параметры функции. Но этот макет нам не понадобится. Как вы наверняка помните, в определении `Editor` используется функция `explode`, которая преобразует строку в список символов. Естественно, имеется противоположная функция `implode`, выполняющая обратное преобразование, то есть:

```
> (implode
  (cons "p" (cons "o" (cons "s" (cons "t" '())))))
"post"
```

Если использовать эту функцию, то определение `editor-text` приобретает тривиально простой вид:

```
(define (editor-text s)
  (text (implode s) FONT-SIZE FONT-COLOR))
```

Упражнение 180. Спроектируйте `editor-text` без использования функции `implode`. ■

Когда дело доходит до тестирования двух функций, нас поджидает настоящий сюрприз: наш тест для `editor-text` выполняется успешно, а тест для `editor-render` терпит неудачу. Исследование ошибки показывает, что строка слева от курсора – "ре" – перевернута. Мы забыли, что эта часть состояния редактора хранится в обратном порядке. К счастью, модульные тесты для двух функций помогли точно определить, в какой функции скрывается ошибка, и даже сообщили нам, что не так с функцией, и предложили способы решения:

```
(define (editor-render ed)
  (place-image/align
    (beside (editor-text (reverse (editor-pre ed)))
            CURSOR
            (editor-text (editor-post ed))))
```

```
1 1  
"left" "top"  
MT))
```

В этом определении к полю `rge` структуры `ed` применяется функция `reverse`.

Примечание. Современные приложения позволяют пользователям перемещать курсор с помощью мыши (или жестов). Мы могли бы добавить эту возможность в наш редактор, но давайте отложим этот шаг до раздела 32.4.

11. Проектирование методом композиции

К настоящему времени вы уже знаете, что программы – это сложные продукты и для их создания требуется спроектировать множество функций, взаимодействующих друг с другом. Такой подход к разработке особенно хорош, когда разработчик знает, как создать несколько функций и как объединить их в одну программу.

Вы уже не раз сталкивались с необходимостью проектирования взаимосвязанных функций. Иногда постановка задачи подразумевает решение нескольких разных подзадач, и каждую такую подзадачу лучше всего реализовать в виде отдельной функции. Иногда определение данных может ссылаться на другое определение, и в этом случае функция, обрабатывающая данные первого типа, полагается на функцию, обрабатывающую данные второго типа.

В этой главе мы представим вашему вниманию несколько примеров проектирования программ, состоящих из множества функций. Для поддержки такого подхода в главе представлены некоторые неформальные рекомендации по разделению функций и их комбинированию. Однако, поскольку эти примеры требуют списков сложной формы, мы начнем эту главу с раздела, посвященного краткой записи списков.

11.1. Функция `list`

Мы закончили осваивать язык BSL, и теперь пришла пора открыть меню *Language > Choose Language* (Язык > Выбрать язык) и выбрать язык *Beginning Student with List Abbreviations* (Начинающий студент со списковыми сокращениями).

Вы наверняка уже устали писать бесчисленное множество операторов `cons` для создания списков, особенно списков с большим количеством значений. К счастью, у нас есть расширенный язык обучения, который также называется BSL+ и поддерживает механизмы, упрощающие эту сторону работы программиста.

Ключевым нововведением в этом языке является функция `list`, которая принимает произвольное количество значений и создает список. Этую функцию можно рассматривать как средство сокращенной записи списков. В частности, каждое выражение в форме

```
| (list exp-1 ... exp-n)
```

фактически является сокращенным представлением выражения с операторами `cons`:

```
| (cons exp-1 (cons ... (cons exp-n '()))))
```

Обратите внимание, что `'()` здесь не является элементом списка, в действительности это «остальная» (*rest*) часть списка. Вот три простых примера:

Краткая форма записи	Длинная форма записи
(list "ABC")	(cons "ABC" '())
(list #false #true)	(cons #false (cons #true '()))
(list 1 2 3)	(cons 1 (cons 2 (cons 3 '()))))

В этих примерах создаются списки с одним, двумя и тремя элементами соответственно.

Разумеется, функцию `list` можно применить не только к значениям, но и к выражениям:

```
| > (list (+ 0 1) (+ 1 1))
| (list 1 2)
| > (list (/ 1 0) (+ 1 1))
| /:division by zero
```

Выражения вычисляются перед созданием списка. Если в процессе вычислений возникнет ошибка, то список не будет создан. Проще говоря, `list` действует точно так же, как любая другая элементарная операция, принимающая произвольное количество аргументов; просто она возвращает список, созданный с помощью `cons`.

Использование `list` значительно упрощает определение списков с большим количеством элементов или содержащих другие списки либо структуры. Например:

```
| (list 0 1 2 3 4 5 6 7 8 9)
```

Этот список содержит 10 элементов, для формирования его в ином случае потребовалось бы 10 раз использовать оператор `cons` и один экземпляр `'()`. Аналогично для создания следующего списка

```
(list (list "bob" 0 "a")
      (list "carl" 1 "a")
      (list "dana" 2 "b")
      (list "erik" 3 "c")
      (list "frank" 4 "a")
      (list "grant" 5 "b")
      (list "hank" 6 "c")
      (list "ian" 7 "a")
      (list "john" 8 "d")
      (list "karel" 9 "e"))
```

потребовалось 11 раз использовать функцию `list` вместо 40 операторов `cons` и 11 дополнительных элементов `'()`.

Упражнение 181. Сконструируйте следующие списки с помощью `list`:

- 1) (cons "a" (cons "b" (cons "c" (cons "d" '()))))
- 2) (cons (cons 1 (cons 2 '())) '())
- 3) (cons "a" (cons (cons 1 '()) (cons #false '()))))
- 4) (cons (cons "a" (cons 2 '())) (cons "hello" '()))

Также попробуйте сконструировать такой список:

```
| (cons (cons 1 (cons 2 '())))
  (cons (cons 2 '())
         '()))
```

Для начала определите количество элементов в каждом списке и во всех вложенных списках. Выразите свои ответы с помощью `check-expect`; это поможет вам убедиться в эквивалентности сокращенной и полной форм записи. ■

Упражнение 182. Используйте `cons` и `'()`, чтобы сформировать эквивалентные формы для следующих списков:

- 1) `(list 0 1 2 3 4 5)`
- 2) `(list (list "he" 0) (list "it" 1) (list "lui" 14))`
- 3) `(list 1 (list 1 2) (list 1 2 3))`

Выразите свои ответы с помощью `check-expect`. ■

Упражнение 183. Иногда списки формируются с помощью `cons` и `list`.

- 1) `(cons "a" (list 0 #false))`
- 2) `(list (cons 1 (cons 13 '()))))`
- 3) `(cons (list 1 (list 13 '()))) '())`
- 4) `(list '() '() (cons 1 '()))`
- 5) `(cons "a" (cons (list 1) (list #false '()))))`

Перепишите эти выражения, используя только `cons` или только `list`. Выразите свои ответы с помощью `check-expect`. ■

Упражнение 184. Определите значения следующих выражений:

- 1) `(list (string=? "a" "b") #false)`
- 2) `(list (+ 10 20) (* 10 20) (/ 10 20))`
- 3) `(list "dana" "jane" "mary" "laura")`

Выразите свои ответы с помощью `check-expect`. ■

Упражнение 185. В процессе изучения языка BSL вы познакомились с селекторами `first` и `rest`. Язык BSL+ добавляет к ним еще несколько селекторов. Определите значения следующих выражений:

- 1) `(first (list 1 2 3))`
- 2) `(rest (list 1 2 3))`
- 3) `(second (list 1 2 3))`

Загляните в документацию и прочитайте описание селекторов `third` и `fourth`. ■

11.2. Композиция функций

И не забывайте про тесты. В главе 3 мы говорили, что программы – это коллекции определений: определений типов структур, определений данных, определений констант и определений функций. Чтобы разграничить зоны ответственности между функциями, в третьей главе предлагалось следующее правило:

Спроектируйте одну функцию для каждой задачи. Сформулируйте определения вспомогательных функций для каждой зависимости между величинами в задаче.

В этой части книги вводится еще одно правило, касающееся вспомогательных функций:

Создайте один макет для каждого определения данных. Сформулируйте определения вспомогательных функций, когда одно определение данных ссылается на другое определение данных.

В этом разделе мы рассмотрим одно конкретное место в процессе проектирования, где могут потребоваться дополнительные вспомогательные функции: этап создания полноценного определения из макета. Чтобы превратить макет в полноценное определение функции, необходимо объединить значения подвыражений в макете в окончательный ответ. При этом можно столкнуться с несколькими ситуациями, указывающими на необходимость создания вспомогательных функций:

- если для объединения значений требуются знания из определенной прикладной области, например комбинирование двух (компьютерных) изображений, бухгалтерский учет, музыка или наука, то спроектируйте вспомогательную функцию;
- если для объединения значений требуется проанализировать имеющиеся значения, например положительное, нулевое или отрицательное числовое значение, то используйте выражение `cond`. Если выражение `cond` получается сложным, то спроектируйте вспомогательную функцию, которая принимает выражения из макета и анализирует их с помощью `cond`;
- если для объединения значений необходимо обработать элемент из определения данных, ссылающегося на самого себя, – список, натуральное число или что-то подобное, – спроектируйте вспомогательную функцию;
- если что-то не получается, то вам может потребоваться спроектировать **более общую** функцию и определить главную функцию как конкретное применение общей функции. Это предложение звучит нелогично, но применимо в очень большом количестве случаев.

Последние два пункта – это ситуации, которые мы не обсуждали подробно, хотя соответствующие примеры уже встречались. Следующие два раздела иллюстрируют эти принципы дополнительными примерами.

Но, прежде чем продолжить, напомним, что ключом к контролируемому проектированию программ является скрупулезное ведение списка желаний.

Список желаний

Список желаний должен содержать заголовки всех функций, необходимых для завершения программы. Записывая полные заголовки функций, вы гарантируете, что сможете протестировать те части программ, которые закончили, что полезно, даже если многие тесты потерпят неудачу. Конечно, когда список желаний реализован полностью, все тесты должны выполняться успешно, и все функции должны быть охвачены тестами.

Прежде чем добавить функцию в список желаний, проверьте, существует ли похожая функция в библиотеке для вашего языка или в списке желаний. BSL, BSL+ и все другие языки программирования предоставляют множество встроенных операций и библиотечных функций. Вам обязательно следует изучать выбранный вами язык, чтобы знать, что вам доступно.

11.3. Повторяющиеся вспомогательные функции

Людям и программам постоянно требуется что-нибудь отсортировать. Консультанты по инвестициям сортируют портфели по уровню прибыли, которую приносит каждый холдинг. Игровые программы сортируют списки игроков по количеству очков. А почтовые программы сортируют сообщения по дате, отправителю или другому критерию.

Вы тоже можете сортировать самые разные элементы данных, если есть возможность сравнить и упорядочить каждую пару элементов. Не все типы данных поддерживают элементарную операцию сравнения, но нам известен один тип, который поддерживает ее, – это числа. Поэтому здесь мы используем упрощенную, но весьма представительную задачу:

Задача. Спроектируйте функцию, которая сортирует список вещественных чисел.

Следующие упражнения объясняют, как адаптировать эту функцию к другим данным.

Поскольку в постановке задачи не упоминается никаких других задач и сама сортировка не предполагает других задач, мы просто выполним все этапы проектирования. Сортировка подразумевает перестановку чисел в коллекции. Постановка задачи предполагает естественное определение входных и выходных данных функции и, следовательно, ее сигнатуру. Учитывая, что у нас уже есть определение списка чисел, первый шаг выполняется просто:

```
| ; List-of-numbers -> List-of-numbers
| ; возвращает отсортированную версию списка alon
(define (sort> alon)
  alon)
```

Возвращая `alon`, мы гарантируем соответствие результата сигнатуре функции, но этот список не отсортирован и такой результат неверен.

Когда дело доходит до создания примеров, быстро выясняется, что постановка задачи содержит неточности. Как и раньше, мы используем определение данных `List-of-numbers` для организации примеров. Поскольку определение данных состоит из двух предложений, мы должны создать два примера. Ясно, что когда `sort>` применяется к `'()`, результатом должен быть пустой список `'()`. Вопрос в том, как должен выглядеть результат для

```
| (cons 12 (cons 20 (cons -5 '()))))
```

Это неотсортированный список, и у нас есть два способа отсортировать его:

- `(cons 20 (cons 12 (cons -5 '()))))`, то есть в порядке убывания;
- `(cons -5 (cons 12 (cons 20 '()))))`, то есть в порядке возрастания.

В реальной ситуации было бы желательно обратиться за разъяснениями к человеку, сформулировавшему задачу. Но в нашем случае мы остановимся на сортировке по убыванию; по аналогии с ней мы без труда сможем спроектировать сортировку по возрастанию.

С учетом принятого решения переиначим заголовок:

```
| ; List-of-numbers -> List-of-numbers
| ; упорядочивает элементы списка alon в порядке убывания

(check-expect (sort> '()) '())
(check-expect (sort> (list 3 2 1)) (list 3 2 1))

(check-expect (sort> (list 1 2 3)) (list 3 2 1))
(check-expect (sort> (list 12 20 -5))
              (list 20 12 -5))

(define (sort> alon)
  alon)
```

Теперь заголовок включает примеры, оформленные в виде модульных тестов, использующих `list`. Если применение этой функции вызывает у вас дискомфорт, то сформулируйте примеры с использованием `cons`, чтобы поупражняться в использовании разных форм записи. Два дополнительных примера требуют, чтобы `sort>` правильно обрабатывала списки, уже отсортированные в порядке возрастания и убывания.

Далее нужно преобразовать это определение данных в макет функции. Раньше мы уже имели дело со списками чисел, поэтому этот шаг выполняется просто:

```
(define (sort> alon)
  (cond
    [(empty? alon) ...]
    [else (... (first alon) ...
                ... (sort> (rest alon)) ...)]))
```

Используя этот макет, мы можем, наконец, перейти к самому интересному этапу разработки программы. Рассмотрим каждое условие в `cond` отдельно, начав с наиболее простого случая. Если `sort>` получает '()', то она должна вернуть '()', как определено в примере. Если `sort>` получает непустой список, то этому случаю в макете соответствуют два выражения:

- `(first alon)` извлекает первое число из входного списка;
- `(sort> (rest alon))` сортирует `(rest alon)` в порядке убывания, согласно описанию назначения функции.

Чтобы прояснить эти абстрактные ответы, воспользуемся вторым примером и подробно исследуем его части. Когда `sort>` получает `(list 12 20 -5)`,

- 1) `(first alon)` возвращает 12;
- 2) `(rest alon)` возвращает `(list 20 -5)`;
- 3) `(sort> (rest alon))` возвращает `(list 20 -5)`, потому что этот список уже отсортирован.

Чтобы получить желаемый ответ, `sort>` должна вставить 12 между двумя числами в последнем списке. Иначе говоря, мы должны найти выражение, которое вставляет `(first alon)` в нужное место в результат выражения `(sort> (rest alon))`. Если мы сможем это сделать, сортировка превратится в простую задачу.

Очевидно, что вставить число в отсортированный список непросто. Требуется выполнить поиск по отсортированному списку, чтобы найти нужное место. Для поиска в любом списке нужна вспомогательная функция, потому что списки имеют произвольный размер и, согласно пункту 3 из предыдущего раздела, для обработки значений произвольного размера необходимо спроектировать вспомогательную функцию.

Добавим новую запись в список желаний:

```
; Число List-of-numbers -> List-of-numbers
; вставляет число n в отсортированный список чисел alon
(define (insert n alon) alon)
```

То есть `insert` получает число и список, отсортированный в порядке убывания, и возвращает отсортированный список, в который в нужное место вставлено указанное число.

Имея функцию `insert`, мы с легкостью можем завершить определение `sort>`:

```
(define (sort> alon)
  (cond
```

```
[empty? alon] ')'
[else
  (insert (first alon) (sort> (rest alon)))]))
```

Чтобы получить окончательный результат, `sort>` извлекает первый элемент из непустого списка, получает отсортированную версию оставшейся части списка и применяет `insert` для создания полного отсортированного списка из двух частей.

Стоп! Протестируйте программу в текущем виде. Некоторые тестовые примеры будут выполняться успешно, а некоторые нет. Это прогресс. Следующий шаг – создание примеров применения функции. Поскольку первым аргументом `insert` является произвольное число, мы используем значение 5 и определение данных для списка чисел List-of-numbers, чтобы создать примеры для второго аргумента.

Сначала рассмотрим, что должна вернуть `insert`, получив число и пустой список `'()`. Согласно описанию назначения функции `insert`, результатом должен быть список, содержащий все числа из второго аргумента и число из первого аргумента. То есть:

```
| (check-expect (insert 5 '()) (list 5))
```

Далее используем непустой список с одним элементом:

```
| (check-expect (insert 5 (list 6)) (list 6 5))
| (check-expect (insert 5 (list 4)) (list 5 4))
```

И снова результат должен содержать все числа из списка во втором аргументе и дополнительное число из первого аргумента. А кроме того, результат должен быть отсортирован.

Наконец, создадим пример со списком во втором аргументе, содержащим более одного элемента. Мы можем получить такой пример из примеров для `sort>`. В частности, из нашего анализа второго предложения `cond` мы знаем, что `sort>` должна вставить 12 в `(list 20 -5)` в надлежащее место:

```
| (check-expect (insert 12 (list 20 -5))
|   (list 20 12 -5))
```

То есть `insert` получает во втором аргументе список, который уже отсортирован в порядке убывания.

Обратите внимание, чему нас учит развитие примеров. Функция `insert` должна найти первое число, которое меньше заданного. Если такого числа нет, то функция в конечном итоге достигнет конца списка и должна добавить `n` в конец. Теперь, прежде чем перейти к макету, создайте еще несколько примеров. Для этого вы можете использовать дополнительные примеры для функции `sort>`.

В отличие от `sort>`, функция `insert` принимает два аргумента. Мы знаем, что первый является числом и элементарным значением, поэтому сосредоточимся на втором аргументе – списке чисел:

```
(define (insert n alon)
  (cond
    [(empty? alon) ...]
    [else (... (first alon) ...
                ... (insert n (rest alon)) ...)]))
```

Единственное отличие этого макета от макета `sort` заключается в обработке дополнительного аргумента `n`.

Чтобы заполнить пробелы в макете функции `insert`, снова рассмотрим условия по отдельности. Первое условие выполняется, когда функция получает пустой список. Согласно первому примеру, в первом предложении `cond` следует использовать выражение `(list n)`, создающее отсортированный список из `n` и `alon`.

Второе условие сложнее, поэтому воспользуемся вопросами из табл. 11:

- 1) `(first alon)` – это первое число в `alon`;
- 2) `(rest alon)` остальная часть списка `alon`, отсортированная в порядке убывания;
- 3) `(insert n (rest alon))` возвращает отсортированный список, включающий `n` и числа из `(rest alon)`.

Проблема в том, как объединить эти данные, чтобы получить окончательный ответ.

Давайте рассмотрим несколько конкретных примеров:

```
| (insert 7 (list 6 5 4))
```

Здесь `n` равно 7 и больше любого числа в списке во втором аргументе. Чтобы понять это, достаточно взглянуть на первый элемент списка – число 6. Поскольку, согласно условию задачи, список отсортирован, то все остальные числа в нем меньше 6. То есть, чтобы получить окончательный ответ, достаточно объединить в список число 7 и `(list 6 5 4)`.

Напротив, когда применение `insert` выглядит так:

```
| (insert 0 (list 6 2 1 -1))
```

то число `n` должно быть вставлено в середину оставшейся части списка. Точнее говоря, `(first alon)` – это 6; `(rest alon)` – это `(list 2 1 -1)`; и применение `(insert n (rest alon))` должно вернуть `(list 2 1 0 -1)`, как того требует описание назначения. Добавляя 6 обратно в этот последний список, мы получаем желаемый ответ для `(insert 0 (list 6 2 1 -1))`.

Чтобы получить полное определение функции, обобщим эти примеры. Анализ условий предлагает анализ вложенного условия, которое сравнивает `n` с выражением `(first alon)`:

- если `n` больше (или равно) `(first alon)`, то это означает, что все элементы в `alon` меньше `n` и `alon` уже отсортирован; результатом в этом случае будет результат выражения `(cons n alon)`;

- если n меньше ($\text{first } \text{alon}$), то функция еще не нашла подходящего места для вставки n в alon . Первым элементом в результате должен быть ($\text{first } \text{alon}$), а число n должно быть вставлено в ($\text{rest } \text{alon}$). Окончательным результатом в этом случае будет

```
| (cons (first alon) (insert n (rest alon)))
```

потому что этот список содержит n и все элементы из alon в отсортированном порядке.

Для продолжения обсуждения этой темы необходимо переключиться на использование оператора `if`, поддерживаемого языком BSL+ для таких случаев. Условие ($\geq n (\text{first } \text{alon})$) и выражения для двух ветвей мы уже сформулировали.

В листинге 37 представлена полная программа сортировки. Скопируйте код в область определений DrRacket, добавьте тестовые примеры и протестируйте программу. Все тесты должны выполняться успешно и охватывать все выражения.

Терминология. Этот конкретный алгоритм сортировки известен в литературе по программированию как алгоритм *сортировки вставкой*. Позже мы познакомимся с альтернативными программами сортировки списков, при создании которых используем совершенно другую стратегию проектирования.

Листинг 37. Сортировка списка чисел

```
; List-of-numbers -> List-of-numbers
; возвращает отсортированную версию списка l
(define (sort> l)
  (cond
    [(empty? l) '()]
    [(cons? l) (insert (first l) (sort> (rest l)))]))

; Number List-of-numbers -> List-of-numbers
; вставляет число n в отсортированный список чисел l
(define (insert n l)
  (cond
    [(empty? l) (cons n '())]
    [else (if (>= n (first l))
              (cons n l)
              (cons (first l) (insert n (rest l))))]))
```

Упражнение 186. Еще раз бегло прочитайте интермеццо 1, где описывается язык BSL и поддерживаемые им способы оформления тестов. Одна из последних проверок – `check-satisfied` – определяет, удовлетворяет ли выражение определенному свойству. Используйте функцию `sorted>?` из упражнения 145 и сформулируйте тесты для `sort>` с `check-satisfied`.

Теперь рассмотрим определение следующей функции:

```
; List-of-numbers -> List-of-numbers
; возвращает отсортированную версию списка l
(define (sort>/bad l)
  (list 9 8 7 6 5 4 3 2 1 0))
```

Попробуйте сформулировать тестовый пример, который показывает, что `sort>/bad` не является функцией сортировки. Можно ли сформулировать такой тест с использованием `check-satisfied`?

Примечания. (1) Вас может удивить, что мы определяем функцию для создания теста. Однако в реальной жизни это обычный шаг, и иногда действительно нужно спроектировать функции для тестов, с их собственными тестами и всем остальным. (2) Сформулировать тесты с `check-satisfied` иногда проще, чем с `check-expect` (или с другими проверками), и это тоже вполнеично. Когда предикат полностью описывает отношения между всеми возможными входными и выходными данными функции, программисты называют такой предикат *спецификацией*. В разделе 17.4 объясняется, как написать спецификацию для `sort>`. ■

Упражнение 187. Спроектируйте программу, которая сортирует списки игроков по количеству набранных очков:

```
(define-struct gp [name score])
; GamePlayer -- это структура:
;   (make-gp String Number)
; интерпретация: (make-gp p s) представляет игрока p,
; набравшего s очков
```

Подсказка. Сформулируйте функцию, сравнивающую два экземпляра `GamePlayer`. ■

Упражнение 188. Спроектируйте программу, которая сортирует списки электронных писем по дате:

```
(define-struct email [from date message])
; EmailMessage -- это структура:
;   (make-email String Number String)
; интерпретация: (make-email f d m) представляет сообщение m,
; отправленное адресатом f, в момент времени d, измеряемый как число
; секунд, прошедших с начального момента времени
```

Также разработайте программу, которая сортирует списки электронных писем по именам отправителей. Для сравнения строк используйте элементарную функцию `string<?`. ■

Упражнение 189. Вот функция поиска `search`:

```
; Number List-of-numbers -> Boolean
(define (search nalon)
  (cond
    [(empty? alon) #false]
    [else (or (= (first alon) n)
              (search n (rest alon)))])))
```

Она определяет присутствие заданного числа в указанном списке чисел. Функция может выполнить обход всего списка, чтобы обнаружить, что интересующего числа нет.

Разработайте функцию `search-sorted`, которая определяет присутствие заданного числа в отсортированном списке чисел. Функция должна использовать тот факт, что список отсортирован. ■

Упражнение 190. Спроектируйте функцию `prefixes`, которая принимает список символов (`1String`) и создает список всех префиксов. Список `p` считается префиксом для `l`, если все элементы в `p` совпадают с первыми элементами в `l`. Например, (`list "a" "b" "c"`) является префиксом для самого себя и для (`list "a" "b" "c" "d"`).

Спроектируйте функцию `suffixes`, которая принимает список символов (`1String`) и создает список всех суффиксов. Список `s` считается суффиксом для `l`, если все элементы в `s` совпадают с последними элементами в `l`. Например, (`list "b" "c" "d"`) является суффиксом для самого себя и для (`list "a" "b" "c" "d"`). ■

11.4. Обобщающие вспомогательные функции

Иногда вспомогательная функция – это не просто маленькая функция, решающая узкую задачу, а более общая функция, способная решать широкий класс задач. Такие вспомогательные функции необходимы, когда постановка задачи слишком узкая. Прорабатывая шаги рецепта проектирования, программисты могут обнаружить, что «естественное» решение неверно. При анализе этого решения может возникнуть идея предложить несколько иную, но более общую постановку задачи, а также простой способ использования общего решения для исходной задачи.

Проиллюстрируем это на примере решения следующей задачи.

Эту задачу предложил
Пол С. Фишер
(Paul C. Fisher).

Задача. Спроектируйте функцию, которая добавляет многоугольник в заданную сцену.

На всякий случай, если вы позабыли основы геометрии (предметная область), добавим (упрощенное) определение многоугольника:

Многоугольник – это плоская фигура, имеющая не менее трех точек (не расположенных на одной прямой), соединенных тремя прямыми отрезками.

Одним из естественных представлений данных, описывающих многоугольники, является список структур `Posn`. Вот пример двух определений:

```
(define triangle-p
  (list
    (make-posn 20 10)
    (make-posn 20 20)
    (make-posn 30 20)))
(define square-p
  (list
    (make-posn 10 10)
    (make-posn 20 10)
    (make-posn 20 20)
    (make-posn 10 20)))
```

Здесь определяются треугольник (слева) и квадрат (справа). Теперь у вас может появиться вопрос: можно ли интерпретировать '(' или

(*list* (*make-posn* 30 40)) как многоугольники? Конечно нет! Эти определения не описывают многоугольники. Поскольку многоугольник состоит как минимум из трех точек, то хорошим представлением данных, описывающим многоугольники, будет множество списков, содержащих не менее трех экземпляров *Posn*.

После разработки определения данных для непустых списков температур (*NEList-of-temperature*, в разделе 9.2) мы с легкостью можем создать представление данных для многоугольников:

```
| ; Polygon – это одно из значений:  
| ; -- (List Posn Posn Posn)  
| ; -- (cons Posn Polygon)
```

Первое предложение сообщает, что список с тремя экземплярами *Posn* – это *Polygon* (многоугольник), а второе предложение – что объединение *Posn* с некоторым существующим многоугольником создает другой многоугольник. Поскольку это определение данных является первым, использующим *list* в одном из предложений, представим то же определение с использованием *cons*, просто чтобы вы могли видеть, как краткая форма записи преобразуется в длинную:

```
| ; Polygon – это одно из значений:  
| ; -- (cons Posn (cons Posn (cons Posn '())))  
| ; -- (cons Posn Polygon)
```

Как показывает этот пример, первоначально выбранное представление данных – простые списки структур *Posn* – может неправильно представлять предполагаемую информацию. Изменение определения данных на начальном этапе проектирования – это нормально; на самом деле необходимость таких изменений может возникнуть на любом другом этапе проектирования. Однако пока вы придерживаетесь систематического подхода, изменения в определении данных будут естественным образом распространяться на остальную часть процесса проектирования.

На втором шаге мы должны определить сигнатуру, сформулировать описание назначения и заголовок функции. Поскольку в постановке задачи упоминается только одна задача и не подразумевается никаких других задач, начнем с проектирования одной функции:

```
| ; простое фоновое изображение  
(define MT (empty-scene 50 50))  
| ; Image Polygon -> Image  
| ; отображает заданный многоугольник p в img  
(define (render-poly img p)  
    img)
```

Дополнительное определение *MT* необходимо, потому что оно упрощает формулировку примеров.

В первом примере мы используем вышеупомянутый треугольник. Беглый обзор библиотеки *2htdp/image* подсказывает, что функция *scene+line* – это то, что позволит нам нарисовать три стороны треугольника:

```
(check-expect
  (render-poly MT triangle-p)
  (scene+line
    (scene+line
      (scene+line MT 20 10 20 20 "red")
      20 20 30 20 "red")
      30 20 20 10 "red"))
```

Самый внутренний вызов `scene+line` отображает линию, связывающую первую и вторую точки `Posn`; средний – вторую и третью; а внешний – третью и первую.

Учитывая, что первый и самый маленький многоугольник – это треугольник, то в качестве второго примера напрашивается прямоугольник или квадрат. Мы используем `square-p`:

```
(check-expect
  (render-poly MT square-p)
  (scene+line
    (scene+line
      (scene+line
        (scene+line
          (scene+line MT 10 10 20 10 "red")
          20 10 20 20 "red")
          20 20 10 20 "red")
          10 20 10 10 "red")))
```

Количество точек (вершин) в квадрате всего на одну больше, чем в треугольнике, и его легко нарисовать. Если хотите, можете нарисовать эти фигуры на миллиметровой бумаге.

Создание макета представляет определенную сложность. В частности, первый и второй вопросы в табл. 10 спрашивают, различают ли определение данных отдельные подклассы и чем они различаются. Наше определение данных четко отделяет треугольники от всех других многоугольников в первом предложении, поэтому не сразу понятно, как их отличить. Оба предложения описывают списки структур `Posn`. Первое описывает списки из трех структур `Posn`, а второе – списки `Posn`, содержащие не менее четырех экземпляров. Одна из альтернатив – спросить, содержит ли данный многоугольник три вершины:

```
| (= (length p) 3)
```

Расширенная версия первого предложения, то есть

```
| (cons Posn (cons Posn (cons Posn '()))))
```

подсказывает второй способ формулировки первого условия: проверку, является ли данный многоугольник пустым после трех применений функции `rest`:

```
| (empty? (rest (rest (rest p))))
```

Поскольку все многоугольники имеют не менее трех вершин, трехкратное применение `rest` допустимо. В отличие от

Конечно же, нам пришлось поэкспериментировать в области взаимодействий `DrRacket`, чтобы получить правильное выражение.

Условия действительно лучше формулировать в терминах встроенных предикатов и селекторов, чем с использованием собственных (рекурсивных) функций. См. пояснения в интермецо 5.

`length`, `rest` – это элементарная и простая операция с понятным смыслом. Она просто выбирает второе поле в структуре `cons`.

На остальные вопросы в табл. 10 у нас есть прямые ответы, поэтому получаем такой макет:

```
(define (render-poly img p)
  (cond
    [(empty? (rest (rest (rest p))))]
    (... (first p) ... img ...)
    ... (second p) ...
    ... (third p) ...)
    [else (... (first p) ...
      ... (render-poly img (rest p)) ...)])))
```

Так как в первом условии `p` описывает треугольник, в списке должно содержаться точно три структуры `Posn`, которые извлекаются с помощью селекторов `first`, `second` и `third`. Во втором предложении `p` состоит из экземпляров `Posn` и `Polygon`, то есть `(first p)` и `(rest p)`. Селектор `first` извлекает экземпляр `Posn` из `p`, а `rest` – экземпляр `Polygon`. Поэтому заключаем второй селектор в рекурсивный вызов функции; также важно иметь в виду, что работа с `(first p)` в этом предложении и с тремя экземплярами `Posn` в первом предложении может потребовать от нас спроектировать вспомогательную функцию.

Теперь перейдем к определению функции и разберем предложения по одному. Первое предложение касается треугольников, что дает простой ответ. В частности, мы имеем три точки `Posn`, и `render-poly` должна соединить их отрезками в пустой сцене размером 50 на 50 пикселей. Учитывая, что `Posn` – это отдельное определение данных, добавляем очевидную запись в список желаний:

```
; Image Posn Posn -> Image
; рисует красную линию от точки Posn p до точки Posn q в img
(define (render-line img p q)
  img)
```

Первое предложение `cond` в `render-poly`, использующее эту функцию, выглядит так:

```
(render-line
  (render-line
    (render-line MT (first p) (second p))
    (second p) (third p))
    (third p) (first p)))
```

Это выражение отображает заданный многоугольник `p` как треугольник, соединяя отрезками первую вершину со второй, вторую с третьей и третью с первой.

Второе предложение в `cond` касается многоугольников с одной дополнительной вершиной. В макете имеются два выражения, поэтому, следуя вопросам в табл. 11, напомним себе, что эти выражения вычисляют:

- 1) (*first p*) извлекает первый экземпляр *Posn*;
- 2) (*rest p*) извлекает многоугольник *Polygon* из *p*;
- 3) (*render-polygon img (rest p)*) отображает (*rest p*), как указано в описании назначения функции.

Вопрос в том, как использовать все это для рисования заданного многоугольника *p*.

Первое, что приходит в голову: так как (*rest p*) содержит как минимум три экземпляра *Posn*, можно извлечь хотя бы один из них и соединить (*first p*) с этой точкой. Вот как эта идея выглядит в коде на BSL+:

```
| (render-line (render-poly MT (rest p)) (first p)
|   (second p))
```

Выделенное подвыражение рисует внутренний многоугольник в пустой сцене 50 на 50. Вызов *render-line* добавляет в эту сцену еще один отрезок, связывающий первую и вторую вершины.

По результатам анализа получаем довольно естественное и полное определение функции:

```
(define (render-poly img p)
  (cond
    [(empty? (rest (rest (rest p))))
     (render-line
       (render-line
         (render-line
           (render-line MT (first p) (second p))
           (second p) (third p))
           (third p) (first p)))
     [else
      (render-line (render-poly img (rest p))
                  (first p)
                  (second p)))]))
```

Проектирование *render-line* относится к категории задач, которые мы решали в первой части книги. Поэтому мы просто приведем окончательное определение, чтобы вы могли протестировать получившуюся функцию:

```
; Image Posn Posn -> Image
; рисует линию от p до q в img
(define (render-line img p q)
  (scene+line
    img
    (posn-x p) (posn-y p) (posn-x q) (posn-y q)
    "red"))
```

Стоп! Разработайте тест для *render-line*.

Наконец, мы должны протестировать функции. Тесты для *render-poly* терпят неудачу. С одной стороны, неудача теста – это большая удача для нас, потому что цель тестов – находить проблемы до того, как они будут обнаружены обычными пользователями. С другой стороны, неудачи вызывают сожаление, потому что мы неукоснительно

следовали рецепту проектирования и сделали естественный выбор, но функция не работает.

Стоп! Как вы думаете, почему тесты терпят неудачу? Нарисуйте изображения, которые создают разные части в макете `render-poly`. Затем нарисуйте линию, объединяющую их. Или просто поэкспериментируйте в области взаимодействий DrRacket:

```
> (render-poly MT square-p)
```



Как видно на рисунке, `render-polygon` соединяет три точки из (`rest p`), а затем добавляет отрезок, соединяющий (`first p`) с первой точкой в (`rest p`), то есть (`(second p)`). Вы можете легко проверить это утверждение в области взаимодействий, передав (`(rest square-p)` непосредственно в `render-poly`:

```
> (render-poly MT (rest square-p))
```



Вдобавок у вас может появиться вопрос, что нарисует `render-poly`, если добавить еще одну точку, скажем (`(make-posn 40 30)`), к исходному квадрату:

```
> (render-poly
  MT
  (cons (make-posn 40 30) square-p))
```



Вместо желаемого многоугольника `render-polygon` всегда рисует треугольник в конце, соединяя отрезками предшествующие вершины.

Эксперименты подтвердили ошибочность нашей реализации, но они также показывают, что она «почти правильная». Функция последовательно соединяет точки Posn, указанные в списке, а затем рисует отрезок, соединяющий первую и последнюю точки в замыкающем треугольнике. Если пропустить этот последний шаг, то функция просто «соединит точки» и нарисует «открытый» многоугольник. Если затем соединить первую и последнюю точки в многоугольнике, то мы решим поставленную задачу.

Таким образом, анализ нашей неудачи предлагает двухэтапное решение:

- 1) найти более общее решение;
- 2) использовать это решение для решения исходной задачи.

Начнем с постановки более общей задачи:

Задача. Спроектируйте функцию, которая рисует отрезки между точками в заданном наборе в заданной сцене.

Хотя наша реализация `render-poly` почти решает эту задачу, тем не менее мы начнем проектирование практически с самого начала. Во-первых, определим данные. Бессмысленно пытаться рисовать отрезок, не имея хотя бы пары точек. Для простоты потребуем наличия хотя бы одной точки:

```
; NELoP -- это одно из значений:  
; -- (cons Posn '())  
; -- (cons Posn NELoP)
```

Во-вторых, сформулируем сигнатуру, описание назначения и заголовок функции, «соединяющей точки отрезками»:

```
; Image NELoP -> Image  
; соединяет отрезками точки из списка p и отображает результат в img  
(define (connect-dots img p)  
  MT)
```

В-третьих, адаптируем примеры для `render-poly` к особенностям этой новой функции. Как показывает анализ неудачи, функция соединяет первую точку `Posn` в `p` со второй, вторую с третьей, третью с четвертой и т. д., вплоть до последней, которая не соединяется ни с какой другой точкой. Вот адаптация первого примера со списком из трех экземпляров `Posn`:

```
(check-expect (connect-dots MT triangle-p)  
  (scene+line  
    (scene+line MT 20 20 30 20 "red")  
    20 10 20 20 "red"))
```

Ожидаемый результат – изображение с двумя отрезками, соединяющими первую точку со второй и вторую с третьей.

Упражнение 191. Адаптируйте второй пример для функции `render-poly`. ■

В-четвертых, возьмем за основу макет функции, обрабатывающей непустые списки:

```
(define (connect-dots img p)  
  (cond  
    [(empty? (rest p)) (... (first p) ...)]  
    [else (... (first p) ...  
               ... (connect-dots img (rest p)) ...)])))
```

Макет состоит из условного выражения `cond` с двумя предложениями: первое обрабатывает списки с одним экземпляром `Posn`, а второе – с несколькими. Поскольку в обоих случаях имеется не менее одного экземпляра `Posn`, макет содержит `(first p)` в обоих предложениях; второе предложение содержит также выражение `(connects-dots (rest p))`, напоминающее нам о ссылке на само определение данных.

Пятый и самый важный шаг – превратить макет в определение функции. Поскольку первое предложение является самым простым, начнем с него. Как мы уже говорили, бессмысленно пытаться рисовать отрезки, если в заданном списке имеется только одна точка. По-

этому первое предложение в `cond` просто возвращает `Mt`. Теперь перейдем ко второму предложению и напомним себе, что вычисляют выражения в нем:

- 1) `(first p)` извлекает первый экземпляр `Posn` из списка;
- 2) `(rest p)` извлекает `NELoP` из `p`;
- 3) `(connect-dots img (rest p))` соединяет отрезками точки из `(rest p)` в `img`.

По результатам первой попытки спроектировать `render-poly` мы знаем, что `connect-dots` должна добавить еще один отрезок в результат, возвращаемый выражением `(connect-dots img (rest p))`, соединяющий `(first p)` и `(second p)`. Мы знаем, что `p` содержит вторую точку, потому что в противном случае было бы выбрано первое предложение в `cond`.

С учетом вышесказанного получаем следующее определение:

```
(define (connect-dots img p)
  (cond
    [(empty? (rest p)) img]
    [else
      (render-line
        (connect-dots img (rest p))
        (first p)
        (second p)))]))
```

Это неформальное рассуждение. Если для обоснования подобных утверждений о взаимо- связях между множествами или функциями вам когда-нибудь понадобится **формальное** рассуждение, то вам придется изучить логику. Процесс проектирования, обсуж- даемый в этой книге, действительно прочно основан на логике, и изучение логики было бы естественным дополнением к изучению этого процесса. Вообще говоря, логика относит- ся к вычислениям так же, как анализ – к инженерии.

Это определение выглядит проще ошибочной версии `render-poly`, даже притом что обрабатывает более широкое множество списков `Posn`, чем `render-poly`.

Мы говорим, что `connect-dots` обобщает `render-poly`. Любое входное значение, допустимое для `render-poly`, также является допустимым вводом для `connect-dots`. Если использовать терминологию определений данных, то каждый экземпляр `Polygon` также является экземпляром `NELoP`. Но есть такие экземпляры `NELoP`, которые не являются экземплярами `Polygon`. Если говорить точнее, то любые списки экземпляров `Posn`, содержащие один или два элемента, принадлежат множеству `NELoP`, но не принадлежат множеству `Polygon`. Однако самый важный для нас вывод заключается в том, что способность функции обрабатывать более широкое множество экземпляров входных данных, по сравнению с другой функцией, не означает, что первая сложнее второй; обобщенные версии функций часто оказываются более простыми.

Как говорилось выше, `render-polygon` может использовать `connect-dots` для последовательного соединения отрезками всех экземпляров `Posn` в заданном экземпляре `Polygon`. А чтобы выполнить поставленную перед ней задачу, она должна добавить отрезок, соединяющий первую и последнюю точки в `Polygon`. С точки зрения кода это просто означает создание двух функций: `connect-dots` и `render-line`, но нам также нужна функция, извлекающая последнюю точку `Posn` из `Poly-`

gon. Как только это желание будет исполнено, определение render-poly сократится до одной строки:

```
; Изображение Polygon -> Изображение
; добавляет изображение многоугольника p в сцену img
(define (render-polygon img p)
  (render-line (connect-dots img p)
    (first p)
    (last p)))
```

Определение последней функции в списке желаний выглядит просто:

```
; Polygon -> Posn
; извлекает последний экземпляр из p
```

И снова достаточно очевидно, что с практической точки зрения полезнее реализовать last как обобщенную функцию, принимающую экземпляр NELoP:

```
; NELoP -> Posn
; извлекает последний элемент из p
(define (last p)
  (first p))
```

Стоп! Почему селектор first является допустимой заглушкой для last?

Упражнение 192. Аргументируйте допустимость применения last к экземплярам Polygon. Также объясните, почему макет connect-dots можно использовать для last:

```
(define (last p)
  (cond
    [(empty? (rest p)) (... (first p) ...)]
    [else (... (first p) ... (last (rest p)) ...)]))
```

Наконец, добавьте примеры для last, преобразуйте их в тесты и убедитесь, что они успешно выполняются с определением last из листинга 38. ■

Подводя итог, можно сказать, что разработка render-poly естественным образом заставила нас задуматься над более общей задачей соединения отрезками точек в списке. После этого мы смогли решить исходную задачу, определив функцию, которая комбинирует обобщенную функцию с другими вспомогательными функциями. В результате мы получили программу, состоящую из относительно простой основной функции render-poly и сложных вспомогательных функций, которые выполняют большую часть работы. В своей практике вы снова и снова будете видеть, что такой подход к проектированию является наиболее эффективным методом разработки и организации программ.

Упражнение 193. Вот еще две идеи для определения render-poly:

- render-poly может добавить последний элемент из p в начало p, а затем вызвать connect-dots;

- `render-poly` может добавить первый элемент из `p` в конец `p` с помощью версии `add-at-end`, работающей с экземплярами `Polygon`.

Реализуйте обе идеи и убедитесь, что в обоих случаях тесты выполняются успешно. ■

Упражнение 194. Измените функцию `connect-dots` так, чтобы она принимала дополнительный экземпляр `Posn` для соединения отрезком с последним элементом в списке точек `Posn`. Затем измените `render-poly`, чтобы она использовала эту новую версию `connect-dots`. ■

В полноценных языках программирования уже имеются такие функции, как `last`, а функция, напоминающая `render-poly`, имеется в библиотеке `2htdp/image`. Если вам интересно, почему мы только что спроектировали эти функции, то обратите внимание на название книги и этого раздела. Цель не в том, чтобы (просто) спроектировать полезные функции, а в том, чтобы изучить систематический подход к проектированию кода. В частности, этот раздел посвящен идее обобщения в процессе проектирования; и мы еще не раз вернемся к ней в III и VI частях книги.

Листинг 38. Рисование многоугольника

```

; Image Polygon -> Image
; добавляет изображение многоугольника p в сцену MT
(define (render-polygon img p)
  (render-line (connect-dots img p) (first p) (last p)))

; Image NELoP -> Image
; соединяет отрезками точки из списка p в img
(define (connect-dots img p)
  (cond
    [(empty? (rest p)) MT]
    [else (render-line (connect-dots img (rest p))
                       (first p)
                       (second p))]))

; Image Posn Posn -> Image
; рисует красную линию от точки p до точки q в im
(define (render-line im p q)
  (scene+line
    im (posn-x p) (posn-y p) (posn-x q) (posn-y q) "red"))

; Polygon -> Posn
; извлекает последний элемент из p
(define (last p)
  (cond
    [(empty? (rest (rest (rest p))))] (third p)]
    [else (last (rest p))]))

```

12. Проекты: списки

В данной главе представлено несколько дополнительных упражнений, направленных на укрепление вашего понимания таких аспектов проектирования, как проектирование пакетных и интерактивных программ, проектирование методом композиции, составление списков желаний и рецепты проектирования функций. Первые разделы посвящены проблемам, связанным с реальными данными: словарями английских слов и медиатеками iTunes. Затем следуют два раздела, посвященных задаче проектирования игр со словами: в одном иллюстрируется проектирование методом композиции, а в другом обсуждается суть проблемы. Остальные разделы посвящены играм и конечным автоматам.

В этой главе используется библиотека `2http/batch-io`.

12.1. Реальные данные: словари

В реальном мире наблюдается тенденция увеличения объема информации, которую требуется проанализировать, поэтому для ее обработки желательно использовать компьютерные программы. Например, словарь английского языка содержит сотни тысяч слов. Программы, обрабатывающие такие большие объемы информации, должны проектироваться тщательно, с использованием небольших примеров. Убедившись в правильной работе программы, ее можно использовать для обработки реальных данных, чтобы получить реальные результаты. Если программа слишком медленно обрабатывает такой большой объем данных, то нужно внимательно исследовать каждую функцию и оценить ее работу – возможно, она выполняет какие-либо избыточные вычисления.

Проблемы производительности будут рассматриваться в части V. А мы с этого момента будем уделять основное внимание систематическому проектированию программ, чтобы потом вы могли успешно исследовать проблемы с производительностью.

Листинг 39. Чтение словаря

```
; В OS X:  
(define LOCATION "/usr/share/dict/words")  
  
; В LINUX: /usr/share/dict/words или /var/lib/dict/words  
; В WINDOWS: возьмите файл words у своих друзей или коллег,  
; использующих Linux  
; Dictionary – это список строк List-of-strings.  
(define AS-LIST (read-lines LOCATION))
```

В листинге 39 показана одна строка кода, необходимая для чтения словаря английского языка целиком. Чтобы получить представление о том, насколько велики такие словари, адаптируйте код из листинга 39 для вашего конкретного компьютера и используйте функцию

`length`, чтобы узнать, сколько слов в вашем словаре. Сегодня 25 июля 2017 года, и в нашем словаре имеется 235 886 слов.

Буквы в следующих упражнениях играют важную роль. Вы можете добавить следующий код в начало вашей программы в добавок к адаптированному коду из листинга 39:

```
; Letter -- один из следующих символов:  
; -- "a"  
; -- ...  
; -- "z"  
; что эквивалентно применению member? к следующему списку:  
(define LETTERS  
  (explode "abcdefghijklmnopqrstuvwxyz"))
```

Подсказка. Формулируйте примеры и тесты для упражнений с использованием `list`.

Упражнение 195. Спроектируйте функцию `starts-with#`, которая принимает экземпляры `Letter` (букву) и `Dictionary` (словарь), а затем подсчитывает слова в словаре, начинающиеся с указанной буквы. Когда вы добьетесь правильной работы вашей функции, определите, сколько слов в словаре на вашем компьютере начинаются с буквы «е», а сколько с буквы «з». ■

Упражнение 196. Спроектируйте функцию `count-by-letter`. Она должна принимать словарь `Dictionary` и подсчитывать слова в данном словаре, начинающиеся с каждой буквы. Функция должна возвращать список `Letter-Count`, экземпляров данных, объединяющих буквы и числа.

Закончив проектировать функцию, определите, сколько слов в вашем словаре начинаются на каждую букву.

Замечание о выборе дизайна. Как вариант можно спроектировать вспомогательную функцию, которая принимает список букв и словарь, создает список `Letter-Count`, содержащий указанные буквы, и подсчитывает слова в словаре, начинающиеся с каждой из них. Можно повторно использовать решение упражнения 195. **Подсказка.** Если вы решите спроектировать этот вариант, то обратите внимание, что функция принимает два списка, а это требует решения проблемы проектирования, подробно описанной в главе 23. Рассматривайте словарь `Dictionary` как атомарный фрагмент данных, который просто передается в `start-with#` по мере необходимости. ■

Упражнение 197. Спроектируйте функцию `most-frequent`. Она должна принимать словарь `Dictionary` и возвращать экземпляр `Letter-Count` с буквой, с которой начинается больше всего слов в данном словаре.

С какой буквы начинается больше всего слов в вашем словаре, и сколько таких слов?

Замечание о выборе дизайна. В этом упражнении предлагается получить решение комбинированием функции из предыдущего упражнения с новой функцией, которая выбирает правильную пару из списка экземпляров `Letter-Count`. Вот два из возможных способов спроектировать эту последнюю функцию:

- спроектируйте функцию, которая выбирает пару с максимальным значением счетчика;
- спроектируйте функцию, которая выбирает первую пару из списка, отсортированного по значениям счетчика.

Попробуйте спроектировать оба варианта. Какой из них вам нравится больше? Почему? ■

Упражнение 198. Спроектируйте функцию `words-by-first-letter`. Она должна принимать словарь `Dictionary` и возвращать список словарей `Dictionary`, по одному для каждой буквы.

Измените функцию `most-frequent` из упражнения 197 так, чтобы она использовала эту новую функцию. Назовите новую функцию `most-frequent.v2`. Завершив проектирование, убедитесь, что обе функции дают один и тот же результат для вашего словаря:

```
(check-expect
  (most-frequent AS-LIST)
  (most-frequent.v2 AS-LIST))
```

ЗАМЕЧАНИЕ О ВЫБОРЕ ДИЗАЙНА. При проектировании `words-by-first-letter` вы должны учесть возможность ситуации, когда данный словарь не содержит слов для какой-либо буквы:

- один из вариантов – исключить получившиеся пустые слова из общего результата. Это упростит тестирование функции и проектирование `most-frequent.v2`, но также потребует проектирования вспомогательной функции;
- другой вариант – включить '()' в результат, если обнаружится, что слов, начинающихся с определенной буквы, нет в словаре. Этот вариант позволяет избежать проектирования вспомогательной функции, но усложняет проектирование `most-frequent.v2`. **КОНЕЦ.**

Замечание о промежуточных данных и дефорестации. Эта вторая версия функции подсчета слов вычисляет результат путем создания большой структуры с промежуточными данными, которая нужна только для подсчета. Некоторые языки программирования поддерживают возможность автоматического избавления от таких структур путем *объединения* двух функций в одну. Такое преобразование программ иногда также называют *дефорестация*. Если вы знаете, что ваш язык не поддерживает это преобразование, то подумайте о том, как можно избавиться от таких структур данных, если программа обрабатывает данные недостаточно быстро. ■

12.2. Реальные данные: iTunes

Приложение Apple iTunes широко используется для создания коллекций музыки, видео, телешоу и т. д. Вы можете проанализировать ин-

формацию, которую собирает ваше приложение iTunes. Извлечь эту информацию довольно просто. Выберите в меню приложения пункт **File > Library > Export** (Файл > Библиотека > Экспорт), и вам будет предложено указать имя файла XML для сохранения информации из iTunes. Обработка XML-файлов подробно рассматривается в главе 22, а пока мы просто будем использовать библиотеку *2htdp/itunes*. Эта библиотека позволяет, в частности, извлекать музыкальные треки, содержащиеся в медиатеке iTunes.

Некоторые тонкости могут отличаться в зависимости от версии, но в общем и целом iTunes поддерживает следующие элементы информации для каждого музыкального трека:

- *идентификатор трека* – уникальный идентификатор трека в вашей медиатеке, например 442;
- *название* – название трека, например Wild Child;
- *автор* – авторы трека, например Enya;
- *альбом* – название альбома, в состав которого входит трек, например A Day Without;
- *жанр* – музыкальный жанр трека, например New Age;
- *тип* – вид кодировки трека, например MPEG audio file;
- *размер* – размер файла, например 4562044;
- *общее время* – продолжительность трека в миллисекундах, например 227996;
- *номер трека* – порядковый номер трека в альбоме, например 2;
- *количество треков* – количество треков в альбоме, например 11;
- *год* – год выпуска, например 2000;
- *дата добавления* – когда был добавлен трек, например 2002-7-17 3:55:14;
- *счетчик проигрывания* – сколько раз воспроизводился трек, например 20;
- *дата воспроизведения* – когда в последний раз воспроизводился трек, например 3388484113 секунд от начала эпохи Unix;
- *дата воспроизведения UTC* – когда в последний раз воспроизводился трек по всемирному координированному времени, например 2011-5-17 17:35:13.

Кроме *2htdp/batch-io*,
в этом разделе
используется также
библиотека *2htdp/itunes*.

Как всегда, первая задача – выбрать представление данных для этой информации. В этом разделе мы используем два представления музыкальных треков: на основе структур и на основе списков. Первое хранит фиксированное количество атрибутов треков и только тех, которые доступны, последнее позволяет сохранить любую доступную информацию. Каждое представление хорошо подходит для конкретного использования; в некоторых случаях могут пригодиться оба представления.

Листинг 40. Представление треков в iTunes в виде структур (структуры)

```
; документация к библиотеке 2htdp/itunes, часть 1:
; LTracks -- это одно из значений:
; -- '()
; -- (cons Track LTracks)

(define-struct track
  [name artist album time track# added play# played])
; Track -- это структура:
;   (make-track String String String N N Date N Date)
; интерпретация: представляет информацию в следующем порядке:
; название трека, автор, альбом,
; продолжительность в миллисекундах, порядковый номер в
; альбоме, дата добавления, счетчик проигрываний
; и дата последнего воспроизведения

(define-struct date [year month day hour minute second])
; Date -- это структура:
;   (make-date N N N N N N)
; интерпретация: представляет шесть элементов информации:
; год, месяц (от 1 до 12 включительно),
; день (от 1 до 31), часы (от 0 до 23),
; минуты (от 0 до 59) и
; секунды (тоже от 0 до 59).
```

Листинг 41. Представление треков в iTunes в виде структур (функции)

```
; Any Any Any Any Any Any Any Any -> Track или #false
; создает экземпляр Track, если на входе получены допустимые данные
; иначе возвращает #false
(define (create-track name artist album time
                      track# added play# played)
  ...)

; Any Any Any Any Any Any -> Date или #false
; создает экземпляр Date, если на входе получены допустимые данные
; иначе возвращает #false
(define (create-date y mo day h m s)
  ...)

; Стока -> LTracks
; создает список треков из содержимого XML-файла
; с именем file-name (экспортированного из iTunes)
(define (read-itunes-as-tracks file-name)
  ...)
```

В листингах 40 и 41 показано представление треков на основе структур, реализованное в библиотеке *2htdp/itunes*. Структура *track* состоит из восьми полей, каждое представляет определенное свойство трека. Большинство полей содержат данные элементарных типов, такие как строки и числа; другие содержат даты, представленные структурами с шестью полями. Библиотека *2htdp/itunes* экспортирует все предикаты и селекторы для структур *track* и *date*, но вместо простых конструкторов предоставляет проверяющие конструкторы.

Последний элемент из описания библиотеки *2htdp/itunes* – функция, которая читает данные из медиатеки iTunes в формате XML

и возвращает список треков LTracks. После экспорта медиатеки из какого-либо приложения iTunes вы можете запустить следующий фрагмент кода и получить все записи:

```
; подставьте свое имя файла
(define ITUNES-LOCATION "itunes.xml")

; LTracks
(define itunes-tracks
  (read-itunes-as-tracks ITUNES-LOCATION))
```

Сохраните этот фрагмент в той же папке, куда экспортировали XML-файл из iTunes. Кроме того, не используйте `itunes-tracks` в примерах, потому что она выполняет слишком много работы для этого. Если вы проигнорируете это предупреждение, то тестовые примеры будут читать файл при каждом запуске программы в DrRacket, а это может занять очень много времени. Поэтому мы предлагаем закомментировать эту вторую строку и раскомментировать ее только тогда, когда вам понадобится получить информацию о своей коллекции iTunes.

Упражнение 199. Хотя важные определения данных уже представлены, первый шаг рецепта проектирования еще не завершен. Придумайте примеры дат (Date), треков (Track) и списков треков (LTracks). Эти примеры пригодятся в следующих упражнениях в роли исходных данных. ■

Упражнение 200. Спроектируйте функцию `total-time`, которая принимает экземпляр LTracks и возвращает общее время воспроизведения всех треков в нем. Закончив проектирование, вычислите общее время воспроизведения вашей коллекции iTunes. ■

Упражнение 201. Спроектируйте `select-all-album-titles`. Функция должна принимать LTracks и возвращать список с названиями альбомов в виде списка строк типа `List-of-strings`.

Также спроектируйте функцию `create-set`, которая принимает список строк `List-of-strings` и создает список строк, содержащихся в исходном списке точно один раз. **Подсказка.** Если строка `s` типа `String` находится в начале исходного списка и повторно встречается в остальной части этого списка, то `create-set` не должна добавлять `s` в выходной список.

Наконец, спроектируйте функцию `select-album-title/unique`, которая принимает список треков LTracks и возвращает список уникальных названий альбомов. Используйте эту функцию, чтобы определить названия всех альбомов в вашей коллекции iTunes и узнать, сколько уникальных названий альбомов она содержит. ■

Упражнение 202. Спроектируйте `select-album`. Функция должна принимать название альбома и список треков LTracks и извлекать из LTracks список треков, принадлежащих данному альбому. ■

Упражнение 203. Спроектируйте `select-album-date`. Функция должна принимать название альбома, дату и LTracks и извлекать из LTracks список треков, принадлежащих данному альбому и воспроизводив-

шихся после указанной даты. **Подсказка.** Спроектируйте вспомогательную функцию, которая принимает две даты типа Date и определяет, является ли первая дата более ранней, чем вторая. ■

Упражнение 204. Спроектируйте `select-albums`. Функция должна принимать экземпляр `LTracks` и создавать список экземпляров `LTracks`, по одному для каждого альбома. Альбомы должны уникально идентифицироваться по названию и присутствовать в результатах только один раз.

Подсказки. (1) Используйте некоторые решения предыдущих упражнений. (2) Функция группировки должна принимать два списка: список названий альбомов и список треков; последний считается атомарным, пока не будет передан вспомогательной функции. См. упражнение 196. ■

ТЕРМИНОЛОГИЯ. Функции, имена которых начинаются с `select-`, называются *запросами к базе данных*. Более подробно запросы рассматриваются в разделе 23.7. **КОНЕЦ.**

Листинг 42. Представление списков треков в iTunes

```
; документация к библиотеке 2htdp/itunes, часть 2:
; LLists -- это одно из значений:
; -- '()
; -- (cons LAssoc LLists)

; LAssoc -- это одно из значений:
; -- '()
; -- (cons Association LAssoc)
;

; Association -- это список с двумя элементами:
;   (cons String (cons BSDN '()))

; BSDN -- это одно из значений:
; -- Boolean
; -- Number
; -- String
; -- Date

; String -> LLists
; создает список списков, представляющий все треки в
; файле file-name, экспортированном из iTunes
(define (read-itunes-as-lists file-name)
  ...)
```

В листинге 42 показано, как библиотека `2htdp/itunes` представляет треки в виде списков. `LLists` – это представление трека в виде списка, являющееся списком списков, объединяющих строки с четырьмя видами значений. Функция `read-itunes-as-lists` читает XML-медиатеку iTunes и создает экземпляр `LLists`, посредством которого вы можете получить доступ ко всей информации о треках, если добавите в программу следующие определения:

```
; подставьте свое имя файла
(define ITUNES-LOCATION "itunes.xml")
```

```

; LLists
(define list-tracks
  (read-itunes-as-lists ITUNES-LOCATION))

```

Сохраните этот фрагмент в той же папке, куда экспорттировали XML-файл из iTunes.

Упражнение 205. Разработайте примеры использования LAssoc и LLists, то есть представление треков в виде списков и списка таких треков. ■

Упражнение 206. Спроектируйте функцию `find-association`. Она должна принимать три аргумента (строку String с именем `key`, экземпляр LAssoc и экземпляр Any с именем `default`) и возвращать первый экземпляр Association, первый элемент которого равен строке `key`, или значение `default`, если такого экземпляра нет в списке ассоциаций.

Примечание. Закончив проектировать функцию, найдите в документации описание функции `assoc` и прочитайте его. ■

Упражнение 207. Спроектируйте `total-time/list`, которая принимает экземпляр списка LLists и возвращает общее время воспроизведения всех треков в нем. **Подсказка.** Сначала решите упражнение 206.

Закончив проектирование, вычислите общее время воспроизведения вашей коллекции iTunes. Сравните полученный результат с результатом, который возвращает функция `total-time` из упражнения 200. Почему результаты отличаются? ■

Упражнение 208. Спроектируйте `boolean-attributes`. Функция должна принимать экземпляр LList и извлекать строки, связанные с логическими атрибутами. **Подсказка.** Используйте `create-set` из упражнения 201.

Закончив проектирование, определите, сколько логических атрибутов содержится в вашей медиатеке iTunes. Имеют ли они смысл? ■

ПРИМЕЧАНИЕ. Представление треков в виде списков менее организованное, чем представление с использованием структуры. В таких случаях иногда используют слово *полуструктурированный* (*semi-structured*). Подобные списковые представления позволяют хранить свойства, которые появляются редко и, следовательно, не подходят для включения в структуру. Люди часто используют такие представления для исследования неизвестной информации и после изучения формата создают структуры. Спроектируйте функцию `track-as-struct`, которая по возможности преобразует LAssoc в Track. **КОНЕЦ.**

12.3. Игры со словами, иллюстрация приема композиции

Многие из нас решают словесные головоломки в газетах и журналах. Попробуйте такую задачку:

Задача. Для данного слова найдите все слова, состоящие из одинаковых букв. Например, из букв в слове «cat» можно составить слово «act».

Рассмотрим этот пример поближе. Предположим, нам дали слово «dear». Всего есть двадцать четырех возможных варианта перестановок этих четырех букв:

```
ader aedr aerdae rade reda
daer eadr eard dare rade raed
dear edar erad drae rdae read
dera edra erda drea rdea reda
```

В этом списке имеется три допустимых слова: «read», «dear» и «dare».

ПРИМЕЧАНИЕ. Если слово содержит одну и ту же букву дважды, набор всех перестановок может содержать несколько копий одной и той же строки. Для наших целей это допустимо. Но в реальных программах желательно избегать такого дублирования, используя множества вместо списков. См. раздел 9.6. **КОНЕЦ**.

Систематическое перечисление всех возможных вариантов, как и поиск в словаре английского языка, лучше поручить программе. В этом разделе описывается процесс проектирования функции поиска, а решение другой задачи мы отложим до следующего раздела. Разделив поставленную задачу на две подзадачи, в этом первом разделе мы сосредоточимся на высокоуровневых идеях систематического проектирования программ.

См. раздел 12.1, где описываются приемы работы с реальными словарями.

Давайте представим на мгновение, как бы мы решили задачу вручную. При наличии достаточного времени мы могли бы выписать все возможные варианты перестановки букв в данном слове, а затем оставить те варианты, которые тоже встречаются в словаре. Очевидно, что программа может действовать точно так же, и это предполагает использование приема композиции. Но, как всегда, будем действовать систематически и начнем с представления входных и выходных данных.

На первый взгляд кажется естественным представить слова в виде строк типа *String*, а результат в виде списка слов, то есть списка строк *List-of-String*. Опираясь на этот выбор, можно сформулировать сигнатуру и описание назначения:

```
; String -> List-of-strings
; отыскивает все слова, в которых используются те же буквы,
; что и в исходном слове s
(define (alternative-words s)
  ...)
```

Далее нужно определить несколько примеров. Если задано слово «cat», мы имеем дело с тремя буквами: *c*, *a* и *t*. Всего есть шесть перестановок этих букв: *cat*, *cta*, *tca*, *tac*, *act* и *atc*. Две из них соответствуют

настоящим словам: «cat» и «act». Поскольку `alternative-words` создает список строк, мы можем представить результат двумя способами: (`list "act" "cat"`) и (`list "cat" "act"`). К счастью, в BSL есть возможность указать, что функция может вернуть один из двух вариантов:

```
(check-member-of (alternative-words "cat")
                  (list "act" "cat")
                  (list "cat" "act"))
```

Стоп! Прочтайте описание `check-member-of` в документации. Этот пример обнажает две проблемы:

- первая связана с тестированием. Предположим, мы использовали слово «rat», у которого есть три альтернативы: «rat», «tar» и «art». В этом случае мы должны составить шесть списков, каждый из которых мог бы быть результатом функции. Для такого слова, как «dear» с четырьмя возможными альтернативами, сформулировать примеры еще сложнее;
- вторая проблема касается выбора представления слов. Выбор строк сначала выглядит естественным, но, как показывают примеры, некоторые из функций должны рассматривать слова как последовательности букв с возможностью их перестановки. Конечно, буквы можно переставить и в строке, но списки букв лучше подходят для этой цели.

Давайте разберемся с этими проблемами по очереди, начав с тестов.

Предположим, нам нужно сформировать тест для `alternative-words` и слова «rat». Мы уже знаем, что результат должен содержать слова «rat», «tar» и «art», но мы не можем знать, в каком порядке эти слова появятся в результате.

См. интермеццо 1. В этой ситуации нам пригодится `check-satisfied`. Мы можем использовать эту проверку с функцией, которая проверяет, содержит ли список строк три указанные строки:

```
; List-of-strings -> Boolean
(define (all-words-from-rat? w)
  (and (member? "rat" w)
       (member? "art" w)
       (member? "tar" w)))
```

С помощью этой функции легко сформулировать проверку `alternative-words`:

```
(check-satisfied (alternative-words "rat")
                  all-words-from-rat?)
```

ПРИМЕЧАНИЕ. Это обсуждение предполагает, что функция `alternative-words` создает множество, а не список. Подробное обсуждение различий см. в разделе 9.6. Здесь достаточно знать, что множества представляют коллекции значений без учета порядка их следования

или количества их вхождений в коллекцию. Когда язык не поддерживает множества, программисты склонны использовать близкие альтернативы, такие как список строк, например. Но по мере развития программы этот выбор начинает преследовать программистов, однако решение подобных проблем является предметом второй книги.

КОНЕЦ.

Листинг 43. Поиск альтернативных слов

```
; List-of-strings -> Логическое значение
(define (all-words-from-rat? w)
  (and
    (member? "rat" w) (member? "art" w) (member? "tar" w)))

; Стока -> List-of-strings
; отыскивает все слова, в которых используются те же буквы,
; что и в исходном слове

(check-member-of (alternative-words "cat")
  (list "act" "cat")
  (list "cat" "act"))

(check-satisfied (alternative-words "rat")
  all-words-from-rat?)

(define (alternative-words s)
  (in-dictionary
    (words->strings (arrangements (string->word s)))))

; List-of-words -> List-of-strings
; преобразует в строки все экземпляры Word в списке low
(define (words->strings low) '())

; List-of-strings -> List-of-strings
; оставляет в списке только те слова, которые присутствуют в словаре
(define (in-dictionary los) '())(index "in-dictionary")
```

О проблеме с представлением слов мы поговорим в следующем разделе. В частности, в следующем разделе мы создадим (1) представление данных Word для слов, подходящее для перестановки букв, (2) определение данных List-of-words для списка слов и (3) функцию, которая отображает Word в List-of-words – список всех допустимых перестановок:

```
; Word -- это ...
; List-of-words -- это ...
; Word -> List-of-words
; отыскивает все перестановки букв, присутствующих в слове word
(define (arrangements word)
  (list word))
```

Упражнение 209. После вышесказанного в нашем списке желаний появились два пункта: функция, которая принимает строку и создает соответствующий ей экземпляр Word, и противоположная ей функ-

ция, преобразующая экземпляр Word в строку. Вот соответствующие записи из списка желаний:

```
| ; String -> Word
| ; преобразует s в представление Word
| (define (string->word s) ...)

| ; Word -> String
| ; преобразует w в строку
| (define (word->string w) ...)
```

Найдите определение данных Word в следующем разделе и дополните определения `string->word` и `word->string`. **Подсказка.** Можете посмотреть список функций, которые предоставляет язык BSL. ■

Теперь, когда вы написали эти две короткие функции, вернемся к проектированию `alternative-words`. Итак, у нас есть: (1) сигнатура, (2) описание назначения, (3) примеры и тест, (4) понимание причин такого выбора представления данных и (5) идея деления задачи на две подзадачи.

Поэтому вместо создания макета мы запишем задуманную композицию:

```
| (in-dictionary (arrangements s))
```

Это выражение говорит о том, что к заданному слову `s` применяется функция `arrangements`, создающая список всех возможных перестановок букв, и `in-dictionary` для выбора перестановок, которые присутствуют в словаре.

Стоп! Найдите сигнатуры для этих двух функций и убедитесь, что такая композиция возможна. На что нужно обратить внимание?

Это выражение не отражает четвертого пункта – причин решения не использовать простые строки для поиска перестановок букв. Прежде чем вручить `s` функции `arrangements`, нужно преобразовать его в слово – экземпляр Word. К счастью, в упражнении 209 как раз было предложено спроектировать эту функцию:

```
| (in-dictionary
|   (... (arrangements (string->word s))))
```

Точно так же получившийся список слов нужно преобразовать в список строк. В упражнении 209 было предложено спроектировать функцию, преобразующую одно слово, а здесь нам нужна функция, применяющая преобразование к списку слов. Пора добавить в список еще одно желание:

```
| (in-dictionary
|   (words->strings
|     (arrangements (string->word s))))
```

Стоп! Определите для `words->strings` сигнатуру и описание назначения.

В листинге 43 представлены все части. В следующих упражнениях вам будет предложено спроектировать остальные функции.

Упражнение 210. Завершите проектирование функции `words->strings`, заголовок которой показан в листинге 43. **Подсказка.** Используйте свое решение из упражнения 209. ■

Упражнение 211. Завершите проектирование `in-dictionary`, заголовок которой показан в листинге 43. **Подсказка.** См. раздел 12.1, чтобы вспомнить, как читать словарь. ■

12.4. Игры со словами, суть проблемы

Цель состоит в том, чтобы спроектировать функцию `аг-
rangements`, которая принимает слово – экземпляр `Word` – и создает список разных комбинаций из составляющих его букв. Это усложненное упражнение усиливает потребность в создании подробного списка желаний, то есть списка желаемых функций, который, кажется, только увеличивается с каждой спроектированной нами функцией.

Как уже упоминалось, для представления слов можно использовать строки, но строки являются атомарными значениями, а сам факт того, что `аг-
rangements` должна переупорядочивать буквы слова, требует другого представления. Поэтому для представления данных, определяющих слово, мы выбрали список символов, в котором каждый элемент представляет букву:

```
; Word – это одно из значений:  
; -- ()  
; -- (cons 1String Word)  
; интерпретация: Word – это список символов (1String)
```

Упражнение 212. Запишите определение данных для списка слов `List-of-words`. Придумайте примеры слов и списков слов `List-of-words`. Наконец, сформулируйте пример применения функции выше с помощью `check-expert`. Для простоты подумайте, как функция будет обрабатывать слово, состоящее всего из двух букв, скажем «д» и «е». ■

Макет `аг-
rangements` имеет типичную организацию для функций, обрабатывающих списки:

```
; Word -> List-of-words  
; отыскивает все перестановки букв, присутствующих в слове w  
(define (arrangements w)  
(cond  
  [(empty? w) ...]  
  [else (... (first w) ...  
            ... (arrangements (rest w)) ...)])))
```

Готовясь сделать пятый шаг, посмотрим на строки `cond` в макете.

- Если входное слово представлено пустым списком '(), то возможна только одна перестановка: '(), то есть результатом яв-

Математический термин – перестановки.

ляется (`list '()`), список, содержащий единственный элемент с пустым списком.

2. В противном случае в слове точно будет иметься первая буква – (`first w`), а рекурсия даст нам список всех возможных перестановок для остальной части слова. Например, для списка

```
| (list "d" "e" "r")
```

рекурсия будет иметь вид (`arrangements (list "e" "r")`) и вернет

```
| (cons (list "e" "r")
|       (cons (list "r" "e")
|                 '()))
```

Чтобы получить все возможные перестановки для всего списка, мы должны теперь вставить первый элемент, в нашем случае "`d`", во все эти слова между всеми возможными буквами, а также в начало и в конец.

Наш анализ показывает, что мы можем завершить `arrangements`, если сумеем каким-то образом вставить одну букву во все возможные позиции во всех словах в полученном списке. Последний аспект этого описания задачи неявно указывает на списки и, как того требуют советы в этой главе, предполагает создание вспомогательной функции. Назовем эту функцию `insert-everywhere/in-all-words` и воспользуемся этим именем, чтобы завершить определение `arrangements`:

```
| (define (arrangements w)
|   (cond
|     [(empty? w) (list '())]
|     [else (insert-everywhere/in-all-words (first w)
|                                           (arrangements (rest w))))]))
```

Упражнение 213. Спроектируйте `insert-everywhere/in-all-words`. Она должна принимать символ и список слов и возвращать список слов из второго аргумента, но с первым аргументом, вставленным в начало, между всеми буквами и в конец каждого слова в данном списке.

Добавьте эту функцию в список желаний. Дополните его тестами, проверяющими применение функции к пустому списку, к списку с однобуквенным словом, к списку с двухбуквенным словом и т. д. Прежде чем продолжить, внимательно прочитайте следующие три подсказки.

Подсказки. (1) Еще раз рассмотрите пример выше. В нем говорится, что "`d`" нужно вставить в слова (`list "e" "r"`) и (`list "r" "e"`). Следующее применение является естественным кандидатом на один из примеров:

```
| (insert-everywhere/in-all-words "d"
|   (cons (list "e" "r")
|         (cons (list "r" "e")
|               '()))))
```

(2) Использовать операцию `append` из языка BSL+, которая принимает два списка и возвращает объединенный список:

```
| > (append (list "a" "b" "c") (list "d" "e"))
| (list "a" "b" "c" "d" "e")
```

Разработка таких функций, как `append`, посвящена глава 23.

(3) Для решения этого упражнения потребуется спроектировать серию функций. Неукоснительно придерживайтесь рецепта проектирования и систематически работайте со своим списком желаний. ■

Упражнение 214. Объедините `aggangements` с частично готовой программой из раздела 12.3. Убедившись, что все тесты успешно выполняются, опробуйте программу на примерах по своему выбору. ■

12.5. «Питон»

«Питон», или «Змея», – одна из самых старых компьютерных игр. В начале игры на экране появляются питон и яблоко. Питон движется к краю игровой сцены. Вы должны управлять его движением с помощью клавиш со стрелками так, чтобы он не достиг края, иначе игра завершится.

Цель игры – заставить питона съесть как можно больше яблок. Съев очередное яблоко, питон вырастает в размерах, а в игровой сцене появляется очередное яблоко. Однако с ростом питона увеличивается опасность наткнуться на самого себя. Когда питон станет достаточно длинным, он может наткнуться на собственное тело, и тогда игра тоже закончится.

На рис. 13 показана последовательность снимков экрана, иллюстрирующих процесс игры. Слева показано начальное игровое поле. Питон состоит из одного красного сегмента – головы. Он движется к яблоку, изображенному в виде зеленого кружка. На снимке экрана в центре изображена ситуация, когда питон уже съел несколько яблок. На самом правом снимке экрана питон достиг правого края игрового поля. Игра окончена; игрок набрал 11 очков.

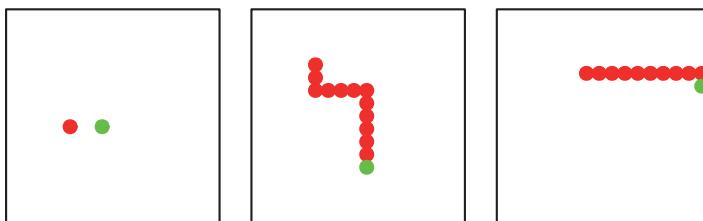


Рис. 13. Игра «Питон»

Следующие упражнения помогут вам спроектировать и реализовать игру «Питон». Как и в разделе 10.2, эти упражнения демон-

стрируют решение нетривиальной задачи методом итеративного уточнения. То есть интерактивная программа проектируется не вся сразу, а в несколько этапов, которые называются *итерациями*. Каждая итерация добавляет детали и уточняет программу, и так продолжается до тех пор, пока не будет достигнут результат, удовлетворяющий вас или вашего клиента. Если результат упражнений вас не устраивает, не останавливайтесь на достигнутом и придумывайте свои варианты.

Упражнение 215. Спроектируйте интерактивную программу, которая непрерывно перемещает односегментного питона и позволяет игроку управлять направлением его движения с помощью четырех клавиш со стрелками. Для отображения сегмента питона программа должна использовать красный кружок. За каждый тик часов питон должен перемещаться на диаметр этого кружка.

Подсказки. (1) Прочитайте еще раз раздел 3.6, чтобы вспомнить, как разрабатывать интерактивные программы. Когда будете определять функцию `worm-main`, предусмотрите возможность передачи ей аргумента, определяющего скорость хода часов. О том, как менять скорость хода часов, можно узнать в описании функции `on-tick` в документации. (2) При разработке представления данных для питона поразмышляйте о двух разных видах представлений: физического и логического. **Физическое** представление определяет фактическое **положение** питона на холсте, а **логическое** – на каком расстоянии от левого верхнего угла сцены (в сегментах) находится питон. С каким из двух представлений проще изменить внешний вид (размер сегмента, размер игрового поля) «игры»? ■

Упражнение 216. Измените программу из упражнения 215 так, чтобы она останавливалась, когда питон достигает края игрового поля. При остановке по этому условию программа должна показать финальную сцену с текстом "worm hit border" («питон достиг края») в нижнем левом углу сцены. **Подсказка.** Чтобы показать эту заключительную сцену, можно использовать предложение `stop-when` в `big-bang`. ■

Упражнение 217. Разработайте представление данных для питона с хвостом. Хвост – это, возможно пустая, последовательность «связанных» сегментов. Слово «связанных» означает здесь, что координаты каждого следующего сегмента отличаются от координат предшествующего сегмента не более чем на единицу в одном из направлений. Чтобы не усложнять задачу, считайте все сегменты – голову и хвост – равноправными.

Теперь измените программу из упражнения 215 так, чтобы она могла перемещать питона, состоящего из нескольких сегментов. Для простоты: (1) программа может отображать все сегменты питона в виде красных кружков и (2) игнорировать тот факт, что питон может достичь края игрового поля или наткнуться на самого себя. **Подсказка.** Один из способов сымитировать движение питона – добавить сегмент спереди, в направлении движения, и удалить последний. ■

Листинг 44. Случайное размещение яблок

```

; Posn -> Posn
; ???
(define (food-create p)
  (food-check-create
    p (make-posn (random MAX) (random MAX)))))

; Posn Posn -> Posn
; генеративная рекурсия
; ???
(define (food-check-create p candidate)
  (if (equal? p candidate) (food-create p) candidate))

; Posn -> Boolean
; используется только для тестирования
(define (not=-1-1? p)
  (not (and (= (posn-x p) 1) (= (posn-y p) 1))))

```

Упражнение 218. Измените программу из упражнения 217 так, чтобы она останавливалась, когда питон достигает края игрового поля или наталкивается на самого себя. В таких случаях программа должна выводить сообщение, подобное предложенному в упражнении 216, объясняющее причину прекращения игры – из-за достижения края игрового поля или из-за того, что питон наткнулся на самого себя.

Подсказки. (1) Чтобы определить момент, когда питон наталкивается на самого себя, проверьте совпадение координат головы с координатами каждого сегмента хвоста, если он продолжит движение в прежнем направлении. (2) Прочитать описание функции `member?`. ■

Упражнение 219. Добавьте в программу из упражнения 218 яблоко. В любой момент в сцене должно иметься одно яблоко. Чтобы не усложнять задачу, яблоко должно иметь такой же размер, как сегмент питона. Когда голова питона оказывается в той же точке, что и яблоко, питон должен съедать его и удлиняться на один сегмент. После этого в произвольном месте в сцене должно появляться новое яблоко.

Добавление яблок в игру требует изменения представления данных о состояниях мира. Помимо питона, состояние должно также включать представление яблока – его текущего местоположения. Изменение представления игры предполагает проектирование новых функций для обработки событий, однако эти функции могут повторно использовать функции из упражнения 218 и их тестовые примеры. Это также означает, что обработчик тактов должен не только перемещать питона, но и управлять процессом поедания и созданием нового яблока.

Ваша программа должна размещать яблоки в игровом поле случайным образом. Для этого вам придется использовать метод проектирования, с которым вы прежде не

Чтобы познакомиться с особенностями работы функции `random`, загляните в документацию или прочтайте упражнение 99.

сталкивались, – так называемая генеративная рекурсия, которая будет представлена в части V, – поэтому мы показали соответствующие функции в листинге 44. Однако прежде чем использовать их, попробуйте объяснить, как они работают, предположив, что MAX больше 1, а затем сформулируйте описание назначения.

Подсказки. (1) Один из вариантов реализовать «поедание» – переместить голову в позицию, где находилось яблоко, а в хвост добавить один сегмент на место, где раньше находилась голова. Почему этот способ проще всего реализовать в виде функции? (2) Мы сочли полезным добавить второй параметр в функцию `worm-main` для этого последнего шага, логическое значение, которое определяет, должно ли выражение `big-bang` отобразить текущее состояние мира в отдельном окне; см. документацию с описанием функции `state`, чтобы узнать, как запросить эту информацию. ■

Выполнив это последнее упражнение, вы получите законченную игру «Питон». Теперь измените функцию `worm-main` так, чтобы она возвращала текущую длину питона. Затем в меню DrRacket выберите пункт **Racket > Create Executable** (Racket > Создать исполняемый файл), чтобы получить файл, который сможет запустить любой желающий, а не только те, кто знаком с языком BSL+.

Вы также можете добавить в игру что-то свое, чтобы сделать ее действительно вашей игрой. Мы поэкспериментировали с забавными сообщениями в конце игры, с несколькими разными яблоками, с размещением дополнительных препятствий в сцене и некоторыми другими идеями. Сможете ли вы придумать что-то свое?

12.6. Простой «Тетрис»

«Тетрис» – еще одна старая-старая компьютерная игра. Разработка полноценной игры «Тетрис» требует много труда, поэтому в этом разделе мы займемся проектированием упрощенной ее версии. Если вы достаточно амбициозны, то посмотрите, как работает настоящая игра «Тетрис», и создайте полноценную версию.

В нашей упрощенной версии игра начинается с того, что из-за верхнего края сцены появляются блоки и падают вниз. Достигнув поверхности внизу, блок останавливается, и из-за верхнего края в случайном месте появляется следующий. Игрок может перемещать падающий блок с помощью клавиш со стрелками «влево» и «вправо». Как только блок достигает нижнего края сцены или поверхности какого-либо другого неподвижного блока, он останавливается. Блоки могут образовывать стопки, растущие вверх; если одна такая стопка блоков достигнет верхнего края сцены, то игра прекращается. Естественно, цель этой игры – разместить внизу как можно больше блоков. См. рис. 14, иллюстрирующий эту идею.

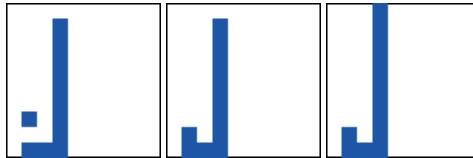


Рис. 14. Упрощенная игра «Тетрис»

Теперь, имея это описание, можно обратиться к руководству по проектированию интерактивных программ из раздела 3.6. Оно предлагает отделить постоянные свойства от переменных. Первые можно обозначить как «физические» и графические константы, а вторые – изменяющиеся данные, которые описывают любые возможные состояния игры «Тетрис». Итак, вот несколько примеров.

- Ширина и высота игрового поля и блоков фиксированы. То есть нам понадобятся следующие определения на BSL+:

```
(define WIDTH 10) ; ширина в блоках по горизонтали
(define SIZE 10) ; блоки -- это квадраты
(define SCENE-SIZE (* WIDTH SIZE))
(define BLOCK ; красные квадраты с черным кантом
  (overlay
    (square (- SIZE 1) "solid" "red")
    (square SIZE "outline" "black")))
```

Объясните эти определения, прежде чем читать дальше.

- «Ландшафт», образуемый блоками, меняется от игры к игре и с течением времени в одной и той же игре. Уточним это обстоятельство. Внешний вид блоков не меняется, но их позиции различаются.

Теперь мы оказываемся перед одной из главных задач проектирования – определение представления данных для падающих блоков и ландшафта, образованного опустившимися блоками. В случае с падающим блоком мы имеем две возможности: первая – выбрать «физическое» представление, а вторая – «логическое». Физическое представление определяет фактическое местоположение блока в сцене, а логическое – на каком расстоянии от левого верхнего угла сцены (в блоках) находится данный блок. В случае с неподвижными блоками вариантов еще больше: список физических позиций, список логических позиций, список высот столбиков и т. д.

См. упражнение 215, где представлены соответствующие проектные решения.

В этом разделе мы выбрали следующее представление данных:

```
(define-struct tetris [block landscape])
(define-struct block [x y])

; Tetris -- это структура:
;   (make-tetris Block Landscape)
; Landscape -- это одно из значений:
```

```

; -- '()
; -- (cons Block Landscape)
; Block -- это структура:
;   (make-block N N)
; интерпретация:
; (make-block x y) обозначает блок, удаленный на
; (* x SIZE) пикселей от левого края и на
; (* y SIZE) пикселей от верхнего края;
; (make-tetris b0 (list b1 b2 ...)) означает b0 -- падающий блок,
; а b1, b2 и ... -- остановившиеся блоки

```

Это то, что мы назвали логическим представлением, потому что координаты не отражают физического местоположения блоков, а только расстояния в блоках, на которых они находятся от верхнего и левого края сцены. Наш выбор подразумевает, что x всегда имеет значение в пределах между 0 и `WIDTH` (не включая его), а y – между 0 и `HEIGHT` (не включая его), но мы пока игнорируем это знание.

Упражнение 220. Всякий раз, создав сложное определение данных, подобное определению состояния игры «Тетрис», следующим шагом должно быть создание экземпляров различных коллекций данных. Вот несколько говорящих имен для примеров, которые вы можете использовать для примеров применения функций:

```

(define landscape0 ...)
(define block-dropping ...)
(define tetris0 ...)
(define tetris0-drop ...)
...
(define block landed (make-block 0 (- HEIGHT 1)))
...
(define block-on-block (make-block 0 (- HEIGHT 2)))

```

Спроектируйте программу `tetris-render`, которая превращает данный экземпляр Tetris в изображение. Используйте область взаимодействий в DrRacket для разработки выражения, которое отображает некоторые из ваших простых примеров данных. Затем сформулируйте примеры применения функций в виде модульных тестов и спроектируйте сами функции. ■

Упражнение 221. Спроектируйте интерактивную программу `tetris-main`, которая отображает блоки, падающие вертикально вниз от верхнего края сцены и приземляющиеся на нижний край или на блоки, которые уже лежат. `tetris-main` должна принимать аргумент, определяющий скорость хода часов. О том, как менять скорость хода часов, можно узнать в описании функции `op-tick` в документации.

Чтобы определить, приземлился ли блок, можно проверить достижение им нижнего края сцены или верхнего края одного из блоков в списке неподвижных блоков. **Подсказка.** Прочитайте описание функции `member?`.

Когда блок приземляется, программа должна немедленно создать другой блок, опускающийся правее текущего. Если текущий блок уже находится в крайнем правом положении, следующий блок должен на-

чать спуск в левом крайнем столбце. Как вариант определите функцию `block-generate`, которая случайным образом выбирает столбец, отличный от текущего; см. упражнение 219. ■

Упражнение 222. Измените программу из упражнения 221 так, чтобы игрок мог перемещать опускающийся блок влево и вправо. Каждый раз, когда игрок нажимает клавишу со стрелкой «влево», опускающийся блок должен сдвигаться на один столбец влево, если он не находится в крайнем левом столбце или слева от него уже находится стопка неподвижных блоков. Точно так же каждый раз, когда игрок нажимает клавишу со стрелкой «вправо», опускающийся блок должен сдвигаться на один столбец вправо, если это возможно. ■

Упражнение 223. Добавьте в программу из упражнения 222 предложение `stop-when`. Игра должна заканчиваться, когда любой из столбцов блоков становится настолько высоким, что касается верхнего края сцены. ■

Решив упражнение 223, вы получите простую игру «Тетрис». Вы можете немного отполировать ее, прежде чем показывать своим друзьям. Например, в последней сцене может отображаться текст, сообщающий, сколько блоков игрок смог уложить. Ту же информацию, впрочем, можно показывать в любой сцене. Выбор за вами.

12.7. Полная игра «Космические захватчики»

В главе 6 мы предприняли попытку спроектировать игру «Космические захватчики», но она поддерживала слишком мало действий; игрок мог лишь перемещать наземный танк влево и вправо. В разделе 9.5 мы добавили возможность сделать множество выстрелов. В данном разделе представлены упражнения, которые помогут вам закончить эту игру.

Согласно сценарию, на Землю пытается приземлиться НЛО. Задача игрока – не допустить этого. Для этого в игре есть танк, который может сделать произвольное количество выстрелов. Когда одна из выпущенных ракет оказывается недалеко от центра НЛО, игра заканчивается, и игрок побеждает. Если НЛО достигает поверхности Земли, игрок проигрывает.

Упражнение 224. Используйте знания и навыки, приобретенные в двух предыдущих разделах, и тщательно спроектируйте расширение игры, добавляя новые возможности по одной. Неукоснительно следуйте рецепту проектирования и руководствуйтесь инструкциями по вспомогательным функциям. Если игра понравилась вам, то добавьте другие возможности: покажите бегущую строку; оснастите НЛО ракетами, способными уничтожить танк; создайте целый флот НЛО; и прежде всего используйте свое воображение. ■

Если вам не нравятся НЛО и танки, стреляющие друг в друга, используйте те же идеи для создания похожей невоенной игры, в которой требуется, например, уничтожать падающие метеориты.

Упражнение 225.

Спроектируйте игру о тушении пожара. Действие игры разворачивается в западных штатах, где в обширных лесах бушуют пожары. Она должна имитировать действия по тушению пожара с воздуха. В частности, игрок выступает в роли пилота самолета, который сбрасывает воду на очаги пожара на земле. Игрок управляет перемещением самолета по горизонтали и может сбрасывать порции воды.

Программа должна зажигать пожары в случайных местах на земле. Вы можете ограничить количество пожаров, спроектировав для этого функцию, которая определяет количество имеющихся в данный момент пожаров или учитывает какие-то другие факторы. Цель игры – потушить все пожары за ограниченное время. **Подсказка.** Используйте итеративный подход к проектированию игры, как было показано выше в этой главе. ■

12.8. Конечные автоматы

Конечные автоматы (Finite State Machines, FSM) и регулярные выражения – широко распространенные элементы программирования. Как рассказывается в разделе 4.7, конечные автоматы – это один из способов представления интерактивных программ. В упражнении 109 было показано, как проектировать интерактивные программы, реализующие конечный автомат, и как проверить, нажал ли игрок определенную последовательность клавиш.

Также вы можете вспомнить, что конечный автомат эквивалентен регулярному выражению. Специалисты в области информатики склонны говорить, что конечный автомат принимает нажатия клавиш, соответствующие определенному регулярному выражению, например такому:

| a (b|c)* d

из упражнения 109.

Если вам требуется, чтобы программа распознавала какой-то другой шаблон, скажем

| a (b|c)* a

то вы должны просто изменить существующую программу соответствующим образом. Эти две программы будут похожи друг на друга, и если вы повторите это упражнение для нескольких регулярных выражений, то получите целую кучу похожих программ.

Естественно, возникает идея – найти общее решение, то есть спроектировать интерактивную программу, которая использует **представление данных в виде конечного автомата** и распознает нажатие соответствующих последовательностей клавиш. В этом разделе представлен проект именно такой интерактивной программы, хотя

и значительно упрощенный. В частности, конечные автоматы не имеют начального или конечного состояния, и сопоставление игнорирует фактические нажатия клавиш; вместо этого переход из одного состояния в другое происходит всякий раз, когда нажимается любая клавиша. Кроме того, каждое состояние определяет свой цвет. При таком подходе программа интерпретации конечных автоматов может просто отобразить текущее состояние в виде цвета.

ЗАМЕЧАНИЯ ПО ВЫБОРУ ДИЗАЙНА. Вот еще одна попытка обобщения:

Задача. Спроектируйте программу, которая интерпретирует данный конечный автомат с учетом определенного списка событий клавиатуры. То есть программа использует представление данных конечного автомата и строки. Ее результатом должно быть значение `#true`, если строка соответствует регулярному выражению конечного автомата, и `#false` в противном случае.

Однако, как оказывается, эту программу **нельзя спроектировать**, руководствуясь принципами, представленными в первых двух частях. Решение этой задачи мы отложим до главы 29; см. упражнение 476. **КОНЕЦ.**

Листинг 45. Обобщенное представление и интерпретация конечных автоматов

```
; FSM -- одно из значений:
; -- '()
; -- (cons Transition FSM)

(define-struct transition [current next])
; Transition -- это структура:
;   (make-transition FSM-State FSM-State)

; FSM-State -- это Color (цвет).

; интерпретация: Конечный автомат (FSM) представляет переходы
; между состояниями в ответ на нажатия клавиш
```

Упрощенная постановка задачи диктует ряд моментов, включая необходимость определения данных для представления конечных автоматов, природы их состояний и их внешнего вида в виде изображения. Вся эта информация представлена в листинге 45. Он начинается с определения данных FSM для конечных автоматов. Как видите, конечный автомат (экземпляр FSM) – это просто список переходов (экземпляров Transition). Мы должны использовать список, чтобы наша интерактивная программа могла работать с любыми конечными автоматами, а это означает конечное и вместе с тем произвольно большое количество состояний. Каждый переход Transition объединяет два состояния в структуру: текущее состояние `current` и следующее состояние `next`, то есть состояние, в которое автомат перейдет,

когда игрок нажмет клавишу. В последней части определения данных говорится, что состояние – это просто строка с названием цвета.

Упражнение 226. Спроектируйте предикат `state=?`, сравнивающий состояния. ■

Это сложное определение, поэтому последуем рецепту проектирования и создадим пример:

```
(define fsm-traffic
  (list (make-transition "red" "green")
        (make-transition "green" "yellow")
        (make-transition "yellow" "red")))
```

Вы, наверное, догадались, что эта таблица переходов описывает светофор. Первый переход сообщает, что светофор переключается с красного ("red") на зеленый ("green"), второй представляет переключение с зеленого ("green") на желтый ("yellow"), и последний – с желтого ("yellow") на красный ("red").

Упражнение 227. Черно-белый автомат – это конечный автомат, который переключается с черного на белый и обратно по каждому нажатию клавиши. Сформулируйте представление данных для черно-белого автомата. ■

Решением этой задачи является интерактивная программа:

```
; FSM -> ???
; сопоставляет нажатые клавиши с заданным конечным автоматом FSM
(define (simulate an-fsm)
  (big-bang ...
    [to-draw ...]
    [on-key ...]))
```

Предполагается, что эта программа получает конечный автомат, но мы понятия не имеем, что должна вернуть программа. Мы дали программе имя `simulate`, потому что она отвечает на нажатия клавиш согласно заданному конечному автомату.

Давайте последуем рецепту проектирования интерактивных программ, чтобы увидеть, как далеко это нас заведет. Рецепт требует отдельить вещи «реального мира», которые могут меняться, от тех, которые остаются неизменными. Функция `simulate` принимает экземпляр конечного автомата `FSM`, но мы знаем, что этот конечный автомат не меняется. Единственное, что меняется, – это текущее состояние автомата.

Константа `empty-image` определяет «невидимое» изображение. Это хорошее значение по умолчанию для

Из этого анализа вытекает следующее определение данных:

| ; SimulationState.v1 -- это FSM-State.

Согласно рецепту проектирования интерактивных программ, это определение данных завершает главную функцию:

```
(define (simulate.v1 fsm0)
  (big-bang initial-state
    [to-draw render-state.v1]
    [on-key find-next-state.v1]))
```

и предполагает добавление двух записей в список желаний:

```
; SimulationState.v1 -> Image
; преобразует состояние мира в изображение
(define (render-state.v1 s)
  empty-image)

; SimulationState.v1 KeyEvent -> SimulationState.v1
; определяет следующее состояние по ke и cs
(define (find-next-state.v1 cs ke)
  cs)
```

Эти записи вызывают два вопроса. Первый: определение `SimulationState.v1`. В настоящее время выбранное состояние `initial-state` отмечено серым цветом, чтобы предупредить о проблеме. Второй: следующая запись в списке желаний должна вызвать некоторый ужас:

Как `find-next-state` должна определять следующее состояние, если все, что она имеет, – это текущее состояние и нажатая клавиша?

Этот вопрос звучит особенно актуально, потому что, согласно упрощенной постановке задачи, точное значение нажатой клавиши не имеет значения; автомат переходит в следующее состояние независимо от того, какая клавиша нажата.

Эта вторая проблема раскрывает **фундаментальное ограничение BSL+**. Чтобы уяснить суть этого ограничения, начнем издалека. Фактически необходимо, чтобы функция `find-next-state` получала не только текущее состояние, но и конечный автомат, используя который, она могла бы выполнить поиск в списке переходов и выбрать следующее состояние. Иначе говоря, состояние мира должно включать как текущее состояние конечного автомата, так и сам конечный автомат:

```
(define-struct fs [fsm current])
; SimulationState.v2 -- это структура:
; (make-fs FSM FSM-State)
```

Алонзо Черч (*Alonzo Church*) и Аллан Тьюринг (*Alan Turing*), первые ученые в области информатики, доказали в 1930-х годах, что все языки программирования могут применять определенные функции к числам. Следовательно, как они утверждали, все языки программирования равны. Первый автор этой книги не согласен с этим утверждением. Он различает языки в зависимости от того, как они позволяют программистам выражать решения.

Согласно рецепту проектирования интерактивных программ, это изменение также означает, что обработчик события клавиатуры должен возвращать данную комбинацию:

```
; SimulationState.v2 -> Image
; преобразует состояние мира в изображение
(define (render-state.v2 s)
  empty-image)

; SimulationState.v2 KeyEvent -> SimulationState.v2
; определяет следующее состояние по ke и cs
(define (find-next-state.v2 cs ke)
  cs)
```

Наконец, главная функция теперь должна принимать два аргумента: конечный автомат и его начальное состояние. В конце концов, разные конечные автоматы, которые могут передаваться функции `simulate`, могут иметь самые разные состояния; мы не можем быть уверены, что все они будут иметь одинаковое начальное состояние. Вот обновленный заголовок функции:

```
| ; FSM FSM-State -> SimulationState.v2
| ; сопоставляет нажатые клавиши с заданным конечным автоматом FSM
| (define (simulate.v2 an-fsm s0)
|   (big-bang (make-fs an-fsm s0)
|             [to-draw state-as-colored-square]
|             [on-key find-next-state]))
```

Теперь вернемся к примеру с конечным автоматом, имитирующим работу светофора. В этом случае функции `simulate` следует передать сам автомат и начальное состояние "red":

*Инженеры назвали бы
состояние "red"
безопасным состоянием.* Стоп! Как вы думаете, почему для светофора лучше использовать начальное состояние "red"?

ПРИМЕЧАНИЕ О ВЫРАЗИТЕЛЬНОСТИ. Теперь мы можем объяснить ограничение языка BSL. Несмотря на то что данный конечный автомат не изменяется в ходе выполнения программы, его описание должно быть частью состояния мира. В идеале программа должна бы выразить описание конечного автомата как константу, но вместо этого она вынуждена рассматривать его как часть постоянно меняющегося состояния. Читатель программы не сможет вывести этот факт только на основании первого фрагмента `big-bang`.

Следующая часть книги решает эту загадку введением нового языка программирования и особой лингвистической конструкции: ISL и локальных определений. Подробности см. в разделе 16.3. **КОНЕЦ**.

Теперь мы можем вернуться к списку желаний и по очереди реализовать записи из него. Сначала спроектируем функцию `state-as-colored-square`. Она настолько проста, что мы не будем отвлекаться на подробные объяснения и сразу дадим полное определение:

```
| ; SimulationState.v2 -> Image
| ; отображает текущее состояние мира в виде цветного квадрата

| (check-expect (state-as-colored-square
|                  (make-fs fsm-traffic "red"))
|                  (square 100 "solid" "red"))

| (define (state-as-colored-square an-fsm)
|   (square 100 "solid" (fs-current an-fsm)))
```

Проектирование обработчика `key-event`, напротив, требует некоторых пояснений. Вспомним, как выглядит его заголовок:

```
; SimulationState.v2 KeyEvent -> SimulationState.v2
; определяет следующее состояние по ke и cs
(define (find-next-state an-fsm current)
  an-fsm)
```

Согласно рецепту проектирования, обработчик должен принимать состояние мира и событие клавиатуры KeyEvent и возвращать следующее состояние мира. Такое описание простыми словами также помогает при оформлении примеров. Вот первые два:

```
(check-expect
  (find-next-state (make-fs fsm-traffic "red") "n")
  (make-fs fsm-traffic "green"))
(check-expect
  (find-next-state (make-fs fsm-traffic "red") "a")
  (make-fs fsm-traffic "green"))
```

В примерах говорится, что если текущее состояние мира объединяет автомат fsm-traffic и его состояние "red", то результатом будет объединение того же автомата с его состоянием "green", независимо от того, какую клавишу нажал игрок – "n" или "a". Еще один пример:

```
(check-expect
  (find-next-state (make-fs fsm-traffic "green") "q")
  (make-fs fsm-traffic "yellow"))
```

Попробуйте сами объяснить этот пример, прежде чем продолжить. Сможете придумать еще один пример?

Поскольку функция принимает структуру, используем макет обработки структур:

```
(define (find-next-state an-fsm ke)
  (... (fs-fsm an-fsm) .. (fs-current an-fsm) ...))
```

Кроме того, так как желаемым результатом является SimulationState.v2, мы можем уточнить макет, добавив соответствующий конструктор:

```
(define (find-next-state an-fsm ke)
  (make-fs
    ... (fs-fsm an-fsm) ... (fs-current an-fsm) ...))
```

Как показывают примеры, извлеченный конечный автомат становится первым компонентом новой структуры SimulationState.v2, и функция должна просто вычислить следующее состояние на основе текущего и списка переходов в данном конечном автомате. Поскольку список переходов может иметь произвольную длину, добавим в список желаний еще одну запись – функцию find, которая должна просмотреть список в поисках перехода Transition, значение current которого совпадает с текущим состоянием (fs-current an-fsm):

```
(define (find-next-state an-fsm ke)
  (make-fs
```

```
(fs-fsm an-fsm)
(find (fs-fsm an-fsm) (fs-current an-fsm))))
```

Вот формулировка нового желания:

```
; FSM FSM-State -> FSM-State
; находит элемент в списке transitions,
; соответствующий текущему состоянию current,
; и извлекает поле next
(check-expect (find fsm-traffic "red") "green")
(check-expect (find fsm-traffic "green") "yellow")
(check-error (find fsm-traffic "black")
             "not found: black")
(define (find transitions current)
  current)
```

Эти примеры заимствованы из примеров для `find-next-state`.

Стоп! Придумайте несколько дополнительных примеров, а затем выполните упражнения.

Упражнение 228. Завершите проектирование `find`.

После тестирования вспомогательных функций поэкспериментируйте с `simulate` и попробуйте передать ей `fsm-traffic` и черно-белый автомат из упражнения 227. ■

Наша программа моделирования намеренно упрощена. В частности, ее нельзя использовать для представления конечных автоматов, которые переходят из одного состояния в другое, в зависимости от нажатой клавиши. Однако, используя систематический подход к проектированию, вы сможете дополнить программу такой возможностью.

Упражнение 229. Вот исправленное определение данных для `Transition`:

```
(define-struct ktransition [current key next])
; Transition.v2 -- это структура:
;   (make-ktransition FSM-State KeyEvent FSM-State)
```

Перепишите определение конечного автомата `FSM` из упражнения 109, используя списки `Transition.v2`, игнорируя ошибки и конечные состояния.

Измените определение `simulate` так, чтобы теперь она обрабатывала нажатия клавиш соответствующим образом. Следуйте рецепту проектирования, начав с адаптации примеров данных.

Используйте исправленную программу для моделирования с конечным автоматом из упражнения 109 и следующей последовательностью нажатых клавиш: "a", "b", "b", "c" и "d". ■

Конечные автоматы бывают с начальным и конечным состояниями. Когда программа, которая «запускает» конечный автомат, достигает конечного состояния, она должна остановиться. В заключительном упражнении мы еще раз изменим представление данных для конечных автоматов, чтобы реализовать эти идеи.

Упражнение 230. Рассмотрим следующее представление данных для конечных автоматов:

```
(define-struct fsm [initial transitions final])
(define-struct transition [current key next])
; FSM.v2 -- это структура:
;   (make-fsm FSM-State LOT FSM-State)
; LOT -- это одно из значений:
;   '()
;   (cons Transition.v3 LOT)
; Transition.v3 -- это структура:
;   (make-transition FSM-State KeyEvent FSM-State)
```

Представьте конечный автомат из упражнения 109 в этом контексте.

Спроектируйте функцию `fsm-simulate`, которая принимает экземпляр `FSM.v2` и запускает его, когда игрок нажимает клавиши. Если по результатам нажатий клавиш `FSM.v2` достигает конечного состояния, то `fsm-simulate` должна останавливаться. **Подсказка.** Для проверки текущего состояния функция должна использовать поле `initial` данной структуры `fsm`. ■

ЗАМЕЧАНИЕ ОБ ИТЕРАТИВНОМ УТОЧНЕНИИ. В последних двух проектах было введено понятие «проектирование путем итеративного уточнения». Основная идея состоит в том, что первая версия программы реализует только часть желаемого поведения, следующая – чуть больше и т. д. В конечном итоге вы получаете программу, которая реализует все желаемые возможности или, по крайней мере, достаточное их количество, чтобы удовлетворить клиента. Дополнительную информацию по этой теме вы найдете в главе 20. **КОНЕЦ.**

13. Итоги

Вторая часть книги была посвящена проектированию программ, обрабатывающих произвольно большие данные. Нетрудно понять, что программное обеспечение особенно полезно, если оно способно обрабатывать информацию без заранее определенных ограничений по размеру, а это означает, что освоение приемов обработки «произвольно больших данных» является важным шагом на вашем пути к тому, чтобы стать настоящим программистом. Вот три основных урока, которые вы должны вынести:

1. В этой части мы **уточнили рецепт проектирования**, добавив в него инструкции по обработке определений данных, ссылающихся на самих себя и с перекрестными ссылками. В первом случае требуется использовать рекурсивные функции, а во втором – вспомогательные функции.
2. Сложные задачи должны **делиться** на более мелкие и простые подзадачи. При выполнении такого деления вам потребуются две составляющие: функции, решающие подзадачи, и определения данных, объединяющие эти решения в одно. Чтобы гарантировать успех объединения после того, как вы потратите время на создание отдельных решений, вам необходимо сформулировать свои «желания» вместе с необходимыми определениями данных.
Проектирование методом декомпозиции-композиции (деления и объединения) особенно полезно, когда в постановке задачи явно или неявно упоминаются вспомогательные задачи, когда на этапе программирования функции обнаруживается необходимость обхода (другого) произвольно большого фрагмента данных и, что может показаться удивительным, когда обобщенная задача имеет более простое решение, чем конкретная, описанная в постановке задачи.
3. **Прагматика имеет значение.** Чтобы разрабатывать программы `big-bang`, вы должны понимать их цели и задачи. Например, если перед вами стоит задача спроектировать программу, решающую математические уравнения, то вы должны узнать, какие математические операции предлагает выбранный язык и его библиотеки.

В этой части книги в основном рассматриваются списки, которые являются собой хороший пример произвольно больших данных, особенно если учесть их широкое распространение в таких языках, как Haskell, Lisp, ML, Racket и Scheme, однако изложенные здесь идеи применимы к любым видам таких данных: файлам, папкам с файлами, базам данных и т. д.

В части IV мы продолжим исследование «больших» структурированных данных и посмотрим, как распространить рецепт проектирования на самые сложные типы данных. Но прежде вам предстоит прочитать часть III, которая избавит от беспокойства, которое у вас должно возникнуть на данном этапе, а именно из-за того, что работа программиста состоит в создании программ одного и того же типа.

Интермеццо 2. Quote, unquote

Обязательно выберите язык *BSL+* или старше. Списки играют важную основополагающую роль в нашей книге и в Racket. При разработке программ важно понимать, как конструируются списки, исходя из основных принципов. Однако для постоянной работы со списками необходима компактная форма их записи, подобная той, что дает функция *list*, представленная в разделе 11.1.

С конца 1950-х годов языки в стиле Lisp предлагают еще более мощную пару инструментов для создания списков: цитирование (*quote*) и антицитирование (*unquote*). Многие современные языки программирования теперь поддерживают их, а язык для разработки веб-приложений PHP привнес эту идею в коммерческий мир.

Это интермеццо рассказывает о механизме цитирования, а также знакомит с *символами*, формой данных, которая тесно связана с цитированием. Данное введение носит неформальный характер и использует упрощенные примеры, а основная мощь этой идеи на почти реалистичных примерах будет показана далее в книге. Вернитесь к этому интермеццо, если у вас возникнут проблемы с каким-либо из примеров.

Цитирование

Цитирование (*quote*) – это механизм, упрощающий запись длинных списков. Проще говоря, цитирование позволяет еще больше упростить форму записи списка, по сравнению с функцией *list*.

Технически *quote* – это ключевое слово для составного предложения и используется так: (*quote* (1 2 3)). DrRacket интерпретирует это выражение как (*list* 1 2 3). У многих из вас может возникнуть вопрос: почему мы утверждаем, что *quote* обеспечивает возможность более краткой записи списков, ведь на самом деле выражение с *quote* выглядит сложнее? Дело в том, что ключевое слово *quote* можно заменить его псевдонимом ' (одиночной кавычкой). Вот несколько коротких примеров:

```
> '(1 2 3)
(list 1 2 3)
> ("a" "b" "c")
(list "a" "b" "c")
> '#true "hello world" 42
(list #true "hello world" 42)
```

Как видите, оператор ' создает обещанные списки. Если вы забыли, что означает (*list* 1 2 3), то прочитайте еще раз раздел 11.1, где объясняется, что этот (*list* 1 2 3) является краткой формой записи для (*cons* 1 (*cons* 2 (*cons* 3 '()))).

Пока что *quote* не выглядит впечатляющим улучшением по сравнению с *list*, но взгляните на следующий пример:

```
> '(("a" 1)
  ("b" 2)
  ("d" 4))
(list (list "a" 1) (list "b" 2) (list "d" 4))
```

С помощью ' можно конструировать даже списки с несколькими уровнями вложенности.

Чтобы понять, как работает quote, представьте то ключевое слово как функцию, которая выполняет обход указанных элементов. Когда оператор ' встречает простой экземпляр элементарных данных – число, строку, логическое значение или изображение, – он исчезает (ничего не делает). Но когда он находится перед открывающей круглой скобкой (, то добавляет функцию list справа от этой скобки и вставляет ' перед всеми элементами списка, находящимися между (и). Например:

'(1 2 3) – это краткая форма записи для (list '1 '2 '3)

Как уже говорилось, ' перед числами исчезает, поэтому далее это выражение интерпретируется просто. А вот пример создания вложенных списков:

'(("a" 1) 3) – это краткая форма записи для (list '("a" 1) '3)

Продолжим этот пример и развернем первый оператор ':

(list '("a" 1) '3) – это краткая форма записи для (list (list '"a" '1) 3)

Закончите этот пример самостоятельно.

Упражнение 231. Замените quote функцией list в следующих выражениях:

- '(1 "a" 2 #false 3 "c");
- '();
- а также в такой табличной форме:

```
'(("alan" 1000)
  ("barb" 2000)
  ("carl" 1500))
```

Затем замените list операторами cons. ■

Квазицитирование и антицитирование

Предыдущий раздел должен был убедить вас в преимуществах ' и quote. Возможно, вас удивит, почему в этой книге механизм цитирования представлен только сейчас, а не с самого начала, ведь он значительно упрощает формулировку тестовых примеров со списками, а также исследование больших наборов данных. Но все хорошее сопровождается сюрпризами, это относится и к quote.

Начиная изучать вопросы проектирования программ, новички часто по ошибке думают, что списки – это значения, конструируемые

с помощью `quote` или даже `list`. Конструирование списков с операторами `cons` намного яснее демонстрирует пошаговое создание программ, чем сокращенные формы записи, такие как `quote`, которые скрывают структуру списков. Поэтому вспоминайте о `cons` всякий раз, застопорившись при проектировании функций, обрабатывающих списки.

Итак, перейдем к сюрпризам, кроющимся в `quote`. Представьте, что мы ввели в области определений следующее выражение:

```
| (define x 42)
```

Теперь представьте, что мы запустили эту программу и ввели в области взаимодействий выражение:

```
| '(40 41 x 43 44)
```

Каким будет результат? Стоп! Попробуйте выполнить вычисления в уме, применив правила интерпретации `'`, с которыми вы познакомились выше.

Вот результат этого эксперимента:

```
| > '(40 41 x 43 44)
| (list 40 41 'x 43 44)
```

Важно запомнить, что DrRacket отображает значения. Все элементы в списке – это значения, включая `'x`. Это значение, которое вы прежде не видели, является *символом (Symbol)*. Для нас символ выглядит как имя переменной, за исключением того, что он начинается с `'` и **является значением**. Переменные только обозначают значения; они не являются значениями сами по себе. Символы подобны строкам; они являются отличным способом представления символьской информации в виде данных. Как это делается, мы покажем в части IV, а пока просто будем воспринимать символы как еще одну форму данных.

Чтобы понять суть символов, рассмотрим второй пример:

```
| '(1 (+ 1 1) 3)
```

Вы могли бы подумать, что это выражение сконструирует `(list 1 2 3)`. Но, воспользовавшись правилами интерпретации `'`, легко обнаружить, что

`'(1 (+ 1 1) 3)` – это краткая форма записи для `(list '1 '+ 1 1) '3)`

Оператор `'` перед вторым элементом в этом списке не исчезает, а создает вложенный список, то есть весь пример выглядит так:

```
| (list 1 (list '+ 1 1) 3)
```

Это означает, что `+` является таким же символом, как и `'x`. То есть как `'x` не имеет отношения к переменной `x`, так же и `+` не имеет никакого отношения к функции `+` в BSL+. И снова вы должны представить, что `+` может служить элегантным представлением данных функции

+ так же, как '(+ 1 1) может служить представлением данных (+ 1 1). Более детально мы исследуем эту идею в части IV.

Иногда нежелательно создавать вложенные списки, например когда требуется подставить истинное значение выражения, включенного в список, сформированный с помощью quote. В таких случаях можно использовать ключевое слово quasiquote, которое, как и quote, определяет составное выражение: (quasiquote (1 2 3)). Как и для quote, для quasiquote имеется сокращенное обозначение – оператор `.

На первый взгляд оператор ` действует так же, как ', – он создает списки:

```
> `(1 2 3)
(list 1 2 3)
> `("a" "b" "c")
(list "a" "b" "c")
> `(#true "hello world" 42)
(list #true "hello world" 42)
```

Самая замечательная особенность ` состоит в том, что его можно использовать для отмены цитирования – *антицитирования*, то есть внутри списка, созданного с помощью quasiquote, можно потребовать вернуться к нормальной интерпретации кода. Проиллюстрируем эту идею на примерах, приведенных выше:

```
> `(40 41 ,x 43 44)
(list 40 41 42 43 44)
> `(1 ,(+ 1 1) 3)
(list 1 2 3)
```

Первое взаимодействие предполагает, что область определений содержит (define x 42). Лучший способ понять суть этого синтаксиса – подставить фактические ключевые слова вместо их сокращений ` и ,:

```
(quasiquote (40 41 (unquote x) 43 44))
(quasiquote (1 (unquote (+ 1 1)) 3))
```

Ключевые слова quasiquote и unquote интерпретируются точно так же, как quote, но с одним дополнительным правилом. Когда оператор ` стоит перед открывающей круглой скобкой, его действие распространяется на все элементы до соответствующей закрывающей круглой скобки. Если ` оказывается перед экземпляром элементарных данных, то он исчезает. Если ` стоит перед именем переменной, то вы получаете символ. И новое правило: если сразу за ` следует unquote, то оба символа исчезают:

`(1 ,(+ 1 1) 3) – это краткая форма записи для (list `1 ,(+ 1 1) `3),
а

(list `1 ,(+ 1 1) `3) – это краткая форма записи для (list 1 (+ 1 1) 3).

И в результате получается (list 1 2 3), как было показано выше.

От этой точки рукой подать до создания веб-страниц. Да, вы правильно прочитали – веб-страниц! Конечно, веб-страницы пишутся на языках разметки HTML и CSS. Но уже мало кто пишет код HTML

непосредственно; вместо этого создаются программы, генерирующие веб-страницы. И вы тоже можете писать такие функции на BSL+. В листинге 46 показан упрощенный пример. Как видите, эта функция принимает две строки и создает глубоко вложенный список – представление данных веб-страницы.

Внимательно приглядевшись, можно заметить, что параметр `title` дважды появляется в теле функции: во внутреннем списке с меткой `'head` и внутреннем списке с меткой `'body`. Другой параметр появляется только один раз. Вложенный список представляет макет страницы, а параметры – это заполнители в шаблоне, которые нужно заменить фактическими значениями. Как вы понимаете, такой стиль создания веб-страниц особенно удобен, когда требуется создать много похожих страниц для сайта.

Листинг 46. Упрощенный генератор кода HTML

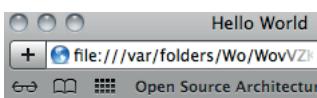
```
| ; String String -> ... многоуровневый список ...
| ; генерирует веб-страницу с заданными значениями для тегов author и title
| (define (my-first-web-page author title)
|   `(html
|     (head
|       (title ,title)
|       (meta ((http-equiv "content-type"
|               (content "text-html")))))
|     (body
|       (h1 ,title)
|       (p "I, " ,author ", made this page."))))
```

Чтобы понять, как работает эта функция, давайте поэксперименируем с ней в области взаимодействий в DrRacket. После знакомства с `quasiquote` и `unquote` вы наверняка сможете предсказать, какой результат вернет следующее выражение:

```
| (my-first-web-page "Matthias" "Hello World")
```

Вы можете воспользоваться командой `show-in-browser` из библиотеки `web-io.rkt`, чтобы отобразить результат в веб-браузере.

И снова DrRacket действует настолько быстро, что проще показать вам результат: посмотрите на правый столбец в табл. 20 – в нем показан эквивалентный код HTML. Если вы откроете эту веб-страницу в браузере, то увидите что-то вроде этого:



Hello World

I, Matthias, made this page.

Обратите внимание, что текст "Hello World" тоже появляется дважды: в заголовке окна веб-браузера (что обусловлено особенностями тега `<title>`) и тексте веб-страницы.

Если бы сейчас был 1993 год, то вы могли бы открыть свою интернет-компанию и продавать эту функцию, которую пользователи могли бы вызывать для создания своих домашних страниц. Увы, в наши дни это всего лишь упражнение.

Таблица 20. Представление данных с использованием вложенных списков

Представление с вложенными списками	HTML-код веб-страницы
<pre>'(html (head (title "Hello World")) (meta ((http-equiv "content-type" (content "text-html")))) (body (h1 "Hello World") (p "I, "Matthias" ", made this page.")))</pre>	<pre><html> <head> <title> Hello World </title> <meta http-equiv="content-type" content="text-html" /> </head> <body> <h1> Hello World </h1> <p> I, Matthias, made this page. </p> </body> </html></pre>

Упражнение 232. Удалите `quasiquote` и `unquote` из следующих выражений и замените их функцией `list` так, чтобы суть выражений не изменилась:

- `(1 "a" 2 #false 3 "c");
- следующую табличную форму:

<pre>`(("alan" ,(* 2 500)) ("barb" 2000) ,(string-append "carl" " , the great") 1500) ("dawn" 2300))</pre>
--

- веб-страницу:

<pre>`(html (head (title ,title)) (body (h1 ,title) (p "A second web page"))),</pre>
--

где `(define title "ratings")`.

Также запишите вложенные списки, генерируемые выражениями. ■

Объединение с антицитированием

Когда при интерпретации `quasiquote` встречается `unquote`, эти два ключевых слова отменяют действие друг друга:

```
|`('tr           - это краткая форма записи для      (list 'tr
  ,(make-row
    '(3 4 5)))                                     (make-row
                                                       (list 3 4 5)))
```

То есть результат, который вернет `make-row`, станет вторым элементом списка. В частности, если `make-row` создаст список, то этот список станет вторым элементом внешнего списка. Если предположить, что `make-row` преобразует заданный список чисел в список строк, то в результате получится:

```
| (list 'tr (list "3" "4" "5"))
```

Иногда, однако, может потребоваться объединить такой вложенный список с внешним, чтобы, если взять наш пример, получилось следующее:

```
| (list 'tr "3" "4" "5")
```

Одно из возможных решений этой небольшой проблемы – вернуться к `cons`. То есть создать выражение, в котором смешиваются `cons` с `quote`, `quasiquote` и `unquotec`. В конце концов, все эти операторы – лишь сокращенные формы записи для `cons`. Вот как выглядит такое решение для нашего примера:

```
| (cons 'tr (make-row '(3 4 5)))
```

Убедитесь, что это выражение в точности соответствует `(list 'tr "3" "4" "5")`.

Поскольку на практике такая ситуация встречается довольно часто, в механизм создания списков в BSL+ был добавлен еще один оператор: `,@` – сокращенная форма ключевого слова `unquote-splicing`. Этот оператор позволяет легко и просто объединить вложенный список с вмещающим. Например, выражение

```
| `('tr ,@(make-row '(3 4 5)))
```

транслируется в

```
| (cons 'tr (make-row '(3 4 5))),
```

то есть в то, что нам нужно.

Теперь рассмотрим проблему создания HTML-таблицы в нашем представлении вложенного списка. Вот таблица с двумя строками по четыре ячейки в каждой:

```
| '(table ((border "1"))
  |   (tr (td "1") (td "2") (td "3") (td "4"))
  |   (tr (td "2.8") (td "-1.1") (td "3.4") (td "1.3")))
```

Первый вложенный список требует нарисовать тонкую рамку вокруг каждой ячейки в таблице; два других вложенных списка представляют строки.

На практике часто требуется создавать такие таблицы со строками произвольной ширины и произвольным количеством строк. Но сейчас мы имеем дело с первой проблемой, для решения которой требуется функция, преобразующая списки чисел в строки HTML-таблиц:

```
; List-of-numbers -> ... вложенный список ...
; создает строку для HTML-таблицы из l
(define (make-row l)
  (cond
    [(empty? l) '()]
    [else (cons (make-cell (first l))
                 (make-row (rest l))))]))

; Number -> ... вложенный список ...
; создает из числа ячейку для HTML-таблицы
(define (make-cell n)
  `(td ,(number->string n)))
```

Вместо добавления примеров исследуем поведение этих функций в области взаимодействий в DrRacket:

```
> (make-cell 2)
(list 'td "2")
> (make-row '(1 2))
(list (list 'td "1") (list 'td "2"))
```

Эти взаимодействия демонстрируют, как создаются списки, представляющие ячейку и строку.

Чтобы превратить такие списки строк в фактические строки HTML-таблицы, нужно объединить их в список, который начинается с 'tr:

```
; List-of-numbers List-of-numbers -> ... вложенный список ...
; создает HTML-таблицу из двух списков чисел
(define (make-table row1 row2)
  '(table ((border "1"))
          (tr ,@(make-row row1))
          (tr ,@(make-row row2))))
```

Эта функция принимает два списка чисел и создает HTML-таблицу. С помощью make-row она преобразует списки чисел в списки представлений ячеек, а с помощью ,@ объединяет эти списки в таблицу:

```
> (make-table '(1 2 3 4 5) '(3.5 2.8 -1.1 3.4 1.3))
(list 'table (list (list 'border "1")) '....)
```

Такое применение make-table раскрывает еще одну причину, по которой люди пишут программы для создания веб-страниц, а не создают их вручную.

Упражнение 233. Разработайте альтернативы следующим выражениям, которые возвращают те же результаты, но исключительно с использованием list:

Многоточия
не являются
частью вывода.

- `(0 ,@')(1 2 3) 4;
- следующую табличную форму:

```
| `(("alan" ,( * 2 500))
|   ("barb" 2000)
|   (,@'("carl" " , the great") 1500)
|   ("dawn" 2300))
```

- и такую веб-страницу:

```
| `(html
|   (body
|     (table ((border "1"))
|       (tr ((width "200"))
|         ,@(make-row '( 1 2)))
|       (tr ((width "200"))
|         ,@(make-row '(99 65))))),
```

где `make-row` – это функция, представленная выше.

Используйте `check-expect` для проверки своих выражений. ■

Упражнение 234. Спроектируйте функцию `make-rank`, которая принимает список с названиями песен, упорядоченный по рейтингу, и создает его представление в виде HTML-таблицы. Рассмотрим следующий пример:

```
| (define one-list
|   '("Asia: Heat of the Moment"
|     "U2: One"
|     "The White Stripes: Seven Nation Army"))
```

Если применить `make-rank` к `one-list` и вывести получившуюся веб-страницу в браузере, то вы должны увидеть изображение, напоминающее скриншот на рис. 14.

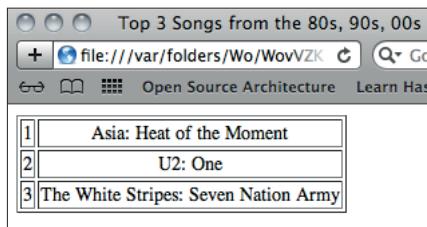
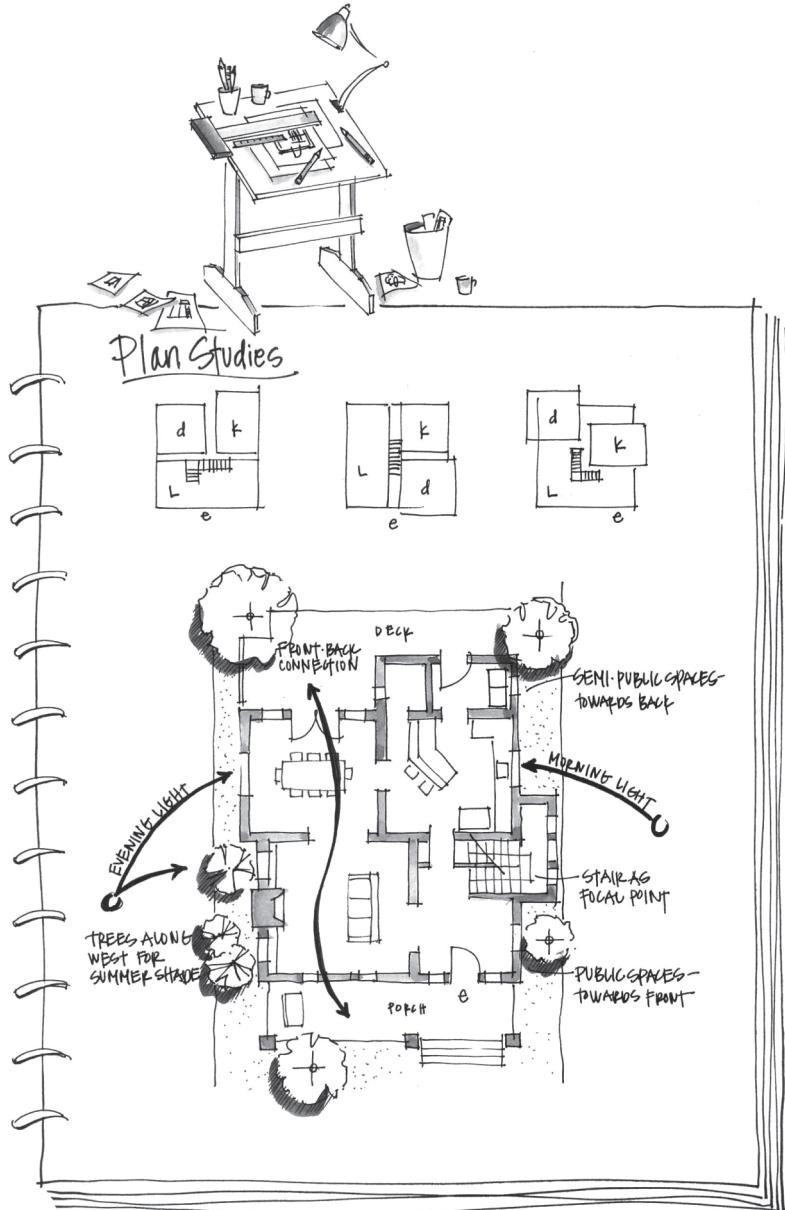


Рис. 14. Веб-страница, сгенерированная функцией на BSL+

Подсказка. У вас может появиться желание спроектировать функцию, которая упорядочивает список строк по рейтингу, но мы хотели бы, чтобы вы целиком сосредоточились на создании таблиц. Поэтому мы предоставляем следующие функции:

```
(define (ranking los)
  (reverse (add-ranks (reverse los))))  
  
(define (add-ranks los)
  (cond
    [(empty? los) '()]
    [else (cons (list (length los) (first los))
                (add-ranks (rest los)))])))
```

Прежде чем использовать эти функции, снабдите их сигнатурами и заявлениями о цели. Затем изучите их работу в области взаимодействий в DrRacket. В части VI мы добавим в рецепт проектирования способ создания более простых функций для вычисления рейтингов, чем ranking и add-ranks. ■



III АБСТРАКЦИИ

Многие наши определения данных и функций выглядят одинаково. Например, определения списков строк и списков чисел имеют только два отличия: в названиях классов данных и в использовании слов «String» и «Number». Точно так же функция, выполняющая поиск конкретной строки в списке строк, почти неотличима от функции, выполняющей поиск конкретного числа в списке чисел.

Как показывает опыт, подобные сходства нередко становятся проблемой. Сходство обусловлено тем, что программист – вольно или невольно – копирует код. Когда программист сталкивается с задачей, напоминающей другую, уже решенную им, он копирует решение и модифицирует новую копию так, чтобы она решала новую задачу. Такое поведение легко заметить как в «реальном» программировании, так и в мире электронных таблиц и математического моделирования. Однако, копируя код, программисты копируют и ошибки, и когда ошибка будет обнаружена, ее придется исправить уже в нескольких копиях. Кроме того, при изменении или расширении определения базовых данных придется отыскать и соответствующим образом изменить все копии. Этот трудоемкий и утомительный процесс, подверженный ошибкам и требующий ненужных затрат труда.

Хорошие программисты стараются устраниТЬ сходство настолько, насколько позволяет язык программирования. Под «устранением» подразумевается необходимость выявления сходств (и других проблем) в черновом варианте программы и избавление от них. На последнем этапе либо выполняется *абстрагирование*, либо используются существующие (абстрактные) функции. Часто этот процесс требуется повторить несколько раз, чтобы привести программу в удовлетворительную форму.

В первой половине этой части мы покажем, как абстрагировать сходства определений функций и данных. Результат этого процесса программисты также называют *абстракцией*. Во второй половине мы затронем вопросы использования существующих абстракций и познакомим с новыми конструкциями языка, упрощающими этот процесс. Несмотря на то что примеры для этой части взяты из области списков, излагаемые здесь идеи применимы повсеместно.

Программу можно сравнить с сочинением на заданную тему. Первая версия – это черновик, а черновики требуют правки.

14. Сходства повсюду

Если вы решали (хотя бы некоторые) упражнения в части II, то знаете, что многие решения выглядят одинаково. Такое сходство могло подталкивать вас скопировать решение одной задачи, чтобы решить следующую. Но вы *не должны красть код*, даже у самого себя. Вместо этого

*в DrRacket выберите пункт **Choose language** (Выбрать язык...) в меню **Language** (Язык) и в открывшемся диалоге выберите пункт **Intermediate Student** (Средний студент).*

следует абстрагировать похожие фрагменты кода, и данная глава научит вас этому.

Наши приемы борьбы со сходствами основаны на возможностях языка для студентов со средним уровнем подготовки «*Intermediate Student Language*», или просто ISL. Почти все другие языки программирования предлагают аналогичные возможности; в объектно-ориентированных языках можно найти дополнительные механизмы абстракции. Однако все эти механизмы обладают похожими характеристиками, описанными в этой главе, поэтому идеи проектирования, представленные здесь, применимы и в других контекстах.

14.1. Сходства в функциях

Рецепт проектирования определяет базовую организацию функции, потому что ее макет создается из определения данных безотносительно цели. Поэтому неудивительно, что функции, использующие одни и те же данные, выглядят одинаково.

Листинг 47. Две похожие функции

<pre>; Los -> Boolean ; проверят присутствие слова "dog" в l (define (contains-dog? l) (cond [(empty? l) #false] [else (or (string=? (first l) "dog") (contains-dog? (rest l)))]))</pre>	<pre>; Los -> Boolean ; проверят присутствие слова "cat" в l (define (contains-cat? l) (cond [(empty? l) #false] [else (or (string=? (first l) "cat") (contains-cat? (rest l))))]))</pre>
---	--

Рассмотрим функции из листинга 47, которые принимают список строк и отыскивают конкретную строку. Функция слева ищет слово "dog", а функция справа – слово "cat". Эти две функции почти неотличимы. Обе принимают список строк; обе состоят из выражения cond с двумя предложениями; обе возвращают #false, если на входе получен пустой список '(); обе используют выражение or, чтобы определить, является ли первый элемент искомой строкой, и обе использует рекурсию для просмотра остальной части списка, если это не так. Единственная разница – искомая строка, которая используется во вложенном выражении cond: в contains-dog? используется строка

"dog", а в `contains-cat?` используется строка "cat". Чтобы сделать различия более заметными, мы выделили их фоном.

Хороший программист слишком ленив, чтобы определять несколько похожих функций. Вместо этого он старается определить одну функцию, способную отыскать и слово "dog", и слово "cat". Эта универсальная функция принимает дополнительный параметр – исключенную строку – и в остальном подобна двум исходным функциям:

```
; String Los -> Boolean
; проверят присутствие строки s в l
(define (contains? s l)
  (cond
    [(empty? l) #false]
    [else (or (string=? (first l) s)
              (contains? s (rest l)))]))
```

Если вам действительно понадобится такая функция, как `contains-dog?`, то вы сможете определить ее как односстрочную функцию. То же самое верно и для `contains-cat?`. Эти функции определены в листинге 48. Сравните их с листингом 47 и проверьте, понимаете ли вы, как они были получены. Самое замечательное в новой функции `contains?` то, что с ее помощью можно выполнить поиск любой строки, и навсегда отпадает необходимость определять специализированные функции, такие как `contains-dog?`.

Листинг 48. Две похожие функции, измененные версии

<pre>; Los -> Boolean ; проверят присутствие слова "dog" в l (define (contains-dog? l) (contains? "dog" l))</pre>	<pre>; Los -> Boolean ; проверят присутствие слова "cat" в l (define (contains-cat? l) (contains? "cat" l))</pre>
--	--

То, что вы только что увидели, называется *абстракцией*, или, точнее, *функциональной абстракцией*. Абстрагирование различных версий функций – это один из способов устранения сходств из программ, и, как вы увидите далее, устранение сходств помогает не прикасаться к программе в течение длительного периода.

Упражнение 235. Используйте `contains?`, чтобы определить функции, которые ищут в списке строк слова "atom", "basic" и "zoo". ■

Упражнение 236. Создайте наборы тестов для следующих двух функций:

```
; Lon -> Lon
; прибавляет 1 к каждому элементу в l
(define (add1* l)
  (cond
    [(empty? l) '()]
    [else
      (cons
        (add1 (first l))
        (add1* (rest l))))]))
```

```
; Lon -> Lon
; прибавляет 5 к каждому элементу в l
(define (plus5 l)
  (cond
    [(empty? l) '()]
    [else
      (cons
        (+ (first l) 5)
        (plus5 (rest l))))]))
```

Специалисты в области информатики заимствовали термин «*абстракция*» из математики, где число «6» – это *абстрактное понятие*, потому что оно может представлять наборы из любых шести вещей. Напротив, «6 дюймов» или «6 яиц» – это конкретные варианты использования.

Затем абстрагируйте их. Определите эти две функции в терминах абстракции как односторонние функции и используйте существующие наборы тестов, чтобы убедиться, что исправленные определения работают правильно. Наконец, создайте функцию, которая вычитает 2 из каждого числа в заданном списке. ■

14.2. Отличающиеся сходства

Нам легко удалось абстрагировать функции `contains-dog?` и `contains-cat?`. Потребовалось лишь сравнить определения двух функций, заменить литерал параметром и проверить возможность определения прежних функций через абстрактную функцию. Такая абстракция настолько естественна, что мы без особых сложностей применяли ее в двух предыдущих частях книги.

В этом разделе мы покажем, как те же самые принципы дают более мощную форму абстракции. Взгляните на листинг 49. Обе функции в этом листинге принимают список чисел и пороговое значение. Левая функция возвращает список всех чисел, которые ниже порога, а функция справа – выше порога.

Эти две функции отличаются только оператором сравнения, который определяет, какое число из заданного списка должно быть частью результата. Функция слева использует оператор `<`, а справа – оператор `>`. В остальном они совершенно одинаковые, если не считать их имен.

Листинг 49. Две похожие функции

<pre>; Lon Number -> Lon ; выбирает из l числа, ; которые меньше t (define (small l t) (cond [(empty? l) '()] [else (cond [(< (first l) t) (cons (first l) (small (rest l) t)))] [else (small (rest l) t))]))]</pre>	<pre>; Lon Number -> Lon ; выбирает из l числа, ; которые больше t (define (large l t) (cond [(empty? l) '()] [else (cond [(> (first l) t) (cons (first l) (large (rest l) t)))] [else (large (rest l) t))]))]</pre>
---	---

Рассмотрим первый пример и абстрагируем эти две функции, добавив дополнительный параметр. Этот параметр будет представлять оператор сравнения, а не строку:

```
(define (extract R l t)
  (cond
    [(empty? l) '()]
    [else (cond
```

```

[(R (first l) t)
 (cons (first l)
       (extract R (rest l) t))]
[else
 (extract R (rest l) t))])

```

Чтобы применить эту новую функцию, мы должны передать ей три аргумента: функцию R, которая сравнивает два числа, список чисел l и порог t. Функция выберет все элементы i из l, для которых выражение (R i t) вернет #true.

Стоп! Здесь у вас должен возникнуть вопрос: имеет ли смысл такое определение. Мы, нимало не беспокоясь, создали функцию, которая принимает другую функцию. Маловероятно, что раньше вы видели нечто подобное. Однако, как оказывается, наш простой обучающий язык ISL поддерживает такую возможность, и поддержка определения подобных функций является одним из самых мощных инструментов хороших программистов, даже в языках, в которых функции, принимающие другие функции, кажутся невозможными.

Тестирование показывает, что (extract < l t) вычисляет тот же результат, что и (small l t):

```

(check-expect (extract < '() 5) (small '() 5))
(check-expect (extract < '(3) 5) (small '(3) 5))
(check-expect (extract < '(1 6 4) 5)
              (small '(1 6 4) 5))

```

Аналогично (extract > l t) вычисляет тот же результат, что и (large l t). Теперь мы можем определить две исходные функции через абстрактную:

```

; Lon Number -> Lon
(define (small-1 l t)
  (extract < l t))

```

```

; Lon Number -> Lon
(define (large-1 l t)
  (extract > l t))

```

Самое важное во всем этом не то, что определения small-1 и large-1 состоят из одной строки. Имея абстрактную функцию, такую как extract, вы можете найти ей хорошее применение в другом месте:

- 1) (extract = l t): это выражение извлечет из l все числа, которые равны t;
- 2) (extract <= l t): создаст список чисел из l, которые меньше или равны t;
- 3) (extract >= l t): а это последнее выражение извлечет из l все числа, которые больше или равны пороговому значению.

На самом деле первый аргумент в extract не обязательно должен быть одной из операций, предопределенных в языке ISL, – вы можете передать любую функцию, которая принимает два аргумента и возвращает логическое значение. Рассмотрим следующий пример:

Изучавшие курс математического анализа наверняка знакомы с дифференциальным оператором и неопределенным интегралом. И тот, и другой являются функциями, которые принимают и возвращают функции. Но мы не будем исходить из предположения, что вы прошли этот курс.

```
| ; Number Number -> Boolean
| ; площадь квадрата со стороной x, которая больше с
(define (squared? x c)
  (> (* x x) c))
```

Функция `squared?` проверяет, выполняется ли утверждение $x^2 > c$, и ее можно использовать вместе с `extract`:

```
| (extract squared? (list 3 4 5) 10)
```

Это применение извлечет из `(list 3 4 5)` все числа, квадрат которых больше 10.

Преимущества абстракции доступны на всех уровнях программирования: текстовые документы, электронные таблицы, небольшие приложения и крупные промышленные проекты. Создание абстракций для последних стимулирует исследование языков программирования и программной инженерии.

Упражнение 237. Вычислите `(squared? 3 10)` и `(squared? 4 10)` в DrRacket. Какой результат вернет выражение `(squared? 5 10)?` ■

Теперь вы знаете, что абстрактные функции могут быть более полезными, чем конкретные. Например, `contains?` определено полезнее, чем `contains-dog?` и `contains-cat?`, а `extract` полезнее, чем `small` и `large`. Также важно отметить, что абстракция дает нам единую точку контроля над всеми этими функциями. Если выяснится, что абстрактная функция содержит ошибку, достаточно исправить определение этой единственной функции, чтобы исправить все остальные функции, использующие ее. Точно так же, если будет найден способ ускорить вычисления в абстрактной функции или уменьшить ее энергопотребление, тогда все функции, определенные в терминах этой функции, ускорятся или уменьшат энергопотребление без каких-либо дополнительных усилий. Следующие упражнения показывают, как это работает.

Листинг 50. Поиск наименьшего (`inf`) и наибольшего (`sup`) чисел в списке

<pre> ; Nelon -> Number ; определяет наименьшее ; число в l (define (inf l) (cond [(empty? (rest l)) (first l)] [else (if (< (first l) (inf (rest l))) (first l) (inf (rest l))))]))</pre>	<pre> ; Nelon -> Number ; определяет наибольшее ; число в l (define (sup l) (cond [(empty? (rest l)) (first l)] [else (if (> (first l) (sup (rest l))) (first l) (sup (rest l))))]))</pre>
---	---

Упражнение 238. Абстрагируйте функции в листинге 50. Обе принимают непустые списки чисел (`Nelon`) и возвращают одно число. Функция слева возвращает наименьшее число в списке, а функция справа – наибольшее.

Определите функции `inf-1` и `sup-1` в терминах абстрактной функции. Проверьте их с помощью этих двух списков:

```
| (list 25 24 23 22 21 20 19 18 17 16 15 14 13
      12 11 10 9 8 7 6 5 4 3 2 1)
```

```
(list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
     17 18 19 20 21 22 23 24 25)
```

Почему они медленно обрабатывают длинные списки?

Измените исходные функции, используя в них функцию `max`, которая выбирает наибольшее из двух чисел, и `min`, которая выбирает наименьшее. Затем снова определите абстрактную функцию и функции `inf-2` и `sup-2` и снова протестируйте их с теми же входными данными. Почему эти версии работают намного быстрее?

Другой ответ на эти вопросы вы найдете в разделе 16.2. ■

14.3. Сходства в определениях данных

Теперь внимательно рассмотрим следующие два определения данных:

<pre>; Lon (List-of-numbers -- список чисел) ; имеет одно из значений: ; -- '() ; -- (cons Number Lon)</pre>	<pre>; Los (List-of-String -- список строк) ; имеет одно из значений: ; -- '() ; -- (cons String Los)</pre>
--	---

Слева определяется класс списков чисел, а справа – списков строк. Эти два определения данных очень похожи. Подобно похожим функциям, два определения данных имеют разные имена, но это не имеет значения, потому что имя может быть любым. Единственное реальное отличие касается первой позиции внутри `cons` во втором предложении, которое определяет, какие элементы содержит список.

Чтобы абстрагировать это единственное различие, будем действовать так, как если бы определение данных было функцией. Введем параметр, который сделает определение данных похожим на функцию, и там, где раньше были разные ссылки, используем этот параметр:

```
; [List-of ITEM] имеет одно из значений:
; -- '()
; -- (cons ITEM [List-of ITEM])
```

Такие абстрактные определения данных называются *параметрическими определениями данных*, потому что в них присутствует параметр. Проще говоря, параметрическое определение данных абстрагирует ссылку на конкретный набор данных так же, как функция абстрагирует конкретное значение.

Возникает вопрос, что можно использовать в роли параметра. В функциях параметры обозначают неизвестные значения; когда функция применяется, значение становится известным. В параметрическом определении данных параметр обозначает целый класс значений. Процесс подстановки имени коллекции данных в параметрическое определение данных называется *созданием экземпляра*; вот несколько примеров создания экземпляров абстракции `List-of`:

- подставляя [List-of Number], мы говорим, что ITEM представляет все числа, то есть это просто еще одно название для List-of-numbers;
- аналогично [List-of String] определяет тот же класс данных, что и List-of-String;
- если бы мы определили класс инвентарных записей, например

```
| (define-struct ir [name price])
| ; IR -- это структура:
| ;   (make-ir String Number)
```

то [List-of IR] стало бы именем списка инвентарных записей.

Следуя соглашениям, мы используем в параметрах определений данных имена, состоящие только из заглавных букв, а аргументы записываются, как это требуется.

Для проверки всех этих сокращений можно заменить параметр ITEM фактическим именем определения данных, например Number (число), и использовать подходящее имя для получившегося определения данных:

```
| ; List-of-numbers-again -- одно из значений:
| ; -- '()
| ; -- (cons Number List-of-numbers-again)
```

Поскольку определение данных ссылается само на себя, мы скопировали полное имя определения. Получившееся определение выглядит точно так же, как обычное определение для списков чисел и действительно идентифицирует тот же класс данных.

Теперь рассмотрим второй пример, начав с определения структуры:

```
| (define-struct point [hori veri])
```

В этой структуре используются два разных определения данных:

```
| ; Pair-boolean-string -- это структура:
| ;   (make-point Boolean String)
|
| ; Pair-number-image -- это структура:
| ;   (make-point Number Image)
```

В этом случае определения данных имеют два отличия, оба отмечены серым фоном. Здесь мы имеем различия в поле hori и два различия в поле veri. Соответственно, мы должны ввести два параметра, чтобы получить абстрактное определение данных:

```
| ; [CP H V] -- это структура:
| ;   (make-point H V)
```

Здесь H – это параметр, определяющий коллекцию данных для поля hori, а V – коллекцию данных для поля veri.

Чтобы создать экземпляр определения данных с двумя параметрами, нам понадобятся два имени коллекций данных. Используя Number (число) и Image (изображение) для параметров CP, мы получаем определение [CP Number Image], которое описывает коллекцию point, элементы которой объединяют число с изображением. Напротив, [CP Boolean String] объединяет логические значения со строками.

Упражнение 239. Список из двух элементов – еще одна часто используемая форма данных в программировании на ISL. Вот определение данных с двумя параметрами:

```
| ; [List X Y] -- это структура:  
| ; (cons X (cons Y '()))
```

Создайте экземпляры этого определения для описания следующих классов данных:

- пары чисел (Number);
- пары чисел (Number) и символов (1Strings);
- пары строк (String) и логических значений (Boolean).

Также создайте по одному конкретному примеру для каждого из этих трех определений данных. ■

Параметрические определения данных можно смешивать и сопоставлять. Например:

```
| ; [List-of [CP Boolean Image]]
```

Внешнее определение [List-of ...] означает, что мы имеем дело со списком. Вопрос в том, какие данные содержит список, и чтобы ответить на этот вопрос, нужно изучить внутреннюю часть определения List-of:

```
| ; [CP Boolean Image]
```

Внутренняя часть объединяет логическое значение и изображение в структуру. Соответственно,

```
| ; [List-of [CP Boolean Image]]
```

– это список экземпляров point, объединяющих логические значения и изображения. Аналогично

```
| ; [CP Number [List-of Image]]
```

– это экземпляр CP, объединяющий одно число со списком изображений.

Упражнение 240. Вот два странных, но похожих определения данных:

<pre> ; LStr -- одно из значений: ; -- String ; -- (make-layer LStr)</pre>	<pre> ; LNum -- одно из значений: ; -- Number ; -- (make-layer LNum)</pre>
---	---

В обоих случаях используется одно и то же определение структуры:

```
| (define-struct layer [stuff])
```

В обоих определяются вложенные формы данных: в первом используются числа, а во втором – строки. Приведите примеры данных для обоих. Абстрагируйте их. Затем создайте экземпляры абстрактного определения, чтобы получить оригиналы. ■

Упражнение 241. Сравните определения *NEList-of-temperature* и *NEList-of-Booleans*. Затем сформулируйте абстрактное определение данных *NEList-of*. ■

Упражнение 242. Вот еще одно параметрическое определение данных:

```
| ; [Maybe X] -- одно из значений:
| ; -- #false
| ; -- X
```

Интерпретируйте следующие определения данных: *[Maybe String]*, *[Maybe [List-of String]]* и *[List-of [Maybe String]]*.

Что означает такая сигнатура функции:

```
| ; String [List-of String] -> [Maybe [List-of String]]
| ; возвращает оставшуюся часть списка los, начиная с s
| ; иначе #false
| (check-expect (occurs "a" (list "b" "a" "d" "e"))
|                 (list "d" "e"))
| (check-expect (occurs "a" (list "b" "c" "d")) #f)
| (define (occurs s los)
|     los)
```

Выполните остальные шаги в рецепте проектирования. ■

14.4. Функции – это значения

Функции в этой части расширяют наше понимание вычисления значения в программе. Легко понять, как функции могут принимать не только числа, но также строки или изображения. Со структурами и списками немного сложнее, но они тоже представляют нечто конечное. Однако функции, принимающие другие функции, выглядят странно. Действительно, сама идея нарушает положения, изложенные в интермеццо 1: (1) при применении функций в качестве аргументов используются имена примитивов и функций, и (2) в позициях применения функций используются параметры.

Перечисление проблемы показывает, чем грамматика ISL отличается от грамматики BSL. Во-первых, язык выражений должен позволять включать в определение имена функций и элементарных операций. Во-вторых, первая позиция в применении должна допускать вставку чего-то другого, отличного от имен функций и элементарных операций; как минимум она должна допускать использование параметров, определяющих переменные и функции.

Кажется, что изменения грамматики требуют изменений правил вычисления, но в действительности меняется только набор значений. В частности, чтобы использовать функции в качестве аргументов функций, достаточно сказать, что функции и элементарные операции **являются** значениями.

Упражнение 243. Предположим, что область определений в DrRacket содержит:

```
| (define (f x) x)
```

Идентифицируйте значения по следующим выражениям:

- 1) (cons f '());
- 2) (f f);
- 3) (cons f (cons 10 (cons (f 10) '()))).

Объясните, почему они являются (или не являются) значениями. ■

Упражнение 244. Объясните, почему следующие предложения теперь являются допустимыми:

- 1) (define (f x) (x 10));
- 2) (define (f x) (x f));
- 3) (define (f x y) (x 'a y 'b)).

Аргументируйте свои рассуждения. ■

Упражнение 245. Разработайте функцию `function=?`. Она должна принимать две функции, которые преобразуют число в число, и определять, возвращают ли они одинаковые результаты для 1.2, 3 и -5.775.

Математики говорят, что две функции равны, если они возвращают один и тот же результат для одних и тех же входных значений – всех возможных входных значений.

Можно ли определить `function=?`, которая определяет равенство двух функций, преобразующих числа в числа? Если да, то определите ее. Если нет, то объясните, почему это невозможно, и подумайте о том, почему для такой простой идеи невозможно определить функцию. ■

14.5. Вычисления с функциями

Переключение с языка BSL+ на ISL позволяет передавать функции в аргументах и использовать имена в первой позиции применений. DrRacket работает с именами в этих позициях, как и в любых других, но, естественно, он ожидает встретить в них функции. Удивительно, но простой адаптации законов алгебры достаточно для оценки программ на ISL.

Давайте посмотрим, как это работает, на примере `extract` из раздела 14.2. Очевидно, что

```
| (extract < '() 5) == '()
```

Мы можем использовать закон подстановки из интермеццо 1 и продолжить вычисления с телом функции. Как и прежде, параметры R, l и t заменяются их аргументами <, '() и 5 соответственно. Далее следует простая арифметика, начиная с условных выражений:

```

==  

(cond  

  [(empty? '()) '())  

  [else (cond  

    [(< (first '()) t)  

     (cons (first '()) (extract < (rest '()) 5))]  

    [else (extract < (rest '()) 5))])]  

==  

(cond  

  (#true '())  

  [else (cond  

    [(< (first '()) t)  

     (cons (first '()) (extract < (rest '()) 5))]  

    [else (extract < (rest '()) 5))])]  

== '()

```

Теперь рассмотрим список с одним элементом:

```
| (extract < (cons 4 '()) 5)
```

Результатом должен быть список (cons 4 '()), потому что список содержит единственный элемент 4 и условие (< 4 5) истинно. Вот первый шаг оценки:

```

(extract < (cons 4 '()) 5)  

==  

(cond  

  [(empty? (cons 4 '())) '())  

  [else (cond  

    [(< (first (cons 4 '())) 5)  

     (cons (first (cons 4 '()))  

       (extract < (rest (cons 4 '()) 5)))]  

    [else (extract < (rest (cons 4 '()) 5))])])

```

И снова все вхождения R заменяются на <, l – на (cons 4 '()), и t – на 5. Дальше все просто:

```

(cond  

  [(empty? (cons 4 '())) '())  

  [else (cond  

    [(< (first (cons 4 '())) 5)  

     (cons (first (cons 4 '()))  

       (extract < (rest (cons 4 '()) 5)))]  

    [else (extract < (rest (cons 4 '()) 5))])]  

==  

(cond  

  [#false '())  

  [else (cond  

    [(< (first (cons 4 '())) 5)  

     (cons (first (cons 4 '()))  

       (extract < (rest (cons 4 '()) 5)))]  

    [else (extract < (rest (cons 4 '()) 5))])])

```

```

==  

(cond  

  [(< (first (cons 4 '())) 5)  

   (cons (first (cons 4 '()))  

         (extract < (rest (cons 4 '())) 5))]  

  [else (extract < (rest (cons 4 '())) 5)])  

==  

(cond  

  [(< 4 5)  

   (cons (first (cons 4 '()))  

         (extract < (rest (cons 4 '())) 5))]  

  [else (extract < (rest (cons 4 '())) 5)])

```

Это был ключевой шаг, когда после подстановки используется `<`. Дальше продолжается обычная арифметика:

```

==  

(cond  

  [#true  

   (cons (first (cons 4 '()))  

         (extract < (rest (cons 4 '())) 5))]  

  [else (extract < (rest (cons 4 '())) 5)])  

==  

(cons 4 (extract < (rest (cons 4 '())) 5))  

==  

(cons 4 (extract < '() 5))  

==  

(cons 4 '())

```

Последний шаг – это уравнение сверху, позволяющее применить закон замены равного равным.

Наш последний пример – применение `extract` к списку с двумя элементами:

```

(extract < (cons 6 (cons 4 '())) 5)
== (extract < (cons 4 '()) 5)
== (cons 4 (extract < '() 5))
== (cons 4 '())

```

Шаг 1 – новый для нас. Он обрабатывает случай, когда `extract` встречает первый элемент, который не меньше порогового значения.

Упражнение 246. Проверьте шаг 1 в последнем вычислении

```

(extract < (cons 6 (cons 4 '())) 5)
==  

(extract < (cons 4 '()) 5)

```

используя движок пошаговых вычислений в DrRacket. ■

Упражнение 247. Вычислите выражение `(extract < (cons 8 (cons 4 '())) 5)` с помощью движка пошаговых вычислений в DrRacket. ■

Упражнение 248. Вычислите выражения `(squared>? 3 10)` и `(squared>? 4 10)` с помощью движка пошаговых вычислений в DrRacket. ■

Рассмотрим следующее взаимодействие:

```

> (extract squared>? (list 3 4 5) 10)
(list 4 5)

```

Вот несколько шагов, которые показывает движок пошаговых вычислений:

```
(extract squared>? (list 3 4 5) 10)           ①
==                                     ②
(cond
  [(empty? (list 3 4 5)) '()]
  [else
    (cond
      [(squared>? (first (list 3 4 5)) 10)
       (cons (first (list 3 4 5))
             (extract squared>?
                       (rest (list 3 4 5))
                       10))]
      [else (extract squared>?
                      (rest (list 3 4 5))
                      10))])
  == ... ==
(cond
  [(squared>? 3 10)
   (cons (first (list 3 4 5))
         (extract squared>?
                   (rest (list 3 4 5))
                   10))]
  [else (extract squared>?
                  (rest (list 3 4 5))
                  10))])]
```

③

Используйте движок пошаговых вычислений, чтобы подтвердить шаги ① и ②. Продолжайте пошаговые вычисления и заполните промежуток между шагами ② и ③. Объясните каждый шаг, используя закон.

Упражнение 249. Функции – это значения: аргументы, результаты, элементы в списках. Поместите следующие определения и выражения в область определений DrRacket и используйте движок пошаговых вычислений, чтобы понять, как работает эта программа:

```
(define (f x) x)
(cons f '())
(f f)
(cons f (cons 10 (cons (f 10) '())))
```

Движок пошаговых вычислений отображает функции как лямбда-выражения; см. главу 17. ■

15. Проектирование абстракций

По сути, под абстрагированием подразумевается превращение чего-то конкретного в параметр. Мы уже видели несколько примеров этого в предыдущей главе. Чтобы абстрагировать функции с похожими определениями, мы добавляли параметры, заменяющие конкретные значения в определении. Чтобы абстрагировать похожие определения данных, мы создавали параметрические определения. Когда вы будете работать с другими языками программирования, то увидите, что их механизмы абстракции тоже требуют введения параметров, которые, впрочем, могут не быть параметрами функций.

15.1. Абстрагирование примеров

Когда вы впервые учились складывать числа, вы практиковались на конкретных примерах. Ваши родители, вероятно, учили вас складывать маленькие числа на пальцах. Позже вы узнали, как складывать два произвольных числа, и познакомились с вашим первым видом абстракции. Еще позже вы научились формулировать выражения, преобразующие температуру из градусов Цельсия в градусы Фаренгейта или вычисляющие расстояние, которое автомобиль проезжает с заданной скоростью и за определенное время. Проще говоря, вы перешли от очень конкретных примеров к абстрактным отношениям.

В этом разделе мы представим вашему вниманию рецепт проектирования для создания абстракций из примеров. Как показала предыдущая глава, создавать абстракции легко. Мы оставим все сложности до следующего раздела, а здесь просто покажем вам, как находить и использовать существующие абстракции.

Вспомним, что мы узнали в главе 14. Мы начали с двух конкретных определений, сравнили их, отметили различия, а затем абстрагировали. Собственно, это все, что нужно для создания абстракций.

1. Шаг 1 – **сравнить** два элемента и выявить сходства.

Обнаружив определения двух функций, которые почти неотличимы, кроме их имен и некоторых значений в аналогичных местах, сравните их и отметьте различия. Если определения различаются более чем в одном месте, соедините соответствующие различия линией.

Рецепт требует существенной модификации, чтобы абстрагировать различия, не выражающиеся значениями.

Листинг 51. Пара похожих функций

```
; List-of-numbers -> List-of-numbers  
; преобразует список температур в градусах  
; Цельсия в список температур в  
; градусах Фаренгейта  
  
(define (cf* l)
```

```
; Inventory -> List-of-strings  
; извлекает названия игрушек  
; из инвентаризационного списка  
  
(define (names i)
```

```

(cond
  [(empty? l) '()]
  [else
    (cons
      (C2F (first l))
      (cf* (rest l)))]))

; Number -> Number
; преобразует температуру в градусах
; Цельсия в температуру в градусах
; Фаренгейта
(define (C2F c)
  (+ (* 9/5 c) 32))

(define (IR-name i)
  (names (rest i)))

(define-struct IR
  [name price])
; IR -- это структура:
; (make-IR String Number)
; Inventory -- одно из значений:
; -- '()
; -- (cons IR Inventory)

```

В листинге 51 показана пара похожих определений функций. Эти две функции применяют некоторую функцию к каждому элементу списка. Они различаются только применяемой функцией. Существенные различия выделены цветом. Также имеются два несущественных отличия: имена функций и имена параметров.

2. Далее мы выполняем абстрагирование. *Абстрагирование* означает замену выделенного кода новыми именами и добавление этих имен в список параметров. В данном примере, после замены различий на имя `g`, мы получаем следующую пару функций; см. листинг 52. Это первое изменение устраниет существенное отличие. Теперь обе функции просматривают список и применяют заданную функцию к каждому элементу.

Несущественные различия – имена функций, а иногда и имена некоторых параметров – легко устранить. Вы уже знаете, что проверка синтаксиса в DrRacket позволяет делать это систематически и легко; см. нижнюю часть листинга 52. Мы решили использовать для функции имя `map1` и для параметра списка – имя `k`. Независимо от выбранных имен, в результате получатся два идентичных определения функций.

Листинг 52. Те же две функции после абстрагирования

```

(define (cf* l g)
  (cond
    [(empty? l) '()]
    [else
      (cons
        (g (first l))
        (cf* (rest l) g))]))

(define (map1 k g)
  (cond
    [(empty? k) '()]
    [else
      (cons
        (g (first k))
        (map1 (rest k) g)))])

(define (names i g)
  (cond
    [(empty? i) '()]
    [else
      (cons
        (g (first i))
        (names (rest i) g)))])

(define (map1 k g)
  (cond
    [(empty? k) '()]
    [else
      (cons
        (g (first k))
        (map1 (rest k) g))]))

```

Мы разобрали простой пример. Во многих случаях вы обнаружите, что существует более чем одна пара различий. Главное – найти все эти пары. Когда вы отметите различия ручкой или карандашом, соедините их линиями. Затем добавьте по одному дополнительному параметру для каждой пары. И не забудьте изменить все рекурсивные применения функции, задействовав дополнительные параметры.

3. Теперь нужно убедиться, что новая функция является правильной абстракцией исходной пары функций. Убедиться означает **протестировать**, то есть определить две исходные функции в терминах абстракции.

Для этого предположим, что одна исходная функция называется `f-original` и принимает один аргумент, а абстрактная функция называется `abstract`. Если `f-original` отличается от другой конкретной функции использованием одного значения, скажем `val`, то следующая функция `f-from-abstract`:

```
| (define (f-from-abstract x)
|   (abstract x val))
```

должна действовать в точности как `f-original`. То есть выражение `(f-from-abstract V)` должно возвращать тот же результат, что и `(f-original V)`, для всех допустимых значений `V`. В частности, новая функция должна возвращать те же значения, которые возвращает `f-original` в тестах. Соответственно, вы должны переформулировать эти тесты для `f-from-abstract` и убедиться, что они выполняются успешно.

Вернемся к текущему примеру:

```
| ; List-of-numbers -> List-of-numbers
| (define (cf*-from-map1 l) (map1 l C2F))
|
| ; Inventory -> List-of-strings
| (define (names-from-map1 i) (map1 i IR-name))
```

Полный пример должен включать несколько тестов, поэтому можно предположить, что функции `cf*` и `names` сопровождаются своими тестами:

```
| (check-expect (cf* (list 100 0 -40))
|               (list 212 32 -40))

| (check-expect (names
|                 (list
|                   (make-IR "doll" 21.0)
|                   (make-IR "bear" 13.0)))
|               (list "doll" "bear"))
```

Чтобы убедиться в правильной работе функций, определенных в терминах `map1`, можно скопировать тесты и изменить имена функций в них:

```
(check-expect
  (cf*-from-map1 (list 100 0 -40))
  (list 212 32 -40))

(check-expect
  (names-from-map1
    (list
      (make-IR "doll" 21.0)
      (make-IR "bear" 13.0)))
  (list "doll" "bear"))
```

4. Для новой абстракции нужно добавить сигнатуру, потому что, как объясняется в главе 16, повторное использование абстракций начинается с изучения их сигнатур. Правильно сформулировать сигнатуру – серьезная задача. Используем текущий пример, чтобы проиллюстрировать эту сложность; решение будет показано в разделе 15.2.

Рассмотрим проблему, возникающую при формулировании сигнатуры для `map1`. С одной стороны, если рассматривать `map1` как абстракцию `cf*`, то можно подумать, что сигнатура должна иметь такой вид:

```
| ; List-of-numbers [Number -> Number] -> List-of-numbers
```

то есть оригинальная сигнатура дополняется фрагментом

```
| ; [Number -> Number]
```

Поскольку дополнительный параметр `map1` является функцией, использование сигнатуры функции для его описания не должно вас удивлять. Эта сигнатура функции также довольно проста; она обозначает все функции, которые преобразуют число в число. Здесь такой функцией является `C2F`, но к этому классу функций относятся также `add1`, `sin` и `imag-part`.

С другой стороны, если рассматривать `map1` как абстракцию `names`, то сигнатура получится совершенно другой:

```
| ; Inventory [IR -> String] -> List-of-strings
```

На этот раз дополнительным параметром является `IR-name` – функция-селектор, которая принимает экземпляр `IR` и возвращает строку. Очевидно, что вторая сигнатуре не имеет смысла для первого случая, и наоборот. Чтобы подходить для обоих случаев, сигнатуре `map1` должна выражать соответствие между `Number`, `IR` и `String`.

Кроме того, вы, вероятно, уже решили использовать имя `List-of`. Очевидно, что проще написать `[List-of IR]`, чем подробно описывать определение данных для `Inventory`. Так что пока мы будем использовать `List-of`, когда речь идет о списках, и вам советуем поступить так же.

После абстрагирования функций следует проверить, есть ли другие варианты использования абстрактной функции. Если есть, то такая

абстракция будет по-настоящему полезна. Рассмотрим, например, `map1`. Легко заметить, что с ее помощью можно прибавить 1 к каждому числу в списке чисел:

```
; List-of-numbers -> List-of-numbers
(define (add1-to-each l)
  (map1 l add1))
```

Точно так же `map1` можно использовать для извлечения цены из каждой инвентарной записи. Когда у вас получится представить множество таких применений новой абстракции, добавьте ее в библиотеку полезных функций, чтобы всегда иметь ее под рукой. Вполне вероятно, что кто-то уже подумал об этом и функция уже является частью языка. Информацию о такой функции, как `map1`, см. в главе 16.

Листинг 53. Похожие функции для упражнения 250

<pre>; Number -> [List-of Number] ; вычисляет синусы для значений от n ; до 0 (включительно) и возвращает их ; в виде списка (define (tab-sin n) (cond [(= n 0) (list (sin 0))] [else (cons (sin n) (tab-sin (sub1 n)))]))</pre>	<pre>; Number -> [List-of Number] ; вычисляет квадратные корни для значений от n ; до 0 (включительно) и возвращает их ; в виде списка (define (tab-sqrt n) (cond [(= n 0) (list (sqrt 0))] [else (cons (sqrt n) (tab-sqrt (sub1 n))))]))</pre>
---	--

Листинг 54. Похожие функции для упражнения 251

<pre>; [List-of Number] -> Number ; вычисляет сумму чисел ; в списке l (define (sum l) (cond [(empty? l) 0] [else (+ (first l) (sum (rest l))))]))</pre>	<pre>; [List-of Number] -> Number ; вычисляет произведение чисел ; в списке l (define (product l) (cond [(empty? l) 1] [else (* (first l) (product (rest l))))]))</pre>
---	--

Упражнение 250. Спроектируйте функцию `tabulate`, абстрагирующую две функции из листинга 53. Закончив проектирование, попробуйте с помощью `tabulate` определить функции `tab-sqr` и `tab-tan`. ■

Упражнение 251. Спроектируйте функцию `fold1`, абстрагирующую две функции из листинга 54. ■

Упражнение 252. Спроектируйте функцию `fold2`, абстрагирующую две функции из листинга 55. Сравните это упражнение с упражнением 251. Несмотря на то что в обоих используется функция `prod-uct`, это упражнение создает дополнительную проблему, потому что вторая функция, `image*`, использует список экземпляров `Posn` и создает изображение. Тем не менее вам вполне под силу найти решение, опираясь на знания, полученные в этом разделе. Сравнение поможет вам получить интересное представление об абстрактных сигнатурах. ■

Листинг 55. Похожие функции для упражнения 252

```

; [List-of Number] -> Number
(define (product l)
  (cond
    [(empty? l) 1]
    [else
      (* (first l)
         (product
           (rest l))))]))
```



```

; [List-of Posn] -> Image
(define (image* l)
  (cond
    [(empty? l) emt]
    [else
      (place-dot
        (first l)
        (image* (rest l))))]))
```



```

; Posn Image -> Image
(define (place-dot p img)
  (place-image
    dot
    (posn-x p) (posn-y p)
    img))
```



```

; графические константы:
(define emt
  (empty-scene 100 100))
(define dot
  (circle 3 "solid" "red"))
```

Наконец, абстрагирование определений данных производится аналогично. Дополнительные параметры в определениях данных представляют коллекции значений, а под тестированием подразумевается составление определений данных для некоторых конкретных примеров. В общем случае абстрагирование определений данных дается, как правило, проще, чем абстрагирование функций, поэтому мы оставляем вам адаптировать рецепт проектирования как самостоятельное упражнение.

15.2. Сходства в сигнатурах

Как оказывается, сигнатура функции является ключом к ее повторному использованию. Поэтому важно научиться правильно формулировать сигнатуры, описывающие абстракции в наиболее общих терминах. Чтобы понять эту идею, начнем с того, что выразим простую, но, возможно, поразительную для вас мысль, что сигнатуры фактически являются определениями данных.

И сигнатуры, и определения данных описывают классы данных; разница лишь в том, что определения данных также дают имя классу данных, а сигнатуре – нет. Тем не менее, записывая

```

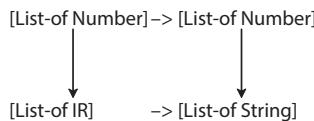
; Number Boolean -> String
(define (f n b) "hello world")
```

вы в первой строке описываете весь класс данных, а во второй утверждаете, что f принадлежит этому классу. Точнее говоря, сигнатура описывает класс **всех функций**, которые принимают Number и Boolean и возвращают String.

В общем случае стрелочная нотация в сигнатурах подобна обозначению List-of из раздела 14.3. Последнее принимает (имя) одного класса данных, скажем X, и описывает все списки экземпляров X, не присваивая ему имени. Стрелочная нотация принимает произвольное количество классов данных и описывает коллекцию функций.

Это означает, что рецепт проектирования абстракции применим и к сигнатурам. Вы сравниваете похожие сигнатуры; подчеркиваете различия; а затем заменяете их параметрами. Процесс абстрагирования сигнатур кажется более сложным, чем абстрагирование функций, отчасти потому, что сигнатуры уже являются абстрактными частями рецепта проектирования, а отчасти потому, что стрелочная нотация намного сложнее, чем все остальное, с чем мы сталкивались.

Начнем с сигнатур cf^* и names:



Эта диаграмма получена после шага сравнения. Сравнение двух сигнатур показывает, что они имеют два отличия: слева стрелка связывает классы данных Number и IR, а справа – Number и String.

Если заменить два различия какими-то параметрами, скажем X и Y, то получится та же сигнатура:

| ; [X Y] [List-of X] -> [List-of Y]

Новая сигнатура начинается с последовательности переменных, по аналогии с определениями функций и определениями данных. Эти переменные являются параметрами сигнатуры, если можно так выразиться, которые подобны параметрам функций и определений данных. Если говорить более конкретно, то последовательность переменных аналогична ITEM в определении List-of или X и Y в определении CP в разделе 14.3. И точно так же X и Y обозначают классы значений.

Реализацией этого списка параметров является остальная часть сигнатуры, параметры которой заменяются коллекциями данных: либо их именами, либо другими параметрами, либо сокращениями, например List-of. Таким образом, заменив X и Y на Number, мы получаем сигнатуру для cf^* :

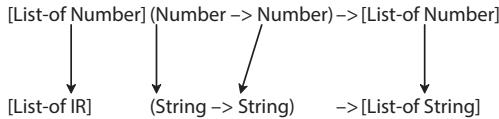
| ; [List-of Number] -> [List-of Number]

Если подставить IR и String, то получится сигнатура для names:

| ; [List-of IR] -> [List-of String]

Это объясняет, почему данную параметризованную сигнатуру можно рассматривать как абстракцию исходных сигнатур для cf^* и names.

Добавив дополнительный функциональный параметр в эти две функции, мы получим map1 со следующими сигнатурами:



И снова сигнатуры представлены в графической форме со стрелками, соединяющими соответствующие различия. Согласно этой диаграмме, различия во втором аргументе – функции – связаны с различиями в исходных сигнатурках. В частности, Number и IR слева относятся к элементам в первом аргументе – списке, – а Number и String справа – к элементам в результате, тоже в списке.

Поскольку перечисление параметров сигнатурки – это дополнительная работа, то с этого момента мы просто будем говорить, что все переменные в сигнатурке являются параметрами. Другие языки программирования, однако, требуют явного перечисления параметров сигнатур, но зато позволяют сформулировать дополнительные ограничения и проверять сигнатурки перед запуском программы.

Теперь применим тот же трюк, чтобы получить сигнатурку для `map1`:

```
| ; [X Y] [List-of X] [X -> Y] -> [List-of Y]
```

Эта сигнатурка указывает, что функция `map1` принимает список элементов, каждый из которых принадлежит некоторой коллекции данных (которую еще предстоит определить) с именем `X`. Она также принимает функцию, которая принимает элементы `X` и создает элементы из второй неизвестной коллекции с именем `Y`. Результатом `map1` является список, содержащий элементы из `Y`.

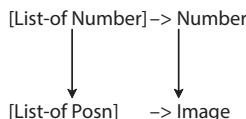
Абстрагирование сигнатур требует практики. Вот вторая пара:

```
| ; [List-of Number] -> Number
| ; [List-of Posn] -> Image
```

Это сигнатурки функций `product` и `image*` из упражнения 252. Несмотря на некоторые общие черты, эти сигнатурки имеют существенные отличия. Давайте сначала подробно рассмотрим сходства:

- обе сигнатурки описывают функцию с одним аргументом;
- в обоих описаниях аргументов используется конструкция `List-of`.

В отличие от первого примера, здесь одна сигнатурка дважды ссылается на `Number`, а другая – на `Posn` и `Image` в тех же позициях. Сравнение показывает, что первое вхождение `Number` соответствует `Posn`, а второе – `Image`:



Чтобы абстрагировать сигнатуры двух функций в упражнении 252, сделаем два первых шага из рецепта проектирования:

```
(define (pr* l bs jn)
  (cond
    [(empty? l) bs]
    [else
      (jn (first l)
           (pr* (rest l)
                 bs
                 jn)))))

(define (im* l bs jn)
  (cond
    [(empty? l) bs]
    [else
      (jn (first l)
           (im* (rest l)
                 bs
                 jn))))]))
```

Поскольку функции различаются двумя парами значений, исправленные версии должны принимать два дополнительных значения: одно является элементарным значением, которое будет использоваться в базовом случае, а другое – функцией, объединяющей результат естественной рекурсии с первым элементом в данном списке.

Теперь это знание нужно преобразовать в две сигнатуры для двух новых функций. Сделав это для pr^* , получаем:

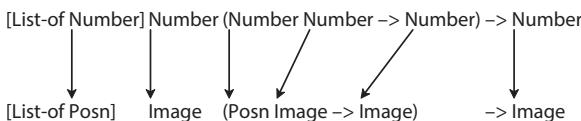
```
; [List-of Number] Number [Number Number -> Number]
; -> Number
```

потому что результат базового случая является числом и потому что во втором предложении в `cond` используется функция `+`. Аналогично сигнтура для im^* приобретает вид:

```
; [List-of Posn] Image [Posn Image -> Image]
; -> Image
```

Как видно из определения функции im^* , базовый случай возвращает изображение, а для комбинирования используется функция `place-dot`, которая объединяет `Posn` и `Image` в изображение.

Теперь возьмем диаграмму на рисунке выше и добавим в нее дополнительные входные значения:



На этой диаграмме сразу видно, что две исправленные сигнатуры имеют еще более детальную организацию. Кроме того, отличия наблюдаются не только в базовых случаях, но и в части, где описывается функция комбинирования. Всего имеется шесть пар отличий, но все они сводятся к двум:

- 1) некоторые вхождения `Number` соответствуют `Posn`;
- 2) а некоторые соответствуют `Image`.

Итак, чтобы абстрагировать эту сигнатуру, нам нужны две переменные, по одной для каждого вида соответствия.

Вот абстрагированная сигнатура для `fold2` из упражнения 252:

```
| ; [X Y] [List-of X] Y [X Y -> Y] -> Y
```

Стоп! Убедитесь, что замена обоих параметров, `X` и `Y`, на `Number` даст в результате сигнатуру для `rg*`, а замена тех же параметров на `Posn` и `Image` даст сигнатуру для `in*`.

Два примера показывают, как искать обобщенные сигнатуры. Процесс поиска ничем не отличается от процесса абстрагирования функций. Однако из-за неформального характера сигнатур результат нельзя проверить с помощью тестов. Вот пошаговая инструкция.

1. Возьмите два похожих определения функций, назовем их `f` и `g`, и выявите сходства и различия в них. Цель состоит в том, чтобы определить организацию сигнатур и отметить места, где одна сигнатура отличается от другой. Соедините различия попарно стрелками, как мы делаем это, когда анализируем тела функций.
2. Преобразуйте `f` и `g` в `f-abs` и `g-abs`. То есть добавьте параметры, устраниющие различия между `f` и `g`. Создавайте сигнатуры для `f-abs` и `g-abs`. Не забывайте, что новые параметры предназначены для замены, это поможет вам определить новые элементы сигнатур.
3. Проверьте, распространяется ли анализ на шаге 1 на сигнатуре `f-abs` и `g-abs`. Если да, то замените различия переменными, которые охватывают классы данных. Если сигнатуре совпадают, то это означает, что вы получили сигнатуру для абстрактной функции.
4. Протестируйте абстрактную сигнатуру. Во-первых, убедитесь, что соответствующие подстановки вместо переменных в абстрактной сигнатуре дают сигнатуре `f-abs` и `g-abs`. Во-вторых, убедитесь, что обобщенная сигнатура соответствует коду. Например, если `r` – это новый параметр и его сигнатура имеет вид:

```
| ; ... [A B -> C] ...
```

тогда `r` всегда должна применяться к двум аргументам, `A` и `B`. А результатом применения `r` должно быть значение `C`, и ее следует использовать там, где ожидаются значения `C`.

Так же как в случае с абстрагированием функций, ключевым моментом является сравнение конкретных сигнатур и определение сходств и различий. Обладая достаточной практикой и интуицией, вы сможете абстрагировать сигнатур без особого труда.

Упражнение 253. Каждая из следующих сигнатур описывает класс функций:

```
| ; [Number -> Boolean]
| ; [Boolean String -> Boolean]
| ; [Number Number Number -> Number]
| ; [Number -> [List-of Number]]
| ; [[List-of Number] -> Boolean]
```

Опишите каждый класс хотя бы одним примером на языке ISL. ■

Упражнение 254. Сформулируйте сигнатуры для следующих функций:

- `sort-n`, которая принимает список чисел и функцию, принимающую два числа (из списка) и возвращающую логическое значение; `sort-n` должна возвращать отсортированный список чисел;
- `sort-s`, которая принимает список строк и функцию, принимающую две строки (из списка) и возвращающую логическое значение; `sort-s` должна возвращать отсортированный список строк.

Затем абстрагируйте эти две сигнатуры, выполнив перечисленные выше шаги. Также покажите, что обобщенную сигнатуру можно использовать для описания функции сортировки списков экземпляров IR. ■

Упражнение 255. Сформулируйте сигнатуры для следующих функций:

- `map-p`, которая принимает список чисел и функцию, преобразующую одно число в другое, `map-p` должна возвращать список преобразованных чисел;
- `map-s`, которая принимает список строк и функцию, преобразующую одну строку в другую, `map-s` должна возвращать список преобразованных строк.

Затем абстрагируйте эти две сигнатуры, выполнив перечисленные выше шаги. Также покажите, что обобщенную сигнатуру можно использовать для описания функции `map1`, упоминавшейся выше. ■

15.3. Единая точка управления

Программы можно сравнить с черновиками статей, а черновики необходимо править, чтобы устранить опечатки, грамматические ошибки, обеспечить последовательность изложения мыслей и исключить повторы. Мало кому нравятся статьи, в которых много повторов, и мало кому понравится читать такие же программы.

Устранение сходства с помощью абстракций имеет много преимуществ. Абстракции упрощают определения. Они также могут помочь выявить проблемы с существующими функциями, особенно если они не совсем похожи. Но самым важным преимуществом является создание единой точки контроля над некоторыми широко используемыми функциями.

Размещение определения некоторых функций в одном месте упрощает обслуживание программы. Когда обнаружится ошибка, достаточно внести правки только в одном месте, чтобы исправить ее. Когда обнаружится, что код должен обрабатывать данные в другой форме,

достаточно добавить код только в одно место. Когда будет найден более эффективный алгоритм, одно изменение повысит эффективность всех других функций, использующих усовершенствованную абстракцию. Если вы просто копировали функции или код, то вам придется найти все копии и исправить их; иначе ошибка может остаться неисправленной или только одна из функций будет работать быстрее.

Поэтому мы сформулируем следующее правило:

Создавайте абстракции вместо копирования и изменения какого-либо кода.

Наш рецепт проектирования абстракций очень прост. Но, чтобы в полной мере овладеть приемами абстрагирования, нужна практика. По мере накопления практического опыта вы будете совершенствовать свои навыки чтения, систематизации и поддержки программ. Лучшие программисты – это те, кто активно правит свои программы и создает новые абстракции, чтобы собрать в одном месте все, что связано с определенной задачей. Здесь мы изучаем эту практику на примере абстрагирования функций; в других книгах по программированию вы встретите иные формы абстрагирования, как, например, наследование в объектно-ориентированных языках.

15.4. Абстрагирование макетов

В первых двух главах в этой части представлено множество функций, основанных на одном и том же макете. В конце концов, рецепт проектирования требует организовать функции вокруг определений входных данных, поэтому неудивительно, что многие определения функций похожи друг на друга.

На самом деле абстрагировать необходимо макеты непосредственно, и делать это следует автоматически; некоторые экспериментальные языки программирования поддерживают такую возможность. Несмотря на то что эта тема все еще является предметом исследований, вы уже сможете понять основную идею. Рассмотрим макет функции, обрабатывающей списки:

```
(define (fun-for-l l)
  (cond
    [(empty? l) ...]
    [else (... (first l) ...
                ... (fun-for-l (rest l)) ...)]))
```

Здесь мы имеем два пропуска, по одному в каждой ветке. Используя этот макет для определения функции, обрабатывающей списки, мы обычно заполняем пропуски значениями в первом предложении `cond` и функциями `combine` во втором. Функция `combine` принимает первый элемент списка и результат естественной рекурсии и создает результат из этих двух элементов данных.

Теперь, зная, как создавать абстракции, вы можете завершить определение абстракции из этого неформального описания:

```
; [X Y] [List-of X] Y [X Y -> Y] -> Y
(define (reduce l base combine)
  (cond
    [(empty? l) base]
    [else (combine (first l)
                  (reduce (rest l) base combine))]))
```

Эта функция принимает два дополнительных аргумента: `base` – значение для базового случая и `combine` – функцию, которая комбинирует значения во втором предложении.

Используя `reduce`, можно определить многие простые функции обработки списков. Вот определения `sum` и `product` – двух функций, которые уже не раз использовались в первых нескольких разделах этой главы:

```
; [List-of Number] -> Number
(define (sum lon)
  (reduce lon 0 +))
```

```
; [List-of Number] -> Number
(define (product lon)
  (reduce lon 1 *))
```

В `sum` в базовом случае всегда получается 0; сложение первого элемента списка с результатом естественной рекурсии объединяет значения во втором предложении. Аналогично выглядит описание работы функции `product`. Другие функции обработки списков можно определить похожим образом с помощью `reduce`.

16. Использование абстракций

После создания абстракции ее следует использовать везде, где это возможно. Абстракции создают единые точки контроля и позволяют вам экономить время и силы, а также помогают **читателям** кода понять ваши намерения. Если абстракция хорошо известна и встроена в язык или определена в его стандартных библиотеках, то она более четко сигнализирует о том, что делает ваша функция, чем специально разработанный код.

Эта глава посвящена повторному использованию абстракций, имеющихся в языке ISL. Она начинается с раздела, перечисляющего абстракции, имеющиеся в ISL, часть из которых вы уже видели выше под другими именами. Остальные разделы посвящены повторному использованию таких абстракций. Одним из ключевых компонентов является новая синтаксическая конструкция `local`, используемая для определения локальных функций и переменных (и даже структур) внутри функций. В последнем разделе будет представлен вспомогательный компонент `lambda`, используемый для создания безымянных функций; `lambda` предлагает дополнительные удобства (хотя и несущественные) при повторном использовании абстрактных функций.

Листинг 56. Абстрактные функции в ISL для обработки списков (1)

```
; [X] N [N -> X] -> [List-of X]
; конструирует список, применяя f к 0, 1, ..., (sub1 n)
; (build-list n f) == (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

; [X] [X -> Boolean] [List-of X] -> [List-of X]
; создает список из тех элементов lx, для которых выполняется p
(define (filter p lx) ...)

; [X] [List-of X] [X X -> Boolean] -> [List-of X]
; создает версию lx, отсортированную с помощью стр
(define (sort lx cmp) ...)

; [X Y] [X -> Y] [List-of X] -> [List-of Y]
; конструирует список, применяя f к каждому элементу в lx
; (map f (list x-1 ... x-n)) == (list (f x-1) ... (f x-n))
(define (map f lx) ...)

; [X] [X -> Boolean] [List-of X] -> Boolean
; определяет, выполняется ли p для всех элементов в lx
; (andmap p (list x-1 ... x-n)) == (and (p x-1) ... (p x-n))
(define (andmap p lx) ...)

; [X] [X -> Boolean] [List-of X] -> Boolean
; определяет, выполняется ли p хотя бы для одного элемента в lx
; (ormap p (list x-1 ... x-n)) == (or (p x-1) ... (p x-n))
(define (ormap p lx) ...)
```

16.1. Имеющиеся абстракции

В языке ISL имеется множество абстрактных функций для обработки натуральных чисел и списков. В листинге 56 перечислены заголовки наиболее важных из них. Первая функция строит списки, преобразуя натуральные числа:

```
| > (build-list 3 add1)
| (list 1 2 3)
```

Следующие три обрабатывают и производят списки:

```
| > (filter odd? (list 1 2 3 4 5))
| (list 1 3 5)
| > (sort (list 3 2 1 4 5) >)
| (list 5 4 3 2 1)
| > (map add1 (list 1 2 2 3 3 3))
| (list 2 3 3 4 4 4)
```

Функции `andmap` и `ormap` «сворачивают» входные списки в логические значения:

```
| > (andmap odd? (list 1 2 3 4 5))
| #false
| > (ormap odd? (list 1 2 3 4 5))
| #true
```

Вычисления такого вида называются *сверткой*.

Две функции в листинге 57 – `foldr` и `foldl` – чрезвычайно эффективны. Обе **сворачивают** списки в значения. Примеры вычислений объясняют абстрактные примеры в заголовках `foldr` и `foldl` с применением этих функций к $+$, θ и короткому списку. Как видите, `foldr` обрабатывает список справа налево, а `foldl` – слева направо. Для некоторых вычислений направление не имеет значения, но в общем случае это не так.

Функции, для которых порядок не имеет значения, в математике называются ассоциативными. Оператор $=$ в языке ISL является ассоциативным для целых чисел, но не для неточных. См. ниже.

Упражнение 256. Объясните следующую абстрактную функцию:

```
| ; [X] [X -> Number] [NEList-of X] -> X
| ; отыскивает (первый) элемент в lx, максимизирующий результат f
| ; if (argmax f (list x-1 ... x-n)) == x-i,
| ; then (>= (f x-i) (f x-1)), (>= (f x-i) (f x-2)), ...
| (define (argmax f lx) ...)
```

Проверьте ее работу на конкретных примерах в ISL. Попробуйте сформулировать описание назначения для похожей функции `argmin`. ■

Листинг 57. Абстрактные функции в ISL для обработки списков (2)

```
| ; [X Y] [X Y -> Y] Y [List-of X] -> Y
| ; применяет f справа налево к каждому элементу в lx и b
| ; (foldr f b (list x-1 ... x-n)) == (f x-1 ... (f x-n b))
| (define (foldr f b lx) ...)
|
| (foldr + 0 '(1 2 3 4 5))
```

```

== (+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0)))))  

== (+ 1 (+ 2 (+ 3 (+ 4 5))))  

== (+ 1 (+ 2 (+ 3 9)))  

== (+ 1 (+ 2 12))  

== (+ 1 14)

; [X Y] [X Y -> Y] Y [List-of X] -> Y  

; применяет f слева направо к каждому элементу в lx и b  

; (foldl f b (list x-1 ... x-n)) == (f x-n ... (f x-1 b))  

(define (foldl f b lx) ...)

(foldl + 0 '(1 2 3 4 5))  

== (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))  

== (+ 5 (+ 4 (+ 3 (+ 2 1))))  

== (+ 5 (+ 4 (+ 3 3)))  

== (+ 5 (+ 4 6))  

== (+ 5 10)

```

Листинг 58 иллюстрирует возможность композиции функций из листингов 56 и 57. Главная функция в нем – *listing*. Ее цель состоит в том, чтобы создать строку из списка экземпляров *address*. Описание назначения предполагает решение трех задач и, следовательно, создание трех функций:

- 1) первая извлекает имена из данного списка *Addr*;
- 2) вторая сортирует эти имена в алфавитном порядке;
- 3) третья объединяет строки, полученные на шаге 2;

Прежде чем продолжить чтение, попробуйте самостоятельно выполнить этот план. То есть спроектируйте все три функции, а затем объедините их для решения общей задачи, как описывалось в разделе 11.2.

Листинг 58. Программа, использующая абстракции

```

(define-struct address [first-name last-name street])
; Addr -- это структура:
;   (make-address String String String)
; интерпретация: объединяет адрес и имя человека

; [List-of Addr] -> String
; создает строку из имен,
; отсортированных в алфавитном порядке,
; разделяя их пробелами
(define (listing l)
  (foldr string-append-with-space " "
    (sort (map address-first-name l) string<?)))

; String String -> String
; объединяет две строки, добавляя " " в начало результата
(define (string-append-with-space s t)
  (string-append " " s t))

(define ex0
  (list (make-address "Robert" "Findler" "South")
        (make-address "Matthew" "Flatt" "Canyon")
        (make-address "Shriram" "Krishna" "Yellow")))

(check-expect (listing ex0) " Matthew Robert Shriram ")

```

В новом мире абстракций ту же задачу можно решить, спроектировав единственную функцию. Внимательно посмотрите на самое внутреннее выражение в *listing*:

```
| (map address-first-name l)
```

Согласно описанию назначения функции `map`, она применяет `address-first-name` к каждому экземпляру `address`, создавая список имен в виде строк. Его окружает выражение:

```
| (sort ... string<?>)
```

Многоточие представляет результат выражения с функцией `map`. Поскольку это выражение выдает список строк, то выражение с функцией `sort` создает отсортированный список имен. И наконец, самое внешнее выражение:

```
| (foldr string-append-with-space " " ...)
```

Оно сворачивает отсортированный список имен в единственную строку, применяя функцию с `string-append-with-space`. Такое говорящее имя функции позволяет понять, что свертка заключается в объединении всех строк желаемым образом, даже если вы не совсем понимаете, как работает комбинация `foldr` и `string-append-with-space`.

Упражнение 257. При желании можно спроектировать функции `build-list` и `foldl`, используя известные нам рецепты проектирования, но они не будут похожи на соответствующие функции в языке ISL. Например, проект нашей собственной функции `foldl` требует использования функции `reverse`:

```
; [X Y] [X Y -> Y] Y [List-of X] -> Y
; f*oldl действует подобно foldl
(check-expect (f*oldl cons '()) '(a b c))
  (foldl cons '() '(a b c)))
(check-expect (f*oldl / 1 '(6 3 2))
  (foldl / 1 '(6 3 2)))
(define (f*oldl f e l)
  (foldr f e (reverse l)))
```

Спроектируйте функцию `build-l*st`, которая работает так же, как `build-list`. **Подсказка.** Вспомните функцию `add-at-end` из упражнения 193. **Замечание о выборе дизайна.** В части VI описываются все понятия, необходимые для разработки таких функций с нуля. ■

16.2. Локальные определения

Давайте еще раз рассмотрим листинг 58. Функция `string-append-with-space` явно играет второстепенную роль и не имеет смысла вне этого узкого контекста. Кроме того, организация тела функции не отражает задач, указанных выше.

Почти все языки программирования поддерживают тот или иной способ обозначения подобных отношений. Идея заключается в создании локальных определений, также называемых *приватными определениями*. В языке ISL имеется инструкция `local`, позволяющая определять локальные функции, переменные и структуры.

Этот раздел знакомит с механикой `local`. В общем случае выражение `local` имеет следующую форму:

```
(local (def ...)
; --- IN ---
тело-выражения)
```

Вычисление такого выражения происходит как вычисление полной программы. Сначала производится анализ определений, которые могут включать вычисление правой части определения константы. Так же как в случае с определениями верхнего уровня, которые вы знаете и любите, определения в выражении `local` могут ссылаться друг на друга. Они также могут ссылаться на параметры вмещающей функции. Затем вычисляется тело-выражения, и его результат становится результатом выражения `local`. Часто бывает полезно отделить определения `local` от тела-выражения комментарием; мы можем использовать комментарий `---` `IN` `--`, потому что это слово предполагает, что определения доступны в определенном выражении.

Листинг 59. Определение функции с использованием выражения `local`

```
; [List-of Addr] -> String
; создает строку из имен,
; отсортированных в алфавитном порядке,
; разделяя их пробелами
(define (listing.v2 l)
  (local (; 1. извлечение имен
          (define names (map address-first-name l))
          ; 2. сортировка
          (define sorted (sort names string<?)))
         ; 3. объединение через пробелы
         ; String String -> String
         ; объединяет две строки, добавляя " " в начало
         (define (helper s t)
           (string-append " " s t))
         (define concat+spaces
           (foldr helper " " sorted)))
  concat+spaces))
```

В листинге 59 показан вариант реализации решения из листинга 58 с использованием `local`. Тело функции `listing.v2` теперь является выражением `local`, состоящим из двух частей: последовательности определений и тела выражения. Последовательность локальных определений выглядит точно так же, как в области определений DrRacket.

Поскольку имена видны только внутри выражения `local`, можно использовать более короткие имена.

В этом примере последовательность определений состоит из трех определений констант и одного определения функции. Каждое определение константы представляет

одну из трех запланированных задач. Определение функции – это переименованная версия `string-append-with-space`; она используется с `foldr` для реализации третьей задачи. Тело выражения `local` – это просто название третьей задачи.

Наибольшую привлекательность представляет организация в целом. Она ясно показывает, что функция решает три задачи и в каком порядке. По сути, этот пример демонстрирует общий принцип поддержки удобочитаемости:

Используйте `local` для переформулирования глубоко вложенных выражений. Используйте хорошо подобранные имена, чтобы выразить суть выражений.

Будущие читатели ценят удобочитаемый код, потому что могут понять его, глядя только на имена и тело выражения `local`.

Замечание об организации. Выражение `local` на самом деле является самым обычным выражением. Его можно использовать везде, где допустимо простое выражение. Следовательно, можно точно указать, где требуется вспомогательная функция. Рассмотрим следующий пример реорганизации выражения `local` в функции `listing.v2`:

```
... (local ((define names ...)
           (define sorted ...))
        (define concat+spaces
          (local ( ; String String -> String
                  (define (helper s t)
                    (string-append " " s t)))
                 (foldr helper " " sorted))))
        concat+spaces) ...
```

Новая версия состоит ровно из трех определений, предполагающих необходимость вычислений в три этапа. Третье определение состоит из выражения `local` в правой части, которое наглядно показывает, что вспомогательная функция необходима только при выполнении третьего шага.

Желание выразить отношения между частями программы с такой точностью зависит от двух ограничивающих факторов: языка программирования и ожидаемой продолжительности жизни кода. Некоторые языки даже не способны выразить идею о том, что функция `helper` необходима только на третьем шаге. С другой стороны, часто приходится учитывать время, выделенное для создания программы, и возможную необходимость вам или кому-то другому вернуться к ней позже и понять ее код. Команда Racket предпочитает перестраховываться, так как члены команды на собственном опыте знают, что ни одна программа никогда не будет завершена и все программы рано или поздно приходится исправлять. **Конец.**

Листинг 60. Организация взаимосвязанных определений функций с помощью `local`

```
| ; [List-of Number] [Number Number -> Boolean]
| ; -> [List-of Number]
```

```

; вычисляет версию alon0, отсортированную с помощью cmp
(define (sort-cmp alon0 cmp)
  (local (; [List-of Number] -> [List-of Number]
          ; вычисляет отсортированную версию alon
          (define (isort alon)
            (cond
              [(empty? alon) '()]
              [else
                (insert (first alon) (isort (rest alon)))]))

          ; Number [List-of Number] -> [List-of Number]
          ; вставляет n в отсортированный список чисел alon
          (define (insert n alon)
            (cond
              [(empty? alon) (cons n '())]
              [else (if (cmp n (first alon))
                        (cons n alon)
                        (cons (first alon)
                              (insert n (rest alon))))])))))
  (isort alon0)))

```

В листинге 60 представлен второй пример. Организация этого определения функции информирует читателя о том, что `sort-cmp` вызывает две вспомогательные функции: `isort` и `insert`. По локальности ясно видно, что слово «отсортированный» в описании назначения для `insert` ссылается на `isort`. Иначе говоря, `insert` представляет интерес только в этом контексте; программист не должен пытаться использовать ее где-нибудь вне контекста. Хотя это ограничение уже подчеркнуто в первоначальном определении функции `sort-cmp`, выражение `local` делает его частью программы.

Другой важный аспект этой реорганизации определения `sort-cmp` касается видимости `cmp`, второго параметра функции. Локальные функции могут ссылаться на `cmp`, потому что этот параметр доступен в контексте определений. Благодаря тому, что мы **не** передаем `cmp` из `isort` в `insert` (или обратно), читатель может сразу сделать вывод, что `cmp` остается неизменной на протяжении всего процесса сортировки.

Упражнение 258. Используйте выражение `local` для организации функций рисования многоугольника из листинга 38. Если функция, определенная глобально, может использоваться в разных местах, не делайте ее локальной. ■

Упражнение 259. Используйте выражение `local`, чтобы организовать функции перестановки слов из раздела 12.4. ■

Листинг 61. Использование local может улучшить производительность

```

; Nelon -> Number
; отыскивает наименьшее число в l
(define (inf.v2 l)
  (cond
    [(empty? (rest l)) (first l)]
    [else
      (local ((define smallest-in-rest (inf.v2 (rest l))))
        (if (< (first l) smallest-in-rest)
            (first l)
            smallest-in-rest))]))

```

Наш последний пример, демонстрирующий пользу выражений `local`, касается производительности. Рассмотрим определение функции `inf` в листинге 50. Она содержит две копии выражения

```
| (inf (rest l))
```

которое является естественной рекурсией во втором предложении в `cond`. В зависимости от результата условия выражение оценивается дважды. Использование `local` с целью создания имени для этого выражения улучшает читаемость кода, а также его производительность.

В листинге 61 показана измененная версия. Здесь выражение `local` находится в середине выражения `cond`. Оно определяет константу с результатом естественной рекурсии. Теперь напомним вам, что выражения `local` вычисляются один раз перед переходом к телу, то есть `(inf (rest l))` вычисляется только один раз, тогда как тело выражения `local` ссылается на результат дважды. Благодаря этому `local` экономит на повторном вычислении `(inf (rest l))` в каждой итерации.

Упражнение 260. Повторите эксперимент из упражнения 238, чтобы убедиться, что понимаете, почему `inf.v2` имеет более высокую производительность. ■

Упражнение 261. Исследуйте определение функции в листинге 62. Оба предложения во вложенном выражении `cond` извлекают первый элемент из `an-inv`, и оба вычисляют `(extract1 (rest an-inv))`. Используйте `local`, чтобы дать имя этому выражению. Помогло ли это увеличить скорость вычисления результата? Существенно? Совсем немного? Нисколько? ■

Листинг 62. Функция инвентаризации, см. упражнение 261

```
; Inventory -> Inventory
; создает экземпляр Inventory из an-inv, включая все
; элементы, стоимость которых меньше доллара
(define (extract1 an-inv)
  (cond
    [(empty? an-inv) '()]
    [else
      (cond
        [(<= (ir-price (first an-inv)) 1.0)
         (cons (first an-inv) (extract1 (rest an-inv)))]
        [else (extract1 (rest an-inv))])]))
```

16.3. Локальные определения добавляют выразительности

Третий и последний пример показывает, как `local` добавляет выразительности в BSL и BSL+. В разделе 12.8 представлен проект мировой программы, моделирующей работу конечного автомата, распознавающего последовательности нажатий клавиш. Анализ информации естественным образом приводит к определениям данных в листин-

ге 45, однако попытка спроектировать главную функцию мировой программы терпит неудачу. В частности, даже притом что данный конечный автомат остается неизменным в ходе выполнения программы, состояние мира должно включать его, чтобы программа могла переходить из одного состояния в другое, когда игрок нажимает клавишу.

В листинге 63 показано решение этой проблемы на ISL. В нем используются определения локальных функций, и, таким образом, состояние мира можно приравнять к текущему состоянию конечного автомата. В частности, `simulate` определяет локальный обработчик события клавиатуры, который принимает только текущее состояние мира и событие `KeyEvent`, представляющее нажатую клавишу. Эта локальная функция может ссылаться на заданный конечный автомат `fsm` и, соответственно, имеет возможность определить следующее состояние по таблице переходов, даже если эта таблица не является аргументом функции.

Листинг 63. Выразительность локальных функций

```
; FSM FSM-State -> FSM-State
; сопоставляет нажатые клавиши с заданным конечным автоматом FSM
(define (simulate fsm s0)
  (local (; состояние мира: FSM-State
          ; FSM-State KeyEvent -> FSM-State
          (define (find-next-state s key-event)
            (find fsm s)))
    (big-bang s0
      [to-draw state-as-colored-square]
      [on-key find-next-state])))

; FSM-State -> Image
; отображает текущее состояние мира в виде цветного квадрата
(define (state-as-colored-square s)
  (square 100 "solid" s))

; FSM FSM-State -> FSM-State
; определяет следующее состояние в fsm
(define (find transitions current)
  (cond
    [(empty? transitions) (error "not found")]
    [else
      (local ((define s (first transitions)))
        (if (state=? (transition-current s) current)
            (transition-next s)
            (find (rest transitions) current))))]))
```

Все остальные функции определяются на одном уровне с главной функцией, в том числе и функция `find`, которая выполняет поиск в таблице переходов. Ключевым улучшением по сравнению с BSL является способность локальной функции ссылаться как на параметры вмещающей функции, так и на глобальные вспомогательные функции.

Проще говоря, такая организация программы сообщает будущему читателю именно те идеи, которые мы нашли на этапе анализа данных в рецепте проектирования мировых программ. Во-первых,

данное представление конечного автомата остается неизменным. Во-вторых, в ходе выполнения изменяется текущее состояние конечного автомата.

Этот пример наглядно показывает, как выбранный язык программирования влияет на способность программиста выражать решения, а также на способность будущего читателя распознавать идеи создателя.

Упражнение 262. Спроектируйте функцию `identityM`, которая создает квадратные таблицы с единицами на главной диагонали и с нулями во всех остальных ячейках:

В линейной алгебре такие таблицы называют единичными матрицами.

```
| > (identityM 1)
| (list (list 1))
| > (identityM 3)
| (list (list 1 0 0) (list 0 1 0) (list 0 0 1))
```

Используйте рецепт проектирования структур и добавьте выразительности с помощью `local`. ■

16.4. Вычисления с локальными определениями

Выражение `local` в языке ISL требует применения первого правила вычислений, которое выходит за рамки начального уровня знаний. Правило это относительно простое, но довольно необычное. Лучше всего проиллюстрировать его несколькими примерами. Для начала рассмотрим еще раз следующее определение:

```
| (define (simulate fsm s0)
|   (local ((define (find-next-state s key-event)
|             (find fsm s)))
|     (big-bang s0
|       [to-draw state-as-colored-square]
|       [on-key find-next-state])))
```

Теперь предположим, что мы решили посмотреть, что вычислит DrRacket для следующего выражения:

```
| (simulate AN-FSM A-STATE)
```

где `AN-FSM` и `A-STATE` – неизвестные значения. Используем обычное правило подстановки:

```
| ==
| (local ((define (find-next-state s key-event)
|             (find AN-FSM s)))
|   (big-bang A-STATE
|     [to-draw state-as-colored-square]
|     [on-key find-next-state])))
```

Это тело `simulate`, в котором все вхождения `fsm` и `s` заменены значениями аргументов `AN-FSM` и `A-STATE` соответственно.

И тут мы застряли, потому что выражение является локальным, и мы не знаем, как производить вычисления с его помощью. Итак, встретив выражение `local` при оценке программы, мы должны выполнить два шага:

- 1) переименовать локальные константы и функции, используя имена, которые нигде в программе больше не встречаются;
- 2) вынести определения из выражения `local` на верхний уровень, а затем вычислить тело локального выражения.

Стоп! Не стоит пока глубоко задумываться, просто примите к сведению эти два шага.

Давайте поочередно применим эти шаги к нашему примеру:

```
==  
(local ((define (find-next-state-1 s key-event)  
          (find an-fsm a-state))  
       (big-bang s0  
                 [to-draw state-as-colored-square]  
                 [on-key find-next-state-1]))
```

Мы просто добавили «`-1`» в конец имени функции. Если такое имя уже существует, то мы добавили бы «`-2`» и т. д. Итак, вот результат шага 2:

Мы использовали символ
 \oplus , чтобы показать,
что этот шаг создает
два фрагмента кода.

```
==  
(define (find-next-state-1 s key-event)  
        (find an-fsm a-state))  
 $\oplus$   
(big-bang s0  
          [to-draw state-as-colored-square]  
          [on-key find-next-state-1])
```

В результате получилась самая обычная программа: с несколькими глобальными константами и функциями, за которыми следует выражение. В этом случае действуют обычные правила, и нам больше нечего сказать.

Теперь займемся рационализацией этих двух шагов. На этапе переименования мы используем вариант функции `inf` из листинга 61. Ясно, что

```
| (inf (list 2 1 3)) == 1
```

Вопрос в том, сможете ли вы показать, какие вычисления выполняет DrRacket, чтобы получить этот результат.

Первый шаг прост:

```
(inf (list 2 1 3))  
==  
(cond  
  [(empty? (rest (list 2 1 3)))  
   (first (list 2 1 3))]  
  [else  
   (local ((define smallest-in-rest
```

```
(inf (rest (list 2 1 3))))
(cond
  [(< (first (list 2 1 3)) smallest-in-rest)
   (first (list 2 1 3))]
  [else smallest-in-rest]))]
```

Мы подставили `(list 2 1 3)` вместо `l`.

Поскольку список не пустой, мы пропускаем оценку условного выражения и сосредоточиваемся на следующем выражении, которое нужно вычислить:

```
...
==

(local ((define smallest-in-rest
            (inf (rest (list 2 1 3)))))
  (cond
    [(< (first (list 2 1 3)) smallest-in-rest)
     (first (list 2 1 3))]
    [else smallest-in-rest]))
```

Применение двух шагов из правила вычисления локальных выражений дает нам два фрагмента кода: локальное определение, поднятое на самый верх, и тело выражения `local`. Вот как мы это записываем:

```
==

(define smallest-in-rest-1
  (inf (rest (list 2 1 3))))
⊕
(cond
  [(< (first (list 2 1 3)) smallest-in-rest-1)
   (first (list 2 1 3))]
  [else smallest-in-rest-1])
```

Интересно отметить, что следующее выражение, которое нужно вычислить, – это правая часть определения константы в выражении `local`. Но суть процесса вычисления в том, что мы можем заменять где угодно выражения их эквивалентами:

```
==

(define smallest-in-rest-1
  (cond
    [(empty? (rest (list 1 3))) (first (list 1 3))]
    [else
      (local ((define smallest-in-rest
                  (inf (rest (list 1 3)))))
        (cond
          [(< (first (list 1 3)) smallest-in-rest)
           (first (list 1 3))]
          [else smallest-in-rest])))])
⊕
(cond
  [(< (first (list 2 1 3)) smallest-in-rest-1)
   (first (list 2 1 3))]
  [else smallest-in-rest-1])
```

И снова мы пропускаем оценку условия и сосредоточиваемся на предложении `else`, которое также является выражением `local`. На са-

мом деле это еще один вариант выражения `local` в определении `inf`, с другим значением списка, которое мы подставили вместо параметра:

```
(define smallest-in-rest-1
  (local ((define smallest-in-rest
            (inf (rest (list 1 3)))))
         (cond
           [(< (first (list 1 3)) smallest-in-rest)
            (first (list 1 3))]
           [else smallest-in-rest]))
  ⊕
  (cond
    [(< (first (list 2 1 3)) smallest-in-rest-3)
     (first (list 2 1 3))]
    [else smallest-in-rest-3]))
```

Поскольку оно исходит из того же выражения `local` в `inf`, для него используется то же имя константы `smallest-in-rest`. **Если бы мы не переименовали локальные определения до их переноса на верхний уровень, то создали бы два противоречащих друг другу определения с одним и тем же именем**, а конфликтующие определения катастрофичны для математических расчетов.

Продолжаем:

```
===
(define smallest-in-rest-2
  (inf (rest (list 1 3))))
⊕
(define smallest-in-rest-2
  (cond
    [(< (first (list 1 3)) smallest-in-rest-2)
     (first (list 1 3))]
    [else smallest-in-rest-2]))
⊕
(define smallest-in-rest-2
  (cond
    [(< (first (list 2 1 3)) smallest-in-rest-2)
     (first (list 2 1 3))]
    [else smallest-in-rest-2]))
```

Суть в том, что теперь у нас есть **два** определения, генерированных **из одного и того же** выражения `local` в определении функции. Фактически мы получаем по одному такому определению для каждого элемента в заданном списке (минус 1).

Упражнение 263. Используйте движок пошаговых вычислений в DrRacket и подробно изучите этапы этих вычислений. ■

Упражнение 264. Используйте движок пошаговых вычислений в DrRacket и исследуйте вычисление следующего выражения:

```
| (sup (list 2 1 3))
```

где `sup` – это функция из листинга 50 с выражением `local`. ■

Чтобы пояснить шаг переноса объявлений на верхний уровень, используем простой пример, раскрывающий суть проблемы, а именно тот факт, что функции теперь являются значениями:

```
| ((local ((define (f x) (+ (* 4 (sqrt x)) 3))) f)
  1)
```

В глубине души мы знаем, что это выражение эквивалентно выражению `(f 1)`, где

```
| (define (f x) (+ (* 4 (sqrt x)) 3))
```

но базовые алгебраические правила здесь не действуют. Дело в том, что **функции могут быть результатами выражений, включая выражения local**. Поэтому лучше переместить такие локальные определения на верхний уровень и обращаться с ними как с обычными определениями. Такое перемещение делает определение видимым на каждом этапе вычислений. Теперь вы уже знаете, что этап переименования гарантирует невозможность появления после переноса случайных конфликтов с существующими именами определений.

Вот первые два шага:

```
| ((local ((define (f x) (+ (* 4 (sqrt x)) 3))) f)
  1)
 ==
| ((local ((define (f-1 x) (+ (* 4 (sqrt x)) 3)))
  f-1)
  1)
 ==
| (define (f-1 x) (+ (* 4 (sqrt x)) 3))
 ⊕
(f-1 1)
```

Второй шаг правила вычисления локальных выражений, как вы помните, заключается в замене выражения `local` его телом. В данном случае тело – это просто имя функции, а его окружение – применение к 1. Остальное просто:

```
| (f-1 1) == (+ (* 4 (sqrt 1)) 3) == 7
```

Упражнение 265. Используйте движок пошаговых вычислений в DrRacket, чтобы заполнить все пропущенные выше шаги. ■

Упражнение 266. Используйте движок пошаговых вычислений в DrRacket, чтобы узнать, как ISL вычисляет

```
| ((local ((define (f x) (+ x 3))
         (define (g x) (* x 4)))
    (if (odd? (f (g 1)))
        f
        g))
  2)
```

и получает 5. ■

16.5. Использование абстракций на примерах

Теперь, когда вы узнали, как вычисляются выражения `local`, вы сможете использовать абстракции из листингов 56 и 57. Давайте рассмотрим примеры, начав со следующего:

Задача. Спроектируйте `add-3-to-all`. Функция должна принимать список экземпляров `Posn` и прибавлять число 3 к координате X каждого из них.

Если следовать рецепту проектирования и принять постановку задачи за описание назначения, то мы быстро выполним первые три шага:

```
| ; [List-of Posn] -> [List-of Posn]
| ; прибавляет 3 к координате X каждого экземпляра Posn в заданном списке
|
| (check-expect
|   (add-3-to-all
|     (list (make-posn 3 1) (make-posn 0 0)))
|     (list (make-posn 6 1) (make-posn 3 0)))
|
|   (define (add-3-to-all lop) '())
```

Запустив программу, мы получим сообщение об ошибке в одном тесте, потому что функция возвращает значение по умолчанию `'()`.

На этом этапе мы останавливаемся и спрашиваем себя, с какой функцией имеем дело. Очевидно, что `add-3-to-all` – это функция обработки списка. Вопрос в том, похожа ли она на какую-либо из функций в листингах 56 и 57. Сигнатура говорит нам, что `add-3-to-all` – это функция обработки списка, которая принимает и порождает список. В двух упомянутых листингах есть несколько функций с похожими сигнатурами: `map`, `filter` и `sort`.

Описание назначения и пример дополнительно сообщают нам, что `add-3-to-all` обрабатывает каждый экземпляр `Posn` отдельно и собирает результаты в единый список. Очевидно также, что список в результате содержит столько же элементов, сколько исходный список. Все эти размышления указывают на одну функцию – `map`, – потому что идея функции `filter` состоит в том, чтобы отбрасывать элементы из списка, а `sort` имеет чрезвычайно конкретную цель.

Вот сигнатура функции `map`:

```
| ; [X Y] [X -> Y] [List-of X] -> [List-of Y]
```

Он сообщает, что `map` принимает функцию, преобразующую X в Y, и список X. Учитывая, что `add-3-to-all` принимает список экземпляров `Posn`, мы знаем, что X – это `Posn`. Точно так же мы знаем, что `add-3-to-all` создает список `Posn`, а это означает, что мы должны заменить Y на `Posn`.

По результатам анализа сигнатуры мы приходим к выводу, что `map` может решить задачу, возложенную на `add-3-to-all`, если передать ей правильную функцию преобразования `Posn` в `Posn`. Мы можем выразить эту идею с помощью `local` в форме макета для `add-3-to-all`:

```
| (define (add-3-to-all lop)
|   (local (; Posn -> Posn
|         ; ...
```

```
(define (fp p)
  ... p ...))
(map fp lop))
```

В результате задача сводится к определению функции преобразования Posn.

Учитывая пример для add-3-to-all и абстрактный пример для map, мы можем представить, как происходит вычисление:

```
(add-3-to-all (list (make-posn 3 1) (make-posn 0 0)))
== 
(map fp (list (make-posn 3 1) (make-posn 0 0)))
== 
(list (fp (make-posn 3 1)) (fp (make-posn 0 0))))
```

Здесь можно видеть, как fp применяется к каждому экземпляру Posn в данном списке, то есть ее задача – прибавить 3 к координате X.

Теперь мы с легкостью можем завершить определение:

```
(define (add-3-to-all lop)
  (local (; Posn -> Posn
          ; прибавляет 3 к координате X в p
          (define (add-3-to-1 p)
            (make-posn (+ (posn-x p) 3) (posn-y p))))
        (map add-3-to-1 lop)))
```

Мы выбрали для локальной функции имя add-3-to-1, потому что оно четко сообщает, что делает эта функция. Она прибавляет 3 к координате X в экземпляре Posn.

У вас может сложиться впечатление, что пользоваться абстракциями сложно. Однако имейте в виду, что этот первый пример подробно описывает каждую деталь, потому что мы хотим научить вас правильно выбирать абстракции. Давайте рассмотрим второй пример, но на этот раз пойдем вперед немного быстрее.

Задача. Спроектируйте функцию, которая удаляет из некоторого заданного списка все экземпляры Posn с координатой Y больше 100.

Первые два шага рецепта проектирования дают:

```
; [List-of Posn] -> [List-of Posn]
; удаляет экземпляры Posn с координатой y > 100

(check-expect
  (keep-good (list (make-posn 0 110) (make-posn 0 60)))
  (list (make-posn 0 60)))

(define (keep-good lop) '())
```

Вы наверняка уже догадались, что эта функция похожа на функцию filter, цель которой – отделить «хорошее» от «плохого».

Благодаря local следующий шаг выглядит просто:

```
(define (keep-good lop)
  (local (; Posn -> Boolean
          ; должен ли этот экземпляр Posn остаться в списке
          (define (good? p) #true))
    (filter good? lop)))
```

Выражение `local` определяет локальную вспомогательную функцию, необходимую функции `filter`, а его тело применяет `filter` к этой локальной функции и данному списку. Локальная функция называется `good?`, потому что `filter` сохраняет все элементы, для которых `good?` дает `#true`.

Прежде чем читать дальше, проанализируйте сигнатуру `filter` и `keep-good` и определите, почему вспомогательная функция принимает отдельные экземпляры `Posn` и возвращает логические значения.

Объединив все наши идеи, получаем следующее определение:

```
(define (keep-good lop)
  (local (; Posn -> Posn
          ; должен ли этот экземпляр Posn остаться в списке
          (define (good? p)
            (not (> (posn-y p) 100))))
    (filter good? lop)))
```

Объясните, как работает `good?`, и упростите ее.

Прежде чем изложить рецепт проектирования, рассмотрим еще один пример.

Задача. Спроектируйте функцию, которая определяет, находится ли какая-нибудь точка из списка экземпляров `Posn` поблизости от заданной точки `pt`, где «поблизости» означает «на расстоянии не более 5 пикселей».

Эта задача явно состоит из двух частей: обработки списка и применения функции, которая определяет, находится ли некоторая точкой «поблизости» от заданной. Поскольку вторая часть не связана с повторным использованием абстракций для списков, то будем предполагать, что существует соответствующая функция:

```
; Posn Posn Number -> Boolean
; расстояние между p и q меньше d
(define (close-to p q d) ...)
```

Дополните это определение самостоятельно.

В соответствии с постановкой задачи функция принимает два аргумента: список экземпляров `Posn` и «заданную» точку `pt` и в результате дает логическое значение:

```
; [List-of Posn] Posn -> Boolean
; имеется ли в lop точка Posn, близкая к pt

(check-expect
  (close? (list (make-posn 47 54) (make-posn 0 60))
           (make-posn 50 50)))
```

```
| #true)
| (define (close? lop pt) #false)
```

Сигнатура отличает этот пример от предыдущих.

Диапазон логических значений подсказывает нам выбор из функций в листингах 56 и 57. Только две функции в этом списке дают логические значения – `andmap` и `ormap`, – и они являются основными кандидатами на использование в теле `close?`. В описании `andmap` говорится, что все элементы списка должны обладать определенным свойством, а описание `ormap` сообщает, что эта функция пытается найти только один такой элемент. Учитывая это, `close?` просто проверяет близость одного из экземпляров `Posn` к `pt`, мы должны сначала попробовать `ormap`.

Применим наш стандартный «трюк» с выражением `local`, тело которого использует выбранную абстракцию для применения к некоторой локальной функции и заданного списка:

```
(define (close? lop pt)
  (local (; Posn -> Boolean
          ; ...
          (define (is-one-close? p)
            ...))
    (ormap close-to? lop)))
```

Согласно описанию `ormap`, локальная функция принимает элементы списка по одному. Это объясняет наличие `Posn` в ее сигнатуре. Также ожидается, что локальная функция вычисляется в `#true` или `#false`, а `ormap` проверяет эти результаты, пока не встретит `#true`.

Сравним сигнатуры `ormap` и `close?`, начав с первой:

```
| ; [X] [X -> Boolean] [List-of X] -> Boolean
```

В нашем случае аргумент со списком представляет список экземпляров `Posn`. Следовательно, `X` означает `Posn`, что объясняет тип аргумента, принимаемого функцией `is-one-close?`. Кроме того, сигнатура указывает, что результат локальной функции должен быть логическим значением, чтобы ее можно было передать в первом аргументе функции `ormap`.

Далее требуется немного поразмышлять. Функция `is-one-close?` принимает один аргумент – `Posn`, а функция `close-to` – три: два экземпляра `Posn` и величину «допуска». Аргумент `is-one-close?` является одним из двух экземпляров `Posn`. Также очевидно, что другой экземпляр `Posn` является заданной точкой `pt`, которая сама передается в `close?` как аргумент. Естественно, аргумент «допуска» равен 5, как указано в задаче:

```
(define (close? lop pt)
  (local (; Posn -> Boolean
          ; данная точка находится поблизости от pt
          (define (is-one-close? p)
            (close-to p pt CLOSENESS))))
```

```
(ormap is-one-close? lop)))  
(define CLOSENESS 5) ; в пикселях
```

Обратите внимание на две особенности этого определения. Во-первых, мы придерживаемся правила, требующего создавать определения для констант. Во-вторых, ссылка на `pt` в `is-one-close?` сигнализирует, что мы используем один и тот же экземпляр `Posn` на протяжении обхода всего списка `lop`.

16.6. Проектирование с использованием абстракций

Трех примеров, которые мы рассмотрели в предыдущем разделе, вполне достаточно, чтобы сформулировать рецепт проектирования.

- Шаг 1: выполнить три первых шага из рецепта проектирования функций. В частности, опираясь на постановку задачи, необходимо сформулировать сигнатуру, описание назначения, примеры и заготовку определения.

Рассмотрим задачу определения функции, которая помещает маленькие красные кружки на холст размером 200×200 , используя координаты из заданного списка экземпляров `Posn`. Вот что дают первые три шага рецепта проектирования:

```
; [List-of Posn] -> Image  
; добавляет Posn из lop в пустую сцену  
  
(check-expect (dots (list (make-posn 12 31)))  
              (place-image DOT 12 31 MT-SCENE))  
  
(define (dots lop)  
  MT-SCENE)
```

Добавьте определения констант, чтобы этот код можно было запустить в DrRacket.

- Затем, используя сигнатуру и описание назначения, нужно найти подходящую абстракцию. Под «подходящей» понимается такая абстракция, назначение которой включает в себя назначение проектируемой функции; это также означает, что их сигнатуры должны соответствовать друг другу. Часто лучше начать с желаемого результата и затем найти абстракцию, которая дает такой же или более общий результат.

В текущем примере желаемый результат – изображение. Ни одна из доступных абстракций не создает изображений, но две из них имеют переменную справа:

```
; foldr : [X Y] [X Y -> Y] Y [List-of X] -> Y  
; foldl : [X Y] [X Y -> Y] Y [List-of X] -> Y
```

А это означает, что с их помощью можно обрабатывать любые коллекции данных и получать результат в желаемом виде. Подстановка `Image` в сигнатуру слева от `->` требует определить вспомогательную функцию, которая принимает `X` вместе с `Image` и создает `Image`. Кроме того, поскольку данный список содержит экземпляры `Posn`, то `X` обозначает коллекцию `Posn`.

3. Написать **макет**. Макет, повторно использующий абстракции, включает выражение `local`, преследуя две разные цели. Первая – показать, какая абстракция используется и как. Вторая – определить заготовку вспомогательной функции: ее сигнатуру, описание назначения (необязательно) и заголовок. Обратите внимание, что большую часть сигнатуры вспомогательной функции нам помогло определить сравнение сигнатур на предыдущем шаге.

Вот как выглядит макет для текущего примера, если выбрать функцию `foldr`:

```
(define (dots lop)
  (local (; Posn Image -> Image
          (define (add-one-dot p scene) ...))
        (foldr add-one-dot MT-SCENE lop)))
```

Согласно описанию `foldr`, мы должны передать ей изображение для «базового случая», которое будет использоваться, если список окажется пустым. В нашем примере базовым случаем является пустой холст. В противном случае `foldr` выполнит обход экземпляров `Posn` в списке и применит к каждому вспомогательную функцию.

4. После этого можно определить вспомогательную функцию внутри `local`. В большинстве случаев такие функции принимают относительно простые типы данных и подобны функциям, описанным в части I. Вы уже знаете, как их проектировать. Разница лишь в том, что теперь можно использовать не только аргументы самой функции и глобальные константы, но также аргументы вмещающей функции.

Цель вспомогательной функции в нашем примере – добавить изображение одной точки в заданную сцену, которое вы можете (1) определить сами или (2) получить из примера:

```
(define (dots lop)
  (local (; Posn Image -> Image
          ; добавляет DOT в позицию p в сцену scene
          (define (add-one-dot p scene)
            (place-image DOT
                          (posn-x p) (posn-y p)
                          scene)))
        (foldr add-one-dot MT-SCENE lop)))
```

5. Последний шаг – протестировать определение обычным способом.

Для абстрактных функций иногда можно использовать абстрактный пример из описания назначения, чтобы подтвердить правильную работу на общем уровне. То есть в данном случае можно использовать абстрактный пример для `foldr`, чтобы убедиться, что точки из `dots` добавляются одна за другой в исходную сцену.

На третьем шаге мы без рассуждений выбрали `foldr`. Поэкспериментируйте с `foldl` и посмотрите, как решить задачу с ее использованием. Такие функции, как `foldl` и `foldr`, хорошо известны и часто используются в различных формах. Будет неплохим познакомиться с ними поближе, чем мы и займемся в следующих двух разделах.

16.7. Практические упражнения: абстракция

Упражнение 267. Используйте `map` и определите функцию `convertEuro`, которая преобразует список денежных сумм в долларах США в список денежных сумм в евро, взяв за основу обменный курс 1,16 доллара США за евро (на октябрь 2021 года).

Также используйте `map` и определите функцию `convertFC`, которая преобразует список температур по Фаренгейту в список температур по Цельсию.

Наконец, попробуйте определить функцию `translate`, которая преобразует список экземпляров `Posn` в список списков пар чисел. ■

Упражнение 268. В инвентарной записи указывается название товара, описание, закупочная цена и рекомендованная розничная цена.

Определите функцию, которая сортирует список записей по разнице между двумя этими ценами. ■

Упражнение 269. Определите функцию `eliminate-expensive`. Она должна принимать число `ia` и список инвентарных записей и создавать список с записями, в которых розничная цена ниже `ia`.

Затем, используя `filter`, определите функцию `recall`, которая принимает параметр `ty` с названием товара и список инвентарных записей и создает список записей, названия товаров в которых отличаются от `ty`.

Наконец, определите функцию `selection`, которая принимает два списка с названиями и выбирает из второго списка названия, также присутствующие в первом списке. ■

Упражнение 270. Используйте `build-list`, чтобы определить функцию, которая

- 1) создает список (`list 0 ... (- n 1)`) для любого натурального числа `n`;
- 2) создает список (`list 1 ... n`) для любого натурального числа `n`;
- 3) создает список (`list 1 1/2 ... 1/n`) для любого натурального числа `n`;
- 4) создает список первых `n` четных чисел;

- 5) создает диагональную матрицу из нулей и единиц; см. упражнение 262.

Затем, используя `build-list`, определите функцию `tabulate` из упражнения 250. ■

Упражнение 271. Используйте `ogmap`, чтобы определить функцию `find-name`, которая должна принимать имя и список имен и определять, имеются ли в списке имена, целиком совпадающие с указанным или содержащие его.

Используйте `andmap`, чтобы определить функцию, которая проверяет, все ли имена в списке начинаются с буквы "а".

Какую функцию следует использовать – `ogmap` или `andmap`, – чтобы определить функцию, которая гарантирует, что ни одно имя в некотором списке не длиннее заданного числа? ■

Упражнение 272. Функция `append` в языке ISL соединяет два списка в один, или, что то же самое, заменяет '()' в конце первого списка вторым списком:

```
| (equal? (append (list 1 2 3) (list 4 5 6 7 8))
         (list 1 2 3 4 5 6 7 8))
```

Используйте `foldr`, чтобы определить `append-from-fold`. Что произойдет, если заменить `foldr` на `foldl`?

Теперь используйте одну из функций свертки, чтобы определить функции, которые вычисляют сумму и произведение для списка чисел.

Используйте одну из функций свертки, чтобы определить функцию, которая добавляет в сцену изображения из списка, располагая их по горизонтали. **Подсказки.** (1) Загляните в описание функций `beside` и `empty-image`. Попробуйте решить ту же задачу с помощью другой функции свертки. Определите также функцию, которая располагает изображения из списка по вертикали. (2) Загляните в описание функции `above`. ■

Упражнение 273. Функции свертки настолько универсальные, что с их помощью можно определить практически любую функцию обработки списков. Используйте `fold`, чтобы определить функцию `mapvia-fold`, имитирующую поведение функции `map`. ■

Упражнение 274. Используйте существующие абстракции, чтобы определить функции `rgfixes` и `suffixes` из упражнения 190. Убедитесь, что они успешно проходят те же тесты, что и исходная функция. ■

16.8. Проекты: абстракция

Теперь, получив некоторый опыт использования существующих абстракций для обработки списков, вам предстоит заняться переделкой некоторых уже реализованных проектов. Ваша задача – найти два способа улучшения программ. Во-первых, найдите в своих программах функции, обходящие списки. Для этих функций у вас уже есть

сигнатуры, описания назначения, тесты и действующие определения, удовлетворяющие тестам. Измените определения так, чтобы в них использовались абстракции из листингов 56 и 57. Во-вторых, определите, можно ли создать новые абстракции. Вы можете абстрагировать эти классы программ и реализовать обобщенные функции, которые помогут вам писать другие программы.

Возможно, вы захотите решить эти упражнения еще раз после изучения главы 17.

Упражнение 275. В разделе 12.1 рассматриваются относительно простые задачи, связанные со словарем английского языка. При проектировании двух из них существующие абстракции сами напрашиваются:

- спроектируйте функцию `most-frequent`, которая принимает словарь и возвращает экземпляр `Letter-Count` с буквой, с которой начинается больше всего слов в данном словаре;
- спроектируйте функцию `words-by-first-letter`, которая принимает словарь и возвращает список словарей, по одному для каждой буквы. Не нужно включать в результат '()', если для какой-то буквы нет слов; вместо этого просто игнорируйте ее.

См. определения данных в листинге 39. ■

Упражнение 276. В разделе 12.2 рассказывается, как анализировать информацию в XML-медиатеке iTunes:

- спроектируйте функцию `select-album-date`, которая принимает название альбома, дату и `LTracks` и извлекает из `LTracks` список треков, принадлежащих данному альбому и воспроизведенных после указанной даты;
- спроектируйте функцию `select-albums`, которая принимает экземпляр `LTracks` и создает список экземпляров `LTracks`, по одному для каждого альбома. Альбомы должны уникально идентифицироваться по названию и присутствовать в результатах только один раз.

См. определения данных и функции в листинге 42, предоставляемые библиотекой `2htdp/itunes`. ■

Упражнение 277. В разделе 12.7 описывается игра «Космические захватчики». В базовой версии сверху спускается НЛО, а игрок пытается сбить его, стреляя из танка. Мы предлагаем оснастить НЛО бомбами, которые он может сбрасывать на танк; танк должен уничтожаться, если бомба оказывается достаточно близко к танку.

Изучите код проекта и выявите все места, где можно извлечь выгоду от использования существующих абстракций для обработки списков ракет и бомб.

Затем, когда вы упростите код с помощью существующих абстракций, найдите возможность для определения новых абстракций. Например, можно попробовать абстрагировать перемещение списков объектов. ■

Упражнение 278. В разделе 12.5 рассказывается о еще одной из старейших компьютерных игр. В этой игре имеется питон, который

движется с постоянной скоростью в направлении, выбирамом игроком. Достигнув яблока, он съедает его и вырастает в размерах. Если голова питона достигает края сцены или одного из сегментов своего тела, то игра заканчивается.

Этот проект тоже может извлечь выгоду из абстрактных функций обработки списков, имеющихся в языке ISL. Найдите, где можно их использовать, и измените существующий код. Тесты помогут вам гарантировать отсутствие ошибок. ■

17. Безымянные функции

При использовании абстрактных функций бывает необходимо передавать другие функции в аргументах. Иногда это существующие элементарные функции, библиотечные функции или ваши собственные:

- (`(build-list n add1)` создает `(list 1 ... n)`);
- (`(foldr cons another-list a-list)` объединяет списки `a-list` и `another-list`;
- (`(foldr above empty-image images)` располагает изображения из списка по вертикали).

А иногда достаточно определить простую вспомогательную функцию, определение которой состоит из одной строки. Рассмотрим пример использования функции `filter`:

```
; [List-of IR] Number -> Boolean
(define (find l th)
  (local (; IR -> Boolean
          (define (acceptable? ir)
            (<= (ir-price ir) th)))
    (filter acceptable? l)))
```

В DrRacket выберите пункт **Choose language** (Выбрать язык...) в меню **Language** (Язык), после чего в открывшемся диалоге выберите пункт **Intermediate student with lambda** (Средний студент с лямбда-выражениями). История лямбда-выражений тесно связана с ранней историей проектирования языков программирования.

главе мы разберем эту концепцию: ее синтаксис, значение и практическое применение. С помощью лямбда-выражений приведенное выше определение можно выразить в одной строке:

```
; [List-of IR] Number -> Boolean
(define (find l th)
  (filter (lambda (ir) (<= (ir-price ir) th)) l))
```

Первые два раздела посвящены механике лямбда-выражений; остальные демонстрируют их применение для создания экземпляров абстракций, тестирования и определения, а также для представления бесконечных данных.

Она отыскивает все товары в инвентарном списке, цена которых ниже `th`. Вспомогательная функция почти тривиальна, но ее определение занимает три строки.

Эта ситуация требует улучшений в языке. Программисты должны иметь возможность определять такие маленькие и незначительные функции без особых усилий. Следующий уровень в нашей иерархии обучающих языков – язык для студентов со средним уровнем подготовки, поддерживающий лямбда-выражения (*Intermediate Student Language with lambda*), – решает проблему с помощью нового понятия – безымянных функций. В этой

17.1. Определение функций с помощью лямбда-выражений

Лямбда-выражения имеют простой синтаксис:

```
| (lambda (переменная-1 ... переменная-N) выражение)
```

Его отличительной особенностью является ключевое слово `lambda`. За ключевым словом следует последовательность переменных, заключенная в пару круглых скобок. Последний элемент – произвольное выражение, вычисляющее результат функции с использованием значений ее параметров.

Вот три простых примера, каждый из которых принимает один аргумент:

- 1) `(lambda (x) (expt 10 x))` предполагает, что аргумент является числом, и вычисляет 10 в степени x;
- 2) `(lambda (n) (string-append "To " n ","))` использует заданную строку, чтобы сгенерировать адреса с помощью `string-append`;
- 3) `(lambda (ir) (<= (ir-price ir) th))` принимает структуру IR, извлекает из нее цену и сравнивает ее с th.

Лямбда-выражения можно рассматривать как более короткую форму записи выражений `local`. Например:

```
| (lambda (x) (* 10 x))
```

фактически является сокращенной формой записи для

```
| (local ((define (some-name x) (* 10 x))) some-name)
```

Такой взгляд на
лямбда-выражения
лишний раз показывает
сложность правил
вычислений
с выражением `local`.

Такая подмена допустима, только если в теле функции отсутствует ссылка на `some-name`. Это означает, что лямбда-выражение создает функцию с именем, которого никто не знает. Если никто не знает имени, то с таким же успехом функция может быть безымянной.

Чтобы применить функцию, созданную лямбда-выражением, ей нужно передать правильное количество аргументов. Например:

```
> ((lambda (x) (expt 10 x)) 2)
100
> ((lambda (name rst) (string-append name ", " rst))
  "Robby"
  "etc.")
"Robby, etc."
> ((lambda (ir) (<= (ir-price ir) th))
  (make-ir "bear" 10))
#true
```

Обратите внимание, что во втором примере функция требует двух аргументов, а в последнем предполагается наличие определения `th` в области определений, такое как:

```
| (define th 20)
```

Результатом последнего примера будет значение #true, потому что поле цены в инвентарной записи содержит 10, а 10 меньше 20.

Важно отметить, что такие безымянные функции можно использовать везде, где нужна функция, в том числе с абстракциями из листинга 56:

```
> (map (lambda (x) (expt 10 x))
      '(1 2 3))
(list 10 100 1000)
> (foldl (lambda (name rst)
            (string-append name " ", "rst))
          "etc."
          '("Matthew" "Robby"))
"Robby, Matthew, etc."
> (filter (lambda (ir) (<= (ir-price ir) th))
           (list (make-ir "bear" 10)
                 (make-ir "doll" 33)))
(list (ir ...))
```

Многоточия не являются частью вывода. И снова последний пример предполагает наличие определения th.

Упражнение 279. Определите, какие из следующих выражений являются допустимыми лямбда-выражениями:

- 1) (lambda (x y) (x y y));
- 2) (lambda () 10);
- 3) (lambda (x) x);
- 4) (lambda (x y) x);
- 5) (lambda x 10).

Объясните, почему они допустимы или недопустимы. Если сомневаетесь, поэкспериментируйте в области взаимодействий в DrRacket. ■

Упражнение 280. Вычислите результаты следующих выражений:

- 1) ((lambda (x y) (+ x (* x y)))
1 2)
- 2) ((lambda (x y)
(+ x
(local ((define z (* y y)))
(+ (* 3 z) (/ 1 x)))))
1 2)
- 3) ((lambda (x y)
(+ x
((lambda (z)
(+ (* 3 z) (/ 1 z)))
(* y y))))
1 2)

Проверьте свои результаты в DrRacket. ■

Упражнение 281. Определите лямбда-выражение, которое:

- 1) принимает число и определяет, меньше ли оно, чем число 10;
- 2) умножает два заданных числа и преобразует результат в строку;
- 3) принимает натуральное число и возвращает 0, если число четное, и 1 – если нечетное;
- 4) принимает две инвентарные записи и сравнивает их цены;
- 5) добавляет красную точку в заданное изображение в заданную позицию Posn.

Опробуйте эти выражения в области взаимодействий. ■

17.2. Вычисления с лямбда-выражениями

Мы уже знаем, что лямбда-выражения являются сокращенной формой записи определенного вида выражений `local`, и такое их представление неразрывно связано с определениями констант и функций. Вместо представления определений функций в том виде, в каком они даны, мы можем рассматривать лямбда-выражения как еще одну фундаментальную концепцию и утверждать, что определение функции является сокращением определения константы, в котором в правой части используется лямбда-выражение.

Это утверждение проще продемонстрировать на конкретном примере:

<pre>(define (f x) является краткой формой для (define f (* 10 x))</pre>	<pre>(lambda (x) (* 10 x)))</pre>
--	---

Определение функции в этом примере состоит из двух шагов: создания функции и связывания ее с именем. Лямбда-выражение справа создает функцию одного аргумента `x`, которая вычисляет $10 \cdot x$; это определение `define` называет лямбда-выражение именем `f`. Мы даем имена функциям по двум разным причинам. С одной стороны, функции часто вызываются из других функций, и было бы нежелательно определять одно и то же лямбда-выражение снова и снова, когда потребуется вызвать эту функцию. С другой стороны, функции часто бывают рекурсивными, потому что обрабатывают рекурсивные данные, а присваивание имен функциям упрощает создание рекурсивных функций.

Упражнение 282. Поэкспериментируйте с определениями выше в DrRacket.

Также добавьте

```
; Number -> Boolean
(define (compare x)
  (= (f-plain x) (f-lambda x)))
```

в область определений, предварительно переименовав функцию `f` слева в `f-plain` и функцию `f` справа в `f-lambda`. Затем вычислите выражение

```
| (compare (random 100000))
```

несколько раз, чтобы убедиться, что обе функции согласуются – дают одинаковый результат для любых входных данных. ■

Если определения функций – это просто сокращенные формы записи определений констант, то мы можем заменить имя функции соответствующим лямбда-выражением:

```
| (f (f 42))
| ==
| ((lambda (x) (* 10 x)) ((lambda (x) (* 10 x)) 42))
```

Как ни странно, эта подстановка создает выражение, нарушающее известную нам грамматику. Точнее говоря, в результате подстановки получается выражение применения функции, где на месте функции находится лямбда-выражение.

Алонзо Черч (Alonzo Church), который изобрел лямбда-выражения в конце 1920-х годов, надеялся создать универсальную теорию функций.

Примерно так он и сформулировал свою бета-аксиому. В его работе показано, что с теоретической точки зрения язык не нуждается в локальных определениях, если в нем есть лямбда-выражения. К сожалению, поле книжной страницы слишком мало, чтобы достаточно хорошо объяснить эту идею. Если вам интересно, почитайте об Y-комбинаторе.

Дело в том, что грамматика ISL+ отличается от грамматики ISL: она поддерживает лямбда-выражения, а также допускает появление произвольных выражений в позиции применения функции. Это означает, что вам может потребоваться вычислить функцию, прежде чем продолжить ее применение, но вы уже знаете, как вычислять большинство выражений. Конечно, в результате вычисления выражения может получиться лямбда-выражение. Функции действительно являются значениями. Ниже приводится исправленная грамматика из интермеццо 1, включающая эти различия:

```
выражение = ...
| (выражение выражение ...)

значение = ...
| (lambda (переменная переменная ...) выражение)
```

Что вам действительно нужно знать, так это то, как оценить применение лямбда-выражения к аргументам, а делается это на удивление просто:

```
| ((lambda (x-1 ... x-n) f-body) v-1 ... v-n) == f-body
| ; в котором все вхождения x-1 ... x-n
| ; заменяются на v-1 ... v-n соответственно [beta-v]
```

То есть применение лямбда-выражения происходит так же, как применение обычной функции. Нужно заменить параметры функции фактическими значениями аргументов и вычислить значение тела функции.

Вот как применяется это правило в первом примере в этой главе:

```
((lambda (x) (* 10 x)) 2)
==
(* 10 2)
==
20
```

Второй пример вычисляется аналогично:

```
((lambda (name rst) (string-append name ", " rst))
  "Robby" "etc.")
==
(string-append "Robby" ", " "etc.")
==
"Robby, etc."
```

Стоп! Воспользуйтесь своей интуицией и вычислите третий пример:

```
((lambda (ir) (<= (ir-price ir) th))
  (make-ir "bear" 10))
```

предположив, что *th* больше или равно 10.

Упражнение 283. Убедитесь, что движок пошаговых вычислений в DrRacket может обрабатывать лямбда-выражения. Используйте его для вычисления третьего примера, а также определите, как DrRacket вычисляет следующие выражения:

```
(map (lambda (x) (* 10 x))
      '(1 2 3))

(foldl (lambda (name rst)
          (string-append name ", " rst))
      "etc."
      '("Matthew" "Robby"))

(filter (lambda (ir) (<= (ir-price ir) th))
        (list (make-ir "bear" 10)
              (make-ir "doll" 33))) ■
```

Упражнение 284. Вычислите по шагам это выражение:

```
| ((lambda (x) x) (lambda (x) x))
```

А затем это:

```
| ((lambda (x) (x x)) (lambda (x) x))
```

Стоп! Как вы думаете, что стоит попробовать дальше?

Верно, попробуйте вычислить это выражение:

```
| ((lambda (x) (x x)) (lambda (x) (x x)))
```

но будьте готовы щелкнуть на кнопке **STOP** (Остановить). ■

17.3. Абстрагирование с помощью лямбда-выражений

Вам может потребоваться некоторое время, чтобы привыкнуть к лямбда-выражениям, но скоро вы заметите, что лямбда-выражения делают короткие функции более удобочитаемыми, чем локальные определения. И действительно, мы можем немного видоизменить шаг 4 рецепта проектирования из раздела 16.6 и использовать `lambda` вместо `local`. Рассмотрим пример для этого раздела. Вот как выглядит макет на основе `local`:

```
| (define (dots lop)
|   (local (; Posn Image -> Image
|           (define (add-one-dot p scene) ...))
|         (foldr add-one-dot BACKGROUND lop)))
```

Если описать параметры так, чтобы их имена включали сигнатуры, вы легко сможете перепаковать всю информацию из `local` в единственное лямбда-выражение:

```
| (define (dots lop)
|   (foldr (lambda (a-posn scene) ...) BACKGROUND lop))
```

и затем завершить определение по аналогии с исходным шаблоном:

```
| (define (dots lop)
|   (foldr (lambda (a-posn scene)
|               (place-image DOT
|                           (posn-x a-posn)
|                           (posn-y a-posn)
|                           scene))
|         BACKGROUND lop))
```

Теперь проиллюстрируем этот прием замены еще на нескольких примерах из раздела 16.5:

- назначение первой функции – прибавить 3 к координате x каждого элемента в заданном списке структур `Posn`:

```
| ; [List-of Posn] -> [List-of Posn]
| (define (add-3-to-all lop)
|   (map (lambda (p)
|               (make-posn (+ (posn-x p) 3) (posn-y p)))
|         lop))
```

Поскольку `map` ожидает получить функцию одного аргумента, мы должны передать ей `(lambda (p) ...)`. Затем функция извлекает координату x из `p`, прибавляет 3 и переупаковывает данные в структуру `Posn`;

- вторая функция исключает структуры `Posn` из списка, в которых координата у больше 100:

```
| ; [List-of Posn] -> [List-of Posn]
| (define (keep-good lop)
|   (filter (lambda (p) (<= (posn-y p) 100)) lop))
```

Мы знаем, что `filter` ожидает получить функцию одного аргумента, которая дает логическое значение. Сначала лямбда-выражение извлекает координату `y` из структуры `Posn`, к которой `filter` применяет заданную функцию, и затем сравнивает ее с нужной нам границей – с числом 100;

- а третья определяет близость какой-либо точки `Posn` из списка `lop` к некоторой заданной точке:

```
| ; [List-of Posn] -> Boolean
| (define (close? lop pt)
|   (ogmap (lambda (p) (close-to p pt CLOSENESS))
|          lop))
```

Как и в предыдущих двух примерах, `ogmap` ожидает получить функцию одного аргумента и применяет ее к каждому элементу в заданном списке. Если хотя бы один из результатов – `#true`, то `ogmap` также вернет `#true`; если все результаты – `#false`, то и `ogmap` вернет `#false`.

Сравните определения из раздела 16.5 и определения, приведенные выше, поместив их рядом друг с другом. Вы должны заметить, насколько просто перейти от `local` к `lambda` и насколько лаконичны версии с лямбда-выражениями по сравнению с версиями, использующими локальные определения. Таким образом, если у вас появятся сомнения, то сначала спроектируйте функцию с использованием `local`, а затем, протестирував эту версию, преобразуйте ее в версию с `lambda`. Но имейте в виду, что лямбда-выражения – не панацея. У локальной функции есть имя, объясняющее ее назначение, и если оно длинное, то абстракцию с именованной функцией намного проще понять, чем абстракцию с длинным лямбда-выражением.

В следующих упражнениях предлагается решить задачи из раздела 16.7 на ISL+.

Упражнение 285. Используя `map`, определите функцию `convert-euro`, которая преобразует список сумм в долларах США в список сумм в евро, взяв за основу обменный курс 1,22 доллара США за евро.

Затем, используя `map`, определите функцию `convertFC`, которая преобразует список температур в градусах по Фаренгейту в список температур по Цельсию.

Наконец, попробуйте определить функцию `translate`, которая преобразует список структур `Posn` в список списков с парами чисел. ■

Упражнение 286. В инвентарной записи указывается название товара, его описание, закупочная цена и рекомендованная розничная цена.

Определите функцию, которая сортирует список записей по разнице между двумя этими ценами. ■

Упражнение 287. Используя `filter`, определите функцию `exclu-exp`. Она должна принимать число `ua` и список инвентарных записей и создавать список с записями, в которых розничная цена ниже `ua`.

Затем используйте `filter`, чтобы определить функцию `recall`, которая принимает параметр `ty` с назначением товара и список инвентарных записей и создает список записей, названия товаров в которых отличаются от `ty`.

Наконец, определите функцию `selection`, которая принимает два списка с названиями и выбирает из второго списка названия, также присутствующие в первом списке. ■

Упражнение 288. Используя `build-list` и `lambda`, определите функцию, которая

- 1) создает список `(list 0 ... (- n 1))` для любого натурального числа `n`;
- 2) создает список `(list 1 ... n)` для любого натурального числа `n`;
- 3) создает список `(list 1 1/2 ... 1/n)` для любого натурального числа `n`;
- 4) создает список первых `n` четных чисел;
- 5) создает диагональную матрицу из нулей и единиц; см. упражнение 262.

Также определите с помощью `lambda` функцию `tabulate`. ■

Упражнение 289. Используя `ogmar`, определите функцию `find-name`, которая должна принимать имя и список имён и определять, имеются ли в списке имена, целиком совпадающие с указанным или содержащие его.

Используя `andmap`, определите функцию, которая проверяет, все ли имена в списке начинаются с буквы "а".

Какую функцию следует использовать – `ogmar` или `andmap`, – чтобы определить функцию, которая позволит убедиться, что ни одно имя в некотором списке не длиннее заданного числа? ■

Упражнение 290. Функция `append` в языке ISL объединяет два списка в один, или, что то же самое, заменяет '()' в конце первого списка вторым списком:

```
| (equal? (append (list 1 2 3) (list 4 5 6 7 8))
|   (list 1 2 3 4 5 6 7 8))
```

Используйте `foldr`, чтобы определить `append-fgom-fold`. Что произойдет, если заменить `foldr` на `foldl`?

Теперь используйте одну из функций свертки, чтобы определить функции, которые вычисляют сумму и произведение для списка чисел.

Используйте одну из функций свертки, чтобы определить функцию, которая добавляет в сцену изображения из списка, располагая их по горизонтали. **Подсказки.** (1) Загляните в описание функций `beside` и `empty-image`. Попробуйте решить ту же задачу с помощью другой функции свертки. Определите также функцию, которая располагает изображения из списка по вертикали. (2) Загляните в описание функции `above`. ■

Упражнение 291. Функции свертки настолько универсальные, что с их помощью можно определить практически любую функцию обработки списков. Используйте `fold`, чтобы определить `map-via-fold`, которая делает то же, что и `map`. ■

17.4. Определение спецификаций с помощью лямбда-выражений

В листинге 60 показана обобщенная функция сортировки, которая принимает список значений и функцию сравнения. Для удобства в листинге 64 повторно приводятся основные элементы определения. В теле `sort-cmp` определяются две локальные вспомогательные функции: `isort` и `insert`. Кроме того, в листинге показаны два тестовых примера, которые иллюстрируют работу `sort-cmp`. Один демонстрирует работу функции на строках, а другой – на числах.

Листинг 64. Обобщенная функция сортировки

```
; [X] [List-of X] [X X -> Boolean] -> [List-of X]
; сортирует alon0 с помощью cmp

(check-expect (sort-cmp '("c" "b") string<?) '("b" "c"))
(check-expect (sort-cmp '(2 1 3 4 6 5) <) '(1 2 3 4 5 6))

(define (sort-cmp alon0 cmp)
  (local (; [List-of X] -> [List-of X]
         ; создает отсортированную версию alon
         (define (isort alon) ...)

         ; X [List-of X] -> [List-of X]
         ; вставляет n в отсортированный список чисел alon
         (define (insert n alon) ...))
    (isort alon0)))
```

Теперь вернемся к упражнению 186. В нем предлагается сформулировать тесты `check-satisfied` для `sort>` с помощью предиката `sorted?>`, где `sort>` – это функция, которая сортирует списки чисел в порядке убывания, а `sorted?>` – это функция, которая проверяет, отсортирован ли список чисел в порядке убывания. Вот решение этого упражнения:

```
(check-satisfied (sort> '()) sorted?>)
(check-satisfied (sort> '(12 20 -5)) sorted?>)
(check-satisfied (sort> '(3 2 1)) sorted?>)
(check-satisfied (sort> '(1 2 3)) sorted?>)
```

Вопрос в том, как аналогичным образом сформулировать тесты для `sort-cmp`.

Поскольку `sort-cmp` принимает функцию сравнения и список, то обобщенная версия `sorted?>` тоже должна принимать эти же параметры. Соответственно, тестовые примеры могут выглядеть так:

```
(check-satisfied (sort-cmp '("c" "b") string<?)  
                  (sorted string<?))  
(check-satisfied (sort-cmp '(2 1 3 4 6 5) <)  
                  (sorted <))
```

Оба выражения – (`sorted string<?)` и (`sorted <`) – должны создавать предикаты. Первый проверяет, отсортирован ли некоторый список строк в соответствии с функцией `string<?`, а второй – отсортирован ли список чисел с помощью операции `<`.

Таким образом, определяем желаемую сигнатуру и описание назначения для `sorted`:

```
; [X X -> Boolean] -> [ [List-of X] -> Boolean ]  
; возвращает функцию, которая проверяет,  
; отсортирован ли некоторый список согласно cmp  
(define (sorted cmp)  
  ...)
```

Теперь остается только выполнить остальные шаги процесса проектирования.

Для начала закончим заглушку. Как вы помните, заглушка создает значение, которое соответствует сигнатуре и может нарушать большинство тестов/примеров. Здесь нам нужно, чтобы `sorted` создавала функцию, принимающую список и возвращающую логическое значение. Теперь, когда у нас есть `lambda`, эту задачу легко решить:

```
(define (sorted cmp)  
  (lambda (l)  
    #true))
```

Стоп! Это ваша первая функция, создающая функции. Прочтите ее определение еще раз. Попробуйте объяснить его своими словами.

Далее мы должны написать примеры. Согласно нашему анализу, `sorted` принимает предикаты, такие как `string<?` и `<`, но очевидно, что точно так же она должна принимать `>`, `<=` и ваши собственные функции сравнения. На первый взгляд, это требование предлагает примерно такие тестовые примеры:

```
(check-expect (sorted string<?) ...)  
(check-expect (sorted <) ...)
```

Но результат (`sorted ...`) – функция, а согласно упражнению 245 функции невозможно сравнивать.

Следовательно, чтобы сформулировать разумные тестовые примеры, нужно применить результат (`sorted ...`) к соответствующим спискам. Учитывая это, тестовые примеры формулируются почти сами; в действительности их легко получить из примеров для `sort-cmp` в листинге 64:

```
(check-expect [(sorted string<?) '("b" "c")] #true)  
(check-expect [(sorted <) '(1 2 3 4 5 6)] #true)
```

Примечание. Использование квадратных скобок вместо круглых подчеркивает, что первое выражение создает функцию, которая затем применяется к аргументам.

С этого момента проектирование продолжается как по маслу. Фактически нам нужно получить обобщенную версию функции `sorted()?` из раздела 9.2; назовем ее `sorted/l`. Необычность `sorted/l` в том, что она «живет» в теле лямбда-выражения внутри `sorted`:

```
(define (sorted cmp)
  (lambda (l0)
    (local ((define (sorted/l l) ... cmp ...))
      ...)))
```

Обратите внимание, что `sorted/l` определяется локально, но ссылается на `cmp`.

Упражнение 292. Спроектируйте функцию `sorted?` со следующей сигнатурой и описанием назначения:

```
; [X X -> Boolean] [NEList-of X] -> Boolean
; определяет, отсортирован ли список l в соответствии с cmp

(check-expect (sorted? < '(1 2 3)) #true)
(check-expect (sorted? < '(2 1 3)) #false)

(define (sorted? cmp l)
  #false)
```

В список желаний даже включены примеры. ■

В листинге 65 показан результат проектирования. Функция `sorted` принимает функцию сравнения `cmp` и создает предикат. Предикат принимает список `l0` и использует локальную функцию, чтобы определить, все ли элементы в `l0` упорядочены согласно `cmp`. В частности, локальная функция проверяет наличие элементов в списке; внутри `local` функция `sorted` сначала проверяет, является ли `l0` пустым списком, и в этом случае просто возвращает `#true`, потому что пустой список можно считать отсортированным.

Стоп! Попробуйте переопределить `sorted` с использованием `sorted?` из упражнения 292. Объясните, почему `sorted/l` не принимает `cmp` в виде аргумента.

Листинг 65. Каррированный предикат для проверки упорядоченности элементов в списке

```

; [X X -> Boolean] -> [[List-of X] -> Boolean]
; определяет, отсортирован ли список l0 в соответствии с cmp
(define (sorted cmp)
  (Lambda (l0)
    (local ( ; [NEList-of X] -> Boolean
            ; отсортирован ли l в соответствии с cmp
            (define (sorted/l l)
              (cond
                [(empty? (rest l)) #true]

```

```
| [else (and (cmp (first l) (second l))
|   (sorted/l (rest l))))]
| (if (empty? l0) #true (sorted/l l0)))
```

Слово «каррирование» используется как дань уважения Хаскеллу Карри (Haskell Curry), второму человеку, придумавшему эту идею.

Первым был Моисей Эльевич Шейнфинкель (Moses Schönfinkel).

Функция `sorted` в листинге 65 – это *каррированная* версия функции, принимающей два аргумента: `cmp` и `l0`. Вместо получения двух аргументов сразу она принимает только один аргумент и возвращает функцию, которая принимает второй.

В упражнении 186 спрашивается, как сформулировать тестовый пример, выявляющий ошибки в функциях сортировки. Рассмотрим следующее определение:

```
| ; List-of-numbers -> List-of-numbers
| ; создает отсортированную версию l
(define (sort-cmp/bad cmp)
  '(9 8 7 6 5 4 3 2 1 0))
```

Сформулировать такой тестовый пример с помощью функции `check-expect` очень просто.

Чтобы спроектировать предикат, который определяет результат `sort-cmp/bad` как ошибочный, нужно понять назначение `sort-cmp` и сортировки в целом. Совершенно неприемлемо отбрасывать данный список и создавать вместо него какой-то другой. Поэтому в описании назначения `isort` говорится, что функция «создает **версию**» данного списка. «Версия» означает, что функция ничего не выбрасывает из данного списка.

Учитывая это, мы можем сказать, что нам нужен предикат, который проверяет, отсортирован ли результат и содержит ли он все элементы из заданного списка:

```
| ; [List-of X] [X X -> Boolean] -> [[List-of X] -> Boolean]
| ; отсортирован ли l0 в соответствии с cmp
| ; и все ли элементы в списке k являются членами списка l0
(define (sorted-variant-of k cmp)
  (lambda (l0) #false))
```

Две строки из описания назначения предполагают следующие примеры:

```
| (check-expect [(sorted-variant-of '(3 2) <) '(2 3)]
|               #true)
| (check-expect [(sorted-variant-of '(3 2) <) '(3)]
|               #false)
```

Так же как `sorted`, `sorted-variant-of` принимает аргументы и создает функцию. В первом случае `sorted-variant-of` дает результат `#true`, потому что список '(2 3) отсортирован и содержит все числа из '(3 2). Во втором случае функция возвращает `#false`, потому что в '(3) не хватает 2 из исходного списка.

Двухстрочное описание назначения определяет две задачи, а это означает, что сама функция является комбинацией из двух функций:

```
(define (sorted-variant-of k cmp)
  (lambda (l0)
    (and (sorted? cmp l0)
         (contains? l0 k))))
```

Тело функции составляет выражение `and`, объединяющее вызовы двух функций. Вызовом функции `sorted?` из упражнения 292 она реализует первую строку в описании назначения. Второй вызов (`(contains? k l0)`) – это неявное требование внести вспомогательную функцию в список желаний.

Рассмотрим полное определение:

```
; [List-of X] [List-of X] -> Boolean
; все элементы списка k являются элементами списка l

(check-expect (contains? '(1 2 3) '(1 4 3)) #false)
(check-expect (contains? '(1 2 3 4) '(1 3)) #true)

(define (contains? l k)
  (andmap (lambda (in-k) (member? in-k l)) k))
```

С одной стороны, мы не обсуждали систематическое проектирование функций, принимающих два списка, но мы сделаем это в отдельной главе (см. главу 23). С другой стороны, определение функции явно удовлетворяет описанию назначения. Выражение `andmap` проверяет, каждый ли элемент в `k` является членом (`member?`) списка `l`, как обещано в описании назначения.

К сожалению, `sorted-variant-of` не может правильно описать функции сортировки. Рассмотрим, например, такой вариант функции сортировки:

```
; [List-of Number] -> [List-of Number]
; создает отсортированную версию списка l
(define (sort-cmp/worse l)
  (local ((define sorted (sort-cmp l <)))
    (cons (- (first sorted) 1) sorted)))
```

С помощью `check-expect` легко обнаружить непригодность в этой функции: следующий тест должен выполняться успешно, но этого не происходит:

```
| (check-expect (sort-cmp/worse '(1 2 3)) '(1 2 3))
```

Удивительно, но `sorted-variant-of` благополучно проходит тест `check-satisfied`:

```
| (check-satisfied (sort-cmp/worse '(1 2 3))
                  (sorted-variant-of '(1 2 3) <))
```

В действительности этот тест успешно выполняется для любых списков чисел, а не только для `'(1 2 3)`, потому что генератор предикатов просто проверяет, что список результата содержит все элементы

из исходного списка; он не проверяет обратное утверждение, что исходный список содержит все элементы из списка результата.

Самый простой способ организовать эту третью проверку в `sorted-variant-of` – добавить третье подвыражение в выражение `and`:

```
(define (sorted-variant-of.v2 k cmp)
  (lambda (l0)
    (and (sorted? cmp l0)
         (contains? l0 k)
         (contains? k l0))))
```

Мы повторно использовали `contains?`, но переставили аргументы местами.

На этом этапе у вас может появиться вопрос: почему мы не озабочились разработкой такого предиката, коль скоро есть возможность исключить плохие функции сортировки с помощью простых тестов `check-expect`. Дело в том, что `check-expect` проверяет только, насколько правильно наши функции работают с конкретными списками. С помощью такого предиката, как `sorted-variant-of.v2`, мы можем сформулировать утверждение, что функция сортировки правильно работает **со всеми** возможными входными данными:

```
(define a-list (build-list-of-random-numbers 500))

(check-satisfied (sort-cmp a-list <)
  (sorted-variant-of.v2 a-list <))
```

Давайте внимательно рассмотрим эти две строки. Первая строка генерирует список из 500 чисел. Каждый раз, когда выполняется этот тест, он с высокой долей вероятности создает уникальный список. Вторая строка – это тестовый пример, согласно которому при сортировке этого сгенерированного списка создается другой список, который (1) отсортирован, (2) содержит все числа из сгенерированного списка и (3) не содержит ничего другого. Иначе говоря, для всех возможных списков функция `sort-cmp` возвращает результат, который удовлетворяет предикату `sorted-variant-of.v2`.

Специалисты по информатике назвали бы `sorted-variant-of.v2` *спецификацией* функции сортировки. Утверждение о том, что вышеуказанный тест благополучно выполняется для **всех** списков чисел, является **теоремой** о связи между спецификацией функции сортировки и ее реализацией. Если программист может доказать эту теорему математически, то мы говорим, что функция **верна** в отношении ее спецификации. Доказательство правильности функций или программ выходит за рамки этой книги, но хороший курс информатики покажет вам, как построить такие доказательства.

Упражнение 293. Спроектируйте функцию `found?` – спецификацию функции `find`:

```
; [X] [List-of X] -> [Maybe [List-of X]]
; возвращает первый подсписок из l, который начинается
```

```
; с x, иначе #false
(define (find x l)
  (cond
    [(empty? l) #false]
    [else
      (if (equal? (first l) x) l (find x (rest l))))])
```

Используйте `found?`, чтобы сформулировать тест `check-satisfied` для `find`. ■

Упражнение 294. Спроектируйте `is-index?` – спецификацию функции `index`:

```
; [X] [List-of X] -> [Maybe N]
; определяет индекс первого вхождения
; x в l, иначе #false
(define (index x l)
  (cond
    [(empty? l) #false]
    [else (if (equal? (first l) x)
              0
              (local ((define i (index x (rest l))))
                  (if (boolean? i) i (+ i 1))))]))
```

Используйте `is-index?`, чтобы сформулировать тест `check-satisfied` для `index`. ■

Упражнение 295. Спроектируйте `n-inside-playground?` – спецификацию функции `random-posns`. Она должна генерировать предикат, сравнивающий длину данного списка с некоторым заданным числом и проверяющий, что все экземпляры `Posn` в этом списке находятся внутри прямоугольника с размерами `WIDTH × HEIGHT`:

```
; расстояния в пикселях
(define WIDTH 300)
(define HEIGHT 300)

; N -> [List-of Posn]
; генерирует n случайных точек Posn
; с координатами X в диапазоне [0,WIDTH) и
; с координатами Y в диапазоне [0,HEIGHT)
(define (random-posns n)
  (check-satisfied (random-posns 3)
    (n-inside-playground? 3)))
(define (random-posns n)
  (build-list
    n
    (lambda (i)
      (make-posn (random WIDTH) (random HEIGHT)))))
```

Определите функцию `random-posns/bad`, удовлетворяющую предикату `n-inside-playground?`, но не соответствующую описанию назначения. **Примечание.** Эта спецификация **неполная**. На ум может прийти слово «частичная», но в информатике фраза «частичная спецификация» используется для других целей. ■

17.5. Представление с помощью лямбда-выражений

Так как функции в ISL+ являются обычными значениями, их можно рассматривать как еще одну форму данных и использовать для представления данных. Далее мы подробно рассмотрим эту идею и будем опираться на нее в следующих нескольких главах. В названии раздела

используется слово «представление», потому что люди считают функции еще одним представлением данных.

Как всегда, начнем с постановки задачи.

Эту задачу можно решить также с помощью представления данных, ссылающегося на самого себя и утверждающего, что фигура – это круг, прямоугольник или комбинация двух фигур. См. следующую часть книги, чтобы узнать больше об этом способе проектирования.

Задача. Стратеги ВМФ представляют флот кораблей в виде прямоугольников (сами корабли) и кругов (радиус действия их оружия). Область, которую может контролировать флот, определяется сочетанием всех этих форм. Спроектируйте представление данных для прямоугольников, кругов и их комбинаций. Затем спроектируйте функцию, которая определяет, находится ли некоторая точка внутри контролируемой области.

Задача допускает все возможные конкретные интерпретации, которые мы здесь опустим. Немного более сложная версия была предметом соревнований по программированию в середине 1990-х, проводившихся Йельским университетом по заказу министерства обороны США.

Один из математических подходов заключается в рассмотрении фигур (описывающих контролируемую область) как предикатов точек. То есть фигура – это функция, которая отображает декартову точку в логическое значение. Давайте переведем эти слова в определение данных:

```
; Shape -- это функция:  
; [Posn -> Boolean]  
; интерпретация: если s -- это фигура и p -- точка Posn, то (s p)  
; дает #true, если p находится внутри s, иначе -- #false
```

Интерпретация получилась довольно многословной, потому что это необычное представление данных. Такое представление требует немедленного исследования с примерами. Однако мы пока отложим этот шаг и определим функцию, которая проверяет, находится ли точка внутри какой-либо фигуры:

```
; Shape Posn -> Boolean  
(define (inside? s p)  
(s p))
```

Для данной интерпретации сделать это совсем несложно. Кроме того, как оказывается, это проще, чем создавать примеры, и, что особенно удивительно, функцию удобно использовать для формулирования примеров данных.

Стоп! Объясните, как и почему работает `inside?`.

Теперь вернемся к задаче определения экземпляров Shape. Вот упрощенный экземпляр класса:

```
| ; Posn -> Boolean
| (lambda (p) (and (= (posn-x p) 3) (= (posn-y p) 4)))
```

Эта функция принимает `p` – экземпляр `Posn` – и сравнивает координаты `p` с координатами точки $(3,4)$. То есть эта функция представляет одну точку. Представление данных в виде `Shape` может показаться странным, но оно подсказывает, как можно определить функции, создающие экземпляры `Shape`:

```
| ; Number Number -> Shape
| (define (mk-point x y)
|   (lambda (p)
|     (and (= (posn-x p) x) (= (posn-y p) y))))
```

```
| (define a-sample-shape (mk-point 3 4))
```

Мы использовали префикс «`mk`», потому что это функция, а не обычный конструктор.

Стоп! Убедитесь, что последняя строка создает представление данных $(3,4)$. Используйте для этого движок пошаговых вычислений в DrRacket.

Если бы мы **проектировали** такую функцию, мы сформулировали бы описание назначения и представили несколько иллюстративных примеров. Описание назначения можно было бы сформулировать очевидным образом:

```
| ; создает представление для точки  $(x, y)$ 
```

или более кратко и более точно:

```
| ; представляет точку  $(x, y)$ 
```

В примерах мы хотим использовать интерпретацию `Shape`. Например, результатом вычисления выражения `(mk-point 3 4)` должна быть функция, которая возвращает `#true` тогда и только тогда, когда у нее на входе `(make-posn 3 4)`. Вот как это утверждение можно выразить в тестах с помощью `inside?`:

```
| (check-expect (inside? (mk-point 3 4) (make-posn 3 4))
|               #true)
| (check-expect (inside? (mk-point 3 4) (make-posn 3 0)))
|               #false)
```

Проще говоря, чтобы создать представление точки, мы определяем функцию-конструктор, которая принимает две координаты точки. Вместо записи эта функция использует лямбда-выражение для создания другой функции. Создаваемая функция принимает экземпляр `Posn` и проверяет, равны ли его поля `x` и `y` заданным координатам.

Теперь расширим эту идею от просто точек до фигур, скажем окружности. Из курса геометрии вы знаете, что окружность – это совокупность точек, располагающихся на одинаковом расстоянии от

центра окружности, которое называется радиусом. Для точек внутри окружности расстояние от центра меньше или равно радиусу. Следовательно, функция, которая создает представление окружности, должна принимать три аргумента: две координаты центра и радиус:

```
| ; Number Number Number -> Shape
| ; создает представление окружности с радиусом r
| ; и с центром в точке (center-x, center-y)
(define (mk-circle center-x center-y r)
  ...)
```

Так же как `mk-point`, она создает функцию с помощью лямбда-выражения. Возвращаемая функция определяет, находится ли некоторая точка `Posn` внутри окружности. Вот несколько примеров, сформулированных в виде тестов:

```
| (check-expect
|   (inside? (mk-circle 3 4 5) (make-posn 0 0)) #true)
| (check-expect
|   (inside? (mk-circle 3 4 5) (make-posn 0 9)) #false)
| (check-expect
|   (inside? (mk-circle 3 4 5) (make-posn -1 3)) #true)
```

Начало координат (`make-posn 0 0`) находится ровно в пяти шагах от (3,4), центра окружности; см. раздел 5.4. Стоп! Объясните оставшиеся примеры.

Упражнение 296. Для проверки результатов тестирования нарисуйте схемы с помощью циркуля и карандаша.

С математической точки зрения `Posn` p находится внутри окружности, если расстояние между p и центром круга меньше радиуса r . Добавим в список желаний соответствующую вспомогательную функцию и запишем, что у нас есть.

```
| (define (mk-circle center-x center-y r)
|   ; [Posn -> Boolean]
|   (lambda (p)
|     (<= (distance-between center-x center-y p) r)))
```

Определение функции `distance-between` – простое упражнение. ■

Упражнение 297. Спроектируйте функцию `distance-between`. Она должна принимать два числа и экземпляр `Posn` (x , y и p) и вычислять расстояние между точками (x, y) и p .

Знание предметной области. Расстояние между (x_0, y_0) и (x_1, y_1) вычисляется по формуле:

$$\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2},$$

то есть расстояние от точки $(x_0 - x_1; y_0 - y_1)$ до начала координат. ■

Представление данных для прямоугольников выражается аналогично:

```
| ; Number Number Number Number -> Shape
| ; представляет прямоугольник с заданными шириной и высотой,
```

```
; левый верхний угол которого находится в точке (ul-x, ul-y)

(check-expect (inside? (mk-rect 0 0 10 3)
                        (make-posn 0 0))
              #true)
(check-expect (inside? (mk-rect 2 3 10 3)
                        (make-posn 4 5))
              #true)
```

Стоп! Сформулируйте негативный тест.

```
(define (mk-rect ul-x ul-y width height)
  (lambda (p)
    (and (<= ul-x (posn-x p) (+ ul-x width))
         (<= ul-y (posn-y p) (+ ul-y height)))))
```

Конструктор этой фигуры принимает четыре числа: координаты верхнего левого угла, его ширину и высоту. Результатом снова является лямбда-выражение. Так же как для окружностей, эта функция принимает Posn и возвращает логическое значение, проверяя, находятся ли поля x и y экземпляра Posn на допустимом расстоянии.

На данный момент нам осталось решить только одну задачу, а именно: проектировать функцию, которая отображает два представления Shape в их комбинацию. Сигнатура и заголовок выглядят просто:

```
; Shape Shape -> Shape
; комбинирует две фигуры в одну
(define (mk-combination s1 s2)
  ; Posn -> Boolean
  (lambda (p)
    #false))
```

Даже значение по умолчанию выглядит просто. Мы знаем, что фигура представлена как функция, преобразующая Posn в логическое значение, поэтому запишем лямбда-выражение, принимающее некоторое количество экземпляров Posn и возвращающее значение #false, означающее, что комбинация не имеет смысла.

Итак, предположим, что мы должны объединить круг и прямоугольник сверху:

```
(define circle1 (mk-circle 3 4 5))
(define rectangle1 (mk-rect 0 3 10 3))
(define union1 (mk-combination circle1 rectangle1))
```

Некоторые точки находятся внутри этой комбинации фигур, а другие – вовне:

```
(check-expect (inside? union1 (make-posn 0 0)) #true)
(check-expect (inside? union1 (make-posn 0 9)) #false)
(check-expect (inside? union1 (make-posn -1 3)) #true)
```

Поскольку точка (make-posn 0 0) находится внутри обеих фигур, нет никаких сомнений в том, что она находится внутри их комбинации. Точно так же (make-posn 0 -1) не находится ни в одной из фигур, а зна-

чит, и в их комбинации. Наконец, (`make-posn -1 3`) находится внутри `circle1`, но за пределами `rectangle1`. Но мы считаем, что точка находится внутри комбинации двух фигур, если оказывается внутри хотя бы одной из них.

Данный анализ примеров подразумевает доработку `mk-combination`:

```
| ; Shape Shape -> Shape
| (define (mk-combination s1 s2)
|   ; Posn -> Boolean
|   (lambda (p)
|     (or (inside? s1 p) (inside? s2 p))))
```

Выражение `or` говорит нам, что результатом будет `#true`, если хотя бы одно из подвыражений дает `#true`: `(inside? s1 p)` или `(inside? s2 p)`. Первое подвыражение определяет, находится ли `p` в `s1`, а второе – находится ли `p` в `s2`. Это точный перевод нашего объяснения на ISL+.

Упражнение 298. Спроектируйте `my-animate`. Напомним, что функция `animate` принимает представление *потока* изображений, по одному на натуральное число. Поскольку потоки бесконечно длинные, их нельзя представить в форме обычных составных данных. Поэтому вместо них мы используем функции:

```
| ; ImageStream -- это функция:
| ; [N -> Image]
| ; интерпретация: поток s обозначает последовательность изображений
```

Вот пример данных:

```
| ; ImageStream
| (define (create-clock-scene height)
|   (place-image 🕒 50 height (empty-scene 60 60)))
```

Возможно, вы узнали один из первых фрагментов кода в прологе.

Функция `(my-animate s n)` должна показать `n` изображений – `(s 0), (s 1) и т. д. – со скоростью 30 изображений в секунду. Ее результат – количество тактов часов, прошедших с момента запуска.`

Примечание. Это тот случай, когда легко записать примеры и тесты, но эти примеры и тесты мало помогают процессу проектирования функции `big-bang`. Использование функций в качестве представлений данных требует знания большего количества концепций проектирования, чем предлагает эта книга. ■

Упражнение 299. Спроектируйте представление данных для конечных и бесконечных множеств и представьте с его помощью множество всех нечетных чисел, всех четных чисел, всех чисел, делящихся на 10, и т. д.

Спроектируйте функции: `add-element`, которая добавляет элемент в множество; `union`, объединяющую элементы двух множеств; и `intersect`, выбирающую все элементы, присутствующие в двух множествах.

Подсказка. Математики оперируют множествами как функциями, которые принимают потенциальный элемент `ed` и отвечают `#true`, только если `ed` принадлежит множеству. ■

18. Итоги

Эта третья часть книги посвящена роли абстракций в разработке программ. Абстракции имеют две стороны: создание и использование. Поэтому мы завершаем эту часть двумя итогами:

- 1) **повторяющиеся шаблоны кода требуют абстрагирования.**
Под абстрагированием понимается исключение повторяющихся фрагментов кода и параметризация различий. Создавая правильные абстракции, программисты избавляются от лишней работы и головной боли в будущем, потому что ошибки, неэффективность и другие проблемы сосредоточиваются в одном месте. Благодаря этому исправление единственной абстракции устраниет любую конкретную проблему раз и навсегда. Напротив, наличие повторяющегося кода означает, что при обнаружении проблемы программист должен будет найти все копии и исправить их все;
- 2) большинство языков имеют большое количество уже реализованных абстракций. Некоторые из них добавлены создателями языка; другие – программистами, использующими этот язык. Чтобы обеспечить эффективное повторное использование этих абстракций, их создатели должны предоставить соответствующее описание – **описание назначения, сигнатуру и хорошие примеры**, – а программисты будут использовать их для применения абстракций.

Все языки программирования имеют средства для создания абстракций, в одних лучше, в других хуже. Все программисты должны знать эти средства и уже готовые абстракции, предлагаемые языком. Взыскательный программист научится различать языки программирования по этим характеристикам.

Помимо абстракций, в этой части также была представлена идея, что

**функции – это значения,
и они тоже могут представлять информацию.**

Это довольно древняя идея для семейства языков программирования Lisp (таких как ISL+) и для специалистов, занимающихся исследованием языков программирования, но она лишь недавно получила признание в большинстве современных ведущих языков – C#, C++, Java, JavaScript, Perl, Python.

Интермеццо 3. Область видимости и абстракции

В предыдущей части мы не дали неформального объяснения `local` и `lambda`, однако для дальнейшего обсуждения этих механизмов абстракции необходимо познакомиться с дополнительной терминологией. В частности, нам потребуются новые термины, очерчивающие области внутри программ и указывающие на конкретное использование переменных.

Это интермеццо начинается с раздела, в котором определяются новые термины: область видимости, связывание переменных и связанные переменные. Затем эти термины сразу же используются для представления двух механизмов абстракции, часто встречающихся в языках программирования: циклов `for` и сопоставления с образцом. Цикл `for` может служить альтернативой таким функциям, как `map`, `build-list`, `andmap` и т. д., а сопоставление с образцом абстрагирует условные выражения, широко использовавшиеся в функциях в первых трех частях книги. Оба механизма требуют не только определения функций, но также создания совершенно новых языковых конструкций, то есть они не являются чем-то, что должны проектировать программисты и добавлять в свои словари.

Область видимости

Рассмотрим следующие два определения:

```
| (define (f x) (+ (* x x) 25))  
| (define (g x) (+ (f (+ x 1)) (f (- x 1))))
```

Совершенно очевидно, что вхождения `x` в `f` никак не связаны с вхождениями `x` в `g`. Мы могли бы заменить `x` в определении `f` на `y`, и функция все равно вычисляла бы тот же самый результат. Проще говоря, выделенные вхождения `x` имеют значение только внутри определения `f` и больше нигде.

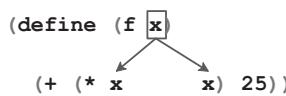
В то же время первое появление `x` в `f` отличается от других. Когда вычисляется выражение `(f n)`, вхождение `f` полностью исчезает, а вхождения `x` заменяются на `n`. Чтобы различить эти два типа вхождений переменных, мы называем `x` в заголовке функции *связывающим вхождением*, а в теле функции – *связанным вхождением*. Мы также говорим, что связывающее вхождение `x` связывает все вхождения `x` в теле `f`. У тех, кто изучает языки программирования, есть даже название области, в которой действует связывающее вхождение, – *лексическая область видимости*.

Определения `f` и `g` связывают еще два имени: `f` и `g`. Их область видимости называется *областью видимости верхнего уровня*, потому что области видимости могут быть вложенными (см. ниже).

Термин *свободное вхождение* обозначает переменную без связывающего вхождения. Это имя без определения, то есть ни язык, ни его библиотеки, ни программа не связывают это имя с каким-либо значением. Например, если скопировать указанную выше программу в область определений и запустить ее, то после ввода `f`, `g` и `x` в области взаимодействий вы увидите, что первые два имени определены, а последнее – нет:

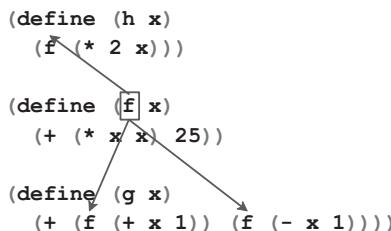
```
> f
f
> g
g
> x
x:this variable is not defined
```

Описание лексической области видимости предлагает наглядное представление определения `f`:



Если щелкнуть на кнопке **Check Syntax**
(Проверить синтаксис) в DrRacket и навести указатель мыши на имя переменной, то будет нарисована диаграмма, как показано на этом рисунке.

Вот диаграмма для области видимости верхнего уровня:



Обратите внимание, что область видимости имени `f` включает все определения выше и ниже его определения. Маркер над первым вхождением показывает, что это связывающее вхождение. Стрелки от связывающего вхождения к связанным вхождениям указывают, где используется это значение. Когда значение связывающего вхождения становится известным, связанные вхождения получают свои значения оттуда.

Точно так же эти диаграммы объясняют, как работает переименование. Если вы решите переименовать параметр функции, то вам придется найти все связанные вхождения в его области видимости и заменить их. Например, чтобы переименовать параметр `x` в `y` в приведенной выше функции `f`, потребуется заменить в

```
(define (f x) (+ (* x x) 25))
(define (g x) (+ (f (+ x 1)) (f (- x 1))))
```

только два вхождения `x`:

```
(define (f y) (+ (* y y) 25))
(define (g x) (+ (f (+ x 1)) (f (- x 1))))
```

Упражнение 300. Вот простая программа на ISL+:

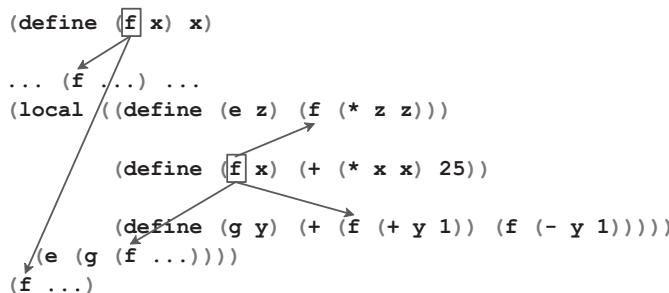
```
(define (p1 x y)
  (+ (* x y)
     (+ (* 2 x)
        (+ (* 2 y) 22)))))

(define (p2 x)
  (+ (* 55 x) (+ x 11)))

(define (p3 x)
  (+ (p1 x 0)
     (+ (p1 x 1) (p2 x))))
```

Нарисуйте стрелки от параметра x в функции $p1$ ко всем его связанным вхождениям. Нарисуйте стрелки от $p1$ ко всем связанным вхождениям $p1$. Проверьте результаты, щелкнув на кнопке **Check Syntax** (Проверить синтаксис) в DrRacket. ■

В отличие от определений функций верхнего уровня, локальные определения имеют ограниченную область видимости. В частности, область видимости локальных определений ограничивается выражением `local`. Рассмотрим определение вспомогательной функции f в выражении `local`. С этим определением будут связаны все вхождения в выражении `local`, при этом с ним не будут связаны никакие вхождения того же имени вовне:



Два вхождения за пределами выражения `local` не были связаны с локальным определением f . Как обычно, параметры функции, локальной или нет, связываются только с вхождениями в теле функции.

Поскольку область видимости имени функции или ее параметров является текстовой областью, многие также рисуют коробчатые диаграммы, чтобы показать область видимости. То есть рисуют рамку вокруг тела функции:

```
(define (f x)
  (+ (* 2 x) 10))
```

В случае с выражением local рамка охватывает все выражение:

```
(define (f z)
  (local ((define (f x) (+ x (* x x) 55))
          (define (g y) (+ (f y) 10)))
    (f z)))
```

В этом примере рамка описывает область видимости определений f и g.

Нарисовав рамку вокруг области видимости, легко понять смысл повторного использования имени функции внутри выражения local:

```
(define (a-function y)
  (local ((define (f z y) (+ (* x y) (+ x y)))
          (define (g z)
            (local ((define (f x) (+ (* x x) 55))
                   (define (g y) (+ (f y) 10)))
              (f z)))
          (define (h x) (f x (g x))))
    (h y)))
```

Серый фон показывает область видимости внутреннего определения f, а рамка очерчивает область видимости внешнего определения f. Соответственно, все вхождения f в сером прямоугольнике ссылаются на внутреннее определение в выражении local, а все вхождения внутри рамки с белым фоном – на определение во внешнем выражении local. Иными словами, серый прямоугольник – это *дыра* в области видимости внешнего определения f.

Дыры также могут появляться в области видимости параметров:

```
(define (f x)
  (local ((define (g x)
            (+ x (* x 2)))
          (g x)))
    (g x)))
```

В этой функции параметр x используется дважды: в теле f и в теле g, соответственно, область видимости второй функции – это дыра в области видимости первой.

Как правило, если одно и то же имя встречается в функции несколько раз, то рамки, ограничивающие их области видимости, никогда не перекрываются. Иногда рамки могут вкладываться друг в друга, обраzuя дыры. Но в любом случае общая картина всегда имеет иерархический вид – множество рамок, вложенных друг в друга.

Листинг 66. Код для упражнения 301

```
(define (insertion-sort alon)
  (local ((define (sort alon)
            (cond
              [(empty? alon) '()]
              [else
                (add (first alon) (sort (rest alon))))])
          (define (add an alon)
            (cond
              [(empty? alon) (list an)]
              [else
                (cond
                  [(> an (first alon)) (cons an alon)]
                  [else (cons (first alon)
                               (add an (rest alon))))]))]
        (sort alon)))
```

Упражнение 301. Обведите рамкой область видимости каждого связывающего вхождения `sort` и `alon` в листинге 66. Затем нарисуйте стрелки от каждого вхождения `sort` к соответствующему связывающему вхождению. Потом повторите упражнение для кода в листинге 67. Отличаются ли две функции чем-то еще, кроме имен? ■

Листинг 67. Код для упражнения 301 (версия 2)

```
(define (sort alon)
  (local ((define (sort alon)
            (cond
              [(empty? alon) '()]
              [else
                (add (first alon) (sort (rest alon))))])
          (define (add an alon)
            (cond
              [(empty? alon) (list an)]
              [else
                (cond
                  [(> an (first alon)) (cons an alon)]
                  [else (cons (first alon)
                               (add an (rest alon))))]))]
        (sort alon)))
```

Упражнение 302. Как уже говорилось выше, каждое вхождение переменной получает свое значение из соответствующего связывающего вхождения. Рассмотрим следующее определение:

```
| (define x (cons 1 x))
```

Где находится связывающее вхождение для символа `x`, выделенного серым фоном? Поскольку это определение является определением константы, а не функции, то мы должны немедленно вычислить правую часть. Какое значение должно получиться по нашим правилам? ■

Как обсуждалось в разделе 17.1, лямбда-выражение – это просто сокращенная форма записи локального выражения. То есть если предположить, что `a-new-name` не встречается в `exp`, то выражение

```
| (lambda (x-1 ... x-n) exp)
```

можно заменить на:

```
| (local ((define (a-new-name x-1 ... x-n) exp))
      a-new-name)
```

Из нашего объяснения следует, что

```
| (lambda (x-1 ... x-n) exp)
```

вводит x_1, \dots, x_n как связывающие вхождения с областью видимости, ограниченной выражением exp , например

```
(define f
  (lambda (x)
    (+ (* x x) 25)))
```

Конечно, если exp содержит дополнительные конструкции связывания (скажем, вложенное выражение $local$), тогда в области видимости параметров может появиться дыра.

Упражнение 303. Нарисуйте стрелки от вхождений x , выделенных серым фоном, к соответствующим связывающим вхождениям в каждом из следующих лямбда-выражений:

1. (lambda (x y)
 (+ **x** (* x y)))
2. (lambda (x y)
 (+ **x**
 (local ((define x (* y y)))
 (+ (* 3 **x**)
 (/ 1 x)))))
3. (lambda (x y)
 (+ **x**
 ((lambda (x)
 (+ (* 3 **x**)
 (/ 1 x)))
 (* y y))))

Также нарисуйте рамки, ограничивающие области видимости каждого выделенного вхождения x и дыры в них, если это необходимо. ■

Циклы в языке ISL

В части III мы уже познакомились с циклами, даже притом, что в ней никогда не упоминалось это слово. Говоря общим языком, цикл просматривает составные данные и обрабатывает их элементы по одно-

Подключите библиотеку
2htdp/abstraction.

Преподаватели, которые будут использовать ее до конца книги, должны объяснить, как принципы проектирования применяются к языкам, не имеющим инструкций

for и *match*.

му. При этом циклы синтезируют новые данные. Например, *map* просматривает список, применяет заданную функцию к каждому элементу и собирает результаты в новый список. Точно так же *build-list* перечисляет все натуральные числа, предшествующие заданному (от 0 до (- n 1)), отображает каждое из них в некоторое значение и тоже собирает результаты в новый список.

Циклы в ISL+ имеют два отличия от циклов в традиционных языках. Во-первых, обычный цикл не создает новых данных непосредственно – для создания новых данных используются такие абстракции, как *map* и *build-list*. Во-вторых, традиционные языки часто предоставляют только фиксированное количество циклов, тогда как в ISL+ программист может определять новые циклы по мере необходимости. Иначе говоря, традиционные языки рассматривают циклы как синтаксические конструкции, подобные *local* или *cond*, и для их освоения необходимо изучить их словарь, грамматику, область видимости и назначение.

Циклы как синтаксические конструкции имеют два преимущества перед функциональными циклами, представленными в предыдущей части. С одной стороны, их форма лучше сигнализирует о намерениях, чем сочетание функций. С другой – реализации языков обычно преобразуют синтаксические циклы в более быстрые машинные команды, чем функциональные циклы. Поэтому даже функциональные языки программирования – со всем их упором на функции и композицию функций – часто предоставляют синтаксические циклы.

В этом разделе мы познакомимся с так называемыми циклами *for* в ISL+. Наша цель состоит в том, чтобы показать, как представлять обычные циклы в виде языковых конструкций и как программы, сконструированные с помощью абстракций, могут использовать циклы. В листинге 68 представлена грамматика циклов *for* в виде расширения грамматики BSL из интермеццо 1. Каждый цикл является выражением и, как и все составные конструкции, начинается с ключевого слова, за которым следует заключенная в скобки последовательность так называемых *поясняющих предложений* и одно выражение. Предложения вводят так называемые *переменные цикла*, а выражение в конце – это *тело цикла*.

Листинг 68. Грамматика циклов *for* в ISL+

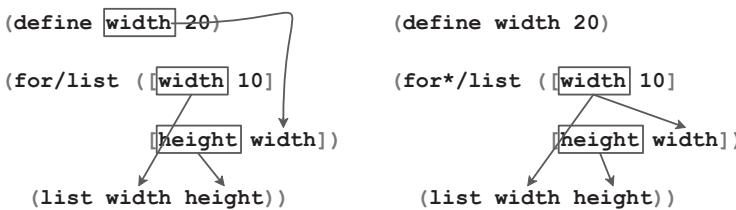
```
выражение = ...
    | (for/list (предложение предложение ...) выражение)
    | (for*/list (предложение предложение ...) выражение)
    | (for/and (предложение предложение ...) выражение)
    | (for*/and (предложение предложение ...) выражение)
    | (for/or (предложение предложение ...) выражение)
    | (for*/or (предложение предложение ...) выражение)
    | (for/sum (предложение предложение ...) выражение)
    | (for*/sum (предложение предложение ...) выражение)
    | (for/product (предложение предложение ...) выражение)
    | (for*/product (предложение предложение ...) выражение)
```

```
| (for/string (предложение предложение ...) выражение)
| (for*/string (предложение предложение ...) выражение)

предложение = [переменная выражение]
```

Даже беглый взгляд на грамматику показывает, что дюжина циклических конструкций делится на шесть пар: по одному варианту `for` и `for*` для каждого из `list`, `and`, `or`, `sum`, `product` и `string`. Все циклы `for` связывают переменные в своих предложениях с переменными в теле; варианты `for*` также связывают переменные в последующих предложениях.

Следующие два почти идентичных фрагмента кода иллюстрируют разницу между этими двумя правилами области видимости:



Синтаксические отличия состоят в том, что слева используется `for/list`, а справа `for*/list`. С точки зрения области видимости эти два цикла сильно отличаются, как показывают стрелки. В обоих вводятся переменные цикла `width` и `height`, но слева используется внешняя переменная, задающая начальное значение `height`, а справа – первая переменная цикла.

Семантически выражение `for/list` вычисляет выражения в своих предложениях, чтобы сгенерировать последовательность значений. Если в результате вычисления выражения в предложении получается:

- список, то последовательность значений формируется из его элементов;
- натуральное число n , то последовательность значений будет состоять из $0, 1, \dots, (- n 1)$;
- строка, то последовательность значений формируется из односимвольных строк.

Затем `for/list` вычисляет тело цикла, поочередно подставляя значения из сгенерированной последовательности (последовательностей) на место переменных цикла. Наконец, он собирает значения, полученные в теле цикла, в список. Вычисление выражения `for/list` останавливается по исчерпании самой короткой последовательности.

Терминология. Каждое вычисление выражения в теле цикла называется *итерацией*. Точно так же говорят, что цикл *перебирает* значения своих переменных цикла.

Основываясь на этом объяснении, мы легко можем сгенерировать список чисел от 0 до 9:

Версия этих циклов в Racket обладает более широкими возможностями, чем показано здесь, а также в языке имеется гораздо большее разнообразие циклов.

```
| > (for/list ([i 10])
|   i)
| (list 0 1 2 3 4 5 6 7 8 9)
```

Эта конструкция эквивалентна применению функции `build-list`:

```
| > (build-list 10 (lambda (i) i))
| (list 0 1 2 3 4 5 6 7 8 9)
```

Второй пример «сшивает» вместе две последовательности:

```
| > (for/list ([i 2] [j '(a b)])
|   (list i j))
| (list (list 0 'a) (list 1 'b))
```

Для сравнения ниже приводится аналогичное выражение на чистом ISL+:

```
| > (local ((define i-s (build-list 2 (lambda (i) i)))
|           (define j-s '(a b)))
|           (map list i-s j-s))
| (list (list 0 'a) (list 1 'b))
```

Последний пример подчеркивает особенности проектирования с `for/list`:

Задача. Спроектируйте функцию `enumerate`. Она должна принимать список и возвращать список с теми же элементами, объединенными с их индексами.

Стоп! Спроектируйте эту функцию, используя систематический подход и абстракции из ISL+.

С использованием `for/list` эта задача решается просто:

```
; [List-of X] -> [List-of [List N X]]
; объединяет элементы списка lx с их индексами
(define (enumerate lx)
  (for/list ([x lx] [ith (length lx)])
    (list (+ ith 1) x)))
```

Цикл `for/list` используется в теле функции для перебора элементов заданного списка и списка чисел от 0 до (`length lx`) (минус 1). А тело цикла объединяет индекс (плюс 1) с элементом списка.

С семантической точки зрения, `for*/list` выполняет итерации по последовательностям иерархически, а `for/list` просматривает их параллельно. То есть выражение `for*/list` фактически разворачивается в набор циклов:

```
| (for*/list ([i 2] [j '(a b)])
|   ...)
```

это более краткая форма записи для

```
(for/list ([i 2])
  (for/list ([j '(a b)])
    ...))
```

Кроме того, `for*/list` собирает вложенные списки в один список, объединяя их с помощью `foldl` и `append`.

Упражнение 304.

Вычислите

```
| (for/list ([i 2] [j '(a b)]) (list i j))
```

и

```
| (for*/list ([i 2] [j '(a b)]) (list i j))
```

в области взаимодействий в DrRacket. ■

Продолжим наше исследование, превратив различия областей видимости `for/list` и `for*/list` в семантические различия:

```
> (define width 2)
> (for/list ([width 3][height width])
  (list width height))
(list (list 0 0) (list 1 1))
> (for*/list ([width 3][height width])
  (list width height))
(list (list 1 0) (list 2 0) (list 2 1))
```

Чтобы лучше понять суть первого взаимодействия, вспомним, что `for/list` выполняет обход двух последовательностей параллельно и останавливается по достижении конца более короткой. В данном случае мы имеем две последовательности:

```
width = 0, 1, 2
height = 0, 1
body = (list 0 0) (list 1 1)
```

В первых двух строках перечислены значения двух переменных цикла, которые присваиваются им последовательно. Последняя строка показывает результаты итераций, что объясняет первый результат и отсутствие пары, содержащей 2.

Теперь сравним с `for*/list`:

```
width = 0 1          2
height = 0           0, 1
body =   (list 1 0)  (list 2 0) (list 2 1)
```

Если первая строка похожа на строку для `for/list`, то вторая отображает последовательности чисел в своих ячейках. Неявное вложение в `for*/list` означает, что каждая итерация повторно вычисляет `height` для определенного значения `width` и таким образом создает отдельную последовательность значений `height`. Это объясняет, почему первая ячейка в списке значений `height` пустая; в конце концов, между 0 (включительно) и 0 (исключая) нет натуральных чисел. Наконец, каждый вложенный цикл `for` дает последовательности пар, которые объединяются в список пар.

Вот пример задачи, которая иллюстрирует такое использование `for*/list`:

Задача. Спроектируйте функцию `cross`. Она должна принимать два списка, `l1` и `l2`, и создавать список пар всех элементов из этих списков.

Стоп! Найдите минутку и попробуйте спроектировать функцию, используя существующие абстракции. Руководствуйтесь при проектировании следующей таблицей:

cross	'a	'b	'c
1	(list 'a 1)	(list 'b 1)	(list 'c 1)
2	(list 'a 2)	(list 'b 2)	(list 'c 2)

В первой строке отображается содержимое списка `l1`, а в крайнем левом столбце – содержимое списка `l2`. Каждая ячейка в таблице соответствует одной из создаваемых пар.

Поскольку перечисление всех таких пар является целью `for*/list`, то определить `cross` совсем несложно:

```
| ; [List-of X] [List-of Y] -> [List-of [List X Y]]
| ; генерирует все пары из элементов списков l1 и l2
|
| (check-satisfied (cross '(a b c) '(1 2))
|                   (lambda (c) (= (length c) 6)))
|
| (define (cross l1 l2)
|   (for*/list ([x1 l1][x2 l2])
|     (list x1 x2)))
```

Мы используем здесь `check-satisfied` вместо `check-expect`, чтобы не полагаться на конкретный порядок следования пар в результате, возвращаемом `for*/list`.

Листинг 69. Компактное определение функции `arrangements` с использованием `for*/list`

```
| ; [List-of X] -> [List-of [List-of X]]
| ; создает список всех перестановок элементов в w
(define (arrangements w)
  (cond
    [(empty? w) '()]
    [else (for*/list ([item w]
                     [arrangement-without-item
                      (arrangements (remove item w))])
      (cons item arrangement-without-item))]))
```



```
| ; [List-of X] -> Boolean
(define (all-words-from-rat? w)
  (and (member? (explode "rat") w)
       (member? (explode "art") w)
       (member? (explode "tar") w)))
```



```
(check-satisfied (arrangements '("r" "a" "t"))
                  all-words-from-rat?)
```

ПРИМЕЧАНИЕ. В листинге 69 показан еще один пример применения `for*/list`. Это компактное решение задачи по созданию всех возможных перестановок букв в заданном списке.

В разделе 12.4 было показано не самое лучшее определение этой сложной программы, в отличие от решения в листинге 69, где используются удивительные возможности `for*/list` и необычная форма рекурсии цикла, уместившегося всего в пять строк кода. Этот пример представлен здесь, только чтобы наглядно показать силу абстракций, а описание процесса проектирования вы найдете в упражнении 477. **КОНЕЦ.**

Мы благодарим Марка Энгельберга (*Mark Engelberg*) за этот весьма выразительный пример.

Окончание `.../list` явно указывает, что выражение цикла создает список. Кроме того, в библиотеке есть циклы `for` и `for*`, имеющие одинаковые окончания:

- `.../and` собирает значения всех итераций с помощью `and`:

```
| > (for/and ([i 10]) (> (- 9 i) 0))
| #false
| > (for/and ([i 10]) (if (>= i 0) i #false))
| 9
```

Для прагматиков: этот цикл возвращает последнее сгенерированное значение или `#false`;

- `.../or` действует подобно `.../and`, но использует `or` вместо `and`:

```
| > (for/or ([i 10]) (if (= (- 9 i) 0) i #false))
| 9
| > (for/or ([i 10]) (if (< i 0) i #false))
| #false
```

Этот цикл возвращает первое значение, отличное от `#false`;

- `.../sum` складывает числа, получаемые в результате итераций:

```
| > (for/sum ([c "abc"]) (string->int c))
| 294
```

- `.../product` перемножает числа, получаемые в результате итераций:

```
| > (for/product ([c "abc"]) (+ (string->int c) 1))
| 970200
```

- `.../string` создает строки из последовательностей 1String (символов):

```
| > (define a (string->int "a"))
| > (for/string ([j 10]) (int->string (+ a j)))
| "abcdefghijklm"
```

Стоп! Представьте, как будет работать цикл `for/fold`.

Стоп еще раз! Было бы поучительно переформулировать все выше-перечисленные примеры с использованием абстракций, имеющих-

ся в ISL+. Такое упражнение помогло бы понять, как проектировать функции с циклами `for` вместо абстрактных функций. **Подсказка.** Спроектируйте функции `and-map` и `or-map`, действующие подобно `and-map` и `or-map` соответственно, но возвращающие соответствующие значения, отличные от `# false`.

Цикл по числам не всегда сводится к перечислению от 0 до (`- n 1`). Часто программам необходимо перебирать числа, следующие не по порядку, а иногда требуется неограниченная последовательности чисел. Чтобы освоить эту форму программирования, Racket предлагает функции, которые генерируют последовательности, и в листинге 70 представлены две из них.

Листинг 70. Конструирование последовательностей натуральных чисел

```
; N -> sequence?
; конструирует бесконечную последовательность натуральных чисел,
; начинающуюся с n
(define (in-naturals n) ...)

; N N N -> sequence?
; конструирует следующую конечную последовательность натуральных чисел:
;   start
;   (+ start step)
;   (+ start step step)
;   ...
; пока очередное число не превысит значение end
(define (in-range start end step) ...)
```

С помощью первой можно немного упростить функцию `enumerate`:

```
(define (enumerate.v2 lx)
  (for/list ([item lx] [ith (in-naturals 1)])
    (list ith item)))
```

Здесь `in-naturals` используется для получения бесконечной последовательности натуральных чисел, начиная с 1; цикл `for` останавливается по достижении конца `l`.

С помощью второй функции можно, например, обойти в цикле все четные числа, которые меньше `n`:

```
; N -> Number
; складывает четные числа между 0 и n (не включая)
(check-expect (sum-evens 2) 0)
(check-expect (sum-evens 4) 2)
(define (sum-evens n)
  (for/sum ([i (in-range 0 n 2)]) i))
```

Такое применение может показаться тривиальным, но многие задачи в математике требуют именно таких циклов, и именно поэтому подобные функции, как `in-range`, встречаются во многих языках программирования.

Упражнение 305. Определите `convert-euro` с использованием циклов; см. упражнение 267. ■

Упражнение 306. Используйте циклы и определите функцию, которая:

- 1) создает список (`list 0 ... (- n 1)`) для любого натурального числа n ;
- 2) создает список (`list 1 ... n`) для любого натурального числа n ;
- 3) создает список (`list 1 1/2 ... 1/n`) для любого натурального числа n ;
- 4) создает список первых n четных чисел;
- 5) создает диагональную квадратную матрицу из нулей и единиц; см. упражнение 262.

Наконец, определите с использованием циклов функцию `tabulate` из упражнения 250. ■

Упражнение 307. Определите функцию `find-name`. Она должна принимать имя и список имён и извлекать из списка первое имя, совпадающее с заданным или содержащее его.

Определите функцию, которая гарантирует, что ни одно имя в заданном списке имён не длиннее заданного значения. Сравните с упражнением 271. ■

Сопоставление с образцом

Проектируя функцию для определения данных с шестью предложениями, мы используем выражение `cond` с шестью условиями. Формулируя одно из предложений `cond`, мы используем предикат, чтобы определить, должно ли это предложение обрабатывать данное значение, и если да, то добавляем селекторы для извлечения любых составных значений. Эта идея многократно объяснялась в трех первых частях книги.

Заинтересованный преподаватель может пожелать изучить определения алгебраических типов данных, имеющиеся в библиотеке `2htdp/abstraction`.

Для устранения таких повторений необходима абстракция. В части III объясняется, как создавать некоторые из таких абстракций, но шаблон «предикат/селектор» может быть реализован только разработчиком языка. В частности, разработчики языков функционального программирования осознали необходимость абстрагирования этого повторяющегося использования предикатов и селекторов, поэтому в этих языках имеется языковая конструкция *сопоставления с образцом*, которая объединяет и упрощает подобные предложения `cond`.

В этом разделе представлена упрощенная схема сопоставления с образцом в Racket. В листинге 71 показана грамматика этой конструкции; как видите, сопоставление – это синтаксически сложная конструкция. Ее схема похожа на схему `cond`, но в ней используются образцы (шаблоны), а не условия, и они имеют свои правила.

Листинг 71. Выражение `match` в *ISL+*

```
выражение = ...
| (match выражение [образец выражение] ...)
образец = переменная
| лiteralная константа
| (cons образец образец)
```

| (имя-структурой *образец* ...)
| (? имя-предиката)

Проще говоря, выражение

```
(match выражение
  [образец1 выражение1]
  [образец2 выражение2]
  ...)
```

действует подобно выражению `cond`, то есть вычисляет выражение и последовательно пытается сопоставить результат с образцом₁, образцом₂ и т. д., пока не достигнет успеха с образцом_i, после чего вычисляет значение выражения_i, которое становится результатом всего выражения `match`.

Ключевое отличие состоит в том, что `match` образует новую область видимости, что лучше всего иллюстрирует следующий снимок экрана DrRacket:

```
(define (sum-items a-lon)
  (match a-lon
    [(cons fst '()) → fst]
    [(cons fst rst)
     (+ fst (sum-items rst))]))
```

Как показано на рисунке, каждое предложение с образцом связывает переменные. Более того, область видимости связанных переменных ограничивается телом предложения, поэтому даже если два образца используют одни и те же имена переменных в связывающих вхождениях, как в примере выше, соответствующие им связанные вхождения никак не влияют друг на друга.

Синтаксически образец напоминает вложенные структуры, листьями которых являются литеральные константы, переменные или предикаты.

| (? имя-предиката)

В последнем случае имя-предиката должно ссылаться на функцию-предикат, доступную в данной области видимости, то есть на функцию, которая принимает одно значение и возвращает логическое значение.

Семантически образец должен совпадать со значением *v*. Если образец является:

- литеральной константой, то значение должно совпадать только с этой литературальной константой:

```
> (match 4
  ['four 1]
  ["four" 2]
  [#true 3])
```

```

| [4      "hello world"])
| "hello world"

```

- переменной, то он должен совпадать с любым значением и связываться с этой переменной в ходе вычисления соответствующего предложения:

```

| > (match 2
|   [3 "one"]
|   [x (+ x 3)])
| 5

```

Здесь число 2 не совпадает с первым образцом, который является литеральной константой 3, поэтому `match` находит совпадение между 2 и вторым образцом, который является простой переменной и по этой причине соответствует любому значению. Как результат `match` выбирает второе предложение и вычисляет его тело, связывая переменную `x` со значением 2;

- выражением (`cons` образец₁ образец₂), то он должен совпадать с экземпляром `cons`, если первое поле соответствует образцу₁ а остальные – образцу₂:

```

| > (match (cons 1 '())
|   [(cons 1 tail) tail]
|   [(cons head tail) head])
| '()
| > (match (cons 2 '())
|   [(cons 1 tail) tail]
|   [(cons head tail) head])
| 2

```

Эти взаимодействия показывают, как `match` сначала раскладывает `cons` на составляющие, а затем использует литературные константы и переменные в роли листьев для сопоставления с заданным списком;

- выражением (имя-структуры образец₁ ... образец_n), то он должен совпадать только со структурой имя-структуры, значения полей в которой соответствуют образцу₁, ..., образцу_n:

```

| > (define p (make-posn 3 4))
| > (match p
|   [(posn x y) (sqrt (+ (sqr x) (sqr y)))])
| 5

```

Очевидно, что сопоставление экземпляра `Posn` с образцом осуществляется подобно сопоставлению с образцом `cons`. Но обратите внимание, что в образце используется имя `posn`, а не имя конструктора.

В сопоставлении также можно использовать свои определения структур:

```

| > (define-struct phone [area switch four])
| > (match (make-phone 713 664 9993)

```

```
| [(phone x y z) (+ x y z)])
11370
```

И снова в образце используется имя структуры `phone`, а не конструктор.

Наконец, в сопоставлении могут участвовать вложенные конструкции:

```
| > (match (cons (make-phone 713 664 9993) '())
|   [(cons (phone area-code 664 9993) tail)
|     area-code])
|   713
```

Это выражение `match` извлекает код города из телефонного номера в списке, если код коммутатора (`switch`) равен 664, а последние четыре цифры – 9993;

- выражением (? имя-предиката), то он должен совпадать, только если применение (имя-предиката v) вернет #`true`:

```
| > (match (cons 1 '())
|   [(cons (? symbol?) tail) tail]
|   [(cons head tail) head])
| 1
```

Это выражение возвращает 1 – результат второго предложения, потому что 1 не является символом.

Стоп! Поэкспериментируйте с разными выражениями `match`, прежде чем читать дальше.

Теперь пришла пора продемонстрировать полезность `match`:

Задача. Спроектируйте функцию `last-item`, которая извлекает последний элемент из непустого списка. Напомним, что непустые списки определяются следующим образом:

```
| ; [Non-empty-list X] -- это одно из значений:
| ; -- (cons X '())
| ; -- (cons X [Non-empty-list X])
```

Стоп! Эта задача рассматривалась в части II. Найдите решение.

Используя `match`, можно исключить из решения три селектора и два предиката:

```
| ; [Non-empty-list X] -> X
| ; извлекает последний элемент из не-l
| (check-expect (last-item '(a b c)) 'c)
| (check-error (last-item '()))
| (define (last-item ne-l)
|   (match ne-l
|     [(cons lst '()) lst]
|     [(cons fst rst) (last-item rst)])))
```

Вместо предикатов и селекторов в этом решении используются образцы, аналогичные тем, что указаны в определении данных. Для каждой ссылки на себя и набора параметров из определения данных

образцы используют переменные уровня программы. Тела предложений в выражении `match` больше не извлекают соответствующие элементы из списка с помощью селекторов, а просто ссылаются на соответствующие имена. Как и раньше, функция выполняет рекурсию для поля `rest` в заданном списке `cons`, потому что в этой позиции определение данных ссылается на самого себя. В базовом случае результатом является `lst` – переменная, которая обозначает последний элемент в списке.

Рассмотрим вторую задачу из части II:

Задача. Спроектируйте функцию `depth`, которая подсчитывает количество матрешек, в которые вложена данная. Вот определение данных:

```
(define-struct layer [color doll])
; RD.v2 (сокращенно от Russian doll -- русская матрешка) -- это одно из значений:
; -- "doll"
; -- (make-layer String RD.v2)
```

Вот определение `depth`, использующее выражение `match`:

```
; RD.v2 -> N
; сколько матрешек в данном наборе ап-гд
(check-expect (depth (make-layer "red" "doll")) 1)
(define (depth a-doll)
  (match a-doll
    ["doll" 0]
    [(layer c inside) (+ (depth inside) 1)]))
```

Образец в первом предложении `match` ищет строку `"doll"`, а во втором соответствует любой структуре `layer`, связывая с со значением в поле `color`, а `inside` – со значением в поле `doll`. Проще говоря, `match` снова сокращает определение функции.

Последняя задача затрагивает фрагмент кода из игры «Космические захватчики»:

Задача. Спроектируйте функцию `move-right`. Она должна принимать список экземпляров `Posn`, представляющих позиции объектов на холсте, и число и добавлять заданное число к координате `X` каждого объекта, чтобы те сдвинулись вправо.

Вот наше решение, использующее всю мощь ISL+:

```
; [List-of Posn] -> [List-of Posn]
; смещает все объекты вправо на delta-x пикселей

(define input `((,(make-posn 1 1) ,(make-posn 10 14)))
  (define expect `((,(make-posn 4 1) ,(make-posn 13 14)))

  (check-expect (move-right input 3) expect)

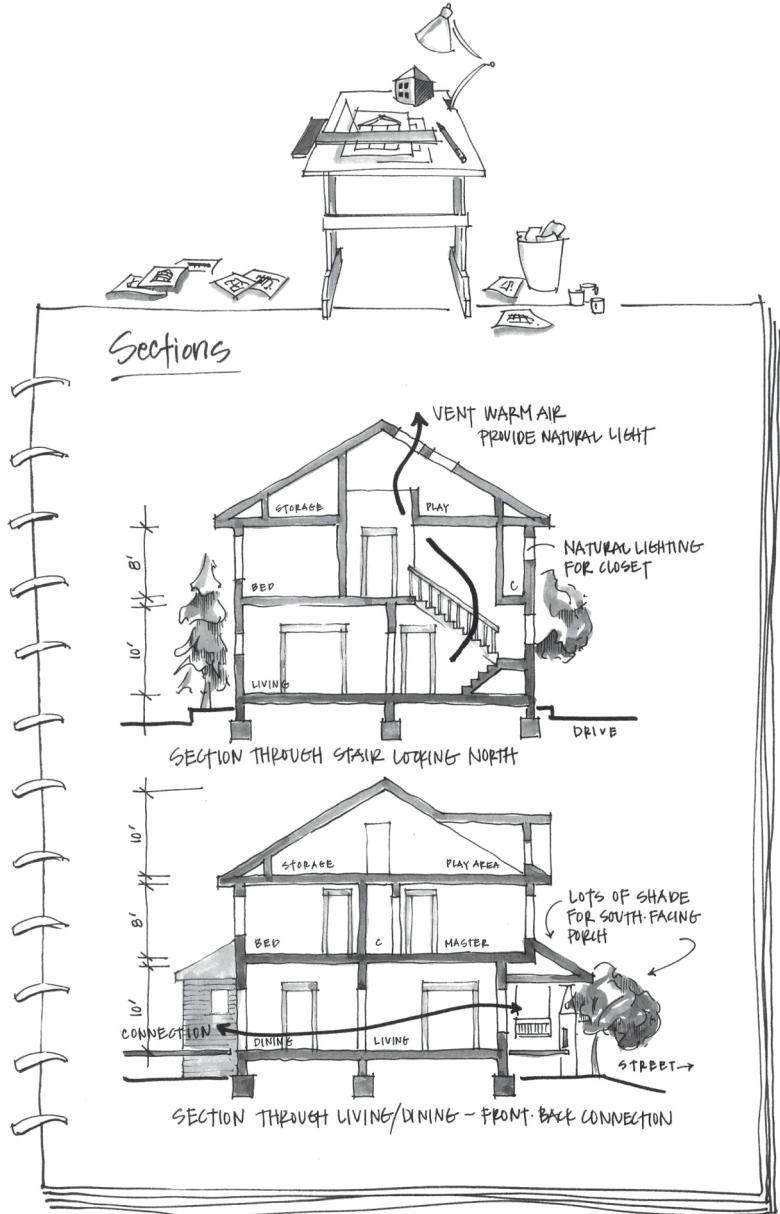
  (define (move-right lop delta-x)
    (for/list ((p lop))
      (match p
        [(posn x y) (make-posn (+ x delta-x) y)])))
```

Стоп! Заметили, что здесь используются определения `define` для тестов? Если дать примерам ожидаемых результатов функций хорошие имена с помощью `define` и записать их рядом с тестами, то позднее вам проще будет читать код.

Стоп! Как сравнить решения с `cond` и селекторами? Запишите их рядом друг с другом и сравните. Какое из решений вам нравится больше?

Упражнение 308. Спроектируйте функцию `replace`, которая меняет код зоны 713 на 281 в списке записей `phone` с телефонными номерами. ■

Упражнение 309. Спроектируйте функцию `words-on-line`, которая определяет количество строк в каждом элементе в списке списков строк. ■



IV ПЕРЕПЛЕТАЮЩИЕСЯ ДАННЫЕ

Может показаться, что определения данных для списков и натуральных чисел довольно необычны. Они ссылаются на самих себя и, по всей вероятности, являются первыми такими определениями, с которыми вы когда-либо сталкивались. Но, как оказывается, многие классы данных имеют еще более сложные определения. Некоторые обобщения могут включать множество ссылок на самих себя в одном определении или набор определений данных, ссылающихся друг на друга. Эти формы данных стали повсеместными, поэтому программист должен научиться работать с любым набором определений данных. И в этом суть рецепта проектирования.

Эта часть книги начинается с обобщения рецепта проектирования, чтобы его можно было применить ко всем формам определений структурных данных. Затем мы познакомимся со строгим обоснованием понятия итеративного уточнения из главы 12, потому что сложные определения данных разрабатываются не одним махом, а в несколько этапов. Использование приема итеративного уточнения – одна из причин, почему все программисты являются немного учеными, а наша дисциплина – информатика – считается наукой. Две последние главы в этой части иллюстрируют эти идеи: в одной объясняется, как спроектировать интерпретатор для BSL, а в другой рассказывается об обработке XML, языка обмена данными в интернете. Последняя глава вновь расширяет рецепт проектирования, добавляя в него функции, одновременно обрабатывающие два сложных аргумента.

19. Поэзия S-выражений

Программирование сродни поэзии. Подобно поэтам, программисты совершенствуют свои навыки на, казалось бы, бессмысленных идеях. Они все время пересматривают и редактируют свой код, как объясняется в предыдущей главе. В этой главе мы будем знакомиться со все более сложными формами данных без видимой реальной цели. Выбранные нами типы данных являются до крайности академическими, и маловероятно, что вы когда-либо столкнетесь с ними снова.

Тем не менее эта глава демонстрирует всю мощь рецепта проектирования и знакомит с типами данных, с которыми способны справиться реальные программы. Чтобы связать этот материал с тем, с чем вы столкнетесь в своей карьере программиста, мы озаглавили каждый раздел соответствующими названиями: деревья, леса, XML. Последнее название не совсем точное, потому что на самом деле в этом разделе рассматриваются S-выражения, а связь между S-выражениями и XML разъясняется в главе 22, которая, в отличие от этой главы, намного ближе к реальному использованию сложных форм данных.

19.1. Деревья

У всех нас есть родословная. Один из способов нарисовать генеалогическое древо – добавлять новый элемент с рождением каждого ребенка и соединять линиями этот элемент с элементами отца и матери. Тем, чьи родители неизвестны, построить такое древо не удастся. В результате получается *генеалогическое древо предков*, перечисляющее всех известных предков человека.

На рис. 15 показано трехуровневое генеалогическое древо. Густав (Gustav) – сын Евы (Eva) и Фреда (Fred), а Ева – дочь Карла (Carl) и Беттины (Bettina). Помимо имен людей и родственных отношений, в древе также записаны год рождения и цвет глаз. Основываясь на этой схеме, можно легко представить генеалогическое древо, насчитывающее многие поколения и отражающее другие сведения.

Большое генеалогическое древо предпочтительнее представить в виде данных и спроектировать программы, обрабатывающие такие данные. Учитывая, что точка в генеалогическом древе объединяет пять видов информации – ссылки на отца и мать, имя, дату рождения и цвет глаз, – мы должны определить тип структуры:

```
| (define-struct child [father mother name date eyes])
```

Определение типа структуры приводит к определению данных:

```
| ; Child – это структура:  
|   ; (make-child Child Child String N String)
```

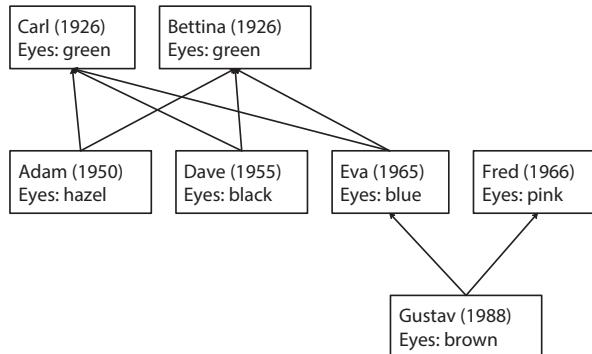


Рис. 15. Генеалогическое древо

Это определение данных настолько же простое, насколько и бесполезное. Оно ссылается на самого себя, но, так как в нем нет предложений, невозможно создать правильный экземпляр `Child`. Проще говоря, нам пришлось бы писать

```
| (make-child (make-child (make-child ... ) ...) ...)
```

без конца. Чтобы избежать таких бессмысленных определений данных, мы требуем, чтобы определение данных, ссылающееся на самого себя, имело несколько предложений и хотя бы одно из них не ссылалось на это определение данных.

Давайте отложим на время определение данных и поэксперименируем. Предположим, мы собираемся добавить ребенка в существующее генеалогическое древо и у нас уже есть представления его родителей. В этом случае мы можем просто создать новый экземпляр структуры `child`. Например, чтобы представить Адама (`Adam`) в программе, которая уже содержит представления Карла и Беттины, достаточно добавить следующую структуру `child`:

```
| (define Adam
|   (make-child Carl Bettina "Adam" 1950 "hazel"))
```

если предположить, что Карл и Беттина являются родителями Адама.

С другой стороны, родители могут быть неизвестны, как, например, у Беттины в генеалогическом древе на рис. 15. Но даже в этом случае мы должны заполнить соответствующие поля в представлении `child`. Какие бы данные мы ни выбрали, они должны сигнализировать об отсутствии информации. С одной стороны, мы могли бы использовать `#false`, `"none"` или `'()`. С другой – поскольку мы должны сообщить от отсутствии генеалогической информации, то лучше всего добавить еще один структурный тип с соответствующим именем:

```
| (define-struct no-parent [])
```

Теперь мы можем создать структуру `child` для Беттины, записав:

```
(make-child (make-no-parent)
            (make-no-parent)
            "Bettina" 1926 "green")
```

Конечно, если отсутствует информация только об одном из родителей, то мы заполняем этим специальным значением только соответствующее поле.

Наш эксперимент позволяет сделать два вывода. Во-первых, мы должны искать не определение данных, описывающее порядок создания экземпляров структур `child`, а определение данных, описывающее порядок представления родословной. Во-вторых, определение данных состоит из двух предложений, один вариант для описания неизвестных родословных, а другой – для известных:

```
(define-struct no-parent [])
(define-struct child [father mother name date eyes])
; FT (сокращенно от family tree – генеалогическое древо) -- одно из значений:
; -- (make-no-parent)
; -- (make-child FT FT String N String)
```

Поскольку элементы «`no parent`» будут часто встречаться в наших программах, определим сокращение `NP` и немного изменим определение данных:

```
(define NP (make-no-parent))
; FT – одно из значений:
; -- NP
; -- (make-child FT FT String N String)
```

Следуя рецепту проектирования из главы 9, создадим несколько примеров генеалогических древ для этого определения данных. В частности, преобразуем в пример генеалогическое древо на рис. 15. Информация для Карла преобразуется в такие данные:

```
| (make-child NP NP "Carl" 1926 "green")
```

Беттина и Фред представлены похожими экземплярами `child`. В экземпляр, представляющий Адама, требуется добавить две структуры `child`, по одной для Карла и Беттины:

```
(make-child (make-child NP NP "Carl" 1926 "green")
            (make-child NP NP "Bettina" 1926 "green"))
            "Adam"
            1950
            "hazel")
```

Поскольку структуры с данными о Карле и Беттине также необходимы для создания структур, представляющих Дейва (Dave) и Еву, хорошо бы ввести именованные определения, представляющие конкретные экземпляры `child`, и использовать их имена в других местах. В листинге 72 иллюстрируется этот подход на примере полного представления данных генеалогического древа, изображенного на рис. 15.

Листинг 72. Представление данных для простого генеалогического древа

```

; Старое поколение:
(define Carl (make-child NP NP "Carl" 1926 "green"))
(define Bettina (make-child NP NP "Bettina" 1926 "green"))

; Среднее поколение:
(define Adam (make-child Carl Bettina "Adam" 1950 "hazel"))
(define Dave (make-child Carl Bettina "Dave" 1955 "black"))
(define Eva (make-child Carl Bettina "Eva" 1965 "blue"))
(define Fred (make-child NP NP "Fred" 1966 "pink"))

; Молодое поколение:
(define Gustav (make-child Fred Eva "Gustav" 1988 "brown"))

```

Внимательно рассмотрите эти определения, потому что древо послужит нам рабочим примером в следующем упражнении.

Отложим пока проектирование конкретной функции для применения к генеалогическому древу и рассмотрим общую организацию такой функции. То есть попробуем проследовать за рецептом проектирования, насколько это возможно, не сосредоточиваясь на какой-либо конкретной задаче. Начнем с заголовка, то есть с шага 2 рецепта:

```

; FT -> ???
; ...
(define (fun-FT an-ftree) ...)

```

Несмотря на то что мы не указали назначение функции, мы знаем, что она применяется к генеалогическому древу. Многоточие «...» в сигнатуре говорит о том, что мы не знаем, какие данные возвращает функция, и напоминает нам, что мы не знаем ее назначения.

Не зная назначения, мы не можем создать примеры применения функции. Но мы можем использовать организацию определения данных FT для проектирования макета. Поскольку определение состоит из двух предложений, макет должен состоять из выражения cond с двумя предложениями:

```

(define (fun-FT an-ftree)
  (cond
    [(no-parent? an-ftree) ...]
    [else ...]))

```

Если аргумент fun-FT удовлетворяет предикату no-parent?, то это означает, что структура не содержит дополнительных данных, и первое предложение на этом завершается. Во втором предложении входные данные содержат пять элементов, на которые мы ссылаемся с помощью пяти селекторов:

```

; FT -> ???
(define (fun-FT an-ftree)
  (cond
    [(no-parent? an-ftree) ...]
    [else (... (child-father an-ftree) ...
               ... (child-mother an-ftree) ...
               ...)])

```

```

| ... (child-name an-ftree) ...
| ... (child-date an-ftree) ...
| ... (child-eyes an-ftree) ...])))
```

Последнее дополнение в макет касается ссылок определения на самого себя. Если определение данных ссылается само на себя, то функция почти наверняка должна быть рекурсивной. Определение FT имеет две ссылки на себя, поэтому нужно добавить в макет два рекурсивных вызова:

```

; FT -> ???
(define (fun-FT an-ftree)
  (cond
    [(no-parent? an-ftree) ...]
    [else (... (fun-FT (child-father an-ftree)) ...
                ... (fun-FT (child-mother an-ftree)) ...
                ... (child-name an-ftree) ...
                ... (child-date an-ftree) ...
                ... (child-eyes an-ftree) ...))))
```

В частности, fun-FT применяется во втором предложении cond к представлениям данных для отца и матери, потому что второе предложение в определении данных содержит соответствующие ссылки на себя.

А теперь обратимся к конкретному примеру – функции blue-eyed-child?. Ее назначение – определить наличие в заданном генеалогическом древе структуры child, описывающей человека с голубыми глазами. Вы можете скопировать и переименовать fun-FT, чтобы получить нужный макет; замените «???» словом «Boolean» и добавьте описание назначения:

```

; FT -> Boolean
; определяет наличие в an-ftree структуры child,
; описывающей человека со строкой "blue" в поле eyes
(define (blue-eyed-child? an-ftree)
  (cond
    [(no-parent? an-ftree) ...]
    [else (... (blue-eyed-child?
                 (child-father an-ftree)) ...
                ... (blue-eyed-child?
                     (child-mother an-ftree)) ...
                ... (child-name an-ftree) ...
                ... (child-date an-ftree) ...
                ... (child-eyes an-ftree) ...))))
```

Далее, следуя рецепту проектирования, мы должны заменить обобщенное имя макета конкретным.

Сверившись с нашим рецептом, мы понимаем, что нужно вернуться на шаг назад и создать несколько примеров, и только потом приступить к определению тела функции. Если начать с Карла, первого в генеалогическом древе, то можно заметить, что в его генеалогическом древе нет экземпляра child со строкой "blue" в поле eyes. В частности, экземпляр child, представляющий самого Карла, содержит в поле eyes

строку "green", а учитывая, что предки Карла неизвестны, в его древе не может быть экземпляра `child` со строкой "blue" в поле `eyes`:

```
| (check-expect (blue-eyed-child? Carl) #false)
```

В записи с информацией о Густаве (Gustav), напротив, имеется ссылка на структуру `child` для Евы, у которой голубые глаза:

```
| (check-expect (blue-eyed-child? Gustav) #true)
```

Теперь все готово для определения фактической функции. Функция различает два случая: `no-parent` и `child`. В первом случае ответ очевиден, хотя мы не приводили никаких примеров. Поскольку данное генеалогическое древо вообще не содержит записей `child`, в нем не может быть структур `child` со строкой "blue" в поле `eyes`. Следовательно, результатом первого предложения в `cond` будет значение `#false`.

Чтобы определить результат второго предложения в `cond`, требуется приложить чуть больше усилий. И снова, следуя рецепту проектирования, мы сначала напоминаем себе, что делают выражения в макете.

1. В соответствии с описанием назначения функции выражение

```
| (blue-eyed-child? (child-father an-ftree))
```

определяет, имеется ли в отцовском древе FT структура `child`, представляющая человека с голубыми глазами.

2. Точно так же выражение `(blue-eyed-child? (child-mother an-ftree))` определяет наличие в материнском древе FT искомой структуры `child`.
3. Выражения с селекторами `(child-name an-ftree)`, `(child-date an-ftree)` и `(child-eyes an-ftree)` извлекают имя, дату рождения и цвет глаз из заданной структуры `child` соответственно.

Теперь нам осталось лишь выяснить, как объединить эти выражения.

Очевидно, что если структура `child` содержит строку "blue" в поле `eyes`, то функция должна вернуть `#true`. Далее, выражения, извлекающие имя и дату рождения, бесполезны в данном контексте. Как уже говорилось, выражение `(blue-eyed-child? (child-father an-ftree))` выполняет обход дерева со стороны отца, а выражение `(blue-eyed-child? (child-mother an-ftree))` – со стороны матери. Если любое из этих выражений вернет `#true`, это означает, что в `an-ftree` имеется структура `child`, представляющая человека с голубыми глазами.

Наш анализ показывает, что функция должна вернуть `#true`, если `#true` возвращает одно из следующих выражений:

- `(string=? (child-eyes an-ftree) "blue")`;
- `(blue-eyed-child? (child-father an-ftree))`;
- `(blue-eyed-child? (child-mother an-ftree))`.

А это означает, что мы должны объединить их с помощью оператора or:

```
(or (string=? (child-eyes an-ftree) "blue")
    (blue-eyed-child? (child-father an-ftree))
    (blue-eyed-child? (child-mother an-ftree)))
```

В листинге 73 показано законченное определение функции.

Листинг 73. Поиск экземпляра child, представляющего человека с голубыми глазами

```
; FT -> Boolean
; определяет наличие в an-ftree структуры child,
; описывающей человека со строкой "blue" в поле eyes

(check-expect (blue-eyed-child? Carl) #false)
(check-expect (blue-eyed-child? Gustav) #true)

(define (blue-eyed-child? an-ftree)
  (cond
    [(no-parent? an-ftree) #false]
    [else (or (string=? (child-eyes an-ftree) "blue")
              (blue-eyed-child? (child-father an-ftree))
              (blue-eyed-child? (child-mother an-ftree))))]))
```

Поскольку это наша первая функция, в которой имеется два рекурсивных вызова, мы покажем вам, как вычисляется (blue-eyed-child? Carl) в движке пошагового выполнения, чтобы вы могли понять, как все это работает:

```
(blue-eyed-child? Carl)
==
(blue-eyed-child?
 (make-child NP NP "Carl" 1926 "green"))
```

Давайте будем действовать так, как если бы NP было значением, и используем carl для представления экземпляра child:

```
 ==
(cond
  [(no-parent?
    (make-child NP NP "Carl" 1926 "green"))
   #false]
  [else (or (string=? (child-eyes carl) "blue")
            (blue-eyed-child? (child-father carl))
            (blue-eyed-child? (child-mother carl))))])
```

После отбрасывания первого предложения в выражении cond можно заменить carl соответствующими значениями и выполнить три вспомогательных вычисления, показанных в листинге 74. После их подстановки остальные вычисления легко объяснить:

```
 ==
(or (string=? "green" "blue")
     (blue-eyed-child? (child-father carl)))
```

```

  (blue-eyed-child? (child-mother carl)))
== (or #false #false #false)
== #false

```

Мы уверены, что вы видели подобные вспомогательные вычисления на уроках математики, но мы хотели бы особо отметить, что движок пошаговых вычислений будет выполнять не все вычисления, а только те, которые абсолютно необходимы.

Листинг 74. Вычисления с древами

```

1. (child-eyes (make-child NP NP "Carl" 1926 "green"))
==
"green"

2. (blue-eyed-child?
  (child-father
    (make-child NP NP "Carl" 1926 "green"))))
==
(blue-eyed-child? NP)
==
#false

3. (blue-eyed-child?
  (child-mother
    (make-child NP NP "Carl" 1926 "green"))))
==
(blue-eyed-child? NP)
==
#false

```

Упражнение 310. Определите функцию `count-persons`, которая принимает генеалогическое дерево и подсчитывает количество структур `child` в нем. ■

Упражнение 311. Определите функцию `average-age`, которая принимает генеалогическое дерево и текущий год и вычисляет средний возраст по всем структурам `child` в этом дереве. ■

Упражнение 312. Определите функцию `eye-colors`, которая принимает генеалогическое дерево и возвращает список всех цветов глаз в дереве. Один и тот же цвет глаз может встречаться несколько раз в списке с результатами. **Подсказка.** Используйте `append` для объединения списков, полученных рекурсивными вызовами. ■

Упражнение 313. Представьте, что нам потребовалась функция `blue-eyed-ancestor?`, которая действует подобно функции `blue-eyed-child?`, но возвращает `#true`, только когда голубые глаза были у одного из предков, но не у человека, представленного данной структурой `child`.

Несмотря на разное назначение, эти функции имеют одинаковые сигнатуры:

```

; FT -> Boolean
(define (blue-eyed-ancestor? an-ftree) ...)

```

Стоп! Сформулируйте описание назначения для функции.

Чтобы оценить разницу, возьмем экземпляр `child` с данными для Евы:

```
| (check-expect (blue-eyed-child? Eva) #true)
```

У Евы голубые глаза, но у нее нет известных предков с голубыми глазами. Поэтому

```
| (check-expect (blue-eyed-ancestor? Eva) #false)
```

Напротив, Густав – сын Евы, и у него есть предок с голубыми глазами:

```
| (check-expect (blue-eyed-ancestor? Gustav) #true)
```

Теперь допустим, что наш друг предложил такое решение:

```
(define (blue-eyed-ancestor? an-ftree)
  (cond
    [(no-parent? an-ftree) #false]
    [else
      (or
        (blue-eyed-ancestor?
          (child-father an-ftree))
        (blue-eyed-ancestor?
          (child-mother an-ftree))))]))
```

Объясните, почему эта функция терпит неудачу в одном из тестов. Какой результат вернет `(blue-eyed-ancestor? A)` независимо от выбора `A`? Попробуйте исправить это решение. ■

19.2. Леса

От генеалогического дерева до генеалогического леса всего несколько шагов:

```
; FF (сокращенно от family forest – генеалогический лес) – это одно из значений:
; -- '()
; -- (cons FT FF)
; интерпретация: генеалогический лес представляет несколько семей
; (например, в городе) и их генеалогические деревья
```

Вот несколько фрагментов из дерева на рис. 15, представленных в виде лесов:

```
(define ff1 (list Carl Bettina))
(define ff2 (list Fred Eva))
(define ff3 (list Fred Eva Carl))
```

Первые два леса содержат две несвязанные семьи, а третий показывает, что, в отличие от настоящих лесов, деревья в генеалогических лесах могут перекрываться.

Теперь рассмотрим типичную задачу, касающуюся генеалогических древ:

Задача. Спроектируйте функцию `blue-eyed-child-in-forest?`, которая определяет наличие в заданном лесу генеалогических древ структуры `child` со строкой "blue" в поле `eyes`.

Листинг 75. Поиск экземпляра `child`, представляющего человека с голубыми глазами, в лесу генеалогических древ

```
; FF -> Boolean
; определяет наличие в лесу a-forest структуры child,
; описывающей человека со строкой "blue" в поле eyes

(check-expect (blue-eyed-child-in-forest? ff1) #false)
(check-expect (blue-eyed-child-in-forest? ff2) #true)
(check-expect (blue-eyed-child-in-forest? ff3) #true)

(define (blue-eyed-child-in-forest? a-forest)
  (cond
    [(empty? a-forest) #false]
    [else
      (or (blue-eyed-child? (first a-forest))
          (blue-eyed-child-in-forest? (rest a-forest)))]))
```

В листинге 75 показано простое решение этой задачи. Изучите сигнатуру, описание назначения и примеры самостоятельно. В данном случае мы делаем упор на организацию программы. Что касается макета, то за основу можно взять макет функции, обрабатывающей списки, потому что функция принимает список. Если бы все элементы в списке были структурами с полем `eyes` и ничем другим, то функция могла бы просто перебрать эти структуры, используя селектор для извлечения поля `eyes` – сравнение строк. Однако в данном случае каждый элемент является генеалогическим древом, но, к счастью, мы уже знаем, как обрабатывать такие древа.

Давайте сделаем шаг назад и вернемся к нашему описанию листинга 75. Отправной точкой является **пара** определений данных, в которой второе определение ссылается на первое и оба ссылаются на самих себя. В результате получается **пара** функций, в которой вторая ссылается на первую и обе ссылаются на самих себя. Иначе говоря, определения функций ссылаются друг на друга так же, как соответствующие определения данных. В первых главах мы умолчали о таких отношениях, но сейчас ситуация существенно сложнее и потому заслуживает особого внимания.

Упражнение 314. Переформулируйте определение данных для FF, используя абстракцию `List-of`. А затем проделайте то же самое для функции `blue-eyed-child-in-forest?`. Наконец, определите `blue-eyed-child-in-forest?`, используя одну из абстракций для работы со списками из предыдущей главы. ■

Упражнение 315. Спроектируйте функцию `average-age`. Она должна принимать лес генеалогических древ и год (N) и на основе этих

данных вычислять средний возраст по всем структурам child в этом лесу. **Примечание.** Если деревья в лесу перекрываются, то результат не является истинным средним значением, потому что одни люди будут вносить больший вклад, чем другие. В этом упражнении вы должны действовать так, будто деревья не пересекаются. ■

19.3. S-выражения

В интермеццо 2 мы дали неформальное введение в S-выражения, теперь представим их официально. Любое S-выражение можно описать с использованием комбинации из трех определений данных:

```
; S-expr -- это одно из значений:
; -- Atom
; -- SL

; SL -- это одно из значений:
; -- '()
; -- (cons S-expr SL)

; Atom -- это одно из значений:
; -- Number
; -- String
; -- Symbol
```

Напомним, что Symbol (символ) выглядит как строка, начинающаяся с одиночной кавычки, но без кавычки в конце.

Идея S-выражений принадлежит Джону Маккарти (John McCarthy) и другим поклонникам языка Lisp, которые создали S-выражения в 1958 году, чтобы с их помощью обрабатывать программы на Lisp в других программах на Lisp. Это, казалось бы, циклическое рассуждение может показаться странным, но, как упоминалось в интермечцо 2, S-выражения – это универсальная форма данных, которую часто заново открывают в последнее время в веб-приложениях. Проще говоря, более близкое знакомство с S-выражениями подготовит нас к обсуждению особенностей проектирования функций для обработки определений данных, тесно связанных друг с другом.

Упражнение 316. Определить функцию atom?. ■

До настоящего момента в этой книге не встречались столь же сложные определения данных, как определение S-выражений. И все же, следуя рецепту проектирования, вы сможете проектировать функции, обрабатывающие S-выражения. Рассмотрим конкретный пример:

Задача. Спроектируйте функцию count, которая определяет количество вхождений некоторого символа в S-выражении.

На первом шаге нужно создать определения данных. Они у нас уже есть, и кажется, что этот шаг выполнен, но не забывайте, что на этом шаге необходимо также создать примеры данных, что особенно важно для сложных определений.

Предполагается, что определение данных является рецептом для создания данных, а «тесты» доказывают возможность их использования. Из определения данных S-expr следует, что каждый Atom является элементом S-expr и мы знаем, что экземпляр Atom легко создать:

```
'hello
20.12
"world"
```

Точно так же каждый экземпляр SL представляет список, а также S-выражение:

```
()()
(cons 'hello (cons 20.12 (cons "world" '())))
(cons (cons 'hello (cons 20.12 (cons "world" '())))
      '())
```

Первые два значения не требуют объяснений, но третье заслуживает более пристального внимания. Здесь повторяется второе S-выражение, но оно вложено внутрь (cons ... '()). То есть третье значение – это список, содержащий единственный элемент, а именно второй пример. Его можно упростить, используя функцию list:

```
| (list (cons 'hello (cons 20.12 (cons "world" '()))))
```

или

```
| (list (list 'hello 20.12 "world"))
```

А если прибегнуть к механизму цитирования, представленному в интермеццо 2, то запись S-выражения будет выглядеть еще проще. Вот как будут выглядеть эти три примера:

```
> '()
'()
> '(hello 20.12 "world")
(list 'hello #i20.12 "world")
> '((hello 20.12 "world"))
(list (list 'hello #i20.12 "world"))
```

Мы опробовали эти примеры в области взаимодействий DrRacket, чтобы вы могли видеть результат, который ближе к приведенным выше конструкциям, чем форма записи quote.

С помощью quote довольно легко составлять сложные примеры:

```
> '(define (f x)
           (+ x 55))
(list 'define (list 'f 'x) (list '+ 'x 55))
```

Этот пример может показаться странным, потому что он выглядит как определение функции на языке BSL, но если выполнить этот код в области взаимодействий DrRacket, то можно увидеть, что это всего лишь фрагмент данных.

Вот еще один пример:

```
> '((6 f)
  (5 e)
  (4 d))
(list (list 6 'f) (list 5 'e) (list 4 'd))
```

Этот фрагмент данных выглядит как таблица, в которой буквы связаны с числами. А вот еще один пример, который выглядит как произведение искусства:

```
> '(wing (wing body wing) wing)
(list 'wing (list 'wing 'body 'wing) 'wing)
```

А теперь запишем заголовок для функции count:

```
; S-expr Symbol -> N
; подсчитывает все вхождения sy в sexp
(define (count sexp sy)
  0)
```

Заголовок достаточно очевиден, поэтому перейдем сразу к примерам применения функции. Если задано S-выражение 'world и символ для подсчета 'world, то очевидно, что функция должна вернуть 1. Вот еще несколько примеров, сразу же сформулированных как тесты:

```
(check-expect (count 'world 'hello) 0)
(check-expect (count '(world hello) 'hello) 1)
(check-expect (count '(((world) hello) hello) 'hello) 2)
```

Как видите, цитирование очень удобно использовать в тестах. Однако использование этого механизма в макетах вызывает сплошное расстройство.

Прежде чем перейти к созданию макета, познакомимся со следующим обобщением рецепта проектирования:

ПОДСКАЗКА. Для взаимосвязанных определений данных создайте по одному макету для каждого из них. Создавайте их параллельно. Убедитесь, что они ссылаются друг на друга так же, как определения данных. **КОНЕЦ.**

Эта подсказка звучит сложнее, чем есть на самом деле. Для нашей задачи это означает, что мы должны создать три макета:

- 1) один для функции count, которая подсчитывает количество вхождений символа в S-выражение;
- 2) один для функции, которая подсчитывает количество вхождений символа в SL;
- 3) один для функции, которая подсчитывает количество вхождений символа в экземпляра Atom.

А вот три незаконченных макета с условными выражениями, соответствующими трем определениям данных:

```
(define (count sexp sy)
  (cond
    [(atom? sexp) ...]
    [else ...]))

(define (count-sl sl sy)
  (cond
    [(empty? sl) ...]
    [else ...]))

(define (count-atom at sy)
  (cond
    [(number? at) ...]
    [(string? at) ...]
    [(symbol? at) ...]))
```

Макет для `count` содержит условное выражение с двумя предложениями, потому что определение данных S-expr имеет два предложения. В нем используется функция `atom?`, чтобы разделить обработку экземпляров Atom и SL. Макет с именем `count-sl` принимает экземпляр SL и символ, а поскольку SL фактически является списком, то `count-sl` тоже содержит условное выражение с двумя предложениями. Наконец, предполагается, что `count-atom` может применяться как к экземплярам Atom, так и к экземплярам Symbol. А это означает, что она должна различать три разные формы данных, упомянутых в определении данных Atom.

Следующий шаг – разбиение составных данных на соответствующие предложения:

```
(define (count sexp sy)
  (cond
    [(atom? sexp) ...]
    [else ...]))

(define (count-sl sl sy)
  (cond
    [(empty? sl) ...]
    [else
      (... (first sl) ...
            ... (rest sl))]))

(define (count-atom at sy)
  (cond
    [(number? at) ...]
    [(string? at) ...]
    [(symbol? at) ...]))
```

Почему мы добавили в `count-sl` только два селектора?

Последний шаг в процессе создания макета требует проверки ссылок в определении данных на самого себя. В нашем случае это означает ссылки из определения данных на самого себя, ссылки из других определений данных и (возможно) обратные ссылки на другие определения. Давайте проверим предложения в выражениях `cond` в трех макетах.

1. Применение `atom?` в `count` соответствует первому предложению в определении S-expr. Чтобы учесть перекрестную ссылку из этого определения на определение Atom, добавляем `(count-atom sexp sy)`. Это означает, что мы интерпретируем `sexp` как экземпляр Atom и обрабатываем его с помощью соответствующей функции.
2. Аналогично второе предложение в выражении `cond` в функции `count` требует добавить выражение `(count-sl sexp sy)`.
3. Применение `empty?` в `count-sl` соответствует предложению в определении данных, которое не ссылается ни на какое другое определение данных.
4. Предложение `else`, напротив, содержит два выражения с селекторами, каждый из которых извлекает значение определенного типа. В частности, `(first sl)` извлекает экземпляр S-expr, поэтому мы должны заключить его в `(count ...)`. В конце концов, `count` отвечает за подсчет символов внутри произвольных S-выражений. Предложение `(rest sl)` соответствует ссылке определения данных на самого себя, и мы уже знаем, что такие ссылки обрабатываются с помощью рекурсивных вызовов функций.
5. Наконец, все три предложения в Atom относятся к элементарным формам данных. Следовательно, функцию `count-atom` изменять не нужно.

Листинг 76. Макет для обработки S-выражений

```
(define (count sexp sy)
  (cond
    [(atom? sexp)
     (count-atom sexp sy)]
    [else
     (count-sl sexp sy)]))

(define (count-sl sl sy)
  (cond
    [(empty? sl) ...]
    [else
     (... (count (first sl) sy)
          ...
          (count-sl (rest sl) sy)
          ...)]))

(define (count-atom at sy)
  (cond
    [(number? at) ...]
    [(string? at) ...]
    [(symbol? at) ...]))
```

В листинге 76 представлены три полных макета. Заполнить пробелы в этих макетах, как показано в листинге 77, совсем несложно. Но вы обязаны уметь объяснить любую строку в трех определениях. Например:

```
| [(atom? sexp) (count-atom sexp sy)]
```

определяет, является ли аргумент `sexp` экземпляром `Atom`, и если да, то интерпретирует `S-expr` как `Atom` и обрабатывает его с помощью `count-atom`. Выражение

```
| [else
  (+ (count (first sl) sy) (count-sl (rest sl) sy))]
```

означает, что данный список состоит из двух элементов: `S-expr` и `SL`. Для подсчета вхождений заданного символа `sy` в обоих элементах используются функции `count` и `count-sl` соответственно, и их результаты складываются, что дает общее количество `sy` во всем экземпляре `sexp`.

```
| [(symbol? at) (if (symbol=? at sy) 1 0)]
```

говорит нам, что если экземпляр `Atom` является экземпляром `Symbol`, то аргумент `sy` встречается в нем ровно один раз, если он равен `sexp`, иначе он не встречается вообще. Поскольку оба аргумента представляют элементарные данные, ничего другого просто невозможно.

Листинг 77. Программа для обработки *S*-выражений

```
; S-expr Symbol -> N
; подсчитывает все вхождения sy в sexp
(define (count sexp sy)
  (cond
    [(atom? sexp) (count-atom sexp sy)]
    [else (count-sl sexp sy)]))

; SL Symbol -> N
; подсчитывает все вхождения sy в sl
(define (count-sl sl sy)
  (cond
    [(empty? sl) 0]
    [else
      (+ (count (first sl) sy) (count-sl (rest sl) sy))]))

; Atom Symbol -> N
; подсчитывает все вхождения sy в at
(define (count-atom at sy)
  (cond
    [(number? at) 0]
    [(string? at) 0]
    [(symbol? at) (if (symbol=? at sy) 1 0)]))
```

Упражнение 317. Программа, состоящая из трех взаимосвязанных функций, должна выражать эту взаимосвязь с помощью выражения `local`.

Скопируйте и реорганизуйте программу из листинга 77 в единую функцию, используя `local`. Проверьте исправленный код с помощью набора тестов для `count`.

Второй аргумент локальных функций – `sy` – никогда не изменяется. Он всегда совпадает с исходным символом. Следовательно, его мож-

но исключить из определений локальных функций, чтобы сообщить читателю, что `sy` является константой. ■

Упражнение 318. Спроектируйте функцию `depth`. Она должна принимать S-выражение и определять его глубину. Экземпляр `Atom` имеет глубину 1. Глубина списка S-выражений равна максимальному количеству его элементов плюс 1. ■

Упражнение 319. Спроектируйте функцию `substitute`. Она должна принимать S-выражение `s` и два символа, старый `old` и новый `new`. В результате должно получиться то же S-выражение `s`, в котором все вхождения символа `old` заменены символом `new`. ■

Упражнение 320. Переформулируйте определение данных для `S-expr`, заменив первое предложение тремя предложениями из определения `Atom`, а второе предложение – абстракцией `List-of`. Измените функцию `count` в соответствии с этим определением данных.

Затем интегрируйте определение `SL` в определение `S-expr`. Еще раз упростите `count`. Попробуйте использовать лямбда-выражение.

Примечание. Такое упрощение не всегда возможно, но опытные программисты склонны допускать подобные возможности. ■

Упражнение 321. Абстрагируйте определения данных `S-expr` и `SL`, абстрагировав типы атомов `Atom`, которые могут появиться. ■

19.4. Проектирование с использованием взаимосвязанных данных

Перейти от определений данных, ссылающихся на самих себя, к коллекциям определений данных с перекрестными ссылками намного проще, чем перейти от определений конечных данных к определениям данных, ссылающимся на самих себя. Для этого требуется лишь немного скорректировать рецепт проектирования определений данных со ссылками на самих себя – см. главу 9 – и применить его к этой, казалось бы, сложной ситуации.

1. Потребность в таких взаимосвязанных определениях данных подобна потребности в определениях данных со ссылками на самих себя. Постановка задачи включает множество разных видов информации, и одна форма информации ссылается на другие.

Прежде чем начать действовать в таких ситуациях, нарисуйте стрелки, соответствующие ссылкам в определениях. Рассмотрим пример слева на рис. 16. Он представляет определение `S-expr`, содержащее ссылки на `SL` и `Atom`, изображенные стрелками. Точно так же определение `SL` содержит одну ссылку на `S-expr` и одну ссылку на само определение `SL`, которые также изображены стрелками.

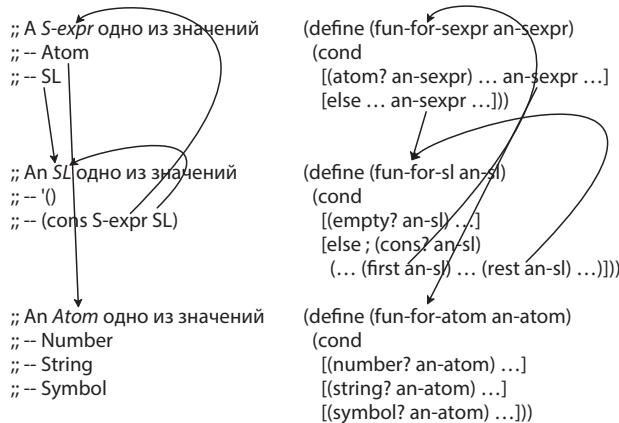


Рис. 16. Стрелки, изображающие ссылки в определениях данных и макетах

Подобно определениям данных, ссылающимся на самих себя, этот набор определений тоже требует проверки. Как минимум вы должны создать несколько примеров для каждого отдельного определения. Начинайте с предложений, которые не ссылаются ни на какие другие определения данных. Имейте в виду, что определение может быть недействительным, если для него невозможно генерировать примеры.

2. Ключевое изменение заключается в необходимости спроектировать несколько функций, количество которых совпадает с количеством определений данных. Каждая функция специализируется на одном из определений данных; все остальные аргументы остаются прежними. Исходя из этого, начинать следует с сигнатуры, описания назначения и фиктивного определения **для каждой функции**.
 3. Обязательно добавьте примеры применения функций, использующие все перекрестные ссылки в группе определений данных.
 4. Для каждой функции создайте макет, соответствующий определению данных. Используйте табл. 10, чтобы довести создание макета до конца. На последнем шаге в обновленном рецепте требуется проверить все ссылки в определениях данных на самих себя и перекрестные ссылки. Используйте определения данных с нарисованными стрелками, чтобы реализовать этот шаг. Для каждой стрелки в определениях данных добавьте стрелку в макеты. См. пример справа на рис. 16, где представлены версии макетов со стрелками.
- Затем замените стрелки фактическими вызовами функций. Набравшись опыта, вы будете пропускать этап рисования стрелок и сразу подставлять вызовы функций.

Примечание. Обратите внимание, что определения данных и макеты функций содержат по четыре стрелки и что пары стрелок соответствуют друг другу. Исследователи называют это соответствие *симметрией*. Это свидетельство того, что рецепт дизайна обеспечивает естественный путь от постановки задачи к ее решению.

5. Приступая к проектированию тела функции, начинайте с предложений в `cond`, которые не содержат рекурсивных вызовов или вызовов других функций. Эти предложения называются *базовыми случаями*. Соответствующие результаты обычно легко сформулировать, или даже они могут быть даны в примерах. После этого переходите к предложениям со ссылками на сами определения и с перекрестными ссылками. Руководствуйтесь при этом вопросами и ответами в табл. 11.
6. Закончив определение функций, выполните тесты. Если вскроется проблема во вспомогательной функции, то вы получите два сообщения об ошибке, одно для основной функции, а другое – для вспомогательной. Одно исправление должно устранить обе ошибки. Убедитесь, что тесты охватывают все части функции.

Наконец, если вы застопорились на шаге 5, вспомните табличный подход к угадыванию комбинации функций. В случае со взаимосвязанными данными может потребоваться составить таблицу не только для каждого случая, но и для каждого случая и для каждой функции.

19.5. Проект: BST

Программисты часто используют древовидные представления данных для увеличения производительности своих функций. Наиболее известной формой дерева является **бинарное дерево поиска**, потому что оно позволяет быстро сохранять и извлекать информацию.

Давайте рассмотрим конкретное применение бинарных деревьев для управления информацией о людях. Вместо структур `child`, как в примере с генеалогическими древами, бинарное дерево содержит узлы `node`:

```
(define-struct no-info [])
(define NONE (make-no-info))

(define-struct node [ssn name left right])
;BT (сокращенно от BinaryTree -- двоичное дерево) -- это одно из значений:
; -- NONE
; -- (make-node Number Symbol BT BT)
```

Определение бинарного дерева похоже на определение генеалогических древ. Значение `NONE` указывает здесь на отсутствие информации, а каждый узел `node` хранит номер социального страхования, имя

и два других бинарных дерева. Последние подобны родительским древам в определении генеалогических древ с той лишь разницей, что отношения между узлом *node* и его левым и правым деревьями не основаны на семейных отношениях.

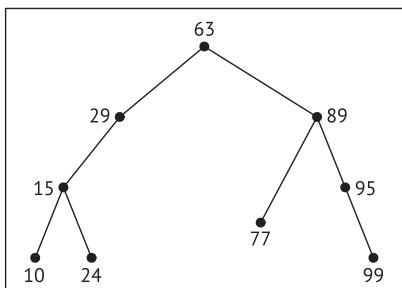
Вот два бинарных дерева:

```
(make-node
  15
  'd
  NONE
  (make-node
    24 'i NONE NONE))
```

```
(make-node
  15
  'd
  (make-node
    87 'h NONE NONE)
  NONE)
```

На рис. 17 показаны схематические изображения подобных деревьев. Деревья нарисованы вверх ногами: корень находится вверху, а крона – внизу. Каждый кружок соответствует узлу и подписан значением поля *ssn* в структуре *node*. На схеме отсутствует значение *NONE*.

Дерево А



Дерево В

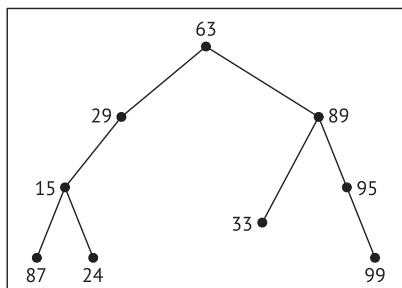


Рис. 17. Бинарное дерево поиска и бинарное дерево

Упражнение 322. Нарисуйте два вышеупомянутых дерева, как показано на рис. 17. Затем спроектируйте функцию *contains-bt?*, которая определяет наличие данного числа в данном дереве BT. ■

Упражнение 323. Спроектируйте функцию *search-bt*. Она должна принимать число *n* и дерево *BT*. Если дерево содержит структуру *node*, поле *ssn* в которой равно *n*, то функция должна вернуть значение поля *name* в этом узле. Иначе функция должна вернуть *#false*.

Подсказка. Попробуйте использовать *contains-bt?*, чтобы сначала проверить наличие искомого узла в дереве и *boolean?* для проверки результата естественной рекурсии на каждом этапе. ■

Если прочитать числа в двух деревьях на рис. 17 слева направо, то мы получим две разные последовательности:

Дерево А	10	15	24	29	63	77	89	95	99
Дерево В	87	15	24	29	63	33	89	95	99

Последовательность для дерева А отсортирована в порядке возрастания, а для В – нет. Бинарное дерево первого типа – это **бинарное**

дерево поиска. Каждое бинарное дерево поиска является бинарным деревом, но не каждое бинарное дерево является бинарным деревом поиска. Давайте сформулируем условие – инвариант данных, – отличающее бинарное дерево поиска от простого бинарного дерева:

BST-инвариант

BST (сокращенно от *binary search tree* – бинарное дерево поиска) – это ВТ, согласно следующим условиям:

- NONE всегда является бинарным деревом поиска BST;
- (`make-node ssn0 name0 L R`) является бинарным деревом поиска BST, если:
 - L является бинарным деревом поиска BST;
 - R является бинарным деревом поиска BST;
 - все поля `ssn` в L меньше `ssn0`;
 - все поля `ssn` в R больше `ssn0`.

Иначе говоря, чтобы проверить, является ли бинарное дерево ВТ бинарным деревом поиска BST, нужно проверить все числа во всех поддеревьях и убедиться, что они меньше или больше некоторого заданного числа. Это накладывает дополнительное ограничение на процесс конструирования данных, но, как показывают следующие упражнения, оно того стоит.

Упражнение 324. Спроектируйте `inorder`. Она должна принимать бинарное дерево и возвращать последовательность всех номеров `ssn` в дереве, получающуюся при просмотре дерева слева направо.

Подсказка. Используйте функцию `append`, которая объединяет списки, как показано ниже:

```
| (append (list 1 2 3) (list 4) (list 5 6 7))
| ==
| (list 1 2 3 4 5 6 7)
```

Что должна вернуть `inorder` при применении к бинарному дереву поиска? ■

При поиске узла `node` с заданным `ssn` в BST можно использовать инвариант BST. Чтобы узнать, содержит ли ВТ узел с определенным значением `ssn`, функции может потребоваться просмотреть каждый узел в дереве. Напротив, чтобы выяснить, содержит ли бинарное дерево поиска узел с определенным значением `ssn`, функция может исключить одно из двух поддеревьев для каждого проверяемого узла.

Проиллюстрируем эту идею на примере следующего дерева BST:

```
| (make-node 66 'a L R)
```

Если мы ищем 66, то сразу же найдем требуемый узел. Если мы ищем меньшее число, скажем 63, то можем сфокусировать поиск на L, потому что все узлы с полями `ssn` меньше 66 находятся в L. Точно так же, выполняя поиск 99, мы могли бы игнорировать L и сфокусироваться на R, потому что все узлы с `ssn` больше 66 находятся в R.

Упражнение 325. Спроектируйте функцию `search-bst`. Она должна принимать число `n` и `BST`. Если дерево содержит узел `node` с числом `n` в поле `ssn`, то функция должна вернуть значение поля `name` этого узла. Иначе она должна вернуть `NONE`. Также функция должна использовать инвариант `BST`, чтобы выполнялось как можно меньше сравнений.

Сравните это упражнение с упражнением 189, где предлагалось спроектировать функцию поиска в отсортированном списке. ■

Построить бинарное дерево легко, но построить бинарное дерево поиска намного сложнее. Имея два любых `BT`, число и имя, достаточно просто применить `make-node` к этим значениям в правильном порядке, и мы получим новое дерево `BT`. Однако эта процедура не годится для `BST`, потому что не во всех случаях результатом будет `BST`. Например, если одно дерево `BST` содержит узлы со значениями 3 и 5 в полях `ssn`, а другое дерево содержит узлы со значениями 2 и 6, то простое объединение двух деревьев с еще одним номером социального страхования и именем не даст `BST`.

Следующие два упражнения объясняют, как построить `BST` из списка номеров и имен. В частности, первое упражнение предлагает спроектировать функцию, которая вставляет заданные `ssn0` и `name0` в `BST`; то есть создает дерево `BST`, подобное исходному, но с добавлением еще одного узла, со значениями `ssn0` и `name0`. А во втором упражнении предлагается спроектировать функцию, обрабатывающую полный список номеров и имен.

Упражнение 326. Спроектируйте функцию `create-bst`. Она должна принимать `BST B`, число `N` и символ `S` и возвращать `BST`, подобное исходному дереву `B`, в котором вместо одного поддерева `NONE` содержится структура узла `node`.

```
| (make-node N S NONE NONE)
```

Завершив проектирование, попробуйте применить функцию к дереву `A` на рис. 17. ■

Упражнение 327. Спроектируйте функцию `create-bst-from-list`. Она должна принимать список номеров и имен и создавать дерево `BST`, многократно применяя `create-bst`. Вот ее сигнатура:

```
| ; [List-of [List Number Symbol]] -> BST
```

Завершив проектирование, попробуйте применить эту функцию к следующему списку:

```
'((99 o)
 (77 l)
 (24 i)
 (10 h)
 (95 g)
 (15 d)
 (89 c)
 (29 b)
 (63 a))
```

В результате должно получиться дерево А на рис. 17. Используя существующие абстракции, можно получить «инвертированное» дерево. Объясните почему? ■

19.6. Упрощение функций

Упражнение 317 демонстрирует, как использовать выражение `local` для организации функции, обрабатывающей взаимосвязанные формы данных. Такая организация помогает также упрощать функции, если известно, что определение данных является окончательным. Для демонстрации мы покажем, как упростить решение упражнения 319.

В листинге 78 показано полное определение функции `substitute`. В нем используется выражение `local` с определениями трех вспомогательных функций, соответствующих определениям данных. В листинге также представлен тестовый пример, чтобы вы могли тестиировать функцию после каждого изменения, предложенного ниже. Стоп! Разработайте дополнительные тесты; одного теста почти всегда недостаточно.

Упражнение 328. Скопируйте код из листинга 120 в область определений DrRacket; добавьте свои тесты. Убедитесь, что тесты успешно выполняются. Читая далее этот раздел, вносите изменения в код и повторно запускайте набор тестов, чтобы подтвердить обоснованность наших изменений. ■

Мы знаем, что SL описывает списки S-expr, поэтому можем использовать `map`, чтобы упростить `for-sl`, как показано в листинге 79. В исходной программе используется `for-sexp`, которая применяется к каждому элементу в `sl`, а в исправленном определении используется `map`, которая позволяет выразить ту же идею более кратко.

Листинг 78. Программа для упрощения

```
; S-expr Symbol Atom -> S-expr
; замещает все вхождения old в sexp значениями new

(check-expect (substitute '(((world) bye) bye) 'bye '42)
               '(((world) 42) 42))

(define (substitute sexp old new)
  (local (; S-expr -> S-expr
          (define (for-sexp sexp)
            (cond
              [(atom? sexp) (for-atom sexp)]
              [else (for-sl sexp)])))
    ; SL -> S-expr
    (define (for-sl sl)
      (cond
        [(empty? sl) '()]
        [else (cons (for-sexp (first sl))
                    (for-sl (rest sl))))]))
  ; Atom -> S-expr
```

```
(define (for-atom at)
  (cond
    [(number? at) at]
    [(string? at) at]
    [(symbol? at) (if (equal? at old) new at))])
  (for-sexp sexp)))
```

Второе упрощение основано на применении функции `equal?`, которая сравнивает два произвольных значения. В результате этого третья локальная функция становится односторонней. В листинге 80 показан результат этого второго упрощения.

Несмотря на то что `sexp` является параметром, такая замена действительно возможна, потому что она также заменяет фактическое значение.

Теперь мы имеем два локальных определения, состоящих из одной строки кода каждое. Более того, ни одно из определений не является рекурсивным. Следовательно, эти функции можно встроить в `for-sexp`. Под встраиванием подразумевается замена `(for-atom sexp)` на `(if (equal? sexp old) new sexp)`, то есть мы заменяем параметр `at` фактическим аргументом `sexp`. То же относится к выражению `(for-sl sexp)`, которое можно заменить на `(map for-sexp sexp)`; см. нижнюю половину листинга 79. Теперь мы имеем функцию, определение которой содержит одну локальную функцию, принимающую тот же основной аргумент. Если бы мы систематически передавали два других аргумента, мы бы сразу увидели, что локальная функция может использоваться вместо внешней.

Листинг 79. Упрощение программы, шаг 1

```
(define (substitute sexp old new)
  (local (; S-expr -> S-expr
          (define (for-sexp sexp)
            (cond
              [(atom? sexp) (for-atom sexp)]
              [else (for-sl sexp)])))
        ; SL -> S-expr
        (define (for-sl sl)
          (map for-sexp sl))
        ; Atom -> S-expr
        (define (for-atom at)
          (cond
            [(number? at) at]
            [(string? at) at]
            [(symbol? at) (if (equal? at old) new at))])
          (for-sexp sexp)))
```

Вот результат воплощения этой последней задумки:

```
(define (substitute sexp old new)
  (cond
    [(atom? sexp) (if (equal? sexp old) new sexp)]
    [else
      (map (lambda (s) (substitute s old new)) sexp)]))
```

Стоп! Объясните, почему пришлось использовать лямбда-выражение для этого последнего упрощения.

Листинг 80. Упрощение программы, шаги 2 и 3

```
(define (substitute SEXP OLD NEW)
  (local (; S-expr -> S-expr
            (define (FOR-SEXP SEXP)
              (cond
                [(atom? SEXP) (FOR-ATOM SEXP)]
                [else (FOR-SL SEXP)])))
    ; SL -> S-expr
    (define (FOR-SL SL) (map FOR-SEXP SL))
    ; Atom -> S-expr
    (define (FOR-ATOM AT)
      (if (equal? AT OLD) NEW AT)))
  (FOR-SEXP SEXP)))

(define (substitute.v3 SEXP OLD NEW)
  (local (; S-expr -> S-expr
            (define (FOR-SEXP SEXP)
              (cond
                [(atom? SEXP)
                 (if (equal? SEXP OLD) NEW SEXP)]
                [else
                 (map FOR-SEXP SEXP)])))
  (FOR-SEXP SEXP)))
```

20. Итеративное уточнение

При разработке реальных программ можно столкнуться со сложными формами информации и проблемой их представления в виде данных. Лучшая стратегия для решения этой проблемы – использовать *итеративное уточнение*, хорошо известный научный процесс. Задача ученого состоит в том, чтобы представить часть реального мира, используя какую-либо форму математики. Результат этих усилий называется моделью. Затем ученый проводит всестороннее тестирование модели, предсказывая результаты экспериментов. Если расхождения между прогнозами и измерениями слишком велики, модель уточняется с целью улучшения прогнозов. Этот итеративный процесс продолжается до тех пор, пока прогнозы не станут достаточно точными.

Рассмотрим ученого-физика, который желает предсказать траекторию полета ракеты. Представление «ракеты в виде точки» достаточно простое, но неточное. Такое представление, например, не учитывает трение о воздухе. Стремясь уточнить модель, физик может добавить грубый контур ракеты и ввести необходимые формулы для расчета силы трения. Эта вторая модель является *уточнением* первой модели. Этот процесс уточнения повторяется до тех пор, пока модель не научится предсказывать траекторию полета ракеты с достаточной точностью.

Программист, получивший образование на факультете информатики, должен поступать так же, как этот физик. Суть в том, чтобы найти точное представление реальной информации в виде данных и функций, обрабатывающих эти данные должным образом. Сложные ситуации требуют многократного повторения процесса уточнения, чтобы получить в результате достаточно точное представление данных в сочетании с надлежащими функциями. Процесс начинается с представления основных элементов информации и при необходимости вовлекает другие элементы. Иногда программисту приходится уточнять модель **после** развертывания программы, если пользователи запрашивают дополнительные возможности.

До сих пор мы использовали итеративное уточнение, когда дело касалось сложных форм данных. Эта глава иллюстрирует итеративное уточнение как принцип разработки программ на расширенном примере, представляющем и обрабатывающем (элементы) файловой системы компьютера. Сначала мы кратко обсудим файловую систему, а затем последовательно разработаем три представления данных. Попутно будет представлено несколько упражнений по программированию, чтобы вы могли увидеть, как рецепт проектирования помогает изменять существующие программы.

20.1. Анализ данных

Прежде чем закрыть DrRacket, убедитесь, что сохранили свой код. В противном случае вам придется повторно вводить его весь при сле-

дующем запуске DrRacket. Итак, вы просите свой компьютер сохранять программы и данные в *файлы*. Файлы очень похожи на длинные строки.

В большинстве компьютерных систем файлы организованы в *каталоги*, или *папки*. Каталог содержит несколько файлов и, возможно, несколько вложенных каталогов. Последние называются *подкаталогами* и могут содержать другие подкаталоги и файлы. Эту иерархическую организацию файлов и каталогов мы называем *деревом каталогов*.

На рис. 18 изображено небольшое дерево каталогов. Этот рисунок наглядно объясняет, почему мы называем их деревьями. Вопреки общепринятым правилам, на рисунке показано дерево, растущее вверх, с корневым каталогом с именем `TS`. Корневой каталог содержит один файл с именем `read!` и два подкаталога с именами `Text` и `Libs` соответственно. Подкаталог `Text` содержит три файла; подкаталог `Libs` содержит два подкаталога, в каждом из которых имеется не менее одного файла. Наконец, все поля на рисунке сопровождаются аннотациями: каталоги отмечены аннотацией `DIR` (от англ. *directory* – каталог), а файлы отмечены аннотациями с их размерами.

На самом деле файл представляет собой последовательность байтов, идущих один за другим. Попробуйте определить класс файлов.

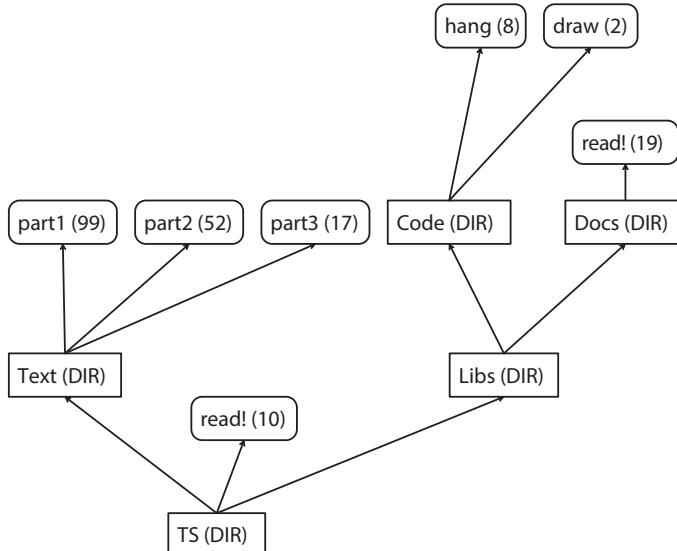


Рис. 18. Пример дерева каталогов

Упражнение 329. Сколько файлов с именем `read!` присутствует в дереве каталогов `TS`? Попробуйте описать путь к каждому из этих файлов, начиная от корневого каталога. Каков общий размер всех файлов в дереве? Каков общий размер каталога, если размер каждого узла каталога равен 1? Сколько уровней каталогов в этом дереве? ■

20.2. Уточнение определений данных

В упражнении 329 перечислены некоторые вопросы о каталогах, которые обычно задают пользователи. Для получения ответов на такие вопросы в операционной системе имеются специальные программы. Если вы решите заняться проектированием подобных программ, то вам придется разработать представление данных для деревьев каталогов.

В этом разделе мы используем прием итеративного уточнения и разработаем три таких представления данных. На каждом этапе нам нужно будет решить, какие атрибуты включать, а какие игнорировать. Рассмотрим дерево каталогов, изображенное на рис. 18, и представим, как оно создавалось. Сразу после создания каталог ничего не содержит – он пустой. Но потом пользователь может добавить в него файлы и другие каталоги. Обычно пользователь обращается к файлам по именам, а каталоги представляют как контейнеры.

МОДЕЛЬ 1. Согласно мысленному эксперименту, наша первая модель должна рассматривать файлы как атомарные объекты с именами, а каталоги – как контейнеры. Вот определение данных, представляющее каталоги в виде списков, а файлы в виде строк с их именами:

```
| ; Dir.v1 (сокращенно от directory -- каталог) -- это одно из значений:  
| ; -- '()'  
| ; -- (cons File.v1 Dir.v1)  
| ; -- (cons Dir.v1 Dir.v1)  
  
; File.v1 -- это строка.
```

В названиях присутствует окончание `.v1`, которое поможет нам отличать эту версию от будущих уточнений.

Упражнение 330. Преобразуйте дерево каталогов на рис. 18 в представление данных, описываемое моделью 1. ■

Упражнение 331. Спроектируйте функцию `how-mapу`, определяющую количество файлов в данном списке `Dir.v1`. Не забывайте следовать рецепту проектирования; примеры данных вы найдете в упражнении 330. ■

МОДЕЛЬ 2. Решившие упражнение 331 наверняка согласятся с тем, что это первое определение данных достаточно простое, и вместе с тем оно скрывает природу каталогов. Это первое представление не позволяет перечислить все подкаталоги, находящиеся внутри некоторого заданного каталога. Для более точного моделирования каталогов необходимо ввести тип структуры, объединяющий имя с контейнером:

```
| (define-struct dir [name content])
```

Этот новый тип структуры, в свою очередь, предполагает изменения в определении данных:

```

; Dir.v2 -- это структура:
;   (make-dir String LOFD)

; LOFD (list of files and directories -- список файлов и каталогов) -- это
;   одно из значений:
;   -- '()
;   -- (cons File.v2 LOFD)
;   -- (cons Dir.v2 LOFD)

; File.v2 -- это строка.

```

Обратите внимание, что определение данных Dir.v2 ссылается на определение LOFD, а определение LOFD – на определение Dir.v2. Эти два определения взаимно рекурсивны.

Упражнение 332. Преобразуйте дерево каталогов на рис. 18 в представление данных, описываемое моделью 2. ■

Упражнение 333. Спроектируйте функцию how-many, определяющую количество файлов в Dir.v2. Примеры данных вы найдете в упражнении 332. Сравните свой результат с результатом упражнения 331. ■

Упражнение 334. Покажите, как добавить в каталог еще два атрибута: размер и доступность для чтения. Первый сообщает, сколько места занимает сам каталог (без его содержимого), а второй – может ли кто-либо еще, кроме пользователя, просматривать содержимое каталога. ■

МОДЕЛЬ 3. Файлы, как и каталоги, тоже имеют атрибуты. Чтобы добавить их, будем действовать так же, как выше. Сначала определим структуру для представления файлов:

```
| (define-struct file [name size content])
```

А затем запишем определение данных:

```

; File.v3 -- это структура:
;   (make-file String N String)

```

Согласно именам полей, первая строка в структуре хранит имя файла, натуральное число – его размер, а вторая строка – содержимое.

Теперь разделим поле, представляющее содержимое каталога, на две части: список файлов и список подкаталогов. Для этого изменим определение структуры:

```
| (define-struct dir.v3 [name dirs files])
```

И определение данных:

```

; Dir.v3 -- это структура:
;   (make-dir.v3 String Dir* File*)

; Dir* -- это одно из значений:
;   -- '()
;   -- (cons Dir.v3 Dir*)

; File* -- это одно из значений:

```

```
| ; -- '()
| ; -- (cons File.v3 File*)
```

Согласно соглашениям, принятым в информатике, звездочка (*) в конце имени означает «многие» и является признаком, отличающим имя с этим окончанием от других похожих имен, таких как `File.v3` и `Dir.v3`.

Упражнение 335. Преобразуйте дерево каталогов на рис. 18 в представление данных, описываемое моделью 3. Используйте "" для представления содержимого файлов. ■

Упражнение 336. Спроектируйте функцию `how-mapу`, определяющую количество файлов в `Dir.v3`. Примеры данных вы найдете в упражнении 335. Сравните свой результат с результатом упражнения 333.

Учитывая сложность определения данных, подумайте, как спроектировать корректные функции для каждого из них. Как убедиться, что `how-mapу` возвращает правильные результаты? ■

Упражнение 337. Упростите определение данных `Dir.v3` с помощью `List-of`. Затем используйте функции обработки списков из `ISL+` из листингов 56 и 57, чтобы упростить определения функций, спроектированных в упражнении 336. ■

Начав с простой первой модели, мы шаг за шагом уточняли ее и получили достаточно точное представление данных для деревьев каталогов. Это третье представление данных намного точнее отражает природу дерева каталогов, чем первые два. На основе этой модели мы можем создать несколько других функций, которые пригодятся пользователям операционной системы.

20.3. Уточнение функций

Добавьте (`require htdp/dir`) в область определений. Чтобы сделать следующие упражнения более реалистичными, используем библиотеку `dir.rkt` из первого издания этой книги. В этом обучающем пакете определены два типа структур из модели 3, но без окончания `.v3`. Кроме того, в обучающем пакете имеется функция, создающая представления деревьев каталогов на вашем компьютере:

```
| ; String -> Dir.v3
| ; создает представление каталога a-path
(define (create-dir a-path) ...)
```

Например, если открыть DrRacket и ввести в область определений следующие три строки:

```
| (define O (create-dir "/Users/...")) ; в OS X
| (define L (create-dir "/var/log/")) ; в Linux
| (define W (create-dir "C:\\\\Users\\\\...")) ; в Windows
```

то вы получите представления данных каталогов на вашем компьютере после **сохранения** и последующего запуска программы. На самом деле с помощью функции `create-dir` можно получить представление всей файловой системы на вашем компьютере в виде экземпляра `Dir.v3`.

Внимание. (1) Для конструирования представлений больших деревьев каталогов может потребоваться много времени. Сначала попробуйте применить `create-dir` к небольшим деревьям каталогов. (2) **Не** определяйте свой тип структуры `dir`. Обучающий пакет уже содержит ее, и вы не должны определять этот тип структуры повторно.

Функция `create-dir` создает упрощенное, но достаточно реалистичное представление дерева каталогов, чтобы вы могли понять, как проектировать подобные программы. Следующие упражнения иллюстрируют этот момент. Они используют `Dir` для обозначения общей идеи представления данных для деревьев каталогов. Используйте простейшее определение данных `Dir`, которое позволит вам выполнить соответствующее упражнение. При желании вы можете свободно использовать определение данных из упражнения 337 и функции из листингов 56 и 57.

Упражнение 338. Используйте `create-dir`, чтобы преобразовать некоторые из ваших каталогов в представления данных `ISL+`. Затем используйте функцию `how-many` из упражнения 336, чтобы подсчитать, сколько файлов они содержат. Как убедиться, что `how-many` возвращает правильные результаты? ■

Упражнение 339. Спроектируйте функцию `find?`. Она должна принимать `Dir` и имя файла и определять, присутствует ли файл с таким именем в дереве каталогов. ■

Упражнение 340. Спроектируйте функцию `ls`, которая возвращает список с именами всех файлов и подкаталогов в заданном каталоге `Dir`. ■

Упражнение 341. Спроектируйте функцию `du`, которая принимает `Dir` и вычисляет общий размер всех файлов во всем дереве каталогов. При обработке каталогов исходите из предположения, что каждый из них имеет размер, равный 1. В реальном мире каталог – это специальный файл, и его размер зависит от размера связанного с ним каталога. ■

Остальные упражнения основаны на понятии пути, который в нашем случае имеет вид списка имен:

```
| ; Path -- это [List-of String].  
| ; интерпретация: путь в дереве каталогов
```

Взгляните еще раз на рис. 18. На этом рисунке путь от `TS` к `part1` выглядит так: (`list "TS" "Text" "part1"`). Аналогично выглядит путь от `TS` к `Code`: (`list "TS" "Libs" "Code"`).

Упражнение 342. Спроектируйте функцию `find`. Она должна принимать каталог `d` и имя файла `f`. Если (`find? D f`) вернет `#true`, то `find`

должна создать путь к файлу с именем f ; в противном случае она должна вернуть #false.

Подсказка. Кажется заманчивым сначала проверить присутствие файла в дереве каталогов, но это придется сделать для каждого отдельного подкаталога. Поэтому лучше совместить функциональность find? и find .

Усложненное задание. Функция find обнаруживает только один из двух файлов с именем read! на рис. 18. Спроектируйте функцию find-all , обобщенную версию функции find , которая создает список всех путей, ведущих к f в d . Что должна вернуть find-all , если $(\text{find? } d \ f)$ вернет #false? Действительно ли это усложненное задание сложнее базового? ■

Упражнение 343. Спроектируйте функцию ls-R , которая возвращает список путей ко всем файлам, содержащимся в данном каталоге Dir . ■

Упражнение 344. Измените функцию find-all из упражнения 342, используя ней функцию ls-R из упражнения 343. Это упражнение на умение применять прием композиции функций, и если вы решили усложненное задание в упражнении 342, то ваша новая функция также сможет находить каталоги. ■

21. Уточнение интерпретатора

DrRacket – это программа, сложная и способная обрабатывать множество различных типов данных. Как и большинство сложных программ, DrRacket состоит из множества функций: одни позволяют программистам редактировать текст, другие обслуживают область взаимодействий, третьи проверяют, являются ли определения и выражения грамматически правильными, и т. д.

В этой главе мы покажем, как спроектировать функцию, которая реализует основные возможности области взаимодействий. Естественно, что в данном проекте мы используем прием последовательного уточнения. По сути, сама идея сосредоточить внимание на этом аспекте DrRacket – это еще один пример уточнения, а именно на реализации небольшой части обширной функциональности.

Фактически область взаимодействий решает задачу вычисления вводимых вами выражений. Когда вы щелкаете на кнопке **RUN** (Выполнить), область взаимодействий получает все определения, введенные в области определений, и подготавливается к приему ваших выражений, которые могут ссылаться на эти определения. После ввода выражения она вычисляет его значение, выводит на экран и переходит к ожиданию ввода следующего выражения. Этот цикл может повторяться снова и снова. По этой причине многие называют область взаимодействия циклом *чтения-вычисления-вывода*, а функцию, которая реализует этот цикл, – *интерпретатором*.

Начнем наш процесс уточнения с числовых выражений BSL. Они простые, не предполагают ссылок на сложные определения, и даже пятиклассник сможет определить их значение. Как только вы пройдете этот первый шаг, вы поймете разницу между выражениями BSL и их представлениями. Затем мы перейдем к выражениям с переменными. На последнем шаге мы добавим определения.

21.1. Интерпретация выражений

Наша первая задача – согласовать представление данных для программ BSL. То есть мы должны выяснить, как представить выражение BSL в форме данных BSL. Поначалу это может звучать странно и не-привычно, но здесь нет никаких сложностей. Предположим для начала, что нам нужно представить только числа и операции сложения и умножения. Очевидно, что числа могут представлять числа. Однако для представления выражения со сложением требуются составные данные, потому что оно содержит два подвыражения и потому что отличается от выражения умножения, для которого также необходимо создать представление данных.

Как описывалось в главе 5, самый простой способ представления сложения и умножения – определить два типа структур с двумя полями:

```
| (define-struct add [left right])
| (define-struct mul [left right])
```

Здесь предполагается, что поле `left` содержит один операнд – тот, что «слева» от оператора, а поле `right` содержит второй операнд. В таблице ниже показаны три примера:

Выражение BSL	Представление выражения BSL
3	(make-add 1 1)
(+ 1 1)	(make-mul 300001 100000)
(* 300001 100000)	(make-mul 300001 100000)

Следующий вопрос касается выражения с подвыражениями:

```
| (+ (* 3 3) (* 4 4))
```

На этот вопрос есть удивительно простой ответ: поля могут содержать любое значение. В этом конкретном случае `left` и `right` могут содержать представления выражений, и выражения могут вкладываться друг в друга настолько глубоко, насколько это потребуется. В табл. 21 приводится еще несколько дополнительных примеров.

Таблица 21. Представление выражений BSL в BSL

Выражение BSL	Представление выражения BSL
(+ (* 1 1) 10)	(make-add (make-mul 1 1) 10)
(+ (* 3 3) (* 4 4))	(make-add (make-mul 3 3) (make-mul 4 4))
(+ (* (+ 1 2) 3) (* (* (+ 1 1) 2) 4))	(make-add (make-mul (make-add 1 2) 3) (make-mul (make-mul (make-add 1 1) 2) 4))

Упражнение 345. Сформулируйте определение данных для представления выражений BSL на основе определений структур `add` и `mul`. Дайте новому классу имя *BSL-expr* по аналогии с *S-expr*.

Преобразуйте следующие выражения в данные:

Здесь под словом «интерпретируйте» подразумевается: «преобразуйте данные в информацию». Слово «интерпретатор» в названии этой главы относится к программе, которая принимает представление программы и вычисляет ее значение. Эти два понятия родственные, но обозначают не одно и то же.

1. (+ 10 -10)
2. (+ (* 20 3) 33)
3. (+ (* 3.14 (* 2 3)) (* 3.14 (* -1 -9)))

Интерпретируйте следующие данные как выражения:

1. (make-add -1 2)
2. (make-add (make-mul -2 -3) 33)
3. (make-mul (make-add 1 (make-mul 2 3)) 3.14) ■

Теперь, когда у нас есть представление данных для программ BSL, можно приступить к проектированию функции вычисления (интерпретатора). Эта функция должна принимать представление выражения BSL и возвращать его зна-

чение. Но эта функция не похожа ни на одну из тех, что мы создавали до сих пор, поэтому давайте поэкспериментируем с некоторыми примерами. Вы можете поэкспериментировать в уме, используя правила арифметики, чтобы определить значения выражений, или «поиграть» с областью взаимодействий в DrRacket. Взгляните на следующую таблицу с нашими примерами:

Выражение BSL	Его представление	Его значение
3	3	3
(+ 1 1)	(make-add 1 1)	2
(* 3 10)	(make-mul 3 10)	30
(+ (* 1 1) 10)	(make-add (make-mul 1 1) 10)	11

Упражнение 346. Сформулируйте определение данных для класса значений, которые могут получаться в результате оценки представлений выражений BSL. ■

Упражнение 347. Спроектируйте функцию eval-expression. Она должна принимать представление выражения BSL и вычислять его значение. ■

Упражнение 348. Разработайте представление данных для логических выражений BSL, конструируемых из значений #true, #false и операторов or, and и not. Затем спроектируйте функцию eval-bool-expression, принимающую представление логического выражения BSL и вычисляющую его значение. Какие значения получаются в результате вычисления логических выражений? ■

Поддержка S-выражений предлагает удобный способ представления выражений BSL на нашем языке программирования:

```
> (+ 1 1)
2
> '(+ 1 1)
(list '+ 1 1)
> (+ (* 3 3) (* 4 4))
25
> '(+ (* 3 3) (* 4 4))
(list '+ (list '* 3 3) (list '* 4 4))
```

Просто поставив кавычки перед выражениями, мы получаем данные ISL+.

Интерпретировать представления в форме S-выражений неудобно, в основном потому, что не все S-выражения могут служить представлениями BSL-выражений. Например, #true, "hello" и '(+ x 1) не являются представителями выражений на языке BSL. Поэтому S-выражения довольно неудобны для разработчиков интерпретаторов.

Чтобы преодолеть разрыв между удобством использования и реализацией, программисты изобрели *синтаксические анализаторы*, или *парсеры*. Парсер одновременно проверяет, соответствует ли некоторый фрагмент данных определению, и если соответствует, то создает элемент из выбранного класса данных. Если данные не соответству-

ют определению, то синтаксический анализатор сообщает об ошибке, подобно проверяющим функциям из раздела 6.3.

В листинге 81 представлен синтаксический анализатор языка BSL на основе S-выражений. Он принимает S-выражение и возвращает BSL-expr тогда и только тогда, когда данное S-выражение является результатом цитирования выражения BSL, имеющего представление BSL-expr.

Упражнение 349. Создавайте тесты для функции `parse`, пока DrRacket не сообщит вам, что тесты охватывают все элементы в области определений. ■

Упражнение 350. Что необычного в определении этой программы с точки зрения рецепта проектирования?

Примечание. Один необычный аспект заключается в том, что `parse` применяет `length` к списку аргументов. Настоящие парсеры избегают этого, потому что это замедляет их работу. ■

Упражнение 351. Спроектируйте функцию `interpreter-expr`. Она должна принимать S-выражения. Если `parse` распознает их как BSL-expr, то функция должна возвращать их значения. В противном случае она должна возвращать ту же ошибку, которую вернет `parse`. ■

Листинг 81. Преобразование S-expr в BSL-expr

```

; S-expr -> BSL-expr
(define (parse s)
  (cond
    [(atom? s) (parse-atom s)]
    [else (parse-sl s)]))

; SL -> BSL-expr
(define (parse-sl s)
  (local ((define L (length s)))
    (cond
      [(< L 3) (error WRONG)]
      [(and (= L 3) (symbol? (first s)))
       (cond
         [(symbol=? (first s) '+)
          (make-add (parse (second s)) (parse (third s)))]
         [(symbol=? (first s) '*)
          (make-mul (parse (second s)) (parse (third s)))]
         [else (error WRONG)])]
        [else (error WRONG)]))]

; Atom -> BSL-expr
(define (parse-atom s)
  (cond
    [(number? s) s]
    [(string? s) (error WRONG)]
    [(symbol? s) (error WRONG)]))

```

21.2. Интерпретация переменных

В первом разделе мы игнорировали определения констант, поэтому для нас выражение не имеет значения, если оно содержит перемен-

ную. И действительно, бессмысленно вычислять выражение $(+ 3 x)$, если мы не знаем, что означает x . Поэтому следующим нашим шагом на пути уточнения интерпретатора станет добавление поддержки переменных. Предположим, что в области определений находится следующее определение:

```
| (define x 5)
```

и программист вычисляет выражения с переменной x в области взаимодействий:

```
> x
5
> (+ x 3)
8
> (* 1/2 (* x 3))
7.5
```

Вы можете представить себе второе определение, скажем $(define y 3)$, и взаимодействия, включающие две переменные:

```
> (+ (* x x)
      (* y y))
34
```

В предыдущем разделе в роли переменных неявно предлагаются символы. В конце концов, если для представления выражений с переменными выбрать цитированные S-выражения, то символы появятся естественным образом:

```
> 'x
'x
> '(* 1/2 (* x 3))
(list '* 0.5 (list '* 'x 3))
```

Очевидной альтернативой является строка, соответственно, " x " будет представлять x , но эта книга не о разработке интерпретаторов, поэтому продолжим придерживаться символов. Это решение подсказывает, как изменить определение данных из упражнения 345:

```
; BSL-var-expr -- это одно из значений:
; -- Number
; -- Symbol
; -- (make-add BSL-var-expr BSL-var-expr)
; -- (make-mul BSL-var-expr BSL-var-expr)
```

Мы просто добавили одно предложение в определение данных.

Теперь перейдем к примерам данных: в следующей таблице показаны некоторые выражения BSL с переменными и их представление в форме BSL-var-expr:

Выражение BSL	Представление выражения BSL
x	'x
$(+ x 3)$	(make-add 'x 3)
$(* 1/2 (* x 3))$	(make-mul 1/2 (make-mul 'x 3))
$(+ (* x x) (* y y))$	(make-add (make-mul 'x 'x) (make-mul 'y 'y))

Все они взяты из приведенных выше взаимодействий, то есть что уже вы знаете результаты, когда x равно 5, а y равно 3.

Один из способов определить значение выражения с переменными – заменить все переменные их значениями. Это замечательный способ, и он известен вам из школьных уроков математики.

Упражнение 352. Спроектируйте функцию `subst`. Она должна принимать BSL-var-expr `ex`, Symbol `x` и Number `v` и возвращать результат типа BSL-var-expr, подобный `ex`, в котором все вхождения `x` заменены на `v`. ■

Упражнение 353. Спроектируйте функцию `numegic?`. Она должна определять, является ли выражение BSL-var-expr выражением BSL-expr. Здесь предполагается, что решение упражнения 345 – это определение BSL-var-expr без символов (Symbol). ■

Упражнение 354. Спроектируйте функцию `eval-variable`. Это должна быть проверяющая функция, которая принимает BSL-var-expr и вычисляет его значение, если `numegic?` вернет для него `#true`. В противном случае она должна сообщить об ошибке.

Как правило, программа определяет множество констант в области определений, и выражения часто содержат более одной переменной. Для вычисления таких выражений необходимо иметь представление области определений, содержащей серию определений констант. Для этого упражнения мы используем ассоциативные списки:

```
| ; AL (сокращенно от association list -- ассоциативный список) -- это
| ; [List-of Association].
| ; Association -- это список с двумя элементами:
| ; (cons Symbol (cons Number '())).
```

Спроектируйте функцию `eval-variable*`. Она должна принимать BSL-var-expr `ex` и ассоциативный список `da`. Начиная с `ex`, он должна итеративно применить `subst` ко всем ассоциациям в `da`. Если для полученного результата `numegic?` вернет `#true`, то затем функция должна вычислить значение выражения; в противном случае она должна вернуть ту же ошибку, что и `eval-variable`. **Подсказка.** Рассматривайте заданное выражение BSL-var-expr как элементарное значение и просто выполните обход заданного ассоциативного списка. Мы даем эту подсказку, потому что для проектирования данной функции необходимы некоторые знания из главы 23. ■

МОДЕЛЬ ОКРУЖЕНИЯ. Упражнение 354 основано на математическом понимании определений констант. Если имя обозначает некоторое значение, то все вхождения этого имени можно заменить этим значением. Замена выполняется один раз перед началом процесса вычисления.

Альтернативный подход, получивший название *модель окружения*, заключается в поиске значения переменной по мере необходимости. Интерпретатор начинает обработку выражения немедленно, но точно так же использует представление области определений. Каждый раз, встретив переменную, интерпретатор ищет ее значение в области определений и использует его.

Упражнение 355. Спроектируйте функцию eval-var-lookup. Она имеет ту же сигнатуру, что и eval-variable*:

```
| ; BSL-var-expr AL -> Number
| (define (eval-var-lookup e da) ...)
```

Только вместо предварительной подстановки значений переменных эта функция должна вычислять выражение, как предложено в рецепте проектирования для BSL-var-expr. Анализируя выражение, она использует da и, встретив символ x, assq для поиска значения x в ассоциативном списке. Если значение отсутствует, eval-var-lookup должна сообщить об ошибке. ■

21.3. Интерпретация функций

На этом этапе вы уже имеете представление о том, как вычислять программы на BSL, состоящие из определений констант и выражений с переменными. Следующий естественный шаг – добавление определений функций, чтобы знать – по крайней мере в принципе, – как работать с полным набором возможностей BSL.

Цель этого раздела – уточнить интерпретатор из раздела 21.2, чтобы он мог обрабатывать функции. Поскольку определения функций находятся в области определений, об интерпретаторе можно сказать, что он имитирует DrRacket, когда область определений содержит несколько определений функций, а программист вводит выражение в области взаимодействий, которое применяет эти функции.

Для простоты предположим, что все функции в области определений принимают один аргумент и что существует только одно такое определение. Необходимые знания предметной области получены вами еще в школе, где вы узнали, что $f(x) = e$ представляет определение функции f , что $f(a)$ представляет применение f к a и что для вычисления последнего нужно заменить a на x в e . Как оказывается, вычисление результата применения функции на таком языке, как BSL, работает практически так же.

Перед выполнением следующих упражнений вы можете освежить свои знания терминологии, касающейся функций, представленной в интермэццо 1. В школьном курсе алгебры этот аспект математики часто игнорируется, тем не менее точное понимание терминологии совершенно необходимо для решения подобных задач.

Упражнение 356. Расширьте представление данных из раздела 21.2, добавив в него применение функции, определяемой программистом. Напомним, что выражение применения функции состоит из двух частей: имени и выражения. Имя – это имя применяемой функции; а выражение – это аргумент.

Представьте следующие выражения: $(k (+ 1 1))$, $(* 5 (k (+ 1 1)))$, $(* (i 5) (k (+ 1 1)))$. Назовем этот вновь определенный класс данных *BSL-fun-expr*. ■

Упражнение 357. Спроектируйте функцию `eval-definition1`. Она должна принимать четыре аргумента:

- 1) BSL-fun-expr `ex`;
- 2) символ `f`, представляющий имя функции;
- 3) символ `x`, представляющий параметр функции;
- 4) BSL-fun-expr `b`, представляющий тело функции.

Она должна определить значение `ex`. Встретив применение `f` к некоторому аргументу, `eval-definition1` должна:

- 1) вычислить значение аргумента;
- 2) подставить значение аргумента вместо `x` в `b`;
- 3) вычислить получившееся выражение вызовом `eval-definition1`.

Вот как можно выразить эти шаги в виде кода, если предположить, что `arg` является аргументом в выражении применения функции:

```
(local ((define value (eval-definition1 arg f x b))
       (define plugd (subst b x arg-value)))
      (eval-definition1 plugd f x b))
```

Обратите внимание, что в этой строке используется форма рекурсии, которую мы еще не рассматривали. Рецепт проектирования таких функций обсуждается в части V.

Если `eval-definition1` обнаруживает отсутствие определения переменной, то возвращает ту же ошибку, что и `eval-variable` из упражнения 354. Она также сообщает об ошибке для выражений применения функций, которые ссылаются на функции, имена которых отличаются от `f`.

Внимание. Использование этой формы открытой рекурсии вводит в вычисления новый элемент: незавершенность. То есть программа может зациклиться, вместо того чтобы вернуть результат или сообщить об ошибке. Если вы будете следовать рецептам проектирования из первых четырех частей, то у вас не получится писать такие программы. Ради интереса определите такие входные аргументы для `eval-definition1`, которые заставят ее работать вечно. Используйте кнопку **STOP** (Остановить), чтобы остановить программу. ■

Для интерпретатора, имитирующего область взаимодействий, необходимо представление области определений. Мы предполагаем, что это представление имеет вид списка определений.

Упражнение 358. Разработайте структуру и определение данных для представления определений функций. Напомним, что такое определение имеет три основных атрибута:

- 1) имя функции, представленное символом;
- 2) параметр функции, который тоже является именем;
- 3) тело функции, которое является выражением с переменной.

Для обозначения этого класса данных мы используем имя `BSL-fun-def`.

Используйте свое определение данных для представления следующих определений функций:

1. (define (f x) (+ 3 x))
2. (define (g y) (f (* 2 y)))
3. (define (h v) (+ (f v) (g v)))

Затем определите класс *BSL-fun-def** для представления области определений, состоящей из последовательности определений функций с одним аргументом. Преобразуйте область определений с функциями f, g и h в ваше представление данных и назовите его da-fgh.

Наконец, попробуйте реализовать следующий пункт в списке желаний:

```
| ; BSL-fun-def* Symbol -> BSL-fun-def
| ; извлекает определение f из da
| ; сообщает об ошибке, если такого определения нет
| (check-expect (lookup-def da-fgh 'g) g)
| (define (lookup-def da f) ...)
```

Поиск определения необходим для вычисления выражений применения функций. ■

Упражнение 359. Спроектируйте функцию eval-function*. Она должна принимать BSL-fun-expr ex, представление области определений BSL-fun-def* da и возвращать результат, который показывает DrRacket при вычислении ex в области взаимодействий, если область определений содержит da.

Эта функция действует так же, как eval-definition1 из упражнения 357. Для применения некоторой функции f она:

- 1) вычисляет значение аргумента;
- 2) ищет определение f в представлении BSL-fun-def da, которое имеет параметр и тело;
- 3) заменяет параметр в теле функции значением аргумента;
- 4) вычисляет получившееся выражение с помощью рекурсии.

Как и DrRacket, функция eval-function* должна сообщать об ошибке, встретив имя переменной или функции, определение которого отсутствует в области определений. ■

21.4. Интерпретация всего и вся

Взгляните на следующую программу на BSL:

```
(define close-to-pi 3.14)

(define (area-of-circle r)
  (* close-to-pi (* r r)))

(define (volume-of-10-cylinder r)
  (* 10 (area-of-circle r)))
```

Считайте, что эти определения находятся в области определений в DrRacket. После щелчка на кнопке **RUN** (Выполнить) вы сможете вычислить выражения в области взаимодействий, включающие `close-to-pi`, `area-of-circle` и `volume-of-10-cylinder`:

```
> (area-of-circle 1)
#i3.14
> (volume-of-10-cylinder 1)
#i31.40000000000002
> (* 3 close-to-pi)
#i9.42
```

Цель этого раздела – еще раз уточнить интерпретатор, чтобы он мог имитировать большую часть возможностей DrRacket.

Упражнение 360. Сформулируйте определение данных для представления области определений в DrRacket. В частности, представление данных должно иметь вид последовательности, которая без ограничений может включать определения констант и функций с одним аргументом. Убедитесь, что ваше определение может представить область определений с тремя определениями в начале этого раздела. Назовите этот класс данных *BSL-da-all*.

Спроектируйте функцию `lookup-con-def`. Она должна принимать BSL-da-all `da` и символ `x` и возвращать представление определения константы с именем `x`, если такая имеется в `da`; в противном случае функция должна сообщать, что такая константа не определена.

Спроектируйте функцию `lookup-fun-def`. Он должна принимать BSL-da-all `da` и символ `f` и возвращать представление определения функции с именем `f`, если такая имеется в `da`; в противном случае функция должна сообщать, что такая функция не определена. ■

Упражнение 361. Спроектируйте функцию `eval-all`. Подобно функции `eval-function*` из упражнения 359, она должна принимать представление выражения и области определений и возвращать то же значение, что и DrRacket, если ввести это выражение в области взаимодействий, а область определений содержит соответствующие определения. **Подсказка.** Функция `eval-all` должна обрабатывать переменные в заданном выражении так же, как `eval-var-lookup` в упражнении 355. ■

Упражнение 362. Вводить структурные представления выражений BSL и области определений очень неудобно. Как было показано в конце раздела 21.1, гораздо проще использовать цитирование выражений и (списки) определений.

Спроектируйте функцию `interpret`. Она должна принимать представление выражения `S-expr` и список определений `sl`, выполнять синтаксический анализ с помощью соответствующих функций, а затем использовать `eval-all` из упражнения 361 для вычисления выражения. **Подсказка.** Адаптируйте идеи из упражнения 350, чтобы создать синтаксический анализатор определений и списков определений.

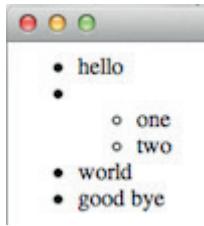
Имейте в виду, что проверить, насколько точно interpreter имитирует работу DrRacket, не составляет никакого труда. ■

На данный момент вы многое знаете об интерпретации языка BSL. Вот еще несколько недостающих элементов: логические значения в cond и if; строки и операции со строками, такие как string-length и string-append; списки с '(), empty?, cons, cons?, first, rest и т. д. Как только ваш интерпретатор сможет справляться со всем этим, он будет практически закончен, потому что ваш интерпретатор уже знает, как интерпретировать рекурсивные функции. Теперь, говоря вам: «проверьте, вы уже знаете, как проектировать эти уточнения», – мы имеем в виду именно это.

22. Проект: обработка XML

XML – это широко используемый язык представления данных. Одно из его применений – обмен сообщениями между программами, работающими на разных компьютерах. Например, когда вы вводите адрес сайта в своем веб-браузере, вы фактически подключаете программу на своем компьютере к программе на другом компьютере, и программа на другом компьютере посыпает вашей программе данные в формате XML. Получив данные XML, браузер отображает их в виде изображения на мониторе вашего компьютера.

Следующее сравнение иллюстрирует эту идею на конкретном примере:

Данные в формате XML	Изображение в браузере
<pre> hello one two world good bye </pre>	

Слева показан фрагмент данных в формате XML, которые веб-сайт мог отправить вашему веб-браузеру, а справа – как один популярный браузер отображает этот фрагмент в своем окне.

Если вы считаете, что формат XML слишком устарел, то можете заменить его форматом JSON или любым другим.

Принципы проектирования от этого не изменяются.

В этой главе объясняются основы обработки XML как еще одно упражнение по проектированию, затрагивающее определения взаимосвязанных данных и итеративное уточнение. Следующий раздел начинается с неформального сравнения S-выражений и XML и использует его для формулирования полноценного определения данных. В остальных разделах на примерах объясняется, как обрабатывать S-выражение с данными XML.

22.1. XML как S-выражения

В простейшем виде данные в формате XML выглядят так:

```
| <machine> </machine>
```

Это называется *элементом*, а «*machine*» – это имя элемента. Две части элемента похожи на скобки, ограничивающие *содержимое* элемента. Когда между двумя частями нет содержимого, кроме пробела, язык XML допускает сокращение:

```
| <machine />
```

Но для нас это сокращение эквивалентно версии, заключенной в скобки.

С точки зрения S-выражений, элемент XML – это **именованная** пара скобок, окружающих некоторое содержимое. Представление выше естественно выражается с помощью S-выражения:

В Racket имеется библиотека `xml`, определяющая структуры и S-выражения для представления XML.

```
| '(machine)
```

Этот фрагмент данных имеет открывающую и закрывающую круглые скобки, и в нем есть место для содержимого.

Вот фрагмент XML-данных с содержимым:

```
| <machine><action /></machine>
```

Как вы наверняка догадались, `<action />` – это сокращенное представление, то есть полная версия данных выглядит так:

```
| <machine><action></action></machine>
```

В общем случае содержимое элемента XML – это последовательность других элементов XML:

```
| <machine><action /><action /><action /></machine>
```

Стоп! Прежде чем продолжить, разверните сокращение `<action />`.

Представление в виде S-выражения остается простым. Вот представление для первого примера:

```
| '(machine (action))
```

А вот для второго:

```
| '(machine (action) (action) (action))
```

Просматривая фрагмент XML-данных, включающий последовательность из трех элементов `<action />`, возникает подспудное желание как-то отличить их друг от друга. С этой целью XML-элементы имеют *атрибуты*. Например:

```
| <machine initial = "red"> </machine>
```

Этот элемент «`machine`» имеет один атрибут с именем «`initial`» и значением «`red`», заключенным в двойные кавычки. Вот более сложный XML-элемент сложенными элементами, тоже имеющими атрибуты:

```
<machine initial="red">
  <action state="red"    next="green" />
  <action state="green"  next="yellow" />
  <action state="yellow" next="red" />
</machine>
```

Мы используем пробелы для оформления отступов и разрывы строк, чтобы сделать элемент более удобочитаемым, но для XML-данных эти пробелы не имеют значения.

XML на 40 лет моложе S-выражений. Естественно, S-выражения для этих элементов «machine» очень похожи на своих собратьев в XML:

```
| '(machine ((initial "red")))
```

Чтобы добавить атрибуты в элемент, мы используем список списков, каждый из которых содержит два элемента: символ и строку. Символ представляет имя атрибута, а строка – его значение. Эта идея, естественно, применима и к более сложным формам XML-данных:

```
'(machine ((initial "red"))
  (action ((state "red") (next "green")))
  (action ((state "green") (next "yellow")))
  (action ((state "yellow") (next "red"))))
```

Обратите внимание, что атрибуты выделяются двумя открывающими круглыми скобками, а оставшийся список (представлений) элементов XML – одной открывающей скобкой.

Возможно, вы помните, как в интермеццо 2 мы говорили, что S-выражения используются для представления XHTML, особого диалекта XML. В частности, в интермеццо было показано, насколько легко программист может записывать нетривиальные XML-данные и даже макеты XML-представлений, используя `backquote` и `unquote`. В разделе 21.1 отмечалось, что нужен синтаксический анализатор (парсер), чтобы определить, является ли данное S-выражение представлением XML-данных, а синтаксический анализатор – это сложный и необычный вид функции.

Тем не менее мы выбрали представление XML на основе S-выражений, чтобы продемонстрировать практическую пользу этой старой и очень интересной идеи. Теперь мы постепенно переходим к разработке определения данных, следуя методике итеративного уточнения. Вот первая попытка:

```
| ; Xexprg.v0 (сокращенно от X-expression -- X-выражение) -- это
| ;   список с одним элементом:
| ;   (cons Symbol '())
```

Это выражение идеи «именных скобок», представленной в начале раздела. Добавить содержимое в это представление элемента очень просто:

```
| ; Xexprg.v1 -- это список:
| ;   (cons Symbol [List-of Xexprg.v1])
```

Символьное имя становится первым элементом в списке, остальная часть которого состоит из представлений элементов XML.

Последний шаг уточнения – добавление атрибутов. Поскольку атрибуты являются необязательными, обновленное определение данных содержит два пункта:

```

; Xexpr.v2 -- это список:
; -- (cons Symbol Body)
; -- (cons Symbol (cons [List-of Attribute] Body))
; где Body представляет [List-of Xexpr.v2]
; Attribute -- это список с двумя элементами:
;   (cons Symbol (cons String '()))

```

Упражнение 363. Все элементы в Xexpr.v2 начинаются с символа, но за некоторыми следует список атрибутов, а за некоторыми – просто список Xexpr.v2s. Переформулируйте определение Xexpr.v2, чтобы отделить общее начало от различных типов окончаний.

Исключите List-of из Xexpr.v2. ■

Упражнение 364. Представьте следующие XML-данные в виде элементов Xexpr.v2:

1. <transition from="seen-e" to="seen-f" />
2. <word /><word /><word />

Какие из них можно представить с помощью Xexpr.v0 или Xexpr.v1? ■

Упражнение 365. Интерпретируйте следующие элементы Xexpr.v2 как данные XML:

1. '(server ((name "example.org")))
2. '(cargas (board (grass)) (player ((name "sam"))))
3. '(start)

Какие из них являются элементами Xexpr.v0 или Xexpr.v1? ■

В общем случае X-выражения моделируют структуры через списки. Такая модель удобна для программистов; она требует минимального ввода с клавиатуры. Например, если X-выражение не имеет списка атрибутов, то он просто опускается. Такой выбор представления данных является компромиссом между созданием подобных выражений вручную и их автоматической обработкой. Лучший способ организовать подобную автоматическую обработку – спроектировать функции, которые делают X-выражения похожими на структуры, особенно функции, которые обращаются к квазиполям:

- xexpr-name, извлекает тег, представляющий элемент;
- xexpr-attr, извлекает список атрибутов;
- xexpr-content, извлекает список элементов содержимого.

Имея такие функции, мы можем использовать для представления XML списки, которые действуют так, как если бы они были экземплярами структур.

Эти функции анализируют S-выражения, и их сложно спроектировать. Поэтому подробно рассмотрим процесс проектирования, начав с некоторых примеров данных:

```

(define a0 '((initial "X")))

(define e0 '(machine))
(define e1 `(machine ,a0))
(define e2 `(machine (action)))
(define e3 `(machine () (action)))
(define e4 `(machine ,a0 (action) (action)))

```

Первое определение вводит список атрибутов и дважды в последующих X-выражениях. Определение `e0` напоминает нам, что X-выражение не может иметь атрибутов или содержимого. Вы должны уметь объяснить, почему `e2` и `e3` почти эквивалентны друг другу.

Затем мы формулируем сигнатуру, заявление о назначении и заголовок:

```
| ; Xexpr.v2 -> [List-of Attribute]
| ; извлекает список атрибутов из xe
| (define (xexpr-attr xe) '())
```

Тут мы сосредоточимся только на `xexpr-attr`, а остальные функции оставляем вам в качестве упражнения.

Составляя примеры применения функции, мы должны решить, как поступить, если они отсутствуют в X-выражении. Мы выбрали представление, в котором опущены отсутствующие атрибуты, тем не менее мы должны вернуть `'()`, чтобы обеспечить структурное представление XML. Поэтому для таких X-выражений функция должна возвращать `'()`:

```
| (check-expect (xexpr-attr e0) '())
| (check-expect (xexpr-attr e1) '((initial "X"))))
| (check-expect (xexpr-attr e2) '())
| (check-expect (xexpr-attr e3) '())
| (check-expect (xexpr-attr e4) '((initial "X")))
```

Теперь переходим к макету. Учитывая сложность определения данных для `Xexpr.v2`, будем двигаться не торопясь, шаг за шагом. Во-первых, определение данных различает два вида X-выражений, но оба предложения описывают данные, построенные путем включения символа в список. Во-вторых, эти два предложения отличаются оставшейся (`rest`) частью списка, в частности необязательным списком атрибутов. Давайте воплотим эти две идеи в макет:

```
| (define (xexpr-attr xe)
|   (local ((define optional-loa+content (rest xe)))
|     (cond
|       [(empty? optional-loa+content) ...]
|       [else ...])))
```

Локальное определение отбрасывает имя X-выражения и оставляет оставшуюся часть списка, который может начинаться или не начинаться со списка атрибутов. Важно помнить, что это просто список, и два предложения в операторе `cond` указывают на это. В-третьих, этот список определяется **не** через ссылку на себя, а как список, включающий необязательный список атрибутов и, возможно, пустой список X-выражений. Иначе говоря, мы должны различать два обычных случая и извлекать обычные части:

```
| (define (xexpr-attr xe)
|   (local ((define optional-loa+content (rest xe)))
|     (cond
```

```
[(empty? optional-loa+content) ...]
[else (... (first optional-loa+content)
... (rest optional-loa+content) ...))])
```

На этом этапе мы уже видим, что для решения данной задачи не нужна рекурсия. Теперь переходим к пятому шагу рецепта проектирования. Очевидно, что если данное X-выражение не содержит ничего, кроме имени, то в нем нет атрибутов. Во втором предложении мы должны определить, является ли первый элемент в списке списком атрибутов или это просто Xexpr.v2. Звучит довольно сложно, поэтому добавляем еще одну функцию в список желаний:

```
; [List-of Attribute] или Xexpr.v2 -> ???
; определяет, является ли x списком [List-of Attribute]
; если нет, то возвращает #false
(define (list-of-attributes? x)
#false)
```

Имея эту функцию, мы легко можем завершить `xexpr-attr`; см. листинг 82. Если первый элемент является списком атрибутов, то функция создает его; в противном случае атрибутов нет.

Листинг 82. Законченное определение `xexpr-attr`

```
(define (xexpr-attr xe)
  (local ((define optional-loa+content (rest xe)))
    (cond
      [(empty? optional-loa+content) '()]
      [else
        (local ((define loa-or-x
                      (first optional-loa+content)))
          (if (list-of-attributes? loa-or-x)
            loa-or-x
            '()))]))))
```

Проектируем функцию `list-of-attributes?` как обычно и получаем следующее определение:

```
; [List-of Attribute] или Xexpr.v2 -> Boolean
; является ли x списком атрибутов
(define (list-of-attributes? x)
  (cond
    [(empty? x) #true]
    [else
      (local ((define possible-attribute (first x)))
        (cons? possible-attribute))))
```

Мы опустили описание процесса проектирования, потому что он ничем не примечателен. Но обратите внимание на сигнатуру этой функции. Вместо одного определения данных для входных данных в сигнатуре указаны два определения, разделенных словом «или». В ISL+ такие неформальные сигнатуры иногда допустимы.

Упражнение 366. Спроектируйте `xexpr-name` и `xexpr-content`. ■

Упражнение 367. Рецепт проектирования требует наличия в макете `xexpr-attr` в определении данных ссылки на самого себя. Добавьте

эту ссылку в макет, а затем объясните, почему законченная функция синтаксического анализа не содержит ее. ■

Упражнение 368. Сформулируйте определение данных, которое заменяет неформальное слово «или» в сигнатуре `list-of-attributes?`. ■

Упражнение 369. Спроектируйте `find-attr`. Функция должна принимать список атрибутов и символ. Если список атрибутов связывает символ со строкой, то функция должна вернуть эту строку; иначе она должна вернуть `#false`. Найдите `assq` и используйте ее в определении функции. ■

В оставшейся части этой главы под *Xexpr* мы будем подразумевать *Xexpr.v2*. Кроме того, мы будем предполагать, что *xexpr-name*, *xexpr-attr* и *xexpr-content* уже определены. Наконец, мы будем использовать `find-attr` из упражнения 369 для получения значений атрибутов.

22.2. Отображение XML-перечислений

На самом деле XML – это **семейство** языков. Люди определяют свои диалекты для обмена данными. Например, XHTML – это язык для передачи веб-контента в формате XML. В этом разделе мы покажем, как спроектировать функцию для отображения для небольшого фрагмента XHTML, в частности перечислений, приводившихся в начале этой главы.

Тег `ul` определяет так называемый неупорядоченный список HTML. Каждый элемент этого списка заключен в тег `li` и обычно содержит слова, а также другие элементы, в том числе и иные перечисления. Под словом «неупорядоченный» подразумевается, что каждый элемент должен отображаться с маркером в начале вместо числа.

Поскольку *Xexpr* не содержит простых строк, то на первый взгляд неочевидно, как представлять перечисления XHTML. Один из вариантов – еще раз уточнить представление данных, чтобы *Xexpr* мог быть строкой. Другой вариант – ввести представление для текста:

```
| ; XWord -- это '(word ((text String))).
```

Далее мы будем использовать второй вариант; язык Racket, на котором основаны обучающие языки, предлагает библиотеки, которые включают определения `String` в *Xexpr*.

Упражнение 370. Сконструируйте три примера для *XWord*. Спроектируйте функцию `word?`, которая проверяет, находится ли какое-либо значение `ISL+` в *XWord*, и функцию `word-text`, извлекающую значение единственного атрибута экземпляра *XWord*. ■

Упражнение 371. Уточните определение *Xexpr*, чтобы получить возможность представлять элементы XML, включая элементы перечислений, которые являются простыми строками. ■

Опираясь на представление слов, мы легко можем сформулировать представление перечисления слов в стиле XHTML:

```
; XEnum.v1 -- это одно из значений:  
; -- (cons 'ul [List-of XItem.v1])  
; -- (cons 'ul (cons Attributes [List-of XItem.v1]))  
  
; XItem.v1 -- это одно из значений:  
; -- (cons 'li (cons XWord '()))  
; -- (cons 'li (cons Attributes (cons XWord '()))))
```

Для полноты определение данных включает списки атрибутов, даже притом что они не влияют на отображение.

Стоп! Докажите, что каждый элемент XEnum.v1 также является экземпляром XExpr. Вот пример элемента XEnum.v1:

```
(define e0  
  '(ul  
    (li (word ((text "one"))))  
    (li (word ((text "two"))))))
```

Это соответствует внутреннему перечислению из примера в начале главы. При отображении с помощью библиотеки *2htdp/image* должно получиться такое изображение:



Размер маркеров и расстояние между маркером и текстом относятся к вопросу эстетически; для нас же важнее сама идея.

Чтобы создать такое изображение, можно воспользоваться следующей программой на ISL+:

Мы разработали эти выражения в области взаимодействий. А как поступили бы вы?

```
(define e0-rendered  
  (above/align  
    'left  
    (beside/align 'center BT (text "one" 12 'black))  
    (beside/align 'center BT (text "two" 12 'black))))
```

где BT – это изображение маркера.

Теперь спроектируем функцию. Для представления данных требуется два определения, поэтому, согласно рецепту проектирования, мы должны проектировать две функции параллельно. Однако, внимательно изучив задачу, можно заметить, что в этом конкретном случае второе определение данных не связано с первым, а это означает, что проектировать функции можно по отдельности.

Кроме того, определение XItem.v1 состоит из двух предложений, то есть сама функция должна состоять из оператора cond с двумя предложениями. Однако цель рассмотрения XItem.v1 как подъязыка Xexpr состоит в том, чтобы интерпретировать эти два предложения в терминах функций-селекторов Xexpr, в частности xexpr-content. С помощью этой функции можно извлечь текстовую часть элемента независимо от наличия атрибутов:

```
| ; XItem.v1 -> Image
| ; отображает элемент как "слово" с маркером перед ним
| (define (render-item1 i)
|   (... (xexpr-content i) ...))
```

В общем случае `xexpr-content` извлекает список `XExpr`, но в данной ситуации список содержит ровно один экземпляр `XWord` с текстом:

```
| (define (render-item1 i)
|   (local ((define content (xexpr-content i))
|           (define element (first content))
|           (define a-word (word-text element)))
|         (... a-word ...)))
```

Дальше все просто:

```
| (define (render-item1 i)
|   (local ((define content (xexpr-content i))
|           (define element (first content))
|           (define a-word (word-text element))
|           (define item (text a-word 12 'black)))
|         (beside/align 'center BT item)))
```

После извлечения текста для отображения в элементе остается только отобразить его и снабдить начальным маркером; см. примеры выше, чтобы узнать, как выполнить этот последний шаг.

Упражнение 372. Прежде чем продолжить, добавьте тестовые примеры для `render-item1`. Обязательно сформулируйте тесты таким образом, чтобы они не зависели от константы `BT`. Затем объясните, как работает функция; имейте в виду, что в заявлении о назначении объясняется, что она делает. ■

Теперь можно сосредоточиться на проектировании функции, которая отображает перечисление. С учетом примеров выше первые два шага проектирования не вызывают никаких сложностей:

```
| ; XEnum.v1 -> Image
| ; отображает простое перечисление в виде изображения
| (check-expect (render-enum1 e0) e0-rendered)
| (define (render-enum1 xe) empty-image)
```

Ключевым этапом является разработка макета. Согласно определению данных, экземпляр `XEnum.v1` содержит один интересный элемент данных, а именно XML-элементы (точнее их представление). Первым элементом всегда является `ul`, поэтому нет необходимости извлекать его, а второй, необязательный элемент представляет список атрибутов, которые мы игнорируем. Учитывая это, первый черновой макет выглядит так же, как макет для `render-item1`:

```
| (define (render-enum1 xe)
|   (... (xexpr-content xe) ...)) ; [List-of XItem.v1]
```

Рецепт проектирования, основывающийся на данных, говорит, что мы должны создавать отдельные функции всякий раз, когда встречается сложная форма данных, а рецепт проектирования, основываю-

щийся на абстракциях и представленный в части III, предлагает повторно использовать существующие абстракции, например функции обработки списков из листингов 56 и 57, если это возможно. Функция `render-enum1` должна обрабатывать список и создавать из него изображение, а единственными двумя абстракциями обработки списков, сигнатуры которых соответствуют всем требованиям, являются `foldr` и `foldl`. Если также вспомнить их заявления о назначении, то мы приедем к макету, похожему на пример `e0-rendered` выше. Попробуем использовать его, следуя рецепту проектирования, рекомендующему повторно использовать существующие абстракции:

```
(define (render-enum1 xe)
  (local ((define content (xexpr-content xe))
          ; XItem.v1 Image -> Image
          (define (deal-with-one item so-far)
            ...))
    (foldr deal-with-one empty-image content)))
```

Вы также наверняка помните, что:

- 1) первым аргументом `foldr` должна быть функция с двумя аргументами;
- 2) второй аргумент должен быть изображением;
- 3) последний аргумент – это список, представляющий содержимое XML.

Естественно, что правильной отправной точкой будет для нас `empty-image`.

Рецепт проектирования на основе повторного использования фокусирует наше внимание на функции, выполняющей «свертку» списка. Она превращает один элемент и изображение, созданное `foldr`, в другое изображение. Сигнатура `deal-with-one` выражает это понимание. Поскольку первым аргументом является `XItem.v1`, то `render-item1` – это функция, которая его отображает. В результате получаются два изображения, которые необходимо объединить: изображение первого элемента и изображение остальных элементов. Чтобы объединить их, используем функцию `above`:

```
(define (render-enum1 xe)
  (local ((define content (xexpr-content xe))
          ; XItem.v1 Image -> Image
          (define (deal-with-one item so-far)
            (above/align 'left
              (render-item1 item)
              so-far)))
    (foldr deal-with-one empty-image content)))
```

Листинг 83. Представление данных для XML-перечислений

```
; An XItem.v2 -- это одно из значений:
; -- (cons 'li (cons XWord '()))
; -- (cons 'li (cons [List-of Attribute] (list XWord)))
; -- (cons 'li (cons XEnum.v2 '()))
```

```

; -- (cons 'li (cons [List-of Attribute] (list XEnum.v2)))
;
; An XEnum.v2 -- это одно из значений:
; -- (cons 'ul [List-of XItem.v2])
; -- (cons 'ul (cons [List-of Attribute] [List-of XItem.v2]))

```

Если вам интересно, является ли произвольное вложение правильным решением данной задачи, то разработайте определение данных, которое допускает только три уровня вложенности, а затем попробуйте использовать его.

Плоские перечисления встречаются чаще всего, но они являются лишь частным случаем. В реальном мире веб-браузеры должны уметь отображать произвольно вложенные перечисления. В XML и его диалекте XHTML вложение выполняется просто. Любой элемент может отображаться как содержимое любого другого элемента. Чтобы обозначить эту взаимосвязь в нашем ограниченном представлении XHTML, мы говорим, что элемент является либо словом, либо другим перечислением. В листинге 83 показана вторая версия определения данных. Она включает измененные определения данных для представления перечислений, где первое определение ссылается на правильную форму элемента.

Следующий вопрос: как эти изменения в определениях данных влияют на функции отображения. Иначе говоря, мы должны изменить `render-enum1` и `render-item1` так, чтобы они могли обрабатывать `XEnum.v2` и `XItem.v2` соответственно. Инженеры-программисты постоянно сталкиваются с подобными вопросами, и это еще одна ситуация, в которой рецепт проектирования предстает во всем своем блеске.

Полный ответ показан в листинге 84. Поскольку изменения затронули только `XItem.v2`, неудивительно, что изменить потребовалось лишь функцию отображения элементов. Функция `render-item1` не обязана различать разные формы `XItem.v1`, но `render-item` вынуждена использовать `cond`, потому что `XItem.v2` определяет два разных типа элементов. Учитывая, что это определение данных близко к определению из реального мира, отличительной характеристикой является не что-то простое, как, например, `'()` или `cons`, а конкретная часть данного элемента. Если содержимым элемента является `Xword`, то функция отображения действует так же, как раньше. Но если элемент содержит перечисление, то `render-item` применяет `render-enum` к данным, потому что в этой части `XItem.v2` ссылается на `XEnum.v2`.

Листинг 84. Уточненные версии функций, учитывающие уточнения в определениях данных

```

(define SIZE 12) ; размер шрифта
(define COLOR "black") ; цвет шрифта
(define BT ; графическая константа
  (beside (circle 1 'solid 'black) (text " " SIZE COLOR)))

; Image -> Image
; добавляет маркер в элемент
(define (bulletize item)
  (beside/align 'center BT item))

; XEnum.v2 -> Image

```

```

; отображает XEnum.v2 как изображение
(define (render-enum xe)
  (local ((define content (xexpr-content xe))
          ; XItem.v2 Image -> Image
          (define (deal-with-one item so-far)
            (above/align 'left (render-item item) so-far)))
    (foldr deal-with-one empty-image content)))

; XItem.v2 -> Image
; отображает один XItem.v2 как изображение
(define (render-item an-item)
  (local ((define content (first (xexpr-content an-item))))
    (bulletize
      (cond
        [(word? content)
         (text (word-text content) SIZE 'black)]
        [else (render-enum content)]))))
```

Упражнение 373. В листинге 84 отсутствуют тестовые примеры. Разработайте тестовые примеры для всех функций. ■

Упражнение 374. Определения данных в листинге 83 используют `list`. Перепишите их с применением `cons`. Затем спроектируйте функции отображения для `XEnum.v2` и `XItem.v2` с нуля, используя рецепт проектирования. Вы должны получить те же определения, что и в листинге 84. ■

Упражнение 375. Обертывание `cond` с помощью

```
| (beside/align 'center BT ...)
```

может вас удивить. Измените определение функции так, чтобы обертывание выполнялось один раз в каждом предложении. Как можно убедиться, что это изменение работает правильно? Какая версия вам нравится больше? ■

Упражнение 376. Спроектируйте программу, которая подсчитывает все вхождения слова «hello» в экземпляре `XEnum.v2`. ■

Упражнение 377. Спроектируйте программу, которая заменяет все слова «hello» в перечислении словами «bye». ■

22.3. Предметно-ориентированные языки

Инженеры часто создают большие программные системы, требующие настройки для работы в определенном контексте. Задача настройки обычно выпадает на долю *системных администраторов*, которым приходится иметь дело с множеством различных программных систем. Слово «настройка» относится к данным, которые необходимо передать главной функции при запуске программы. В некотором смысле настройки (или конфигурация) – это просто дополнительный аргумент, хотя часто они настолько сложны, что разработчики программ предпочитают использовать дополнительные механизмы для их передачи.

Поскольку настройки абстрагируют программу в отношении различных элементов данных, Пол Худак (Paul Hudak) утверждал в 1990-х годах, что предметно-ориентированные языки являются полноценной абстракцией, то есть они в совершенстве обобщают идеи, представленные в части III.

Инженеры-программисты не могут опираться на предположение о том, что системные администраторы знают все языки программирования, поэтому они часто создают простые специализированные языки для описания конфигурации. Эти специальные языки также известны как *предметно-ориентированные языки* (Domain Specific Language, DSL). Разработка предметно-ориентированных языков на основе общего ядра, скажем хорошо известного синтаксиса XML, упрощает жизнь системным администраторам. Они могут писать небольшие «программы» на XML и таким образом настраивать свои системы.

Разработка DSL часто считается задачей для опытных программистов, но вы уже обладаете достаточным объемом знаний, чтобы спроектировать и реализовать довольно сложный DSL. В этом разделе мы расскажем вам, как это сделать. Сначала освежим наши знания о конечных автоматах (Finite State Machine, FSM). Затем посмотрим, как спроектировать, реализовать и использовать DSL для настройки системы, моделирующей произвольные конечные автоматы.

Конечные автоматы. Конечные автоматы занимают важное положение в информатике, и в этой книге мы уже применяли их несколько раз. Здесь мы повторно используем пример из раздела 12.8 в качестве основы и реализуем DSL для его настройки.

Для удобства в листинге 85 повторно представлен весь код примера, хотя и несколько измененный для использования всех возможностей ISL+. Программа включает два определения данных, один пример и два определения функций: *simulate* и *find*. В отличие от похожих программ в предыдущих главах, эта представляет переход в виде списка с двумя элементами: текущим состоянием и следующим.

Листинг 85. Конечные автоматы, улучшенная версия

```
; FSM -- это [List-of 1Transition]
; 1Transition -- это список с двумя элементами:
;   (cons FSM-State (cons FSM-State '()))
; FSM-State -- это строка (String), определяющая цвет

; примеры данных
(define fsm-traffic
  '(("red" "green") ("green" "yellow") ("yellow" "red")))

; FSM FSM-State -> FSM-State
; сопоставляет нажатые клавиши с заданным конечным автоматом FSM
(define (simulate state0 transitions)
  (big-bang state0 ; FSM-State
            [to-draw
             (lambda (current)
               (square 100 "solid" current))]
            [on-key
             (lambda (current key-event)
               (find transitions current)))))

; [X Y] [List-of [List X Y]] X -> Y
```

```
; отыскивает состояние Y, соответствующее состоянию X в списке alist
(define (find alist x)
  (local ((define fm (assoc x alist)))
    (if (cons? fm) (second fm) (error "not found"))))
```

Главная функция, `simulate`, принимает таблицу переходов и начальное состояние; затем она вычисляет выражение `big-bang`, которое реагирует на каждое событие нажатия клавиши переходом к следующему состоянию. Состояния отображаются в виде цветных квадратов. Предложения `to-draw` и `on-key` реализованы с использованием лямбда-выражений, которые принимают текущее состояние и событие клавиатуры и создают изображение для следующего состояния.

Как можно заметить по сигнатуре, вспомогательная функция `find` полностью независима от приложения конечного автомата. Она принимает список списков с двумя элементами и искомый элемент, но фактическая природа элементов задается с помощью параметров. В контексте этой программы X и Y представляют состояния конечного автомата, а это означает, что `find` принимает таблицу переходов с исходным состоянием и возвращает новое состояние. Большую часть работы в теле функции выполняет встроенная функция `assoc`. Загляните в документацию с описанием `assoc`, чтобы понять, почему в теле `local` используется выражение `if`.

Упражнение 378. Измените функцию рендеринга так, чтобы она накладывала название состояния на цветной квадрат. ■

Упражнение 379. Сформулируйте тестовые примеры для функции `find`. ■

Упражнение 380. Переформулируйте определение данных `1Transition`, чтобы можно было ограничить переходы нажатием определенных клавиш. Попробуйте сформулировать изменения так, чтобы это не потребовало вносить изменения в `find`. Что еще нужно изменить, чтобы программа продолжала работать? Какая часть рецепта проектирования дает ответ(ы)? См. исходное задание в упражнении 229. ■

Конфигурация. Функция моделирования конечного автомата принимает два аргумента, которые вместе описывают состояние автомата. Вместо того чтобы учить потенциального «покупателя», как запускать программу на `ISL+` в `DrRacket` и вызывать функцию с двумя аргументами, «продавец» функции `simulate` может пожелать дополнить продукт компонентом конфигурации.

Компонент конфигурации состоит из двух частей. Первая часть – это широко используемый и простой язык, который клиенты используют для описания начальных аргументов для главной функции (функций). Вторая – это функция, которая преобразует конфигурацию, написанную клиентом, в вызов главной функции. Таким образом, для симулятора конечного автомата мы должны договориться о том, как представлять конечные автоматы в XML. В разделе 22.1 был представлен ряд примеров, идеально подходящих для данной задачи. Вспомните последний пример тега `machine` в том разделе:

```
<machine initial="red">
  <action state="red"    next="green" />
  <action state="green"  next="yellow" />
  <action state="yellow" next="red" />
</machine>
```

Сравните это с таблицей переходов `fsm-traffic` в листинге 85. Также вспомните согласованное представление Xexpr из этого примера:

```
(define xm0
  '(machine ((initial "red"))
            (action ((state "red") (next "green")))
            (action ((state "green") (next "yellow")))
            (action ((state "yellow") (next "red")))))
```

Нам не хватает только общего определения данных, описывающе-го все возможные представления конечных автоматов Xexpr:

```
; XMachine -- вложенный список следующей формы:
;   ` (machine ((initial ,FSM-State)) [List-of X1T])
; X1T -- вложенный список следующей формы:
;   ` (action ((state ,FSM-State) (next ,FSM-State)))
```

Так же как XEnum.v2, определение XMachine описывает подмно-жество всех Xexpr. Благодаря этому, проектируя функции для обработки этой новой формы данных, мы можем продолжать использовать обобщенные функции Xexpr для доступа к отдельным элементам.

Упражнение 381. В определениях XMachine и X1T используется цитирование, которое с трудом воспринимается начинающими проектировщиками программ. Перепишите их, сначала с использованием `list`, а затем `cons`. ■

Упражнение 382. Сформулируйте XML-конфигурацию для конечного автомата BW, переключающегося между белым и черным цветами по нажатии любой клавиши. Преобразуйте XML-конфигурацию в представление XMachine. Реализацию автомата вы найдете в упражнении 227. ■

Прежде чем углубиться в обсуждение части компонента конфигурации, отвечающей за преобразование конфигурации в вызов главной функции, давайте уточним задачу:

Постановка задачи. Спроектируйте программу, которая принимает конфигурацию XMachine и запускает `simulate`.

Несмотря на то что эта задача специфична для нашего конкретно-го случая, легко можно представить обобщенный вариант для подобных систем, и мы рекомендуем вам это сделать.

Постановка задачи предполагает такую схему:

```
; XMachine -> FSM-State
; запускает запуск симуляции конечного автомата с заданной конфигурацией
(define (simulate-xmachine xm)
  (simulate ... ...))
```

Согласно постановке задачи, наша функция должна определить два аргумента и передать их в вызов `simulate`. Чтобы завершить опре-

деление функции, нам понадобятся два компонента: начальное состояние и таблица переходов. Эти два компонента являются частью xm, поэтому добавим в список желаний соответствующие функции:

- xm-state0 извлекает начальное состояние из заданного XMachine:
| (check-expect (xm-state0 xm0) "red")
- xm->transitions преобразует встроенный список экземпляров X1T в список экземпляров 1Transition:
| (check-expect (xm->transitions xm0) fsm-traffic)

Листинг 86. Интерпретация программы на DSL

```
; XMachine -> FSM-State
; интерпретирует заданную конфигурацию как конечный автомат
(define (simulate-xmachine xm)
  (simulate (xm-state0 xm) (xm->transitions xm)))

; XMachine -> FSM-State
; извлекает и преобразует таблицу переходов из xm0

(check-expect (xm-state0 xm0) "red")

(define (xm-state0 xm0)
  (find-attr (xexpr-attr xm0) 'initial))

; XMachine -> [List-of 1Transition]
; извлекает таблицу переходов из xm

(check-expect (xm->transitions xm0) fsm-traffic)

(define (xm->transitions xm)
  (local (; X1T -> 1Transition
         (define (xaction->action xa)
           (list (find-attr (xexpr-attr xa) 'state)
                 (find-attr (xexpr-attr xa) 'next))))
        (map xaction->action (xexpr-content xm))))
```

Поскольку XMachine является подмножеством Xexpr, определить xm-state0 совсем несложно. Учитывая, что начальное состояние задано как атрибут, функция xm-state0 должна просто извлечь список атрибутов с помощью xexpr-attr, а затем получить значение атрибута 'initial.

Теперь перейдем к функции xm->transitions, преобразующей переходы внутри конфигурации XMachine в таблицу переходов:

```
; XMachine -> [List-of 1Transition]
; извлекает и преобразует таблицу переходов из xm
(define (xm->transitions xm)
  '())
```

Имя функции предполагает ее сигнатуру и заявление о назначении. В нашем заявлении о назначении описывается двухэтапный процесс: (1) извлечь представление Xexpr переходов и (2) преобразовать его в экземпляр [List-of 1Transition].

Для извлечения можно просто использовать функцию `xexpr-content`, которая возвращает список, но для преобразования требуется дополнительный анализ. Если посмотреть на определение данных `XMachine`, то можно заметить, что `Xexpr` – это список экземпляров `X1T`. Сигнатура также говорит нам, что таблица переходов представлена списком экземпляров `1Transition`. Соответственно, каждый элемент в первом списке преобразуется в один элемент во втором, что предполагает использование функции `map`:

```
(define (xm->transitions xm)
  (local ( ; X1T -> 1Transition
    (define (xaction->action xa)
      ...))
    (map xaction->action (xexpr-content xm))))
```

Как видите, мы следуем идеям проектирования из раздела 16.5 и определяем локальную функцию, которую использует `map`. Функция `xaction->action` просто извлекает соответствующие значения из `Xexpr`.

Полное решение показано в листинге 86. Здесь размер компонента, преобразующего конфигурацию на языке DSL в вызов функции, не уступает размеру исходной программы. Но в реальном мире такое соотношение – большая редкость; обычно компонент преобразования DSL составляет лишь малую часть всего продукта, поэтому такой подход пользуется большой популярностью.

Упражнение 383. Запустите код в листинге 86 с конфигурацией конечного автомата BW из упражнения 382. ■

22.4. Чтение XML

Листинг 87. Файл с конфигурацией конечного автомата

```
machine-configuration.xml
<machine initial="red">
  <action state="red" next="green" />
  <action state="green" next="yellow" />
  <action state="yellow" next="red" />
</machine>
```

В этом разделе используются библиотеки `2htdp/batch-io`, `2htdp/universe` и `2htdp/image`.

Системные администраторы ожидают, что сложные приложения будут читать свои конфигурации из файлов или, возможно, откуда-то из сети. Программы на `ISL+` тоже могут получать (некоторую) информацию из файлов XML.

В листинге 88 показан соответствующий отрывок из обучающего пакета. Для согласованности в названиях определений данных в этом листинге используется окончание `.v3` даже в тех определениях данных, для которых нет версии 2:

```
; Xexpr.v3 -- это одно из значений:
; -- Symbol
```

```

; -- String
; -- Number
; -- (cons Symbol (cons Attribute*.v3 [List-of Xexpr.v3]))
; -- (cons Symbol [List-of Xexpr.v3])
;
; Attribute*.v3 -- это [List-of Attribute.v3].
;
; Attribute.v3 -- это список с двумя элементами:
; (list Symbol String)

```

Предположим, что у нас есть файл, представленный в листинге 87. Если подключить библиотеку *2htdp/batch-io*, то программа сможет прочитать его с помощью *read-plain-xexpr*. Эта функция извлекает элемент XML в формате, соответствующем определению данных XMachine. В обучающем пакете также доступна функция извлечения XML-элементов из интернета. Попробуйте выполнить следующий код в DrRacket:

```

> (read-plain-xexpr/web
  (string-append
   "http://www.ccs.neu.edu/"
   "home/matthias/"
   "HtDP2e/Files/machine-configuration.xml"))

```

Листинг 88. Чтение X-выражений

```

; Any -> Boolean
; проверяет, является ли x экземпляром Xexpr.v3
; выводит фрагмент, если x не является экземпляром Xexpr.v3
(define (xexpr? x) ...)

; String -> Xexpr.v3
; возвращает первый элемент XML из файла f
(define (read-xexpr f) ...)

; String -> Boolean
; возвращает #false, если при обращении к данному URL-адресу и
; получен код '404', иначе возвращает #true
(define (url-exists? u) ...)

; String -> [Maybe Xexpr.v3]
; извлекает первый элемент XML (HTML) из URL-адреса u
; возвращает #false, если выполняется условие (not (url-exists? u))
(define (read-plain-xexpr/web u) ...)

; String -> [Maybe Xexpr.v3]
; извлекает первый элемент XML (HTML) из URL-адреса u
; возвращает #false, если выполняется условие (not (url-exists? u))
(define (read-xexpr/web u) ...)

```

Если ваш компьютер подключен к интернету, то это выражение вернет нашу стандартную конфигурацию конечного автомата.

Чтение файлов или веб-страниц вводит в нашу вычислительную модель совершенно новую идею. Как объясняется в интермешо 1, программа на BSL вычисляется точно так же, как вы сами вычисляете выражения с переменными в алгебре. Определения функций тоже

интерпретируются как в алгебре. В школьном курсе алгебры также вводятся определения условных функций, поэтому выражение `cond` не должно вызывать никаких проблем. Наконец, даже притом что в ISL+ функции интерпретируются как значения, модель вычислений от этого практически не меняется.

Одно из важных свойств этой вычислительной модели заключается в том, что независимо от частоты вызова функции `f` для некоторого аргумента (аргументов) `a` ...

```
| (f a ...)
```

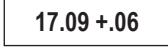
результат остается неизменным. Однако функции `read-file`, `read-xexpr` и подобные им нарушают это свойство. Проблема в том, что файлы и веб-сайты могут меняться со временем, поэтому каждый раз, когда программа читает файл или веб-сайт, она может получать новый результат.

Рассмотрим идею поиска стоимости акций компании. Откройте в браузере `google.com/finance` или любой другой финансовый веб-сайт и введите название компании по своему выбору, например `Ford`. В ответ сайт отобразит текущую стоимость акций этой компании и другие сведения, например насколько изменилась цена с момента последней публикации, текущее время и многие другие факты. Важно отметить, что если повторно открыть эту страницу через день или неделю, то часть информации на ней изменится.

Вместо поиска такой информации о компании вручную можно написать небольшую программу, которая будет извлекать информацию через определенные интервалы времени, скажем через каждые 15 секунд. Вы можете написать такую программу на ISL, запустить ее:

```
| > (stock-alert "Ford")
```

и наблюдать на экране примерно такое изображение:



17.09 +.06

Для разработки подобной программы требуются навыки, выходящие за рамки обычного проектирования программ. Во-первых, нужно изучить особенности форматирования информации, возвращаемой веб-сайтом. Например, страница финансовых услуг Google возвращает сведения в таком формате:

```
<meta content="17.09" itemprop="price" />
<meta content="+0.07" itemprop="priceChange" />
<meta content="0.41" itemprop="priceChangePercent" />
<meta content="2013-08-12T16:59:06Z" itemprop="quoteTime" />
<meta content="NYSE real-time data" itemprop="dataSource" />
```

Если бы у нас была функция, которая способна выполнять поиск в Xexpr.v3 и извлекать (представление XML) элементы `meta` с атрибу-

тами "price" и "priceChange", то остальная часть функции stock-alert выглядела бы очень просто.

В листинге 89 показано ядро программы. Мы предлагаем вам самостоятельно спроектировать функцию `get`, которая основана на анализе взаимосвязанных данных.

Как показано в листинге 89, главная функция определяет две локальные функции: обработчик тактов часов и функцию отображения. Спецификация `big-bang` требует, чтобы такты часов следовали друг за другом с интервалом 15 секунд. С каждым новым тактом часов `ISL+` применяет `retrieve-stock-data` к текущему состоянию мира, которое просто игнорируется. Вместо этого функция получает страницу с веб-сайта вызовом `read-xexpr/web` и извлекает из нее соответствующую информацию с помощью `get`. Таким образом, новое состояние мира формируется на основе информации, полученной из интернета, а не из некоторых локальных данных.

Листинг 89. Извлечение данных из интернета по событиям

```
(define PREFIX "https://www.google.com/finance?q=")
(define SIZE 22) ; font size

(define-struct data [price delta])
; StockWorld -- это структура: (make-data String String)

; String -> StockWorld
; извлекает стоимость акций компании со и ее изменение каждые 15 с
(define (stock-alert co)
  (local ((define url (string-append PREFIX co))
         ; [StockWorld -> StockWorld]
         (define (retrieve-stock-data __w)
           (local ((define x (read-xexpr/web url)))
             (make-data (get x "price")
                       (get x "priceChange"))))
         ; StockWorld -> Image
         (define (render-stock-data w)
           (local (; [StockWorld -> String] -> Image
                  (define (word sel col)
                    (text (sel w) SIZE col))
                  (overlay (beside (word data-price 'black)
                                  (text " " SIZE 'white)
                                  (word data-delta 'red))
                           (rectangle 300 35 'solid 'white))))))
    (big-bang (retrieve-stock-data 'no-use)
              [on-tick retrieve-stock-data 15]
              [to-draw render-stock-data])))
```

Упражнение 384. В листинге 89 упоминается функция `read-xexpr/web`. Посмотрите на ее сигнатуру и заявление о назначении в листинге 88, а затем прочтите документацию, чтобы определить, чем она отличается от «простой» родственной функции.

В листинге 89 также не хватает нескольких важных элементов, в частности во всех локальных функциях отсутствуют интерпретация данных и заявление о назначении. Сформулируйте недостающие элементы, чтобы сделать программу понятнее. ■

Упражнение 385. Отыщите стоимость акций компании по вашему выбору на странице финансовых услуг Google. Если у вас нет особых предпочтений, выберите компанию Ford. Затем сохраните исходный код страницы в файл в своем рабочем каталоге и попробуйте прочитать его вызовом `read-xexpr` в DrRacket, чтобы увидеть, как содержимое файла выглядит в представлении `Xexpr.v3`. ■

Упражнение 386. Вот определение функции `get`:

```
; Xexpr.v3 String -> String
; извлекает значение атрибута "content"
; из элемента 'meta, имеющего атрибут "itemprop" со значением s
(check-expect
  (get '(meta ((content "+1") (itemprop "F")))) "F")
  "+1")

(define (get x s)
  (local ((define result (get-xexpr x s)))
    (if (string? result)
        result
        (error "not found"))))
```

Она предполагает существование функции `get-xexpr`, которая отыскивает требуемый атрибут в произвольном экземпляре `Xexpr.v3` и возвращает [Maybe String].

Сформулируйте тестовые примеры, выполняющие поиск значений, отличных от "F", и вынуждающие `get` сообщить об ошибке.

Спроектируйте `get-xexpr`. Добавьте примеры применения этой функции, взяв за основу примеры для `get`. Обобщите эти примеры и убедитесь, что `get-xexpr` может обрабатывать произвольные экземпляры `Xexpr.v3`. Наконец, сформулируйте тест, использующий веб-данные, которые вы сохранили в упражнении 385. ■

23. Одновременная обработка

Некоторые функции должны принимать два аргумента, принадлежащих классам с нетривиальными определениями данных. Порядок проектирования таких функций зависит от отношений между аргументами. Во-первых, один из аргументов, возможно, придется интерпретировать как атомарное значение. Во-вторых, возможно, что функция должна обрабатывать два аргумента синхронно. Наконец, функция может обрабатывать входные данные с учетом всех возможных случаев. Эта глава иллюстрирует три случая с примерами и описывает дополненный рецепт проектирования. В последнем разделе обсуждается равенство составных данных.

23.1. Одновременная обработка двух списков: случай 1

Рассмотрим следующие сигнатуру, назначение и заголовок:

```
| ; [List-of Number] [List-of Number] -> [List-of Number]
| ; заменяет '() в конце front на end
| (define (replace-eol-with front end)
|   front)
```

Согласно сигнатуре, эта функция принимает два списка. Посмотрим, как применить рецепт проектирования в этом случае.

Начнем с создания примеров. Если первый аргумент – это пустой список '(), то replace-eol-with должна дать в результате второй аргумент, каким бы он ни был:

```
| (check-expect (replace-eol-with '() '(a b)) '(a b))
```

Напротив, если первый аргумент не является пустым списком '(), то, согласно назначению, мы должны заменить '() в конце front на end:

```
| (check-expect (replace-eol-with (cons 1 '()) '(a))
|               (cons 1 '(a)))
| (check-expect (replace-eol-with
|               (cons 2 (cons 1 '())) '(a))
|               (cons 2 (cons 1 '(a))))
```

Назначение и примеры предполагают, что если второй аргумент является списком, то функции не требуется ничего знать о нем, то есть ее макет должен повторять макет функции обработки списков относительно первого аргумента:

```
| (define (replace-eol-with front end)
|   (cond
|     [(empty? front) ...]
|     [else
```

```
| (... (first front) ...
|   ... (replace-eol-with (rest front) end) ...))))
```

Давайте заполним пробелы в макете, следуя пятому шагу рецепта проектирования. Если `front` – это `'()`, то `replace-eol-with` выдает `end`. Если `front` – не `'()`, то мы должны вспомнить, что вычисляют выражения в макете:

- `(first front)` вычисляется в первый элемент списка;
- `(replace-eol-with (rest front) end)` замещает `'()` в конце `(rest front)` значением `end`.

Стоп! Воспользуйтесь табличным методом, чтобы понять, что означают эти пункты в текущем примере.

Теперь осталось сделать совсем небольшой шаг, чтобы получить полное определение:

```
(define (replace-eol-with front end)
  (cond
    [(empty? front) end]
    [else
      (cons (first front)
            (replace-eol-with (rest front) end))))
```

Упражнение 387. Спроектируйте функцию `cross`. Она должна принимать список символов и список чисел и давать все возможные упорядоченные пары символов и чисел. То есть для списков `'(a b c)` и `'(1 2)` ожидается результат `'((a 1) (a 2) (b 1) (b 2) (c 1) (c 2))`. ■

23.2. Одновременная обработка двух списков: случай 2

В разделе 10.1 была представлена функция `wages*`, которая вычисляет недельную заработную плату рабочих с учетом отработанных часов. Она принимает список чисел с отработанными часами за неделю и выдает список чисел с соответствующими заработными платами. Эта задача предполагает, что все сотрудники имеют одинаковую ставку заработной платы, но мы знаем, что даже в небольших компаниях работникам назначаются разные ставки заработной платы.

Рассмотрим более реалистичную версию. Теперь функция принимает два списка: список отработанных часов и список с размерами почасовой оплаты труда. Опираясь на новую постановку задачи, исправим заголовок:

```
; [List-of Number] [List-of Number] -> [List-of Number]
; перемножает соответствующие элементы
; в hours и wages/h
; предполагается, что входные списки имеют одинаковую длину
(define (wages*.v2 hours wages/h)
  '())
```

Добавим примеры:

```
(check-expect (wages*.v2 '()) '())
(check-expect (wages*.v2 (list 5.65) (list 40))
              (list 226.0))
(check-expect (wages*.v2 '(5.65 8.75) '(40.0 30.0))
              '(226.0 262.5))
```

Согласно требованиям, во всех трех примерах используются списки одинаковой длины.

Предположение, касающееся входных данных, тоже можно использовать для разработки макета. В частности, как вытекает из условия, выражение (`empty? hours`) истинно, когда истинно выражение (`empty? wages/h`), кроме того, выражение (`cons? hours`) истинно, когда истинно выражение (`cons? wages/h`). Таким образом, мы вполне можем использовать макет для одного из двух списков:

```
(define (wages*.v2 hours wages/h)
  (cond
    [(empty? hours) ...]
    [else
      (... (first hours)
            ... (first wages/h) ...)
      ... (wages*.v2 (rest hours) (rest wages/h))))
```

В первой ветви `cond` оба аргумента – `hours` и `wages/h` – являются пустыми списками '(), поэтому не требуется использовать никаких селекторов. Во второй ветви `hours` и `wages/h` являются непустыми списками, а это означает, что мы должны использовать селекторы четырежды. Наконец, поскольку два списка имеют одинаковую длину, они оказываются естественными кандидатами для естественной рекурсии `wages*.v2`.

Единственный необычный аспект этого шаблона – рекурсивное применение состоит из двух выражений, оба выражения являются селекторами, формирующими два аргумента. Но эта идея прямо следует из предположения.

Теперь мы без труда можем закончить определение функции:

```
(define (wages*.v2 hours wages/h)
  (cond
    [(empty? hours) '()]
    [else
      (cons
        (weekly-wage (first hours) (first wages/h))
        (wages*.v2 (rest hours) (rest wages/h))))])
```

Из первого примера следует, что результатом первой ветви `cond` будет пустой список '(). Во втором примере мы имеем три значения:

1. (`first hours`) – первое количество часов, отработанных за неделю;
2. (`first wages/h`) – первая почасовая ставка оплаты труда;

3. (`wages*.v2 (rest hours) (rest wages/h)`) – согласно заявлению о назначении, это выражение вычисляет список с недельными заработными платами по оставшимся частям двух списков.

Теперь, чтобы получить окончательный ответ, нужно просто объединить эти значения. Согласно примерам, мы должны вычислить недельную заработную плату для первого работника и построить список из заработной платы этого работника и списка с заработными платами остальных работников:

```
| (cons (weekly-wage (first hours) (first wages/h))
      (wages*.v2 (rest hours) (rest wages/h)))
```

Вспомогательная функция `weekly-wage` принимает количество отработанных часов и почасовую ставку и вычисляет заработную плату за неделю для одного работника:

```
| ; Number Number -> Number
| ; вычисляет недельную заработную плату на основе pay-rate и hours
| (define (weekly-wage pay-rate hours)
|   (* pay-rate hours))
```

Стоп! Какую функцию нужно использовать, чтобы вычислить заработную плату для одного работника? Какую функцию нужно изменить, чтобы удержать подоходный налог?

Упражнение 388. В реальном мире `wages*.v2` принимает списки структур с информацией о работниках и списки с данными об отработанном времени. Структура с информацией о сотруднике содержит имя сотрудника, номер социального страхования и ставку заработной платы. В данных об отработанном времени также указывается имя сотрудника и количество отработанных часов в неделю. Результатом функции является список структур, содержащих имя сотрудника и его недельную заработную плату.

Измените программу в этом разделе так, чтобы она работала с этими более реалистичными версиями данных. Добавьте все необходимые определения структур и данных. Используйте рецепт проектирования для управления процессом изменения программы. ■

Упражнение 389. Спроектируйте функцию `zip`, которая принимает список имен в виде строк и список телефонных номеров, тоже в виде строк. Она должна объединить эти списки одинаковой длины в список записей телефонного справочника:

```
| (define-struct phone-record [name number])
| ; PhoneRecord -- это структура:
| ;   (make-phone-record String String)
```

Исходите из предположения, что соответствующие элементы списков принадлежат одному и тому же человеку. ■

23.3. Одновременная обработка двух списков: случай 3

Вот задача третьего типа:

Задача. Имеются список символов *los* и натуральное число *n*. Требуется спроектировать функцию *list-pick*, которая будет извлекать *n*-й символ из *los*, а если такого символа нет, то сообщать об ошибке.

Вопрос в том, насколько применим рецепт проектирования к *list-pick*.

Определение данных для представления списка символов нам уже хорошо знакомо, но давайте вспомним определение класса натуральных чисел из раздела 9.3:

```
| ; N -- одно из значений:  
| ; -- 0  
| ; -- (add1 N)
```

Теперь перейдем ко второму шагу:

```
| ; [List-of Symbol] N -> Symbol  
| ; извлекает n-й символ из l;  
| ; сообщает об ошибке, если нет символа с таким порядковым номером  
(define (list-pick l n)  
  'a)
```

И список символов, и натуральные числа являются классами со сложными определениями данных. Такое сочетание делает задачу нестандартной, а это означает, что мы должны внимательно исследовать все детали на каждом этапе рецепта проектирования.

На втором шаге мы обычно создаем несколько примеров входных данных и определяем желаемый результат. Начнем с входных значений, с которыми функция должна работать безупречно: '(a b c) и 2. Получив список с тремя символами и индекс 2, функция *list-pick* должна вернуть какой-то символ. Возникает вопрос: какой символ она должна вернуть – 'b или 'c. В начальной школе вы, не задумываясь, отсчитали бы по порядку 1, 2 и выбрали 'b. Но это информатика, а не начальная школа. Здесь люди начинают счет с 0, соответственно, мы должны выбрать 'c. Давайте использовать именно такой подход:

```
| (check-expect (list-pick '(a b c) 2) 'c)
```

Теперь, преодолев этот скользкий момент выбора элемента в списке, рассмотрим фактическую задачу – выбор входных данных. Цель шага по созданию примеров – максимально охватить пространство входных данных. Для этого мы должны выбрать по одному экземпляру входных данных для каждого предложения в описании сложных форм данных. Здесь эта процедура предлагает выбрать, по крайней

мере, два представителя из каждого класса, потому что оба определения данных имеют два предложения. Мы выбираем '`()`' и `(cons 'a '())` для первого аргумента и `0` и `3` для второго. По два варианта значений для каждого аргумента означает, что всего мы должны создать четыре примера; в конце концов, между двумя аргументами нет очевидной связи, как и нет никаких ограничений в сигнатуре.

Как оказывается, только одна из этих пар дает правильный результат; остальные соответствуют отсутствующей позиции в списке:

```
| (check-error (list-pick '() 0) "list too short")
| (check-expect (list-pick (cons 'a '()) 0) 'a)
| (check-error (list-pick '() 3) "list too short")
```

Согласно постановке задачи, функция должна сигнализировать об ошибке, поэтому выберем здесь свое любимое сообщение.

Стоп! Скопируйте эти фрагменты в область определений DrRacket и запустите частично готовую программу.

Как показало обсуждение примеров, в действительности мы имеем четыре независимых случая, которые следует изучить при проектировании функции. Один из способов обнаружить эти случаи – записать все условия из предложений в таблицу:

	(empty? l)	(cons? l)
(= n 0)		
(> n 0)		

По горизонтали в таблице перечислены вопросы, которые должны задаваться в отношении списка; по вертикали – вопросы о натуральных числах. При таком размещении мы получаем четыре ячейки, каждая из которых представляет случай, когда выполняются оба условия – по горизонтали и вертикали.

Наша таблица предполагает, что выражение `cond` для макета функции должно иметь четыре ветви. Мы можем определить подходящее условие для каждого из этих ветвей, задав условия по горизонтали и вертикали для каждой ячейки в таблице:

	(empty? l)	(cons? l)
(= n 0)	(and (empty? l) (= n 0))	(and (cons? l) (= n 0))
(> n 0)	(and (empty? l) (> n 0))	(and (cons? l) (> n 0))

Структура выражения `cond` прямо вытекает из этой таблицы:

```
| (define (list-pick l n)
|   (cond
|     [(and (= n 0) (empty? l)) ...]
|     [(and (> n 0) (empty? l)) ...]
|     [(and (= n 0) (cons? l)) ...]
|     [(and (> n 0) (cons? l)) ...]))
```

Выражение `cond` позволяет нам различить четыре возможных варианта и сосредоточиться на каждом в отдельности. Теперь добавим в каждую ветвь выражения с селекторами:

```
(define (list-pick l n)
  (cond
    [(and (= n 0) (empty? l))
     ...]
    [(and (> n 0) (empty? l))
     (... (sub1 n) ...)]
    [(and (= n 0) (cons? l))
     (... (first l) ... (rest l)...)]
    [(and (> n 0) (cons? l))
     (... (sub1 n) ... (first l) ... (rest l) ...))]))
```

Первый аргумент, `l`, – это список, поэтому предложение `cond` в макете, соответствующее непустому списку, содержит два селектора. Второй аргумент, `n`, принадлежит классу натуральных чисел `N`, поэтому в макете в ветви `cond`, соответствующей числам, отличным от `0`, требуется только один селектор. В тех случаях, когда выполняется условие (`empty? l`) или (`= n 0`), соответствующий аргумент является атомарным, поэтому нет необходимости в использовании селектора.

Последний шаг на пути построения макета требует добавить рекурсивные вызовы, в которых результаты применений селекторов принадлежат тем же классам, что и исходные входные данные. В этом первом примере мы сосредоточимся на последней ветви в `cond`, которая содержит селекторы для обоих аргументов. Однако пока не ясно, как организовать естественную рекурсию. Если не принимать во внимание назначение функции, то возможны три варианта рекурсии:

1. `(list-pick (rest l) (sub1 n))`
2. `(list-pick l (sub1 n))`
3. `(list-pick (rest l) n)`

Каждый представляет возможную комбинацию доступных выражений. Поскольку мы не можем знать, какие из вариантов нам подходят, то переходим к следующему этапу разработки.

Листинг 90. Извлечение элемента списка по индексу

```
; [List-of Symbol] N -> Symbol
; извлекает n-й символ из l;
; сообщает об ошибке, если нет символа с таким порядковым номером
(define (list-pick l n)
  (cond
    [(and (= n 0) (empty? l))
     (error 'list-pick "list too short")]
    [(and (> n 0) (empty? l))
     (error 'list-pick "list too short")]
    [(and (= n 0) (cons? l)) (first l)]
    [(and (> n 0) (cons? l)) (list-pick (rest l) (sub1 n))])))
```

Перейдя к пятому шагу рецепта проектирования, проанализируем каждое предложение `cond` в макете и решим, каким должен быть правильный ответ:

- Если выполняется условие (`and (= n 0) (empty? l)`), то `list-pick` должна выбрать первый символ из пустого списка, что невозможно. Соответственно, правильным ответом будет сообщение об ошибке.
- Если выполняется условие (`and (> n 0) (empty? l)`), то `list-pick` снова должна выбрать символ из пустого списка.
- Если выполняется условие (`and (= n 0) (cons? l)`), то `list-pick` должна выдать первый символ из `l`. Ответом является выражение селектора (`first l`).
- Если выполняется условие (`and (> n 0) (cons? l)`), то мы должны проанализировать, что вычисляют доступные выражения. Как мы уже видели, для этого шага рекомендуется проработать существующий пример. Выберем сокращенный вариант первого примера:

```
| (check-expect (list-pick '(a b) 1) 'b)
```

Вот что вычисляют перечисленные выше три естественные рекурсии с этими значениями:

- (a) (`list-pick '(b) 0`) вернет 'b;
- (b) (`list-pick '(a b) 0`) вернет 'a, неверный ответ;
- (c) (`list-pick '(b) 1`) сообщит об ошибке.

Из этого мы заключаем, что в последней ветви `cond` желаемый ответ вычисляет (`(list-pick (rest l) (sub1 n))`).

Упражнение 390. Спроектируйте функцию `tree-pick`. Она должна принимать дерево символов и список направлений:

```
(define-struct branch [left right])
;
; TOSXE "TOS, определение данных: : : " (Tree Of Symbols -- дерево символов) -- это одно
; из значений:
; -- Symbol
; -- (make-branch TOS TOS)
;
; Direction -- это одно из значений:
; -- 'left
; -- 'right
;
; Список направлений Direction также называется путем.
```

Очевидно, что направление `Direction` сообщает функции, какую ветвь в дереве выбрать – левую или правую. Какой должен быть результат функции `tree-pick?` Не забудьте оформить полную сигнатуру. Функция должна сообщать об ошибке, если задан символ и непустой путь. ■

23.4. Упрощение функций

Функция `list-pick` в листинге 90 намного сложнее, чем необходимо. Первые два условия в выражении `cond` сообщают об ошибке. То есть если выполняется любое из условий:

```
| (and (= n 0) (empty? alos))
```

(где `alos` расшифровывается как «A List Of Symbols» – список символов) или

```
| (and (> n 0) (empty? alos))
```

функция должна сообщить об ошибке. Мы можем преобразовать это наблюдение в код:

```
(define (list-pick alos n)
  (cond
    [(or (and (= n 0) (empty? alos))
          (and (> n 0) (empty? alos)))
     (error 'list-pick "list too short")]
    [(and (= n 0) (cons? alos)) (first alos)]
    [(and (> n 0) (cons? alos))
     (list-pick (rest alos) (sub1 n))]))
```

Чтобы еще больше упростить эту функцию, нам нужно познакомиться с алгебраическими законами, относящимися к логическим значениям:

Эти уравнения известны как закон де Моргана.

```
| (or (and bexp1 a-bexp)  ==  (and (or bexp1 bexp2)
|       (and bexp2 a-bexp))           a-bexp)
```

Тот же закон действует, когда подвыражения в `and` меняются местами. Применение этих законов к `list-pick` дает следующее:

```
(define (list-pick n alos)
  (cond
    [(and (or (= n 0) (> n 0)) (empty? alos))
     (error 'list-pick "list too short")]
    [(and (= n 0) (cons? alos)) (first alos)]
    [(and (> n 0) (cons? alos))
     (list-pick (rest alos) (sub1 n))]))
```

Теперь рассмотрим `(or (= n 0) (> n 0))`. Это выражение всегда #true, потому что `n` принадлежит N. Поскольку выражение `(and #true (empty? alos))` эквивалентно выражению `(empty? alos)`, мы можем снова переписать функцию:

```
(define (list-pick alos n)
  (cond
    [(empty? alos) (error 'list-pick "list too short")]
    [(and (= n 0) (cons? alos)) (first alos)]
    [(and (> n 0) (cons? alos))
     (list-pick (rest alos) (sub1 n))]))
```

Это последнее определение значительно проще определения в листинге 90, но мы можем пойти еще дальше. Сравните первое условие в последней версии `list-pick` со вторым и третьим. Первое предложение в `cond` отфильтровывает все случаи, когда `alos` – пустой список, подвыражение (`cons? alos`) в последних двух предложениях всегда будет вычисляться в `#true`. Если заменить условие на `#true` и снова упростить выражения, то мы получим трехстрочную версию `list-pick`.

Листинг 91. Извлечение элемента списка по индексу, упрощенная версия

```
; list-pick: [List-of Symbol] N[>= 0] -> Symbol
; извлекает n-й символ из alos, счет начинается с 0;
; сообщает об ошибке, если нет символа с таким порядковым номером
(define (list-pick alos n)
  (cond
    [(empty? alos) (error 'list-pick "list too short")]
    [(= n 0) (first alos)]
    [(> n 0) (list-pick (rest alos) (sub1 n))]))
```

В листинге 91 показана упрощенная версия `list-pick`. Она намного проще оригинала, но важно понимать, что оригинал мы спроектировали, используя системный подход, а затем смогли упростить его с помощью хорошо известных алгебраических законов. Поэтому мы можем доверять этой упрощенной версии. Если попытаться написать упрощенную версию сразу, то в какой-то момент мы бы не справились с ситуацией в нашем анализе и наверняка создали некорректную программу.

Упражнение 391. Спроектируйте функцию `replace-eol-with`, используя стратегию из раздела 23.3. Начните с определения тестов и затем системно упростите результат. ■

Упражнение 392. Упростите функцию `tree-pick` из упражнения 390. ■

23.5. Проектирование функций с двумя сложными аргументами

Правильный подход к проектированию функций с двумя (или более) сложными аргументами – следовать общему рецепту. Вы должны проанализировать данные и определить соответствующие классы. Если вас смущает использование параметрических определений, таких как `List-of`, и сокращенных примеров, таких как `'(1 b &)`, то разверните их, явно используя конструкторы. Далее нужно определить сигнатуру функции и заявление о назначении. На этом этапе можно подумать и решить, с какой из следующих трех ситуаций вы столкнулись:

- 1) если один из параметров играет доминирующую роль, рассматривайте другой как атомарный элемент данных;
- 2) в некоторых случаях параметры принадлежат к одному классу значений и должны иметь одинаковый размер. Например, два

списка должны иметь одинаковую длину, или две веб-страницы должны иметь одинаковую длину, и если одна из них содержит встроенную страницу, другая тоже должна ее содержать. Если два параметра имеют равный статус и назначение функции предполагает синхронную их обработку, то выбирайте один из параметров как основной и организуйте функцию вокруг него, обрабатывая другой параллельно;

- 3) если очевидной связи между двумя параметрами нет, необходимо проанализировать все возможные случаи на примерах. Затем используйте результаты анализа для разработки шаблона, особенно рекурсивных частей.

Если вы решите, что функция относится к третьей категории, составьте таблицу, чтобы убедиться, что ни один из вариантов не остался неохваченным. Давайте снова воспользуемся парой нетривиальных определений данных, чтобы объяснить эту идею:

<pre> ; LOD (List Of Directions) -- это одно из значений: ; -- '() ; -- (cons Direction LOD)</pre>	<pre> ; TID -- это одно из значений: ; -- Symbol ; -- (make-binary TID TID) ; -- (make-with TID Symbol TID)</pre>
---	--

Определение данных слева – это обычное определение списка; определение справа – это вариант определения TOS (Tree Of Symbols – дерево символов) с тремя предложениями. В нем используются два определения структур:

<pre> (define-struct with [lft info rght]) (define-struct binary [lft rght])</pre>

Если учесть, что функция принимает LOD и TID, у вас должна получиться таблица такого вида:

	(empty? l)	(cons? l)
(symbol? t)		
(binary? t)		
(with? t)		

По горизонтали перечислены условия, распознающие подклассы для первого параметра (в данном случае LOD), а по вертикали – условия для второго параметра (TID).

Таблица служит руководством для разработки и примеров, и макета функции. Как уже говорилось, примеры должны охватывать все возможные случаи; то есть в каждой ячейке таблицы должен присутствовать хотя бы один пример. Точно так же для каждой ячейки в выражении `cond` в макете должно быть отдельное условие, объединяющее с помощью `and` условие в заголовке столбца с условием в заголовке строки. Каждое предложение в `cond`, в свою очередь, должно содержать все возможные выражения селекторов для обоих параметров. Если один из параметров является атомарным, то добавлять выра-

жения селекторов для него не нужно. Наконец, необходимо рассмотреть все возможные варианты естественной рекурсии. В общем случае кандидатами являются все комбинации выражений селекторов (и, возможно, атомарных аргументов). Поскольку мы не можем знать, какие из них необходимы, а какие нет, мы должны рассмотреть их все на этапе написания кода.

В заключение можно сказать, что проектирование функций нескольких аргументов – это просто вариация старого рецепта проектирования. Ключевая идея – преобразовать определения данных в таблицу, перечисляющую все возможные и интересные комбинации. Этую таблицу следует использовать при разработке примеров и макета.

23.6. Практические упражнения: два аргумента

Упражнение 393. В табл. 17 представлены два определения данных для конечных множеств. Спроектируйте функцию `union` для представления конечных множеств по вашему выбору. Она должна принимать два множества и выдавать объединенное множество, содержащее элементы из двух исходных множеств.

Спроектируйте функцию `intersect` для того же представления множества. Она должна принимать два множества и выдавать пересечение двух исходных множеств, то есть множество, включающее элементы, присутствующие в обоих исходных множествах. ■

Упражнение 394. Спроектируйте функцию `merge`. Она должна принимать два списка чисел, отсортированных по возрастанию, и выдавать один отсортированный список чисел, содержащий все числа из обоих списков. Каждое число в получившемся списке должно встречаться столько раз, сколько оно встречается в двух исходных списках вместе. ■

Упражнение 395. Спроектируйте функцию `take`. Она должна принимать список `l` и натуральное число `n` и выдавать первые `n` элементов из `l` или все элементы из `l`, если `n` больше длины списка.

Спроектируйте функцию `drop`. Она должна принимать список `l` и натуральное число `n` и возвращать `l` без первых `n` элементов или просто `'()`, если `n` больше длины списка. ■

Упражнение 396. «Виселица» – известная игра в угадывание слов. Один игрок выбирает слово и говорит другому игроку, сколько в нем букв. Второй игрок выбирает букву и спрашивает первого игрока, встречается ли эта буква в выбранном слове и в какой позиции. Игра заканчивается по истечении некоторого времени или после определенного количества попыток угадывания.

В листинге 92 представлена версия игры, использующая ограничение по времени. См. раздел 16.3, где объясняется, почему `checked-compare` определяется локально.

Спроектируйте центральную функцию compare-word. Она должна принимать слово, которое нужно отгадать, слово s с уже угаданными буквами и текущую букву, предложенную вторым игроком. Функция должна выдать s с символами "_" в позициях, в которых буквы еще не угаданы.

Листинг 92. Простая игра «Виселица»

```
; HM-Word -- это [List-of Letter или "_"]
; интерпретация: "_" представляет пока не угаданные буквы

; HM-Word N -> String
; запускает простую игру "Виселица", возвращает текущее состояние
(define (play the-pick time-limit)
  (local ((define the-word (explode the-pick))
          (define the-guess (make-list (length the-word) "_"))
          ; HM-Word -> HM-Word
          (define (do-nothing s) s)
          ; HM-Word KeyEvent -> HM-Word
          (define (checked-compare current-status ke)
            (if (member? ke LETTERS)
                (compare-word the-word current-status ke)
                current-status)))
    (implode
      (big-bang the-guess ; HM-Word
                 [to-draw render-word]
                 [on-tick do-nothing 1 time-limit]
                 [on-key checked-compare])))

; HM-Word -> Image
(define (render-word w)
  (text (implode w) 22 "black"))
```

Закончив проектирование функции, запустите программу, как показано ниже:

```
(define LOCATION "/usr/share/dict/words") ; в OS X
(define AS-LIST (read-lines LOCATION))
(define SIZE (length AS-LIST))
(play (list-ref AS-LIST (random SIZE)) 10)
```

Дополнительные пояснения ищите в листинге 39. Поиграйте в эту игру и доработайте ее, если появится такое желание! ■

Упражнение 397. Работники, трудящиеся на фабрике, отмечаются в журнале учета рабочего времени по прибытии утром и перед уходом вечером. Записи учета рабочего времени содержат табельный номер сотрудника и фиксируют количество отработанных часов в неделю. В отделе кадров хранятся записи о сотрудниках с их именами, табельными номерами и ставками заработной платы.

Спроектируйте функцию wages*.v3. Она должна принимать список записей о сотрудниках из отдела кадров и список записей из журнала учета рабочего времени и выдавать список записей (зарплатную ведомость), каждый из которых содержит имя и недельную заработную плату сотрудника. Функция должна сообщать об ошибке, если она найдет запись о сотруднике в одном списке, но не найдет соответствующей записи в другом списке.

Допущение. Каждому табельному номеру сотрудника соответствует не более одной записи в журнале учета рабочего времени. ■

Упражнение 398. Линейная комбинация – это сумма линейных членов, то есть сумма произведений переменных и чисел. Числа в этом контексте называются коэффициентами. Вот некоторые примеры:

$$5 \cdot x \quad 5 \cdot x + 17 \cdot y \quad 5 \cdot x + 17 \cdot y + 3 \cdot z$$

Во всех примерах при x используется коэффициент 5, при y – коэффициент 17 и при z – коэффициент 3.

Если значения переменных известны, то мы сможем определить значение полинома. Например, если $x = 10$, то $5 \cdot x = 50$; если $x = 10$ и $y = 1$, то $5 \cdot x + 17 \cdot y = 67$; и если $x = 10$, $y = 1$ и $z = 2$, то $5 \cdot x + 17 \cdot y + 3 \cdot z = 73$.

Представить линейные комбинации можно множеством способов. Мы могли бы представить их, например, с помощью функций или списка коэффициентов. В последнем случае комбинации, показанные выше, будут иметь такой вид:

```
(list 5)
(list 5 17)
(list 5 17 3)
```

Подобный способ представления предполагает фиксированный порядок следования переменных.

Спроектируйте функцию `value`. Она должна принимать два списка одинаковой длины – линейную комбинацию и список значений переменных – и возвращать результат комбинации этих значений. ■

Упражнение 399. Луиза, Джейн, Лаура, Дана и Мэри решают пройти лотерею, чтобы сделать подарки друг другу. Поскольку Джейн работает программистом, подруги попросили ее написать программу, которая беспристрастно решает эту задачу. Разумеется, программа не должна выбирать получателем самого дарителя.

Вот основная часть программы, которую написала Джейн:

```
; [List-of String] -> [List-of String]
; случайно выбирает одну из перестановок имен, в которых нет ни одного
; совпадения с именами в соответствующих позициях в исходном списке
(define (gift-pick names)
  (random-pick
    (non-same names (arrangements names)))))

; [List-of String] -> [List-of [List-of String]]
; возвращает все возможные перестановки имен
; см. упражнение 213
(define (arrangements names)
  ...)
```

Она принимает список имен и случайно выбирает одну из перестановок, в которых нет ни одного совпадения с именами в соответствующих позициях в исходном списке.

Ваша задача – спроектировать две вспомогательные функции:

```
; [NList-of X] -> X
; возвращает случайный элемент списка
(define (random-pick l)
  (first l))

; [List-of String] [List-of [List-of String]]
; ->
; [List-of [List-of String]]
; возвращает список из списков в ll, в котором нет ни одного совпадения
; с именами в соответствующих позициях в исходном списке
(define (non-same names ll)
  ll)
```

Напомним, что `random` выбирает случайное число; см. упражнение 99. ■

Упражнение 400. Спроектируйте функцию `DNArefix`. Она должна принимать два аргумента – списки символов 'a, 'c, 'g и 't, которые встречаются в описаниях ДНК (DNA). Первый список называется шаблоном, а второй – цепочкой для поиска. Функция должна возвращать `#true`, если шаблон идентичен начальной части цепочки для поиска; в противном случае – `#false`.

Спроектируйте также функцию `DNAdelta`. Она похожа на `DNArefix`, но возвращает первый элемент в цепочке для поиска, следующий за шаблоном. Если списки идентичны и за шаблоном не следует никакая буква ДНК, то функция должна сообщить об ошибке. Если шаблон не соответствует началу цепочки для поиска, то функция должна вернуть `#false`. Функция не должна выполнять обход элементов списков более одного раза.

Можно ли упростить `DNArefix` или `DNAdelta`? ■

Упражнение 401. Спроектируйте функцию `sexp=?`. Она должна определять равенство двух S-выражений. Ниже для удобства приводится определение данных в сжатом виде:

```
; S-expr (S-выражение) -- это одно из значений:
; -- Atom
; -- [List-of S-expr]
;
; Atom -- Это одно из значений:
; -- Number
; -- String
; -- Symbol
```

Всякий раз, когда вы используете проверку `check-expect`, она применяет функцию, напоминающую `sexp=?`, чтобы проверить равенство двух произвольных значений. Если проверка не увенчалась успехом, то `check-expect` сообщает об этом. ■

Упражнение 402. Прочитайте еще раз упражнение 354. Объясните, почему мы предложили сначала подумать о данном выражении как об атомарном значении. ■

23.7. Проект: база данных

В этом разделе объединены знания из всех четырех частей книги. Многие приложения используют базы данных для хранения данных. Не вдаваясь в подробности, база данных – это таблица, организованная с учетом четко определенных правил. Таблицу называют *содержимым*, а правила – *схемой*. В табл. 22 показаны два примера. Каждая таблица состоит из двух частей: схемы (в строке заголовка) и содержимого.

Таблица 22. Базы данных как таблицы

Name String	Age Integer	Present Boolean	Present Boolean	Description String
"Alice"	35	#true	#true	"presence"
"Bob"	25	#false	#false	"absence"
"Carol"	30	#true		
"Dave"	32	#false		

Остановимся на таблице слева. Она состоит из трех столбцов и четырех строк – записей. Для каждого столбца определено правило, состоящее из двух частей:

- Правило в крайнем левом столбце гласит, что столбец имеет имя «Name» (имя) и каждый элемент данных в этом столбце является строкой (String).
- Средний столбец имеет имя «Age» (возраст) и содержит целые числа.
- Самый правый столбец имеет имя «Present» (присутствует) и содержит логические значения.

Стоп! Дайте аналогичные описания правил для таблицы справа.

Специалисты по информатике рассуждают о таких таблицах как об *отношениях*. Схема вводит терминологию для обозначения столбцов отношения и отдельных ячеек в записи. Каждая запись связывает фиксированное количество значений; а совокупность всех записей составляет полное отношение. Согласно этой терминологии, первая запись в левой таблице в табл. 22 связывает "Alice" со значениями 35 и #true. Кроме того, первая ячейка в записи называется полем «Name», вторая – полем «Age», и третья – полем «Present».

В этом разделе мы попробуем представить базы данных через структуры и списки:

```
(define-struct db [schema content])
; DB -- это структура: (make-db Schema Content)
; Schema -- это [List-of Spec]
; Spec -- это [List Label Predicate]
; Label -- это String
; Predicate -- это [Any -> Boolean]

; Content (возможно, часть содержимого) -- это [List-of Row]
```

```

; Row -- это [List-of Cell]
; Cell -- это Any
; ограничение: ячейки не содержат функций

; ограничение целостности: В (make-db sch con)
; каждая запись в содержимом con
; (I1) имеет одну и ту же длину, как определено схемой sch
; (I2) i-я ячейка Cell удовлетворяет i-му предикату Predicate в схеме sch

```

Стоп! Преобразуйте базу данных в табл. 22 в выбранное представление данных. Обратите внимание, что содержимое таблиц уже использует данные ISL+.

В табл. 23 показано, как представить две таблицы в табл. 22 в виде DB. Слева представлена схема, содержимое и база данных из левого столбца в табл. 22; справа – из правого столбца. Для простоты в примерах используются обозначения в форме quasiquote и unquote. Напомним, что такой формат позволяет включать в цитируемые списки такие значения, как boolean?. Если вам неудобно использовать эти обозначения, представьте данные примеры в формате list.

Упражнение 403. Спецификация объединяет Label и Predicate в список. Такой способ допустим, но он нарушает наши правила использования структурных типов для фиксированного количества единиц информации.

Таблица 23. Базы данных в виде данных ISL+

<pre> (define school-schema `(("Name" ,string?) ("Age" ,integer?) ("Present" ,boolean?))) (define school-content `(("Alice" 35 #true) ("Bob" 25 #false) ("Carol" 30 #true) ("Dave" 32 #false))) (define school-db (make-db school-schema school-content)) </pre>	<pre> (define presence-schema `(("Present" ,boolean?) ("Description" ,string?))) (define presence-content `(#true "presence") (#false "absence"))) (define presence-db (make-db presence-schema presence-content)) </pre>
--	---

Бот альтернативное представление данных:

```

(define-struct spec [label predicate])
; Spec -- это структура: (make-spec Label Predicate)

```

Используйте это альтернативное определение для представления схем баз данных в табл. 22. ■

Проверка целостности. Использование баз данных критически зависит от их целостности. В данном случае под «целостностью» подразумеваются ограничения (I1) и (I2) из определения данных. Очевидно, что проверка целостности базы данных является задачей функции:

```
| ; DB -> Boolean
| ; проверяет, все ли записи в db удовлетворяют условиям (I1) и (I2)

| (check-expect (integrity-check school-db) #true)
| (check-expect (integrity-check presence-db) #true)

| (define (integrity-check db)
|   #false)
```

Формулировка двух ограничений предполагает, что некоторая функция должна давать `#true` для каждой записи в базе данных. Выражение этой идеи в коде требует применения `andmap` к содержимому `db`:

```
| (define (integrity-check db)
|   (local (; Row -> Boolean
|         (define (row-integrity-check row)
|           ...))
|     (andmap row-integrity-check (db-content db))))
```

Следуя рецепту проектирования с использованием существующих абстракций, макет вводит локальное определение вспомогательной функции.

Проектирование `row-integrity-check` начинается с определения заголовка:

```
| ; Row -> Boolean
| ; проверяет, удовлетворяет ли запись row ограничениям (I1) и (I2)
| (define (row-integrity-check row)
|   #false)
```

Как обычно, мы описываем назначение, для того чтобы понять задачу. Здесь говорится, что функция проверяет два условия. Когда задача состоит из двух подзадач, рецепт проектирования требует определить отдельную функцию для каждой подзадачи и объединить их результаты:

```
| (and (length-of-row-check row)
|      (check-every-cell row))
```

Добавьте эти функции в список желаний; их имена наглядно определяют их назначение.

Прежде чем приступить к проектированию этих функций, мы должны подумать, можно ли использовать существующие элементарные операции для вычисления желаемого значения. Например, мы знаем, что `(length row)` подсчитывает количество полей в записи. Если двигаться в этом направлении, то нам определенно нужно:

```
| (= (length row) (length (db-schema db)))
```

Это условие проверяет равенство длины записи `row` и схемы в `db`.

Точно так же в `check-every-cell` мы должны проверить, что некоторая функция дает `#true` для каждого поля в записи. И снова похоже, что мы можем использовать `andmap`:

```
| (andmap cell-integrity-check row)
```

Очевидно, что целью `cell-integrity-check` является проверка ограничения (I2), то есть:

«удовлетворяет ли i -я ячейка i -му предикату в схеме `db`».

И тут нас поджидает первая сложность, потому что в этом заявлении о назначении упоминается относительное положение поля в записи. Однако цель `andmap` состоит в том, чтобы выделение применить `cell-integrity-check` к каждой ячейке.

Всякий раз, застопорившись на чем-то, вы должны разработать примеры. Для вспомогательных или локальных функций лучше всего заимствовать примеры для основной функции. Первый пример для `integrity-check` проверяет соответствие `school-content` ограничениям целостности. Очевидно, что все записи в `school-content` имеют ту же длину, что и `school-schema`. Возникает вопрос, почему такая строка, как

```
| (list "Alice" 35 #true)
```

удовлетворяет предикатам в соответствующей схеме:

```
(list (list "Name" string?)  
      (list "Age" integer?)  
      (list "Present" boolean?))
```

Ответ прост: потому что все три предиката истины для соответствующих полей:

```
> (string? "Alice")  
#true  
> (integer? 35)  
#true  
> (boolean? #true)  
#true
```

Осталось сделать лишь короткий шаг, чтобы заметить, что функция должна обрабатывать эти два списка – схему базы данных и данную запись – параллельно.

Упражнение 404. Спроектируйте функцию `andmap2`. Она должна принимать логическую функцию f двух аргументов и два списка одинаковой длины. Ее результатом является логическое значение. В частности, она должна применять f к парам соответствующих значений из двух списков, и если для всех пар f дает `#true`, то и `andmap2` должна давать `#true`. В противном случае `andmap2` должна выдавать `#false`. Проще говоря, `andmap2` похожа на `andmap`, но обрабатывает два списка. ■

Стоп! Прежде чем продолжить чтение, решите упражнение 404.

Если бы у нас была функция `andmap2`, то проверка второго условия выглядела бы просто:

```
(andmap2 (lambda (s c) [(second s) c])  
        (db-schema db)  
        row)
```

Данная функция принимает `Spec s` из схемы `db`, извлекает второй предикат и применяет его к заданной `Cell c`. Значение, возвращаемое

предикатом, становится результатом лямбда-выражения. Стоп! Объясните `[((second s) c)]`.

Как оказывается, функция `andmap` в ISL+ может действовать подобно `andmap2`:

```
(define (integrity-check db)
  (local (; Row -> Boolean
          ; проверяет, удовлетворяет ли запись ограничениям (I1) и (I2)
          (define (row-integrity-check row)
            (and (= (length row)
                   (length (db-schema db)))
                 (andmap (lambda (s c) [((second s) c)])
                         (db-schema db)
                         row))))
        (andmap row-integrity-check (db-content db))))
```

Стоп! Разработайте тест, вынуждающий `integrity-check` показать ложный результат.

Замечание об эффекте подъема выражений. Наше определение `integrity-check` страдает от нескольких проблем; часть из них видна сразу, а часть – невидима с первого взгляда. Ясно видно, что функция дважды извлекает схему `db`. В существующее локальное определение можно ввести определение константы и избежать этого дублирования:

```
(define (integrity-check.v2 db)
  (local ((define schema (db-schema db))
          ; Row -> Boolean
          ; проверяет, удовлетворяет ли запись ограничениям (I1) и (I2)
          (define (row-integrity-check row)
            (and (= (length row) (length schema))
                 (andmap (lambda (s c) [((second s) c)])
                         schema
                         row))))
        (andmap row-integrity-check (db-content db))))
```

Из раздела 16.2 мы знаем, что такой прием может сократить время, необходимое для проверки целостности. Так же как определение `inf` в листинге 61, исходная версия `integrity-check` извлекает схему `db` для каждой отдельной записи, даже притом что она явно остается неизменной.

ТЕРМИНОЛОГИЯ. Специалисты по информатике называют это «подъемом выражения». Точно так же функция `row-integrity-check` в каждом вызове определяет длину схемы в `db` и всегда получает один и тот же результат.

Листинг 93. Результат систематического подъема выражений

```
(define (integrity-check.v3 db)
  (local ((define schema (db-schema db))
          (define content (db-content db))
          (define width (length schema))
          ; Row -> Boolean
```

```
; проверяет, удовлетворяет ли запись ограничениям (I1) и (I2)
(define (row-integrity-check row)
  (and (= (length row) width)
       (andmap (lambda (s c) [(second s) c])
              schema
              row))))
(andmap row-integrity-check content)))
```

Следовательно, чтобы повысить производительность этой функции, мы можем использовать локальное определение для извлечения информации о базе данных только один раз и сохранения ее в константах. Результат подъема (`(length schema)`) из `row-integrity-check` показан в листинге 93. Для удобства в эту окончательную версию также было добавлено определение, сохраняющее поле `content` из db. **КОНЕЦ**.

Проекции и выборки. Программы, работающие с базами данных, должны извлекать из них их содержимое. Один из способов извлечения – *выборка* (selection) – был описан в разделе 12.2. Другой способ заключается в извлечении сокращенного набора данных; его окрестили *проекцией* (projection). Более конкретно: проекция создает копию базы данных, сохраняя только определенные столбцы.

Описание проекции предполагает следующее:

```
; DB [List-of Label] -> DB
; сохраняет столбцы из db с метками (именами), присутствующими в списке labels
(define (project db labels) (make-db '() '()))
```

Учитывая сложность проекции, в первую очередь нужно разработать пример. Допустим, нам нужно получить содержимое базы данных из табл. 22 слева без столбца Age. Вот как это преобразование выглядит в терминах таблиц:

исходная база данных			...без столбца «Age»	
Name String	Age Integer	Present Boolean	Name String	Present Boolean
"Alice"	35	#true	"Alice"	#true
"Bob"	25	#false	"Bob"	#false
"Carol"	30	#true	"Carol"	#true
"Dave"	32	#false	"Dave"	#false

Для преобразования этого примера в тест воспользуемся табл. 23:

```
(define projected-content
  `(("Alice" #true)
    ("Bob" #false)
    ("Carol" #true)
    ("Dave" #false)))

(define projected-schema
  `(("Name" ,string?) ("Present" ,boolean?)))

(define projected-db
  (make-db projected-schema projected-content))
```

```
; Стоп! Внимательно прочтайте этот тест. Что здесь не так?
(check-expect (project school-db '("Name" "Present"))
               projected-db)
```

Если запустить этот код в DrRacket, то он выведет сообщение об ошибке:

```
first argument of equality cannot be a function
(первый аргумент в сравнении не может быть функцией)
```

еще до того, как DrRacket сможет оценить тест. Как уже говорилось в разделе 14.4, функции – это бесконечно большие объекты, и невозможно гарантировать получение одинаковых результатов при применении двух разных функций к одним и тем же аргументам. Поэтому мы ослабим тестовый пример:

```
(check-expect
  (db-content (project school-db '("Name" "Present")))
  projected-content)
```

В макете мы снова используем существующие абстракции, как показано в листинге 94. Выражение `local` определяет две функции: одну для использования в вызове `filter`, чтобы сузить схему исходной базы данных, а другую для использования в вызове `map`, чтобы сократить содержимое. Кроме того, функция снова извлекает схему и содержимое из исходной базы данных в константы.

Листинг 94. Макет функции project

```
(define (project db labels)
  (local ((define schema (db-schema db))
          (define content (db-content db))
          ; Spec -> Boolean
          ; определяет принадлежность данного поля новой схеме
          (define (keep? c) ...))
         ; Row -> Row
         ; сохраняет поля с метками, присутствующими в списке labels
         (define (row-project row) ...))
    (make-db (filter keep? schema)
             (map row-project content)))
```

Прежде чем перейти к списку желаний, сделаем шаг назад и изучим решение с повторным использованием двух существующих абстракций. Согласно сигнатуре, функция принимает структуру и создает экземпляр `db`, поэтому следующее выражение явно необходимо:

```
(local ((define schema (db-schema db))
        (define content (db-content db)))
      (make-db ... schema ...
                  ... content ...))
```

Также несложно заметить, что новая схема создается из старой схемы, а новое содержимое – из старого содержимого. Кроме того, в описании назначения функции `project` явно говорится, что функция

должна сохранить только метки, упомянутые во втором аргументе. Следовательно, функция `filter` правильно сужает данную схему. Записи, напротив, остаются на месте, просто каждая из них теряет несколько полей. Отсюда можно заключить, что для обработки `content` следует использовать `map`.

Теперь перейдем к проектированию двух вспомогательных функций. Определение `keep?` выглядит просто:

```
| ; Spec -> Boolean
| ; определяет принадлежность данного поля новой схеме
| (define (keep? c)
|   (member? (first c) labels))
```

Функция применяется к структуре `Spec`, объединяющей `Label` и `Predicate` в список. Если первый элемент присутствует в списке `labels`, то данный экземпляр `Spec` сохраняется.

При проектировании `row-project` наша цель состоит в том, чтобы сохранить в каждой записи из `content` те поля, имена которых присутствуют в списке `labels`. Давайте рассмотрим приведенный выше пример. Вот четыре записи:

```
| (list "Alice" 35 #true)
| (list "Bob" 25 #false)
| (list "Carol" 30 #true)
| (list "Dave" 32 #false)
```

Все они соответствуют схеме `school-schema`:

```
| (list "Name" "Age" "Present")
```

Имена в схеме определяют имена полей в данных записях. Следовательно, `row-project` должна оставить в каждой записи первое и третье поля, потому что это их имена имеются в заданном списке `labels`.

Класс `Row` определяется рекурсивно, поэтому процесс сопоставления содержимого `Cell` с их именами требует использования рекурсивной вспомогательной функции, которую `row-project` может применить к содержимому и меткам полей. Добавим это желание в список:

```
| ; Row [List-of Label] -> Row
| ; оставляет в записи поля, которые соответствуют элементам
| ; в списке names
| (define (row-filter row names) '())
```

После добавления этого желания `row-project` превращается в односстрочную функцию:

```
| (define (row-project row)
|   (row-filter row (map first schema)))
```

Выражение `map` извлекает имена ячеек, которые затем передаются функции `row-filter` для извлечения совпадших полей.

Упражнение 405. Спроектируйте функцию `row-filter`. Создайте примеры для `row-filter`, взяв за основу примеры для `project`.

Допущение. Данная база данных проходит проверку целостности – в том смысле, что каждая запись соответствует схеме и, следовательно, содержит поля с именами, указанными в схеме. ■

Листинг 95. Получение проекции базы данных

```
(define (project.v1 db labels)
  (local ((define schema (db-schema db))
          (define content (db-content db))

          ; Spec -> Boolean
          ; определяет принадлежность данного поля новой схеме
          (define (keep? c)
            (member? (first c) labels))

          ; Row -> Row
          ; сохраняет поля с метками, присутствующими в списке labels
          (define (row-project row)
            (row-filter row (map first schema)))

          ; Row [List-of Label] -> Row
          ; оставляет в записи поля, которые соответствуют элементам
          ; в списке names
          (define (row-filter row names)
            (cond
              [(empty? names) '()]
              [else
                (if (member? (first names) labels)
                    (cons (first row)
                          (row-filter (rest row) (rest names)))
                    (row-filter (rest row) (rest names))))]))
          (make-db (filter keep? schema)
                  (map row-project content)))))

| (member? label labels)
```

В листинге 95 приводятся законченные определения всех функций. К имени функции `project` добавлено окончание `.v1`, потому что ее желательно улучшить. Следующие упражнения предложат вам реализовать некоторые из них.

Упражнение 406. Функция `row-project` повторно вычисляет метки для каждой записи в базе данных. Отличается ли результат этих вычислений от вызова к вызову функции? Если нет, то выполните подъем выражения. ■

Упражнение 407. Повторно спроектируйте функцию `row-filter` и используйте в ней `foldr`. После этого вы сможете объединить `row-project` и `row-filter` в одну функцию. **Подсказка.** Функция `foldr` в ISL+ может принимать два списка и обрабатывать их параллельно. ■

И последнее наблюдение: `row-project` проверяет присутствие метки в списке `labels` для каждого отдельного поля. Для одного и того же поля в разных записях результат будет одинаковым. Поэтому есть смысл вынести эти вычисления из функции.

Эта форма подъема выражений несколько сложнее. В данном случае требуется предварительно вычислить результат

```
| (member? label labels)
```

сразу для всех записей и передать его в функцию вместо списка меток. То есть мы должны заменить список меток списком логических значений, указывающих на необходимость сохранения поля в соответствующей позиции. К счастью, для вычисления такого списка достаточно применить уже имеющуюся функцию `keep?` к схеме:

```
| (map keep? schema)
```

Вместо того чтобы сохранять одни экземпляры Spec из данной схемы `schema` и отбрасывать другие, это выражение просто собирает решения.

В листинге 96 приводится окончательная версия функции `project` и решения для предыдущих упражнений. В нем используется выражение `local` для извлечения `schema` и содержимого `content`, а также определяется функция `keep?` для проверки необходимости оставить то или иное поле. Остальные два определения вводят функцию `mask`, которая формирует список логических значений, как обсуждалось выше, и исправленную версию `row-project`. Последняя использует `foldr` для параллельной обработки данной записи `row` строки и маски `mask`.

Сравните это исправленное определение `project` с `project.v1` в листинге 95. Окончательное определение проще и быстрее первоначальной версии. Системное проектирование в сочетании с тщательной оптимизацией окупается сторицей, а наборы тестов гарантируют, что исправления не нарушают функциональность программы.

Упражнение 408. Спроектируйте функцию `select`. Она должна принимать базу данных `db`, список `lol` (List Of Labels – список меток) экземпляров `Label` и предикат и возвращать список записей, удовлетворяющих заданному предикату, спроектированный до заданного набора меток. ■

Упражнение 409. Спроектируйте функцию `reorder`. Она должна принимать базу данных `db` и список `lol` экземпляров `Label` и создавать базу данных, подобную `db`, но со столбцами, упорядоченными в соответствии со списком `lol`. **Подсказка.** Прочтите описание функции `list-ref`¹.

Для начала предположите, что список `lol` включает все метки столбцов, имеющихся в базе данных `db`. Закончив проектирование, внимательно посмотрите, что нужно изменить, если `lol` будет содержать меньше меток, чем имеется столбцов, а также строки, не являющиеся метками столбцов в `db`. ■

Листинг 96. Получение проекции базы данных

```
(define (project db labels)
  (local ((define schema (db-schema db))
          (define content (db-content db)))
```

¹ https://docs.racket-lang.org/htdp-langs/intermediate-lam.html#%28def._htdp-intermediate-lambda._%28%28lib._lang%2Fhtdp-intermediate-lambda..rkt%29._list-ref%29%29. – Прим. ред.

```

; Spec -> Boolean
; проверяет принадлежность столбца новой схеме
(define (keep? c)
  (member? (first c) labels))

; Row -> Row
; оставляет в записи поля, которые соответствуют элементам
; в списке labels
(define (row-project row)
  (foldr (lambda (cell m c) (if m (cons cell c) c))
    '()
    row
    mask))
(define mask (map keep? schema))
(make-db (filter keep? schema)
  (map row-project content)))

```

Упражнение 410. Спроектируйте функцию `db-union`, которая принимает две базы данных с одинаковой схемой и создает новую базу данных с той же схемой и объединенным содержимым. Функция не должна дублировать записи с одинаковым содержимым.

Исходите из предположения, что схемы содержат согласованные предикаты для всех столбцов. ■

Упражнение 411. Спроектируйте функцию `join`. Она должна принимать две базы данных: `db-1` и `db-2`. Схема для `db-2` начинается с той же спецификации `Spec`, на которой заканчивается схема для `db-1`. Функция должна создать базу данных из `db-1`, заменив последнее поле в каждой записи полями *трансляции* в `db-2`.

Вот пример. Возьмем базы данных из табл. 22. Обе они удовлетворяют требованиям этого упражнения, то есть последнее поле в первой схеме соответствует первому полю во второй схеме. Следовательно, их можно соединить:

Name String	Age Integer	Description String
"Alice"	35	"presence"
"Bob"	25	"absence"
"Carol"	30	"presence"
"Dave"	32	"absence"

В данном случае значение `#true` преобразуется в `"presence"`, а `#false` – в `"absence"`.

Подсказки. (1) В общем случае вторая база данных может преобразовывать одно поле в несколько полей. Измените пример и добавьте дополнительные поля к строкам `"presence"` и `"absence"`.

(2) Функция также может преобразовывать одно поле в несколько записей, то есть когда процесс преобразования добавляет в новую базу данных несколько записей. Вот второй пример с парой баз данных, немного отличающихся от показанных в табл. 22:

Name String	Age Integer	Present Boolean	Present Boolean	Description String
"Alice"	35	#true	#true	"presence"
"Bob"	25	#false	#true	"here"
"Carol"	30	#true	#false	"absence"
"Dave"	32	#false	#false	"there"

Соединение левой базы данных с правой дает в результате базу данных с восемью записями:

Name String	Age Integer	Description String
"Alice"	35	"presence"
"Alice"	35	"here"
"Bob"	25	"absence"
"Bob"	25	"there"
"Carol"	30	"presence"
"Carol"	30	"here"
"Dave"	32	"absence"
"Dave"	32	"there"

(3) Используйте прием итеративного уточнения для решения задачи. В первой итерации предположите, что преобразование просто заменяет одно поле другим. Во второй итерации отбросьте это предположение.

Примечание о предположениях. Это упражнение и весь раздел полагаются на неформальные предположения о базах данных. Здесь при проектировании функции `join` предлагается исходить из предположения, что «схема db-2 начинается с той же спецификации, на которой заканчивается схема db-1». На самом деле функции базы данных должны быть проверяющими функциями, как описано в разделе 6.3. Однако спроектировать функцию `checked-join` вам пока не под силу. Сравнение последнего экземпляра Spec в схеме db-1 с первым в db-2 требует сравнения функций. Обсуждение практических решений ищите в специализированных книгах о базах данных. ■

24. Итоги

Эта четвертая часть книги посвящена проектированию функций, обрабатывающих данные, для описания которых требуется множество взаимосвязанных определений. Эти формы данных встречаются повсюду в реальном мире, от локальной файловой системы вашего компьютера до Всемирной паутины и геометрических фигур, используемых в анимационных фильмах. Скрупулезно проработав эту часть книги, вы узнали, что рецепт проектирования также подходит для следующих форм данных:

- 1) когда для описания данных требуется несколько определений, ссылающихся друг на друга, рецепт проектирования требует одновременной разработки макетов, по одному для каждого определения. Если определение данных A ссылается на определение данных B, то макет `function-for-A` должен ссылаться на `function-for-B` в том же месте и таким же образом. В остальном рецепты проектирования работают так же, как раньше;
- 2) когда функция должна обрабатывать два сложных типа данных, нужно различать три случая. Во-первых, функция может интерпретировать один из аргументов как атомарное значение. Во-вторых, два аргумента могут иметь одинаковую структуру, и тогда функция должна обрабатывать их параллельно. В-третьих, функция может обрабатывать все возможные комбинации по отдельности. В этом последнем случае следует создать двухмерную таблицу и по горизонтали перечислить все виды значений из одного определения данных, а по вертикали – из второго. После этого нужно заполнить ячейки таблицы, сформулировав условия и ответы для различных случаев.

В этой части книги рассматриваются функции от двух сложных аргументов. Если вы когда-нибудь столкнетесь с одним из тех редких случаев, когда функция должна принимать три сложных экземпляра данных, то вы должны создать (вообразить) трехмерную таблицу (куб).

Теперь вы познакомились со всеми формами структурных данных, с которыми вам наверняка придется столкнуться в своей карьере, хотя детали будут отличаться. Если вы когда-нибудь застопоритесь, вспомните рецепт проектирования; это поможет вам сдвинуться с мертвой точки.

Интермеццо 4. Природа чисел

Оперируя с числами, языки программирования сокращают разрыв между аппаратным обеспечением и истинной математикой. Типичное компьютерное оборудование представляет числа в виде фрагментов данных фиксированного размера и снабжается процессорами, которые работают именно с такими фрагментами. В вычислениях на бумаге мы не заботимся о том, сколько цифр обрабатываем; в принципе, мы можем производить вычисления с числами, состоящими из одной цифры, 10 цифр или 10 000 цифр. Таким образом, если язык программирования использует числа, поддерживаемые базовым оборудованием, то вычисления будут выполняться максимально эффективно. Но если использовать числа, которые мы знаем из математики, то языку программирования придется преобразовывать их в блоки данных, поддерживаемые оборудованием, и обратно, а эти преобразования требуют времени. Из-за этих накладных расходов большинство создателей языков программирования выбирают аппаратное представление чисел.

Это интермеццо объясняет аппаратное представление чисел как пример представления данных. В первом разделе вводится конкретное представление данных для чисел фиксированного размера, обсуждается, как преобразовывать числа в это представление и как обрабатываются такие числа. Второй и третий разделы иллюстрируют две наиболее фундаментальные проблемы этого выбора: арифметическое переполнение и потерю значимости соответственно. В последнем разделе показано, как работает арифметика в обучающих языках; их система счисления **обобщает** то, что можно найти в большинстве современных языков программирования. Заключительные упражнения показывают, что может пойти не так в программах, выполняющих вычисления с числами.

Эти фрагменты называются битами, байтами и словами.

Арифметика с числами фиксированного размера

Предположим, что мы можем использовать четыре цифры для представления чисел. Если говорить о натуральных числах, то такому представлению соответствует диапазон [0, 10 000). Для действительных чисел мы могли бы выбрать 10 000 дробей от 0 до 1 или 5000 от 0 до 1 и еще 5000 от 1 до 2 и т. д. В любом случае четыре цифры позволяют представить не более 10 000 чисел в некотором выбранном интервале, в то время как числовая прямая для этого интервала содержит бесконечное количество чисел.

В стандартной экспоненциальной записи базовое число находится в диапазоне от 0 до 9, но мы игнорируем это ограничение.

Обычно для представления чисел в аппаратном обеспечении выбирается так называемая экспоненциальная запись, в которой числа делятся на две части:

- 1) мантиссу – базовое число;
- 2) порядок (показатель степени) – используется для определения коэффициента по основанию 10.

На языке формул эти числа записываются так:

$$m \cdot 10^e,$$

где m – это мантисса, а e – порядок. Вот пример представления числа 1200 в этой нотации:

$$120 \cdot 10^1.$$

А вот еще один пример:

$$12 \cdot 10^2.$$

Как правило, число имеет несколько эквивалентных представлений.

Точно так же можно использовать отрицательные порядки, которые дают нам дроби за счет одной дополнительной части данных: знака порядка. Например:

$$1 \cdot 10^{-2},$$

что равноценно:

$$\frac{1}{100}.$$

Чтобы использовать нотацию с мантиссой и порядком, мы должны решить, сколько цифр из четырех допустимых мы будем использовать для представления мантиссы, а сколько для порядка. Здесь мы используем по две цифры плюс знак для порядка, но возможны и другие варианты. С учетом этого решения мы можем представить число 0 как

$$0 \cdot 10^0.$$

Максимально возможное число, которое можно представить:

$$99 \cdot 10^{99},$$

то есть 99 с 99 нулями. Используя отрицательный порядок, мы можем использовать дробные значения, например

$$01 \cdot 10^{-99},$$

которое является наименьшим представимым положительным числом. Таким образом, используя экспоненциальную запись с четырьмя цифрами (и знаком), можно представить широкий диапазон чисел и дробей, но такое ограничение имеет свои проблемы.

Листинг 97. Функции для неточного представления чисел

```
| ; N Number N -> Inex (Inexact -- неточное число)XE "Inex, определение данных: "
| ; создает экземпляр Inex после проверки аргументов
(define (create-inex m s e)
  (cond
    [(and (<= 0 m 99) (<= 0 e 99) (or (= s 1) (= s -1)))
     (make-inex m s e)]
    [else (error "bad values given")])))

| ; Inex -> Number
| ; преобразует inex в числовой эквивалент
(define (inex->number an-inex)
  (* (inex-mantissa an-inex)
     (expt
      10 (* (inex-sign an-inex) (inex-exponent an-inex)))))
```

Чтобы понять, в чем заключаются проблемы, рассмотрим конкретный пример, в котором используются представления данных на ISL+, и проведем несколько экспериментов. Давайте представим число фиксированного размера структурой с тремя полями:

```
| (define-struct inex [mantissa sign exponent])
| ; Inex -- это структура:
| ; (make-inex N99 S N99)
| ; S -- это одно из значений:
| ; -- 1
| ; -- -1
| ; N99 -- это N (натуральное число) в диапазоне от 0 до 99 (включительно).
```

Поскольку условия для полей Inex очень строгие, определим функцию `create-inex` для создания экземпляра структуры; см. листинг 97. В листинге также определяется функция `inex->number`, которая превращает экземпляры Inex в числа, используя приведенную выше формулу.

Давайте преобразуем приведенный выше пример числа 1200 в наше представление данных:

```
| (create-inex 12 1 2)
```

Представление числа 1200 как $120 \cdot 10^1$ недопустимо, согласно нашему определению Inex:

```
| > (create-inex 120 1 1)
| bad values given
```

Однако для других чисел мы можем найти два эквивалентных представления в форме Inex. Один из примеров – число 5e-19:

```
| > (create-inex 50 -1 20)
| (make-inex 50 -1 20)
```

```
| > (create-inex 5 -1 19)
| (make-inex 5 -1 19)
```

Используйте `inex->number`, чтобы подтвердить эквивалентность этих двух чисел.

С помощью `create-inex` также легко ограничить диапазон представимых чисел, который на самом деле довольно мал для многих применений:

```
| (define MAX-POSITIVE (create-inex 99 1 99))
| (define MIN-POSITIVE (create-inex 1 -1 99))
```

Вопрос в том, какое из действительных чисел в диапазоне от 0 до `MAXPOSITIVE` можно преобразовать в `Inex`. В данном случае любое положительное число, которое меньше

$$10^{-99},$$

не имеет эквивалентного представления в форме `Inex`. Аналогично это представление имеет разрывы в середине. Например, два числа, следующих непосредственно друг за другом:

```
| (create-inex 12 1 2)
```

и

```
| (create-inex 13 1 2)
```

Первый экземпляр `Inex` представляет число 1200, а второй – 1300. Числа между ними, например 1240, будут представлены в виде одного или другого экземпляра `Inex` – никакие другие экземпляры `Inex` не имеют смысла. Мы оказываемся перед стандартным выбором – округлить число до ближайшего представимого эквивалента. Именно это специалисты по информатике подразумевают под понятием *неточные числа*. То есть выбранное представление данных заставляет нас отображать математические числа в приближенные эквиваленты.

Наконец, мы должны также рассмотреть арифметические операции со структурами `Inex`. Сложение двух представлений `Inex` с одинаковым экспонентами означает сложение двух мантисс:

```
| (inex+ (create-inex 1 1 0) (create-inex 2 1 0))
| ==
| (create-inex 3 1 0)
```

Преобразовав это в математическую форму записи, получим:

$$\begin{array}{r} 1 \cdot 10^0 \\ + 2 \cdot 10^0 \\ \hline 3 \cdot 10^0 \end{array}$$

Когда в результате сложения двух мантисс получается слишком много цифр, мы должны использовать ближайшего соседа в пред-

ставлении Inex. Представьте сложение числа $55 \cdot 10^0$ с самим собой. В математике мы получим

$$110 \cdot 10^0,$$

но мы не сможем преобразовать это число в выбранное нами представление, потому что $110 > 99$. Правильное корректирующее действие – представить результат как

$$11 \cdot 10^1.$$

Или, переведя это на язык ISL+, мы должны гарантировать, что `inex+` выполнит следующие вычисления:

```
(inex+ (create-inex 55 1 0) (create-inex 55 1 0))
==
(create-inex 11 1 1)
```

Проще говоря, если мантисса результата слишком велика, то мы должны разделить ее на 10 и увеличить порядок на единицу.

Иногда мантисса результата содержит больше цифр, чем можно представить. В этих случаях `inex+` должна округлить ее до ближайшего эквивалента в мире Inex. Например:

```
(inex+ (create-inex 56 1 0) (create-inex 56 1 0))
==
(create-inex 11 1 1)
```

Сравните этот подход с точными вычислениями:

$$56 \cdot 10^0 + 56 \cdot 10^0 = (56 + 56) \cdot 10^0 = 112 \cdot 10^0.$$

Поскольку мантисса результата содержит слишком много цифр, целочисленное деление мантиссы на 10 даст приблизительный результат:

$$11 \cdot 10^1.$$

Это одно из многих приближений в арифметике Inex. *И неточность неизбежна.*
Мы также можем умножать числа Inex. Как вы помните:

$$\begin{aligned} & (a \cdot 10^n) \cdot (b \cdot 10^m) \\ &= (a \cdot b) \cdot 10^n \cdot 10^m \\ &= (a \cdot b) \cdot 10^{(n+m)}. \end{aligned}$$

То есть:

$$2 \cdot 10^{+4} \cdot 8 \cdot 10^{+10} = 16 \cdot 10^{+14},$$

или если записать то же самое на языке ISL+:

```
(inex* (create-inex 2 1 4) (create-inex 8 1 10))
==
(create-inex 16 1 14)
```

Как и в случае со сложением, все не так просто. Если мантисса результата содержит слишком много значащих цифр, `inex*` должна увеличить порядок:

```
| (inex* (create-inex 20 1 1) (create-inex 5 1 4))
| ==
| (create-inex 10 1 6)
```

И так же как `inex+`, `inex*` выполняет приближение, если истинная мантисса не имеет точного эквивалента в Inex:

```
| (inex* (create-inex 27 -1 1) (create-inex 7 1 4))
| ==
| (create-inex 19 1 4)
```

Упражнение 412. Спроектируйте функцию `inex+`. Она должна складывать два числа в представлении Inex с одинаковым порядком. Также функция должна учитывать возможность увеличения порядка в результате сложения. Кроме того, она должна сообщать об ошибке, если результат выходит за пределы допустимого диапазона, не полагаясь на использование `create-inex` для проверки ошибок. ■

Усложненная задача. Расширьте функцию `inex+` так, чтобы она могла обрабатывать входные данные с порядками, отличающимися друг от друга на 1:

```
| (check-expect
|   (inex+ (create-inex 1 1 0) (create-inex 1 -1 1))
|   (create-inex 11 -1 1))
```

Не пытайтесь реализовать обработку более крупных классов входных данных, не прочитав следующий раздел. ■

Упражнение 413. Спроектируйте функцию `inex*`. Она должна умножать два числа в представлении Inex, включая числа, которые вызывают увеличение экспоненты в результате. Так же как `inex+`, она должна сообщать об ошибке, если результат выходит за пределы допустимого диапазона, не полагаясь на использование `create-inex` для проверки ошибок. ■

Упражнение 414. Как показано в этом разделе, разрывы в представлении данных приводят к ошибкам округления. Проблема в том, что такие ошибки округления накапливаются в вычислениях.

Спроектируйте функцию `add`, которая складывает n копий $\#i1/185$. Для ваших примеров используйте 0 и 1; для последнего используйте допуск 0.0001. Какое значение вернет (`add 185`)? Какое значение вы ожидали? Что получится, если умножить результат на большое число?

Спроектируйте функцию `sub`. Она должна определить, сколько раз можно вычесть $1/185$ из аргумента, пока в результате не получится 0. Используйте 0 и $1/185$ в своих примерах. Какие значения вернут (`sub 1`) и (`sub #i1.0`)? Какие значения вы ожидали? Что получится во втором случае? Почему? ■

Переполнение

Экспоненциальная запись расширяет диапазон чисел, которые можно представить с помощью блоков данных фиксированного размера, но она по-прежнему ограничена. Некоторые числа слишком велики, чтобы уместиться в числовое представление фиксированного размера. Например:

$$99 \cdot 10^{500}$$

не может быть представлено в виде Inex, потому что порядок 500 нельзя записать двумя цифрами, а мантисса имеет максимально допустимое значение.

Во время вычислений могут возникнуть числа, которые слишком велики для Inex. Например, сумма двух допустимых чисел может получиться слишком большой, чтобы ее можно было представить в виде экземпляра Inex:

```
| (inex+ (create-inex 50 1 99) (create-inex 50 1 99))
| ==
| (create-inex 100 1 99)
```

потому что нарушается определение данных. Когда в результате получаются числа, которые слишком велики для представления, мы говорим о (арифметическом) переполнении.

Когда происходит переполнение, некоторые языки сообщают об ошибке и останавливают вычисления. Другие возвращают некоторое символьическое значение, называемое бесконечностью и предназначеннное для представления таких чисел, и передают дальше в другие арифметические операции.

ПРИМЕЧАНИЕ. Если бы в представлении Inex имелось поле для знака мантиссы, то в результате сложения двух отрицательных чисел мы могли бы получить настолько большое по модулю отрицательное число, что его также нельзя было бы представить в форме Inex. Это называется *переполнением в отрицательном направлении. КОНЕЦ.*

Упражнение 415. Для борьбы с переполнениями в ISL+ используется значение `+inf.0`. Определите целое число `n` такое, что

```
| (expt #i10.0 n)
```

является неточным числом, а `(expt #i10. (+ n 1))` дает в результате `+inf.0`. **Подсказка.** Спроектируйте функцию для вычисления `n`. ■

Потеря значимости

На другом конце спектра находятся маленькие числа, которые тоже невозможно представить в формате Inex. Например, 10^{-500} – это не 0,

но это число меньше наименьшего ненулевого числа, которое можно представить. Потеря значимости (арифметическая) возникает при перемножении двух маленьких чисел, когда результат оказывается слишком маленьким для Inex:

```
| (inex* (create-inex 1 -1 10) (create-inex 1 -1 99))
| ==
| (create-inex 1 -1 109)
```

Когда происходит потеря значимости, некоторые языки сообщают об ошибке; другие используют 0, чтобы вернуть приближенный результат. Использование 0 для аппроксимации потери значимости качественно отличается от выбора приблизительного представления числа в Inex. В частности, аппроксимация 1250 с помощью (create-inex 12 1 2) отбросит значащие цифры из мантиссы, но результат никогда не будет отличаться более чем на 10 % от числа, которое должно быть представлено. Аппроксимация при потере значимости отбрасывает всю мантиссу, то есть погрешность результата выходит за пределы предсказуемого процента.

Упражнение 416. Для борьбы с потерей значимости в ISL+ используется значение #i0.0. Определите наименьшее целое число n такое, что ($\text{expt } \#i10.0 n$) все еще остается неточным числом в ISL+, а ($\text{expt } \#i10. (- n 1)$) аппроксимируется нулем. **Совет.** Спроектируйте функцию для вычисления n . Попробуйте абстрагировать эту функцию и решение упражнения 415. ■

Числа в *SL

Существует несколько типов приблизительных представлений действительных чисел: *float*, *double*, *extflonum* и т. д.

Большинство языков программирования поддерживают только приблизительные представления чисел и арифметические операции с ними. Обычно целые числа ограничиваются определенным интервалом, протяженность которого обусловлена возможностями оборудования. Представление действительных чем-то напоминает представление Inex, описанное в предыдущих разделах, но основано на большем количестве двоичных цифр.

Учебные языки поддерживают оба вида чисел – точные и неточные. Целые и рациональные числа могут быть произвольно большими и точными и ограничиваются только доступным объемом памяти компьютера. Для вычислений с этими числами наши обучающие языки используют базовые аппаратные средства, если рациональные числа умещаются в поддерживаемые аппаратурой фрагменты данных, и автоматически переключаются на другое представление и другую версию арифметических операций с числами, выходящими за пределы этого интервала. Действительные числа имеют две разновидности: точные и неточные. Точное число представляет истинное значение; неточное – аппроксимирует истинное число примерно так,

как рассказывалось выше. Арифметические операции сохраняют точность, когда это возможно, и дают неточный результат в иных случаях. Таким образом, `sqrt` возвращает неточное число как для точного, так и для неточного представления 2. Однако `sqrt` дает точное число 2, когда получает точное число 4, и `#i2.0`, когда получает `#i4.0`. Наконец, числовая константа воспринимается обучающими языками как точное рациональное число, только если она не имеет префикса `#i`.

Racket интерпретирует все десятичные дроби как неточные и отображает все действительные числа как десятичные дроби, независимо от того, являются они точными или нет. Как следствие ко всем таким числам следует относиться с осторожностью, потому что они могут быть неточными приближениями истинных чисел. Программист может заставить Racket интерпретировать числа с точкой как точные, добавив в определение числовой константы префикс `#e`.

После прочитанного у многих из вас может возникнуть вопрос: насколько результаты программы могут отличаться от истинных результатов из-за использования неточных чисел? Этот вопрос давно волнует многих исследователей в области информатики, для ответа на который они основали даже отдельную область исследований, получившую название *числовой анализ*. Каждый исследователь, а на самом деле каждый человек, использующий компьютеры и программное обеспечение, должен знать о существовании этой области и некоторые из основных представлений о разработке числовых программ, зародившихся в ней. Чтобы дать начальное представление, ниже приводятся упражнения, иллюстрирующие, насколько серьезными могут быть последствия, если не принимать во внимание ошибки, накапливаемые при работе с неточными числами. Выполните их, чтобы потом никогда не упускать из виду проблемы неточных чисел.

Упражнение 417. Вычислите (`(expt 1.001 1e-12)`) в Racket и ISL+. Объясните полученный результат. ■

Упражнение 418. Спроектируйте функцию `my-expt` без использования `expt`. Она должна принимать два числа и возводить первое число в степень, определяемую вторым натуральным числом. Используя эту функцию, проведите следующий эксперимент. Добавьте в область определений следующие строки:

```
| (define inex (+ 1 #i1e-12))
| (define exac (+ 1 1e-12))
```

Что вернет выражение (`my-expt inex 30`)? Что вернет выражение (`my-expt exac 30`)? Какой результат более полезен? ■

Листинг 98. Серия неточных двуликих чисел

```
| (define JANUS
|   (list 31.0
```

Доступное введение на основе Racket вы найдете в статье «Practically Accurate Floating-Point Math», где Нейл Торонто (Neil Toronto) и Джей Маккарти (Jay McCarthy) описывают анализ ошибок. Также можно посоветовать просмотреть увлекательную лекцию «Debugging Floating-Point Math in Racket», прочитанную Нейлом Торонто (Neil Toronto) на RacketCon 2011, которая доступна на YouTube.

```
#i2e+34
#i-1.2345678901235e+80
2749.0
-2939234.0
#i-2e+33
#i3.2e+270
17.0
#i-2.4e+270
#i4.2344294738446e+170
1.0
#i-8e+269
0.0
99.0))
```

Упражнение 419. При сложении двух неточных чисел, существенно отличающихся по величине, могут возникнуть существенные ошибки округления. Например, если числовая система использует только 15 значащих цифр, можно столкнуться с проблемой при сложении чисел, которые отличаются друг от друга больше, чем на 10^{16} :

$$1.0 \cdot 10^{16} + 1 = 1.0000000000000001 \cdot 10^{16}$$

– и получить округленный результат, близкий к 10^{16} .

На первый взгляд, такое приближение не выглядит ужасным. Приближенный результат 10^{16} (десять миллионов миллиардов) достаточно близок к истине. К сожалению, эта маленькая ошибка может перерасти в огромные проблемы. Возьмите, к примеру, список чисел в листинге 98 и определите значения следующих выражений:

- (`sum JANUS`);
- (`sum (reverse JANUS)`);
- (`sum (sort JANUS <)`), –

предположив, что `sum` складывает числа в списке слева направо. Объясните результаты, возвращаемые этими выражениями. Что вы думаете о них?

При работе с неточными числами рекомендуется сначала сложить самые маленькие числа. При сложении большого числа и двух маленьких результат может совпасть с большим числом, а если сначала сложить два маленьких числа, то может получиться достаточно большое число, которое повлияет на результат:

```
> (expt 2 #i53.0)
#i9007199254740992.0
> (sum (list #i1.0 (expt 2 #i53.0)))
#i9007199254740992.0
> (sum (list #i1.0 #i1.0 (expt 2 #i53.0)))
#i9007199254740994.0
```

Этот трюк работает не всегда; см. выражения, оперирующие списком `JANUS`, представленные выше.

В некоторых языках, таких как ISL+, есть возможность преобразовать числа в их точные рациональные представления, выполнить

с ними точные арифметические операции и произвести обратное преобразование результата:

```
| (exact->inexact (sum (map inexact->exact JANUS)))
```

Вычислите это выражение и сравните результат с тремя суммами выше. Что вы думаете о рекомендации сейчас? ■

Упражнение 420. JANUS – это простой фиксированный список, но взгляните на следующую функцию:

```
(define (oscillate n)
  (local ((define (0 i)
              (cond
                [(> i n) '()]
                [else
                  (cons (expt #i-0.99 i) (0 (+ i 1))))]))
    (0 1)))
```

Применение `oscillate` к натуральному числу n создает первые n элементов математической последовательности. Ее лучше представить в виде графика, как показано на рис. 19. Запустите `(oscillate 15)` в DrRacket и проверьте результат.

Суммирование результатов слева направо дает иной результат, чем суммирование справа налево:

```
> (sum (oscillate #i1000.0))
#i-0.49746596003269394
> (sum (reverse (oscillate #i1000.0)))
#i-0.4974659600326953
```

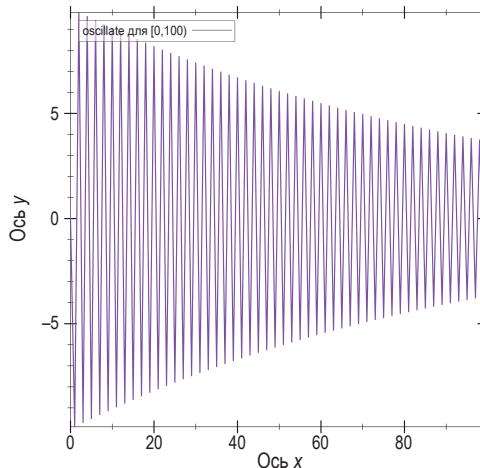


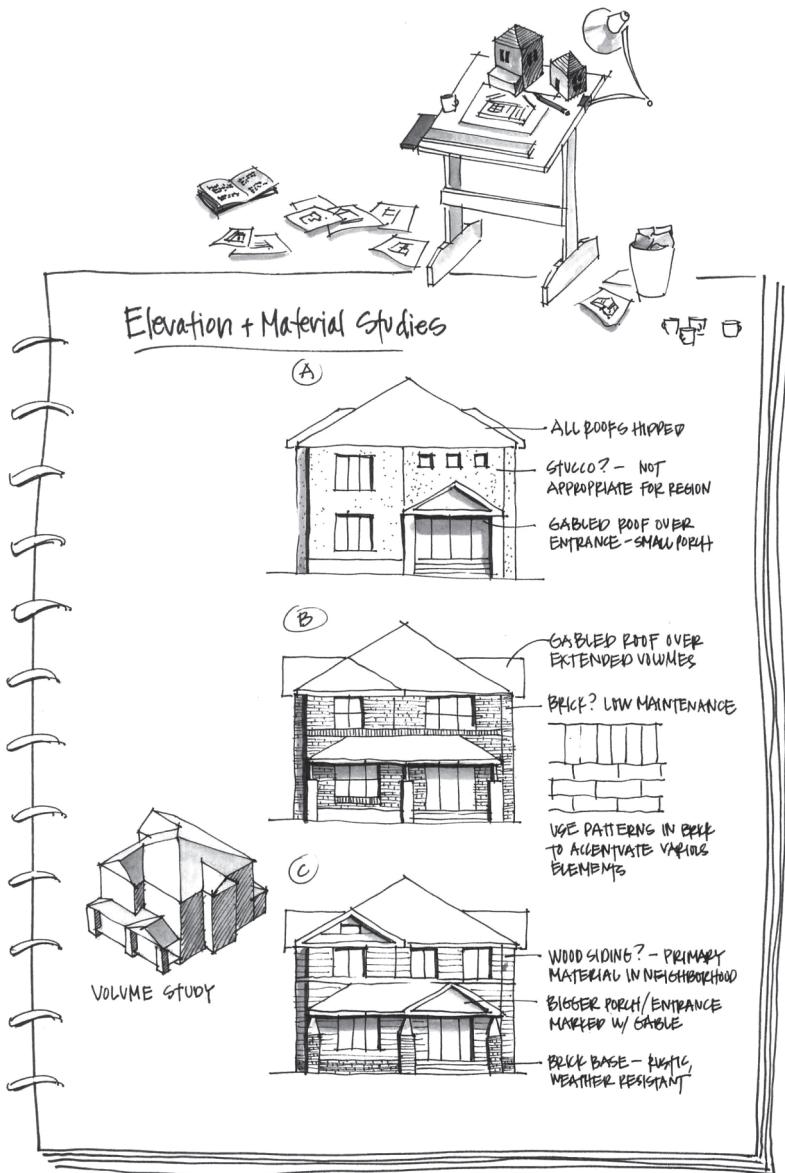
Рис. 19. График функции `oscillate`

И снова разница может показаться несущественной, пока мы не увидим контекста:

```
| > (- (* 1e+16 (sum (oscillate #i1000.0)))
|   (* 1e+16 (sum (reverse (oscillate #i1000.0))))))
| #i14.0
```

Может ли такая разница иметь значение? Можно ли доверять таким вычислениям? ■

Вопрос в том, какие числа должны использовать программисты в своих программах, если у них есть выбор. Ответ зависит от контекста. В мире финансов числовые константы должны интерпретироваться как точные числа, а выкладки в финансовых документах должны полагаться на точность математических операций. В конце концов, закон не предусматривает оправданий ошибкам, получающимся из-за использования неточных чисел и операций. Однако в научных вычислениях иногда можно пожертвовать некоторой точностью ради скорости вычислений. Поэтому ученые, как правило, используют неточные числа, но тщательно анализируют свои программы, чтобы убедиться, что ошибки вычислений не превышают допустимых порогов.



V ГЕНЕРАТИВНАЯ РЕКУРСИЯ

Некоторые функции всего лишь комбинируют такие функции; мы их тоже считаем «структурными».

Следуя за рецептом проектирования в первых четырех частях книги, вы преобразовывали предметные знания в код или использовали структуру определения данных для организации кода. В последнем случае функции обычно разлагают аргументы на структурные компоненты, а затем обрабатывают их. Если один из этих компонентов относится к тому же классу данных, что и сам аргумент, то функция является *структурно-рекурсивной*. Подобные структурные функции составляют подавляющее большинство кода в мире, однако не все задачи можно решить с помощью структурного подхода к проектированию.

Для решения этих сложных задач программисты используют *генеративную рекурсию* – более мощную форму рекурсии, чем структурная рекурсия. Исследования генеративной рекурсии ведутся с незапамятных времен и часто называются исследованием *алгоритмов*. Входные данные алгоритма представляют задачу. Алгоритм стремится преобразовать задачу в набор нескольких подзадач, решает их и объединяет решения в один общий результат. Часто некоторые из этих вновь сгенерированных подзадач имеют тот же вид, что и исходная задача, и в этом случае для их решения можно повторно использовать этот же алгоритм. В таких случаях алгоритм является рекурсивным, но его рекурсия использует вновь сгенерированные данные, не являющиеся частью исходных данных.

На основании одного только описания генеративной рекурсии уже можно сказать, что проектирование генеративно-рекурсивной функции требует больше специфических действий, чем проектирование структурно-рекурсивной функции. Тем не менее многие элементы общего рецепта проектирования применимы и к проектированию алгоритмов, и эта часть книги иллюстрирует, как и в какой степени помогает рецепт проектирования. Основой проектирования алгорит-

*В греческом
это «Эврика!».*

мов является шаг «генерирования», который часто предполагает деление задачи. А для выяснения нового способа деления задачи требуется ее понимание. Иногда достаточно лишь немного вникнуть в задачу. Например, просто применить немного здравого смысла к разделению последовательностей букв. А иногда могут понадобиться глубокие знания математических теорем. На практике программисты проектируют простые алгоритмы самостоятельно и полагаются на знания специалистов в предметной области при создании их сложных собратьев. Любой настоящий программист должен тщательно изучить основные идеи, чтобы суметь воплотить их в код и донести суть до будущих его читателей. Лучший способ понять идею – изучить широкий круг примеров типовых рекурсий, которые могут встретиться в реальном мире.

25. Нестандартная рекурсия

На данный момент вы накопили достаточно богатый опыт проектирования функций, использующих структурную рекурсию. Вы уже знаете, что, приступая к проектированию функции, нужно рассмотреть определение входных данных. Если это определение входных данных содержит ссылку на само себя, то вы знаете, что функция тоже должна ссылаться на саму себя в тех же местах, где определение данных ссылается на себя.

В этой главе представлены два примера программ, которые используют рекурсию иначе. Они иллюстрируют задачи, требующие некоторого озарения, от простых и очевидных до сложных для понимания.

25.1. Рекурсия без структуры

Представьте, что вы присоединились к команде DrRacket, которая работает над созданием службы поддержки совместной работы программистов. В частности, в следующий выпуск DrRacket предполагается включить возможность совместного использования программистами на ISL содержимого их областей определений в DrRacket. Каждый раз, когда один программист изменяет буфер, новая версия DrRacket должна передать содержимое области определений другим экземплярам DrRacket, участвующим в сеансе обмена.

Постановка задачи. Ваша задача – спроектировать функцию `bundle`, которая подготовит содержимое области определений для рассылки. DrRacket хранит это содержимое в виде списка букв `1String`. Задача функции состоит в том, чтобы объединить «буквы» в строки и создать список строк, называемых **фрагментами**, имеющих заданную длину, называемую **размером фрагмента**.

Как видите, постановка задачи недвусмысленно определяет сигнатуру и не требует создания каких-то особых определений данных:

```
| ; [List-of 1String] N -> [List-of String]
| ; объединяет буквы в строки с длиной n
| (define (bundle s n)
|   '())
```

Заявление о назначении переформулирует фрагмент предложения из постановки задачи и использует параметры из заголовка функции.

Третий шаг требует создать примеры применения функции. Вот список букв `1String`:

```
| (list "a" "b" "c" "d" "e" "f" "g" "h")
```

Если потребовать от `bundle` объединить эти буквы в пары, то есть передать число 2 в параметре `n`, то в результате должен получиться следующий список:

```
| (list "ab" "cd" "ef" "gh")
```

Далее, если в `n` передать число 3, то у нас останется неполная строка с одной «буквой». Поскольку заявление о назначении не говорит нам, как поступать с такими буквами, мы можем представить, по крайней мере, два возможных сценария:

- функция возвращает `(list "abc" "def" "g")`, где неполный – последний фрагмент;
- или возвращает `(list "a" "bcd" "efg")`, где неполный – первый фрагмент.

Стоп! Придумайте хотя бы еще один вариант.

Чтобы не усложнять задачу, выберем первый вариант и напишем соответствующий тест:

```
| (check-expect (bundle (explode "abcdefg") 3)
                (list "abc" "def" "g"))
```

Обратите внимание на использование `explode`; это делает тест более удобочитаемым.

Примеры и тесты также должны описывать, что происходит в пограничных случаях. В данной ситуации под пограничным случаем подразумевается передача функции `bundle` списка с размером меньше размера одного фрагмента:

```
| (check-expect (bundle '("a" "b") 3) (list "ab"))
```

Кроме того, мы должны предусмотреть ситуацию, когда функции `bundle` передается `'()`. Для простоты будем считать желаемым результатом `'()`:

```
| (check-expect (bundle '() 3) '())
```

Одна из возможных альтернатив – вернуть `("")`. Сможете предложить другие?

Шаг создания макета показывает, что структурный подход не работает. В листинге 99 показаны четыре возможных макета. Поскольку оба аргумента функции `bundle` не являются элементарными значениями, первые два макета считают один из аргументов атомарным. Этого явно не может быть, потому что функция должна анализировать каждый аргумент. Третий макет основан на предположении синхронной обработки аргументов, что ближе к истине, за исключением того, что `bundle` должна явно сбрасывать размер фрагмента до исходного значения через регулярные интервалы. В последнем макете аргументы обрабатываются независимо, то есть на каждом этапе мы имеем четыре возможности. Этот окончательный вариант

слишком сильно разделяет аргументы, потому что список и счетчик должны обрабатываться вместе. Итак, мы должны признать, что структурные шаблоны оказываются бесполезными для этой задачи проектирования.

Листинг 99. Бесполезные макеты функции, разбивающей строки на фрагменты

```
; N считается составным значением, а s -- атомарным (раздел 23.1)
(define (bundle s n)
  (cond
    [(zero? n) (...)]
    [else (... s ... n ... (bundle s (sub1 n))))]))

; [List-of 1String] считается составным значением,
; а n -- атомарным (раздел 23.1)
(define (bundle s n)
  (cond
    [(empty? s) (...)]
    [else (... s ... n ... (bundle (rest s) n))))])

; [List-of 1String] и N обрабатываются параллельно (раздел 23.2)
(define (bundle s n)
  (cond
    [(and (empty? s) (zero? n)) (...)]
    [else (... s ... n ... (bundle (rest s) (sub1 n))))]))

; рассматривает все возможные варианты (раздел 23.3)
(define (bundle s n)
  (cond
    [(and (empty? s) (zero? n)) (...)]
    [(and (cons? s) (zero? n)) (...)]
    [(and (empty? s) (positive? n)) (...)]
    [else (... (bundle s (sub1 n)) ...
                ... (bundle (rest s) n) ...)]))
```

Листинг 100. Генеративная рекурсия

```
; [List-of 1String] N -> [List-of String]
; объединяет буквы из s в строки с длиной n
; идея: извлечь n элементов и сразу же выбросить их из исходного списка
(define (bundle s n)
  (cond
    [(empty? s) '()]
    [else
      (cons (implode (take s n)) (bundle (drop s n) n))))])

; [List-of X] N -> [List-of X]
; извлекает первые n элементов из l, если возможно,
; или все элементы в ином случае
(define (take l n)
  (cond
    [(zero? n) '()]
    [(empty? l) '()]
    [else (cons (first l) (take (rest l) (sub1 n))))]))

; [List-of X] N -> [List-of X]
; удаляет первые n элементов из l, если возможно,
```

```
; или все элементы в ином случае
(define (drop l n)
  (cond
    [(zero? n) l]
    [(empty? l) l]
    [else (drop (rest l) (sub1 n))]))
```

В листинге 100 показано полное определение функции `bundle`. Она использует функции `drop` и `take`, которые предлагалось спроектировать в упражнении 395; эти функции также доступны в стандартных библиотеках. Их определения приводятся в листинге для полноты картины: `drop` удаляет до `n` элементов из начала списка, `take` возвращает до `n` элементов из начала списка. Использование этих функций упрощает определение `bundle`.

1. Получив список `'()`, `bundle` вернет `'()`, как мы решили выше.
2. В противном случае `bundle` применит `take`, чтобы получить первые `n` букв `1String` из списка `s`, и превратит их в простую строку `String`.
3. Затем она выполнит рекурсивный вызов, передав список без `n` первых элементов, полученный с помощью `drop`.
4. Наконец, `cons` объединит строку, полученную на шаге 2, со списком строк, полученным на шаге 3, чтобы сконструировать результат.

Пункт 3 списка подчеркивает ключевое отличие `bundle` от всех функций, которые вы видели в первых четырех частях этой книги. Поскольку определение `List-of` включает элемент в список, чтобы получить другой список, все функции в первых четырех частях используют `first` и `rest` для разложения непустого списка. Функция `bundle`, напротив, использует функцию `drop`, которая удаляет не один, а `n` элементов.

Такое определение `bundle` выглядит необычным, но основные идеи, заложенные в нем, интуитивно понятны и не слишком отличаются от тех, что мы разбирали до сих пор. Действительно, если размер фрагмента `n` равен 1, то `bundle` приблизится к структурно-рекурсивному определению. Кроме того, `drop` гарантированно возвращает составную часть данного списка, а не произвольно измененную его версию. Именно эту идею мы разберем в следующем разделе.

Упражнение 421. Правильно ли такое использование функции `bundle`: `(bundle '("a" "b" "c") 0)`? Что вернет это выражение? Почему? ■

Упражнение 422. Определите функцию `list->chunks`. Она должна принимать список произвольных данных `l` и натуральное число `n` и возвращать список фрагментов из исходного списка размером `n`. Каждый фрагмент является подпоследовательностью элементов из `l`.

Используйте `list->chunks` для определения `bundle` через композицию функций. ■

Упражнение 423. Определите функцию `partition`. Она должна принимать строку `s` и натуральное число `n` и возвращать список фрагментов строки размером `n`.

Для непустых строк `s` и положительных натуральных чисел `n` выражение

```
| (equal? (partition s n) (bundle (explode s) n))
```

должно возвращать `#true`. Но не используйте это выражение как основу для определения функции `partition`; используйте функцию `substring`.

Подсказка. Применение `partition` к пустой строке должно возвращать естественный результат. См. упражнение 421 для случая, когда `n` равно 0.

Примечание. Функция `partition` несколько ближе к тому, что потребуется для службы совместной работы в DrRacket, чем `bundle`. ■

25.2. Рекурсия, игнорирующая структуру

В главе 11 мы разбирали функцию `sort>`, которая принимает список чисел и переупорядочивает его, обычно по возрастанию или убыванию. В процессе работы она вставляет первое число в соответствующую позицию отсортированного остатка списка. Иначе говоря, это структурно-рекурсивная функция, которая повторно обрабатывает результат естественной рекурсии.

Алгоритм быстрой сортировки Хоара (Hoare) радикально отличается от такой сортировки списков и является классическим примером генеративной рекурсии. В основе генеративного шага используется проверенная временем стратегия «разделяй и властвуй». То есть он делит нетривиальные экземпляры задачи на две более мелкие подзадачи; решает их и объединяет полученные решения в решение исходной задачи. В случае с алгоритмом быстрой сортировки промежуточная цель – разделить список чисел на два списка:

- содержащий все числа, которые строго меньше первого;
- содержащий все числа, которые строго больше первого.

Затем два меньших списка сортируются с помощью алгоритма быстрой сортировки. После этого результаты объединяются с первым элементом, помещенным в середину. Из-за своей особой роли первый элемент в списке называется *опорным элементом* (*pivot item*).

Чтобы лучше понять, как работает алгоритм быстрой сортировки, рассмотрим пример быстрой сортировки списка (`list 11 8 14 7`). На рис. 20 показано, как выглядит этот процесс. Рисунок разделен на две части. В верхней половине показана фаза разделения, а в нижней – фаза властвования.

Фаза разделения представлена прямоугольниками и сплошными стрелками. Из каждого прямоугольника, представляющего список, выходят три стрелки и указывают на прямоугольник с тремя частями: посередине – опорный элемент в кружке, слева от него – пря-

моугольник со списком чисел, которые меньше опорного элемента, и справа – прямоугольник со списком чисел, которые больше опорного элемента. Каждый из этих шагов выделяет хотя бы одно число на роль опорного элемента, что означает, что два соседних списка короче исходного списка. Следовательно, весь процесс рано или поздно завершается.

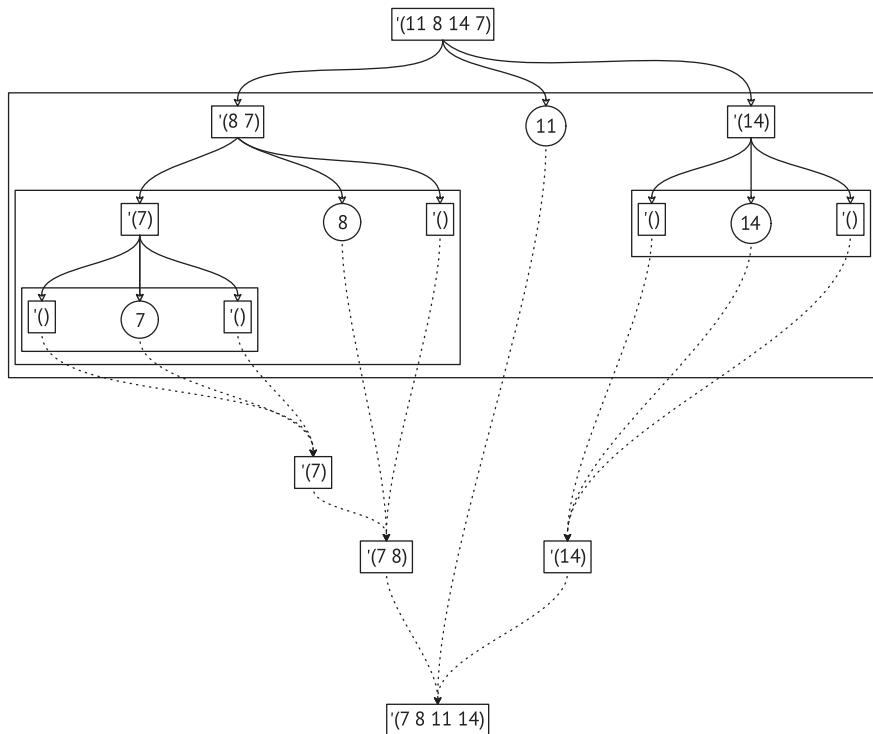


Рис. 20. Графическая иллюстрация алгоритма быстрой сортировки

Рассмотрим первый шаг с исходным списком (*list* 11 8 14 7) на входе. Опорным элементом для него становится число 11 (первый элемент списка). Разделение списка на элементы больше и меньше 11 дает два списка: (*list* 8 7) и (*list* 14). Остальные шаги фазы разделения выполняются аналогично. Разделение заканчивается, когда все числа оказываются изолированными как опорные элементы. На этом этапе вы легко можете определить результат, прочитав опорные элементы слева направо.

Фаза властования представлена пунктирными стрелками и списками в прямоугольниках. В каждом прямоугольнике сходятся три пунктирные стрелки: средняя – от опорного элемента, левая – от результата сортировки списка с числами меньше опорного элемента и правая – от результата сортировки списка с числами больше опор-

ного элемента. На каждом шаге в список результатов добавляется, по крайней мере, одно число – опорный элемент, что означает, что списки растут при движении вниз по диаграмме. Самый нижний прямогольник представляет отсортированный вариант исходного списка.

Взгляните на левый верхний шаг фазы властовования. Он объединяет опорный элемент 7 с двумя пустыми списками, в результате чего получается список '(7). Следующий шаг соответствует шагу разбиения, на котором выделяется число 8 и в результате получается '(7 8). Каждый шаг фазы властовования является отражением соответствующего шага фазы разделения. В конце концов, это обусловлено рекурсивным характером процесса.

Упражнение 424. Нарисуйте диаграмму быстрой сортировки, как на рис. 20, для (list 11 9 2 18 12 14 4 1). ■

Теперь, получив хорошее понимание идеи быстрой сортировки, можно перевести ее на язык ISL+. Очевидно, что quick-sort< различает два случая. Получив '(), она вернет '(), потому что этот список уже отсортирован; в противном случае выполнит генеративную рекурсию. Это предполагает следующее выражение cond:

```
; [List-of Number] -> [List-of Number]
; возвращает отсортированную версиюalon
(define (quick-sort< alon)
  (cond
    [(empty? alon) '()]
    [else ...]))
```

Ответ для первого случая мы уже имеем. Во втором случае, когда quick-sort< получит непустой список, алгоритм использует первый элемент для разделения остальной части списка на два подсписка: список со значениями меньше опорного элемента и список со значениями больше опорного элемента.

Поскольку размер остальной части списка неизвестен, мы перекладываем задачу разделения списка на две вспомогательные функции: *smallers* и *largers*. Они обрабатывают исходный список и оставляют только элементы, которые меньше и больше опорного соответственно. То есть обе функции принимают два аргумента: список чисел и число. Проектирование этих двух функций является упражнением в структурной рекурсии. Попробуйте спроектировать их сами или посмотрите определения в листинге 101.

Каждый из этих списков сортируется отдельно с помощью quick-sort<, что подразумевает использование рекурсии, в частности

- 1) (quick-sort< (*smallers* alon *pivot*)) – сортирует список элементов меньше опорного;
- 2) (quick-sort< (*largers* alon *pivot*)) – сортирует список элементов больше опорного.

Получив отсортированные версии двух списков, quick-sort< должна объединить их с опорным элементом в правильном порядке: снача-

ла список с элементами меньше опорного, затем опорный элемент и, наконец, список с элементами больше опорного. Поскольку первый и последний списки уже отсортированы, quick-sort< может просто использовать append:

```
(append (quick-sort< (smallers alon pivot))
       (list (first alon))
       (quick-sort< (largers alon pivot)))
```

Полная программа приводится в листинге 101; прочитайте ее, перед тем как продолжить.

Листинг 101. Реализация алгоритма быстрой сортировки

```
; [List-of Number] -> [List-of Number]
; возвращает отсортированную версию alon
; предполагается, что все числа разные
(define (quick-sort< alon)
  (cond
    [(empty? alon) '()]
    [else (local ((define pivot (first alon)))
              (append (quick-sort< (smallers alon pivot))
                      (list pivot)
                      (quick-sort< (largers alon pivot))))])))

; [List-of Number] Number -> [List-of Number]
(define (largers alon n)
  (cond
    [(empty? alon) '()]
    [else (if (> (first alon) n)
              (cons (first alon) (largers (rest alon) n))
              (largers (rest alon) n)))]))

; [List-of Number] Number -> [List-of Number]
(define (smallers alon n)
  (cond
    [(empty? alon) '()]
    [else (if (< (first alon) n)
              (cons (first alon) (smallers (rest alon) n))
              (smallers (rest alon) n)))]))
```

Теперь, когда у нас есть фактическое определение функции, мы можем вручную вычислить пример, приведенный выше:

```
(quick-sort< (list 11 8 14 7))
==
(append (quick-sort< (list 8 7))
        (list 11)
        (quick-sort< (list 14)))
==
(append (append (quick-sort< (list 7))
                  (list 8)
                  (quick-sort< '()))
                  (list 11)
                  (quick-sort< (list 14)))
==
(append (append (append (quick-sort< '())

```

```

        (list 7)
        (quick-sort< '()))
      (list 8)
      (quick-sort< '()))
    (list 11)
    (quick-sort< (list 14)))
== 
(append (append (append '()
                         (list 7)
                         '())
                     (list 8)
                     '())
                   (list 11)
                   (quick-sort< (list 14)))
== 
(append (append (list 7)
                  (list 8)
                  '())
                (list 11)
                (quick-sort< (list 14)))
...

```

Процесс вычислений показывает основные этапы сортировки, то есть шаги разделения, рекурсивной сортировки и объединения трех частей. Здесь четко видно, что `quick-sort<` реализует процесс, показанный на рис. 20.

И рис. 20, и процесс вычислений показывают также, что `quick-sort<` полностью игнорирует структуру списка. Первая рекурсия обрабатывает два числа из исходного списка, далеко отстоящих друг от друга, а вторая – третий элемент списка. Эти рекурсии не случайны, но они определенно не зависят от структуры определения данных.

Сравните организацию функций `quick-sort<` и `sort>` из главы 11. Проектируя последнюю, мы следовали за рецептом структурного проектирования, создавая программу, которая обрабатывает список элемент за элементом. Разбивая список, `quick-sort<` может ускорить процесс сортировки, хотя и за счет отказа от простого использования `first` и `rest`.

Упражнение 425. Сформулируйте заявление о назначении для функций `smallers` и `largers` в листинге 101. ■

Упражнение 426. Выполните процесс вычислений вручную, показанный выше. Внимательно изучите предложения, касающиеся еще одного тривиального случая в `quick-sort<`. Каждый раз, когда `quick-sort<` получает список с одним элементом, она возвращает его как есть. В конце концов, отсортированная версия списка из одного элемента – это сам список.

Измените `quick-sort<`, воспользовавшись этим наблюдением. Вычислите пример еще раз. Сколько шагов сэкономил обновленный алгоритм? ■

Упражнение 427. Во многих случаях `quick-sort<` быстро уменьшает размер списка, но с короткими списками она работает недопустимо медленно. По этой причине многие используют `quick-sort<` для разде-

ленияя списка и переключаются на другую функцию сортировки, когда список достаточно мал.

Спроектируйте версию `quick-sort<`, которая использует `sort<` (адаптированный вариант `sort>` из раздела 11.3), если длина входного списка меньше некоторого порога. ■

Упражнение 428. Если входной список, передаваемый в `quick-sort<`, содержит одно и то же число несколько раз, то алгоритм вернет список, который будет короче входного. Почему? Устраните проблему, чтобы в результате список получался той же длины, что и входной список. ■

Упражнение 429. Используйте `filter` в определениях `smallers` и `largers`. ■

Упражнение 430. Разработайте вариант `quick-sort<`, который использует только одну функцию сравнения, например `<`. На этапе разделения она должна делить исходный список на элементы меньше опорного и элементы не меньше опорного.

Используйте `local`, чтобы упаковать программу в единую функцию. Абстрагируйте эту функцию, чтобы она использовала список и функцию сравнения. ■

26. Проектирование алгоритмов

В обзоре к этой части книги мы уже говорили, что проектирование функций с генеративной рекурсией имеет свои отличительные черты, по сравнению со структурным проектированием. Как было показано в предыдущей главе, две генеративные рекурсии могут радикально отличаться по способу обработки. Обе функции, `bundle` и `quick-sort<`, обрабатывают списки, но если первая хотя бы обрабатывает элементы списка последовательно, то вторая переупорядочивает обработку элементов исходного списка по своему усмотрению. Возникает вопрос: существует ли единый рецепт проектирования таких функций, столь сильно отличающихся друг от друга?

В первом разделе этой главы мы покажем, как адаптировать раз мерность процесса в рецепте проектирования к генеративной рекурсии. Второй раздел будет посвящен еще одному явлению: алгоритм может не давать ответа на некоторые входные данные. Поэтому программисты должны анализировать свои программы и дополнять информацию о конструкции комментарием с описанием таких ситуаций. Остальные разделы сравнивают структурную и генеративную рекурсии.

26.1. Адаптация рецепта проектирования

Рассмотрим шесть шагов нашего рецепта структурного проектирования в свете примеров из предыдущей главы.

- Как и прежде, мы должны представить информацию о задаче в виде данных на выбранном языке программирования. Выбор **представления данных** прямо влияет на представление о вычислительном процессе, поэтому требуется некоторое предварительное планирование. В противном случае будьте готовы вернуться назад и исследовать альтернативные представления данных. Так или иначе мы должны проанализировать имеющуюся информацию о задаче и определить коллекции данных.
- Мы все также должны определить сигнатуру, заголовок функции и заявление о назначении. Поскольку шаг генерации не связан со структурой определяемых данных, заявление о назначении должно объяснять не только, **что** вычисляет функция, но и **как** она это делает.
- Объяснение «как» полезно оформить в виде примеров применения функции, как мы делали это при проектировании функций `bundle` и `quick-sort<` в предыдущей главе. То есть если в структурном мире примеры применения функций просто сообщают, какой результат должен получиться для тех или иных входных данных, то в мире генеративной рекурсии примеры должны также объяснять основную идею вычислительного процесса.

В примерах для `bundle` указывается, как функция действует в целом и в некоторых пограничных случаях в частности. В примерах для `quick-sort` на рис. 20 показано, как функция разбивает исходный список с учетом опорного элемента. Добавляя такие примеры работы в заявление о назначении, мы начинаем лучше понимать желаемый процесс вычислений и передаем это понимание будущим читателям кода.

- Наше обсуждение предлагает общий шаблон для алгоритмов. Проще говоря, при разработке алгоритма различают два типа задач: *тривиально решаемые* и нет. Если данная задача имеет тривиальное решение, алгоритм должен выдавать его непосредственно. Примером тривиально решаемой задачи может служить сортировка пустого списка или списка с одним элементом. Список с большим количеством элементов – нетривиальная задача. Для таких нетривиальных задач алгоритмы обычно генерируют новые задачи того же типа, что и данная, рекурсивно решают их и объединяют решения в одно общее решение.

В этой части книги слово «тривиальный» является техническим термином.

```
(define (generative-recursive-fun problem)
  (cond
    [(trivially-solvable? problem)
     (determine-solution problem)]
    [else
     (combine-solutions
      ...
      problem ...
      (generative-recursive-fun
       (generate-problem-1 problem)))
      ...
      (generative-recursive-fun
       (generate-problem-n problem))))])
```

Иногда для объединения решений вновь созданных задач требуется исходная задача (`problem`), поэтому в этом шаблоне алгоритма она передается в функцию объединения решений (`combine-solutions`).

- Этот шаблон представляет лишь общий план. Каждая его часть напоминает нам следующие четыре вопроса:
 - Что такое тривиально решаемая задача?
 - Как решаются тривиальные задачи?
 - Как алгоритм генерирует новые задачи, которые имеют более простое решение, чем исходная? Мы должны сгенерировать одну новую задачу или несколько?
 - Совпадает ли решение данной задачи с решением (одной из) новых задач? Или мы должны объединить решения, чтобы получить решение исходной задачи? И если да, то нужны ли для этого данные из исходной задачи?

Чтобы воплотить алгоритм в функцию, мы должны выразить ответы на эти четыре вопроса в виде функций и выражений в терминах выбранного представления данных.

На этом шаге снова может помочь табличный прием, описанный в главе 9. Еще раз вернемся к функции `quick-sort<` из раздела 25.2. Основная идея `quick-sort<` состоит в том, чтобы разделить исходный список на списки с элементами меньше и больше опорного и отсортировать их по отдельности. В табл. 24 показаны простые числовые примеры для некоторых нетривиальных случаев. Из этих примеров легко догадаться, что ответ на четвертый вопрос выглядит так: объединить отсортированный список меньших чисел, опорное число и отсортированный список больших чисел, который легко преобразовать в код.

- Завершив определение функции, ее нужно протестировать. Как и прежде, целью тестирования является выявление и устранение ошибок.

Таблица 24. Табличный подход для выяснения порядка объединения решений

Исходный список	Опорное значение	Отсортированный список меньших значений	Отсортированный список больших значений	Ожидаемый результат
'(2 3 1 4)	2	'(1)	'(3 4)	'(1 2 3 4)
'(2 0 1 4)	2	'(0 1)	'(4)	'(0 1 2 4)
'(3 0 1 4)	3	'(0 1)	'(4)	'(0 1 3 4)

Упражнение 431. Ответьте на четыре ключевых вопроса, чтобы решить задачу `bundle`, и на первые три вопроса для задачи `quick-sort<`. Сколько требуется экземпляров `generate-problem`? ■

Упражнение 432. В упражнении 219 предлагалось спроектировать функцию `food-create`, которая принимает экземпляр `Posn` и создает другой экземпляр `Posn` со случайными координатами `X` и `Y`, гарантированно не совпадающими с координатами `X` и `Y` заданного экземпляра `Posn`. Сначала объедините две функции в одну, используя `local`; затем объясните полученное определение `food-create`. ■

26.2. Завершимость рекурсии

Генеративная рекурсия привносит в вычисления совершенно новый аспект: неопределенность. Такая функция, как `bundle`, может никогда не вернуть значение или сообщить об ошибке, получив некоторые недопустимые входные данные. В упражнении 421 спрашивается, какой результат вернет вызов (`bundle '("a" "b" "c") 0`), и вот объяснение, почему он не возвращает результата:

```

(bundle '("a" "b" "c") 0)
==
(cons (implode (take '("a" "b" "c") 0))
      (bundle (drop '("a" "b" "c") 0)))
 ==
(cons (implode '())
      (bundle (drop '("a" "b" "c") 0)))
== (cons "" (bundle (drop '("a" "b" "c") 0)))
== (cons "" (bundle '("a" "b" "c") 0))

```

Как показывают пошаговые вычисления, чтобы вычислить выражение `(bundle '("a" "b" "c") 0)`, нужно знать его значение. В контексте ISL+ это означает, что вычисления не прекращаются. Специалисты по информатике говорят, что `bundle` не завершается, когда второй аргумент равен 0; они также говорят, что функция зациклывается или что вычисление попадает в бесконечный цикл.

Сравните эту функцию с функциями, представленными в первых четырех частях. Каждая функция, спроектированная в соответствии с рецептом, либо дает ответ, либо сообщает об ошибке для каждого набора входных данных. В конце концов, рецепт требует, чтобы каждая естественная рекурсия принимала не все входные данные, а только часть их. Поскольку данные имеют иерархическую организацию, на каждом шаге размер входных данных уменьшается. В конце концов функция применяется к элементарному фрагменту данных, и рекурсия останавливается.

Теория вычислений показывает, что мы должны снять эти ограничения.

Это напоминание также объясняет, почему генеративные рекурсивные функции могут расходиться. Согласно рецепту проектирования генеративной рекурсии, алгоритм может генерировать новые задачи без всяких ограничений. Если бы рецепт проектирования требовал, чтобы новые задачи были «меньше» исходной, то рекурсия всегда завершалась бы. Но наложение подобного ограничения без необходимости усложнило бы разработку таких функций, как `bundle`.

Поэтому в этой книге мы оставляем первые шесть шагов рецепта проектирования почти без изменений и дополняем их седьмым шагом: *обоснованием завершения* (*termination argument*). В табл. 25 представлена первая часть рецепта проектирования генеративной рекурсии, а в табл. 26 – вторая. Они иллюстрируют рецепт проектирования в стандартной табличной форме. Неизменявшиеся шаги отмечены тире в столбце «действие». Для других сообщается, чем рецепт проектирования для генеративной рекурсии отличается от рецепта для структурной рекурсии. Последняя строка в табл. 26 описывает полностью новый шаг.

Вы не можете определить предикат для этого класса; в противном случае вы могли бы изменить функцию и убедиться, что она всегда завершается.

Аргумент завершения имеет одну из двух форм. Первая доказывает, почему каждый рекурсивный вызов решает задачу меньше исходной. Обычно этот аргумент формулируется просто, и лишь в очень редких случаях вам может потребоваться поработать с математиком, чтобы доказать теорему для таких аргументов. Вторая

форма иллюстрирует на примере, что функция может не завершаться. В идеале она также должна описывать класс данных, на которых функция может зацикливаться. В редких случаях у вас не получится привести хотя бы один аргумент, потому что они пока неизвестны информатике.

Таблица 25. Проектирование алгоритмов (часть 1)

Шаги	Результат	Действие
Анализ задачи	Представление и определение данных	—
Заголовок	Заявление о назначении, объясняющее, «как» функция вычисляет результат	Дополнить объяснение, что вычисляет функция, коротким описанием, как она это вычисляет
Примеры	Примеры и тесты	Демонстрация «как» несколькими примерами
Макет	Фиксированный макет	—

Таблица 26. Проектирование алгоритмов (часть 2)

Шаги	Результат	Действие
Определение	Законченное определение функции	Сформулировать условия для тривиально решаемых задач; сформулировать ответы на тривиальные случаи; определить, как генерировать новые задачи в нетривиальных случаях, возможно, с использованием вспомогательных функций; определить, как объединить решения генерированных задач в общее решение
Тестирование	Поиск ошибок	—
Завершение	(1) Обоснование уменьшения размера данных для каждого рекурсивного вызова или (2) примеры входных данных, вызывающих зацикливание	Исследовать, уменьшается ли размер входных данных на каждом рекурсивном шаге; найти примеры, которые вызывают зацикливание функции

Проиллюстрируем два вида аргументов завершения на примерах. Для функции `bundle` достаточно предупредить читателей о нулевом размере фрагмента:

```
; [List-of 1String] N -> [List-of String]
; объединяет подпоследовательности из s в строки с длиной n
; завершение: (bundle s 0) зацикливается, если s не является '()
(define (bundle s n) ...)
```

В данном случае можно определить предикат, который точно описывает, когда `bundle` завершается. В случае с `quick-sort<` каждый рекурсивный вызов получает список, который короче, чем `alon`:

```
; [List-of Number] -> [List-of Number]
; возвращает отсортированный версию alon
; завершение: оба рекурсивных вызова quick-sort<
; получают списки, в которых отсутствует опорный элемент
(define (quick-sort< alon) ...)
```

В одном случае список состоит из чисел, которые строго меньше опорного значения; в другом – больше.

Упражнение 433. Разработайте надежную версию `bundle`, которая гарантированно завершается при получении любых входных данных. Она может сообщать об ошибке в случаях, когда оригинальная версия зацикливается. ■

Упражнение 434. Рассмотрим следующее определение `smallers`, одного из двух «генераторов задач» для `quick-sort<`:

```
; [List-of Number] Number -> [List-of Number]
(define (smallers l n)
  (cond
    [(empty? l) '()]
    [else (if (<= (first l) n)
              (cons (first l) (smallers (rest l) n))
              (smallers (rest l) n))]))
```

Что может пойти не так, если использовать эту версию в определении `quick-sort<` из раздела 25.2? ■

Упражнение 435. Работая над упражнением 430 или упражнением 428, вы, возможно, создали решение, способное зациклиться. Точно так же упражнение 434 фактически показывает, насколько хрупким является обоснование завершения для `quick-sort<`. Во всех случаях аргумент основан на идее, что `smallers` и `largers` производят списки, длина которых не превышает длину исходного списка, и на нашем понимании, что ни одна из этих функций не включает указанное опорное значение в результат.

Основываясь на этом объяснении, измените определение `quick-sort<` так, чтобы обе функции возвращали списки короче исходного. ■

Упражнение 436. Сформулируйте аргумент завершения для `food-create` из упражнения 432. ■

26.3. Структурная и генеративная рекурсии

Шаблон проектирования алгоритмов настолько универсальный, что может использоваться и для проектирования структурно-рекурсивных функций. Взгляните на функцию слева в листинге 102. Этот макет содержит одно предложение для тривиального случая и одно – для шага генерации. Если заменить `trivial?` на `empty?` и `generate` на `rest`, то получится типичный макет функций, обрабатывающих списки (справа в листинге 102).

Листинг 102. Генеративная и структурная рекурсии

<pre>(define (general P) (cond [(trivial? P) (solve P)] [else (combine-solutions P (general (generate P))))]))</pre>	<pre>(define (special P) (cond [(empty? P) (solve P)] [else (combine-solutions P (special (rest P))))]))</pre>
--	--

Упражнение 437. Определите функции `solve` и `combine-solutions` так, чтобы `special`:

- вычисляла длину входного списка;
- меняла знак у каждого числа в заданном списке чисел на противоположный;
- преобразовывала в верхний регистр все буквы в заданном списке строк.

Какие выводы можно сделать из этих упражнений? ■

Теперь у кого-то из вас может возникнуть вопрос: есть ли реальная разница между проектированием структурной и генеративной рекурсий? Наш ответ – «зависит от обстоятельств». Конечно, можно сказать, что все функции, использующие структурную рекурсию, являются лишь частным случаем генеративной рекурсии. Однако такая позиция «все равно» не помогает понять процесс проектирования функций. Она смешивает два подхода к проектированию, требующих разных форм знаний и имеющих разные следствия. Первый опирается на систематический анализ данных и не более того; другой требует глубокого, часто математического понимания самого процесса решения задачи. Первый приводит программистов к естественному завершению функций; другой требует аргумента завершения. Объединение этих двух подходов непродуктивно.

26.4. Выбор

Взаимодействуя с функцией `f`, которая сортирует списки чисел, невозможно узнать, реализована `f` как `sort<` или как `quick-sort<`. С точки зрения внешнего наблюдателя эти две функции ведут себя совершенно одинаково. Возникает вопрос: какую из них должен предоставлять язык программирования? То есть когда функцию можно спроектировать с использованием структурной и генеративной рекурсий, мы должны сделать выбор.

Чтобы проиллюстрировать последствия выбора, обсудим классический пример из математики: задачу поиска наибольшего общего делителя (*greatest common divisor, gcd*) для двух натуральных положительных чисел. Все такие числа имеют общий делитель 1. Иногда, например 2 и 3, это также единственный общий делитель. А такие числа, как 6 и 25, имеют несколько делителей:

- 6 делится без остатка на 1, 2, 3 и 6;
- 25 делится без остатка на 1, 5 и 25.

И все же наибольший общий делитель для этой пары чисел равен 1. Напротив, 18 и 24 имеют много общих делителей, и наибольший из них равен 6:

Наблюдаемая эквивалентность – центральная идея изучения языков программирования.

Джон Стоун (John Stone) предложил в качестве примера использовать алгоритм поиска наибольшего общего делителя.

- 18 делится без остатка на 1, 2, 3, 6, 9 и 18;
- 24 делится без остатка на 1, 2, 3, 4, 6, 8, 12 и 24.

Первые три шага рецепта проектирования выполняются просто:

```
| ; N[>= 1] N[>= 1] -> N
| ; отыскивает наибольший общий делитель для n и m
| (check-expect (gcd 6 25) 1)
| (check-expect (gcd 18 24) 6)
| (define (gcd n m) 42)
```

Сигнатура определяет входные данные как натуральные числа, большие или равные 1.

Теперь, согласно этой сигнатуре, спроектируем структурно- и генеративно-рекурсивное решение. Поскольку эта часть книги посвящена генеративной рекурсии, мы просто представляем вашему вниманию готовое структурное решение в листинге 103 и оставляем исследование идей в качестве упражнений. Просто обратите внимание, что $(= (\text{remainder } n i) (\text{remainder } m i) 0)$ реализует проверку, что и n , и m «делятся без остатка» на i .

Листинг 103. Поиск наибольшего общего делителя методом структурной рекурсии

```
(define (gcd-structural n m)
  (local (; N -> N
          ; определяет наибольший общий делитель для n и m,
          ; который меньше i
          (define (greatest-divisor-<= i)
            (cond
              [(= i 1) 1]
              [else
                (if (= (remainder n i) (remainder m i) 0)
                    i
                    (greatest-divisor-<= (- i 1)))])))
    (greatest-divisor-<= (min n m))))
```

Упражнение 438. Объясните своими словами, как работает `greatest-divisor-<=`. Используйте рецепт проектирования, чтобы подобрать нужные слова. Почему локальная функция `greatest-divisor-<=` применяется к выражению `(min n m)`? ■

Хотя определение `gcd-structural` выглядит прямолинейно, оно весьма наивно. Эта функция просто проверяет каждое число, начиная с меньшего из n и m и заканчивая единицей, делит ли оно n и m нацело, и возвращает первое такое число. Эта функция прекрасно справляется с маленькими числами. Но взгляните на следующий пример:

```
| (gcd-structural 101135853 45014640)
```

Результат – число 177. Чтобы найти его, `gcd-structure` проверяет условие «деления без остатка» для 45 014 640, то есть вызывает `remainder` для всех чисел в диапазоне от 45 014 640 до 177. Такое количество проверок требует много времени, даже на достаточно быстром компьютере.

Упражнение 439. Скопируйте gcd-structure в DrRacket и вычислите

```
| (time (gcd-structural 101135853 45014640))
```

в области взаимодействий. ■

Математики давно осознали неэффективность этого структурного решения и глубоко изучили проблему поиска делителей. Основная идея заключается в том, что

для двух натуральных чисел, L – большего и S – меньшего, наибольший общий делитель равен наибольшему общему делителю S и остатку от деления L на S .

Вот как мы можем сформулировать эту идею в виде уравнения:

```
| (gcd L S) == (gcd S (remainder L S))
```

Поскольку результат (`remainder L S`) всегда меньше, чем L и S , в первом аргументе функции `gcd` передается S .

Вот как это знание можно применить к нашему небольшому примеру:

- даны числа 18 и 24;
- согласно имеющимся знаниям, они имеют тот же наибольший общий делитель, что и числа 18 и 6;
- а эти два числа имеют тот же наибольший общий делитель, что и 6 и 0.

Похоже, что мы столкнулись с чем-то неожиданным, потому что 0 – это совершенно неожиданно. Но 0 делится без остатка на любое число, а это означает, что мы нашли ответ: 6.

Работа над примером не только подтверждает базовую идею, но также подсказывает, как превратить ее в алгоритм:

- когда меньшее из чисел равно 0 – это тривиальный случай;
- решением тривиального случая является большее из двух чисел;
- чтобы сгенерировать новую задачу, достаточно одной операции `remainder`;
- приведенное выше уравнение говорит нам, что ответ на новую задачу также является ответом на исходную задачу.

Проще говоря, отпадают ответы на четыре вопроса о рецепте проектирования.

В листинге 104 приводится определение данного алгоритма. Выражение `local` определяет функцию `clever-gcd` – рабочую лошадку алгоритма. Ее первое предложение в `cond` обнаруживает тривиальный случай, сравнивая меньшее число `smaller` с 0, и возвращает соответствующее решение. Генеративный шаг рекурсивно вызывает `clever-gcd` и передает ей `smaller` в первом аргументе и (`remainder large small`) – во втором.

Листинг 104. Поиск наибольшего общего делителя методом генеративной рекурсии

```
(define (gcd-generative n m)
  (local ( ; N[>= 1] N[>=1] -> N
         ; генеративная рекурсия
         ; (gcd L S) == (gcd S (remainder L S))
         (define (clever-gcd L S)
           (cond
             [(= S 0) L]
             [else (clever-gcd S (remainder L S))]))
         (clever-gcd (max m n) (min m n))))
```

Если теперь вызвать `gcd-generative` для той же пары чисел, что и в примере выше:

```
| (gcd-generative 101135853 45014640)
```

то мы получим результат практически мгновенно. Если выполнить вычисления в пошаговом режиме, то можно увидеть, что `clever-gcd` выполняет рекурсивный вызов всего девять раз:

```
...
== (clever-gcd 101135853 45014640)
== (clever-gcd 45014640 11106573)
== (clever-gcd 11106573 588348)
== (clever-gcd 588348 516309)
== (clever-gcd 516309 72039)
== (clever-gcd 72039 12036)
== (clever-gcd 12036 11859)
== (clever-gcd 11859 177)
== (clever-gcd 177 0)
```

То есть условие с `remainder` проверяется только девять раз, что явно намного меньше, чем в `gcd-structure`.

Упражнение 440. Скопируйте `gcd-generative` в область определений DrRacket и вычислите

```
| (time (gcd-generative 101135853 45014640))
```

в области взаимодействий. ■

Теперь вы могли бы подумать, что подход к проектированию на основе генеративной рекурсии позволил найти гораздо более быстрое решение задачи наибольшего общего делителя и прийти к выводу, что генеративная рекурсия – это всегда правильный путь. Однако это суждение слишком опрометчиво по трем причинам. Во-первых, даже хорошо спроектированный алгоритм не всегда быстрее эквивалентного структурно-рекурсивного решения. Например, `quick-sort<` дает выигрыш во времени только для больших списков; для маленьких списков стандартная функция `sort<` дает решение быстрее. Хуже того, плохо спроектированный алгоритм может отрицательно повлиять на производительность программы. Во-вторых, спроектировать структурно-рекурсивную функцию обычно намного проще. И наоборот, разработка генеративно-рекурсивного алгоритма требует представ-

лять, как генерировать новые задачи – этот шаг часто требует глубокого понимания. Наконец, программистам обычно проще понять структурно-рекурсивные функции, даже без подробного описания. Однако генеративный шаг алгоритма основан на «озарении», и без хорошего объяснения будущим читателям будет трудно понять его, даже нам самим, спустя какое-то время.

Как показывает опыт, большинство функций в программах реализованы с использованием структурного подхода к проектированию; и лишь немногие – с использованием генеративной рекурсии. Столкнувшись с задачей, когда можно использовать оба подхода к проектированию – структурный или генеративный, лучше начать со структурной версии. Если получившаяся функция оказывается слишком медленной – и только тогда, – можно исследовать решение на основе генеративной рекурсии.

Упражнение 441. Вычислите

```
| (quick-sort< (list 10 6 8 9 14 12 3 11 14 16 2))
```

в пошаговом режиме. Покажите только те строки, которые вводят новый рекурсивный вызов `quick-sort<`. Сколько требуется рекурсивных вызовов `quick-sort<?` Сколько требуется рекурсивных вызовов функции `append?` Предложите общее правило для списка длиной n .

Вычислите

```
| (quick-sort< (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14))
```

в пошаговом режиме. Сколько требуется рекурсивных вызовов `quick-sort<?` Сколько требуется рекурсивных вызовов функции `append?` Противоречит ли это первой части упражнения? ■

Упражнение 442. Добавьте определения `sort<` и `quick-sort<` в область определений. Выполните тесты функций, чтобы убедиться, что они справляются с простыми примерами. Также разработайте `generate-tests` – функцию, которая создает случайные большие тестовые примеры. Затем проверьте, как быстро каждая из функций справляется с различными списками.

Подтверждает ли эксперимент утверждение о том, что простая функция `sort<` часто выигрывает у функции `quick-sort<` на коротких списках, и наоборот?

Определите точку перегиба. Используйте ее для создания интеллектуальной функции сортировки, которая к большим спискам применяет `quick-sort<` и к коротким – `sort<`. Сравните получившийся результат с упражнением 427. ■

Упражнение 443. Учитывая описание в заголовке для `gcd-structure`, при наивном применении рецепта проектирования может получиться следующий макет или какой-либо его вариант:

```
(define (gcd-structural n m)
  (cond
    [(and (= n 1) (= m 1)) ...]
```

```

[[(and (> n 1) (= m 1)) ...]
 [(<= n 1) (> m 1)) ...]
 [else
  (... (gcd-structural (sub1 n) (sub1 m)) ...
    ... (gcd-structural (sub1 n) m) ...
    ... (gcd-structural n (sub1 m)) ...))])

```

Почему с помощью этой стратегии невозможно найти делитель? ■

Упражнение 444. В упражнении 443 предполагается, что проектирование gcd-structure требует некоторого планирования и подхода «проектирование методом композиции».

В идеале следовало бы использовать множества, а не списки. Само описание алгоритма поиска наибольшего общего делителя предполагает двухэтапный подход. На первом этапе проектируется функция, вычисляющая список делителей

натурального числа. На втором – проектируется функция, которая выбирает наибольшее общее число из списков делителей для n и m . Итоговая функция будет выглядеть так:

```

(define (gcd-structural S L)
  (largest-common (divisors S S) (divisors S L)))

; N[>= 1] N[>= 1] -> [List-of N]
; вычисляет делители для l, которые меньше или равны k
(define (divisors k l)
  '())

; [List-of N] [List-of N] -> N
; находит наибольшее число, общее для k и l
(define (largest-common k l)
  1)

```

Как вы думаете, почему `divisors` принимает два числа? Почему в обоих случаях ей передается число S в первом аргументе? ■

27. Вариации на тему

Проектирование алгоритма начинается с неформального описания процесса создания задачи, которая имеет более простое решение, способствующее решению исходной задачи. Но для этого требуется вдохновение, знание предметной области и опыт работы со множеством различных примеров.

В этой главе мы представим несколько наглядных примеров алгоритмов. Некоторые заимствованы непосредственно из математики, которая является источником многих идей; другие основаны на опыте программирования. Первый пример – треугольник Серпинского – является графической иллюстрацией нашего принципа. Второй объясняет принцип «разделяй и властвуй» на простом математическом примере поиска корня функции. Затем мы покажем, как превратить эту идею в быстрый алгоритм поиска последовательностей, имеющей широкий круг применений. В третьем разделе мы коснемся «синтаксического анализа» последовательностей букв (1String), что также является распространенной задачей в программировании.

27.1. Фракталы, первое знакомство

Фракталы играют важную роль в вычислительной геометрии. Флейк (Flake) пишет в своей книге «The Computational Beauty of Nature» (MIT Press, 1998), что «...геометрию можно расширить и учесть в ней объекты с дробной размерностью. Такие объекты, известные как *фракталы*, по своему богатству отображения очень близки к разнообразию форм в природе. Фракталы обладают структурным самоподобием во многих ... масштабах, а это означает, что часть фрактала часто будет выглядеть как целое».

На рис. 21 показан пример фрактальной формы, известной как треугольник Серпинского. Основная форма – (равносторонний) треугольник, представленный в центре. Комбинируя этот треугольник достаточно много раз, следуя за треугольной формой, мы получим форму, изображенную слева.

Изображение справа на рис. 21 поясняет этап генерирования формы: чтобы получить данную форму, нужно найти середины всех сторон и соединить их друг с другом. В результате получится четыре треугольника. Этот процесс следует повторить для каждого из трех внешних треугольников, если они не слишком маленькие.

Альтернативное объяснение, хорошо подходящее для функций композиции форм, дается в библиотеке *2htdp/image*. Оно основано на переходе от изображения в центре к изображению справа. Поместив два треугольника рядом друг с другом и добавив третий над ними, мы тоже получим форму справа:

Этим решением мы обязаны Марку Смиту (Marc Smith).

```
> (s-triangle 3)

> (beside (s-triangle 3) (s-triangle 3))

> (above (s-triangle 3)
  (beside (s-triangle 3) (s-triangle 3)))

```

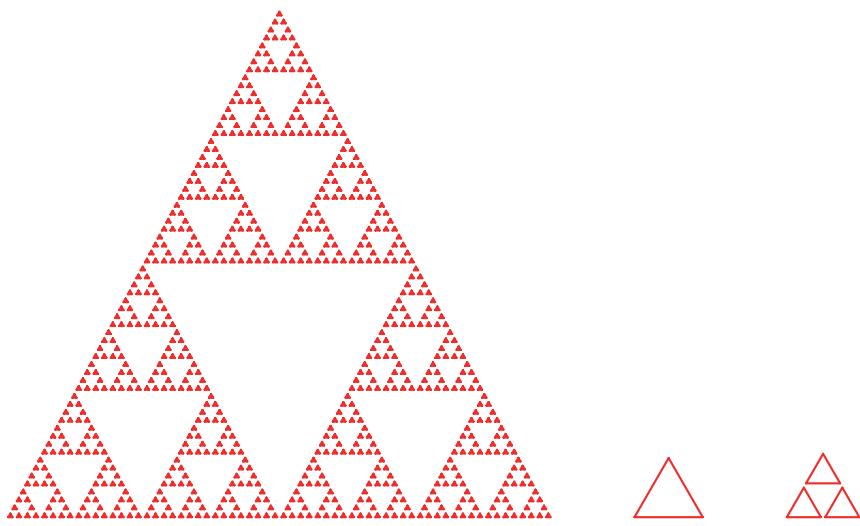


Рис. 21. Треугольник Серпинского

В этом разделе используется альтернативное описание для проектирования алгоритма Серпинского; оригинальное описание обсуждается в разделе 33.3. Учитывая, что цель состоит в том, чтобы сгенерировать изображение равностороннего треугольника, мы будем представлять задачу (положительным) числом – длиной стороны треугольника. Это наше решение приводит к следующим сигнатуре, описанию назначения и заголовку:

```
; Number -> Image
; создает треугольник Серпинского с размером стороны side

(define (sierpinski side)
  (triangle side 'outline 'red))
```

Теперь ответим на четыре вопроса генеративной рекурсии:

- когда заданное число настолько мало, что рисование дополнительных треугольников бессмысленно и случай можно считать тривиальным;

- в этом случае достаточно сгенерировать треугольник;
- в противном случае алгоритм должен сгенерировать треугольник Серпинского с размером $side/2$, потому что вдоль любой стороны требуется расположить по соседству два меньших треугольника;
- если `half-sized` – это треугольник Серпинского с размером $side/2$, то

```
| (above half-sized
  (beside half-sized half-sized))
```

является треугольником Серпинского с размером стороны $side$.

Дав ответы, мы легко можем определить функцию. Детали реализации показаны в листинге 105. Условие тривиальности проверяется выражением ($<=side$ SMALL), сравнивающим размер стороны `side` с некоторой константой `SMALL`. В тривиальном случае функция возвращает треугольник заданного размера. В рекурсивном случае выражение `local` вводит имя `half-sized` для треугольника Серпинского с размером стороны вдвое меньше заданного. После того как рекурсивный вызов генерирует треугольник Серпинского меньшего размера, полученное изображение копируется рядом и выше.

Листинг подчеркивает два других аспекта. Во-первых, описание назначения объясняет, **что** делает функция:

| ; создает треугольник Серпинского с размером стороны `side` ...

и **как** она это делает:

| ; ... генерируя один треугольник с размером стороны (/ `side` 2) и
| ; помещая одну копию над двумя другими

Листинг 105. Алгоритм Серпинского

```
(define SMALL 4) ; минимальный размер стороны треугольника в пикселях

(define small-triangle (triangle SMALL 'outline 'red))

; Number -> Image
; генеративно создает треугольник Серпинского с размером стороны side,
; генерируя один треугольник с размером стороны (/ side 2) и
; помещая одну копию над двумя другими

(check-expect (sierpinski SMALL) small-triangle)
(check-expect (sierpinski (* 2 SMALL))
              (above small-triangle
                    (beside small-triangle small-triangle)))

(define (sierpinski side)
  (cond
    [(<= side SMALL) (triangle side 'outline 'red)]
    [else
      (local ((define half-sized (sierpinski (/ side 2))))
        (above half-sized (beside half-sized half-sized))))]))
```

Во-вторых, примеры иллюстрируют два возможных случая: один, если заданный размер слишком мал, и один, если он достаточно велик. В последнем случае выражение, вычисляющее ожидаемое значение, объясняет смысл описания назначения.

Поскольку функция `sierpinsk` основана на генеративной рекурсии, определение и тестирование функции – это не последний шаг. Мы также должны проверить завершение алгоритма для любых допустимых входных данных. Входными данными для `sierpinsk` является одно положительное число. Если число меньше, чем `SMALL`, алгоритм завершается. В противном случае производится рекурсивный вызов, в который передается число, в два раза меньше заданного. То есть алгоритм должен завершаться для любых положительных значений `side`, если исходить из предположения, что константа `SMALL` тоже является положительным числом.

Процесс создания треугольников Серпинского можно представить как деление задачи пополам, пока не будет достигнут тривиальный случай, имеющий непосредственное решение. Добавив немного воображения, можно заметить, что этот процесс можно также использовать для поиска чисел с определенными свойствами. Эта идея подробно объясняется в следующем разделе.

27.2. Бинарный поиск

Математики, занимающиеся решением прикладных задач, моделируют реальный мир с помощью нелинейных уравнений, а затем пытаются решить их. В частности, они переводят задачу в функцию f , преобразующую одно число в другое, и ищут некоторое число r , такое что

$$f(r) = 0.$$

Значение r называется *корнем* функции f .

Вот постановка задачи на языке физики.

Задача. Ракета летит с постоянной скоростью v миль в час по прямой к какой-то цели, находящейся на расстоянии d_0 миль. Затем она начинает разгон с ускорением a миль в час за час. Когда она достигнет цели?

Физика говорит нам, что пройденное расстояние является функцией от времени:

$$d(t) = (v * t + 1/2 * a * t^2).$$

В постановке задачи предлагается определить момент времени t_0 , когда ракета достигнет цели:

$$d_0 = (v * t_0 + 1/2 * a * t_0^2).$$

Из алгебры мы знаем, что это квадратное уравнение, и такие уравнения имеют решение, если d_0 , a и v удовлетворяют определенным условиям.

Обычно подобные задачи намного сложнее, чем квадратные уравнения, поэтому математики в течение нескольких последних столетий разрабатывали методы поиска корней разных функций. В этом разделе мы исследуем решение, основанное на *теореме о промежуточном значении* (Intermediate Value Theorem, IVT), одном из базовых результатов математического анализа. Полученный алгоритм является простым примером реализации генеративной рекурсии на основе математической теоремы. Специалисты по информатике обобщили его в алгоритм *бинарного поиска* (алгоритм поиска методом дихотомии).

Теорема о промежуточном значении утверждает, что непрерывная функция f имеет корень в интервале $[a, b]$, если $f(a)$ и $f(b)$ находятся по разные стороны от оси X . Под *непрерывной* мы подразумеваем функцию, не имеющую «разрывов» и «гладко» изменяющуюся на протяжении всей области определения.

На рис. 22 показано графическое представление теоремы о промежуточном значении. Функция f – это непрерывная и гладкая функция. Ее график находится ниже оси X в точке a и выше в точке b и пересекает ось X где-то в этом интервале, отмечен как «диапазон 1» на рисунке.

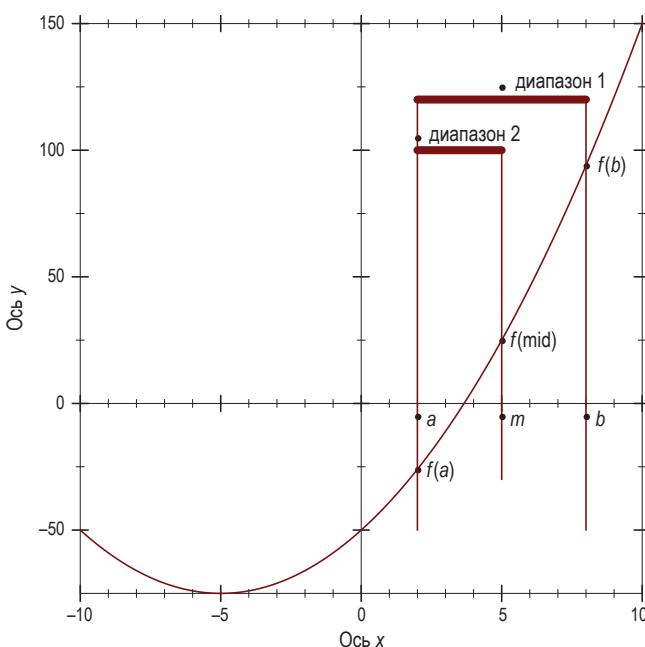


Рис. 22. Числовая функция f с корнем в интервале $[a, b]$ (шаг 1)

Теперь возьмем точку в середине интервала a и b :

$$m = (a + b)/2.$$

Она делит интервал $[a, b]$ на два меньших интервала одинакового размера. Теперь можно вычислить значение f от m и увидеть, находится ли оно ниже или выше оси X . В данном случае $f(m) > 0$, поэтому, согласно теореме о промежуточном значении, корень находится в левом интервале: $[a, m]$. Наш рисунок подтверждает это, где видно, что корень – точка пересечения графика функции с осью X – находится в левой половине интервала, отмеченной как «диапазон 2».

Итак, мы сформулировали описание ключевого шага в процессе поиска корня. Теперь переведем это описание в алгоритм на ISL+. Наша первая задача – сформулировать описание назначения. Очевидно, что алгоритм принимает функцию и границы интервала, в которых, как мы ожидаем, находится корень:

```
| ; [Number -> Number] Number Number -> ...
| (define (find-root f left right) ...)
```

Три параметра не могут быть любыми функциями и числами. Для `find-root` должны выполняться следующие условия:

```
| (or (<= (f left) 0 (f right))
|     (<= (f right) 0 (f left)))
```

то есть точки $(f \text{ left})$ и $(f \text{ right})$ должны находиться на противоположных концах оси X .

DrRacket позволяет использовать греческие символы, такие как ε . Но вы также можете использовать имя `EPSILON`.

Далее нам нужно зафиксировать результат функции и сформулировать описание назначения. Проще говоря, `find-root` находит интервал, содержащий корень. В процессе поиска она делит интервал, пока его размер $(- \text{right} \text{ left})$ не станет слишком маленьким, скажем меньше некоторой константы ε . В этом случае функция может вернуть один из трех результатов: левую границу, правую границу или представление интервала. Любое из этих значений полностью идентифицирует интервал, а так как нам проще вернуть число, будем возвращать левую границу. Вот полное определение заголовка:

```
| ; [Number -> Number] Number Number -> Number
| ; определяет R такое, что f имеет корень в [R, (+ R ε)],
| ; предполагается, что f -- непрерывная функция
| ; (2) (ог (<= (f left) 0 (f right)) (<= (f right) 0 (f left)))
| ; генеративно делит интервал пополам, корень находится в
| ; одной из двух половин, половина выбирается в соответствии с (2)
| (define (find-root f left right)
|   0)
```

Упражнение 445. Взгляните на следующее определение функции:

```
| ; Number -> Number
| (define (poly x)
|   (* (- x 2) (- x 4)))
```

Она определяет двучлен, корни которого можно определить вручную:

```
| > (poly 2)
0
| > (poly 4)
0
```

Используйте `poly`, чтобы сформулировать проверку `check-satisfied` для `find-root`.

Также используйте `poly`, чтобы проиллюстрировать процесс поиска корня. Начните с интервала $[3,6]$ и постройте таблицу, как показано ниже, $\varepsilon = 0$:

<i>шаг</i>	<i>left</i>	<i>f left</i>	<i>right</i>	<i>f right</i>	<i>mid</i>	<i>f mid</i>
<i>n</i> = 1	3	-1	6,00	8,00	4,50	1,25
<i>n</i> = 2	3	-1	4.50	1.25	?	?

Наша следующая задача состоит в том, чтобы ответить на четыре вопроса:

1. Нам нужно условие, описывающее момент, когда задачу можно считать решенной и имеется соответствующий ответ. Учитывая обсуждение выше, это просто:

```
| (<= (- right left) ε)
```

2. Соответствующий результат в тривиальном случае: `left`.
3. Для генеративного случая нам нужно выражение, генерирующее новые задачи для `find-root`. Согласно нашему неформальному описанию, этот шаг требует определения средней точки и значения функции в ней:

```
| (local ((define mid (/ (+ left right) 2))
         (define f@m (f mid)))
  ...)
```

Средняя точка используется для выбора следующего интервала. Согласно теореме о промежуточном значении, следующим кандидатом является интервал $[left, mid]$, если выполняется условие:

```
| (or (<= (f left) 0 f@m) (<= f@m 0 (f left)))
```

и $[mid, right]$, если выполняется условие:

```
| (or (<= f@m 0 (f right)) (<= (f right) 0 f@m))
```

То есть тело выражения `local` должно быть условным выражением:

```
| (cond
  [(or (<= (f left) 0 f@m) (<= f@m 0 (f left)))
```

```

(... (find-root f left mid) ...)]
[(or (<= f@m 0 (f right)) (<= (f right) 0 f@m))
 (... (find-root f mid right) ...)])

```

В обоих случаях мы используем `find-root`, чтобы продолжить поиск.

4. Ответ на последний вопрос очевиден. Поскольку рекурсивный вызов `find-root` находит корень f , то больше ничего делать не нужно.

Полная реализация функции показана в листинге 106. Следующие упражнения уточняют эту реализацию.

Листинг 106. Алгоритм `find-root`

```

; [Number -> Number] Number Number -> Number
; определяет R такое, что f имеет корень в [R,(+ R ε)],
; предполагается, что f -- непрерывная функция
; предполагается, что выполняется условие
; (ор (<= (f left) 0 (f right)) (<= (f right) 0 (f left)))
; генеративно делит интервал пополам, корень находится в
; одной из двух половин, половина выбирается в соответствии с предположением
(define (find-root f left right)
  (cond
    [(<= (- right left) ε) left]
    [else
      (local ((define mid (/ (+ left right) 2))
              (define f@mid (f mid)))
        (cond
          [(or (<= (f left) 0 f@mid) (<= f@mid 0 (f left)))
           (find-root f left mid)]
          [(or (<= f@mid 0 (f right)) (<= (f right) 0 f@mid))
           (find-root f mid right)]))))])

```

Упражнение 446. Добавьте тест из упражнения 445 в программу в листинге 106. Поэкспериментируйте с различными значениями ϵ . ■

Упражнение 447. Функция `poly` имеет два корня. Используйте `find-root` с `poly` и интервалом, содержащим оба корня. ■

Упражнение 448. Алгоритм `find-root` завершается для всех (непрерывных) f , $left$ и $right$, соответствующих предположениям. Почему? Сформулируйте аргумент завершения.

Подсказка. Предположите, что аргументы `find-root` описывают интервал размером s_1 . Насколько велико расстояние между $left$ и $right$ в первом и втором рекурсивном вызовах? После скольких шагов значение $(- right left)$ окажется меньше или равно ϵ ? ■

Упражнение 449. Как показано в листинге 106, чтобы сгенерировать следующий интервал, `find-root` дважды вычисляет значение f для каждой границы. Используйте локальные определения, чтобы избежать таких повторных вычислений.

Кроме того, `find-root` повторно вычисляет значение границы в рекурсивных вызовах. Например, `(find-root f left right)` вычисляет $(f left)$, и если в качестве следующего интервала выбирается $[left, mid]$, то снова вычисляет $(f left)$. Добавьте вспомогательную функцию, по-

хожую на `find-root`, которая на каждом рекурсивном шаге принимает не только `left` и `right`, но также `(f left)` и `(f right)`.

Сколько повторных вычислений `(f left)` такое решение позволяет сэкономить?

Примечание. Два дополнительных аргумента этой вспомогательной функции изменяются на каждом рекурсивном шаге, но изменение связано с изменением числовых аргументов. Эти аргументы являются так называемыми *аккумуляторами*, которые мы рассмотрим в части VI. ■

Упражнение 450. Функция `f` монотонно увеличивается, если всякий раз, когда выполняется условие `(< a b)`, выполняется и условие `(<= (f a) (f b))`. Упростите `find-root`, предположив, что заданная функция не только непрерывна, но и монотонно увеличивается. ■

Упражнение 451. Таблица является структурой с двумя полями: натуральное число `VL` и функция `аггау`, которая принимает натуральное число и для чисел между 0 и `VL` (не включая) возвращает соответствующий результат:

Многие языки программирования, включая Racket, поддерживают массивы и векторы, подобные таблицам.

```
(define-struct table [length array]
; Table -- это структура:
;   (make-table N [N -> Number])
```

Эта структура несколько необычна, поэтому очень важно проиллюстрировать ее примерами:

```
(define table1 (make-table 3 (lambda (i) i)))

; N -> Number
(define (a2 i)
  (if (= i 0)
      pi
      (error "table2 is not defined for i != 0")))

(define table2 (make-table 1 a2))
```

Здесь функция массива `table1` определена для большего количества входных данных, чем позволяет ее поле `length`; `table2` определена только для одного входа, а именно 0. Наконец, определим также вспомогательную функцию для поиска значений в таблицах:

```
; Table N -> Number
; отыскивает i-е значение в массиве t
(define (table-ref t i)
  ((table-array t) i))
```

Корнем таблицы `t` является число в `(table-array t)`, близкое к 0. *Корневой индекс* – это натуральное число `i` такое, что `(table-ref t i)` является корнем таблицы `t`. Таблица `t` монотонно увеличивается, если `(table-ref t 0)` меньше, чем `(table-ref t 1)`, `(table-ref t 1)` меньше, чем `(table-ref t 2)`, и т. д.

Спроектируйте функцию `find-linear`. Она должна принимать монотонно увеличивающуюся таблицу и отыскивать наименьший индекс

для корня таблицы. Используйте структурный рецепт для N , начиная от 0 до 1, 2 и т. д. до `length` данной таблицы. Такой процесс поиска корней часто называют *линейным поиском*.

Спроектируйте функцию `find-binary`, который точно так же отыскивает наименьший индекс для корня монотонно увеличивающейся таблицы, но использует для этого генеративную рекурсию. Как и обычный бинарный поиск, алгоритм должен сужать интервал до минимально возможного размера, а затем выбирать индекс. Не забудьте сформулировать аргумент завершения.

Подсказка. Ключевая проблема заключается в том, что индекс таблицы является не простым, а **натуральным** числом. Поэтому аргументы, определяющие границы интервала для поиска, должны быть натуральными числами. Подумайте, как это наблюдение меняет (1) природу тривиально решаемых задач, (2) порядок вычисления средней точки и (3) решение о том, какой интервал генерировать следующим. Для большей конкретики представьте таблицу с 1024 ячейками и с корнем в 1023-й ячейке. Сколько вызовов `find-linear` и `find-binary` потребуется, чтобы найти его? ■

27.3. Синтаксический анализ

В разных операционных системах используются разные соглашения, но для наших целей это не имеет значения.

Как упоминалось в главе 20, компьютеры хранят файлы, которые можно рассматривать как форму долговременной памяти. С нашей точки зрения *файл* – это просто список букв `1String`, прерываемый специальной строкой:

```
| ; File -- это одно из значений:
| ; -- '()
| ; -- (cons "\n" File)
| ; -- (cons 1String File)
| ; интерпретация: представляет содержимое файла
| ; "\n" -- это символ перевода строки
```

Идея состоит в том, что содержимое файлов разбивается на линии, где `\n` представляет так называемый символ перевода строки, отмечающий конец линии. Давайте также введем определение линии:

```
| ; Line -- это [List-of 1String].
```

Многие функции должны обрабатывать файлы как списки линий. Функция `read-lines` из библиотеки `2htdp/batch-io` – одна из них. В частности, эта функция преобразует содержимое следующего файла

```
| (list
|   "h" "o" "w" " " "a" "r" "e" " " "y" "o" "u" "\n"
|   "d" "o" "i" "n" "g" "?" "\n"
|   "a" "n" "y" " " "p" "r" "o" "g" "r" "e" "s" "s" "?")
```

в три линии:

```
(list
  (list "h" "o" "w" " " "a" "r" "e" " " "y" "o" "u")
  (list "d" "o" "i" "n" "g" "?")
  (list "a" "n" "y" " " "p" "r" "o" "g" "r" "e"
    "s" "s" "?"))
```

Аналогично, содержимому файла

```
| (list "a" "b" "c" "\n" "d" "e" "\n" "f" "g" "h" "\n")
```

тоже соответствует список с тремя линиями:

```
(list (list "a" "b" "c")
      (list "d" "e")
      (list "f" "g" "h"))
```

Стоп! Как можно представить в виде списка линий следующие три случая: '(), (список «\ n») и (список «\ n» «\ n»)? Почему эти примеры важны для тестирования?

Задача превращения последовательности 1String в список линий называется задачей *синтаксического анализа*, или *парсинга*. Многие языки программирования предоставляют функции, которые извлекают из файлов линии, слова, числа и другие виды так называемых токенов. Но даже при наличии таких функций программам часто требуется проводить дополнительный анализ этих токенов. В данном разделе мы поговорим о приемах синтаксического анализа, который настолько сложен и настолько важен для создания полноценных программ, что в во многих институтах, обучающих будущих специалистов в области информатики, имеется как минимум один курс, посвященный синтаксическому анализу. Поэтому не думайте, что вы сходу научитесь корректно решать реальные задачи синтаксического анализа после освоения этого раздела.

Начнем с самого начала: определим сигнатуру, описание назначения, примеры и заголовок для функции, которая превращает файл в список линий:

```
; File -> [List-of Line]
; преобразует файл в список линий

(check-expect (file->list-of-lines
                (list "a" "b" "c" "\n"
                      "d" "e" "\n"
                      "f" "g" "h" "\n"))
              (list (list "a" "b" "c")
                    (list "d" "e")
                    (list "f" "g" "h")))

(define (file->list-of-lines afile) '())
```

Затем опишем процесс синтаксического анализа, воспользовавшись нашим опытом, приобретенным в разделе 25.1.

Задача имеет тривиальное решение, если файл является пустым списком '().

1. В этом случае файл не содержит линий.
2. В противном случае в файле имеется хотя бы один символ "\n" или несколько букв 1String. Все элементы до первого символа "\n" включительно, если он имеется, должны быть отделены от остальной части файла. Остальное содержимое – это новая задача того же типа, которую может решить file->list-of-lines.
3. Затем достаточно объединить начальный сегмент – одну линию – со списком линий, полученным в результате обработки остальной части файла.

Четыре вопроса прямо определяют макет генеративно-рекурсивных функций. Очевидно, что для отделения начального сегмента от остальной части файла требуется просканировать список букв 1String неопределенной длины, поэтому добавим в список желаний две вспомогательные функции: first-line, которая возвращает все буквы 1String до первого символа "\n" (не включая его) или конец списка, и remove-first-line, которая удаляет элементы, выбранные функцией first-line.

Теперь, учитывая эти функции, написать остальную часть программы будет совсем несложно. Первое предложение в file->list-of-lines должно возвращать '(), потому что пустой файл не содержит никаких линий. Второе предложение должно объединить значение (first-line afile) со значением (file->list-of-lines (remove-first-line afile)), потому что первое выражение возвращает первую линию, а второе – все остальные. Наконец, вспомогательные функции просматривают свои входные данные структурно-рекурсивным способом, поэтому их разработку мы оставляем вам как самостоятельное упражнение. Полный код программы показан в листинге 107.

Листинг 107. Преобразование файла в список линий

```

; File -> [List-of Line]
; преобразует файл в список линий
(define (file->list-of-lines afile)
  (cond
    [(empty? afile) '()]
    [else
      (cons (first-line afile)
            (file->list-of-lines (remove-first-line afile)))]))

; File -> Line

(define (first-line afile)
  (cond
    [(empty? afile) '()]
    [(string=? (first afile) NEWLINE) '()]
    [else (cons (first afile) (first-line (rest afile))))]))

; File -> Line

(define (remove-first-line afile)
  (cond
    [(empty? afile) '()]
    [(string=? (first afile) NEWLINE) (rest afile)]))

```

```
[else (remove-first-line (rest afile)))]))

(define NEWLINE "\n") ; the 1String
Вот как file->list-of-lines выполняет второй тест:
(file->list-of-lines
 (list "a" "b" "c" "\n" "d" "e" "\n" "f" "g" "h" "\n"))
 ==
(cons
 (list "a" "b" "c")
 (file->list-of-lines
  (list "d" "e" "\n" "f" "g" "h" "\n")))
 ==
(cons
 (list "a" "b" "c")
 (cons (list "d" "e")
   (file->list-of-lines
    (list "f" "g" "h" "\n"))))
 ==
(cons (list "a" "b" "c")
  (cons (list "d" "e")
    (cons (list "f" "g" "h")
      (file->list-of-lines '())))))
 ==
(cons (list "a" "b" "c")
  (cons (list "d" "e")
    (cons (list "f" "g" "h")
      '())))
)
```

Этот порядок вычислений еще раз напоминает о том, что аргумент рекурсивного применения `file->list-of-lines` почти никогда не является остальной частью заданного файла. Он также показывает, почему в данном случае генеративная рекурсия гарантированно завершается для любого файла. Каждое рекурсивное применение получает список короче заданного, а это означает, что рекурсивный процесс останавливается по достижении '()'.

Упражнение 452. В определениях обеих функций, `first-line` и `remove-first-line`, отсутствуют описания их назначения. Сформулируйте их. ■

Упражнение 453. Спроектируйте функцию `tokenize`. Она должна преобразовывать экземпляр `Line` в список токенов. Под токеном в этом упражнении подразумеваются буквы `1String` или строки `String`, состоящие только из букв нижнего регистра. То есть из исходной строки следует отбросить все пробельные символы и знаки препинания. Остальные буквы остаются как есть, и все буквы, следующие друг за другом, объединяются в «слова». **Подсказка.** Прочитайте описание функции `string-whitespace?`. ■

Упражнение 454. Спроектируйте функцию `create-matrix`. Она должна принимать число n и список из n^2 чисел и возвращать матрицу $n \times n$, например:

```
(check-expect
 (create-matrix 2 (list 1 2 3 4))
 (list (list 1 2)
       (list 3 4)))
```

Добавьте еще один пример. ■

28. Математические примеры

Во многих решениях математических задач используется генеративная рекурсия. Будущий программист обязательно должен познакомиться с такими решениями, потому что, с одной стороны, огромное количество задач по программированию связано с превращением таких математических идей в программы, а с другой – практика решения подобных математических задач часто оказывается весьма полезной для разработки алгоритмов. В этой главе мы рассмотрим три такие задачи.

28.1. Метод Ньютона

В разделе 27.2 был представлен один из методов нахождения корня математической функции. Как показали упражнения в том разделе, этот метод естественным образом обобщается на вычислительные задачи, такие как поиск определенных значений в таблицах, векторах и массивах. В математических приложениях программисты склонны использовать методы, зародившиеся в математическом анализе. Один из самых известных методов был предложен Ньютоном. Подобно бинарному поиску, *метод Ньютона* постепенно приближается к корню, пока результат не станет «достаточно близким». Суть метода состоит в следующем: начиная с предположения, скажем r_1 , строится касательная к графику функции f в точке r_1 , и определяется ее корень. Да, касательная – это всего лишь аппроксимация функции, зато определять ее корень проще некуда. Повторяя этот процесс, Ньютон доказал этот факт. Алгоритм может найти корень r , для которого $(f' r)$ достаточно близко к 0.

Очевидно, что этот алгоритм основан на двух предметных знаниях о касательных: их наклонах и корнях. Неформально касательная к f в некоторой точке r_1 – это прямая, которая проходит через точку $(r_1, f(r_1))$ и имеет тот же наклон, что и f . Один из способов математически определить наклон касательной состоит в том, чтобы выбрать две близкие точки на оси X, равноудаленные от r_1 , и в качестве наклона f использовать наклон прямой, проходящей через эти точки. По соглашению нужно выбрать небольшое число ε и работать с точками $r_1 + \varepsilon$ и $r_1 - \varepsilon$. То есть наклон прямой определяется по точкам $(r_1 - \varepsilon, f(r_1 - \varepsilon))$ и $(r_1 + \varepsilon, f(r_1 + \varepsilon))$:

$$\text{slope}(f, r_1) = \frac{f(r_1 + \varepsilon) - f(r_1 - \varepsilon)}{(r_1 + \varepsilon) - (r_1 - \varepsilon)} = \frac{1}{2 \cdot \varepsilon} \cdot (f(r_1 + \varepsilon) - f(r_1 - \varepsilon)).$$

Упражнение 455. Переведите эту математическую формулу в функцию `slope` на языке ISL+, которая отображает функцию `f` и число `r1` в наклон функции `f` в точке `r1`. Исходите из предположения, что `\varepsilon` – это глобальная константа. Для примеров используйте функции,

точный наклон которых можно вычислить вручную, скажем горизонтальные линии, линейные функции и, возможно, полиномы, если вы знаете, как определяется их наклон.

Вторая часть предметных знаний касается корня касательной, которая является простой прямой, то есть линейной функцией. Касательная проходит через точку $(r_1, f(r_1))$ и имеет указанный выше наклон. Математически это определяется так:

$$\text{tangent}(x) = \text{slope}(f, r_1) \cdot (x - r_1) + f(r_1).$$

Нахождение корня касательной *tangent* означает нахождение такого значения *root-of-tangent*, при котором *tangent(root-of-tangent)* равно 0:

$$0 = \text{slope}(f, r_1) \cdot (\text{root-of-tangent} - r_1) + f(r_1).$$

Вот как можно решить это уравнение:

$$\text{root-of-tangent} = r_1 - \frac{f(r_1)}{\text{slope}(f, r_1)}.$$

Упражнение 456. Спроектируйте функцию *root-of-tangent*, которая отображает *f* и *r1* в корень касательной, проходящей через $(r_1, (f \ r1))$. ■

Теперь мы можем воспользоваться рецептом проектирования и перевести описание алгоритма Ньютона в программу на языке ISL+. Функция – назовем ее *newton* в честь изобретателя – принимает функцию *f* и число *r1*:

```
; [Number -> Number] Number -> Number
; отыскивает такое число r, при котором (f r) достаточно близко к нулю
; генеративно создает все более близкие приближения
(define (newton f r1) 1.0)
```

Чтобы создать макет *newton*, обратимся к четырем основным вопросам рецепта проектирования генеративной рекурсии.

- Если значение $(f \ r1)$ достаточно близко к 0, то задача считается решенной. Близость к 0 может означать, что $(f \ r1)$ является небольшим положительным или отрицательным числом. Следовательно, мы должны проверить его абсолютное значение:

```
| (<= (abs (f r1)) ε)
```

- Решением является *r1*.
- Генеративный шаг алгоритма заключается в нахождении корня касательной к *f* в точке *r1*, который и является следующим приближением. Применяя *newton* к *f* и этому новому приближению, мы продолжаем процесс.
- Результат рекурсивного вызова является ответом на исходную задачу.

Листинг 108. Алгоритм Ньютона

```

; [Number -> Number] Number -> Number
; отыскивает такое число  $r$ , при котором выполняется условие ( $\leq (\text{abs} (f r)) \varepsilon$ )
; check-within (newton poly 1) 2  $\varepsilon$ 
; check-within (newton poly 3.5) 4  $\varepsilon$ 

(define (newton f r1)
  (cond
    [(<= (abs (f r1))  $\varepsilon$ ) r1]
    [else (newton f (root-of-tangent f r1))]))

; см. упражнение 455
(define (slope f r) ...)

; см. упражнение 456
(define (root-of-tangent f r) ...)

```

В листинге 108 показано полное определение newton. Оно включает два теста, которые основаны на тестах в разделе 27.2 для find-root. В конце концов, обе функции ищут корень, а poly имеет два известных корня.

Но мы еще не закончили проектирование newton. Новый, седьмой шаг рецепта проектирования требует исследовать, как завершается функция. Функция newton имеет проблему при применении к таким функциям, как poly:

```

; Number -> Number
(define (poly x) (* (- x 2) (- x 4)))

```

Как уже упоминалось, эта функция имеет два корня: 2 и 4. График poly, изображенный на рис. 23, подтверждает это, а также показывает, что между двумя корнями график функции меняет наклон. С учетом всего вышесказанного у математика возникает вопрос: что вернет newton для начального приближения, равного 3:

```

> (poly 3)
-1
> (newton poly 3)
/:division by zero

```

Функция poly возвращает «плохое» значение, а root-of-poly превращает его в ошибку:

```

> (slope poly 3)
0
> (root-of-tangent poly 3)
/:division by zero

```

Помимо этой ошибки времени выполнения, newton имеет еще две проблемы, связанные с завершением. К счастью, обе можно продемонстрировать с помощью poly. Первая проблема касается природы чисел, которую мы кратко затронули в разделе 1.1. Во многих упраж-

нениях по программированию для начинающих можно без опаски игнорировать различия между точными и неточными числами, но когда дело доходит до реализации математических формул, нужно действовать с особой осторожностью. Рассмотрим следующее выражение:

```
| > (newton poly 2.9999)
```

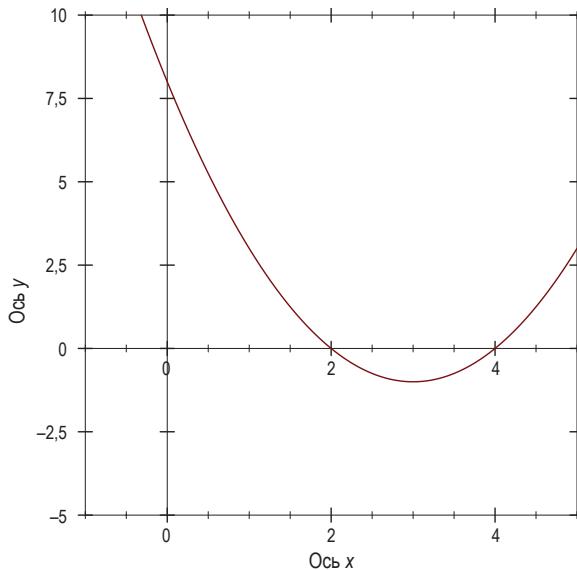


Рис. 23. График функции `poly` в интервале $[-1,5]$

Программа на ISL+ интерпретирует 2.9999 как точное число, и операции в `newton` обрабатывают его соответствующим образом, но так как это число не является целым, в вычислениях используется точная рациональная дробь. Поскольку арифметика дробей выполняется намного медленнее, чем арифметика неточных чисел, приведенный выше вызов функции в DrRacket работает довольно долго. В зависимости от быстродействия вашего компьютера на вычисления может потребоваться от нескольких секунд до минуты или больше. Если вы случайно выберете другое число, то может показаться, что функция `newton` вообще «зависла».

Вторая проблема касается невозможности завершения. Например:

```
| > (newton poly #i3.0)
```

Здесь в качестве начального приближения используется неточное число `#i3.0`, которое, в отличие от 3, вызывает другую проблему. В частности, функция `slope` теперь возвращает неточный 0 для `poly`, и `root-of-tangent` переходит в бесконечность:

```
| > (slope poly #i3.0)
#i0.0
```

```
| > (root-of-tangent poly #i3.0)
| #i+inf.0
```

Вычисления в newton преращают `#i+inf.0` в `+nan.0` – элемент данных, который «не является числом». Большинство арифметических операций просто передают это значение дальше, что объясняет зацикливание newton.

Как результат вычисления зацикливаются.

Проще говоря, newton демонстрирует целый спектр проблем, когда дело доходит до сложного поведения завершения. Для одних входных данных функция дает правильный результат. Для других сообщает об ошибке. А с третьими входит в бесконечный цикл или кажется, что входит в него. Заголовок newton – и любое другое ее описание – должен предупреждать желающих использовать функцию об этих сложностях, и такие предупреждения имеются во всех хороших математических библиотеках на распространенных языках программирования.

Это упражнение предложил Адриан Герман (Adrian German).

Упражнение 457. Спроектируйте функцию `double-amount`, которая вычисляет, сколько месяцев потребуется, чтобы удвоить заданную сумму денег, с учетом фиксированной месячной процентной ставки сберегательного счета.

Знание предметной области. С помощью простых алгебраических манипуляций можно показать, что величина исходной суммы не имеет значения. Важна только процентная ставка. Также эксперты в предметной области знают, что удвоение происходит примерно через $72/r$ месяца, если процентная ставка r «мала». ■

28.2. Интегрирование

Многие физические задачи сводятся к определению площади под кривой:

Постановка задачи. Автомобиль движется с постоянной скоростью v м в секунду. Какое расстояние он проедет за 5, 10, 15 с?

Ракета взлетает с постоянным ускорением 12 м/с². Какой высоты она достигает через 5, 10, 15 с?

Физик скажет вам, что автомобиль проезжает $d_{con}(t) = v \cdot t$ м при движении с постоянной скоростью v в течение t с. Если автомобиль или ракета постоянно ускоряется, то пройденное расстояние зависит от квадрата прошедшего времени t :

$$d_{acc}(t) = \frac{1}{2} \cdot a \cdot t^2.$$

В общем случае закон говорит нам, что пройденное расстояние пропорционально площади под кривой графика изменения скорости $v(t)$ с течением времени t .

Рисунок 24 иллюстрирует эту идею графически. Слева мы видим наложение двух графиков: сплошная горизонтальная линия – это скорость автомобиля, а восходящая пунктирная линия – это прой-

денное расстояние. Быстрая проверка показывает, что расстояние действительно является площадью области между линией скорости и осью абсцисс **в каждый момент времени**. Точно так же графики справа показывают взаимосвязь между постоянным ускорением взлетающей ракеты и высотой. Определение площади области под графиком функции для некоторого конкретного интервала называется *интегрированием* (функции).

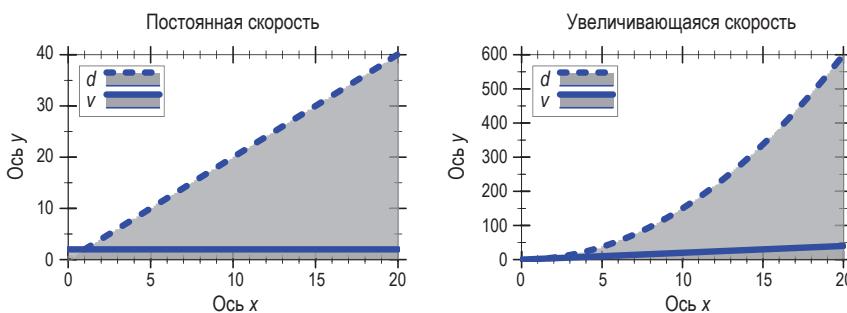


Рис. 24. Расстояние, пройденное с постоянной скоростью и с ускорением

В математике известны формулы для этих двух задач, дающие точные ответы, но в общем случае необходимо использовать численные решения. Проблема в том, что кривые часто имеют сложную форму, больше похожую на те, что показаны на рис. 25, и кому-то может понадобиться узнать площадь области между осью X, вертикальными линиями a и b и графиком f . Прикладные математики определяют

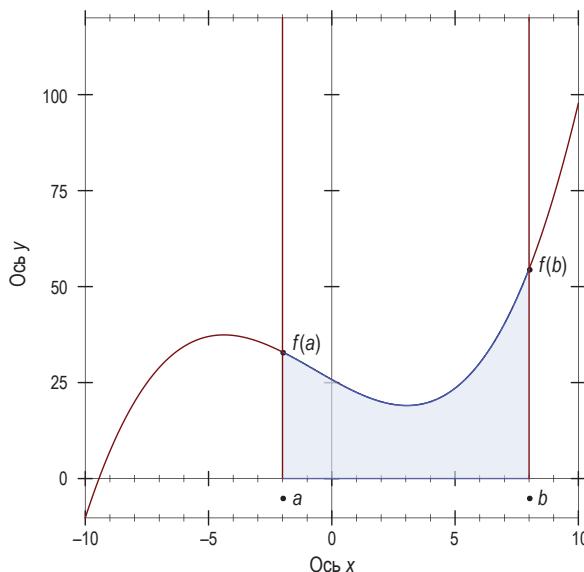


Рис. 25. Интегрирование функции f в интервале между a и b

площади таких областей приблизительно, суммируя площади множества небольших геометрических фигур. Поэтому было бы естественно разработать алгоритмы для таких вычислений.

Алгоритм интегрирования принимает три параметра: функцию f и две границы, а и b . Четвертый параметр – ось X – подразумевается. Из этого вытекает следующая сигнатура:

```
| ; [Number -> Number] Number Number -> Number
```

Понять идею интегрирования проще всего на простых примерах, таких как постоянная или линейная функция. Итак, рассмотрим функцию:

```
| (define (constant x) 20)
```

Функция `constant` вместе с границами 12 и 22 описывает прямоугольник шириной 10 и высотой 20 единиц. Площадь этого прямоугольника равна 200, то есть мы получаем следующий тест:

```
| (check-expect (integrate constant 12 22) 200)
```

Теперь определим линейную функцию и второй тест:

```
| (define (linear x) (* 2 x))
```

Функция `linear` с границами 0 и 10 описывает треугольную область с шириной основания 10 и высотой 20 единиц. Отсюда вытекает следующий тест:

```
| (check-expect (integrate linear 0 10) 100)
```

В конце концов, площадь треугольника равна половине произведения его основания на высоту, проведенную к основанию.

Для третьего примера используем некоторые предметные знания. Как уже упоминалось, математики знают, как точно определить площадь некоторых функций. Например, область под функцией

$$\text{square}(x) = 3 \cdot x^2$$

на интервале $[a, b]$ вычисляется по следующей формуле:

$$b^3 - a^3.$$

А вот так эта идея выглядит в виде конкретного теста:

```
| (define (square x) (* 3 (sqr x)))
| 
| (check-expect (integrate square 0 10)
|                 (- (expt 10 3) (expt 0 3)))
```

Листинг 109. Обобщенная функция интегрирования

```
| (define ε 0.1)
| ; [Number -> Number] Number Number -> Number
```

```

; вычисляет площадь под графиком функции f на интервале от a до b
; предполагается, что (< a b)

(check-within (integrate (lambda (x) 20) 12 22) 200 ε)
(check-within (integrate (lambda (x) (* 2 x)) 0 10) 100 ε)
(check-within (integrate (lambda (x) (* 3 (sqr x))) 0 10)
  1000
  ε)

(define (integrate f a b) #i0.0)

```

В листинге 109 показан результат выполнения первых трех шагов рецепта проектирования. В него добавлено заявление о назначении и очевидное предположение относительно границ интервала. Вместо `check-expect` здесь используется `check-within`, которая учитывает неточности, неизбежные в подобных вычислениях с приближениями. Аналогично заголовок `integrate` определяет `#i0.0` в качестве возвращаемого результата, сообщая, что функция предполагает возврат неточного числа.

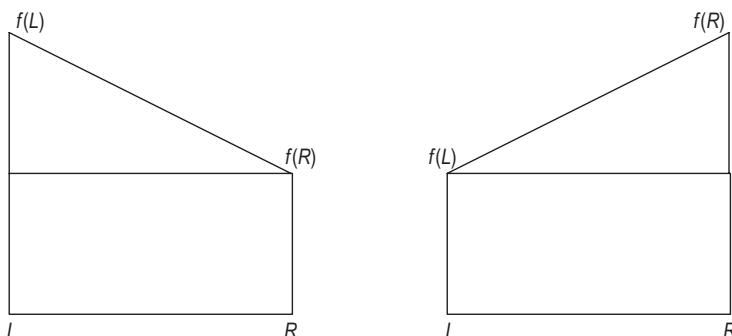
Следующие два упражнения показывают, как превратить предметные знания в функции интегрирования. Обе функции вычисляют довольно грубые приближения. Но первая основывается только на математических формулах, а во вторую заложены некоторые идеи структурного проектирования. Решение этих упражнений даст вам необходимое понимание сути этого раздела, представляющего генеративно-рекурсивный алгоритм интегрирования.

Упражнение 458. Кеплер предложил простой метод интегрирования. Чтобы вычислить оценку площади под f на интервале между a и b , необходимо:

Этот метод известен
как правило Кеппера.

- 1) разделить интервал пополам $mid = (a + b)/2$;
- 2) вычислить площади этих двух трапеций:
 - $[(a, 0), (a, f(a)), (mid, 0), (mid, f(mid))]$;
 - $[(mid, 0), (mid, f(mid)), (b, 0), (b, f(b))]$;
- 3) сложить эти две площади.

Знание предметной области. Давайте взглянем на эти трапеции. Вот две возможные формы:



Форма слева предполагает, что $f(L) > f(R)$, а форма справа показывает случай, когда $f(L) < f(R)$. Несмотря на явную асимметрию, площади этих трапеций можно вычислить по одной и той же формуле:

$$[(R - L) \cdot f(R)] + \left[\frac{1}{2} \cdot (R - L) \cdot (f(L) - f(R)) \right].$$

Стоп! Убедитесь, что для левой трапеции эта формула складывает площадь треугольника с площадью прямоугольника под ним, а для правой – вычитает площадь треугольника из площади большого, описывающего прямоугольника.

Также покажите, что приведенная выше формула равна

$$\frac{1}{2} \cdot (R - L) \cdot (f(L) + f(R)).$$

Это математическое подтверждение асимметрии формулы.

Спроектируйте функцию `integrate-kepler`. То есть преобразуйте математические знания в функцию на ISL+. Адаптируйте тесты из листинга 109 для этого случая. Какой из трех тестов терпит неудачу, и как часто? ■

Упражнение 459. Другой простой метод интегрирования заключается в делении области на множество маленьких прямоугольников. Каждый прямоугольник имеет фиксированную ширину и высоту, соответствующую высоте графика функции в середине верхней стороны прямоугольника. Сумма площадей прямоугольников дает оценку площади под графиком функции.

Используем

$$R = 10$$

для обозначения количества прямоугольников. Исходя из этого, ширина каждого прямоугольника равна:

$$W = (b - a)/R.$$

Высота одного из этих прямоугольников равна значению f в середине верхней стороны прямоугольника. Первая средняя точка равна a плюс половина ширины прямоугольника:

$$S = width/2,$$

отсюда площадь равна:

$$W \cdot f(a + S).$$

Чтобы вычислить площадь второго прямоугольника, нужно прибавить ширину прямоугольника к первой средней точке:

$$W \cdot f(a + W + S).$$

Площадь третьего прямоугольника равна:

$$W \cdot f(a + 2 \cdot W + S).$$

В общем случае площадь i -го прямоугольника вычисляется по формуле:

$$W \cdot f(a + i \cdot W + S).$$

Первый прямоугольник имеет индекс 0, а последний $R - 1$.

Используя эти прямоугольники, можно определить площадь под графиком:

$$\sum_{i=0}^{i=R-1} W \cdot f(a + i \cdot W + S) = W \cdot f(a + 0 \cdot W + S) + \dots + W \cdot f(a + (R - 1) \cdot W + S).$$

Преобразуйте это описание процесса в функцию на ISL+. Адаптируйте тесты из листинга 109 для этого случая.

Чем больше прямоугольников использует алгоритм, тем ближе его оценка к фактической площади. Определите R как константу верхнего уровня и увеличивайте ее в 10 раз, пока точность алгоритма не достигнет $\varepsilon = 0,1$.

Уменьшите ε до 0,01 и увеличивайте R до тех пор, пока все тесты не будут выполняться успешно. Сравните результат с упражнением 458. ■

Метод Кеплера в упражнении 458 предлагает стратегию «разделяй и властвуй», подобно бинарному поиску, представленному в разделе 27.2. Проще говоря, алгоритм делит интервал на две части, рекурсивно вычисляет площадь каждой из них и складывает два результата.

Упражнение 460. Разработайте алгоритм `integrate-dc`, который интегрирует функцию f в интервале между a и b , используя стратегию «разделяй и властвуй». Используйте метод Кеплера, когда интервал достаточно мал. ■

Подход «разделяй и властвуй» в упражнении 460 довольно расточителен. Рассмотрим, например, функцию, график которой в одной части имеет вид горизонтальной линии, а в другой приобретает пилообразную форму (см. пример на рис. 26). В той части, где график имеет форму горизонтальной линии, разбивать его на прямоугольники бессмысленно. Мы легко можем вычислить площадь трапеции для всего интервала. Однако для «пилообразной» части алгоритм должен продолжать делить интервал до тех пор, пока неровности графика не станут достаточно маленькими.

Чтобы определить, когда график f имеет вид горизонтальной прямой, можно изменить алгоритм следующим образом. Вместо простой проверки величины интервала новый алгоритм вычисляет площадь

трех трапеций: для всего интервала и для двух его половин. Если разность площади первой трапеции и суммы площадей двух вторых трапеций меньше площади маленького прямоугольника с высотой ε и шириной $b - a$, то можно с уверенностью предположить, что площадь трапеции, охватывающей весь интервал, является хорошим приближением. Иначе говоря, алгоритм определяет, насколько сильно изменяется f на интервале от a до b и влияет ли это изменение на допустимую погрешность. Если функция изменяется существенно, то алгоритм продолжает придерживаться подхода «разделяй и властвуй»; в противном случае останавливается и использует приближение Кеплера.

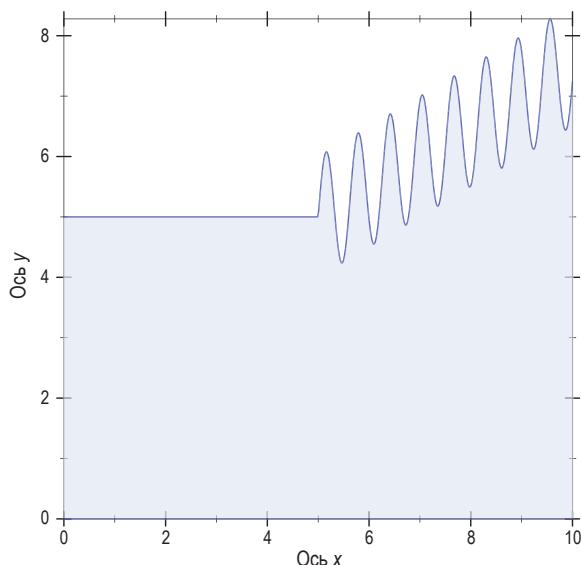


Рис. 26. Пример функции для разработки алгоритма адаптивного интегрирования

Упражнение 461. Спроектируйте функцию `integrate-adaptive`. То есть превратите описание рекурсивного процесса в алгоритм на ISL+. Обязательно адаптируйте тестовые примеры из листинга 109 для этого случая.

Не вдавайтесь пока в рассуждения о завершении `integrate-adaptive`.

Всегда ли `integrate-adaptive` дает более точный результат, чем `integrate-kepler` или `integrate-rectangles`? Какая характеристика `integrate-adaptive` улучшилась? ■

Терминология. Алгоритм называется *адаптивным интегрированием*, потому что он автоматически выделяет время для вычисления площадей в тех частях графика, где это действительно необходимо, и тратит мало времени на другие части. То есть для участков с пологим графиком f алгоритм выполняет всего несколько вычисле-

ний; для других частей он постепенно уменьшает интервалы, чтобы уменьшить погрешность вычислений. Информатика знает множество адаптивных алгоритмов, и адаптивное интегрирование – лишь один из них.

28.3. Проект: гауссово исключение

Математики не только ищут решения уравнений с одной переменной; они также изучают целые системы линейных уравнений:

Постановка задачи. В мире бартера стоимости угля (x), нефти (y) и газа (z) определяются следующими уравнениями обмена:

$$\begin{aligned} 2 \cdot x + 2 \cdot y + 3 \cdot z &= 10 \\ 2 \cdot x + 5 \cdot y + 12 \cdot z &= 31 \quad (\dagger) \\ 4 \cdot x + 1 \cdot y - 2 \cdot z &= 1 \end{aligned}$$

Решение подобной системы уравнений состоит из набора таких чисел, по одному на переменную, что если заменить переменную соответствующим ей числом, то две стороны каждого уравнения дадут одно и то же число. Наш текущий пример имеет следующее решение:

$$x = 1, y = 1 \text{ и } z = 2.$$

Мы легко можем проверить это утверждение:

$$\begin{aligned} 2 \cdot 1 + 2 \cdot 1 + 3 \cdot 2 &= 10 \\ 2 \cdot 1 + 5 \cdot 1 + 12 \cdot 2 &= 31 \\ 4 \cdot 1 + 1 \cdot 1 + 2 \cdot 2 &= 1 \end{aligned}$$

Эти три уравнения можно сократить до:

$$10 = 10, 31 = 31 \text{ и } 1 = 1.$$

В листинге 110 показано определение представления данных для нашей предметной области. Оно включает пример системы уравнений и ее решения. Это представление отражает суть системы уравнений, а именно числовые коэффициенты переменных в левой и правой частях. Имена переменных не играют никакой роли, потому что они подобны параметрам функций; то есть при непротиворечивом переименовании уравнения имеют одни и те же решения.

Листинг 110. Представление данных для системы уравнений

```
; SOE - это непустая матрица Matrix.
; ограничения: для (list r1 ... rn), (length ri) равна (+ n 1)
; интерпретация: представляет систему линейных уравнений

; Equation - это [List-of Number].
; ограничения: Equation содержит как минимум два числа.
; интерпретация: если (list a1 ... an b) является уравнением Equation,
; то a1, ..., an - это набор коэффициентов при переменных слева
```

```

; a b - это правая сторона
; Solution - это [List-of Number]
(define M ; an SOE
  (list (list 2 2 3 10) ; Equation
        (list 2 5 12 31)
        (list 4 1 -2 1)))

(define S '(1 1 2)) ; Solution

```

В оставшейся части этого раздела мы будем использовать следующие функции:

```

; Equation -> [List-of Number]
; извлекает левую часть из строки в матрице
(check-expect (lhs (first M)) '(2 2 3))
(define (lhs e)
  (reverse (rest (reverse e)))))

; Equation -> Number
; извлекает правую часть из строки в матрице
(check-expect (rhs (first M)) 10)
(define (rhs e)
  (first (reverse e)))

```

Упражнение 462. Спроектируйте функцию `check-solution`. Она должна принимать SOE и Solution и возвращать `#true`, если подстановка значений из решения Solution вместо переменных в уравнения Equation в SOE дает одинаковые значения в левой и правой частях; в противном случае функция должна вернуть `#false`. Используйте `check-solution`, чтобы сформулировать тесты с `check-satisfied`.

Подсказка. Сначала спроектируйте функцию `plug-in`, которая принимает левую часть уравнения и решение и вычисляет значение левой части, подставляя числа из решения. ■

Гауссово исключение – это стандартный метод поиска решений систем линейных уравнений. Он состоит из двух шагов. Первый шаг – преобразование системы уравнений в систему другой формы, но с тем же решением. Второй шаг – поиск решений уравнений по одному. Здесь мы сосредоточимся на первом шаге, потому что это еще один интересный пример генеративной рекурсии.

Первый шаг алгоритма гауссова исключения называется «триангуляцией», потому что в результате получается система уравнений в форме треугольника. Исходная система, напротив, имеет вид прямогоугольника. Чтобы понять идею, лежащую в основе этой терминологии, рассмотрим список, представляющий исходную систему:

```

(list (list 2 2 3 10)
      (list 2 5 12 31)
      (list 4 1 -2 1))

```

Триангуляция преобразует эту матрицу, как показано ниже:

```

(list (list 2 2 3 10)
      (list 3 9 21)
      (list 1 2))

```

Как уже отмечалось, форма этой системы уравнений имеет вид треугольника.

Упражнение 463. Убедитесь, что следующая система уравнений

$$\begin{aligned} 2 \cdot x + 2 \cdot y + 3 \cdot z &= 10 \\ 3 \cdot y + 9 \cdot z &= 21 \quad (*) \\ 1 \cdot z &= 2 \end{aligned}$$

имеет то же решение, что и система, отмеченная значком (\dagger). Выполните проверку сначала вручную, а потом с помощью `check-solution` из упражнения 462. ■

Ключевая идея триангуляции состоит в том, чтобы вычесть первое уравнение из остальных. Вычитание одного уравнения из другого означает вычитание соответствующих коэффициентов в обоих уравнениях. В нашем текущем примере вычитание первого уравнения из второго дает следующую матрицу:

```
(list (list 2 2 3 10)
      (list 0 3 9 21)
      (list 4 1 -2 1))
```

Цель этого вычитания состоит в том, чтобы во всех уравнениях, кроме первого, заменить первый коэффициент нулем. Чтобы обнулить первый коэффициент в третьем уравнении, нужно **дважды** вычесть первое уравнение из третьего:

```
(list (list 2 2 3 10)
      (list 0 3 9 21)
      (list 0 -3 -8 -19))
```

Следуя соглашениям, мы отбрасываем члены с нулевыми коэффициентами в последних двух уравнениях:

```
(list (list 2 2 3 10)
      (list 3 9 21)
      (list -3 -8 -19))
```

То есть мы сначала умножили все коэффициенты в первой строке на 2, а затем вычли результат из последней строки. Как уже упоминалось, такие вычитания не меняют решения; то есть решение исходной системы также является решением преобразованной.

Математика учит,
как доказывать такие
факты. Мы же просто
используем их.

Упражнение 464. Убедитесь, что следующая система уравнений

$$\begin{aligned} 2 \cdot x + 2 \cdot y + 3 \cdot z &= 10 \\ 3 \cdot y + 9 \cdot z &= 21 \quad (\ddagger) \\ -3 \cdot y - 8 \cdot z &= 19 \end{aligned}$$

имеет то же решение, что и отмеченное значком (\dagger). Выполните проверку сначала вручную, а потом с помощью `check-solution` из упражнения 462. ■

Упражнение 465. Спроектируйте функцию `subtract`. Она должна принимать два уравнения одинаковой длины и «вычесть» второе уравнение из первого, элемент за элементом, пока в получившемся уравнении не обнулится первый коэффициент. Поскольку известно, что первый коэффициент равен 0, `subtract` должна вернуть оставшуюся часть списка, полученного в результате вычитаний. ■

Теперь рассмотрим остальную часть экземпляра SOE:

```
| (list (list 3 9 21)
      (list -3 -8 -19))
```

Это тоже экземпляр SOE, поэтому мы можем снова применить тот же алгоритм. В нашем текущем примере требуется вычесть первое уравнение из второго -1 раз. Это дает

```
| (list (list 3 9 21)
      (list 1 2))
```

Остатком этого экземпляра SOE является единственное уравнение, поэтому его нельзя упростить.

Упражнение 466. Вот представление для решения системы уравнений SOE методом триангуляции:

```
| ; TM - это [NEList-of Equation]
| ; такой, что каждое следующее уравнение Equation короче предыдущего:
| ; n + 1, n, n - 1, ..., 2.
| ; интерпретация: представляет треугольную матрицу
```

Спроектируйте алгоритм триангуляции:

```
| ; SOE -> TM
| ; триангулирует данную систему уравнений
(define (triangulate M)
  '(1 2))
```

Преобразуйте пример выше в тест и дайте ответы на четыре вопроса проектирования, опираясь на наше подробное описание.

Не отвлекайтесь пока на реализацию шага завершения. ■

К сожалению, иногда решение упражнения 466 не дает желаемой треугольной системы. Например, рассмотрим следующее представление системы уравнений:

```
| (list (list 2 3 3 8)
      (list 2 3 -2 3)
      (list 4 -2 2 4))
```

Она имеет такое решение: $x = 1$, $y = 1$ и $z = 1$.

Первый шаг – вычесть первую строку из второй и дважды вычесть ее из последней. В результате получается такая матрица:

```
| (list (list 2 3 3 8)
      (list 0 -5 -5)
      (list -8 -4 -12))
```

На следующем шаге алгоритм триангуляции должен был бы сосредоточиться на оставшейся части матрицы (после исключения первого уравнения):

```
| (list (list 0 -5 -5)
      (list -8 -4 -12))
```

но первый коэффициент в этой матрицы равен 0. Поскольку делить на 0 нельзя, алгоритм завершается сообщением об ошибке в функции `subtract`.

Чтобы преодолеть эту проблему, нужно воспользоваться еще одним знанием из предметной области. Одно из правил математики гласит, что перестановка уравнений в системе не влияет на решение. Конечно, выполняя перестановку, мы должны найти уравнение, старший коэффициент которого не равен 0. Здесь мы можем просто поменять местами оставшиеся два уравнения:

```
| (list (list -8 -4 -12)
      (list 0 -5 -5))
```

Теперь можно продолжить и вычесть первое уравнение из последнего 0 раз. Итоговая треугольная матрица имеет вид:

```
| (list (list 2 3 3 8)
      (list -8 -4 -12)
      (list -5 -5))
```

Стоп! Покажите, что значения $x = 1$, $y = 1$ и $z = 1$ являются решением этой системы уравнений.

Упражнение 467. Измените функцию `triangulate` из упражнения 466 так, чтобы она сначала устанавливала первым уравнение со старшим коэффициентом, отличным от 0, и только потом выполняла вычитание первого уравнения из оставшихся.

Завершается ли этот алгоритм для всех возможных систем уравнений?

Подсказка. Следующее выражение переставляет элементы непустого списка `L`:

```
| (append (rest L) (list (first L)))
```

Объясните, как оно работает. ■

Некоторые системы уравнений SOE не имеют решения. Вот пример такой системы:

$$\begin{aligned} 2 \cdot x + 2 \cdot y + 2 \cdot z &= 6 \\ 2 \cdot x + 2 \cdot y + 4 \cdot z &= 8 \\ 2 \cdot x + 2 \cdot y + 1 \cdot z &= 2 \end{aligned}$$

Если попытаться триангулировать эту систему – вручную или с помощью решения упражнения 467, – вы получите промежуточную матрицу, в которой уравнения начинаются с нулевых коэффициентов:

$$\begin{aligned}0 \cdot x + 0 \cdot y + 2 \cdot z &= 6 \\0 \cdot x + 0 \cdot y - 1 \cdot z &= 0\end{aligned}$$

Упражнение 468. Измените `triangulate` из упражнения 467 так, чтобы она сообщала об ошибке, столкнувшись с системой уравнений, в которой все старшие коэффициенты равны 0. ■

После получения треугольной системы уравнений, такой как (*) в упражнении 463, уравнения можно решить по одному. В нашем конкретном примере последнее уравнение говорит, что z равно 2. Это значение можно подставить во второе уравнение:

$$3 \cdot y + 9 \cdot 2 = 21.$$

Отсюда легко вычислить значение y :

$$y = (21 - 9 \cdot 2)/3.$$

Теперь известные значения $z = 2$ и $y = 1$ можно подставить в первое уравнение:

$$2 \cdot x + 2 \cdot 1 + 3 \cdot 2 = 10.$$

И получить уравнение с одной переменной, которое решается так:

$$x = (10 - (2 \cdot 1 + 3 \cdot 2))/2.$$

В результате мы получаем значение x и решение всей системы уравнений.

Упражнение 469. Спроектируйте функцию `solve`. Она должна принимать треугольную систему уравнений SOE и возвращать решение.

Подсказка. Используйте структурно-рекурсивный подход к проектированию. Для начала спроектируйте функцию, решающую одно линейное уравнение от $n + 1$ переменных, учитывая решение для последних n переменных. Эта функция должна подставить значения из оставшейся части слева, вычесть результат из правой части и разделить разность на первый коэффициент. Поэкспериментируйте с этой подсказкой и примерами выше.

Усложненное задание. Используйте существующую абстракцию и лямбда-выражение для разработки `solve`. ■

Упражнение 470. Спроектируйте функцию `gauss`, которая комбинирует функцию `triangulate` из упражнения 468 и функцию `solve` из упражнения 469. ■

29. Алгоритмы с возвратами

Решение задач не всегда движется по прямой. Иногда мы можем применить какой-то метод и обнаружить, что застопорились из-за ошибки. Один из очевидных вариантов в таких случаях – вернуться в точку, где было принято судьбоносное решение, и попробовать повернуть в другую сторону. Некоторые алгоритмы именно так и работают. В этой главе мы рассмотрим два примера таких алгоритмов. В первом разделе мы исследуем алгоритм обхода графов, а во втором – рассмотрим расширенное упражнение, в котором возврат используется в контексте решения шахматной головоломки.

29.1. Обход графов

Графы широко распространены в нашем мире и в мире вычислений. Представьте себе группу людей, скажем студентов вашего университета. Запишите все имена и соедините линиями имена тех людей, которые знают друг друга. Вы только что создали свой первый неориентированный граф.

Теперь взгляните на рис. 27, где изображен небольшой ориентированный граф. Он состоит из семи узлов – кружков с буквами – и девяти ребер – стрелок. Граф может представлять уменьшенную версию сети электронной почты. Представьте себе компанию, в компьютерной сети которой постоянно пересыпаются электронные письма. Запишите электронные адреса всех сотрудников. Затем нарисуйте стрелки от каждого адреса ко всем тем адресам, куда с этого адреса посыпались электронные письма в течение недели. Примерно так можно создать ориентированный граф, изображенный на рис. 27, хотя у вас он может получиться намного более сложным и почти недоступным для понимания.

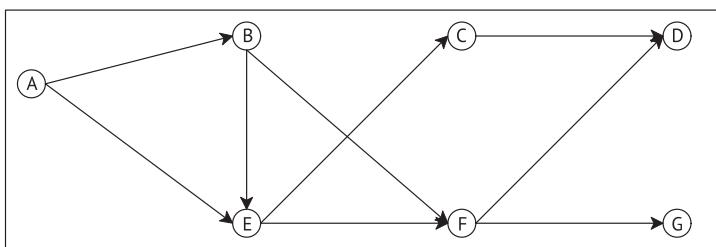


Рис. 27. Ориентированный граф

В общем случае граф состоит из набора узлов и ребер, соединяющих узлы. В *ориентированном графе* ребра представляют односторонние связи между узлами; в *неориентированном графе* ребра представляют двунаправленные связи. В этом контексте широко распространена следующая задача.

Социологи используют такие алгоритмы для определения иерархии управления в компании. Такие же графы используются или для прогнозирования возможных действий людей, даже не зная о содержании электронных писем.

Постановка задачи. Спроектируйте алгоритм передачи информации от одного человека к другому в ориентированном графе электронной почты крупной компании. Программа должна принимать ориентированный граф, представляющий имеющиеся пути передачи электронной почты и два адреса, и возвращать последовательность адресов электронной почты, через которые информация может быть передана от отправителя с первым адресом получателю со вторым адресом.

Ученые-математики называют желаемую последовательность *путем*.

Рисунок 27 помогает конкретизировать задачу. Например, программа должна найти путь от *C* к *D*. Этот конкретный путь состоит из исходного узла *C* и конечного узла *D*. Напротив, информацию из узла *E* в узел *D* можно передать двумя путями:

- отправить электронное письмо из *E* в *F*, а затем в *D*;
- отправить электронное письмо из *E* в *C*, а затем в *D*.

Иногда невозможно проложить путь между двумя узлами. Например, в графе на рис. 27 нельзя перейти из *C* в *G*, следуя по стрелкам.

Глядя на рис. 27, легко понять, как перейти от одного узла к другому, не задумываясь о том, как это делается. Итак, представьте на мгновение, что граф на рис. 27 представляет большой парк. Также представьте, что кто-то говорит, что вы находитесь в точке *E* и должны добраться до точки *G*. Вы ясно видите два пути, один из которых ведет в точку *C*, а другой – в точку *F*. Следуйте по первому пути и обязательно запомните, что из *E* также можно добраться до *F*. Теперь перед вами стоит новая задача, а именно как добраться из *C* в *G*. Самое интересное, что эта новая задача ничем не отличается от исходной; она точно так же предлагает найти путь из одного узла в другой. Более того, если вы сможете решить эту задачу, то вы сможете решить и задачу перехода из *E* в *G* – вам просто нужно будет добавить шаг из *E* в *C* к этому решению. Но перейти из *C* в *G* невозможно. К счастью, вы помните, что так же можно перейти из *E* в *F*, то есть можно вернуться в некоторую точку назад, где у вас есть выбор, и начать поиск оттуда.

Теперь выполним систематическое проектирование этого алгоритма. Следуя общему рецепту проектирования, начнем с анализа данных. Вот два компактных представления графа на рис. 27 в виде списков:

<pre>(define sample-graph '((A (B E)) (B (E F)) (C (D)) (D ())) (E (C F)) (F (D G)) (G ())))</pre>	<pre>(define sample-graph '((A B E) (B E F) (C D) (D)) (E C F) (F D G) (G)))</pre>
---	---

Оба содержат по одному списку на узел. Каждый из списков начинается с имени узла, за которым следуют его (непосредственные) *соседи*, то есть узлы, доступные по прямой стрелке. Они отличаются способом связывания (имен) узлов с их соседями: в представлении слева используется *list*, а справа – *cons*. Например, второй список представляет узел *B* с двумя исходящими ребрами к *E* и *F* на рис. 27. В представлении слева 'B – это первое имя в двухэлементном списке; в представлении справа – первое имя в трехэлементном списке.

Упражнение 471. Переведите одно из приведенных выше определений в надлежащую форму списка, используя *list* и соответствующие символы.

Представление данных для узлов выглядит просто:

```
| ; Node -- это символ.
```

Сформулируйте определение данных *Graph* для описания класса всех графов с любым количеством узлов и ребер.

Спроектируйте функцию *neighbors*. Она должна принимать *Node n* и *Graph g* и создавать список ближайших соседей *n* в *g*. ■

Используя определения *Node* и *Graph*, какими бы они ни были, и функцию *neighbors*, можно сформулировать сигнатуру и заявление о назначении для *find-path*, функции поиска пути в графе:

```
| ; Node Node Graph -> [List-of Node]
| ; ищет путь из origination в destination в графе G
(define (find-path origination destination G)
  '())
```

Этот заголовок оставляет открытой точную форму результата. Он подразумевает, что результатом является список узлов, но не говорит ничего о том, какие узлы он содержит.

Чтобы оценить эту двусмысленность и ее значение, изучим примеры, приведенные выше. В ISL+ их можно сформулировать так:

```
| (find-path 'C 'D sample-graph)
| (find-path 'E 'D sample-graph)
| (find-path 'C 'G sample-graph)
```

Первый вызов *find-path* должен вернуть единственный путь, второй должен выбирать один из двух возможных путей, а третий – сообщить об отсутствии пути из 'C в 'G в *sample-graph*. Итак, вот два возможных способа представить возвращаемое значение:

- результат функции включает все узлы, находящиеся в пути между исходным узлом *origination* и целевым узлом *destination*, включая и эти два. Для обозначения отсутствия пути между двумя узлами можно использовать пустой список;
- в качестве альтернативы, поскольку в вызове функции уже имеются два узла, в выходных данных могут упоминаться только

Легко вообразить другие способы, например опустить любой из двух заданных узлов.

«внутренние» узлы в пути. Тогда первый вызов может вернуть '(), потому что 'D является непосредственным соседом для 'C. Конечно, в этом случае '() больше не может служить признаком отсутствия пути.

Что касается проблемы отсутствия пути, для обозначения этого понятия следует выбрать отдельное значение. Поскольку #false отличается от других возможных результатов и не лишено смысла в данном контексте, мы выбираем его. В отношении возможного наличия нескольких путей мы пока отложим выбор и перечислим обе возможности в следующем разделе:

```
; Path -- это [List-of Node].
; интерпретация: список узлов непосредственных
; соседей на пути из первого узла Node
; в списке в последний.

; Node Node Graph -> [Maybe Path]
; ищет путь из origination в destination в G
; если путь не найден, то функция возвращает #false

(check-expect (find-path 'C 'D sample-graph)
              '(C D))
(check-member-of (find-path 'E 'D sample-graph)
                  '(E F D) '(E C D))
(check-expect (find-path 'C 'G sample-graph)
              #false)

(define (find-path origination destination G)
  #false)
```

Наш следующий шаг – исследовать четыре основные части функции: условие «тривиальной задачи», соответствующее решение, создание новой задачи и этап комбинирования. Обсуждение процесса поиска выше и анализ трех примеров предполагают следующие ответы:

- Если два заданных узла напрямую связаны стрелкой в данном графе, то путь состоит только из этих двух узлов. Но есть еще более простой случай – когда аргументы `origination` и `destination` представляют один и тот же узел.
- Во втором случае задача действительно оказывается тривиальной, и соответствующий ответ – (`list destination`).
- Если аргументы представляют разные узлы, алгоритм должен проверить всех непосредственных соседей узла `origination` и определить, существует ли путь из любого из них в `destination`. Иначе говоря, выбор одного из соседей порождает новый экземпляр задачи «найти путь».
- Наконец, когда алгоритм найдет путь из соседа узла `origination` в узел `destination`, мы сможем построить полный путь из первого во второй, просто добавив узел `origination` в список.

С точки зрения программирования третий пункт имеет особенно важное значение. Поскольку узел может иметь произвольное количество соседей, задача «проверить всех соседей» слишком сложна для одного примитива. Нужна вспомогательная функция, которая принимает список узлов и для каждого генерирует новую задачу поиска пути. Иначе говоря, функция представляет собой версию `find-path`, основанную на списках.

Назовем эту вспомогательную функцию `find-path/list` и сформулируем для нее запись в списке желаний:

```
; [List-of Node] Node Graph -> [Maybe Path]
; ищет путь из некоторого узла в lo-originations в
; узел destination; если путь не найден, то возвращает #false
(define (find-path/list lo-originations destination G)
  #false)
```

Для этой записи в списке желаний можно заполнить обобщенный макет генеративно-рекурсивной функции, чтобы получить первый вариант `find-path`:

```
(define (find-path origination destination G)
  (cond
    [(symbol=? origination destination)
     (list destination)]
    [else
     (... origination ...
          ...(find-path/list (neighbors origination G)
                            destination G) ...)]))
```

Он использует функции `neighbors` из упражнения 471 и `find-path/list` из списка желаний, а также ответы на четыре вопроса о генеративно-рекурсивных функциях.

Остальная часть процесса проектирования заключается в правильном комбинировании этих функций. Рассмотрим сигнатуру `find-path/list`. Так же как `find-path`, она возвращает `[Maybe Path]`. То есть если она находит путь от любого из соседей, она возвращает этот путь; в противном случае, если ни один из соседей не связан с `destination`, возвращает `#false`. Следовательно, ответ `find-path` зависит от типа результата, возвращаемого `find-path/list`, и код может различить два возможных ответа с помощью выражения `cond`:

```
(define (find-path origination destination G)
  (cond
    [(symbol=? origination destination)
     (list destination)]
    [else
     (local ((define next (neighbors origination G))
            (define candidate
              (find-path/list next destination G)))
       (cond
         [((boolean? candidate) ...)
          [(cons? candidate) ...]])))])
```

Эти два условия отражают два типа возможных ответов: логическое значение или список. В первом случае `find-path/list` не смогла найти путь от любого соседа в `destination`, а это означает, что и самой `find-path` не удастся найти такой путь. Во втором случае вспомогательная функция нашла путь и теперь `find-path` должна добавить `origination` в начало этого пути, потому что найденный путь `candidate` начинается с одного из соседей, а не с самого узла `origination`.

В листинге 111 показано полное определение `find-path`. В нем также имеется определение функции `find-path/list`, обрабатывающей свой первый аргумент посредством структурной рекурсии. Для каждого узла в списке `find-path/list` вызывает `find-path`, чтобы найти путь. Если `find-path` возвращает путь, то он становится результатом функции `find-path/list`. В противном случае `find-path/list` выполняет возврат на шаг назад.

ПРИМЕЧАНИЕ. В разделе 19.1 обсуждается возврат в структурном мире. Особенно хороший пример – функция, которая ищет голубоглазых предков в генеалогическом древе. Когда функция встречает узел, она сначала выполняет поиск в одной ветви семейного древа, скажем в отцовской, и если этот поиск возвращает `#false`, то производится поиск в другой ветви. Поскольку графы могут служить обобщенным представлением деревьев, сравнение этой функции с `find-path` является поучительным упражнением. **КОНЕЦ.**

Наконец, нужно проверить, сможет ли `find-path` дать верный ответ для всех возможных входных данных. Относительно легко проверить, что если передать граф, изображенный на рис. 27, и любые два узла в этом графе, то `find-path` всегда вернет какой-то ответ. Стоп! Решите следующее упражнение, прежде чем продолжить чтение.

Упражнение 472. Протестируйте `find-path`. Используйте эту функцию, чтобы найти путь из 'A' в 'G' в `sample-graph`. Какой из возможных путей будет найден? Почему?

Спроектируйте функцию `test-on-all-nodes`, которая принимает граф `g` и определяет наличие пути между любыми двумя узлами. ■

Однако для других графов `find-path` может не завершаться, встретив определенную пару узлов. Рассмотрим граф на рис. 28.

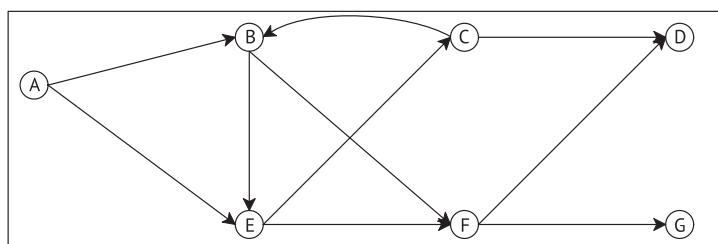


Рис. 28. Ориентированный граф с циклом

Листинг 111. Поиск пути в графе

```

; Node Node Graph -> [Maybe Path]
; ищет путь из origination в destination в G
; если путь не найден, функция возвращает #false
(define (find-path origination destination G)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define next (neighbors origination G))
                  (define candidate
                    (find-path/list next destination G)))
               (cond
                 [(boolean? candidate) #false]
                 [else (cons origination candidate)])))))

; [List-of Node] Node Graph -> [Maybe Path]
; ищет путь из некоторого узла в lo-0s в узел D
; если путь не найден, функция возвращает #false
(define (find-path/list lo-0s D G)
  (cond
    [(empty? lo-0s) #false]
    [else (local ((define candidate
                   (find-path (first lo-0s) D G)))
                (cond
                  [(boolean? candidate)
                   (find-path/list (rest lo-0s) D G)]
                  [else candidate]))]))

```

Стоп! Определите `cyclic-graph` для представления графа на этом рисунке.

По сравнению с графом на рис. 27, этот новый граф содержит только одно дополнительное ребро: из *C* в *B*. Однако это, казалось бы, небольшое добавление заставляет поиск вернуться в тот же самый узел. В частности, можно перейти из *B* в *E*, затем в *C* и вернуться обратно в *B*. Действительно, если в функцию `find-path` передать '*B*', '*D*' и этот граф, то она не сможет завершиться, как подтверждают пошаговые вычисления:

```

(find-path 'B 'D cyclic-graph)
== .. (find-path 'B 'D cyclic-graph) ..
== .. (find-path/list (list 'E 'F) 'D cyclic-graph) ..
== .. (find-path 'E 'D cyclic-graph) ..
== .. (find-path/list (list 'C 'F) 'D cyclic-graph) ..
== .. (find-path 'C 'D cyclic-graph) ..
== .. (find-path/list (list 'B 'D) 'D cyclic-graph) ..
== .. (find-path 'B 'D cyclic-graph) ..

```

Вычисления в пошаговом режиме показывают, что после семи применений `find-path` и `find-path/list` программа на ISL+ должна будет вычислить то же самое выражение, с которого она начинала. Поскольку для одних и тех же входных данных любые функции вычисляют тот же самый результат, функция `find-path` не сможет завершиться.

Таким образом, **аргумент завершения** формулируется следующим образом: если некоторый заданный граф не содержит циклов,

Вы знаете только одно исключение из этого правила: random.

то `find-path` генерирует некоторый результат для любых входных данных. В конце концов, каждый путь может содержать только конечное число узлов, и число путей тоже конечно. Соответственно, функция либо проверит все решения, начиная с некоторого заданного узла, либо найдет путь от начального узла до конечного. Однако если граф содержит цикл, то есть путь от некоторого узла, который ведет обратно к этому узлу, то `find-path` может не дать результата для некоторых входных данных.

В следующей части представлен метод проектирования программ, который решает такие проблемы. В частности, в нем показана версия `find-path`, способная обрабатывать циклы в графе.

Упражнение 473. Протестируйте `find-path` с парой узлов 'B, 'C и графиком на рис. 28. Также попробуйте применить `test-on-all-nodes` из упражнения 472 к этому графу. ■

Упражнение 474. Повторно спроектируйте `find-path` как единую функцию. ■

Упражнение 475. Повторно спроектируйте `find-path/list` так, чтобы она использовала существующую абстракцию списка из листингов 56 и 57 вместо явной структурной рекурсии. **Подсказка.** Прочтайте документацию с описанием `ogtar` в языке Racket. Чем она отличается от функции `ogtar` в ISL+? Можно ли использовать первую из них в данном случае? ■

ЗАМЕЧАНИЕ ОБ АБСТРАКЦИИ ДАННЫХ. Возможно, вы заметили, что функция `find-path` не зависит от особенностей определения `Graph`. Пока гарантируется правильная работа функции `neighbors` для `Graph`, функция `find-path` будет прекрасно справляться со своими обязанностями. Проще говоря, функция `find-path` использует **абстракцию данных**.

Как отмечалось в части III, абстракция данных подобна абстракции функций. Мы могли бы создать функцию `abstract-find-path`, принимающую на один параметр больше, чем `find-path: neighbors`, и такая функция всегда обрабатывала бы график должным образом, если бы всегда получала график `G`, принадлежащий классу `Graph`, и соответствующую функцию `neighbors`. Дополнительный параметр предполагает абстракцию в общепринятом смысле, однако требуемая связь между двумя параметрами – `G` и `neighbors` – на самом деле означает, что `abstract-find-path` также абстрагирована по определению `Graph`, а поскольку `Graph` – это определение данных, то идея получила название абстракции данных.

Когда программы вырастают до определенных размеров, абстракция данных становится критически важным инструментом конструирования компонентов программ. В следующем томе серии «Как проектировать» эта идея рассматривается более подробно, а в следующем разделе эта идея иллюстрируется на примере. **КОНЕЦ.**

Упражнение 476. В разделе 12.8 была поставлена задача, касающаяся конечных автоматов и строк, но сразу же было сказано, что она

будет решена в этой главе, потому что решение требует генеративной рекурсии. Теперь вы обладаете достаточным объемом знаний, чтобы решить эту задачу.

Спроектируйте функцию `fsm-match`. Она должна принимать представление конечного автомата и строки и возвращать `#true`, если последовательность символов в строке вызывает переход конечного автомата из начального состояния в конечное.

Поскольку эта задача связана с проектированием генеративно-рекурсивных функций, мы предоставим основные определения данных и пример:

```
(define-struct transition [current key next])
(define-struct fsm [initial transitions final])

; FSM - это структура:
; (make-fsm FSM-State [List-of 1Transition] FSM-State)
; 1Transition - это структура:
; (make-transition FSM-State 1String FSM-State)
; FSM-State - это строка.

; пример данных: см. упражнение 109

(define fsm-a-bc*-d
  (make-fsm
    "AA"
    (list (make-transition "AA" "a" "BC")
          (make-transition "BC" "b" "BC")
          (make-transition "BC" "c" "BC")
          (make-transition "BC" "d" "DD"))
    "DD"))
```

Пример данных соответствует регулярному выражению $a(b|c)^*d$. Как упоминалось в упражнении 109, этому выражению соответствуют такие строки, как "acbd", "ad" и "abcd"; а строки "da", "aa" и "d" – не соответствуют.

Учитывая это, вы должны спроектировать следующую функцию:

```
; FSM String -> Boolean
; определяет, распознает ли an-fsm данную строку
(define (fsm-match? an-fsm a-string)
  #false)
```

Подсказка. Спроектируйте локальную вспомогательную функцию для `fsm-match?`. В этом контексте представьте задачу как пару параметров: текущее состояние конечного автомата и оставшийся список букв `1String`. ■

Упражнение 477. Изучите определение функции `arrangements` в листинге 112. В этом листинге представлено генеративно-рекурсивное решение расширенной задачи из раздела 12.4, а именно:

для заданного слова создайте все возможные перестановки букв.

Мы благодарим Марка Энгельберга (Mark Engelberg) за это предложенное упражнение.

Структурно-рекурсивный подход к решению этого упражнения требует спроектировать основную и две вспомогательные функции. Причем проектирование последних требует спроектировать еще две вспомогательные функции. Решение в листинге 112, на-против, использует возможности генеративной рекурсии – плюс `foldr` и `map` – для определения той же самой программы в виде единой функции.

Объясните устройство генеративно-рекурсивной версии `arrangements`. Ответьте на все вопросы, которые задает рецепт проектирования на основе генеративной рекурсии, включая вопрос о завершении.

Возвращают ли `arrangements` в листинге 112 те же списки, что и решение в разделе 12.4? ■

Листинг 112. Определение функции arrangements с использованием генеративной рекурсии

```
; [List-of X] -> [List-of [List-of X]]
; создает список всех перестановок элементов в w
(define (arrangements w)
  (cond
    [(empty? w) '(())]
    [else
      (foldr (lambda (item others)
        (local ((define without-item
          (arrangements (remove item w)))
          (define add-item-to-front
            (map (lambda (a) (cons item a))
              without-item)))
        (append add-item-to-front others)))
        '()
        w)))))

(define (all-words-from-rat? w)
  (and (member (explode "rat") w)
    (member (explode "art") w)
    (member (explode "tar") w)))

(check-satisfied (arrangements '("г" "а" "т"))
  all-words-from-rat?)
```

29.2. Проект: возврат

Мы благодарим Марка Энгельберга (*Mark Engelberg*) за правку этого раздела.

Головоломка с *n* ферзями – известная задача в мире шахмат, которая естественным образом иллюстрирует применимость приема возвратов. Для наших целей будем представлять шахматную доску как сетку с размерами $n \times n$ клеток. Ферзь – это фигура, которая может перемещаться на любое количество клеток по горизонтали, вертикали или диагонали, но не «перепрыгивая» через другие фигуры. Мы говорим, что ферзь угрожает клетке (полю), если находится на ней или может перейти на нее. Рисунок 29 иллюстрирует это графически. Ферзь находится на второй вертикали и в шестой

горизонтали. Сплошные линии, исходящие от ферзя, проходят через все поля, которым угрожает ферзь.

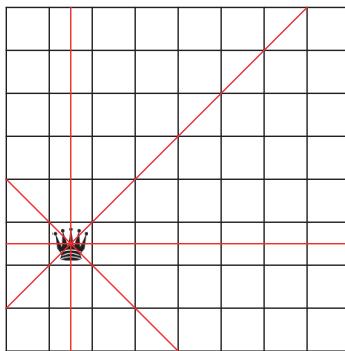


Рис. 29. Шахматная доска с единственным ферзем и полями, которым он угрожает

Классическая задача о ферзях состоит в том, чтобы разместить 8 ферзей на шахматной доске 8×8 так, чтобы они не угрожали друг другу. Специалисты по информатике обобщили эту задачу и спрашивают, можно ли поставить n ферзей на шахматную доску $n \times n$ так, чтобы ферзи не угрожали друг другу.

Очевидно, что для $n = 2$ головоломка не имеет решения. Ферзь, размещенный на любом из четырех полей, угрожает всем остальным полям.

Для $n = 3$ задача тоже не имеет решения. На рис. 30 показаны все различные расстановки двух ферзей, то есть решения для $k = 3$ и $n = 2$. В любом случае левый ферзь занимает поле в левой вертикали, а второй размещается в одном из двух полей, которым не угрожает первый. Размещение второго ферзя угрожает всем оставшимся незанятым полям, а это означает, что невозможно разместить третьего ферзя, не нарушив условий задачи.

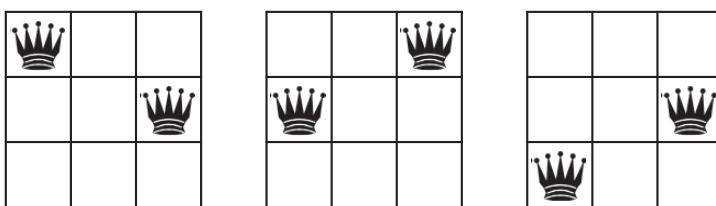


Рис. 30. Три ферзя на шахматной доске 3×3

Упражнение 478. Точно так же можно разместить первого ферзя во всех полях в верхней горизонтали, в правой вертикали и в нижней горизонтали. Объясните, почему все ситуации аналогичны трем сценариям, изображенным на рис. 30.

Остается центральное поле. Можно ли разместить хотя бы второго ферзя, если первого поставить в центральное поле на доске 3×3 ? ■

На рис. 31 показаны два решения головоломки для $n = 4$ (слева) и для $n = 5$ (справа). На рисунке видно, что в каждом решении в каждой горизонтали и в каждой вертикали находится по одному ферзю, что вполне объяснимо, потому что ферзь угрожает всей горизонтали и вертикали, в которых находится занятое им поле.

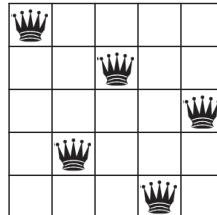
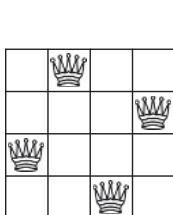


Рис. 31. Решение головоломки для случаев 4×4 и 5×5

Теперь, проведя достаточно подробный анализ, можно переходить к этапу решения. В ходе анализа было определено несколько идей:

1. Задача решается размещением ферзей по очереди. Помещая ферзя на доску, мы можем отметить горизонтали, вертикали и диагонали, непригодные для размещения других ферзей.
2. При выборе места для каждого следующего ферзя должны рассматриваться только безопасные поля.
3. На тот случай, если выбор поля приведет к проблемам позже, необходимо запомнить, какие другие поля подходят для размещения этого ферзя.
4. Если на доске не осталось безопасных полей, то мы должны вернуться к предыдущему шагу, когда выбирали поле, и попробовать одно из оставшихся полей.

Проще говоря, этот процесс решения похож на алгоритм поиска пути.

Переход от описания процесса к проектированию алгоритма явно требует двух представлений данных: для шахматной доски и для полей на доске. Начнем с последнего:

```
(define QUEENS 8)
; QP - это структура:
;   (make-posn CI CI)
; CI - это N в [0,QUEENS].
; интерпретация: (make-posn г с) обозначает поле
; в г-й горизонтали и с-й вертикали
```

В конце концов, выбор, по сути, продиктован шахматной доской.

В определении CI можно было бы использовать [1, QUEENS] вместо [0, QUEENS], но эти два определения практически эквивалентны,

к тому же программисты привыкли начинать счет с 0. Аналогично так называемая алгебраическая нотация обозначения шахматных полей требует использования букв от 'a до 'h для представления одной из размерностей доски, то есть в QP можно было бы использовать СИ и такие буквы. Однако эти обозначения практически эквивалентны, а с использованием натуральных чисел легче создать большое число позиций, чем с буквами.

Упражнение 479. Спроектируйте функцию `threatening?`. Она должна принимать два экземпляра QP и определять, будут ли ферзи, размещенные в двух соответствующих полях, угрожать друг другу.

Знание предметной области. (1) Изучите рис. 29. Ферзь на этом рисунке угрожает всем полям на горизонтальной, вертикальной и диагональных линиях. И наоборот, другой ферзь, размещенный на любом поле в этих линиях, угрожает первому ферзю.

(2) Преобразуйте свои мысли в математические уравнения, связывающие координаты полей друг с другом. Например, все поля на одной горизонтали имеют одинаковую координату Y. Точно так же все поля на одной диагонали имеют координаты, суммы которых равны. Какая это диагональ? Для другой диагонали разница между двумя координатами остается неизменной. Какую диагональ описывает эта идея?

Подсказка. Познакомившись с предметными знаниями, сформулируйте набор тестов, охватывающий горизонтали, вертикали и диагонали. Не забудьте указать аргументы, для которых `threatening?` должна вернуть `#false`. ■

Упражнение 480. Спроектируйте функцию `render-queens`. Она должна принимать натуральное число `n`, список QP и изображение и создавать изображение шахматной доски $n \times n$ с переданным изображением, размещенным в соответствии с координатами в QP. ■

Вы можете попробовать найти изображение ферзя в интернете или создать свое изображение, используя доступные функции рисования.

Что касается представления *Board* для шахматной доски, то мы отложим этот шаг, пока не узнаем, как алгоритм реализует процесс поиска. Это еще одно упражнение в абстрагировании данных. В действительности определение данных *Board* не требуется для формулировки сигнатуры алгоритма:

```
; N -> [Maybe [List-of QP]]
; ищет решение задачи с n ферзями

; пример данных: [List-of QP]
(define 4QUEEN-SOLUTION-2
  (list (make-posn 0 2) (make-posn 1 0)
        (make-posn 2 3) (make-posn 3 1)))

(define (n-queens n)
  #false)
```

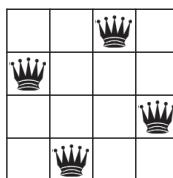
Задача состоит в том, чтобы найти место для n ферзей на шахматной доске $n \times n$. Очевидно, что алгоритм не принимает ничего, кро-

ме натурального числа, и возвращает представление, описывающее размещение n ферзей, если решение имеется. Решение может быть представлено списком QP, поэтому мы выбираем для представления результата

```
| ; [List-of QP] или #false
```

Естественно, #false означает отсутствие решения.

Следующий шаг – разработка примеров и их формулирование в виде тестов. Мы знаем, что задача с 2 или 3 ферзями не имеет решения. Для 4 ферзей есть два решения. Одно из них показано на рис. 31 слева, а вот второе решение:



Однако с точки зрения представления данных существует много разных способов описания этих двух изображений. Некоторые варианты показаны в листинге 113. Добавьте остальные.

Упражнение 481. Тесты в листинге 113 ужасны. Ни один настоящий программист никогда не перечисляет все возможные результаты.

Одно из решений – снова использовать тестирование свойств. Спроектируйте функцию $n\text{-queens-solution?}$, которая принимает натуральное число n и возвращает предикат для оценки размещения ферзей, который определяет, является ли данное размещение решением головоломки:

- решение головоломки с n ферзями должно иметь длину n ;
- QP в таком списке не может угрожать другим QP.

Листинг 113. Решения для головоломки с 4 ферзями

```
; N -> [Maybe [List-of QP]]
; ищет решение задачи с n ферзями

(define 0-1 (make-posn 0 1))
(define 1-3 (make-posn 1 3))
(define 2-0 (make-posn 2 0))
(define 3-2 (make-posn 3 2))

(check-member-of
  (n-queens 4)
  (list 0-1 1-3 2-0 3-2)
  (list 0-1 1-3 3-2 2-0)
  (list 0-1 2-0 1-3 3-2)
  (list 0-1 2-0 3-2 1-3)
  (list 0-1 3-2 1-3 2-0)
  (list 0-1 3-2 2-0 1-3)
  ...
  ...)
```

```
(list 3-2 2-0 1-3 0-1))

(define (n-queens n)
  (place-queens (board0 n) n))
```

Протестирував цей предикат, використуйте його та `check-satisfied`, щоб сформулювати тесты для `n-queens`.

Альтернативним рішенням є представлення списків `QP` в виде множеств. Якщо два списки містять однакові `QP`, але в іншому порядку, то вони еквівалентні, як показано на малюнку. Соответсно, тест для `n-queens` можна сформулювати так:

```
; [List-of QP] -> Boolean
; визначає рівність результату [як множества] одному з двох списків
(define (is-queens-result? x)
  (or (set=? 4QUEEN-SOLUTION-1 x)
      (set=? 4QUEEN-SOLUTION-2 x)))
```

Спроектуйте функцію `set=?`. Вона повинна приймати два списки та визначати, містять вони одні та ж елементи, незалежно від порядку. ■

Упражнення 482. Ключова ідея полягає в тому, щоб спроектувати функцію, розміщаючу `n` ферзів на шахматній досці, на якій вже може стояти декілька ферзів:

```
; Board N -> [Maybe [List-of QP]]
; поміщує n ферзів на доску; інакше повертає #false
(define (place-queens a-board n)
  #false)
```

Код в листингу 113 вже звертається до цієї функції в описі `n-queens`.

Спроектуйте алгоритм `place-queens`, предположивши, що у вас є доступ до наступних функцій для роботи з екземплярами `Board`:

```
; N -> Board
; створює пусту шахматну доску n на n
(define (board0 n) ...)

; Board QP -> Board
; поміщує ферзя в позицію qp на досці a-board
(define (add-queen a-board qp)
  a-board)

; Board -> [List-of QP]
; шукає місце, куди можна поставити ферзя
(define (find-open-spots a-board)
  '())
```

Перша з цих функцій використовується в листингу 113 для створення початкового представлення доски для розміщення ферзів. Две інші вам будуть потрібні, щоб описати генеративний етап алгоритму. ■

Ви поки не зможете підтвердити правильну роботу вашого рішення предыдущого упражнення, тому що воно базується на шир-

ном списке желаний. Для этого нужно представление данных Board, поддерживающее три функции из списка желаний. Это ваша оставшаяся задача.

Упражнение 483. Спроектируйте определение данных Board для представления шахматной доски и три функции, перечисленные в упражнении 482. Вот несколько идей, которые можно было бы реализовать:

- экземпляр Board включает позиции, где еще можно поставить ферзя;
- экземпляр Board содержит список позиций, на которых уже стоят ферзи;
- экземпляр Board представляет сетку из n на n ячеек, каждая из которых может быть занята ферзем. Используйте структуру с тремя полями для представления ячейки: одно для координаты X, одно для координаты Y и одно для хранения признака – находится ли ячейка под угрозой.

Используйте одну из перечисленных идей для решения этого упражнения. ■

Усложненное задание. Используйте все три идеи и придумайте три разных представления Board. Абстрагируйте свое решение в упражнении 482 и убедитесь, что place-queens работает с любым из ваших представлений данных Board. ■

30. Итоги

В этой пятой части книги мы познакомились с идеей *озарения* в проектировании программ. В отличие от структурного подхода к проектированию, описывавшегося в первых четырех частях, проектирование на основе *озарения* начинается с выбора идеи о том, как программа должна решать задачу или обрабатывать данные, представляющие эту задачу. Под проектированием здесь понимается придумывание способа вызова рекурсивной функции для решения новой задачи, похожей на данную, но более простой.

Имейте в виду, что хотя мы назвали этот прием **генеративной рекурсией**, большинство специалистов по информатике называют эти функции **алгоритмами**.

Завершив данную часть книги, вы должны понимать следующее:

1. Стандартная схема рецепта проектирования остается в силе.
2. Основное изменение касается этапа реализации. Он вводит четыре новых вопроса, ответы на которые помогают перейти от обобщенного шаблона генеративной рекурсии к законченной функции. В первых двух вопросах вы прорабатываете «тривиальные» части решения, а в двух других определяете реализацию этапа генеративного решения.
3. Незначительное изменение касается завершности генеративно-рекурсивных функций. В отличие от структурно-рекурсивных функций, алгоритмы могут не завершаться для некоторых входных данных. Иногда эта проблема связана с внутренними ограничениями в самой идее, иногда – с ее воплощением в коде. Как бы то ни было, будущий читатель вашей программы заслуживает предупреждения о потенциально «плохих» входных данных.

В своей практике программирования вы часто будете сталкиваться с простыми или хорошо известными алгоритмами, и мы верим, что вы справитесь с ними. Для создания действительно умных алгоритмов компаний, занимающиеся разработкой программного обеспечения, нанимают высокооплачиваемых специалистов, экспертов в предметной области и математиков, помогающих проработать концептуальные детали, прежде чем программистам будет предложено воплотить эти концепции в программы. Вы тоже должны быть готовы к такого рода задачам, и лучшая подготовка – это практика.

Интермеццо 5. Стоимость вычислений

Что можно сказать о программе f после успешного выполнения тестов:

```
| (check-expect (f 0) 0)
| (check-expect (f 1) 1)
| (check-expect (f 2) 8)
```

Первый ответ, что приходит на ум:

```
| (define (f x) (expt x 3))
```

Но едва ли кто-то догадается, что программа может выглядеть так:

```
| (define (f x) (if (= x 2) 8 (* x x)))
```

Тесты говорят лишь, что программа должным образом обрабатывает некоторые входные данные.

Также можно повторно прочитать раздел 16.2 и обсуждение проверок целостности в разделе 23.7.

Точно так же оценка времени выполнения программы с конкретными входными данными говорит лишь, сколько времени потребуется для вычисления ответа для этих входных данных – и ничего больше. У вас может быть две программы – `rgog-linear` и `rgog-square`, – которые вычисляют один и тот же результат для одинаковых входных данных, и вы можете обнаружить, что во всех созданных вами тестах `rgog-linear` вычисляется быстрее, чем `rgog-square`. В разделе 26.4 представлена именно такая пара программ: структурно-рекурсивная программа `gcd` и эквивалентная ей генеративно-рекурсивная программа `gcd-generative`. Сравнение времени их выполнения показывает, что последняя действует намного быстрее первой.

Но действительно ли `rgog-linear` предпочтительнее во всех случаях? Рассмотрим график на рис. 32. На этом графике по оси X откладывается размер входных данных, например длина списка, а по оси Y – время, необходимое для обработки входных данных определенного размера. Предположим, что прямая линия представляет время выполнения функции `rgog-linear`, а кривая – функции `rgog-square`. В заштрихованной области `rgog-linear` выполняется дольше, чем `rgog-square`, но на краю этой области два графика пересекаются, и справа от нее уже `rgog-square` выполняется дольше, чем `rgog-linear`. Если по каким-либо причинам вы оценили производительность `rgog-linear` и `rgog-square` только для входных данных, соответствующих заштрихованной области, а ваши клиенты применяют программу в основном к данным, попадающим в незаштрихованную область, то это означает, что вы передали им неправильную программу.

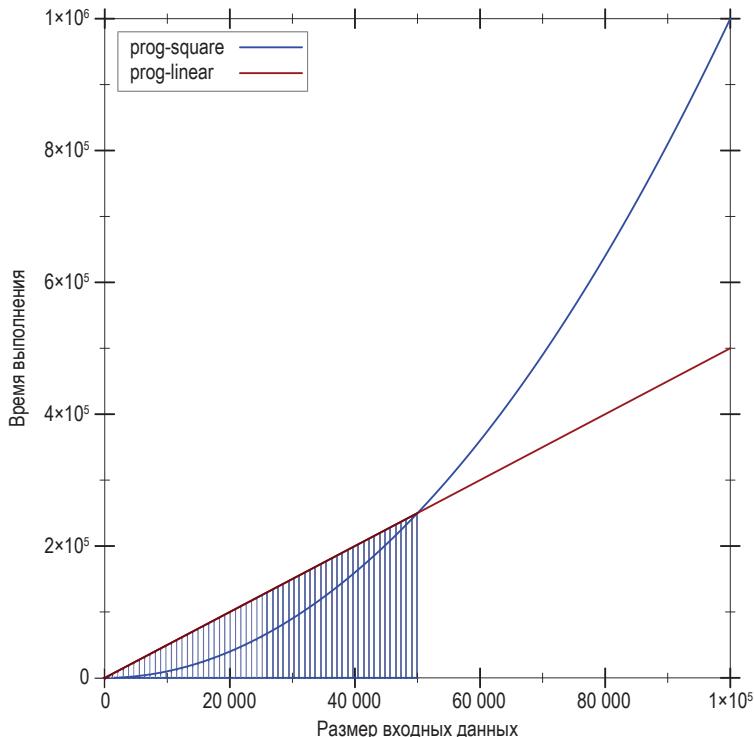


Рис. 32. Сравнение времени выполнения двух функций

Это интермеццо знакомит с понятием *алгоритмического анализа*, который позволяет программистам делать обобщенные утверждения о производительности программ, а всем остальным – о поведении функции. Любой серьезный программист и ученый должен познакомиться с этим понятием. Это основа для анализа характеристик производительности программ. Чтобы правильно понять эту идею, вам нужно засесть за учебники.

Мы благодарим
Прабхакара Рагде
(Prabhakar Ragde) за то,
что поделился своими
замечаниями, касающимися
алгоритмического анализа, в первом
издании этой книги.

Конкретное время, абстрактное время

В разделе 26.4 сравнивается время выполнения gcd и gcd-generative. Там же утверждается, что gcd-generative работает быстрее, потому что всегда выполняет меньше рекурсивных шагов, чем gcd. Давайте используем эту идею в качестве отправной точки для анализа производительности how-many, простой программы из главы 9:

```
(define (how-many a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (how-many (rest a-list)) 1)]))
```

Предположим, что мы решили определить, сколько времени потребуется для вычисления длины некоторого неизвестного непустого списка. Используя правила из интермецо 1, мы можем рассматривать этот процесс как серию алгебраических манипуляций:

```
(how-many some-non-empty-list)
==
(cond
  [(empty? some-non-empty-list) 0]
  [else (+ (how-many (rest some-non-empty-list)) 1)])
 ==
(cond
  [#false 0]
  [else (+ (how-many (rest some-non-empty-list)) 1)])
 ==
(cond
  [else (+ (how-many (rest some-non-empty-list)) 1)])
 ==
(+ (how-many (rest some-non-empty-list)) 1)
```

Первый шаг – заменить *a-list* в определении *how-many* фактическим аргументом – некоторым непустым списком *some-non-empty-list*, который передается для проверки первому выражению в *cond*. Теперь мы должны оценить

```
| (empty? some-non-empty-list)
```

Согласно условию, результат этого выражения – *#false*. Возникает вопрос: сколько времени понадобится выражению проверки условия, чтобы вычислить этот результат? Мы не знаем точного количества времени, но можем с уверенностью сказать, что проверка конструктора списка занимает небольшое и фиксированное время. Это же предположение справедливо и для следующего шага, когда *cond* проверяет значение первого условного выражения. Поскольку это *#false*, первая строка в *cond* отбрасывается. Проверка условия *else* тоже выполняется очень быстро, и теперь мы остаемся с выражением:

```
| (+ (how-many (rest some-non-empty-list)) 1)
```

Наконец, мы можем с уверенностью предположить, что *rest* извлекает оставшуюся часть списка за фиксированное время, но дальше мы, похоже, застряли. Чтобы вычислить, сколько времени требуется для определения длины некоторого списка, нужно знать, сколько времени необходимо для подсчета элементов в оставшейся части этого списка.

Как вариант можно предположить, что предикаты и селекторы выполняются в течение фиксированного времени, поэтому время, необходимое для определения длины списка, зависит от количества рекурсивных шагов. Точнее, для вычисления (*how-many some-list*) потребуется примерно *n* некоторых фиксированных интервалов времени, где *n* – это длина списка, или, что то же самое, количество повторений программы.

Обобщая этот пример, можно сказать, что время выполнения зависит от размера входных данных и количество рекурсивных шагов является хорошей оценкой продолжительности вычислений. По этой причине специалисты по информатике говорят об *абстрактном времени выполнения* программы, когда количество рекурсивных шагов зависит от размера входных данных. В нашем первом примере размер входных данных – это количество элементов в списке. То есть для обработки списка с одним элементом требуется один рекурсивный шаг, для списка с двумя элементами – два шага, а для списка с n элементами – n шагов.

«Абстрактное», потому что эта мера игнорирует такие тонкости, как время, необходимое на выполнение элементарных шагов.

В информатике принято говорить, что программа f выполняет «поп-рядка n шагов», чтобы сформулировать утверждение об абстрактном времени выполнения f . Чтобы использовать эту фразу правильно, ее надо сопроводить объяснением, что такое n , например «количество элементов в заданном списке» или «количество цифр в заданном числе». Без такого объяснения исходная фраза фактически не имеет смысла.

Не для всех программ можно сформулировать такое простое утверждение об абстрактном времени выполнения. Давайте рассмотрим первую рекурсивную программу в этой книге:

```
(define (contains-flatt? lo-names)
  (cond
    [(empty? lo-names) #false]
    [(cons? lo-names)
      (or (string=? (first lo-names) "flatt")
          (contains-flatt? (rest lo-names)))])))
```

Для списка, начинающегося с «flatt», например

```
(contains-flatt?
  (list "flatt" "robot" "ball" "game-boy" "pokemon"))
```

программе не потребуется выполнять рекурсивные вызовы. Напротив, если «flatt» окажется в конце списка, как, например, в

```
(contains-flatt?
  (list "robot" "ball" "game-boy" "pokemon" "flatt"))
```

то программе потребуется выполнить столько рекурсивных шагов, сколько элементов в списке.

Этот второй пример подводит нас ко второй важной идеи анализа программ, а именно к типу выполняемого анализа:

- *анализ наилучшего случая* фокусируется на классе входных данных, для которых программа легко находит ответ. В текущем примере лучшим экземпляром входных данных является список, начинающийся с 'flatt';
- *анализ наихудшего случая*, с другой стороны, определяет, насколько плохо программа справляется с входными данными

ми, которые вызывают наибольшую нагрузку. Функция `contains-flatt?` показывает худшую производительность, когда `'flatt` находится в конце входного списка;

- наконец, *анализ среднего случая* основывается на идее, что программисты не могут исходить из предположения о том, что входные данные всегда имеют наилучшую возможную форму, но могут надеяться, что входные данные также не будут иметь худшую из возможных форм. Во многих случаях желательно оценить среднее время, затрачиваемое программой на выполнение. Например, в среднем `contains-flatt?` находит `'flatt` где-нибудь в середине входного списка. То есть если список содержит n элементов, то среднее время работы `contains-flatt?` будет равно $n/2$, то есть количество рекурсивных вызовов будет вдвое меньше, чем элементов в списке.

Поэтому специалисты по информатике обычно используют слово «порядка» в сочетании со словами «в среднем» или «в худшем случае».

Идея о том, что `contains-flatt?` выполняет в среднем «порядка $n/2$ шагов», приводит нас к еще одной характеристике абстрактного времени выполнения. Поскольку мы игнорируем точное время, необходимое для выполнения элементарных шагов, таких как проверка предикатов, извлечение значений, проверка условий в `cond`, то точно так же мы можем отбросить деление на 2. И вот почему. Предполагается, что каждый элементарный шаг занимает k единиц времени, то есть время, затраченное на выполнение `contains-flatt?`, можно вычислить так:

$$k \cdot \frac{1}{2} \cdot n.$$

Если бы у вас был более мощный компьютер, то эти базовые вычисления могли бы выполняться вдвое быстрее, и в этом случае мы использовали бы $k/2$ в качестве константы, обозначающей выполнение элементарных операций. Назовем эту константу c и вычислим:

$$k \cdot \frac{1}{2} \cdot n = \frac{1}{2} \cdot k \cdot n = c \cdot n,$$

то есть абстрактное время выполнения всегда умножается на константу n , и это все, что имеет значение, чтобы сказать «порядка n шагов».

Теперь рассмотрим нашу программу сортировки из листинга 37. Вот сеанс пошаговой обработки небольшого списка со всеми рекурсивными шагами:

```
(sort (list 3 1 2))
== (insert 3 (sort (list 1 2)))
== (insert 3 (insert 1 (sort (list 2))))
== (insert 3 (insert 1 (insert 2 (sort '()))))
```

```

== (insert 3 (insert 1 (insert 2 '())))
== (insert 3 (insert 1 (list 2)))
== (insert 3 (cons 2 (insert 1 '())))
== (insert 3 (list 2 1))
== (insert 3 (list 2 1))
== (list 3 2 1)

```

Этот сеанс наглядно показывает, как `sort` перемещается по заданному списку и подготавливает применение `insert` для каждого числа в списке. Иначе говоря, `sort` – это двухэтапная программа. На первом этапе рекурсивные шаги в `sort` подготавливают столько применений `insert`, сколько есть элементов в списке. На втором этапе каждый вызов `insert` просматривает отсортированный список.

Вставка элемента похожа на его поиск, поэтому неудивительно, что `insert` и `contains-flatt?` имеют похожую производительность. Применение `insert` к списку с l элементами происходит между каждым рекурсивным шагом от 0 до l . В среднем мы предполагаем, что для этого требуется $l/2$ шагов, то есть `insert` производит «порядка l шагов», где l – длина данного списка.

Вопрос в том, какова длина этих списков, в которые `insert` добавляет числа. Обобщая приведенный выше расчет, можно заметить, что длина первого списка равна $n - 1$, второго $n - 2$ и т. д., вплоть до пустого списка. Следовательно, `insert` выполняет

$$\sum_{l=0}^{l=n-1} \frac{1}{2} \cdot l = \frac{1}{2} \cdot \sum_{l=0}^{n-1} l = \frac{1}{2} \cdot \frac{(n-1) \cdot n}{2} = \frac{1}{4} \cdot (n-1) \cdot n = \frac{1}{4} \cdot (n^2 - n),$$

то есть

$$\frac{1}{4} \cdot n^2 - \frac{1}{4} \cdot n$$

представляет лучшую «догадку» при среднем количестве шагов вставки. В этом последнем члене n^2 является доминирующим фактором, поэтому мы говорим, что процесс сортировки занимает «порядка n^2 шагов». В упражнении 486 вам будет предложено аргументировать, почему такое упрощение утверждения считается правильным.

См. упражнение 486, чтобы узнать, почему это так.

Мы также можем отбросить лишний формализм и строгость. Поскольку `sort` использует `insert` один раз для каждого элемента в списке, мы получаем «порядка n » шагов вставки, где n – размер списка. Так как функции `insert` требуется $n/2$ шагов, то теперь понятно, почему для всего процесса сортировки требуется $n \cdot n/2$ или «порядка n^2 » шагов.

Суммируя все сказанное, можно заключить, что `sort` выполняет «порядка n шагов» плюс n^2 рекурсивных шагов в `insert` для списка с n элементами. В сумме получается

$$n^2 + n$$

шагов. И снова за более подробным объяснением обращайтесь к упражнению 486.

ПРИМЕЧАНИЕ. Этот анализ предполагает, что сравнение двух элементов в списке занимает постоянное время. **КОНЕЦ.**

Наш последний пример – программа `inf` из раздела 16.2:

```
(define (inf l)
  (cond
    [(empty? (rest l)) (first l)]
    [else (if (< (first l) (inf (rest l)))
               (first l)
               (inf (rest l)))]))
```

Возьмем для начала короткий входной список: `(list 3 2 1 0)`. Мы знаем, что результатом будет число `0`. Вот первый важный шаг:

```
(inf (list 3 2 1 0))
==
(if (< 3 (inf (list 2 1 0)))
  3
  (inf (list 2 1 0)))
```

После этого мы должны выполнить первый рекурсивный вызов. Поскольку результат равен `0` и поэтому условное выражение возвращает `#false`, мы также должны выполнить рекурсию в ветке `else`.

После этого мы увидим двукратное вычисление выражения `(inf (list 1 0))`:

```
(inf (list 2 1 0))
==
(if (< 2 (inf (list 1 0))) 2 (inf (list 1 0)))
```

На этом этапе мы можем обобщить шаблон и свести в таблицу:

Оригинальное выражение	Требует дважды вычислить
<code>(inf (list 3 2 1 0))</code>	<code>(inf (list 2 1 0))</code>
<code>(inf (list 2 1 0))</code>	<code>(inf (list 1 0))</code>
<code>(inf (list 1 0))</code>	<code>(inf (list 0))</code>

В сумме механизм пошаговых вычислений выполняет восемь рекурсивных шагов при обработке списка с четырьмя элементами. Если добавить 4 в начало списка, то число рекурсивных шагов удвоится. Выражаясь языком алгебры, функции `inf` требуется выполнить порядка 2^n рекурсивных шагов для обработки списка с n числами, когда последнее число является наименьшим. Очевидно, что это наихудший случай для `inf`.

Стоп! Если приглядеться внимательнее, то можно заметить, что предыдущее утверждение не совсем верное. В действительности программе `inf` требуется выполнить 2^{n-1} рекурсивных шагов для обработки списка с n элементами. Нет ли здесь ошибки?

Дело в том, что на самом деле мы измеряем не точное время, когда говорим «порядка» и пропускаем время, затрачиваемое на выполнение всех встроенных предикатов, селекторов, конструкторов, арифметики и т. д., и сосредоточиваемся только на рекурсивных шагах. Теперь рассмотрим следующий расчет:

$$2^{n-1} = \frac{1}{2} \cdot 2^n.$$

Он показывает, что 2^{n-1} и 2^n отличаются в два раза, то есть выражение «порядка 2^{n-1} шагов» описывает `inf` в мире, где все базовые операции *SL выполняются вдвое медленнее, чем в мире, где программа `inf` выполняет «порядка 2^n шагов». В этом смысле эти два выражения действительно означают одно и то же. Вопрос в том, что именно они означают, и это тема следующего раздела.

Упражнение 484. Список, отсортированный в порядке убывания, является наихудшим случаем для `inf`, анализ абстрактного времени выполнения `inf` объясняет, почему реализация `inf` с использованием `local` сокращает время выполнения. Для удобства повторно воспроизведем эту версию здесь:

```
(define (infL l)
  (cond
    [(empty? (rest l)) (first l)]
    [else (local ((define s (infL (rest l))))
                 (if (< (first l) s) (first l) s))]))
```

Вычислите (`infL (list 3 2 1 0)`) в пошаговом режиме. Затем докажите, что `infL` выполняет «порядка n шагов» в лучшем и худшем случаях. Потом вернитесь к упражнению 261, в котором предлагается исследовать аналогичную проблему. ■

Упражнение 485. Числовое дерево – это либо число, либо пара числовых деревьев. Спроектируйте функцию `sum-tree`, которая вычисляет сумму чисел в дереве. Каково ее абстрактное время выполнения? Какой максимальный размер может иметь дерево? Какая форма дерева считается наихудшей? Какая – наилучшей? ■

Определение термина «порядка»

В предыдущем разделе упоминались все ключевые составляющие фразы, начинающейся со слова «порядка». Теперь пришло время дать строгое определение. Начнем с двух идей, представленных в предыдущем разделе:

1. Абстрактная мера производительности – это связь между двумя величинами: размером входных данных и количеством рекурсивных шагов, необходимых для определения ответа. Отношение на самом деле является математической функцией, ко-

В упражнении 245 решается другой вопрос, а именно можно ли сформулировать программу, которая определяет, равны ли две другие программы. В этом интермецо мы не будем писать такую программу, а воспользуемся простыми математическими выкладками.

- Следовательно, обобщенное утверждение о производительности программы – это утверждение о функции, а сравнение производительности двух программ требует сравнения двух таких функций.
- Как определить, какая функция из двух «лучше»?
- Вернемся к воображаемым программам из введения: `prog-linear` и `prog-square`. Они вычисляют одинаковые результаты, но имеют разную производительность. Программа `prog-linear` выполняет «порядка n шагов», в то время как `prog-square` – «порядка n^2 шагов». С математической точки зрения функция производительности для `prog-linear` имеет следующий вид:

$$L(n) = c_L \cdot n,$$

а для `prog-square` соответствующая функция производительности выглядит так:

$$S(n) = c_S \cdot n^2,$$

где c_L – это стоимость каждого рекурсивного шага в `prog-square`, а c_S – стоимость шага в `prog-linear`.

Допустим, мы выяснили, что $c_L = 1000$ и $c_S = 1$. Теперь можем свести эти абстрактные времена выполнения в таблицу, чтобы перейти к сравнению конкретных цифр:

n	10	100	1000	2000
<code>prog-square</code>	100	10000	1000000	4000000
<code>prog-linear</code>	10000	100000	1000000	2000000

По аналогии с графиком на рис. 32, если судить по первым столбцам таблицы, может показаться, что `prog-square` производительнее, чем `prog-linear`, потому что для обработки списков одинакового размера n функции `prog-square` требуется меньше времени, чем `prog-linear`. Но посмотрите на последний столбец. Когда длина входного списка становится достаточно большой, преимущество `prog-square` уменьшается и сходит на нет при длине входного списка, равной 1000. **После этого** `prog-square` начинает **проигрывать** функции `prog-linear`.

Это последнее открытие является ключом к точному определению фразы «порядка». Если функция f , определенная на натуральных числах, дает больший результат, чем некоторая функция g для **всех** натуральных чисел, то f явно больше, чем g . Но что, если это утверждение неверно для некоторых чисел, скажем для 1000 или 1 000 000, и верно для всех остальных? В этом случае мы все равно можем сказать, что f больше, чем g . И это подводит нас к следующему определению.

Определение. Если дана функция g от натуральных чисел, то $O(g)$ (произносится как « O большое от g ») является классом функций от натуральных чисел. Функция f принадлежит классу $O(g)$, если существуют числа c и $bigEnough$ такие, что

для всех $n \geq bigEnough$ выполняется условие $f(n) \leq c \cdot g(n)$:

Терминология. Если $f \in O(g)$, то можно сказать, что f не хуже, чем g .

Естественно, мы хотели бы проиллюстрировать это определение на примере `prog-linear` и `prog-square`. Вот как будут выглядеть функции производительности для `prog-linear` и `prog-square` после подстановки констант:

$$S(n) = 1 \cdot n^2$$

и

$$L(n) = 1000 \cdot n.$$

Идея состоит в том, чтобы найти магические числа c и $bigEnough$ такие, что $L \in O(S)$, которые подтвердили бы, что производительность `prog-square` не хуже, чем производительность `prog-linear`. А пока мы просто скажем вам, что это за числа:

$$bigEnough = 1000; c = 1.$$

Используя эти числа, мы должны показать, что

$$L(n) \leq 1 \cdot S(n)$$

для любого n больше 1000. Вот наше доказательство:

Выберите конкретное число n_0 , удовлетворяющее условию:

$$1000 \leq n_0.$$

Мы используем символическое имя n_0 , чтобы избежать любых конкретных предположений относительно числа. Из алгебры мы знаем, что обе части неравенства можно умножить на один и тот же положительный множитель, и неравенство по-прежнему будет выполняться. Используем для этого число n_0 :

$$1000 \cdot n_0 \leq n_0 \cdot n_0.$$

Теперь можно заметить, что левая часть неравенства равна $L(n_0)$, а правая часть – $S(n_0)$:

$$L(n_0) \leq S(n_0).$$

Поскольку n_0 – это обобщенное число правильного типа, мы показали именно то, что хотели показать.

Чтобы найти сами числа *bigEnough* и *c*, обычно проходят по этому доказательству в обратном порядке. Такого рода математические рассуждения весьма увлекательны, но мы оставим их курсу алгоритмов.

Определение *O* также объясняет с математической строгостью, почему не нужно обращать внимания на конкретные константы при сравнении абстрактного времени выполнения. Например, мы можем сделать каждый базовый шаг программы *prog-linear* в два раза быстрее, чтобы получить

$$S(n) = \frac{1}{2} \cdot n^2$$

и

$$L(n) = 1000 \cdot n.$$

Чтобы приведенное выше доказательство осталось верным, достаточно удвоить *bigEnough* до 2000.

Наконец, большинство людей используют *O* вместе с сокращенным обозначением функций. Тем самым они говорят, что время выполнения *how-many* равно *O(n)*, потому что они склонны думать об *n* как о сокращенном представлении (математической) функции *id(n) = n*. Такой подход приводит к утверждению, что время выполнения *sort* в наихудшем случае равно *O(n²)*, а *inf - O(2ⁿ)*, потому что *n²* является сокращенной формой представления функции *sqr(n) = n²*, а *2ⁿ* – сокращенной формой представления *expt(n) = 2ⁿ*.

Стоп! Что означает утверждение о том, что производительность функции равна *O(1)*?

Упражнение 486. В первом подразделе этой главы мы утверждали, что функция *f(n) = n² + n* принадлежит классу *O(n²)*. Определите пару чисел *c* и *bigEnough*, подтверждающих это утверждение. ■

Упражнение 487. Рассмотрим функции *f(n) = 2ⁿ* и *g(n) = 1000 · n*. Докажите, что *g* принадлежит классу *O(f)*, то есть, абстрактно говоря, что *f* дороже (или, по крайней мере, не дешевле), чем *g*. Если размер входных данных гарантированно находится в диапазоне от 3 до 12, то какая функция лучше? ■

Упражнение 488. Сравните *f(n) = n log(n)* и *g(n) = n²*. Принадлежит *f* классу *O(g)* или *g* классу *O(f)*? ■

Почему программы используют предикаты и селекторы?

Понятие «порядка» объясняет, почему рецепты проектирования позволяют создавать не только хорошо организованные, но и «эффективные» программы. Проиллюстрируем эту идею на единственном примере, спроектировав программу, которая ищет число в списке. Вот сигнатура, описание назначения и примеры, оформленные в виде тестов:

```

; Number [List-of Number] -> Boolean
; ищет число x в списке l

(check-expect (search 0 '(3 2 1 0)) #true)
(check-expect (search 4 '(3 2 1 0)) #false)

```

Вот два определения, которые соответствуют нашим ожиданиям:

<pre>(define (searchL x l) (cond [(empty? l) #false] [else (or (= (first l) x) (searchL x (rest l))))]))</pre>	<pre>(define (searchS x l) (cond [(= (length l) 0) #false] [else (or (= (first l) x) (searchS x (rest l))))]))</pre>
--	--

Программа слева соответствует рецепту проектирования. В частности, разработка макета требует использования структурных предикатов для каждого варианта в определении данных. Следуя этому правилу, вы получите программу с условным выражением `cond`, первая ветвь в которой обрабатывает пустые списки, а вторая – все остальные. Условие в первой строке в `cond` отвечает на вопрос `empty?`, а во второй – `cons?`, что фактически соответствует `else`.

Программа `searchS` не соответствует рецепту структурного проектирования. Вместо этого она основана на идеи о том, что списки – это контейнеры, имеющие размер. То есть программа может сравнить этот размер с 0, что эквивалентно проверке `empty?`.

*В действительности
в ней используется
генеративная рекурсия.*

С функциональной точки зрения эта идея верна, но она предполагает, что стоимость операций в *SL является фиксированной и постоянной. Однако если функция `length` похожа на `how-many`, то `searchS` будет работать медленнее, чем `searchL`. Используя нашу новую терминологию, можно сказать, что для обработки списка с n элементами `searchL` выполняет $O(n)$ рекурсивных шагов, а `searchS` – $O(n^2)$. Проще говоря, использование произвольных операций *SL для формулирования условий может привести к смещению производительности из одного класса функций в другой, гораздо худший.

Давайте завершим это интермеццо экспериментом, в ходе которого проверим, является ли `length` функцией с постоянным временем выполнения или ей требуется время, пропорциональное длине списка. Самый простой способ – определить программу, которая создает длинный список и определяет, сколько времени выполняется каждая версия программы поиска:

```

; N -> [List Number Number]
; определяет, сколько времени требуется функциям searchS и searchL,
; чтобы найти n в (list 0 ... (- n 1))
(define (timing n)
  (local ((define long-list
              (build-list n (lambda (x) x))))
    (list
      (time (searchS n long-list))
      (time (searchL n long-list)))))
```

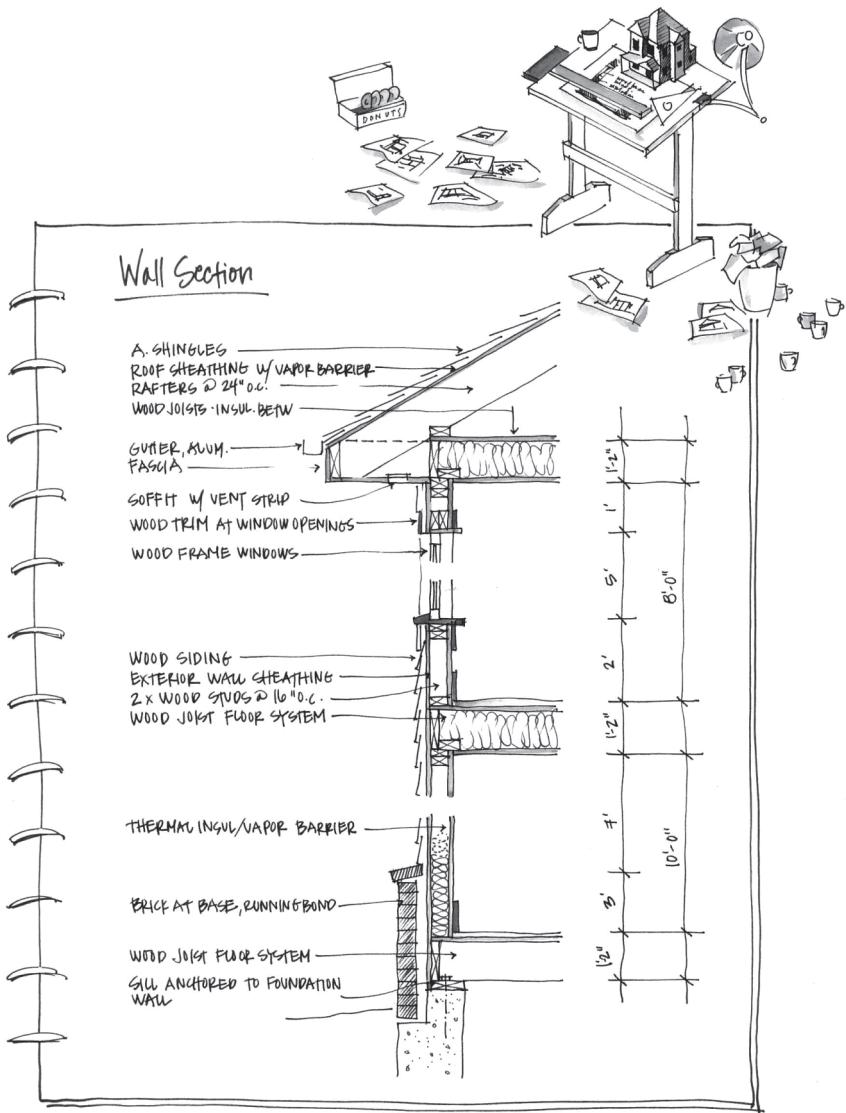
Теперь запустите эту программу с аргументами 10000 и 20000. Если `length` действует подобно `empty?`, то время второго прогона будет примерно вдвое больше, чем первого; иначе время выполнения `searchS` резко увеличится.

Стоп! Выполните данный эксперимент.

См. раздел 33.2, где рассказывается, как другие языки определяют размер контейнера.

Если вы выполнили этот эксперимент, то теперь наверняка знаете, что время выполнения `length` пропорционально размеру списка. Буква «`S`» в `searchS` означает «`squared`» (квадратичная), потому что время его работы равно $O(n^2)$.

Но не спешите с выводами: такого рода рассуждения справедливы не для каждого языка программирования. Многие языки работают с контейнерами иначе, чем `*SL`. Чтобы понять эту идею, вы должны познакомиться с еще одной концепцией проектирования – аккумуляторами, о которых мы расскажем в заключительной части данной книги.



VI АККУМУЛЯТОРЫ

Функция также может зациклиться или завершиться сообщением об ошибке, но мы игнорируем такие возможности.

Мы также игнорируем `random`, которая является исключением из этого правила.

Функция всегда действует в соответствии с заявлением о назначении, и это все, что вам нужно знать.

Этот принцип независимости от контекста играет важную роль в разработке рекурсивных функций. Приступая к проектированию, вы можете смело предположить, что функция вычисляет то, что обеспечает заявление о назначении, даже если функция еще не определена. В частности, вы можете использовать результаты рекурсивных вызовов в коде некоторой функции, обычно в одном из предложений в выражении `cond`. Шаги рецептов проектирования как для структурного, так и для генеративно-рекурсивного, связанные с созданием макета и написанием кода, основываются на этой идее.

Независимость от контекста упрощает проектирование функций, но она вызывает две проблемы. В общем случае независимость от контекста вызывает потерю знаний во время рекурсивных вызовов; функция не «знает», применяется ли она к полному списку или к его части. Для структурно-рекурсивных программ такая потеря знаний означает, что им, возможно, придется обрабатывать одни и те же данные несколько раз, что снижает производительность. Для генеративно-рекурсивных функций потеря означает, что функция может вообще не вычислить результат. Эта вторая проблема была продемонстрирована в предыдущей части на примере функции обхода графа, которая не может найти путь между двумя узлами в циклическом графе.

В этой части мы представим вашему вниманию вариант рецептов проектирования, решающих проблему «потери контекста». Поскольку мы хотим сохранить принцип, согласно которому $(f a)$ всегда возвращает один и тот же результат, независимо от частоты и места его вычисления, единственным решением является **добавление аргумента, представляющего контекст** вызова функции. Мы называем этот дополнительный аргумент **аккумулятором**. Во время обхода данных рекурсивные вызовы продолжают получать обычные аргументы, в то время как аккумуляторы меняются в зависимости от них и от контекста.

Правильно проектировать функции с аккумуляторами намного сложнее, чем любые другие функции, которые мы рассматривали в предыдущих главах. Идея состоит в том, чтобы понять взаимосвязь между обычными аргументами и аккумуляторами. В следующих главах объясняется, как создавать функции с аккумуляторами и как они работают.

Применяя некоторую функцию f к аргументу a в языке ISL+, вы обычно получаете некоторый результат v . Если снова вычислить $(f a)$, то вы снова получите v . Фактически вы будете получать v независимо от того, как часто вызываете $(f a)$. Не имеет значения, применяется ли функция в первый или в сотый раз, выполняется ли она в области взаимодействий DrRacket или внутри другой функции.

31. Потеря знаний

Функции, спроектированные с использованием структурного или генеративного рецепта, страдают от потери знаний, хотя и по-разному. В этой главе мы объясним на двух примерах – по одному для каждой категории, – как отсутствие знаний о контексте влияет на выполнение функций. Первый раздел будет посвящен структурной рекурсии, второй – генеративной.

31.1. Проблема структурной обработки

Начнем с, казалось бы, простого примера.

Постановка задачи. Вы работаете в команде геодезистов, измеряющих длину разных участков дороги. К вам обратились с просьбой спроектировать программу, которая преобразует относительные расстояния между точками на дороге в абсолютные расстояния от некоторой начальной точки.

Например, вам может быть предложена такая схема дороги:



Каждое число определяет расстояние между двумя точками. Вы должны получить следующую схему, на которой каждая точка отмечена расстоянием от самого левого конца:



Проектирование программы, выполняющей эти вычисления, является простым упражнением в проектировании структурных функций. Полный код программы показан в листинге 114. Если исходный список непустой, естественная рекурсия вычисляет абсолютное расстояние до остальных точек от первой точки в (`rest l`).

Поскольку первый элемент в списке не всегда является истинной начальной точкой и находится на некотором расстоянии (`first l`) от начала координат, мы должны прибавить (`first l`) к каждому числу, получаемому в ходе естественной рекурсии. Этот второй шаг – прибавление числа к каждому элементу в списке чисел – требует вспомогательной функции.

Листинг 114. Преобразование относительных расстояний в абсолютные

```
| ; [List-of Number] -> [List-of Number]
| ; преобразует относительные расстояния в списке в абсолютные
| ; первое число в списке представляет расстояние от начала координат
```

```
(check-expect (relative->absolute '(50 40 70 30 30))
              '(50 90 160 190 220))

(define (relative->absolute l)
  (cond
    [(empty? l) '()]
    [else (local ((define rest-of-l
                      (relative->absolute (rest l)))
                  (define adjusted
                      (add-to-each (first l) rest-of-l)))
                 (cons (first l) adjusted))])))

; Number [List-of Number] -> [List-of Number]
; прибавляет n к каждому числу в l

(check-expect (cons 50 (add-to-each 50 '(40 110 140 170)))
              '(50 90 160 190 220))

(define (add-to-each n l)
  (cond
    [(empty? l) '()]
    [else (cons (+ (first l) n) (add-to-each n (rest l))))]))
```

Программа относительно проста, но ее применение к большим спискам обнаруживает проблему. Рассмотрим вычисление следующего выражения:

```
| (relative->absolute (build-list size add1))
```

С увеличением размера списка время, необходимое для его обработки, увеличивается в геометрической прогрессии:

size	1000	2000	3000	4000	5000	6000	7000
время	25	109	234	429	689	978	1365

Времена будут различаться от компьютера к компьютеру и года к году. Эти измерения проводились в 2017 году на MacMini, действующем под управлением OS X 10.11; перед этим измерения проводились в 1998 году на другом компьютере, и время было в 100 раз больше.

При переходе от списка с 1000 элементов к списку с 2000 элементов время увеличилось в четыре раза. При мерно такое же соотношение наблюдается при переходе от 2000 к 4000 и т. д. Используя терминологию из интермеш-цио 5, можно сказать, что производительность функции равна $O(n^2)$, где n – длина заданного списка.

Упражнение 489. Перепишите `add-to-each`, используя `map` и `lambda`. ■

Упражнение 490. Разработайте формулу, описывающую абстрактное время выполнения `relative->absolute`. **Подсказка.** Вычислите выражение

```
| (relative->absolute (build-list size add1))
```

на бумаге. Начните со значений 1, 2 и 3 для `size`. Сколько рекурсивных вызовов `relative->absolute` и `add-to-each` требуется в каждом случае? ■

Объем работы, выполняемой программой для такой простой задачи, вызывает удивление. Если бы мы преобразовывали тот же список вручную, то мы бы суммировали общее расстояние и просто добав-

ляли его к относительным расстояниям на каждом следующем шаге. Почему программа не может этого сделать?

Попробуем спроектировать версию функции, близкую к ручному методу. Как обычно, начнем с макета обработки списка:

```
(define (relative->absolute/a l)
  (cond
    [(empty? l) ...]
    [else
      (... (first l) ...
            ... (relative->absolute/a (rest l)) ...)]))
```

Теперь сымитируем вычисления вручную:

```
(relative->absolute/a (list 3 2 7))
== (cons ... 3 ... (relative->absolute/a (list 2 7)))
== (cons ... 3 ...
          (cons ... 2 ...
                (relative->absolute/a (list 7))))
== (cons ... 3 ...
          (cons ... 2 ...
                (cons ... 7 ...
                      (relative->absolute/a '()))))
```

Очевидно, что первым элементом в списке результатов должно быть число 3, и сконструировать этот список несложно. Но вторым элементом должно быть число (+ 3 2), однако второй рекурсивный вызов `relative->absolute/a` не может «знать», что первым элементом исходного списка является 3. «Знание» потеряно.

Проблема в том, что рекурсивные функции не зависят от своего контекста. Функция обрабатывает L в (`cons N L`) так же, как в (`cons K L`). Если бы L было задано само по себе, то тогда список можно было бы обработать таким образом.

Чтобы компенсировать потерю «знаний», мы добавим в функцию дополнительный параметр: `accu-dist`. Он представляет накопленное расстояние, которое мы храним при преобразовании списка относительных расстояний в список абсолютных расстояний, и должен иметь начальное значение 0. По мере продвижения по списку функция должна прибавлять числа из него к этому аккумулятору.

Бот новое определение:

```
(define (relative->absolute/a l accu-dist)
  (cond
    [(empty? l) '()]
    [else
      (local ((define tally (+ (first l) accu-dist)))
        (cons tally
              (relative->absolute/a (rest l) tally))))])
```

Рекурсивный вызов получает оставшуюся часть списка и новое абсолютное расстояние от текущей точки до начала координат. Оба аргумента меняются при каждом вызове, но изменение второго строго зависит от первого. Функция по-прежнему является простой процедурой обработки списка.

А теперь еще раз вычислим наш рабочий пример:

```
| (relative->absolute/a (list 3 2 7))
| == (relative->absolute/a (list 3 2 7) 0)
| == (cons 3 (relative->absolute/a (list 2 7) 3))
| == (cons 3 (cons 5 (relative->absolute/a (list 7) 5)))
| == (cons 3 (cons 5 (cons 12 ???)))
| == (cons 3 (cons 5 (cons 12 '())))
```

Стоп! Подставьте корректное выражение вместо вопросительных знаков в строке 4.

Пошаговые вычисления наглядно показывают, насколько использование аккумулятора упрощает процесс преобразования. Каждый элемент в списке обрабатывается ровно один раз. Когда `relative->absolute/a` достигает конца списка, результат оказывается полностью определен, и дальнейшая обработка не требуется. В общем случае для списка с N элементами функция выполняет порядка N шагов естественной рекурсии.

Одна из проблем заключается в том, что, в отличие от `relative->absolute`, новая функция принимает два аргумента, а не один. Хуже того, кто-то может случайно злоупотребить `relative->absolute/a`, применив ее к списку чисел и числу, которое не равно 0. Мы можем решить обе проблемы, определив функцию, которая использует определение `local` для инкапсуляции `relative->absolute/a`. Результат показан в листинге 115. Теперь `relative->absolute` неотличима от `relative->absolute.v2` в отношении входных аргументов.

Листинг 115. Преобразование относительных расстояний с использованием аккумулятора

```
; [List-of Number] -> [List-of Number]
; преобразует относительные расстояния в списке в абсолютные
; первое число в списке представляет расстояние от начала координат

(check-expect (relative->absolute.v2 '(50 40 70 30 30))
              '(50 90 160 190 220))
(define (relative->absolute.v2 l0)
  (local (
    ; [List-of Number] Number -> [List-of Number]
    (define (relative->absolute/a l accu-dist)
      (cond
        [(empty? l) '()]
        [else
          (local ((define accu (+ (first l) accu-dist)))
            (cons accu
                  (relative->absolute/a (rest l) accu))))])
    (relative->absolute/a l0 0)))
```

Теперь оценим производительность этой версии программы. Для этого вычислим

```
| (relative->absolute.v2 (build-list size add1))
```

с разными значениями `size` и представим результаты в виде таблицы:

size	1000	2000	3000	4000	5000	6000	7000
время	0	0	0	0	0	1	1

Удивительно, но ни в одном из запусков `relative->absolute.v2` не выполнялась дольше одной секунды, даже когда ей был передан список с 7000 чисел. Сравнивая эту производительность с производительностью `relative->absolute`, можно подумать, что аккумуляторы – чудодейственное лекарство для всех медленно работающих программ. К сожалению, это не так, но всякий раз, когда выясняется, что структурно-рекурсивная функция повторно обрабатывает результат естественной рекурсии, обязательно следует рассмотреть возможность использования аккумуляторов. В этом конкретном случае производительность улучшилась с $O(n^2)$ до $O(n)$.

Упражнение 491. Немного поработав и поразмыслив, ваш друг пришел к следующему решению нашей задачи:

```
(define (relative->absolute l)
  (reverse
    (foldr (lambda (f l) (cons (+ f (first l)) l))
           (list (first l))
           (reverse (rest l)))))
```

Это упражнение предложили Адриан Герман (Adrian German) и Мардин Ядегар (Mardin Yadegar).

В этом простом решении используются хорошо известные функции ISL+: `reverse` и `foldr`. Как известно, `lambda` – это всего лишь удобство. Вы также можете вспомнить из части III, что `foldr` можно спроектировать с применением рецептов проектирования, представленных в первых двух частях книги.

Означает ли решение, найденное вашим другом, что нет никакой необходимости использовать приемы проектирования, описываемые в этом разделе? Чтобы узнать ответ, загляните в раздел 32.1, но сначала подумайте над вопросом. **Подсказка.** Попробуйте самостоятельно спроектировать функцию `reverse`. ■

31.2. Проблема генеративной рекурсии

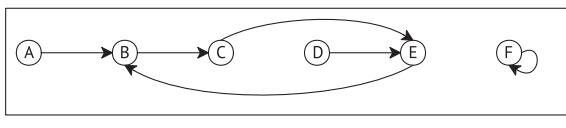
Вернемся к задаче «путешествия» внутри графа.

Постановка задачи. Спроектируйте алгоритм, который проверяет, связаны ли два узла в простом графе. В таком графе каждый узел имеет ровно одну направленную связь с другим узлом и, возможно, с самим собой.

В главе 29 мы уже рассматривали алгоритм поиска пути в графике. Эта задача проще, потому что в данном разделе основное внимание уделяется алгоритмам с аккумулятором.

Рассмотрим пример графа на рис. 33. В нем имеется шесть узлов, от A до F , и шесть ребер. Путь от A до E проходит через узлы B и C . Однако в графике нет пути от A до F или от любого другого узла, кроме самого узла F .

Справа на рис. 33 показано, как представить этот граф в виде вложенных списков. Каждый узел представлен списком из двух символов.



```
(define a-sg
  '((A B)
    (B C)
    (C E)
    (D E)
    (E B)
    (F F)))
```

Рис. 33. Простой граф

Первый символ – это метка узла; второй – единственный узел, до-ступный из первого. Вот соответствующие определения данных:

```

; SimpleGraph - это [List-of Connection]
; Connection - это список с двумя элементами:
;   (list Node Node)
; Node - это символ (Symbol).
```

Это прямой перевод нашего неформального описания.

Мы уже знаем, что задача должна решаться с использованием генеративной рекурсии, поэтому легко приходим к следующему заголовку:

```

; Node Node SimpleGraph -> Boolean
; проверяет наличие пути из узла origin в узел destination
; в простом графе sg

(check-expect (path-exists? 'E a-sg) #true)
(check-expect (path-exists? 'A 'F a-sg) #false)

(define (path-exists? origin destination sg)
  #false)
```

Ответим на четыре основных вопроса рецепта генеративной ре-курсии:

- задача тривиальна, если `origin` совпадает с `destination`;
- тривиальное решение: `#true`;
- если `origin` не совпадает с `destination`, мы можем сделать только одно: перейти к ближайшему соседу и найти путь к `destination` оттуда;
- если решение новой задачи найдено, то больше ничего делать не нужно. Если сосед узла `origin` связан с `destination`, то, значит, и `origin` связан с `destination`. Иначе связи нет.

Затем просто выразим эти ответы на ISL+, чтобы получить полно-ценную программу.

Листинг 116. Поиск пути в простом графе

```

; Node Node SimpleGraph -> Boolean
; проверяет наличие пути из узла origin в узел destination
```

```

; в простом графе sg

(check-expect (path-exists? 'A 'E a-sg) #true)
(check-expect (path-exists? 'A 'F a-sg) #false)

(define (path-exists? origin destination sg)
  (cond
    [(symbol=? origin destination) #t]
    [else (path-exists? (neighbor origin sg)
                         destination
                         sg)]))

; Node SimpleGraph -> Node
; определяет узел, связанный с узлом a-node в sg

(check-expect (neighbor 'A a-sg) 'B)
(check-error (neighbor 'G a-sg) "neighbor: not a node")

(define (neighbor a-node sg)
  (cond
    [(empty? sg) (error "neighbor: not a node")]
    [else (if (symbol=? (first (first sg)) a-node)
              (second (first sg))
              (neighbor a-node (rest sg))))]))

```

В листинге 116 представлена законченная программа, включая функцию поиска соседа узла в простом графе – обычную структурно-рекурсивную функцию – и тестовые примеры для обоих возможных результатов. Но не запускайте эту программу. А если решите сделать это, то будьте готовы принудительно остановить ее. Действительно, даже поверхностный взгляд на функцию наводит на мысль, что в ней есть проблема. Функция должна возвращать `#false`, если в графе нет пути от `origin` до `destination`, но в программе нигде нет `#false`. Кроме того, мы должны выяснить, что на самом деле делает функция, если нет возможности перейти из `origin` в `destination`.

Еще раз взгляните на рис. 33. В этом простом графе нет пути из `C` в `D`. Ребро, исходящее из `C`, обходит стороной узел `D` и соединяет `C` с `E`. А теперь рассмотрим вычисления по шагам:

```

(path-exists? 'C 'D '((A B) ... (F F)))
== (path-exists? 'E 'D '((A B) ... (F F)))
== (path-exists? 'B 'D '((A B) ... (F F)))
== (path-exists? 'C 'D '((A B) ... (F F)))

```

Как видите, функция снова и снова вызывает себя с одними и теми же аргументами. Иначе говоря, вычисление не останавливается.

Проблема с `path-exists?` снова связана с потерей «знания», с которой мы уже сталкивались при проектировании `relative->absolute`. Так же как в случае с `relative->absolute`, при создании `path-exists?` мы использовали рецепт проектирования и предположили, что рекурсивные вызовы не зависят от их контекста. В случае `path-exists?` это означает, в частности, что функция не «знает», получило ли предыдущее применение в текущей цепочке рекурсий те же самые аргументы.

Решается эта проблема так же, как в предыдущем разделе. Нужно добавить параметр с именем `seen`, который представляет накопленный список начальных узлов, уже посещавшихся функцией, начиная с самого первого. Первоначально параметр должен иметь значение `'()`. Когда функция проверит конкретный узел `origin` и решит переместиться в соседний узел, она должна добавить `origin` в `seen`.

Вот первая версия `path-exists?`, которой мы дадим имя `path-exists?/a`:

```
; Node Node SimpleGraph [List-of Node] -> Boolean
; проверяет наличие пути из узла origin в узел destination
; предположение: из узлов в seen нет пути в destination
(define (path-exists?/a origin destination sg seen)
  (cond
    [(symbol=? origin destination) #true]
    [else (path-exists?/a (neighbor origin sg)
                           destination
                           sg
                           (cons origin seen))]))
```

Простое добавление параметра не решает нашей проблемы, но, как показывают вычисления в пошаговом режиме, у нас теперь есть от чего оттолкнуться:

```
(path-exists?/a 'C 'D '((A B) ... (F F)) '())
== (path-exists?/a 'E 'D '((A B) ... (F F)) '(C))
== (path-exists?/a 'B 'D '((A B) ... (F F)) '(E C))
== (path-exists?/a 'C 'D '((A B) ... (F F)) '(B E C))
```

В отличие от первоначальной функции, эта версия больше не вызывает себя с теми же аргументами. Хотя три аргумента в третьем рекурсивном применении совпадают с тремя аргументами в самом первом вызове, аргумент аккумулятора отличается: вместо `'()` он содержит `'(B E C)`. Новое значение говорит нам, что во время поиска пути от 'C к 'D функция проверила узлы 'B, 'E и 'C.

Все, что нам осталось сделать, – это заставить алгоритм использовать накопленные знания. В частности, алгоритм может определить, посещался ли прежде данный узел `origin`. Если да, то задача тривиально разрешима, то есть она имеет тривиальное решение `#false`. В листинге 117 показано определение `path-exists.v2?` – обновленной версии `path-exists?`. Определение ссылается на функцию `member?` – стандартную функцию `ISL+`.

В определении `path-exists.v2?` также устраниены две незначительные проблемы, имевшие место в первой версии. Локализовав определение функции с аккумулятором, мы можем гарантировать, что первый вызов всегда будет получать `'()` в качестве начального значения `seen`. А `path-exists.v2?` соответствует той же сигнатуре и заявлению о назначении, что и `path-exists?`.

И все же между `path-exists.v2?` и `relative->absolute.v2` существенная разница: `relative->absolute.v2` эквивалентна исходной версии функции, а `path-exists.v2?` является улучшенной версией `path-exists?`.

Последняя может не найти ответа в некоторых входных данных, а `path-exists.v2?` находит решение для любого простого графа.

Упражнение 492. Измените определения в листинге 111 так, чтобы программа возвращала `#false`, если она дважды встречает один и тот же исходный узел. ■

Листинг 117. Поиск пути в простом графе с аккумулятором

```
; Node Node SimpleGraph -> Boolean
; проверяет наличие пути из узла origin в узел destination в sg

(check-expect (path-exists.v2? 'A 'E a-sg) #true)
(check-expect (path-exists.v2? 'A 'F a-sg) #false)

(define (path-exists.v2? origin destination sg)
  (local (; Node Node SimpleGraph [List-of Node] -> Boolean
          (define (path-exists?/a origin seen)
            (cond
              [(symbol=? origin destination) #t]
              [(member? origin seen) #f]
              [else (path-exists?/a (neighbor origin sg)
                                    (cons origin seen))]))))
  (path-exists?/a origin '())))
```

32. Проектирование функций с аккумулятором

Предыдущая глава на двух примерах показала необходимость накопления дополнительных знаний. В одном случае накопление упрощает функцию и помогает получить результат намного быстрее. Во втором примере накопление было необходимо для организации правильной работы функции. Однако в обоих случаях необходимость накопления становится очевидной только после получения функций, спроектированных по всем правилам.

Обобщая предыдущую главу, можно предположить, что проектирование функций с аккумуляторами имеет два основных аспекта:

- 1) определить, выиграет ли функция от использования аккумулятора;
- 2) понять, что представляет аккумулятор.

Первые два раздела этой главы будут посвящены этим двум вопросам. Поскольку второй вопрос особенно сложный, в третьем разделе мы проиллюстрируем его серией примеров преобразования обычных функций в функции с аккумуляторами.

32.1. Условия применения аккумулятора

Выявление необходимости аккумуляторов – непростая задача. Мы уже видели две причины, и они наиболее распространены в практике. В любом случае очень важно сначала создать законченную функцию, следуя **обычному** рецепту проектирования, а затем изучить ее, действуя следующим образом.

1. Если структурно-рекурсивная функция повторно вычисляет результат естественной рекурсии с помощью вспомогательной рекурсивной функции, подумайте о возможности использования параметра-аккумулятора.

Листинг 118. Проектирование функций с аккумулятором, структурный пример

```
; [List-of X] -> [List-of X]
; конструирует на основе alox (A List Of X) список с обратным расположением элементов

(check-expect (invert '(a b c)) '(c b a))

(define (invert alox)
  (cond
    [(empty? alox) '()]
    [else
      (add-as-last (first alox) (invert (rest alox))))]

; X [List-of X] -> [List-of X]
```

```
; добавляет an-x в конец alox

(check-expect (add-as-last 'a '(c b)) '(c b a))

(define (add-as-last an-x alox)
  (cond
    [(empty? alox) (list an-x)]
    [else
      (cons (first alox) (add-as-last an-x (rest alox))))]))
```

Взгляните на определение `invert` в листинге 118. Результатом рекурсивного применения является копия остальной части списка с элементами, расположенными в обратном порядке. Она использует `add-as-last` для добавления первого элемента в этот перевернутый список и, таким образом, создает перевернутую версию всего списка. Эта вторая вспомогательная функция тоже является рекурсивной. Таким образом, мы обнаружили кандидата для аккумулятора.

Теперь давайте выполним некоторые вычисления в пошаговом режиме, как было показано в разделе 31.1, и посмотрим, поможет ли добавление аккумулятора в этом случае. Итак:

```
(invert '(a b c))
== (add-as-last 'a (invert '(b c)))
== (add-as-last 'a (add-as-last 'b (invert '(c)))))
== ...
== (add-as-last 'a (add-as-last 'b '(c)))
== (add-as-last 'a '(c b))
== '(c b a)
```

Стоп! Замените многоточие двумя недостающими шагами. После этого вы увидите, что `invert` в конечном итоге достигает конца заданного списка – точно так же, как `add-as-last`, – и если бы она знала, какие элементы туда помещать, ей не пришлось бы обращаться к вспомогательной функции.

2. В случае с генеративно-рекурсивной функцией перед нами встает гораздо более сложная задача. Наша цель должна заключаться в том, чтобы понять, может ли алгоритм не дать результата для каких-то входных данных, для которых он должен дать результат. Если да, то в таком случае добавление параметра, накапливающего знания, может помочь исправить проблему. Но давайте отложим такие сложные ситуации до главы 33.

Упражнение 493. Докажите, что в терминологии интермеццо 5 функция `invert` требует $O(n^2)$ времени для обработки списка из n элементов. ■

Упражнение 494. Нужен ли аккумулятор для функции сортировки вставкой `sort>` из раздела 11.3? Если да, то почему? Если нет, то почему? ■

32.2. Добавление аккумуляторов

После того как вы решили, что в существующую функцию следует добавить аккумулятор, выполните следующие два шага:

- определите, какое знание должен представлять аккумулятор, какое представление данных использовать и как это знание отражается в данных.

Например, для преобразования относительных расстояний в абсолютные достаточно накопить общее расстояние. Поскольку функция обрабатывает список относительных расстояний, она прибавляет каждое новое найденное относительное расстояние к текущему значению аккумулятора. Для задачи поиска пути в графе аккумулятор запоминает каждый посещенный узел. По мере того как функция поиска пути обходит узлы графа, она добавляет каждый новый узел в аккумулятор.

В общем случае нужно действовать следующим образом.

1. Создайте макет функции с аккумулятором:

```
; Domain -> Range
(define (function d0)
  (local (; Domain AccuDomain -> Range
         ; аккумулятор ...
         (define (function/a d a)
           ...))
    (function/a d0 a0)))
```

Исследуйте порядок выполнения функции по шагам, чтобы понять природу аккумулятора.

2. Определите тип данных, хранящихся в аккумуляторе. Опишите аккумулятор как связующее звено между аргументом *d* вспомогательной функции *function/a* и исходным аргументом *d0*.
- Примечание.** В ходе вычислений связь остается постоянной, или **инвариантной**. Из-за этого свойства описание аккумулятора часто называют **инвариантом**.
3. Используйте инвариант, чтобы определить начальное значение *a0* для *a*.
 4. Также используйте инвариант, чтобы определить, как вычислять аккумулятор для рекурсивных вызовов функции в пределах определения *function/a*;
- используйте знание, накапливаемое в аккумуляторе, для проектирования функции *function/a*.

В структурно-рекурсивных функциях значение аккумулятора обычно используется в базовом случае, то есть в предложении *cond*, не содержащем рекурсии. В генеративно-рекурсивных функциях накопленные знания могут использоваться в существующем базовом случае, в новом базовом случае или в ветвях *cond*, выполняющих генеративную рекурсию.

Как видите, ключевым моментом является точное описание роли аккумулятора. Поэтому старайтесь практиковать этот навык.

Рассмотрим обратный пример:

```
(define (invert.v2 alox0)
  (local ( ; [List-of X] ??? -> [List-of X]
          ; конструирует на основе alox список с обратным
          ; расположением элементов
          ; аккумулятор ...
          (define (invert/a alox a)
            (cond
              [(empty? alox) ...]
              [else
                (invert/a (rest alox) ... a ...)])))
  (invert/a alox0 ...))
```

Как было показано в предыдущем разделе, этого макета достаточно, чтобы исследовать порядок выполнения по шагам такого применения функции, как

```
| (invert '(a b c))
```

Вот основная идея:

```
== (invert/a '(a b c) a0)
== (invert/a '(b c) ... 'a ... a0)
== (invert/a '(c) ... 'b ... 'a ... a0)
== (invert/a '() ... 'c ... 'b ... 'a ... a0)
```

Эта схема предполагает, что `invert/a` может запомнить все элементы, которые видела в списке. Начальным значением, очевидно, должен быть пустой список '(); обновление аккумулятора внутри `invert/a` с помощью `cons` дает точно желаемое значение, когда `invert/a` достигает '().

Вот дополненный макет, включающий эти идеи:

```
(define (invert.v2 alox0)
  (local ( ; [List-of X] [List-of X] -> [List-of X]
          ; конструирует на основе alox список с обратным
          ; расположением элементов
          ; аккумулятор a - это список со всеми уже обработанными
          ; элементами из alox, расположенными в обратном порядке
          (define (invert/a alox a)
            (cond
              [(empty? alox) a]
              [else
                (invert/a (rest alox)
                          (cons (first alox) a)))))
  (invert/a alox0 '()))
```

Тело локального определения инициализирует аккумулятор значением '(), а рекурсивный вызов использует `cons`, чтобы добавить текущий головной элемент из `alox` в аккумулятор. В базовом случае `invert/a` применяет накопленные в аккумуляторе знания, т. е. обратный список.

Обратите внимание, что `invert.v2` просто перемещается по списку, тогда как `invert` повторно обрабатывает каждый результат своей естественной рекурсии с помощью `add-as-last`. Стоп! Оцените, насколько `invert.v2` работает быстрее, чем `invert`.

Терминология. Обсуждая функции, использующие параметр аккумулятора, программисты называют их *функциями с аккумуляторами*. Примеры функций с аккумуляторами: `relative->absolute/a`, `invert/a` и `path-exists?/a`.

32.3. Преобразование простых функций в функции с аккумуляторами

Описать аккумулятор непросто, но без хорошей формулировки инварианта невозможно понять, как работает функция с аккумулятором. Поскольку одна из целей программиста состоит в том, чтобы убедиться, что все, кто будет читать код, смогут понять его, развитие этого навыка имеет решающее значение. А формулирование инвариантов требует большой практики.

Цель этого раздела – научить правильно формулировать описание аккумуляторов на трех практических примерах: функции суммирования, функции вычисления факториала и функции обхода дерева. Все эти примеры связаны с преобразованием структурно-рекурсивной функции в функцию с аккумулятором. Фактически ни одна из них не требует обязательного использования параметра аккумулятора. Но они просты и понятны, и благодаря отсутствию других отвлекающих факторов эти примеры позволят нам сосредоточиться на формулировке инварианта аккумулятора.

В качестве первого примера рассмотрим следующие определения функции `sum`:

```
(define (sum.v1 alon) ; A List Of Numbers
  (cond
    [(empty? alon) 0]
    [else (+ (first alon) (sum.v1 (rest alon))))]))
```

Вот первый шаг к версии с аккумулятором:

```
(define (sum.v2 alon0)
  (local (; [List-of Number] ??? -> Number
          ; вычисляет сумму чисел в alon
          ; аккумулятор ...
          (define (sum/a alon a)
            (cond
              [(empty? alon) ...]
              [else (... (sum/a (rest alon)
                            ... a ...) ...)])))
          (sum/a alon0 ...)))
```

Стоп! Добавьте сигнатуру и тестовый пример, который подходит для обеих функций.

Как было предложено в первом шаге, мы включили макет `sum/a` в локальное определение, добавили параметр аккумулятора и переименовали параметр функции `sum`.

Листинг 119. Последовательность вычислений в макете функции с аккумулятором

<pre>(sum.v1 '(10 4)) == (+ 10 (sum.v1 '(4))) == (+ 10 (+ 4 (sum.v1 '()))) == (+ 10 (+ 4 (+ 0))) ... == 14</pre>	<pre>(sum.v2 '(10 4)) == (sum/a '(10 4) a0) == (sum/a '(4) ... 10 ... a0) == (sum/a '() ... 4 ... 10 ... a0) ... == 14</pre>
--	--

В листинге 119 показаны два сеанса пошаговых вычислений. Сравнение сразу же подсказывает главную идею, а именно `sum/a` может использовать аккумулятор для суммирования встречающихся чисел. Что касается инварианта аккумулятора, то вычисления показывают, что `a` содержит сумму чисел, от начала списка до текущего:

`a` – это сумма чисел из `alon0`, недостающих в `alon`.

Например, инвариант заставляет соблюдать следующие отношения:

если	<code>alon0</code>	содержит	'(10 4 6)	'(10 4 6)	'(10 4 6)
и	<code>alon</code>	содержит	'(4 6)	'(6)	'()
то	<code>a</code>	должно быть	10	14	20

Опираясь на этот инвариант, спроектировать все остальное не составляет труда:

```
(define (sum.v2 alon0)
  (local ( ; [List-of Number] ??? -> Number
          ; вычисляет сумму чисел в alon
          ; аккумулятор a – это сумма чисел
          ; из alon0, недостающих в alon
          (define (sum/a alon a)
            (cond
              [(empty? alon) a]
              [else (sum/a (rest alon)
                            (+ (first alon) a)))])))
  (sum/a alon0 0)))
```

Когда функция `sum/a` получает пустой список `'()` в параметре `alon`, она возвращает `a`, потому что он содержит сумму всех чисел из `alon`. Инвариант также подразумевает, что `0` является начальным значением для `a0`, а `+ обновляет` аккумулятор, прибавляя число, которое вот-вот будет «забыто» – `(first alon)`, – в аккумулятор `a`.

Упражнение 495. Завершите пошаговые вычисления `(sum/a '(10 4) 0)` в листинге 119. Это поможет вам увидеть, как `sum` и `sum.v2` складывают указанные числа в обратном порядке. Функция `sum` складывает числа справа налево, а версия с аккумулятором – слева направо.

Примечание о числах. Напомним, что для точных чисел эта разница не влияет на окончательный результат. Но для неточных чисел

Функцию вычисления факториала удобно использовать для анализа алгоритмов.

результаты сложения слева направо и справа налево могут отличаться довольно существенно. См. упражнения в конце интермеццо 4. ■

В качестве второго примера возьмем хорошо известную функцию вычисления факториала:

```
| ; N -> N
| ; вычисляет (* n (- n 1) (- n 2) ... 1)
| (check-expect (!.v1 3) 6)
| (define (!.v1 n)
|   (cond
|     [(zero? n) 1]
|     [else (* n (!.v1 (sub1 n))))]))
```

В отличие от `relative-2-absolute` и `invert`, обрабатывающих списки, функция вычисления факториала работает с натуральными числами, что отражается на ее макете.

Продолжим, как и раньше, начав с макета функции с аккумулятором:

```
| (define (!.v2 n0)
|   (local (; N ??? -> N
|           ; вычисляет (* n (- n 1) (- n 2) ... 1)
|           ; аккумулятор ...
|           (define (!/a n a)
|             (cond
|               [(zero? n) ...]
|               [else (... (!/a (sub1 n)
|                           ... a ...) ...)]))))
|   (!/a n0 ...)))
```

И набросаем примерный вид шагов вычисления:

$\begin{aligned} (!.v1 3) \\ == (* 3 (!.v1 2)) \\ == (* 3 (* 2 (!.v1 1))) \\ \dots \\ == 6 \end{aligned}$	$\begin{aligned} (!.v2 3) \\ == (!/a 3 a0) \\ == (!/a 2 \dots 3 \dots a0) \\ \dots \\ == 6 \end{aligned}$
---	---

В листинге слева показано, как работает исходная версия; справа – как работает функция с аккумулятором. Обе выполняют обход натуральных чисел в обратном порядке, пока не будет достигнут 0. Однако первоначальная версия лишь планирует умножение, а версия с аккумулятором фиксирует каждое число по мере перемещения до заданного натурального числа.

Учитывая, что целью является получение произведения этих чисел, `!/a` может использовать аккумулятор для немедленного их перемножения:

a – это произведение натуральных чисел в интервале $[n0, n]$.

В частности, когда $n0$ равно 3 и n равно 1, a будет равно 6.

Упражнение 496. Каким должно быть значение a , если $n0$ равно 3 и n равно 1? А если $n0$ равно 10 и n равно 8? ■

Этот инвариант диктует начальное значение для a – это 1, – и мы знаем, что правильной операцией обновления является умножение текущего аккумулятора на n :

```
(define (!.v2 n0)
  (local (; N N -> N
            ; вычисляет (* n (- n 1) (- n 2) ... 1)
            ; аккумулятор a – это произведение
            ; натуральных чисел в интервале [n0,n)
            (define (!/a n a)
              (cond
                [(zero? n) a]
                [else (!/a (sub1 n) (* n a))])))
  (!/a n0 1)))
```

Из определения аккумулятора также следует, что когда n равно 0, аккумулятор будет содержать произведение чисел от n_0 до 1, то есть желаемый результат. Итак, подобно `sum`, в этом случае функция `!/a` вернет аккумулятор a , в противном же случае возьмет результат из рекурсии.

Упражнение 497. Подобно `sum`, функция `!.v1` выполняет простые вычисления, в данном случае умножение, в обратном порядке. Как ни странно, но это отрицательно сказывается на производительности функции.

Измерьте, сколько времени необходимо на выполнение `(!.v1 20)` 1000 раз. Напомним, что для этого можно использовать функцию `(time an-expression)`, которая определяет время выполнения `an-expression`. ■

В качестве третьего и последнего примера мы используем функцию, которая измеряет высоту простого бинарного дерева. На этом примере мы покажем, что функции с аккумулятором можно использовать для обработки любых типов данных, а не только тех, что содержат одиночные ссылки на самих себя. На самом деле такие функции широко используются для обработки сложных данных, а не только списков и натуральных чисел.

Бот соответствующие определения:

```
(define-struct node [left right])
; Tree -- это одно из значений:
; -- '()
; -- (make-node Tree Tree)
(define example
  (make-node (make-node '() (make-node '() '()) ')')))
```

Эти деревья не несут никакой информации; их листы являются пустыми списками `'()`. Как показывает рис. 34, существует много разных деревьев; он также показывает, как могут выглядеть эти фрагменты данных.

Одно из свойств дерева, которое можно вычислить, – это его высота:

```
(define (height abt) ; A Binary Tree
  (cond
    [(empty? abt) 0]
    [else (+ (max (height (node-left abt))
                  (height (node-right abt))) 1)]))
```

Стоп! Добавьте сигнатуру и тест. Таблица на рис. 34 показывает, как измерить высоту дерева, но при этом не совсем четко определяет данное понятие: это либо количество узлов от корня дерева до самого высокого листа, либо количество ветвей на пути к этому листу. Функция `height` следует второму толкованию.

Чтобы преобразовать ее в функцию с аккумулятором, пойдем по стандартному пути. Начнем с определения макета:

```
(define (height.v2 abt0)
  (local (; Tree ??? -> N
          ; измеряет высоту бинарного дерева abt
          ; аккумулятор ...
          (define (height/a abt a)
            (cond
              [(empty? abt) ...]
              [else
                (... (height/a (node-left abt)
                               ... a ...))
                (... (height/a (node-right abt)
                               ... a ...))))])
    (height/a abt0 ...)))
```

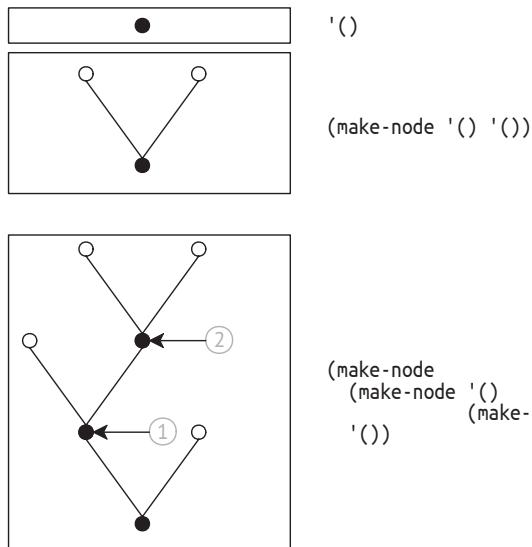


Рис. 34. Примеры бинарных деревьев

Как всегда, задача состоит в том, чтобы определить, какие знания будет представлять аккумулятор. Очевидный выбор – количество пройденных ветвей:

a – количество шагов, которые необходимо сделать, чтобы перейти в abt из $\text{abt}0$.

Этот инвариант аккумулятора проще проиллюстрировать на графическом примере. Взгляните еще раз на рис. 34. Самое нижнее дерево снабжено двумя аннотациями, каждая из которых указывает на одно поддерево.

- Если $\text{abt}0$ – это полное дерево, а abt – это поддерево, на которое указывает цифра 1 в кружке, то аккумулятор должен иметь значение 1, потому что для перехода от корня abt к корню $\text{abt}0$ требуется ровно один шаг.
- Аналогично для под дерева с меткой 2 аккумулятор должен иметь значение 2, потому что требуется два шага, чтобы добраться до этого места.

В двух предыдущих примерах инвариант описывал, как следовать остальной части рецепта проектирования функций с аккумулятором: начальное значение для a равно 0; операция обновления – add1 ; а базовый случай просто возвращает накопленные знания. Переведя это описание в код, получаем следующий макет определения:

```
(define (height.v2 abt0)
  (local (; Tree N -> N
          ; измеряет высоту бинарного дерева abt
          ; аккумулятор a – это количество шагов,
          ; необходимых для перехода в abt из abt0
          (define (height/a abt a)
            (cond
              [(empty? abt) a]
              [else
                (... (height/a (node-left abt)
                               (+ a 1)) ...
                     ... (height/a (node-right abt)
                                   (+ a 1)) ...)])))
        (height/a abt0 0)))
```

Но, в отличие от первых двух примеров, в базовом случае аккумулятор не является конечным результатом. Во втором предложении в выражении `cond` два рекурсивных вызова дают два значения. Рецепт проектирования структурных функций требует их объединения, чтобы сформулировать ответ; многоточия в макете подсказывают, что мы должны выбрать операцию объединения этих значений.

Листинг 120. Версия функции `height` с аккумулятором

```
; Tree -> N
; измеряет высоту бинарного дерева abt0
(check-expect (height.v2 example) 3)
(define (height.v2 abt0)
  (local (; Tree N -> N
          ; измеряет высоту бинарного дерева abt
          ; аккумулятор a – это количество шагов,
          ; необходимых для перехода в abt из abt0
```

```
(define (height/a abt a)
  (cond
    [(empty? abt) a]
    [else
      (max
        (height/a (node-left abt) (+ a 1))
        (height/a (node-right abt) (+ a 1))))]
    (height/a abt0 0)))
```

Согласно рецепту проектирования, мы должны правильно интерпретировать два значения, чтобы найти подходящую функцию. Согласно заявлению о назначении для `height/a`, первое значение – это высота левого поддерева, а второе – правого. Учитывая, что нас интересует высота дерева `abt` и что высота – это наибольшее количество шагов, необходимых для достижения самого высокого листа, мы используем функцию `max`, чтобы выбрать правильное значение. Полное определение приводится в листинге 120.

ЗАМЕЧАНИЕ ОБ АЛЬТЕРНАТИВНОМ ДИЗАЙНЕ. Кроме подсчета количества шагов, необходимых для достижения узла, функция с аккумулятором может также хранить наибольшую высоту, встреченную до сих пор. Вот описание аккумулятора для этой идеи:

Первый аккумулятор хранит количество шагов, необходимых для перехода в `abt` из (корня) `abt0`. Второй – высоту той ветви `abt0`, которая находится строго слева от `abt`.

Это описание предполагает макет с двумя аккумуляторами, с чем мы раньше не сталкивались:

```
... ; Tree N N -> N
; измеряет высоту бинарного дерева abt
; аккумулятор s – это количество шагов,
; необходимых для перехода в abt из abt0
; аккумулятор m – это максимальная высота
; той части abt0, которая находится строго слева от abt
(define (h/a abt s m)
  (cond
    [(empty? abt) ...]
    [else
      (... (h/a (node-left abt)
                 ... s ... ... m ...) ...
            ... (h/a (node-right abt)
                      ... s ... ... m ...)) ...]
```

Упражнение 498. Закончите определение `height.v3`. **Подсказка.** Самое нижнее дерево на рис. 34 не имеет поддерева слева от поддерева с цифрой 1. В нем имеется лишь один полный путь от корня к поддереву слева от поддерева с цифрой 2; этот путь состоит из двух шагов. ■

Эта вторая версия имеет более сложный инвариант аккумулятора. Его реализация требует большей осторожности, чем реализация инварианта в первой версии. В то же время у этой новой версии нет очевидных преимуществ.

Мы считаем, что разные инварианты аккумуляторов дают разные варианты реализации. Вы можете спроектировать оба варианта, используя систематический подход и следуя одному и тому же рецепту проектирования. А когда у вас будут законченные определения функций, вы сможете сравнить полученные результаты, а затем обоснованно решить, какое из них оставить. **КОНЕЦ.**

Упражнение 499. Спроектируйте версию функции `product` с аккумулятором, которая вычисляет произведение списка чисел. Когда вы сформулируете инвариант аккумулятора, остановитесь и попросите кого-нибудь его проверить.

Производительность `product` равна $O(n)$, где n – длина списка. Улучшает ли ее версия с аккумулятором? ■

Упражнение 500. Спроектируйте версию функции `how-many` с аккумулятором, которая определяет количество элементов в списке. Когда вы сформулируете инвариант аккумулятора, остановитесь и попросите кого-нибудь его проверить.

Производительность `how-many` равна $O(n)$, где n – длина списка. Улучшает ли ее версия с аккумулятором?

При вычислении (`how-many some-non-empty-list`) в пошаговом режиме можно заметить, что к тому времени, когда функция достигнет '()', обработки ожидают n применений функции `add1`, где n – количество элементов в списке. Специалисты по информатике иногда говорят, что функции `how-many` требуется пространство с размером $O(n)$ для хранения этих ожидающих выполнения применений функций. Помогает ли аккумулятор уменьшить пространство, необходимое для вычисления результата? ■

Специалисты по информатике называют это пространство пространством стека, но вы пока можете спокойно игнорировать данную терминологию.

Упражнение 501. Создайте версию функции `add-to-pi` с аккумулятором. Она прибавляет натуральное число к `pi` без использования `+`:

```
; N -> Number
; прибавляет n к pi без использования +
(check-within (add-to-pi 2) (+ 2 pi) 0.001)
(define (add-to-pi n)
  (cond
    [(zero? n) pi]
    [else (add1 (add-to-pi (sub1 n))))]))
```

Когда вы сформулируете инвариант аккумулятора, остановитесь и попросите кого-нибудь его проверить.

Упражнение 502. Спроектируйте функцию `palindrome`, которая принимает непустой список и создает палиндром, зеркально отражая список относительно последнего элемента. Для (`explode "abc"`) результат должен быть (`explode "abcba"`).

Подсказка. Вот решение, спроектированное с использованием приема композиции функций:

```
; [NList-of 1String] -> [NList-of 1String]
; создает палиндром из s0
```

```
(check-expect
  (mirror (explode "abc")) (explode "abcba"))
(define (mirror s0)
  (append (all-but-last s0)
          (list (last s0))
          (reverse (all-but-last s0))))
```

Вернитесь к разделу 11.4 и найдите определение функции `last`; спроектируйте аналогичную ей функцию `all-but-last`.

Это решение выполняет обход `s0` четыре раза:

- 1) один раз в `all-but-last`;
- 2) один раз в `last`;
- 3) еще один раз в `all-but-last`;
- 4) один раз в `reverse` – встроенной функции ISL+, действующей подобно `invert`.

Даже с использованием локальных определений для сохранения результата `all-but-last` ей требуется обойти список три раза. Хотя эти обходы не оказывают катастрофического влияния на производительность функции, версия с аккумулятором может вычислить тот же результат за один обход. ■

Упражнение 503. В упражнении 467 предлагается спроектировать функцию, которая перемещает строки в матрице `Matrix` до тех пор, пока первый коэффициент в первой строке не станет отличным от 0. Решение в упражнении 467 использует генеративно-рекурсивную функцию, которая создает новую матрицу, сдвигая первую строку в конец, встретив 0 в первой позиции. Вот решение этого упражнения:

```
; Matrix -> Matrix
; ищет строку в матрице, первый коэффициент в которой
; не 0, и перемещает ее на первое место
; генеративно перемещает первую строку в конец матрицы
; не завершается, если все строки начинаются с 0
(check-expect (rotate '((0 4 5) (1 2 3)))
  '((1 2 3) (0 4 5)))

(define (rotate M)
  (cond
    [(not (= (first (first M)) 0)) M]
    [else
      (rotate (append (rest M) (list (first M))))]))
```

Стоп! Измените эту функцию так, чтобы она сообщала об ошибке, если в матрице все строки начинаются с 0.

Измерив время выполнения этой функции с большими матрицами `Matrix`, вы получите удивительный результат:

строк в матрице M	1000	2000	3000	4000	5000
rotate	17	66	151	272	436

С увеличением количества строк с 1000 до 5000 время выполнения `rotate` увеличивается не в пять, а в двадцать раз.

Проблема в том, что `rotate` использует `append`, которая создает новый список, такой как `(rest M)`, только чтобы добавить в конец `(first M)`. Если `M` содержит 1000 строк, а последняя – единственная строка с ненулевым коэффициентом, то будет создано примерно

$$1000 \cdot 1000 = 1\,000\,000$$

списков. Сколько списков получится, если `M` содержит 5000 строк?

Теперь предположим, что версия с аккумулятором быстрее генеративной. Вот макет структурно-рекурсивной версии `rotate` с аккумулятором:

```
(define (rotate.v2 M0)
  (local (; Matrix ... -> Matrix
         ; аккумулятор ...)
    (define (rotate/a M seen)
      (cond
        [(empty? M) ...]
        [else (... (rotate/a (rest M)
                               ... seen ...)
                    ...))])
    (rotate/a M0 ...)))
```

Цель состоит в том, чтобы запомнить первую строку, если первый коэффициент в ней равен 0, без использования `append` в каждом рекурсивном вызове.

Сформулируйте описание аккумулятора. Затем закончите функцию выше, следуя рецепту проектирования функций с аккумулятором. Измерьте скорость ее выполнения с матрицей, все строки в которой, кроме последней, имеют нулевые первые коэффициенты. Если вы все сделаете правильно, то функция будет работать довольно быстро. ■

Упражнение 504. Спроектируйте функцию `to10`. Она должна принимать список цифр и выдавать соответствующее число. Первый элемент в списке – это **самая значимая цифра**. То есть при применении к '(1 0 2) функция должна дать 102.

Знание предметной области. Возможно, вы помните из школьного курса математики, что результат определяется как $1 \cdot 10^2 + 0 \cdot 10^1 + 2 \cdot 10^0 = ((1 \cdot 10 + 0) \cdot 10) + 2 = 102$. ■

Упражнение 505. Спроектируйте функцию `is-prime`, которая принимает натуральное число и возвращает `#true`, если оно простое, и `#false` в противном случае.

Знание предметной области. Число n является простым, если оно не делится ни на какое число от $n - 1$ до 2.

Подсказка. Рецепт проектирования для N [≥ 1] предлагает следующий макет:

```
; N [>= 1] -> Boolean
; определяет, является ли n простым числом
(define (is-prime? n)
  (cond
```

```
|  [(= n 1) ...]
|  [else (... (is-prime? (sub1 n)) ...))])
```

Одного взгляда на этот макет достаточно, чтобы заметить, что функция забывает n , свой начальный аргумент, при рекурсивном применении. Поскольку значение n необходимо, чтобы определить, делится ли n на $(-n1)$, $(-n2)$ и т. д., становится понятно, что в данном случае предпочтительнее определить функцию с аккумулятором. ■

ЗАМЕЧАНИЕ О СКОРОСТИ.

Программисты, впервые сталкивающиеся с функциями с аккумулятором, часто думают, что они всегда быстрее своих простых аналогов. Что ж, давайте посмотрим на решение упражнения 497:

!.v1	5.760	5.780	5.800	5.820	5.870	5.806
!.v2	5.970	5.940	5.980	5.970	6.690	6.111

В верхней строке таблицы приводится время в секундах вычисления выражения $(!.v1\ 20)$ в пяти попытках, а в нижней – выражения $(!.v2\ 20)$. В последнем столбце приводятся средние значения. Как видите, люди иногда делают спешные выводы; производительность как минимум одной функции с аккумулятором хуже производительности простой версии. *Не верьте предрассудкам.* Измеряйте производительность ваших программ. **КОНЕЦ.**

Упражнение 506. Создайте версию `map` с аккумулятором. ■

Упражнение 507. В упражнении 257 объясняется, как спроектировать `foldl`, используя рецепты проектирования и рекомендации из первых двух частей книги:

```
| (check-expect (f*ldl + 0 '(1 2 3))
|   (foldl + 0 '(1 2 3)))
| (check-expect (f*xldl cons '() '(a b c))
|   (foldl cons '() '(a b c)))
;
; версия 1
(define (f*xldl f e l)
  (foldr f e (reverse l)))
```

То есть `foldl` переворачивает заданный список и затем применяет `foldr` для свертки этого промежуточного списка с применением заданной функции.

Очевидно, что функция `f*xldl` обходит список дважды, но, спроектировав все функции, можно заметить, насколько все сложнее на самом деле:

```
| ; версия 2
| (define (f*xldl f e l)
|   (local ((reverse l)
|           (cond
|             [(empty? l) '()]
|             [else (add-to-end (first l)
|                               (reverse (rest l)))])))
|     (define (add-to-end x l)
|       (cond
```

```

[empty? l) (list x)]
[else (cons (first l)
             (add-to-end x (rest l)))]))
(define (foldr l)
  (cond
    [empty? l] e]
    [else (f (first l) (foldr (rest l))))])
  (foldr (reverse l)))

```

Как видите, `reverse` должна выполнить обход списка один раз для каждого элемента в списке, а это означает, что `f*ldl` в действительности выполняет n^2 обходов списка длиной n . К счастью, мы знаем, как устранить это узкое место с помощью аккумулятора:

```

; версия 3
(define (f*ldl f e l)
  (local ((define (invert/a l a)
            (cond
              [empty? l] a]
              [else (invert/a (rest l)
                               (cons (first l) a))])))
    (define (foldr l)
      (cond
        [empty? l] e]
        [else
          (f (first l) (foldr (rest l))))]))
    (foldr (invert/a l '())))))

```

Добавив аккумулятор в `reverse`, мы уменьшили количество обходов списка до двух. Теперь попробуем ответить на другой вопрос: можно ли улучшить эту версию, добавив аккумулятор в локальное определение `fold`:

```

; версия 4
(define (f*ldl f e l0)
  (local ((define (fold/a a l)
            (cond
              [empty? l] a]
              [else
                (fold/a (f (first l) a) (rest l))])))
    (fold/a e l0)))

```

Поскольку добавление аккумулятора в функцию меняет порядок обхода списка на противоположный, отпадает необходимость в предварительном переворачивании списка.

Задача 1. Вспомните сигнатуру `foldl`:

| ; [X Y] [X Y -> Y] Y [List-of X] -> Y

Она совпадает с сигнатурой `f*ldl`. Сформулируйте сигнатуру для `fold/a` и ее инварианта аккумулятора. **Подсказка.** Предположите, что разница между `l0` и `l` равна (`list x1 x2 x3`). Что же тогда хранит аккумулятор `a`?

Вам также может быть интересно, почему `fold/a` принимает аргументы в таком необычном порядке: сначала аккумулятор, а затем

список. Чтобы понять причину, представьте, что в первом аргументе `fold/a` принимает `f`. После этого вы увидите, что `fold/a` – это `foldl`:

```
; версия 5
(define (f*ldl f i l)
  (cond
    [(empty? l) i]
    [else (f*ldl f (f (first l) i) (rest l))]))
```

Задача 2. Спроектируйте `build-l*st` с аккумулятором. Функция должна соответствовать следующим тестам:

```
| (check-expect (build-l*st n f) (build-list n f))
```

для любого натурального числа `n` и функции `f`. ■

32.4. Графический редактор с поддержкой мыши

В разделе 5.10 вы познакомились с однострочным редактором и решили ряд упражнений по созданию графического редактора. Напомним, что графический редактор – это интерактивная программа, которая интерпретирует события от клавиатуры как действия редактирования строки. В частности, когда пользователь нажимает клавиши со стрелками влево или вправо, курсор перемещается влево или вправо; аналогично нажатие клавиши удаления удаляет символ (`1String`) из редактируемого текста. Программа-редактор использует представление данных, объединяющее две строки в структуру. В разделе 10.4 вы продолжили решать аналогичные упражнения и узнали, как одна и та же программа может извлечь дополнительную выгоду из другой структуры данных, объединяющей две строки.

Ни в одном из этих разделов не рассматривались события от мыши и не использовались для навигации, хотя все современные приложения поддерживают эту функцию. Основная трудность с событиями от мыши состоит в том, чтобы поместить текстовый курсор в нужное место. Поскольку программа работает с одной строкой текста, щелчок мыши в позиции (x, y) можно смело интерпретировать как попытку поместить текстовый курсор между буквами, которые находятся в позиции x или окружают ее. В данном разделе мы восполним этот пробел.

Давайте вспомним соответствующие определения из раздела 10.4:

```
(define FONT-SIZE 11)
(define FONT-COLOR "black")

; [List-of 1String] -> Image
; преобразует список символов в текстовое изображение
(define (editor-text s)
  (text (implode s) FONT-SIZE FONT-COLOR))

(define-struct editor [pre post])
```

```
| ; Editor -- это структура:
| ; (make-editor [List-of 1String] [List-of 1String])
| ; интерпретация: (make-editor p s) - это состояние
| ; интерактивного редактора, (reverse p) соответствует
| ; тексту слева от курсора, а s - тексту справа
```

Упражнение 508. Спроектируйте функцию `split-structural`, используя рецепт структурного проектирования. Функция должна принимать список букв (`1String`) `ed` и натуральное число `x`; список представляет полную строку в некотором экземпляре `Editor`, а `x` – координату `X` щелчка мыши. Результатом функции должно быть

```
| (make-editor p s)
```

где (1) `p` и `s` составляют `ed` и (2) `x` больше, чем изображение `p`, и меньше, чем изображение `p`, дополненное первой буквой из `s` (если она есть).

Вот как выглядит первое условие, выраженное на языке `ISL+`:

```
| (string=? (string-append p s) ed)
```

Второе условие:

```
| (<= (image-width (editor-text p))
|   x
|   (image-width (editor-text (append p (first s))))))
```

предполагающее (`cons?` `s`).

Подсказки. (1) Координата `x` измеряет расстояние от левого края. Следовательно, функция должна проверить, подходят ли все большие и большие префиксы `ed` к заданной ширине. Первый, который не подходит, соответствует полю `pre` заданного экземпляра `Editor`, а все остальное – полю `post`.

(2) Чтобы спроектировать эту функцию, нужно со всем тщанием подойти к разработке примеров и тестов. См. главу 4. ■

Упражнение 509. Спроектируйте функцию `split`. Используйте рецепт проектирования функций с аккумулятором, чтобы улучшить результат упражнения 508. В конце концов, подсказки уже указывают на то, что после обнаружения правильной точки разделения функции потребуются обе части списка, тогда как в ходе рекурсии одна часть безусловно будет потеряна. ■

Выполнив это упражнение, добавьте в функцию `main` из раздела 10.4 предложение для обработки щелчков мышью. Попробовав перемещать текстовый курсор с помощью мыши, можно заметить, что наш редактор ведет себя не совсем так, как другие приложения, даже притом что `split` успешно проходит все тесты.

При разработке графических программ, подобных редакторам, приходится много экспериментировать, чтобы добиться наилучшего внешнего вида. В этом отношении наш редактор действует слишком упрощенно. Другие приложения не только определяют точку разделения, но также выясняют, какая граница между буквами ближе к координате `X` щелчка, и помещают текстовый курсор туда.

Упражнение 510. Многие операционные системы включают программу `fmt`, которая может переставлять слова в файле так, чтобы все строки в получившемся файле имели максимальную длину. Программа `fmt` поддерживает множество связанных функций. В этом упражнении вы должны сосредоточиться только на самых основных ее функциях.

Спроектируйте программу `fmt`. Она должна принимать натуральное число `w`, имя входного файла `in-f` и имя выходного файла `out-f` – в том виде, в каком их принимает `read-file` из библиотеки `2htdp/batch-io`. Ее назначение – прочитать все слова из `in-f`, разместить эти слова в том же порядке в строках с максимальной шириной `w` и записать эти строки в `out-f`. ■

33. Дополнительные примеры использования аккумуляторов

В этой главе представлены еще три примера использования аккумуляторов. В первом разделе демонстрируется применение аккумуляторов в сочетании с функциями обработки деревьев. В нем в качестве наглядного примера используется компиляция ISL+. Во втором разделе объясняется, почему аккумуляторы иногда нужны внутри представлений данных и как их туда поместить. В последнем разделе мы продолжим обсуждать вопросы отображения фракталов.

33.1. Аккумуляторы и деревья

Когда вы даете среде программирования DrRacket команду запустить программу на ISL+, она преобразует указанную программу в команды для вашего конкретного компьютера. Этот процесс преобразования называется *компиляцией*, а компонент DrRacket, выполняющий ее, называется *компилятором*. Перед трансляцией программы компилятор проверяет, объявлены ли все переменные, используемые в определениях функций, структур и лямбда-выражений.

Стоп! Введите в DrRacket x , $(\lambda(y)x)$ и (x^5) как отдельные программы на ISL+ и попробуйте запустить каждую из них. Что вы ожидаете увидеть?

Сформулируем эту идею как задачу:

Постановка задачи. Вас наняли для воссоздания части компилятора ISL+. В частности, ваша задача касается следующего фрагмента языка, записанного в так называемой грамматической нотации, которая используется во многих руководствах по языкам программирования:

выражение = переменная
| (λ (переменная) выражение)
| (выражение выражение)

Мы используем греческую букву λ вместо *lambda*, чтобы показать, что в этом упражнении ISL+ рассматривается как объект изучения, а не только как язык программирования.

Как отмечалось в интермеццо 1, грамматическую нотацию можно читать вслух, заменяя $=$ на «является одним из», а $|$ – на «или».

Напомним, что λ -выражения – это безымянные функции. Они связывают свои параметры с переменными в своем теле. И в обратную сторону: переменная объявляется ближайшей λ , имеющей параметр с тем же именем. Возможно, вы решите вернуться к интермеццо 3, где обсуждается та же проблема, но

с точки зрения программиста. Ищите термины «связывающее вхождение», «связанное вхождение» и «свободное вхождение».

Разработайте представление данных для указанного выше фрагмента языка; используйте символы для обозначения переменных. Затем спроектируйте функцию, которая заменяет все необъявленные переменные на `'*undeclared`.

Эта задача характерна для многих этапов процесса трансляции и в то же время является отличным примером для функций с аккумуляторами.

Прежде чем углубиться в решение задачи, рассмотрим несколько примеров на этом мини-языке и вспомним, что мы знаем о лямбда-выражениях:

- $(\lambda(x)x)$ – это функция, которая просто возвращает переданное ей значение; она также известна как функция идентичности;
- $(\lambda(x)y)$ выглядит как функция, которая возвращает y всякий раз, когда получает аргумент, за исключением того, что y не объявлен;
- $(\lambda(y)(\lambda(x)y))$ – функция, которая, получив некоторое значение v , создает функцию, которая всегда возвращает v ;
- $(\lambda((x)x)(\lambda(x)x))$ применяет функцию идентичности к самой себе;
- $((\lambda(x)(x x))(\lambda(x)(x x)))$ – короткий бесконечный цикл;
- $((((\lambda(y)(\lambda(x)y))(\lambda(z)z))(\lambda(w)w)))$ – сложное выражение, которое следует запустить в ISL+, чтобы узнать, завершается ли оно.

Вы можете попробовать выполнить все вышеперечисленные выражения на ISL+ в DrRacket, чтобы убедиться в верности их описания.

Упражнение 511. Объясните область видимости каждого связывающего вхождения в примерах выше. Нарисуйте стрелки от связывающих вхождений к связанным экземплярам. ■

Разработать представление данных для языка несложно, особенно если в его описании используется грамматическая нотация. Вот одна из возможностей:

; ; Lam – это одно из значений:	
; - символ (Symbol)	
; - (list 'λ (list Symbol) Lam)	
; - (list Lam Lam)	

Благодаря наличию quote это представление данных упрощает создание представлений данных для выражений в нашем подмножестве ISL+:

(define ex1 '(λ (x) x))	
(define ex2 '(λ (x) y))	
(define ex3 '(\lambda (y) (\lambda (x) y)))	
(define ex4 '(((λ (x) (x x)) (\lambda (x) (x x))))	

Эти четыре примера данных представляют некоторые из выражений, приводившихся выше. Стоп! Создайте представления данных для остальных примеров.

Упражнение 512. Определите `is-var?`, `is-λ?` и `is-app?` – предикаты, которые отличают переменные от выражений и применений.

Также определите селекторы:

- `λ-args` – извлекает параметр из λ-выражения;
- `λ-body` – извлекает тело из λ-выражения;
- `app-fun` – извлекает функцию из применения;
- `app-arg` – извлекает аргумент из применения.

С помощью этих предикатов и селекторов вы сможете действовать так, как если бы определили структурно-ориентированное представление данных.

Спроектируйте функцию `undeclareds`, которая создает список всех символов, используемых в качестве параметров в λ-выражении. Не беспокойтесь о возможных повторениях. ■

Упражнение 513. Разработайте представление данных для того же подмножества ISL+, используя структуры вместо списков. Также создайте представления данных для `ex1`, `ex2` и `ex3` в соответствии с вашим определением данных. ■

Если следовать рецепту структурного проектирования, то после выполнения второго и третьего шагов получится вот такой результат:

```
; Lam -> Lam
; заменяет все символы s в le на '*undeclared,
; если они не находятся в теле λ-выражения
; с параметром s

(check-expect (undeclareds ex1) ex1)
(check-expect (undeclareds ex2) '((x) *undeclared))
(check-expect (undeclareds ex3) ex3)
(check-expect (undeclareds ex4) ex4)

(define (undeclareds le0)
  le0)
```

Обратите внимание: мы ожидаем, что `undeclareds` обработает `ex4`, несмотря на то что это выражение зацикливается при запуске; компиляторы не запускают программы, они их читают и создают другие.

Внимательный взгляд на заявление о назначении прямо указывает, что функции нужен аккумулятор. Это станет еще очевиднее, если исследовать макет для `undeclareds`:

```
(define (undeclareds le)
  (cond
    [(is-var? le) ...]
    [(is-λ? le) (... (undeclareds (λ-body le)) ...)]
    [(is-app? le)
      (... (undeclareds (app-fun le))
           ... (undeclareds (app-arg le)) ...))])
```

Когда `undeclareds` выполняет рекурсию в теле (представлении) λ -выражения, оно забывает (λ -рага `le`), объявленную переменную.

Итак, начнем с создания макета с аккумулятором:

```
(define (undeclareds le0)
  (local
    ( ; Lam ??? -> Lam
      ; аккумулятор представляет ...
      (define (undeclareds/a le a)
        (cond
          [(is-var? le) ...]
          [(is-λ? le)
            (... (undeclareds/a (λ-body le)
              ... a ...) ...)]
          [(is-app? le)
            (... (undeclareds/a (app-fun le)
              ... a ...)
              ... (undeclareds/a (app-arg le)
                ... a ...) ...))])
        (undeclareds/a le0 ...))))
```

Теперь мы можем сформулировать инвариант аккумулятора:

`a` представляет список параметров, встречающихся на пути от вершины `le0` к вершине `le`.

Например, если `le0` равно

```
| '(((λ (y) (λ (x) y)) (λ (z) z)) (λ (w) w))
```

и `le` – выделенное поддерево, тогда `a` содержит `y`. На рис. 35 слева показана графическая иллюстрация этого примера. Здесь выражение `Lam` представлено в виде перевернутого дерева, корень которого находится вверху. @-узлы представляют применение с двумя потомками; подписи других узлов говорят сами за себя. Выделенный путь на этой диаграмме ведет от `le0` к `le` через объявление одной переменной.

Точно так же, если выбрать другое поддерево того же фрагмента данных:

```
| '(((λ (y) (λ (x) y)) (λ (z) z)) (λ (w) w))
```

мы получим аккумулятор, содержащий как '`y`, так и '`x`. Дерево справа на рис. 35 подчеркивает это. Здесь выделенный путь ведет через два узла ' λ к поддереву в рамке, а аккумулятор – это список переменных, объявленных на этом пути.

Теперь, когда мы определились с представлением данных аккумулятора и его инварианта, можно решить оставшиеся вопросы проектирования:

- выбрать для аккумулятора начальное значение '();
- использовать `cons` для добавления (λ -рага `le`);
- использовать аккумулятор в ветви, где `undeclareds/a` имеет дело с переменной. А именно функция может использовать аккуму-

лятор для проверки – находится ли переменная в области видимости объявления.

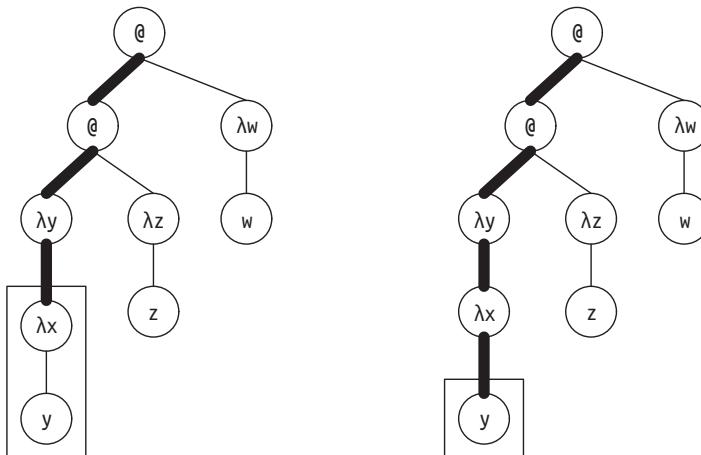


Рис. 35. Выражения Lam в виде деревьев

В листинге 121 показано, как воплотить эти идеи в законченное определение функции. Обратите внимание на имя аккумулятора – `declareds`; оно раскрывает ключевую идею инварианта аккумулятора и помогает программисту понять определение. В базовом случае используется `member?` – функция из ISL+, – чтобы определить, находится ли переменная `le` в `declareds`, и если нет, то заменить ее на `*undeclared`. Во втором предложении в `cond` используется объявление `local` для добавления расширенного аккумулятора `newd`. Поскольку `para` также применяется для перестроения выражения, в ней есть собственное локальное определение. Наконец, последняя ветвь касается применения функций, которые не объявляют переменных и не используют их напрямую. В результате эта ветвь получилась самой простой.

Листинг 121. Поиск необъявленных переменных

```

; Lam -> Lam
(define (undeclareds le0)
  (local ( ; Lam [List-of Symbol] -> Lam
          ; аккумулятор declareds – это список всех λ
          ; параметров на пути из le0 в le
          (define (undeclareds/a le declareds)
            (cond
              [(is-var? le)
               (if (member? le declareds) le '*undeclared)]
              [(is-lambda? le)
               (local ((define para (lambda-para le))
                      (define body (lambda-body le))
                      (define newd (cons para declareds)))
                  (list 'λ (list para
                                  (undeclareds/a body newd))))]
              [else le])))))

```

```
[((is-app? le)
  (local ((define fun (app-fun le))
          (define arg (app-arg le)))
         (list (undeclareds/a fun declareds)
               (undeclareds/a arg declareds))))]
(undeclareds/a le0 '())))
```

Упражнение 514. Составьте выражение на ISL+, в котором x встречается как в свободном, так и в связанном виде. Сформулируйте это как элемент Lam. Правильно ли обрабатывает `undeclareds` ваше выражение? ■

Упражнение 515. Взгляните на следующее выражение:

```
| (λ (*undeclared) ((λ (x) (x *undeclared)) y))
```

Да, оно использует `*undeclared` как переменную. Представьте его в Lam и проверьте, что `undeclareds` возвращает для этого выражения.

Измените `undeclareds` так, чтобы она заменяла свободное вхождение ' x ' на

```
| (list '*undeclared 'x)
```

а связанное вхождение ' y ' на

```
| (list '*declared 'y)
```

Такой прием помогает однозначно определить проблемные места и может использоваться средой разработки программ, такой как DrRacket, для выявления ошибок.

Примечание. Уловка с заменой вхождения переменной представлением применения кажется неудобной. Если она вам не нравится, то можете вместо этого синтезировать символы `'*undeclared:x` и `'declared:y`. ■

Упражнение 516. Повторно спроектируйте функцию `undeclareds` для представления данных на основе структуры из упражнения 513. ■

Упражнение 517. Спроектируйте функцию `static-distance`. Она должна заменять все вхождения переменных натуральным числом, показывающим, насколько далеко находится вхождение переменной от объявляющей λ . Рисунок 36 иллюстрирует идею этого понятия в графическом виде на примере следующего выражения:

```
| '((λ (x) ((λ (y) (y x)) x)) (λ (z) z))
```

Пунктирные стрелки ведут от вхождений переменных к соответствующим объявлениям. Справа на рисунке показано дерево такой же формы, но без стрелок. Узлы ' λ ' не содержат имен, а вхождения переменных заменены натуральными числами, которые указывают, какой узел ' λ ' объявляет эту переменную. Каждое натуральное число n означает, что привязка происходит на n шагов выше к корню дерева Lam. Значение 0 обозначает первый узел ' λ «на пути к корню», 1 – второй и т. д.

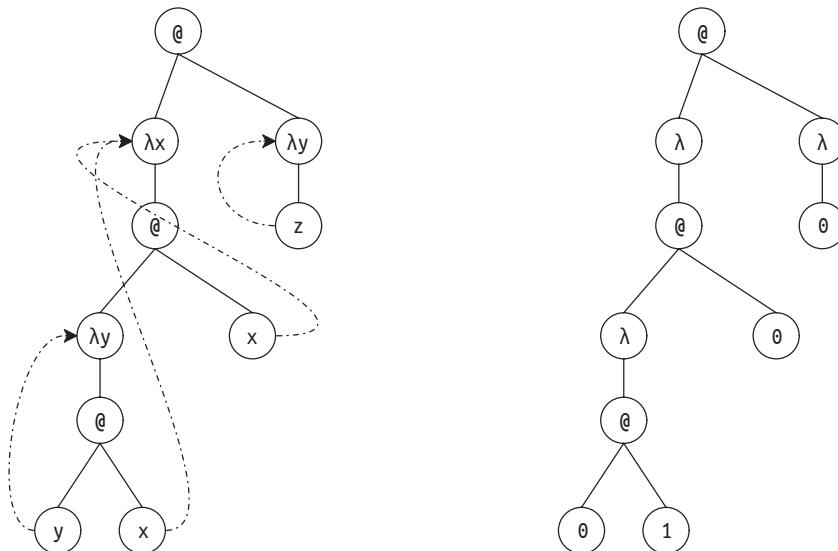


Рис. 36. Статические расстояния

Подсказка. Аккумулятор `undeclareds` в `undeclareds/a` – это **перевернутый** список всех параметров на пути от `le` к `le0` – последний встреченный находится в начале списка. ■

33.2. Представления данных с аккумуляторами

В конце интермессо 5 говорилось, что в *SL размеры контейнеров, например списков, измеряются путем их обхода, и особо отмечалось, что в некоторых других языках используется иной, менее затратный способ вычисления размеров. В данном разделе мы покажем, как реализовать эту идею путем **добавления аккумулятора в представление данных**.

См. предыдущий пример реализации этой идеи в разделе 12.8.

Обратимся к вездесущим спискам. Все списки в *SL конструируются с помощью `cons` и `'()`; такие операции, как `quote` и `list`, например, являются просто более краткими формами этих двух. Как было показано в разделе 8.2, в BSL можно имитировать списки, используя подходящие типы структур и определения функций.

Листинг 122 напоминает основную идею. Стоп! Сможете ли вы теперь определить `our-test?`

Листинг 122. Реализация списков в BSL

```
(define-struct pair [left right])
; ConsOrEmpty – это одно из значений:
; - '()
; - (make-pair Any ConsOrEmpty)
```

```

; Any ConsOrEmpty -> ConsOrEmpty
(define (our-cons a-value a-list)
  (cond
    [(empty? a-list) (make-pair a-value a-list)]
    [(our-cons? a-list) (make-pair a-value a-list)]
    [else (error "our-cons: ...")])))

; ConsOrEmpty -> Any
; извлекает левую часть из заданной пары
(define (our-first mimicked-list)
  (if (empty? mimicked-list)
      (error "our-first: ...")
      (pair-left mimicked-list)))

```

Ключевым моментом является возможность добавить третье поле в определение структуры `sraig`:

```

(define-struct cpair [count left right])
; [MyList X] - это одно из значений:
; - '()
; - (make-cpair (tech "N") X [MyList X])
; аккумулятор в поле count - это количество структур cpair

```

Как сказано в описании аккумулятора, дополнительное поле используется для хранения количества экземпляров `sraig`, участвовавших в создании списка. То есть аккумулятор запоминает факт конструирования списка. Такие поля в структурах мы называем *аккумуляторами данных*.

Добавить поле в основной конструктор списка не так-то просто. Для начала нужно изменить надежную версию конструктора, которая фактически используется программами:

```

; определения данных через функцию-конструктор
(define (our-cons f r)
  (cond
    [(empty? r) (make-cpair 1 f r)]
    [(cpair? r) (make-cpair (+ (cpair-count r) 1) f r)]
    [else (error "our-cons: ...")]))

```

Если расширяется пустой список '(), то в `count` записывается 1; в противном случае функция вычисляет длину, опираясь на заданный экземпляр `sraig`. Теперь легко можно определить функцию `our-length`:

```

; Any -> N
; определяет, сколько элементов содержит l
(define (our-length l)
  (cond
    [(empty? l) 0]
    [(cpair? l) (cpair-count l)]
    [else (error "my-length: ...")]))

```

Функция принимает любое значение. Для '() и экземпляров `sraig` она возвращает натуральное число; иначе сообщает об ошибке.

Вторая проблема, связанная с добавлением поля `count`, касается производительности. На самом деле есть две проблемы. С одной стороны, каждый элемент списка теперь имеет дополнительное поле, а это означает увеличение потребления памяти на 33 %. С другой стороны, снижается скорость конструирования списка функцией `oig-cons`. В дополнение к проверке, является ли расширенный список пустым списком '()' или экземпляром `sra1g`, конструктор вычисляет размер списка. Это вычисление занимает постоянное время, но оно добавляется в каждое применение `oig-cons` – только подумайте, сколько раз в этой книге использовалась `cons` и никогда не вычислялась длина получающегося списка!

Упражнение 518. Докажите, что `oig-cons` тратит постоянное время для вычисления своего результата, независимо от размера входного списка. ■

Упражнение 519. Допустимо ли накладывать дополнительные расходы на `cons` для всех программ, чтобы превратить `length` в функцию с постоянным временем выполнения? ■

Добавление поля `count` в списки вызывает некоторые сомнения, но иногда аккумуляторы для данных играют важную роль в поиске решения. Следующий пример иллюстрирует добавление так называемого *искусственного интеллекта* в игровую программу, и аккумулятор для данных в ней оказывается совершенно необходимым.

Играя в настольные игры или решая головоломки, мы склонны обдумывать возможные ходы на каждом шаге. С опытом вы можете даже научиться видеть возможности уже после первого шага. В результате получается так называемое *дерево партии*, представляющее (часть) дерева всех возможных ходов, разрешенных правилами. Начнем с постановки задачи:

Постановка задачи. Ваш руководитель рассказывает вам следующую историю.

«Давным-давно три каннибала вели трех миссионеров через джунгли. Они направлялись к ближайшей миссионерской станции. Через некоторое время они подошли к широкой реке, наполненной опасными змеями и рыбами. Переправиться через реку можно только на лодке. К счастью, после непродолжительных поисков они нашли лодку с двумя веслами. К сожалению, лодка оказалась слишком маленькой, чтобы перевезти их всех. В ней едва умелись два человека. Хуже того, так как река очень широкая, кому-то приходилось перегонять лодку обратно.

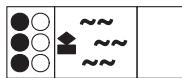
Поскольку миссионеры не могли доверять каннибалам, им пришлось разработать план, как безопасно переправить всех шестерых через реку. Проблема заключалась в том, что каннибалы убивали и съедали миссионеров, как только в одном месте каннибалов оказывалось больше, чем миссионеров. Миссионеры должны были разработать план, гарантирующий, что по обе стороны реки миссионеры никогда не останутся в мень-

шинстве. Однако во всем остальном каннибалам вполне можно было доверить. Они не отказались бы ни от какой потенциальной пищи, но точно так же и миссионеры не отказались бы от потенциальных новообращенных».

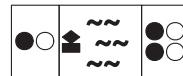
Ваш руководитель не ставит перед вами никаких конкретных задач, а просто хочет выяснить, сможет ли компания разрабатывать (и продавать) программы, которые решают подобные головоломки.

Конечно, головоломка – это не настольная игра, но программа может проиллюстрировать идею деревьев партий самым простым способом.

В принципе, такие головоломки довольно легко решаются вручную. Прежде всего нужно выбрать хорошее графическое представление состояний задачи. В нашем случае поле головоломки делится на три части: слева находятся миссионеры и каннибала; в середине – река и лодка, а справа – противоположный берег реки. Взгляните на следующее представление начального состояния:



Черными кружочками обозначены миссионеры, а белыми – каннибали. Все они находятся на левом берегу реки. Лодка тоже находится у левого берега. На правом берегу никого нет. Вот еще два состояния:



Первое – это конечное состояние, когда все люди и лодка находятся на правом берегу реки. Второе – это какое-то промежуточное состояние, когда два человека с лодкой находятся на левом берегу и четыре человека – на правом.

Теперь, когда у нас есть способ записи состояния головоломки, можно подумать о возможных действиях на каждом шаге. Это даст нам дерево возможных ходов. На рис. 37 показаны первые два с половиной уровня такого дерева. Крайнее левое состояние – начальное. Поскольку лодка может перевозить не более двух человек и на веслах должен сидеть хотя бы один, у нас есть пять возможных вариантов: один каннибал пересек реку; два каннибала пересекли реку; один миссионер и один каннибал; один миссионер; или два миссионера. Эти варианты показаны пятью стрелками, идущими от начального состояния к пяти промежуточным состояниям.

Начав с каждого из этих пяти промежуточных состояний, можно снова сыграть в ту же игру. На рис. 37 показано продолжение игры со среднего (третьего) состояния из пяти возможных. Поскольку на

правом берегу реки всего два человека, мы видим три варианта дальнейшего развития: каннибал возвращается, миссионер возвращается или оба возвращаются. Соответственно, три стрелки соединяют среднее состояние с тремя состояниями справа. Если продолжить систематически рисовать дерево возможных вариантов, вы в конечном итоге обнаружите конечное состояние.

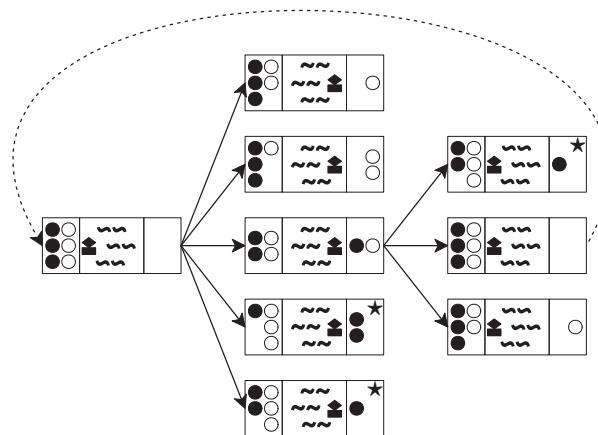


Рис. 37. Создание игрового дерева

Второй взгляд на рис. 37 обнаруживает две проблемы этого упрощенного подхода к построению дерева. Первый – пунктирная стрелка, соединяющая среднее состояние справа с начальным состоянием. Это означает, что если двое вернутся с правого берега на левый, то головоломка вернется в исходное состояние, и нам придется начать все сначала, что, конечно же, нежелательно. Вторая проблема касается состояний, отмеченных звездочкой в правом верхнем углу. В обоих случаях на левом берегу остается больше каннибалов, чем миссионеров, а это значит, что каннибалы съедят миссионеров. Наша цель состоит в том, чтобы избежать таких состояний, сделав подобные ходы нежелательными.

Один из способов превратить эту головоломку в программу – спроектировать функцию, которая определяет, достижимо ли какое-то конечное состояние (в данном случае **конкретное** конечное состояние) из некоторого заданного состояния. Вот подходящее определение такой функции:

```

; PuzzleState -> PuzzleState
; достижимо ли конечное состояние из состояния state0
; генеративно создает дерево возможных перемещений лодки
; завершение ???

(define (solve state0)

  (check-expect (solve initial-puzzle) final-puzzle)
  
```

```
(local ( ; [List-of PuzzleState] -> PuzzleState
      ; генеративно порождает последующие состояния los
      (define (solve* los)
        (cond
          [(empty? final? los)
           (first (filter final? los))]
          [else
           (solve* (create-next-states los))]))
      (solve* (list state0))))
```

Вспомогательная функция использует генеративную рекурсию и генерирует новые варианты с учетом списка имеющихся вариантов. Если один из заданных вариантов является конечным состоянием, функция возвращает его.

Очевидно, что `solve` – довольно универсальная функция. Если у вас есть определение коллекции *PuzzleState*, функция распознавания конечного состояния и функция создания всех «последующих» состояний, то `solve` сможет решить вашу головоломку.

Упражнение 520. Функция `solve*` генерирует все состояния, достижимые за n поездок на лодке, прежде чем переходить к состояниям, требующим $n + 1$ поездок на лодке, даже если некоторые из этих поездок возвращаются в ранее встречавшиеся состояния. Благодаря такому систематическому способу обхода дерева функция `resolve*` не может зациклиться. Почему? **Терминология.** Этот способ поиска в дереве или графе называется *поиском в ширину*. ■

Упражнение 521. Разработайте представление для состояний головоломки о миссионерах и каннибалах. Так же, как в графическом представлении, в представлении данных должно быть указано количество миссионеров и каннибалов на каждой стороне реки, а еще местоположение лодки.

Описание *PuzzleState* требует нового типа структуры. Представьте с помощью своего представления начальное, промежуточное и конечное состояния, описанные выше.

Спроектируйте функцию `final?`, которая определяет, все ли люди находятся на правом берегу реки в заданном состоянии.

Спроектируйте функцию `render-mc`, которая преобразует состояние головоломки с миссионерами и каннибалами в изображение. ■

Проблема в том, что возвращение конечного состояния ничего не говорит о возможности перейти из начального состояния в конечное. Иначе говоря, `create-next-states` забывает, как она попадает в полученные состояния из заданных. И для решения этой проблемы требуется аккумулятор, но накопленные знания лучше всего связывать с каждым отдельным *PuzzleState*, а не с `solve*` или любой другой функцией.

Упражнение 522. Измените представление из упражнения 521 так, чтобы состояние хранило последовательность предыдущих состояний. Используйте список состояний.

Сформулируйте и запишите назначение аккумулятора и определение данных, объясняющее дополнительное поле.

Измените `final?` или `render-mc` для этого представления, если понадобится. ■

Упражнение 523. Спроектируйте функцию `create-next-states`. Она должна принимать список состояний головоломки и генерировать список всех состояний, которые можно достичь.

Игнорируйте аккумулятор в первой версии `create-next-states`, но гарантируйте, что функция не сгенерирует состояния, в котором каннибалы могут съесть миссионеров.

Во второй версии добавьте обновление поля аккумулятора в структурах состояний и используйте его, чтобы исключить состояния, встречавшиеся на пути к текущему состоянию. ■

Упражнение 524. Используйте представление данных с аккумулятором для изменения функции `solve`. Она должна возвращать список состояний, ведущий от начального состояния `PuzzleState` к конечному.

Исследуйте возможность создания фильма на основе этого списка с использованием `render-mc` для создания изображений. Воспользуйтесь функцией `run-movie` для вывода фильма. ■

33.3. Аккумуляторы как результаты

Еще раз взгляните на рис. 21. На нем изображен треугольник Серпинского и подсказывается, как его создать. В частности, изображения справа объясняют одну из возможных версий генеративной идеи, лежащей в основе процесса:

Данная задача представлена треугольником. Когда треугольник слишком мал для дальнейшего деления, алгоритм ничего не делает; в противном случае находит середины трех сторон треугольника и рекурсивно обрабатывает три внешних треугольника.

В разделе 27.1, напротив, был показан способ составления треугольников Серпинского алгебраически, который не соответствует этому описанию.

Большинство программистов подразумевают под «рисованием» добавление треугольника в некоторый холст. Функция `scene+line` из библиотеки `2htdp/image` конкретизирует эту идею. Она принимает изображение `s` и координаты двух точек и добавляет в `s` линию, проходящую через эти две точки. Функцию `scene+line` легко обобщить до `add-triangle`, а затем и до `add-sierpinski`:

Постановка задачи. Спроектируйте функцию `add-sierpinski`. Она должна принимать изображение и три координаты `Posn`, описывающие треугольник. Используйте эти координаты, чтобы добавить в заданное изображение треугольник Серпинского.

Обратите внимание, что эта задача неявно связана с приведенным выше описанием процесса рисования треугольника Серпинского. Иначе говоря, мы имеем классическую генеративно-рекурсивную задачу и можем начать с классического макета генеративной рекурсии и четырех главных вопросов проектирования:

- задача тривиальна, если треугольник слишком мал, чтобы его можно было разделить на части;
- в тривиальном случае функция возвращает заданное изображение;
- иначе определяются середины сторон заданного треугольника, чтобы добавить еще один треугольник; затем каждый «внешний» треугольник обрабатывается рекурсивно;
- каждый из рекурсивных шагов создает изображение; остается только решить, как совместить эти изображения.

В листинге 123 показан результат подстановки этих ответов в макет определения. Поскольку каждая средняя точка используется дважды, скелет применяет выражение `local`, чтобы сформулировать генеративный шаг на ISL+. Выражение `local` добавляет три новые средние точки и выполняет три рекурсивных применения `add-sierpinsk`.

Листинг 123. Аккумуляторы для представления результатов генеративной рекурсии, макет

```
; Image Posn Posn Posn -> Image
; генеративно добавляет треугольник (a, b, c) в s,
; делит его на три треугольника по средним точкам сторон;
; останавливается, когда (a, b, c) оказывается слишком маленьким
(define (add-sierpinsk scene0 a b c)
  (cond
    [(too-small? a b c) scene0]
    [else
      (local
        ((define scene1 (add-triangle scene0 a b c))
         (define mid-a-b (mid-point a b))
         (define mid-b-c (mid-point b c))
         (define mid-c-a (mid-point c a)))
        (define scene2
          (add-sierpinsk scene0 a mid-a-b mid-c-a))
        (define scene3
          (add-sierpinsk scene0 b mid-b-c mid-a-b))
        (define scene4
          (add-sierpinsk scene0 c mid-c-a mid-b-c)))
      ; ---IN---
      (... scene1 ... scene2 ... scene3 ...)))]))
```

Упражнение 525. Составьте список желаний, которые подразумевает макет:

```
; Image Posn Posn Posn -> Image
; добавляет черный треугольник a, b, c в сцену scene
(define (add-triangle scene a b c) scene)

; Posn Posn Posn -> Boolean
```

```

; является ли треугольник a, b, c слишком маленьким для деления
(define (too-small? a b c)
  #false)

; Posn Posn -> Posn
; определяет среднюю точку между a и b
(define (mid-point a b)
  a)

```

Спроектируйте эти три функции.

Знание предметной области. (1) Для функции `too-small?` достаточно измерить расстояние между двумя точками и проверить, не меньше ли оно некоторого выбранного порога, скажем 10. Расстояние между (x_0, y_0) и (x_1, y_1) вычисляется по формуле:

$$\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2},$$

которая представляет расстояние от точки $(x_0 - x_1, y_0 - y_1)$ до начала координат.

Координаты (x, y) средней точки, лежащей на стороне треугольника между вершинами (x_0, y_0) и (x_1, y_1) , вычисляются по формуле:

$$\left(\frac{1}{2} \cdot (x_0 + x_1), \frac{1}{2} \cdot (y_0 + y_1) \right),$$

т. е. берутся средние точки соответствующих отрезков по осям координат. ■

Теперь, когда у нас есть все вспомогательные функции, можно вернуться к проблеме объединения трех изображений, созданных рекурсивными вызовами. Одно из очевидных решений – использовать функцию `overlay` или `underlay`, но эксперимент в области взаимодействий DrRacket показывает, что функции скрывают лежащие в основе треугольники.

В частности, представьте, что три рекурсивных вызова создают следующие пустые сцены, содержащие по одному треугольнику в соответствующем месте:

```

> scene1

> scene2

> scene3


```

Результат объединения должен выглядеть так:



Но объединение этих фигур с помощью `overlay` и `underlay` не дает желаемого результата:

```
> (overlay scene1 scene2 scene3)

> (underlay scene1 scene2 scene3)

```

Библиотека *image* в ISL+ не имеет функции, объединяющей эти сцены так, как нужно нам.

Давайте еще раз посмотрим на эти взаимодействия. Если *scene1* является результатом добавления верхнего треугольника в заданную сцену, а *scene2* является результатом добавления треугольника в левый нижний угол, то второй рекурсивный вызов должен добавить свой треугольник в результат первого вызова.



При таком подходе передача этой сцены в третий рекурсивный вызов даст именно то, что требуется:



Листинг 124. Аккумулятор для хранения результатов генеративной рекурсии, функция

```
; Image Posn Posn Posn -> Image
; генеративно добавляет треугольник (a, b, c) в s,
; делит его на три треугольника по средним точкам сторон;
; останавливается, когда (a, b, c) оказывается слишком маленьким
; аккумулятор scene0 используется функцией для накапливания
; треугольников в сцене
(define (add-sierpinski scene0 a b c)
  (cond
    [(too-small? a b c) scene0]
    [else
      (local
        ((define scene1 (add-triangle scene0 a b c))
         (define mid-a-b (mid-point a b))
         (define mid-b-c (mid-point b c))
         (define mid-c-a (mid-point c a))
         (define scene2
           (add-sierpinski scene1 a mid-a-b mid-c-a))
         (define scene3
           (add-sierpinski scene2 b mid-b-c mid-a-b)))
        ; ---IN---
        (add-sierpinski scene3 c mid-c-a mid-b-c))))])
```

В листинге 124 показана иная формулировка, основанная на этом понимании. Идею определяют три закрашенных участка. Все они относятся к случаю, когда треугольник достаточно большой и добавляется в заданную сцену. Как только его стороны поделены, первый внешний треугольник рекурсивно обрабатывается с использованием

`scene1`, результатом добавления данного треугольника. Точно так же результат этой первой рекурсии, получивший название `scene2`, используется для второй рекурсии, которая обрабатывает второй треугольник. Наконец, в третий рекурсивный вызов передается `scene3`. В общем, новизна состоит в том, что аккумулятор одновременно является аргументом, инструментом для сбора знаний и результатом функции.

Изучение `add-sierpinski` лучше всего начать с равностороннего треугольника и достаточно большого изображения. Вот определения, которые соответствуют этим двум критериям:

```
(define MT (empty-scene 400 400))
(define A (make-posn 200 50))
(define B (make-posn 27 350))
(define C (make-posn 373 350))

(add-sierpinski MT A B C)
```

Проверьте, получается ли фрактал Серпинского при выполнении этого фрагмента кода. Поэкспериментируйте с определениями из упражнения 525, чтобы создать более разреженные или более плотные треугольники Серпинского.

Упражнение 526. Чтобы получить точки вершин равностороннего треугольника Серпинского, нарисуйте окружность и выберите три точки, лежащие на окружности, разнесенные на 120 градусов, например 120, 240 и 360.

Спроектируйте функцию `circle-pt`:

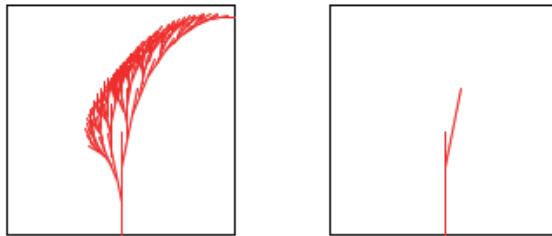
```
(define CENTER (make-posn 200 200))
(define RADIUS 200) ; радиус в пикселях

; Number -> Posn
; определяет координаты для точки на окружности с центром в точке CENTER
; и с радиусом RADIUS в позиции, соответствующей заданному углу factor

; примеры
; какие координаты x и y будут иметь точки:
; 120/360, 240/360, 360/360
(define (circle-pt factor)
  (make-posn 0 0))
```

Знание предметной области. Эта задача проектирования требует знания математики. Одно из возможных решений – преобразовать комплексное число из полярных координат в представление `Posn`. Прочтите документацию к функциям `make-polar`, `real-part` и `image-part` в `ISL+`. Другое решение – использовать тригонометрические функции `sin` и `cos`. Если вы решите пойти данным путем, то помните, что эти функции вычисляют синус и косинус в радианах, а не в градусах. Также имейте в виду, что ось Y на экране направлена сверху вниз, а не снизу вверх. ■

Упражнение 527. Взгляните на два следующих изображения:



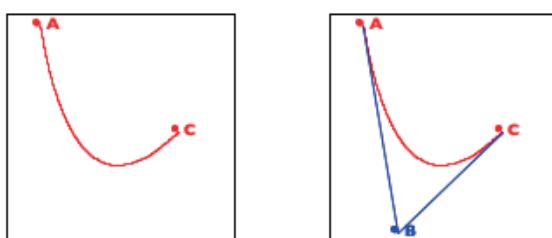
Они демонстрируют, как создать фрактальное дерево саванны, подобно тому, как рис. 21 описывает рисование треугольника Серпинского. На изображении слева показано, как выглядит готовое фрактальное дерево саванны. Изображение справа объясняет этап генеративного построения.

Спроектируйте функцию `add-savannah`. Она должна принимать изображение и четыре числа: (1) координату X базовой точки линии, (2) координату Y базовой точки линии, (3) длину линии и (4) угол наклона линии – и добавлять фрактальное дерево саванны в заданное изображение.

Если линия не слишком короткая, функция добавляет ее в изображение. Затем делит линию на три части и рекурсивно использует две промежуточные точки в качестве новых начальных точек для двух новых линий. Длины и углы наклона двух ветвей изменяются фиксированным образом, но независимо друг от друга. Используйте константы для их определения и поэкспериментируйте с ними, пока вам не понравится получающийся результат.

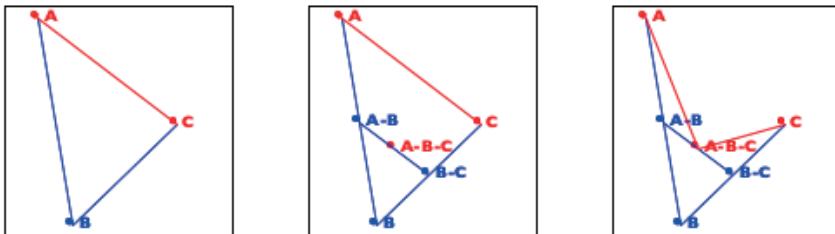
Подсказка. Поэкспериментируйте с укорачиванием каждой левой ветви не менее чем на одну треть и поворотом ее влево не менее чем на $0,15^\circ$. Каждую правую ветвь укорачивайте не менее чем на 20 % и поворачивайте на $0,2^\circ$ в противоположном направлении. ■

Это упражнение предложила Жеральдин Морен (Géraldine Morin). Программистам, занимающимся графикой, часто требуется соединить две точки плавной кривой, где «плавность» определяется некоторой перспективой. Вот два эскиза:



На изображении слева показана плавная кривая, соединяющая точки A и C; на изображении справа – точка перспективы B и угол поля зрения наблюдателя.

Один из методов построения таких кривых предложил французский инженер Пьер Этьен Безье (Pierre Étienne Bézier). Это яркий пример применения генеративной рекурсии, и следующая последовательность объясняет «озарение», стоящее за алгоритмом:



Взгляните на изображение слева. Оно напоминает, что три заданные точки определяют треугольник, и отрезок, соединяющий точки A и C , является фокусом алгоритма. Цель состоит в том, чтобы оттянуть линию, соединяющую A и C , в направлении точки B , *так чтобы получилась плавная кривая*.

Теперь обратимся к изображению посередине. Оно объясняет суть генеративного шага. Алгоритм определяет среднюю точку на двух линиях, ограничивающих угол поля зрения, $A-B$ и $B-C$, а также среднюю точку $A-B-C$.

Наконец, изображение справа показывает, как эти три новые точки генерируют два рекурсивных вызова: один получает новый треугольник слева, а другой – треугольник справа. То есть $A-B$ и $B-C$ становятся новыми точками наблюдения, а линии от A к $A-B-C$ и от C к $A-B-C$ становятся фокусами рекурсивных вызовов.

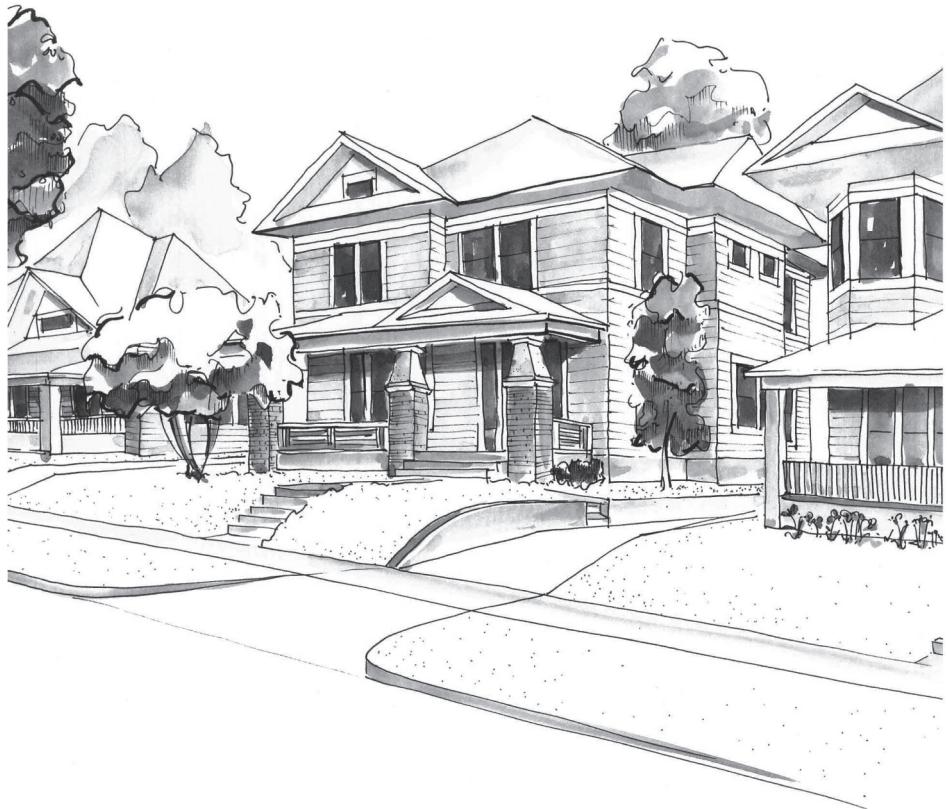
Когда треугольник оказывается достаточно маленьким, мы получаем тривиальный случай. Алгоритм просто рисует треугольник, и он появляется как точка на данном изображении. При реализации данного алгоритма необходимо поэкспериментировать с понятием «достаточно маленький», чтобы кривая выглядела гладкой. ■

34. Итоги

Эта последняя часть была посвящена проектированию с использованием аккумуляторов для накопления знаний во время обхода структуры данных. Добавление аккумулятора помогает устраниить проблемы производительности и обеспечить завершение алгоритма. Вот два с половиной урока этой части:

1. Первый шаг – осознать необходимость использования аккумулятора. Алгоритм, выполняющий рекурсивный обход структуры данных, «забывает» предыдущие аргументы. Если вы обнаружите, что сохранение таких знаний может упростить организацию функции, то подумайте о добавлении аккумулятора. Первым делом нужно переключиться на **макет аккумулятора**.
2. Ключевой шаг – сформулировать описание аккумулятора. Оно должно описывать, **какие знания** и **какие данные** накапливает аккумулятор. В большинстве случаев описание аккумулятора определяет разницу между исходным и текущим аргументами.
3. Третий шаг, второстепенный, состоит в том, чтобы по описанию аккумулятора определить (а) начальное значение аккумулятора, (б) порядок его изменения во время шагов обхода и (в) как использовать накопленные знания.

Идея накопления знаний распространена повсеместно и проявляется во многих формах и видах. Она широко используется в так называемых функциональных языках, таких как ISL+. Программисты, пишущие на императивных языках, применяют аккумуляторы иначе, в основном через операторы присваивания в элементарных конструкциях циклов, потому что последние не могут возвращать значения. Проектирование таких императивных программ с аккумуляторами осуществляется точно так же, как было описано в этой части, но обсуждение соответствующих тонкостей выходит за рамки этой первой книги по систематическому проектированию программ.



copyright © 2002 Tracey Baptiste

Эпилог: что дальше

Вы подошли к концу этого введения в программирование и технику вычислений, или, как мы здесь говорим, в проектирование программ. Вам предстоит еще многое узнать по обоим предметам, но сейчас самое время приостановиться, подвести итоги и заглянуть в будущее.

Компьютерные вычисления

В начальной школе вы познакомились с числами и научились считать. Сначала вы использовали числа для подсчета существующих вещей: три яблока, пять друзей, двенадцать булочек. Чуть позже вы столкнулись со сложением, вычитанием, умножением и даже делением; затем освоили дроби. В конце концов, вы познакомились с переменными и функциями, которые ваши учителя называли алгеброй. Переменные представляют числа, а функции связывают одни числа с другими.

Поскольку числа использовались на протяжении всего этого процесса обучения, вы не слишком задумывались о числах как о способе представления информации о реальном мире. Да, вы начали с трех медведей, пяти волков и двенадцати лошадей; но в старших классах никто не напоминал вам об этой связи с реальностью.

При переходе от математических вычислений к компьютерным особую важность приобретает шаг от информации к данным и обратно. Современные программы обрабатывают изображения, музыку, видео, молекулы, химические соединения, электрические схемы и чертежи. К счастью, вам не нужно кодировать всю эту информацию числами; если бы это было необходимо, то жизнь была бы невообразимо утомительной. Вместо этого компьютерные вычисления обобщают арифметику и алгебру, и, программируя, вы можете писать код (а ваши программы – выполнять вычисления), оперируя строками, логическими значениями, символами, структурами, списками, функциями и многими другими видами данных.

Классы данных и их функции сопровождаются законами эквивалентности, объясняющими их значение, подобно законам для чисел и их функций. Эти законы эквивалентности так же просты, как «`(+ 1 1)` дает в результате `2`» и «`(not #true)` равно `#false`», тем не менее вы можете с их помощью предсказать поведение всей программы. Запуская программу, вы фактически применяете одну из ее многочисленных функций, действие которой можно объяснить с помощью бета-правила, впервые упомянутого в интермеццо 1. После замены переменных значениями вступают в действие законы данных и продолжают действовать до тех пор, пока не останется простое значение или применение очередной функции. Но да, больше нечего сказать о компьютерных вычислениях.

Проектирование программ

Типичный программный проект требует взаимодействия многих программистов, а результат включает тысячи функций. За время существования такого проекта одни программисты уходят, другие приходят. По этой причине проектная структура программ является средством коммуникации между программистами во времени. Код, написанный кем-то другим некоторое время тому назад, должен ясно выражать свою цель и отношения между частями программы, потому что этого другого человека, написавшего его, может уже не быть в команде.

В таком динамичном контексте программисты должны писать программы дисциплинированно, если они хотят создать высококачественный продукт за разумное время. Следование методу системного проектирования гарантирует, что организация программы будет понятна. Другие программисты смогут понять, как работает программа и отдельные ее части, а затем исправить ошибки или добавить новые функциональные возможности.

Процесс проектирования, описанный в этой книге, является одним из таких методов, и вы должны следовать ему всякий раз, когда создаете программы, которые могут кого-то заинтересовать. Вы должны начать с анализа информации и описания данных, представляющих эту информацию. Затем составить план и список необходимых функций. Если список получился большим, то процессу проектирования можно следовать итеративно, начав с подмножества функций и быстро создав продукт, с которым клиент может взаимодействовать. А затем, наблюдая за этими взаимодействиями, вы быстро поймете, какими элементами из списка следует заняться дальше.

Проектирование программы или даже одной функции требует полного понимания того, что она вычисляет. Если вы не сможете кратко описать назначение фрагмента кода, то не сможете создать ничего полезного для будущих программистов. Придумайте и проработайте примеры. Превратите эти примеры в набор тестов. Набор тестов еще приобретет особую важность в будущем, когда потребуется модифицировать программу. Любой, кто изменяет код, сможет повторно запустить тесты и подтвердить, что программа по-прежнему успешно справляется с базовыми примерами.

Рано или поздно в вашей программе обязательно обнаружатся ошибки. Другие программисты могут использовать ее непредвиденным способом. Реальные пользователи могут обнаружить различия между ожидаемым и фактическим поведением. Но если вы будете проектировать код, используя системный подход, то будете знать, как исправить проблему. Вы сможете сформулировать тестовый пример для главной функции программы, который заставляет программу потерпеть неудачу. Из него вы получите тестовые примеры для каждой функции, встречающейся в главной функции. Функции, ко-

торые успешно проходят новые тесты, не являются причиной отказа. Другие функции будут тут же обнаруживать ошибку, а иногда ошибка может проявляться только в определенной комбинации вызовов третьих функций. Если функция, где обнаружилась ошибка, вызывает другие функции, продолжите создание тестов для них. Так, шаг за шагом, вы найдете причину ошибки. Когда программа будет успешно выполнять все тесты, вы будете уверены, что в ней не осталось явных ошибок.

Независимо от вашего усердия, функция или программа может терпеть неудачу при первой же попытке выполнить набор тестов. Найдите время и проверьте ее на наличие конструктивных дефектов и повторяющихся шаблонов кода. Если вы обнаружите какие-либо шаблоны, сформируйте новые или используйте существующие абстракции для их устранения.

Соблюдая эти правила, вы будете создавать надежное программное обеспечение за разумные сроки. Оно будет работать, потому что вы понимаете, почему и как оно работает. Другие, кому потребуется изменить или улучшить ваше программное обеспечение, быстро поймут вашу идею, потому что код ясно сообщает, как он работает и для чего предназначен. Эта книга помогла вам начать долгий путь. Теперь вы должны практиковаться, практиковаться и еще раз практиковаться. И вам предстоит еще узнать о проектировании программ и компьютерных вычислениях намного больше, чем можно поместить в первую книгу.

Разработчикам и программистам

Учитывая текущий уровень ваших знаний, вы без труда выучите Racket – язык, лежащий в основе учебных языков в этой книге. Прекрасным введением в него вам послужит книга «Realm of Racket».

Возможно, вам интересно узнать, что делать дальше. Ответ прост: продолжайте писать программы и изучать информатику.

Ваша следующая задача как студента, изучающего проектирование программ, – узнать, как процесс проектирования применяется в условиях полноценного языка программирования. Некоторые из этих языков похожи на учебные языки, и переход на них будет легким. Другие требуют иного мышления, потому что предлагают иные средства для формулирования определений данных (классов и объектов) и сигнатур, помогающие перекрестно проверять их перед запуском программы. Кроме того, вам также нужно будет познакомиться с приемами масштабирования процесса проектирования для создания и использования так называемых фреймворков («стеков») и компонентов. Фреймворки – это абстрактные компоненты с определенными функциональными возможностями, реализующие, например, графический пользовательский интерфейс, соединение с базами данных и веб-серверами, которые являются общими для многих программных систем. Вам

нужно научиться создавать реализации этих абстракций, и ваши программы будут использовать эти реализации для создания согласованных систем. Точно так же обучение созданию новых компонентов систем является неотъемлемой частью расширения ваших навыков.

Студентам факультетов теоретической информатики придется еще расширить свое понимание вычислительного процесса. В этой книге основное внимание уделяется законам, описывающим сам процесс. Чтобы действовать как настоящий инженер-программист, вам необходимо познакомиться с приемами оценки производительности как на теоретическом, так и на практическом уровне. Более глубокое изучение понятия большого О – первый, маленький шаг в этом направлении; научиться измерять и анализировать производительность программы – вот настоящая цель, потому что вы, как разработчик, будете использовать этот навык постоянно. Помимо этих основных идей, вам также понадобятся знания, касающиеся оборудования, сетей, многоуровневого программного обеспечения и специализированных алгоритмов.

Бухгалтерам, журналистам, хирургам и всем остальным

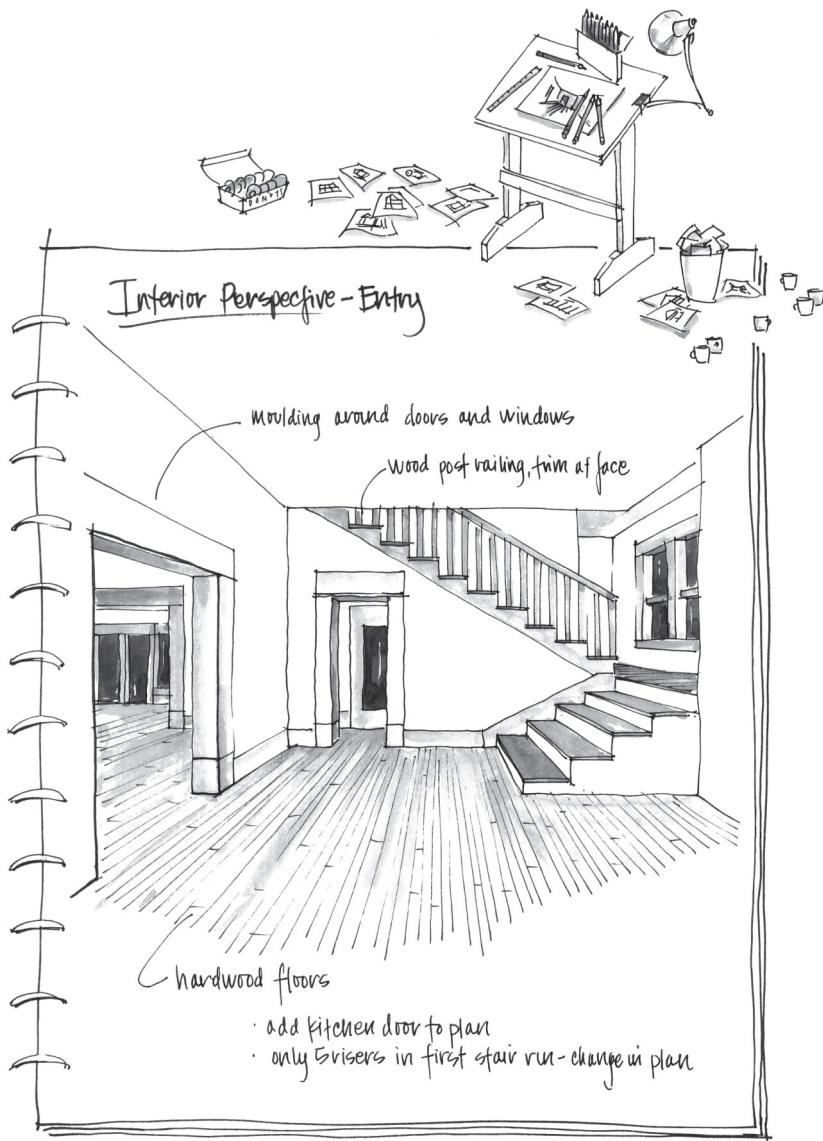
Кто-то из вас наверняка пожелает поближе познакомиться с информатикой и программированием. Теперь вы знаете, что компьютерные вычисления – это всего лишь обобщение математических вычислений, и, возможно, вы даже почувствовали, насколько полезно знание особенностей проектирования программ. Даже если вы никогда больше не будете проектировать программы, вы уже знаете, что отличает гаражного программиста от серьезного разработчика программного обеспечения. Взаимодействуя с разработчиками, вы будете знать, насколько важно системное проектирование, потому что оно влияет на качество вашей жизни и прибыль вашего бизнеса.

Однако многие из вас так или иначе будут заниматься «программированием» на регулярной основе, даже не подозревая об этом аспекте своей деятельности. Представьте на мгновение журналиста. Сначала он собирает информацию и данные, структурирует их, систематизирует и добавляет эпизоды. Немного подумав, вы поймете, что это как раз и есть первый шаг процесса проектирования. Теперь возьмем пример с семейным врачом, который, проверив ваши симптомы, формулирует гипотезу о том, что может на вас повлиять. Видите в этом второй шаг? Или представьте юриста, который иллюстрирует суть спора рядом примеров; примеры – это третий шаг. Наконец, инженер-строитель проверяет конструкцию моста после возведения, чтобы убедиться, что она соответствует плану и лежащим в его основе статическим расчетам. Проверка – это форма тестирования – шестой шаг процесса; он сравнивает фактические результаты изме-

рений с ожидаемыми значениями прогнозных расчетов. Каждый из этих профессионалов разрабатывает систему для эффективной и результативной работы; и если отбросить несущественные отличия, эта система явно напоминает процесс проектирования, представленный в данной книге.

Теперь, когда вы понимаете, что многие действия являются формой программирования, вы сможете перенести дополнительные идеи из процесса проектирования в свою собственную жизнь. Например, обнаружив какие-то шаблоны, вам не потребуется много времени, чтобы создать «абстракцию» – единую точку управления, – упрощающую вашу будущую работу. То есть независимо от того, станете ли вы бухгалтером, врачом или кем-то еще, помните о процессах проектирования, куда бы вы ни пошли и чем бы вы ни занимались.

Упражнение. Напишите небольшой рассказ о том, как процесс проектирования может помочь вам в выбранной вами профессии.



Предметный указатель

Символы

1String (Символ), 129
1Transition, определение данных, 524
2htdp/abstraction, библиотека, 461
2htdp/image, библиотека, 35
2htdp/universe, библиотека, 41
3LON, список с тремя числами, 241
#false, 34
#true, 34
.v1, функция, 674
.v2, функция, 674

А

add1*, функция, 377
add1-to-each, функция, 393
add3, функция, 136
add-3-to-all, функция, 432, 416
add-as-last, функция, 669
add-at-end, функция, 303
add-queen, функция, 641
Addr, структура, 404
add-ranks, функция, 373
add-savannah, функция, 704
add-sierpinsk, функция, 700, 702
add-to-each, функция, 660
add-to-pi, функция, 269, 679
add-triangle, функция, 700
AL, определение данных, 506
all-words-from-rat?, функция, 342, 458
alternative-words, функция, 341
andmap, функция, 402
animate, 41
Any, определение данных, 204
area-of-disk, функция, 203
argmax, функция, 403
arrangements, функция, 343, 636, 458
Association, определение данных, 339, 506
Atom, определение данных, 479
Attribute, определение данных, 515
Attribute.v3, определение данных, 529
average, функция, 261

Б

Ball-1d, определение данных, 168
Ball-2d, определение данных, 168

between-0-and-1?, функция, 208
Block, определение данных, 352
blue-eyed-ancestor?, функция, 476
blue-eyed-child?, функция, 473
blue-eyed-child-in-forest?, функция, 478
board0, функция, 641
Board, определение данных, 642
BSL+, язык, 312
BSL-da-all, определение данных, 510
BSL-expr, определение данных, 502
BSL-fun-def, определение данных, 508
BSL-fun-def*, определение данных, 509
BSL-fun-expr, определение данных, 507
BSL-var-expr, определение данных, 505
BST-инвариант, 489
BT, определение данных, 487
build-list, функция, 402
bulletize, функция, 522
bundle, функция, 575, 577, 589

С

cat, функция, 259
Cell, определение данных, 549
cf*, функция, 389
cf*-from-map1, функция, 391
checked-area-of-disk, функция, 205
checked-f, макет функции, 204
check-error, тест, 229
check-expect, тест, 229
check-member-of, тест, 282, 229
check-random, тест, 229
check-range, тест, 229
check-satisfied, проверка, 321
check-satisfied, тест, 283, 229
check-with, предложение, 208
check-within, тест, 229
Child, структура, 469
CI, определение данных, 638
close?, функция, 418, 433
close-to, функция, 418
Color, определение данных, 178
compare, функция, 429
cond, 43
cond, предложения, 218
connect-dots, функция, 329

Connection, определение данных, 664
 ConsOrEmpty, определение данных, 693, 244
 ConsPair, структура, 243
 constant, функция, 616
 contains?, функция, 377, 439
 contains-cat?, функция, 376
 contains-dog?, функция, 376
 contains-flatt?, функция, 647, 245
 Content, определение данных, 548
 copier, функция, 268
 count, функция, 280, 481, 484
 count-atom, функция, 482, 484
 count-sl, функция, 482, 484
 create-date, функция, 337
 create-dir, функция, 498
 create-inex, функция, 563
 create-rocket-scene, функция, 446, 126
 create-track, функция, 337
 cross, функция, 458
 CTemperature, представление данных, 260

D

Date, определение данных, 337
 DB, определение данных, 548
 define, 38
 define-struct, ключевое слово, 159
 depth, функция, 272, 465
 Dir*, определение данных, 497
 Direction, определение данных, 540
 Dir.v1, определение данных, 496
 Dir.v2, определение данных, 497
 Dir.v3, определение данных, 497
 distance, 50
 distance-to-0, функция, 156, 157
 divisors, функция, 596
 door-action, функция, 153
 door-closer, функция, 151
 door-render, функция, 153
 door-simulation, функция, 153
 dots, функция, 432
 drop, функция, 578

E

Editor, определение данных, 300, 183
 editor-del, функция, 308
 editor-ins, функция, 307
 editor-kh, функция, 305
 editor-lft, функция, 308
 editor-render, функция, 304, 309
 editor-rgt, функция, 308
 editor-text, функция, 309

EmailMessage, структура, 322
 encode-letter, функция, 298
 Entry, определение данных, 167
 enumerate, функция, 456
 enumerate.v2, функция, 460
 Equation, представление данных, 621
 eval-var-lookup, функция, 507
 ExpectsToSee.v1, определение данных, 202
 ExpectsToSee.v2, определение данных, 202
 extract1, функция, 409
 extract, функция, 378

F

FF, определение данных, 477
 file->list-of-lines, функция, 608
 File, определение данных, 606
 File*, определение данных, 497
 file-statistic, функция, 298
 File.v1, определение данных, 496
 File.v2, определение данных, 497
 File.v3, определение данных, 497
 filter, функция, 402
 find, функция, 360, 426, 441, 525, 410
 find-next-state, функция, 359
 find-next-state.v1, функция, 357
 find-next-state.v2, функция, 357
 find-path, функция, 629
 find-path/list, функция, 631
 find-root, функция, 602
 first-line, функция, 608
 f*ldl, функция, 682
 fly, функция, 138, 142
 f*oldl, функция, 405
 foldl, функция, 403
 foldr, функция, 403
 food-check-create, функция, 349
 food-create, функция, 349
 Four, определение данных, 291
 FSM, определение данных, 355, 635, 524
 fsm-match?, функция, 635
 FSM-State, определение данных, 355, 635, 524
 FSM.v2, определение данных, 361

G

GamePlayer, структура, 322
 gcd, функция, 592
 gcd-generative, функция, 594
 gcd-structural, функция, 592, 595
 get, функция, 532
 gift-pick, функция, 546
 Graph, определение данных, 629

Н

h/a, функция, 678
 handle-key-events, функция, 129
 height, функция, 676
 height.v2, функция, 676, 677
 HelpDesk, 32
 HM-Word, определение данных, 545
 how-many, функция, 255, 265, 646

I

im*, функция, 397
 image*, функция, 394
 ImageStream, представление данных, 446
 in?, функция, 281
 index, функция, 441
 in-dictionary, функция, 343
 inex->number, функция, 563
 inf, функция, 380, 650
 infl, функция, 651
 inf.v2, функция, 408
 in-naturals, функция, 460
 in-range, функция, 460
 insert, функция, 318
 insertion-sort, функция, 452
 inside?, функция, 442
 integrity-check, функция, 550, 552
 Inventory, определение данных, 390
 invert, функция, 668
 invert.v2, функция, 671
 IR, определение данных, 390
 IR, структура, 382
 is-prime?, функция, 681
 is-queens-result?, функция, 641

К

keep?, функция, 555
 keep-good, функция, 433
 keyh, функция, 276

Л

Label, определение данных, 548
 Lam, определение данных, 688
 large-1, функция, 379
 large, функция, 378
 largers, функция, 582
 LAssoc, определение данных, 339
 last-item, функция, 464
 launch, функция, 138, 141
 Letter, представление данных, 334
 Letter-Count, определение данных, 334
 lhs, функция, 622

light=? , функция, 209
 light?, функция, 210
 Line, определение данных, 606
 linear, функция, 616
 line-processor, функция, 295
 List, определение данных, 383
 list, функция, 312
 listing, функция, 404
 listing.v2, функция, 406
 List-of, определение данных, 381
 List-of-amounts, определение данных, 258
 list-of-attributes?, функция, 517
 List-of-names, определение данных, 241
 List-of-numbers, определение данных, 381
 List-of-numbers-again, определение данных, 382
 List-of-posns, представление данных, 279
 List-of-shots, представление данных, 275
 List-of-String, определение данных, 381
 List-of-strings, определение данных, 251
 List-of-temperatures, представление данных, 260
 List-of-words, определение данных, 345
 list-pick, функция, 537
 LLists, определение данных, 339
 LLS, определение данных, 297
 LNum, определение данных, 383
 Lo1s, определение данных, 302
 LOD, определение данных, 543
 LOFD, определение данных, 497
 Lon, определение данных, 381
 lookup-def, функция, 509
 Los, определение данных, 381
 LOT, определение данных, 361
 Low, определение данных, 290
 LStr, определение данных, 383
 LTracks, определение данных, 337

М

main, функция, 131, 134
 make-cell, функция, 371
 make-posn, функция, 154
 make-row, функция, 371
 make-table, функция, 371
 map1, функция, 390
 map, функция, 402
 Matrix, определение данных, 299
 Maybe, определение данных, 384
 mid-point, функция, 701
 mirror, функция, 680
 Missile, определение данных, 189
 MissileOrNot, определение данных, 197

missile-or-not?, функция, 206
 missile-render, функция, 194
 missile-render.v2, функция, 197
 mk-circle, функция, 444
 mk-combination, функция, 445
 mk-point, функция, 443
 mk-rect, функция, 445
 move-right, функция, 465
 multiply, функция, 269
 my-first-web-page, функция, 368
 myList, определение данных, 694

N

N99, определение данных, 563
 names, функция, 390
 names-from-map1, функция, 391
 neighbor, функция, 665
 NEList-of-temperatures, представление данных, 262
 Nelon, определение данных, 380
 NELoP, определение данных, 329
 newton, функция, 612
 next, функция, 123
 Node, определение данных, 629, 664
 Non-empty-list, определение данных, 464
 non-same, функция, 547
 not-1-1?, функция, 349
 not-member-1?, функция, 283
 n-queens, функция, 641
 nxt, функция, 131

O

ormap, функция, 402
 oscillate, функция, 571
 our-cons, функция, 694, 244
 our-first, функция, 694, 244
 our-length, функция, 694

P

Pair, представление данных, 279
 Pair-boolean-string, структура, 382
 Pair-number-image, структура, 382
 parse, функция, 504
 parse-atom, функция, 504
 parse-sl, функция, 504
 Path, определение данных, 499, 630
 path-exists?, функция, 664
 path-exists?/a, функция, 666
 path-exists.v2?, функция, 667
 Phone, определение данных, 291
 PhoneRecord, определение данных, 536

place-dot, функция, 394
 place-queens, функция, 641
 play, функция, 545
 plus5, функция, 377
 Polygon, определение данных, 324
 posn+, функция, 174
 posn, структура, 154, 155
 posn-up-x, функция, 170
 posn-x, функция, 156
 posn-y, функция, 156
 pr*, функция, 397
 Predicate, определение данных, 548
 product, функция, 393, 401
 project, функция, 553, 554, 557
 project.v1, функция, 556
 PuzzleState, определение данных, 697

Q

QP, определение данных, 638
 quasiquote и ` , оператор, 367
 quick-sort<, функция, 581, 582, 589
 quote и ', оператор, 364

R

R3, определение данных, 179
 r3-distance-to-0, функция, 180
 random, функция, 195
 random-pick, функция, 547
 random-posns, функция, 441
 ranking, функция, 373
 RD, представление данных (русская матрешка), 271
 RD.v2, определение данных, 465
 read-file, функция, 291
 read-itunes-as-lists, функция, 340
 read-itunes-as-tracks, функция, 337
 read-lines, функция, 293
 read-plain-xexpr/web, функция, 529
 read-words, функция, 293
 read-xexpr, функция, 529
 read-xexpr/web, функция, 529
 reduce, функция, 401
 relative->absolute, функция, 660
 relative->absolute.v2, функция, 662
 remove-first-line, функция, 608
 render, функция, 131
 render-enum, функция, 523
 render-item, функция, 523
 render-line, функция, 326
 render-poly, функция, 325
 render-polygon, функция, 331
 render-state.v1, функция, 357

render-state.v2, функция, 357
 render/status, функция, 134
 render-word, функция, 545
 replace-eol-with, функция, 533
 reset-dot, функция, 170
 rev, функция, 301
 reward, функция, 125
 rhs, функция, 622
 root-of-tangent, функция, 611
 rotate, функция, 680
 rotate.v2, функция, 681
 Row, определение данных, 299, 549
 row-filter, функция, 555, 556
 row-integrity-check, функция, 550

S

S-выражения, 469, 479
 S, определение данных, 563
 sales-tax, 144
 sales-tax, функция, 144, 145
 scene+dot, функция, 168
 Schema, определение данных, 548
 search, функция, 322, 655
 searchS, функция, 655
 set-, функция, 281
 set-.L, функция, 282
 set-.R, функция, 282
 S-expr, определение данных, 479
 Shape, представление данных, 442
 Shot, представление данных, 275
 ShotWorld, представление данных, 275
 show, функция, 138, 139, 140
 si-control, функция, 196
 si-game-over?, функция, 195
 SIGS, определение данных, 189
 SIGS.v2, определение данных, 196
 si-move, функция, 195
 si-move-proper, функция, 196
 SimpleGraph, определение данных, 664
 simulate, функция, 356, 524, 410, 411
 simulate-xmachine, функция, 526, 527
 SimulationState.v1, определение
данных, 357
 SimulationState.v2, определение
данных, 357
 si-render, функция, 192
 si-render-final, функция, 195
 SL, определение данных, 479
 slope, функция, 610
 small-1, функция, 379
 small, функция, 378
 smallers, функция, 582, 590

SOE, представление данных, 621
 solve, функция, 697
 Son, представление данных, 280
 Son.L, представление данных, 280
 Son.R, представление данных, 280
 sort>, функция, 318
 sort>/bad, функция, 322
 sort, функция, 452, 402
 sort-cmp, функция, 435, 408
 sort-cmp/bad, функция, 438
 sort-cmp/worse, функция, 439
 sorted, функция, 436, 437
 sorted-variant-of, функция, 438
 sorted-variant-of.v2, функция, 440
 SpaceGame, определение данных, 181
 Spec, определение данных, 548
 square, функция, 616
 squared>?, функция, 380
 state-as-colored-square, функция, 358, 410
 stock-alert, функция, 530, 531
 StockWorld, определение данных, 531
 string->number, функция, 136
 string-append-with-space, функция, 404
 substitute, функция, 491
 substitute.v3, функция, 493
 sum, функция, 262, 393, 401
 sum-evens, функция, 460
 sum.v2, функция, 672
 sup, функция, 380

T

Table, структура, 605
 tab-sin, функция, 393
 tab-sqrt, функция, 393
 take, функция, 577
 Tank, определение данных, 189
 tank-render, функция, 194
 Tetris, определение данных, 351
 Three, определение данных, 291
 TID, определение данных, 543
 tl-next, функция, 148
 tl-next-numeric, функция, 148
 tl-next-symbolic, функция, 149
 tl-render, функция, 148
 tock, функция, 275
 to-image, функция, 275
 too-small?, функция, 701
 Track, определение данных, 337
 traffic-light-next, функция, 128
 Transition, определение данных, 355
 Transition.v2, определение данных, 360
 Transition.v3, определение данных, 361

transpose, функция, 299

Tree, определение данных, 675

U

UFO, определение данных, 189, 171

ufo-move-1, функция, 172

ufo-render, функция, 194

undeclareds, функция, 689

UnitWorld, определение данных, 208

url-exists?, функция, 529

V

VAnimal, определение данных, 200

VCat, определение данных, 200, 185

VCham, определение данных, 200, 186

vec, определение данных, 205

Vel, определение данных, 167

W

wage, функция, 284

wage*, функция, 284

wages*.v2, функция, 534

weekly-wage, функция, 536

Word, определение данных, 345

words->strings, функция, 343

words-on-line, функция, 294

Work, определение данных, 290

X

x+, функция, 170

X1T, определение данных, 526

XEnum, определение данных, 519, 520

Xexpr, определение данных, 514, 515, 528

xeexpr?, функция, 529

xeexpr-attr, функция, 515

xeexpr-content, функция, 515

xeexpr-name, функция, 515

XItem, определение данных, 519

xm->transitions, функция, 527

XMachine, определение данных, 526

XML, обработка, 512

XML-перечисления, отображение, 518

xm-state0, функция, 527

XWord, определение данных, 518

XYZ, определение данных, 206

A

Абстрагирование, 375

Абстрагирование лямбда-выражений, 432

Абстрагирование макетов, 400

Абстрагирование примеров, 389

Абстрактное время, 645

Абстракция, 375

Адаптация рецепта проектирования, 585

Адаптивное интегрирование, 620

Аккумуляторы, 658, 670

дополнительные примеры, 687

и деревья, 687

как результаты, 699

Алгоритм

быстрой сортировки, 579

Кеплера, 617

Ньютона, 612

Серпинского, 599

Алгоритмы с возвратами, 627

Анализ

данных, 494

наилучшего случая, 647

наихудшего случая, 647

среднего случая, 648

Аргументы, 73

Аргумент завершения, 588, 633

Арифметика

изображений, 63

логических значений, 66

строк, 60

с числами фиксированного

размера, 561

Арифметические выражения, 31

Атомарные данные, 57

Атомарные предложения, 215

Б

Базовая грамматика BSL, 214

Базовые варианты, 254

Базовый словарь BSL, 213

Базы данных, 548

Бинарное дерево, 487

Бинарное дерево поиска, 487, 489

Бинарные функции, 73

Бинарный поиск, 600

Большое О, 653

Буквы, 334

В

Вселенная данных, 174

Вспомогательные функции, 80

Встроенные операции, 34

Выигрыш от использования

аккумулятора, 668

Выражения, 30

Высота, дерева, 675

Вычисления

- по короткой схеме, 223
- с локальными определениями, 411
- с лямбда-выражениями, 429
- со списками, 249
- со структурами, 163
- со структурами `posn`, 155
- Вычисляемые константы, 85

Г

- Гауссово исключение, 621
- Генеалогическое древо, 470
- Генеративная рекурсия, 574
- Глобальная переменная, 84
- Глобальные константы, 83
- Графический пользовательский интерфейс, 85
- Графический редактор, 182
- Графический редактор, еще раз, 300
- Графы, 627

Д

- Данные произвольного размера, 237
- Действительное, 58
- Древо партии, 695
- Деревья, 469
- Детализация, 135
- Диаграмма переходов, 146
- Дополнение, 124
- Дополнительные примеры использования аккумуляторов, 687

Е

- Естественная рекурсия, 254

Ж

- Жучок (bug), 106

З

- Завершение, 588
- Завершимость рекурсии, 587
- Заголовок, 103
- Заголовок функции, 73, 585, 217
- Замечания о списках и множествах, 279
- Заявление о назначении функции, 585
- Знание предметной области, 106
- Значение языка, 212
- Значения и вычисления, 220
- ЗР (запуск ракеты), 137
- ЗРОО (запуск ракеты с обратным отсчетом), 138

И

- Игры со словами, 340, 345
- Изображение, 63
- Импликация, 75
- Инвентаризация, 104
- Интегрирование, 614
- Интерактивные программы, 85, 89
- Интервалы, 131, 132
 - закрытые, 132
 - открытые, 132
- Интерпретация
 - всего и вся, 509
 - выражений, 501
 - переменных, 504
- Информация, 99
- Итеративное уточнение, 291, 361, 494
- Итеративное уточнение задачи, 291

К

- Класс, 101
- Ключевые слова, 214
- Комбинаторы, 255
- Комментарии, 48
- Композиция функций, 80, 314
- Конечное состояние, 202
- Конечные автоматы, 354, 146
- Конкретное время, 645
- Константы, 46, 72, 83
- Константы-литералы, 84
- Конструктор, 226, 159
- Координата, определение данных, 199
- Координата x, 154
- Координата y, 154
- Корень функции, 600
- Космические захватчики, игра, 353
- Кривые Безье, 705

Л

- Левая часть, 217
- Леса, 477
- Логическая ошибка, 106
- Логические выражения, 223
- Локальные определения, 405
- Лямбда-выражения, 426

М

- Магические числа, 49
- Макет, 191, 253, 273, 586, 104, 145
 - функции, 586
 - функции с аккумулятором, 670
- Макеты, 400

Макеты функций, 180
 Манхэттенское расстояние, 158
 Матрица, 299
 Метод Ньютона, 610
 Мировые программы, 91
 Миры с конечными состояниями, 146
 Множества, 279
 Модель-представление-контроллер, 100
 Модульное тестирование, 110
 Модульные тесты, 317

Н

Натуральные числа, 266
 Начальное состояние, 91, 202
 Неотрицательное число, 58
 Непустые списки, 260
 Нестандартная рекурсия, 575
 Неточные числа, 31, 58
 Н-Светофор, 148

О

Область видимости, 448
 взаимодействий, 30
 определений, 30
 Обобщающие вспомогательные функции, 323
 Обработка XML, 512
 Обработчик события, 90
 Обработчики событий, 85
 Одновременная обработка двух списков, 533, 534, 537
 Операнды, 31
 Операционная система, 90
 Описание аккумулятора, 670
 Описание назначения, 103
 Опорный элемент, 579
 Определение
 данных, 187, 190, 200, 251, 253, 590, 212, 213
 константы, 84
 собственных структур, 158
 спецификаций с помощью лямбда-выражений, 435
 структурного типа, 158
 термина «порядка», 651
 функции, 39, 215
 функций с помощью
 лямбда-выражений, 427
 Определения, 72
 данных, 442, 101, 177
 констант, 72, 224
 структур, 226

функций, 72
 Ориентированные графы, 627
 Отличающиеся сходства, 378
 Отрицательное число, 58
 От функций к программам, 107
 Ошибки
 ввода, 203
 времени выполнения, 220
 в BSL, 220

П

Пакетные программы, 85
 Параметры, 73
 Параметрические определения данных, 381
 Парсинг, 503
 Переполнение, 567
 Перечисление, 127
 Питон, игра, 347
 Плоские представления, 162
 Повторяющиеся вспомогательные функции, 316
 Подвыражения, 67
 Подъем выражений, 552
 Позиция, 62, 130
 Поиск наибольшего общего делителя, 591
 Поиск пути в графе, 633
 Полная грамматика BSL, 229
 Положительное целое, 58
 ПоложительноеЧисло, 124
 Потеря значимости, 567
 Поток изображений, 446
 Пошаговый движок, 77
 Поэзия S-выражений, 469
 Правая часть, 84, 217
 Правило проектирования, 400
 Предикат, 70, 226
 равенства, 209
 структуры, 159
 Предложения cond, 123
 Предметная область, 99, 107
 Предметно-ориентированные языки, 523
 Предопределенные операции, 34, 57
 Представление данных, 187, 194, 197, 211, 585, 590, 212, 143, 154
 Представление с помощью
 лямбда-выражений, 442
 Представления данных, 442
 Преобразование простых функций
 в функции с аккумуляторами, 672
 Префиксная запись, 57
 Приглашение к вводу, 30

Применение функции, 40, 74
 Примеры
 данных, 191, 251, 171, 178
 применения функций, 252, 255, 273, 172
 Примитивные операции, 57
 Природа чисел, 561
 Проблема
 генеративной рекурсии, 663
 структурной обработки, 659
 Проверка
 состояния мира, 207
 целостности, 549
 Программирование
 со списками, 245
 со структурами, 167
 с условиями, 122
 с posn, 156
 Программное обеспечение, 90
 Программы, 30, 85
 Программы, управляемые событиями, 85
 Проектирование, 55
 абстракций, 389
 методом композиции, 312
 программ, 110
 с использованием абстракций, 420
 с использованием взаимосвязанных
 данных, 485
 с использованием детализации, 187, 143
 с использованием структур, 178
 функций, 99
 функций с аккумулятором, 668
 функций с двумя сложными
 аргументами, 542
 Проекции и выборки, 553
 Производительность, 658
 Простейшие операции, 33
 Процесс проектирования, 193, 211, 437,
 585, 142

P

Расстояние, 50
 Регулярные выражения, 202
 Редактирование, 30
 Рекурсия, 254
 без структуры, 575
 игнорирующая структуру, 579
 Рефакторинг программы, 47
 Рецепт
 проектирования, 585
 проектирования абстракций, 389
 проектирования, генеративная
 рекурсия, 585

Рецепт проектирования
 два измерения, 211
 и индукция, 254
 мировых программ, 138
 на основе абстракций, 520
 на основе данных с перекрестными
 ссылками, 520
 одновременная обработка, 542
 с абстракциями, 420
 с аккумулятором, 670
 с определениями данных,
 ссылающимися на самих себя, 251
 со смешанными данными, 190
 со структурами, 197
 структурные данные, 585
 структуры, 179
 Рецепт уточнения, 501
 Русская матрешка, 270

C

Свертка, 403
 Светофор, 128
 Селектор, 159
 Селекторы, 226
 Серпинского алгоритм, 599
 Серпинского треугольник, 597
 Сигнатура функции, 102, 191, 585
 Сигнатуры, 394
 Символ (1String), 129
 Символ (Symbol), 366
 Синтаксис BSL, 212
 Синтаксические категории, 213
 Синтаксические ошибки, 220
 Синтаксический анализ, 606
 Синус, 57
 Словари, 333
 Словарь BSL, 212
 Смешанные операции со строками, 61
 Смешивание миров, 200
 Событие, 85
 Событие клавиатуры, 90
 Событие Клавиатуры, 129, 135
 Событие Мышь, 127
 Создание списков, 238
 Сообщения об ошибках в BSL, 229
 Сопоставление с образцом, 461
 Составные данные, 154
 Состояние Двери, 150
 Состояние мира, 91
 Состояние Мира, 132
 Состояние программы, 91
 Списки, 284, 238

- в списках, 291
в мировых программах, 274
создание, 238
желаний, 108
С-Светофор, 149
Стоимость вычислений, 644
Строки, 60
Строки
режима, 63
цветов, 63
Структурная и генеративная рекурсии, 590
Структурные данные, 468
Структурные типы, 154
Структуры, 154
в мире, 181
в списках, 287
Сходства, 376
в определениях данных, 381
в сигнатурах, 394
в функциях, 376
- Т**
- Табличный метод, 254, 302, 534
Текущее состояние, 91
Тело функции, 73, 217
Теорема индукции, 254
Тернарные функции, 73
Тестирование, 108
Тесты в BSL, 228
Тетрис, игра, 350
Токены, 607
Точка привязки, 64
Точные числа, 58
Треугольник Серпинского, 597, 699
Триангуляция, 622
- У**
- Унарные функции, 73
Упрощение функций, 491, 541
Условия cond, 123
Условные вычисления, 124
Условные строки, 43
Уточнение, 494
определений данных, 496
функций, 498
Утраты знаний, 659
Учебные пакеты, 35
- Ф**
- Файлы, 291
Фактические аргументы, 217
Формула расстояния, 59
Фракталы, 597
Фрактальное дерево саванны, 704
Функции, 72
со встроенными проверками, 204
создающие списки, 284
Функции – это значения, 384
Функциональная абстракция, 377
- Ц**
- Целое, 58
Цена, 144
Циклы, 453
Цитирование, 364
- Ч**
- Числа в *SL, 568
ЧислоИлиЛожь, 136
Чтение XML, 528
- Ш**
- Шаблон, 586

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: (499) 782-38-89, электронная почта: books@aliens-kniga.ru.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: <http://www.galaktika-dmk.com/>.

Маттиас Фелляйзен
Роберт Брюс Финдлер
Мэтью Флэтт
Ширам Кришнамурти

Как проектировать программы

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*
Научные редакторы *Иванов П. Б., Чичигин А. Д.,
Сыровецкий Ю. А., Бронников С. В.*
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 58,83. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

Цель этой книги – познакомить читателей с системным подходом к проектированию и сформировать у них правильные навыки программирования, которые предполагают пошаговое планирование и понимание рабочих задач на каждом этапе создания программы.

Рассматриваемые темы:

- фундаментальные понятия системного проектирования;
- язык обучения: синтаксис и семантика;
- типы данных;
- нотация для записи больших объемов данных: цитирование и антицитирование;
- создание и использование абстракций;
- тестирование программ и функций и др.

В фокусе книги – общие принципы проектирования программ, поэтому используется не стандартный промышленный язык, а специализированный язык обучения Racket, а также среда программирования DrRacket, которая обеспечивает увлекательное обучение. Возможности среды растут по мере того, как читатель осваивает материал, – вплоть до поддержки полноценного языка, пригодного для решения всего спектра задач программирования.

Издание адресовано широкому кругу читателей, которые хотят научиться мыслить алгоритмически и создавать по-настоящему ценное программное обеспечение.

The MIT Press

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@aliens-kniga.ru

