Содержание

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.

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.

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.

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.*

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-

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 короткий 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 определение 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 окупаются, когда приходит время сделать творческий шаг – завершить определение функции, потому что оказывают необходимую помощь почти 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 многогранности 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.

Преподавателям. *Вы можете объяснить, что BSL – это школьная алгебра с дополнительными формами данных и множеством предопределенных функций для работы с ними.*

Наше решение: начать со знакомства с нашим собственным 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.**

Выбирая 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 какое-то 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 устраняет 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.

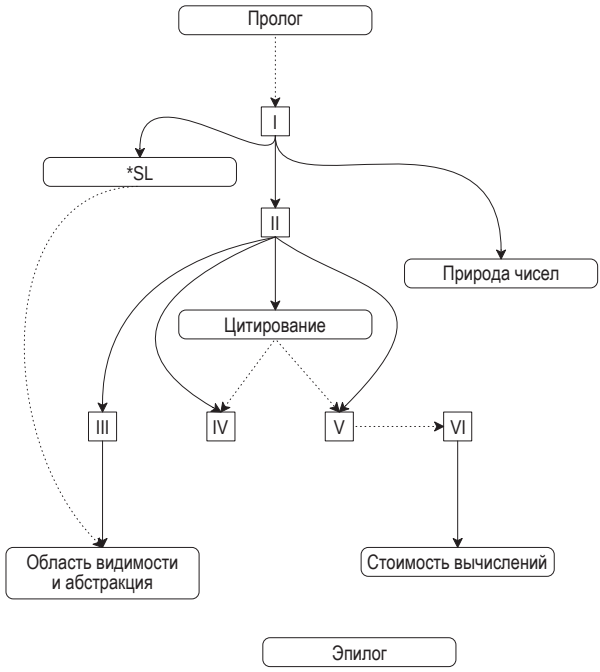


Рис. 1. Зависимости между частями книги и интермеццо

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;

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

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 сайте[1], а их адаптированные версии войдут во второй том данной серии «How to Design Components»;

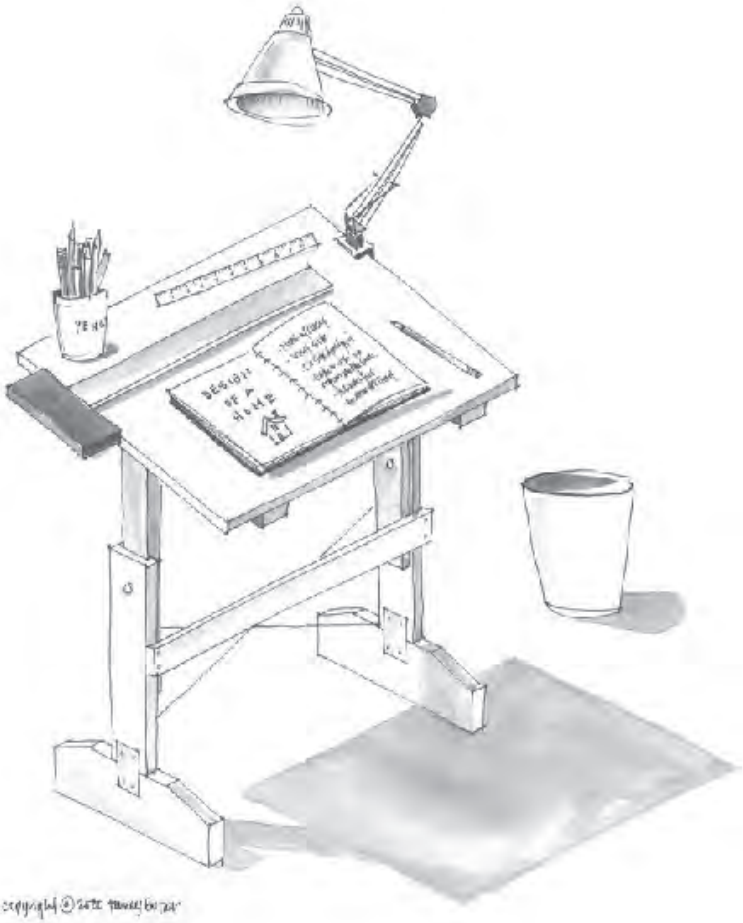
7) в этом издании изменены обучающие пакеты с примерами и упражнениями. Предпочтительным стилем связывания этих библиотек считается применение инструкции `require`, но вы все еще можете добавлять их через меню в DrRacket;

8) наконец, во втором издании несколько изменились терминология и обозначения:

Второе издание	**Первое издание**
сигнатура	контракт
детализация	объединение
`'()`	`empty`
`#true`	`true`
`#false`	`false`

Последние три отличия значительно улучшают цитирование списков.

[1] https://htdp.org/2003-09-26/Book/. – *Прим. перев.*



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 программировали, а вы вычисляли. В каком-то смысле это все, что нужно для программирования и вычислений.

Загрузите DrRacket с веб-сайта проекта.

Теперь пришло время поменяться ролями. Запустите DrRacket. Перед вами откроется окно, как показано на рис. 2[1]. 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.

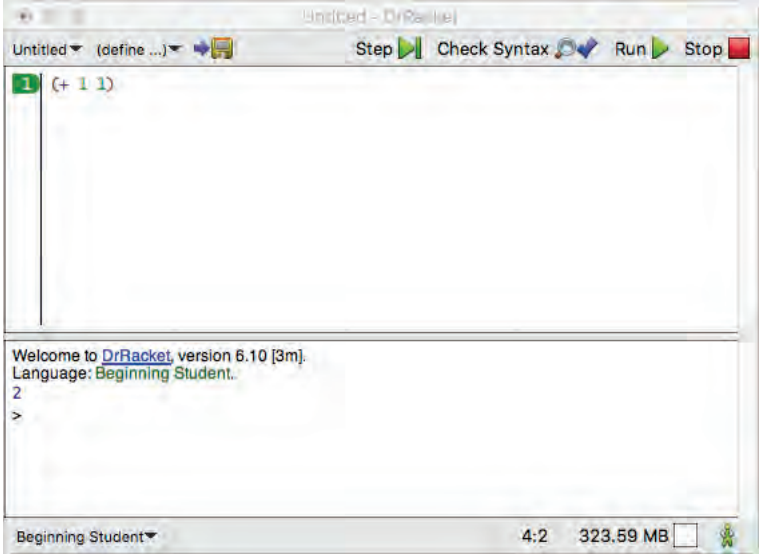


Рис. 2. Общий вид окна DrRacket

[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
  ```

  Введите 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
> (sqr 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
```

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

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 операндов 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
```

```
> (+ 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 операнд 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 операндами, 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 операндами, 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...

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

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.

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 операндов.

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.

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

называют *предопределенными* или *встроенными*) 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.

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

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, программисты на BSL, как и программисты на других языках программирования, создают библиотеки, которые могут оказаться полезными для других. Использование таких библиотек напоминает расширение словаря новыми словами. Мы называем такие библиотеки *учебными пакетами*, потому что они помогают в обучении.

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).*

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.

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 так же, 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:

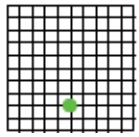
```
> (image-width (square 10 "solid" "red"))
10
> (image-width
    (overlay (rectangle 20 20 "solid" "blue")
             (circle 5 "solid" "red")))
20
```

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:

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

В результате получается:



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.

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.

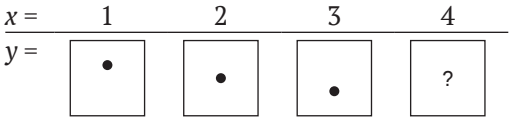
Это и хорошо, и плохо. Хорошо, потому что программирование и вычисления являются естественным обобщением калькулятора. Плохо, потому что цель программирования – обрабатывать большое количество данных и получать много разных результатов, выполняя более или менее одинаковые вычисления. (Программы также 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:



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 числовом 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 y 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 y 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.

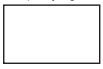
- Во-вторых,

```
(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)
```



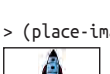
то DrRacket нарисует пустой прямоугольник, который называется сценой. В сцену можно добавлять изображения с помощью `place-image`:

```
> (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)))
```

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

Первую цель можно достичь с помощью функции в листинге 1. Да, это определение функции. Вместо `y` 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)
```

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

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

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-

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 организована функция `animate`.

Но откуда берутся изображения в окне? Если в двух словах, то: `animate` применяет функцию в своем операнде к числам 0, 1, 2 и т. д. и отображает полученные изображения. Вот более 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:

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

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:

Откройте новую вкладку в DrRacket и начните с чистого листа.

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

В общем случае условное выражение имеет следующий вид:

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

*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.*

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-rocket` принимает высоту, на которой она должна изобразить ракету в сцене,

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

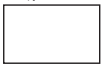
В листинге 2 приводится уточненное определение функции. В нем определяется функция с именем `picture-of-rocket.v2`, чтобы мы могли различать две версии. Применение разных имен также позволяет использовать обе функции в области взаимодействий и сравнивать результаты.

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))]))
```

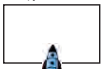
Вот как работает оригинальная версия:

```
> (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 смирится 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 однострочные 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 координаты *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

Будьте внимательны! В этом разделе используются некоторые знания из физики. Если вас пугает физика, пропустите этот раздел при первом чтении; программирование не требует знаний физики.

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)]))
```

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 разгова-

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

; функции
(define (picture-of-rocket.v6 t)
  (cond
    [(<= (distance t) ROCKET-CENTER-TO-TOP)
     (place-image ROCKET X (distance t) MTSCN)]
    [(> (distance t) ROCKET-CENTER-TO-TOP)
      (place-image ROCKET X ROCKET-CENTER-TO-TOP MTSCN)]))

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

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 + и / – 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!

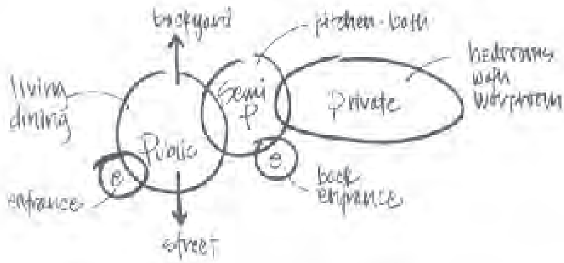
Notes from meeting w/ Client

- DESIGN A SINGLE FAMILY RESIDENCE IN HISTORIC PRESERVATION NEIGHBORHOOD
- PROGRAM: 3 BED 2 BATH · LIVING · DINING · KIT · WORKROOM · PLAYROOM · UTIL · DECK/PORCH ~2000 SF @ $80 PER SF

SPECIAL WISHES:

- MORNING LIGHT IN KITCHEN AND BEDROOMS
- STAIR AS FOCAL ELEMENT IN ENTRY AREA
- STRONG CONNECTION OF PUBLIC SPACES TO OUTSIDE
- PORCH WITH SUN ALL DAY AND PORCH SWING

- BUBBLE DIAGRAM OF FUNCTIONAL RELATIONSHIPS



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.

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

- 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».

```
(+ 1 2)
==
3
```

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

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

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

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 *предопределенными*, *операциями*. Полный перечень функций, доступных в языке BSL, вы найдете в документации, поставляемой с DrRacket.

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 `pi` 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 `sqrt` 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. Прочитайте 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

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

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.

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.

Поэкспериментируйте 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/xy` 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`	`(overlay (square 4 "solid" "orange")`
	`(circle 6 "solid" "yellow"))`
	`==`
	 

`(+ 1 2) == 3`	`(underlay (circle 6 "solid" "yellow")`
	`(square 4 "solid" "orange"))`
	`==`
	 

`(+ 2 2) == 4`	`(place-image (circle 6 "solid" "yellow")`
	`10 10`
	`(empty-scene 20 20))`
	`==`
	 

...	...

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 определений:

Скопируйте и вставьте изображение в DrRacket.

```
(define cat )
```

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 – помощь 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:

1. Первое подвыражение вычисляется всегда. Его результат должен быть логическим значением.
2. Если первое подвыражение возвращает `#true`, то вычисляется второе подвыражение; в противном случае – третье. Результат второго этапа вычислений становится результатом всего выражения `if`.

Щелкнув правой кнопкой мыши на результате, вы сможете выбрать другую форму его представления.

Введите 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: `<`, `<=`, `>`, `>=`.

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 светофо-

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 <image>)
```

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) ...)
```

*Введите 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:* прочитайте (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, поспите и переходите к следующей главе. ■